AN OSGi BASED SOFTWARE INFRASTRUCTURE FOR SMART HOMES OF THE FUTURE

By

SREE CHARAN KUCHIBHOTLA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002

TO MY WONDERFUL PARENTS

ACKNOWLEDGMENTS

I express my gratitude to Dr. Abdelsalam (Sumi) Helal for his encouragement and support, without which this work would not have been possible. I would also like to thank Dr. Michael Frank and Dr. William Mann for agreeing to serve on my committee.

I thank my friend Vijay M. Vokkaarne for his contribution to Magic Wand application and for all his support.

Finally, I would like to thank my friends Sasidhar and Suchindra for their suggestions and for patiently listening to my endless lectures on this thesis design.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Abstract of Thesis Presented to the Graduate School
Of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

AN OSGi BASED INFRASTRUCTURE FOR SMART HOMES OF THE FUTURE

By

Sree Charan Kuchibhotla

December 2002

Chair: Dr. Abdelsalam (Sumi) Helal
Major Department: Computer and Information Sciences and Engineering.

Pervasive computing enables people to accomplish personal and professional activities using intelligent and portable devices. Recent advances in this field have led to the notion of "smart spaces," which are ordinary environments equipped with intelligent devices that can perceive and react to people. "Smart homes" are smart spaces which bring pervasive technology to our homes and their scope is not only to enhance the quality of our lives, but also to enable the elderly and the disabled to lead an independent life.

In 1999, OSGi (Open Services Gateway Initiative) was established as an independent non-profit organization working to define and promote delivery of managed services to networks in home and other environments. OSGi specification requires services to be packaged into software "bundles." It also provides a framework on which these software bundles from different vendors can execute and interact.

This thesis aims at providing a software infrastructure based on OSGi for developing various services in a smart home environment. A generic event broker to facilitate software events to be passed among different services is also developed. To demonstrate the utility of the infrastructure, two applications, namely *Smart App* and *Magic Wand*, have been developed that would allow remotely controlling devices in a smart home. *Magic Wand* is developed for Motorola iDEN cell phones while *Smart App* is developed for desktops.

Smart App provides features to control the appliances in a smart home and to display the events handled by the event broker. It also provides an interface which the user may use to subscribe to events or create custom events. *Magic Wand* is targeted for the elderly and it provides features to turn the appliances on and off in a smart home. It also gives notifications when new mail is delivered or when someone rings the doorbell. Both these applications use the infrastructure and operate on the same set of devices. The interaction among these applications is only through software events.

A cell phone providing many such services can indeed be a magic wand assisting the elderly into successful aging.

CHAPTER 1
INTRODUCTION

Computers have evolved from mere number-crunching machines to indispensable smart machines assisting people in almost all walks of life. The emergence of personal computing in the 1980s changed the general perception towards computers and more recently the advent of the Internet and the emergence of portable and handheld devices lead to the notion of ubiquitous and pervasive computing.

Pervasive computing is all about making technology an integral part of our lives. It aims at enabling people to accomplish professional and personal activities with ease. The concept of pervasive computing lead to an exponential growth in the number of mobile portable devices like PDAs, laptops, smart phones etc. Pervasive computing is realized by creating *smart spaces,* which are environments embedded with devices to perceive and respond to users' actions and events.

Smart spaces not only improve the quality of professional life, but can also help us in making our personal lives more comfortable. A s*mart home*, which is defined as a home with smart spaces, not only makes our life more pleasurable but also has immense potential in promoting independent living and assisting us into successful aging.

1.1 Motivation

In the recent years, many applications are being developed for the *smart home* environment. Each of these applications targets a particular aspect like controlling smart devices, providing security etc. The lack of an organized approach in developing

applications for smart home almost made it impossible to compile all these into a single comprehensive package for any *smart home*.

OSGi was started in 1999, to specify a standard that would allow services on the Internet to be delivered to the local networks. This was the first step towards an organized approach for application development. The standard specifies a framework on which applications packaged in the form of software services, can execute. The framework allows for complex services to be built on top of existing services.

In this thesis, a software infrastructure has been developed on top of OSGi. The infrastructure contains the following three components:

1. Generic event broker
2. X10 Controller
3. Device services

The generic event broker component defines APIs that allows different services on OSGi to communicate by means of software events. The remaining two components are targeted towards the class of applications that control devices in a smart home using the X10 technology.

1.2 Organization of the Thesis

Since the software infrastructure developed as a part of this thesis, primarily targets applications to control devices using X10, an introduction to X10 technology is given in chapter 2. In chapter3, a detailed description of OSGi, which is the framework used by the software infrastructure, is given. Chapter 4 and 5 discuss the design and implementation details of the infrastructure.

To demonstrate the use of this infrastructure, two applications are developed. The details of these are given in chapter 6. Chapter 7 discusses the scope for future work and concludes by summarizing the contributions made by this thesis.

CHAPTER 2
X10-PROTOCOL

X10 is a communication language that allows devices to talk with each other via the existing 110V electrical wiring. X10 is by far the most successful and widely used technology for communication over power lines. Use of electrical wiring is one of the strong aspects of the protocol as it saves the cost of rewiring. The success of X10 can be attributed to the protocol's simplicity and the low cost of X10 devices.

This chapter discusses the history of X10 and provides a detailed explanation of its protocol. The advantages and disadvantages are listed next, followed by a note on the tradeoffs and simple solutions to overcome some of the disadvantages. Finally, a brief overview of CEBUS and Lonworks technologies, which are similar to X10, is presented.

2.1 History of X10

X10 protocol is developed by Pico Electronics Ltd. of Scotland, which was founded in the early 1970's. The company specializes in developing advanced integrated circuits mostly for the calculators. Every time Pico began a new project, it was given an "experiment" number. Experiments from 1 to 9 were mainly in developing integrated circuits for calculators. In the mid seventies, a company named BSR (British Sound Reproduction) signed a project with Pico Electronics to provide an electronic, wireless method of remote control for its equipment. This project was called experiment 10, or simply **X10**.

It was soon realized that X10 has far more potential in many applications in addition to those for which it was originally conceived. Radio Shack, which later became the largest distributor of X10 components, introduced the technology in the American market in 1979 and with in a few years X10 products were widely used in homes across the United States. Lamps and appliances were the most common devices that were controlled by X10 modules. Over the years, more sophisticated X10 modules were developed for a wide range of devices like cameras, motion detectors, thermostats, doorbells etc., but the protocol essentially remained the same.

2.2 X10 Architecture

X10 architecture consists of two entities

1. X10 modules: These are attached to the devices to be controlled. X10 modules plug into the electrical sockets and provide a new electrical socket in which the device to be controlled is to be plugged.

2. X10 controller: These are the cardinal components of X10 architecture. An X10 controller sends control signals to X10 modules, over the power line and receives responses. There can be more than one controller to control a set of X10 modules.

Depending on the functionality, X10 components are also broadly classified into two categories: X10 transmitters and X10 receivers.

1. X10 transmitters are devices that are capable of sending X10 signals over the power line.

2. X10 receivers are devices that are capable of receiving X10 signals over the power line

3. X10 trans-receivers are devices that are capable of both sending and receiving X10 signals over the power line.

In the remainder of the chapter, unless other wise stated, modules, controllers, receivers, transmitters refer to the corresponding X10 components.

It can be noted that modules and controllers fall under one of these three categories. Generally, all modules are receivers and all controllers are transmitters.

All X10 components i.e., modules and controllers, have an address called the X10 address. A typical X10 address consists of two parts, house code and the unit code. House code can take any value from “A” to “P” and unit code can take any value from “1” to “16”. The address is specified as a combination of house and unit codes. For example, “A01,” “B16,” “C02,” etc. are all valid X10 addresses. It can be observed that the total number of addresses possible is 256, ranging from “A01” to “P16”, which means that 256 is the maximum number of devices that can be controlled using the X10 protocol.

Depending on the type of type of user interface, controllers can be broadly classified into three types

1. Mini controllers: These are generally wall-mounted units that are plugged to the power line. These controllers have provision for taking user commands and some of the controllers have LCD displays to show the status of devices.

2. Wireless controllers: These are plugged to the power line and also have an RF interface. They require a remote control through which the user can enter commands, which are transmitted to the controller using RF. The controller parses these commands and performs the appropriate action by communicating with the modules, over the power line.

3. Computer controllers: These controllers are of great practical significance and are used in all computer-controlled X10 projects. They have power line interface and RS-232 interface, which enables them to be plugged to the computer’s serial port. Some of the controllers have RJ11 interface, in which case a RJ11 to RS-232 converter is required to plug them to the serial port. The controllers enable users to use sophisticated and customized applications to enter commands, which are parsed by the controller.

2.3 X10 Protocol

X10 communication comprises of two phases, the selection phase and the command phase. All the communication is done in binary format. In the selection phase, the controller puts the address of the corresponding X10 device to control, on the power line. In the command phase, the controller puts the specific X10 command. Since all X10 communication is broadcasted on the power line, the selection phase provides a way for only the corresponding module to respond to the commands sent during the command phase. Table 2.1 shows the list of X10 commands commonly supported

Table 2.1 List of X10 commands

X10 Commands
All units off
All lights off
All lights on
On
Off
Dim
Bright
Status
Hail

To enable communication in binary format, all the house codes, unit codes and X10 commands are encoded in binary format. Tables 2.2 and 2.3 show the encoding scheme used by X10.

The packet formats in the selection phase and the command phase are shown in figure 2.1. It has to be noted that the command phase may contain more than one command packet. After the selection phase, all the packets sent until the next selection phase fall under the command phase.

Table 2.2 Encoding used for X10 house and unit codes

House code	Encoding	Unit code	Encoding
A	0110	1	01100
B	1110	2	11100
C	0010	3	00100
D	1010	4	10100
E	0001	5	00010
F	1001	6	10010
G	0101	7	01010
H	1101	8	11010
I	0111	9	01110
J	1111	10	11110
K	0011	11	00110
L	1011	12	10110
M	0000	13	00000
N	1000	14	10000
O	0100	15	01000
P	1100	16	11000

Start code	House code	Number code

(a) Packet format in selection

Start code	House code	X10commnand code

(b) Packet format in command phase

Figure 2.1 X10 command packet formats

For example to turn on a lamp with address A03, the controller first sends a packet having “A03” and then sends the packet having the command “ON”. To turn off the same lamp, the command “OFF” is sent. However, to turn off a different lamp with address “A04”, the controller sends the packet “A04” followed by the packet “OFF”.

Every packet starts with a start code, which is a special sequence of electric pulses. Start code helps both the sender and receiver to synchronize. For reliability, every packet is transmitted twice. The details of transmission on power line are explained the next section.

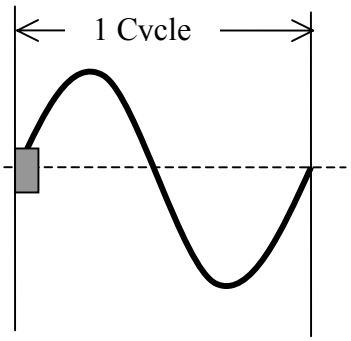
One of the fundamental drawbacks of this protocol is that, it provides no means to indicate whether a particular command is successful or not. For example, when a "light on" command is sent, X10 protocol provides no means of knowing if the light was actually on[1].

X10 provides a very basic form of device discovery. When a new module is plugged into the power line, it sends a packet containing its address. The protocol provides no means to convey the information about the device that is connected to the module. This is yet another drawback of the protocol, since it greatly hampers the development of any service discovery protocol on X10. A simple work around to this situation is presented in section 2.6.1.

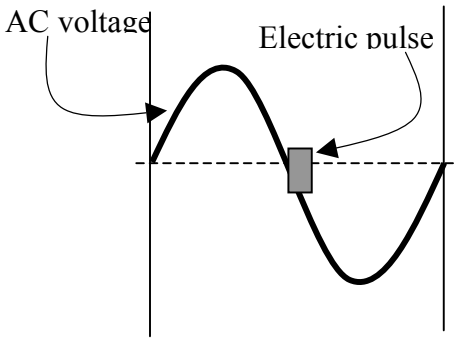
2.4 X10 Power Line Transmission Theory

X10 Binary data is encoded in the form of electric pulses and are transmitted over the power line. These transmissions are synchronized to the zero crossing point of the AC power line. Since the AC voltage varies in a sinusoidal form, there are two zero crossings in each cycle.

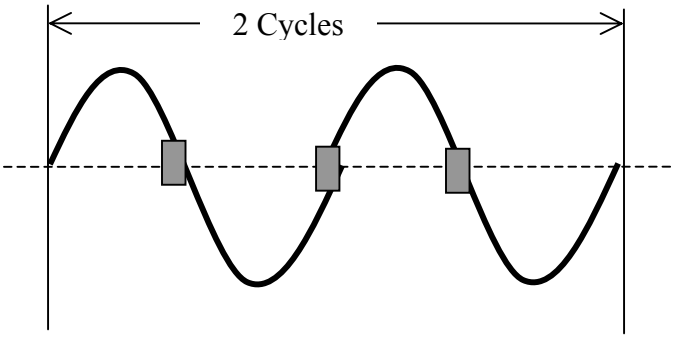
[1] The status command provided by X10 only indicates the internal state of the X10 module. It does not take device failures into account.



(a) Binary ‘1’ is represented by the presence a pulse followed by an absence of the pulse



(b) Binary ‘0’ is represented by the presence a pulse followed by an absence of the pulse



(c) Start code is represented by three pulses as shown in figure.

Figure 2.2 Electrical encoding in X10 [KIN02]

Two zero crossings are required to encode the binary numbers “1” and “0”. A binary “1” is represented by a 1 ms burst of followed by an absence of pulse in the next zero crossing. Similarly “0” is represented by the absence of a pulse followed by the presence of a pulse in the next zero crossing. Start code is represented by 1 ms bursts in three consecutive zero crossings, followed by the absence of pulse in the next zero crossing. Figure 2.2 shows the encoding used for binary “0”, “1” and the start code.

2.5 Advantages and Disadvantages

The advantages and disadvantages of X10 can be summarized as follows

2.5.1 Advantages

- X10 technology is simple and inexpensive
- Require no special wiring
- Offers wide variety of controllers for simple user interface
- Has provision for computer control
- Easily extensible

2.5.2 Disadvantages

- The maximum number of devices that can be controlled is 256
- X10 signals can be degraded, damped or stopped by power conditioning equipment line power supplies, power strips
- X10 signals might be lost in an electrically noisy environment
- Has no provision to check if the command was successful
- X10 does not provide any means by which information about the type of device can be conveyed. For example, using X10, there is no means by which one could find out, if the appliance module is plugged to a radio, television or any other appliance.

2.6 A Scheme for Device Discovery and for Improving Reliability in X10

2.6.1 Device Discovery

As mentioned earlier, X10 provides no provision for device discovery. This eliminates the possibility of developing any device discovery and service discovery applications in a computer controlled X10 environment. However Yi-Min Wang, Wilf Russell and Anish Arora who worked on the Microsoft Aladdin project [WAN00] suggested a simple workaround to this situation.

If each power line socket is assigned a unique X10 address and a scheme is prepared before hand, mapping each socket to a particular device, then this information can be stored in a database and can be used by device discovery applications. It has to be noted that all devices need not be present at the time the scheme is prepared. Whenever a new X10 module is plugged, it puts its address on the power line. This can be used as the key to search the database and retrieve information about the devices.

This is not at all an impractical assumption considering the fact that in any house, most of the appliances are pretty much fixed to the same location. The scheme has been successfully implemented and tested in Microsoft's Aladdin project.

2.6.2 Improving Reliability

X10 modules are not reliable, in the sense that they do not provide any mechanism to check if a particular command is successful. For example if an appliance is turned on using an appliance module, X10 provides no means to find out if the appliance is actually turned on. Though a "status" command provided by X10, it only reflects the internal status of the X10 module and not that of the device. If an appliance is faulty and does not turn on in response to an "on" command, the X10 module attached to it would still respond to a "status" command, with status "on".

This aspect can be easily improved by adding a simple AC current sensor to the X10 module. The sensor detects the current through the device and accordingly prompts the X10 module to respond to the "status" command. This modification is shown in figure 2.3 and was implemented in Microsoft Aladdin project.

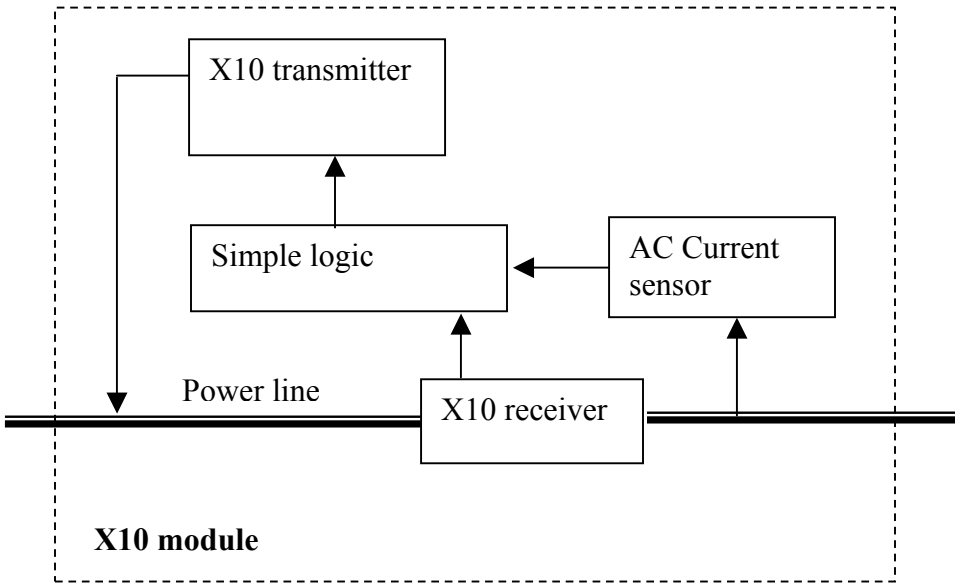


Figure 2.3: A reliable X10 module

2.7 Conclusion

X10 offers a simple and low cost method of controlling devices. The advantages of X10 come with tradeoffs in reliability. To address these issues, two new technologies namely Consumer Electronic Bus (CEBus) [EVA01][CEB00], and Lonworks were developed recently.

CEBus is an open architecture which specifies and define protocols to enable devices to communicate through power line wires, low voltage twisted pairs, coaxial cables, infrared, RF, and fiber optics. All CEBus devices communicate using a Common Application Language (CAL), which is a universal language for home network products. The standard also allows for device discovery as all the CEBus devices follow the Home Plug and Play (HPnP) standard.

Lonworks is networking platform created by Echelon Inc. Lonworks is a networked control system that is significantly more powerful, flexible, and scaleable than a non-networked control system. The devices communicate using a special protocol called LonTalk. Lonworks protocol cannot use the existing power line.

Both Lonworks and CEBus based technologies are considerably expensive and more difficult to install when compared to X10. Though CEBus is an open standard, currently there are no CEBus compliant devices available in the retail stores. As a result of these issues, X10 is still the most popular technology used for controlling devices.

CHAPTER 3
OPEN SERVICES GATEWAY INITIATIVE - OSGi

The open service gateway initiative (OSGi) is an open reference architecture defined primarily for the delivery of services on the Internet to local networks and devices. The host managing the services in the local network is called as the *service gateway*. The OSGi platform specification provides an open, common architecture for service providers, developers and equipment vendors to develop, deploy and manage services in an organized and coordinated fashion. The primary targets of OSGi specification are PCs, set top boxes, cable modems, consumer electronics and more.

The need for this standardization can be attributed to the following reasons [CHE02]:

- Platform independence: Different types of devices have their own native platforms and different configurations. It would be impractical for the service providers to port the services to all platforms. OSGi provides a virtual platform on which all services are developed.
- Vendor independence: Standardized APIs allow for services written by different vendors to cooperate.
- Future-proof: The lifetime of services that are coupled to a particular underlying technology is just as much as that of the underlying technology. An open standard provides an abstract layer and decouples services from the underlying technology thereby increasing the lifetime. For example, a service written to control devices over power line need not even be recompiled if the underlying technology is changed from X10 to CEBus.

The platform independence requirement of OSGi makes Java™ [SCH01] [HOR00] as the default platform and the specification is not valid for any other platform. Most of the terms used to describe the OSGi architecture are taken from Java ™ parlance.

The reader of this document is assumed to have knowledge in Java ™ programming language.

3.1 OSGi Architecture

OSGi has a layered architecture as shown in figure 3.1. The architecture consists of three components:

1. Framework
2. Bundles
3. Services

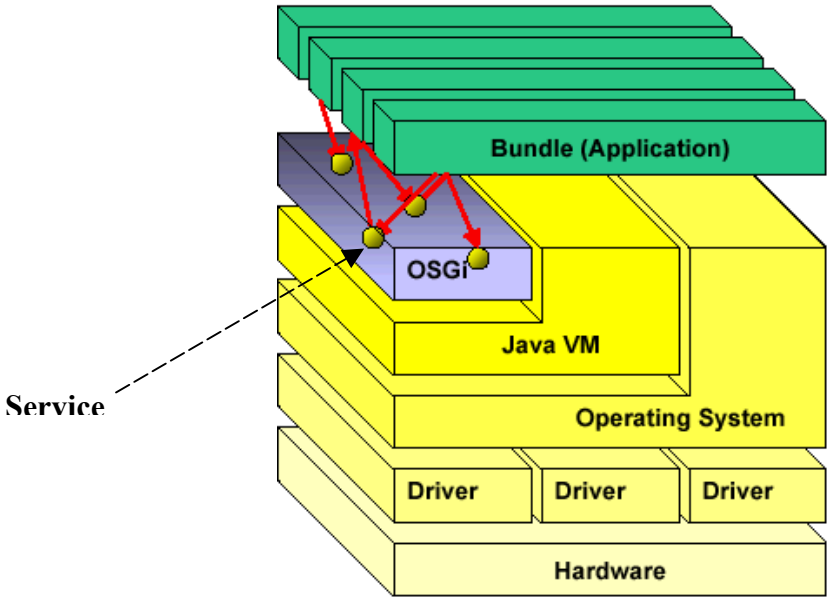


Figure 3.1: OSGi Architecture [OSG01]

Framework is the heart of this specification as it provides a virtual environment on which services from different vendors can execute. The framework provides clear interfaces for registering new services and a powerful service lookup facility. The framework also provides some basic eventing capabilities, which are described in section 3.4.

Services are java objects that implement a standard interface and execute on the framework. A bundle is a collection of one or more services and is responsible for registering services with the framework. Services are exchanged among bundles through the framework in a secure and controlled manner. Thus, bundles may provide services to other bundles as well as use services from other bundles.

Besides providing specification for framework, services and bundles, OSGi also contains *Device access specification*, which is a mechanism to couple *device services* to *driver services*. The terms used in this section are explained in more detail in the subsequent sections.

The remainder of this chapter discusses about the services, bundles and framework in detail followed by the device access specification of OSGi.

3.2 Service

A service can be defined as a Java ™ object implementing a concisely defined interface [OSG01]. In OSGi environment, everything is a service; for example, a web server is a service, so is a library routine that helps in performing a complex calculation or a program to control a particular device. Applications that execute in the OSGi environment are a collection of services; all of which need not necessarily be from the same vendor. As it will be explained later, OSGi provides a mechanism for services to be shared between different applications[1].

Every service consists of a *service interface* and a *service implementation*. Service interface is a Java ™ interface, which specifies the functions provided by that service. The purpose of the service interface is to specify the semantics and the behavior

[1] In OSGi environment, the term ‘bundle’ is used instead of the ‘application’

of a service. The interface also hides the implementation details of the service. Service implementation, which is also called as *service object*, is a Java ™ object of a class that implements the service interface. It has to be noted that in a typical scenario, there can be many service implementations for a single service interface. Such a scenario exists when different vendors provide the same service or if there are many flavors of the same service. For example, a service to control a particular device, say a camera, has a standardized service interface but depending on the model and make of the camera, the service implementations might vary.

3.2.1 Service Properties

According to OSGi specification every service is associated with a *Properties* object[2], which is a collection of (property name, value) pairs. The developer of the service normally defines its properties. Properties provide an information database on top of which, as we shall see in later sections, powerful service lookup mechanisms can be implemented.

In addition to the properties defined by the service developer, OSGi specification requires every service to have three mandatory properties, which are o*bjectClass*, *service.description* and *service.id*. OSGi framework assigns the value for service.id property automatically.

As an example, see Table 3.1 to see a list of properties that can be defined for a dictionary service.

---

[2] 'Properties' is a standard class defined in java.util package. It represents a collection of (property name, value) pairs. Please refer to java API documentation for more details.

Table 3.1 Properties of an example dictionary service

Property name	Value
objectClass	edu.ufl.icta.Dictionary
service.description	"Dictionary service by UFL"
service.id	XYZ
Author	"ICTA"
Version	1.1
SourceLanguage	"English"
TargetLanguage	"English"

3.2.1 Service Registration, Un-registration

When a bundle is started, it registers services with the framework service registry. A registered service is available to other bundles under the control of framework.

The framework provides a powerful lookup mechanism by which bundles can query for services and get their references. A registered service can be unregistered at any time by the bundle that registered it. Whenever a bundle is stopped, the framework automatically un-registers all the services registered by that bundle.

Details about service registration, un-registration and the interfaces provided by the framework are explained in more detail in section 3.4.

3.3 Bundles

A bundle is a collection of services that are packaged in a well-defined fashion. It acts as a means by which services are hooked on to the OSGi framework. This section describes the structure of a bundle and its life cycle. The interaction of bundle with the framework is described in section 3.4.

3.3.1 Bundle Format

Bundle is a JAR[3] file containing class files corresponding to service interfaces and implementations. Bundles also contain all the resource files like image files, HTML files, data files, configuration files etc.

Every bundle should have a special class file that implements the standard interface *org.osgi.BundleActivator*. The interface specifies two functions *start()* and *stop()* which the framework calls when a bundle is installed. Normally, the code to register services and get references to other services is written in the start() function. Code to un-register the service, is written in the stop() function.

Details about registering, un-registering and getting reference to other services are explained in section 3.4.2.

A file called *Manifest* specifies the organization of all the files within a bundle. The bundle developer has to create a manifest file before packaging the bundle. The format of manifest file is specified in OSGi Service platform release 2 [OSG01].

Figure 3.2 shows the anatomy of a bundle

[3] Java Archive, JAR format is a standard format defined by Sun Microsystems

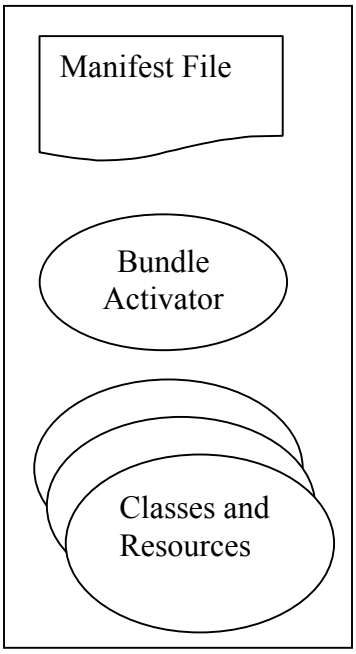


Figure 3.2: Anatomy of a bundle

3.3.2 Bundle Life Cycle

Unlike other archive files, bundle has different states. A bundle can be in one of the following states: Installed, uninstalled, starting, stopping, resolved and active.

Services contained within a bundle can be available only if the bundle is in active state. Figure 3.3 shows the state transition diagram of a bundle. In figure 3.3, the thick arrow lines represent the transitions triggered by the framework and the dotted arrows represent automatic transitions.

The bundles life cycle is explained in more detail in section 3.4.3.

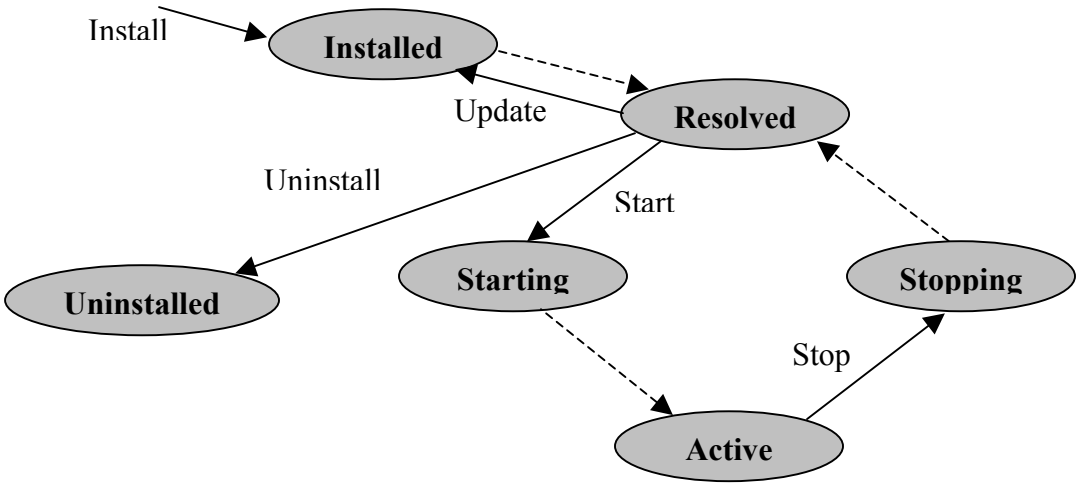


Figure 3.3: Bundle state transition diagram

3.4 OSGi Framework

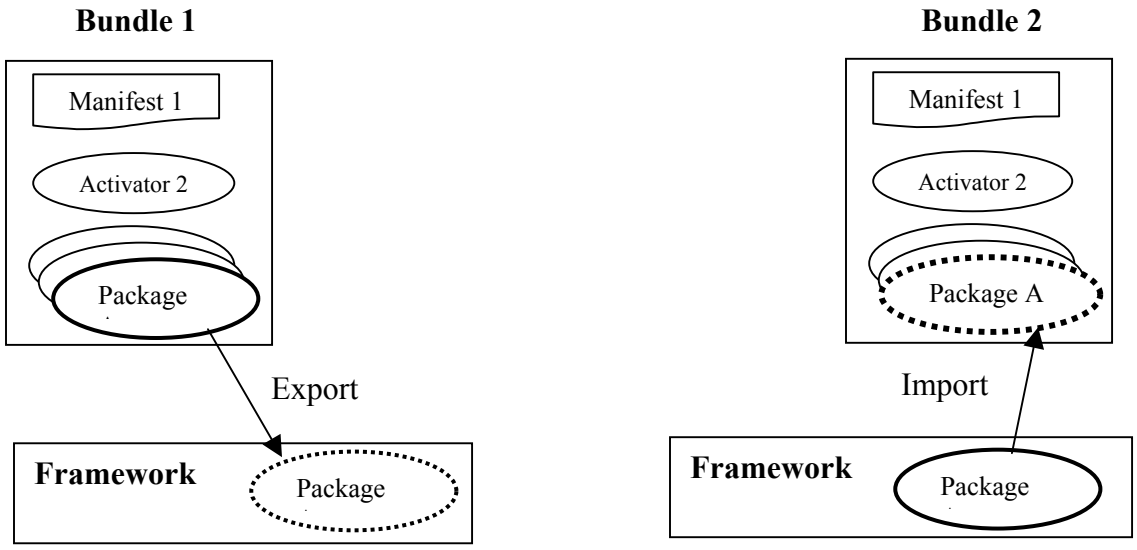
Framework is the heart of OSGi specification. It provides a common ground on which bundles execute. Framework provides the following functionalities:

- Resolves interdependencies among bundles and makes it possible to share classes and resources among bundles.
- Maintains a registry of services and provides APIs to register, un-register and lookup services.
- Manages life-cycle of a bundle
- Provides a basic eventing mechanism to raise events whenever the state of a bundle, service or the framework changes
- Implements a device access manager that helps in attaching device services to driver services. This is a part of device access specification which a part of OSGi. Device access specification is explained in detail in section 3.5.

3.4.1 Interdependencies among Bundles: Exporting and Importing Packages

As we have seen earlier, a bundle contains all the class files corresponding to the services and the resource files required by those services. However, bundles are not entirely self-sufficient. A service might use some packages that are defined in some other bundle. In such a case, the bundle needs to inform to framework, the list of all packages it wishes to import. Similarly a bundle might choose to export some of its packages so that other bundles might import them.

Information about the packages that a bundle wishes to export or import is specified in the bundle's Manifest file. When bundles export packages, they are effectively exporting them to the framework. Similarly, when bundles import packages, they are effectively importing them from framework. The scenario is illustrated in figure 3.4.



(a) Bundle 1 exports Package A to framework

(b) Bundle 2 imports Package A from framework

Figure 3.4: Exporting and importing packages using OSGi framework

3.4.2 APIs to Register, Un-register and Lookup Services

The design philosophy behind OSGi specification is to allow different services to be shared between different applications. For example, a dictionary service may be shared by a word processor service and an English language tutor service. To make such a scenario possible, OSGi requires every bundle to register the services it wishes to share with other bundles[4]. A service can be unregistered when a bundle no longer wishes to share the service.

The framework maintains a service registry having information about all services registered. It is a structure that maps the type and properties of the service (specified in "Properties" object), to the service object. A reference to the service object is added to the registry when a service is registered. Whenever a service is unregistered, the corresponding entry is removed.

As it was mentioned in section 3.2, every service has an associated "Properties" object that contains the service properties. The framework uses this feature to implement a powerful look up mechanism. To use a service, its reference needs to be obtained from the registry. The look up mechanism provided by the framework helps in searching for services by specifying filters, and obtaining the reference to the service object.

3.4.3 Bundle Life Cycle

The framework is responsible for managing the bundle's life cycle, which is shown in Figure 3.3. When a bundle is installed, the state of the bundle is INSTALLED.

The framework takes the responsibility of loading all the class files and the resource files packaged in the bundle. In the process of loading, it reads the bundle's

[4] Bundles that do not register any service all called *library bundles*

manifest file to note the packages exported and imported. This process is called *resolution*. Upon successful resolution, the state of the bundle is changed to RESOLVED. In case of any errors in resolution, the state of the bundle still remains INSTALLED.

Framework provides mechanism to start, stop, update and uninstall a bundle. A bundle can be started only if it is in RESOLVED state and can be stopped only if is in ACTIVE state. When prompted to start a bundle, the framework calls the start() method in the bundle's *BundleActivator* class (Framework gets the information about the BundleActivator class from the Manifest file). Upon successful return form start() function, the bundle's state is changed to ACTIVE.

When prompted to stop the bundle, the framework calls the stop() function in the bundle's activator. Upon successful return, the bundle's state is changed to RESOLVED.

When the framework is prompted to update a bundle, the entire process is repeated i.e., the bundle is reinstalled and after successful resolution, it is started (by calling the start() function).

An important point worth mentioning here is that even after a bundle is uninstalled, the packages that it exported still continue to exist on the framework. Similarly even after a bundle is updated, the older version of the packages it exported exist instead of the newer versions. To reflect the changes in the exported packages, the framework has to be restarted.

3.4.4 Basic Eventing

Framework provides basic eventing mechanism to signal the changes in the internal states of the services, bundles and the framework itself.

Bundle events are generated whenever a bundle is installed, started, stopped, updated or uninstalled. Service events are generated whenever a service is registered or un-registered and framework events are generated when the framework is started or if there is an internal error.

OSGi clearly defines the interfaces for the listeners of these events [OSG01].

It can be noted that the eventing facility provided by OSGi is very basic and does not provide any generic infrastructure to exchange events between services. A generic eventing model to accomplish this task has been developed as a part of this thesis.

3.5 DAS: Device Access Specification

OSGi Device Access Specification, DAS, was developed to address the needs of those applications that involve communication and controlling of devices.

Before defining what DAS is, it is essential to clarify some of the common misconceptions about DAS and define what drivers and devices mean in the context of DAS.

DAS is not about writing low-level device drivers. Such device drivers are usually a part of the operating system and serve as the bridge between software and hardware. In DAS terminology, they are called *base drivers*. It should also be clarified that DAS is not about device/service discovery. There are several protocols like UPnP, Jini etc that deal with service/device discovery. DAS is unbiased towards any service discovery protocol.

Drivers, in the context of DAS are called driver services and they refer to software services that execute on OSGi platform and communicate with the underlying base drivers to control the devices. Similarly devices, in the context of DAS refer to

device services which are OSGi services providing an abstract view of the actual physical device.

With that background, one can define DAS as *the specification that supports automatic detection of device services and defines a mechanism to automatically download and install the appropriate driver services*. The process of associating a device service with the appropriate driver services is called *attaching*.

DAS consists of the following main components

1. Device service
2. Driver service
3. Driver locator
4. Device manager

3.5.1 Device Service

A device service is an abstract view of a device. Device service normally represents a hardware device, but that is not a requirement. There can be a single device service to represent a group of hardware devices, or there can be a device service providing an abstract view of an entire network etc.

All device services in OSGi must extend the standard generic device class *org.osgi.service.device.Device* defined by DAS. When a device service is registered with the framework, the device manager is responsible for finding a suitable driver service and attaching it. The mechanism by which the device manager searches for driver services is explained in section 3.5.4.

A device service that is not used by any other bundle is called an *idle device service.*

3.5.2 Driver Service

A driver service is an OSGi service that takes care of communication and controlling of a physical device. Depending on the functionality, driver services are classified into different categories. A detailed taxonomy of driver services is given in OSGi Service platform specification 2 [OSG01].

All driver services in OSGi must implement the *org.osgi.service.device.Driver* interface, which specifies two methods match() and attach(). These methods are called by the device manager, which is explained in more detail in section 3.5.4.

When a device service is registered with the framework, the device manager queries all the driver services by calling their match() method and passing the device service's reference as a parameter. In the match() method, a driver service checks if it can be used for the device service passed as an argument. The method returns a value indicating whether or not the driver service can be used. This process is called *refining*.

The implementation details and the return value of the function match() are not specified by the DAS, as they are highly dependent on the type of device and driver.

The attach() method of a driver service is called by the device manager if it wants to attach a particular device service to that driver service. The reference to the device service is passed as an argument method.

Since the actual method of attaching is highly device specific, the device manager delegates that responsibility to the driver service i.e., by calling its attach() method. Again, DAS does not specify any mechanism for device attachment process.

3.5.3 Driver Locator

Driver locators are special services that are called by the device manager to help locate driver services for a particular device service. The function of a driver locator service, as the name itself implies, is to search for suitable drivers and install the drivers[5] when asked by the device manager. All driver locator service should implement the standard interface *org.osgi.service.device.DriverLocator*. The interface specifies two methods findDrivers() and loadDriver(). The method findDrivers() is called by the device manager if it wants the device locator to search for drivers and loadDriver() is called by the device manager, whenever it wants the locator to install a particular driver service.

The mechanism used by the driver locator services to search for drivers is not specified by DAS.

3.5.4 Device Manager

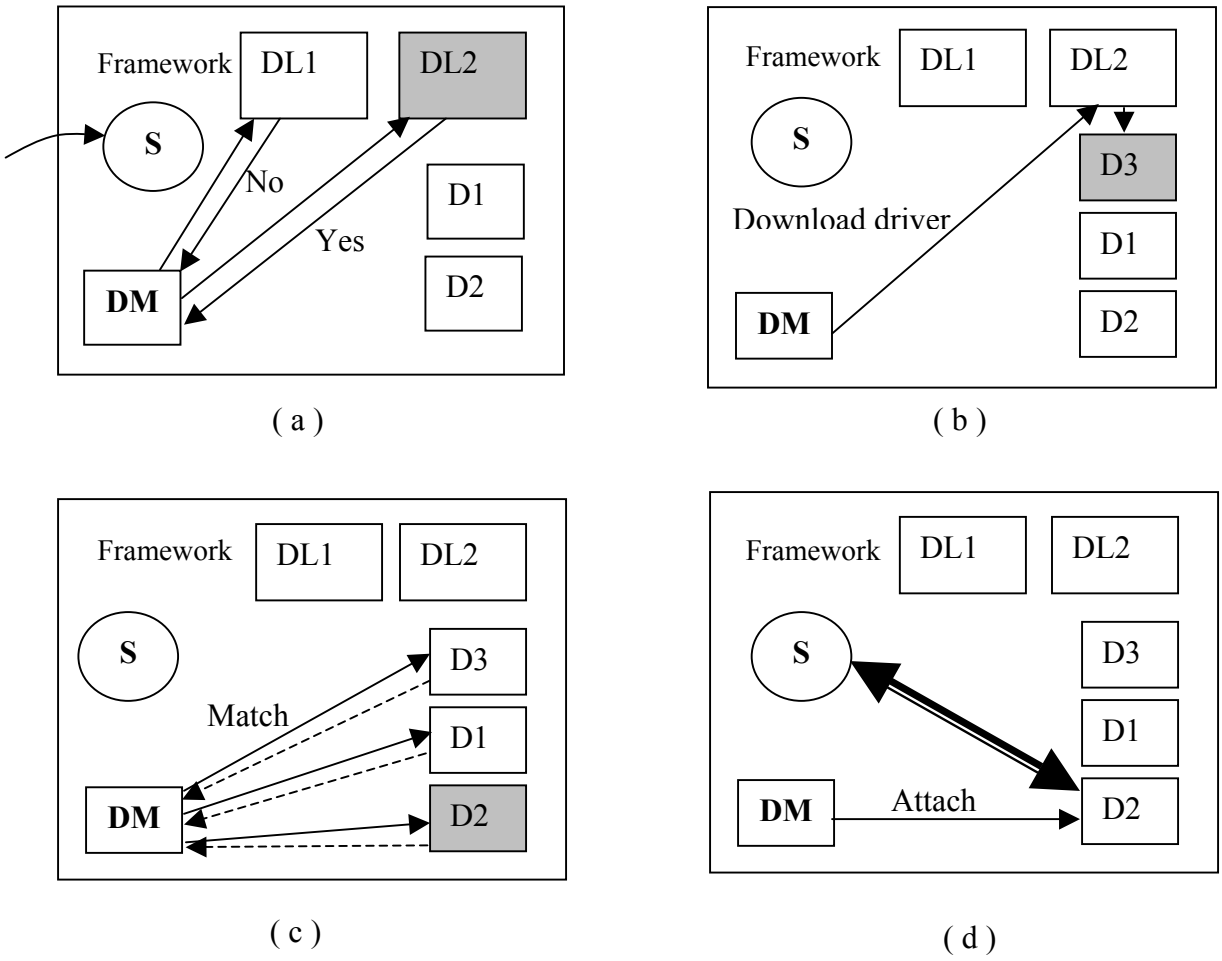
Device manager is the heart of DAS as it coordinates the actions of device services, driver services and the driver locator services. It is responsible for finding suitable driver services for every device service registered with the framework. The device manager accomplishes this task by following these steps

1. It monitors the framework for new device service registrations
2. If a new device service is registered, it asks all available driver locator services for find drivers by calling their findDrivers() method. It then asks the driver locators to install those driver services by calling their loadDriver() method.
3. The device manager then queries all the driver services (which might also include the newly installed drivers by driver locators) by calling their match() method, to see if they can refine the service.

[5] Installing drivers refers to installing the driver bundle and registering the driver service

4. Of the replies, the device manager selects the best driver[6] and instructs that driver to attach to the device service. This is done by calling the attach() method.

This entire process is illustrated in figure 3.5



(a) New device service S is registered and DM queries all the DLs to find drivers for s1. DL1 does not know about any driver, but DL2 knows about a driver D3.
(b) DM asks DL2 to download and install D3. DL2 installs D3
(c) DM queries all the drivers to *match* the device service. D2 gives the best response.

Figure 3.5: Example of Device manager operation [CHE02]

[6] To select the best driver service, the device manager consults a special module called Driver Selector. This module is not specified separately as it is considered a part of device manager.

CHAPTER 4
SOFTWARE INFRASTRUCTURE

This chapter discusses the design of event broker, controller and device services, which are the three main components developed as a part of the software infrastructure. These components are developed as software bundles on OSGi.

Event broker provides a means for services to cooperate by exchanging software events. This component is generic and is not biased towards any particular class of applications. The remaining two components i.e., controller and the device services, are developed for applications that control X10 devices in a smart home.

Controller component takes care of communication with the X10 controller hardware and the device services component is responsible for registering device services[1]. These device services act as software proxy objects to the actual physical devices and provide software APIs for controlling the devices. In this thesis, two device services are developed: one for lamp and the other for any other electrical appliance. The implementation details of these three components are given in the next chapter. The interaction between them is shown in figure 4.1

To demonstrate the capabilities of this infrastructure, two demo applications are developed. The design and implementation details of these applications are presented in chapter 6.

[1] Recall that a device service presents an abstract view of the physical device. See section 3.5.1 for more detailed explanation.

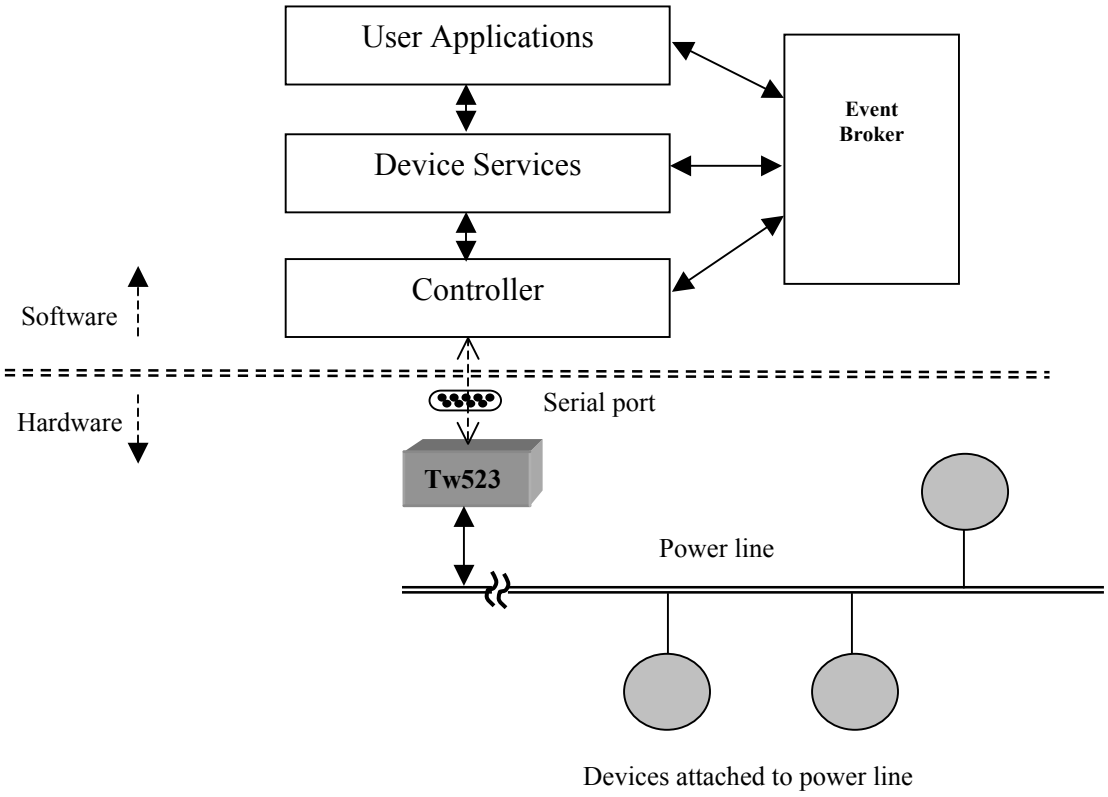


Figure 4.1: Components and their interaction in the software infrastructure

4.1 Event Broker

This component is developed to allow different services on OSGi platform to communicate using events. As it was seen section 3.4.4, OSGi provides basic event facility and has no infrastructure for customized events. All the applications interested in a particular event must register with the event broker. In order to register, the application must provide the event broker with the event name and a *listener function*[2]. This registration process is called *subscribing*.

[2] Listener functions are also called call-back functions

Similarly, when an application is no loner interested in a particular event, it can un-register its listener function from the event broker. This process is called *un-subscribing.*

Whenever a service generates an event, it has to inform the event broker by passing an *event object*. The process of informing an event occurrence is called *signaling*. On receiving an event, the event broker invokes the listener functions of all applications that subscribed to that particular event. The process of invoking is called *dispatching*. The event object is passed to all listener objects while dispatching. Every event object contains the event name, event source, event properties and any other event-specific information.

The event broker component defines a standard interface, which every listener function should implement. Similarly the event broker also standardizes the event object that is passed between event source and the destinations. The structure of listener and the event object are explained in more detail in section 5.1.

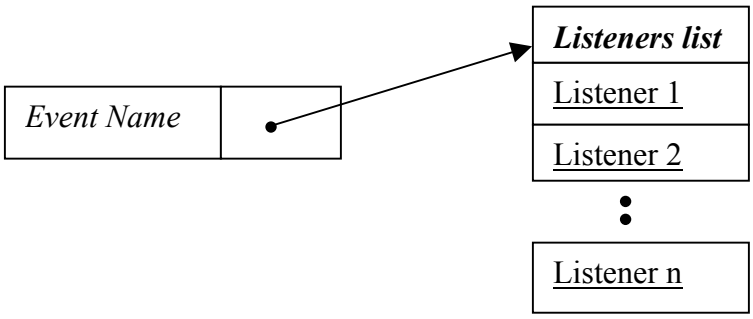


Figure 4.2: Table entry structure

When a service starts, it has to inform the event broker about all events that it might generate during the course of its execution. This process is called *publishing* the events. When all services publish their events, it helps the event broker to provide APIs that would list all the events that are currently available for subscription.

The event broker maintains all the information in a hash table, which is indexed by the event names. Each entry in the table has two fields, the event name and the list of listeners that subscribed to that event. The structure of the table entry is shown in figure 4.2. When a service publishes its event, an entry with the event name and an empty listeners list is created. When an application subscribes to an event, its listener is added to the list of listeners maintained for that event in the table. When an event is signaled, the event broker gets the list of listeners for that event and invokes each one of them.

4.2 Design of Controller

Controller component is designed to interact with the X10 controller device. As mentioned in section 2.2, there are many types of X10 controllers available depending on the type of user interface required. For this software infrastructure, the Tw523 controller is chosen. Tw523 has an interface to connect it to the computer's serial port. The connection details are explained in section 5.2.The main goal of the controller component is to provide a software interface to applications that wish to communicate with Tw523 interface. It is designed to automatically detect whenever a Tw523 X10 controller is plugged to the computer serial port and to create and register Tw523 software service with the OSGi framework. The controller component also detects whenever the Tw523 X10 controller is unplugged and it automatically de-registers the Tw523 software service. The design details are explained in more detail in the next three subsections.

The component comprises of the following three software modules

1. Serial port scanner
2. Serial port listener
3. Tw523 software interface

Since the controller component is packaged as an OSGi bundle, it is started and stopped through the interface provided by the OSGi framework.

4.2.1 Serial Port Scanner

This module is the entry point of the controller component. It begins execution immediately after the controller component is installed and started. The serial port scanner scans for all the available serial ports on the computer. If a serial port is available, it opens the port, sets the serial communication parameters[3] and finally creates a serial port listener module for that serial port. There are as many serial port listener modules as the number of serial ports. When the controller component is stopped, the serial port scanner closes all the serial ports that it opened and also stops the serial port listener modules that it created.

4.2.2 Serial Port Listener

The serial port listener module is responsible to identify if a Tw523 controller is plugged to (or unplugged from) the serial port. It is also responsible for registering and un-registering the Tw523 software service. It has to be noted that the serial port scanner module does the association between serial port listener and the serial port.

When the serial port listener module is created, it listens to all events generated by the serial port to which it is associated.

The serial port raises a “CTS true” event whenever a device is plugged to it. On listening to this event, the serial port listener checks if the plugged device is a Tw523 controller. If it is Tw523, the serial port listener creates a Tw523 service and registers it with the OSGi framework. Similarly, when a device is unplugged from the serial port, it generates a “CTS false” event. On listening to this event, the serial port listener un-registers the Tw523 service, if it exists. Note that the serial port listener need not check if the unplugged device is a Tw523. This is because, if Tw523 service exists, then the device plugged to the serial port has to be Tw523. Apart from registering and un-registering Tw523 software service, the serial port listener module also publishes two events namely *Tw523 plugged* and *Tw523 unplugged* to the event broker.

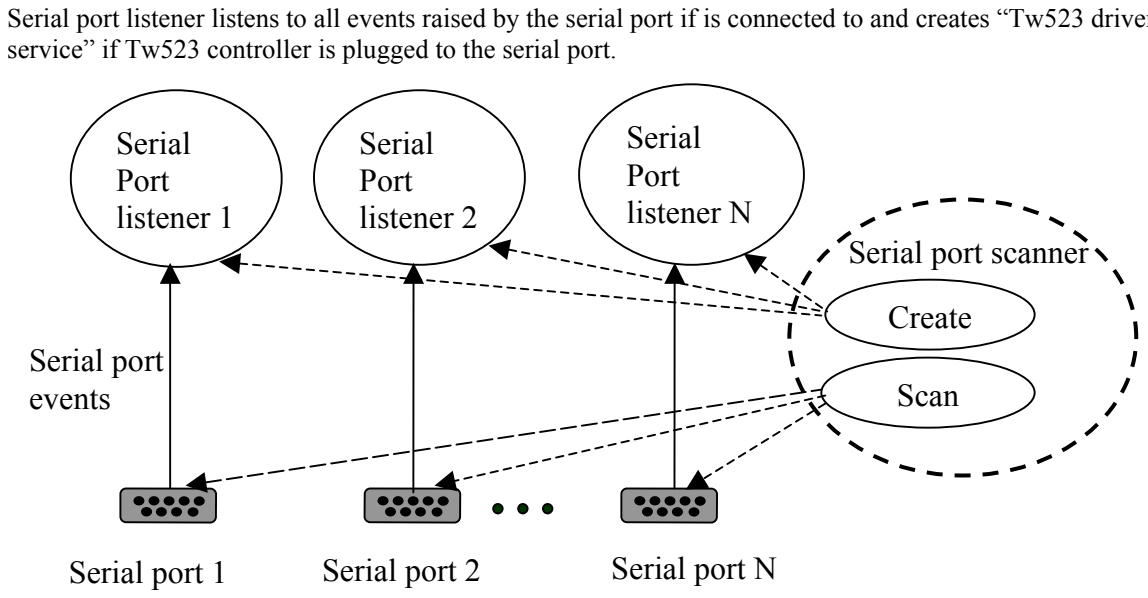


Figure 4.3: Serial port listener and Serial port scanner modules

[3] Serial port parameters required for communication with Tw523 are (Baud rate: 1200, Data bits: 8, Parity: None, Stop bits: 1)

Figure 4.3 shows the interaction between serial port scanner, serial port listener and the serial port.

4.2.3 Tw523 Software Interface

Tw523 software interface module is the heart of the controller component as it provides a software abstraction to the hardware Tw523 device. The module provides a software interface for all applications that wish to communicate to Tw523 device.

The module also senses the power line for any signals generated by X10 devices. If any signal is found, it generates a special event and publishes that event to the eventing broker. This feature is used in the demo applications developed as a part of this thesis.

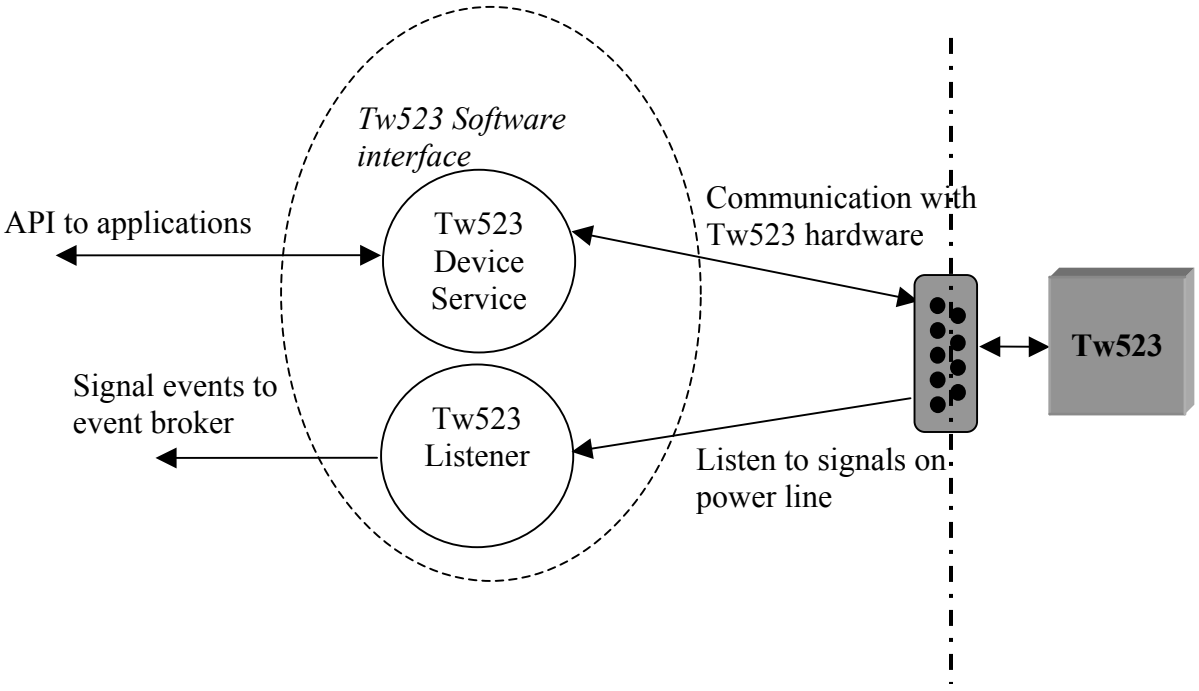


Figure 4.4: Tw523 software interface module

The type of X10 signals and the type of events published by the Tw523 service is explained in more in chapter 6.

Tw523 software module comprises of the following two sub-modules:

- Tw523 device service
- Tw523 listener service

Tw523 device service provides APIs to send commands to the Tw523 hardware device. Tw523 listener service listens to X10 signals on the power line and publishes events with the event broker, whenever a signal is found. Since event broker is a service outside the scope of the controller component, the Tw523 listener service is robustly designed to publish events only if the event broker service is available.

4.3 Device Services

This purpose of device services component is to create device services, which provide software APIs to applications that wish to control devices in a smart home. As it can be seen from figure 4.1, this has a dependency on event broker and controller components.

Both device service and controller components are developed for the class of applications that wish to control X10 devices in a smart home. The reason for separating these two is to for scalability and extensibility. The device services component can work with any controller component as long as the interface provided by the latter remains unchanged.

This component can be divided into two modules

1. Device service initiator
2. Device services

The component is packaged as an OSGi bundle and is started and stopped using the interface provided by the OSGi framework.

4.3.1 Device Service Initiator

The purpose of this module is to read a configuration file containing the list of all X10 devices currently present in the smart home and to create device services corresponding to those devices.

Since X10 technology provides no mechanism for device discovery, providing a configuration file is the only way of conveying the X10 device information and the smart home layout. Each line in the configuration file conveys information about a particular X10 device and has information about the device's properties. The properties are the device's X10 address, location, device type and other device-specific properties. The format of the configuration file is explained in more detail in section 5.3.

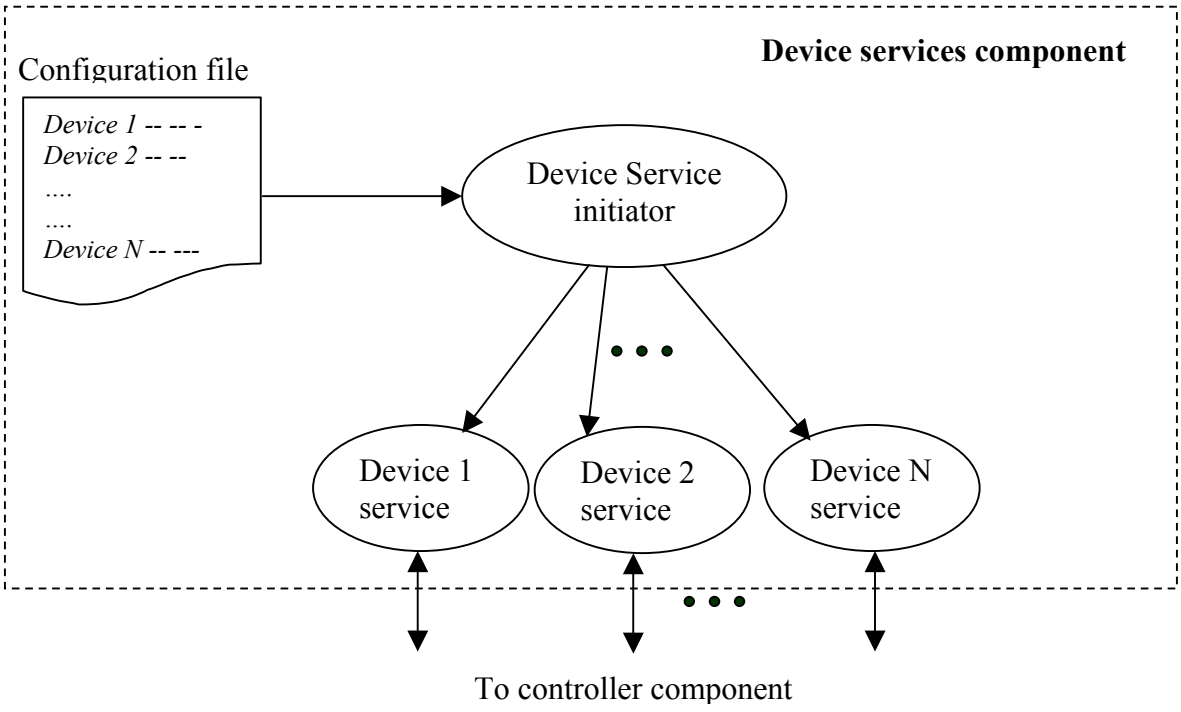


Figure 4.5: Device services component

The device service initiator module reads each line in this configuration file, creates a *properties* object (see section 3.2.1), creates a device service object corresponding to the device type and finally registers the device service with its properties. It has to be noted that each device service represents a single device. So the number of device services created by the device service initiator is equal to the number of entries in the configuration file i.e., the number of controllable X10 devices in the smart home. Figure 4.4 shows the overall design of device services component.

4.3.2 Device Services

As mentioned in section 3.5.1, device services provide an abstract view of a device. In this thesis, device services for two classes of devices have been developed: one for lamp and the other for low power electrical appliances.

The device services publish events to the event broker and are designed to publish events only if the event broker service is available.

4.3.2.1 Lamp Device Service

This service provides APIs to turn the lamp on, off and to get the lamp's status. Apart from these, it also publishes events to the event broker whenever the lamp if turned on or off.

4.3.2.2 Appliance Device Service

This service is similar to lamp service and provides APIs to turn the appliance on, off and to get the appliance's status. It also publishes "appliance on" and "appliance off" to the event broker. The implementation details and the interface definitions of these device services are given in the next chapter.

CHAPTER 5
IMPLEMENTATION

This chapter discusses the implementation details of event broker, controller and device services components. The entire implementation is done in Java™ programming language and the components are packaged into the following three OSGi bundles:

1. *osgiEvent.jar* (Event Broker component)
2. *osgiDevice.jar* (Device services component)
3. *osgiTw523.jar* (Controller component)

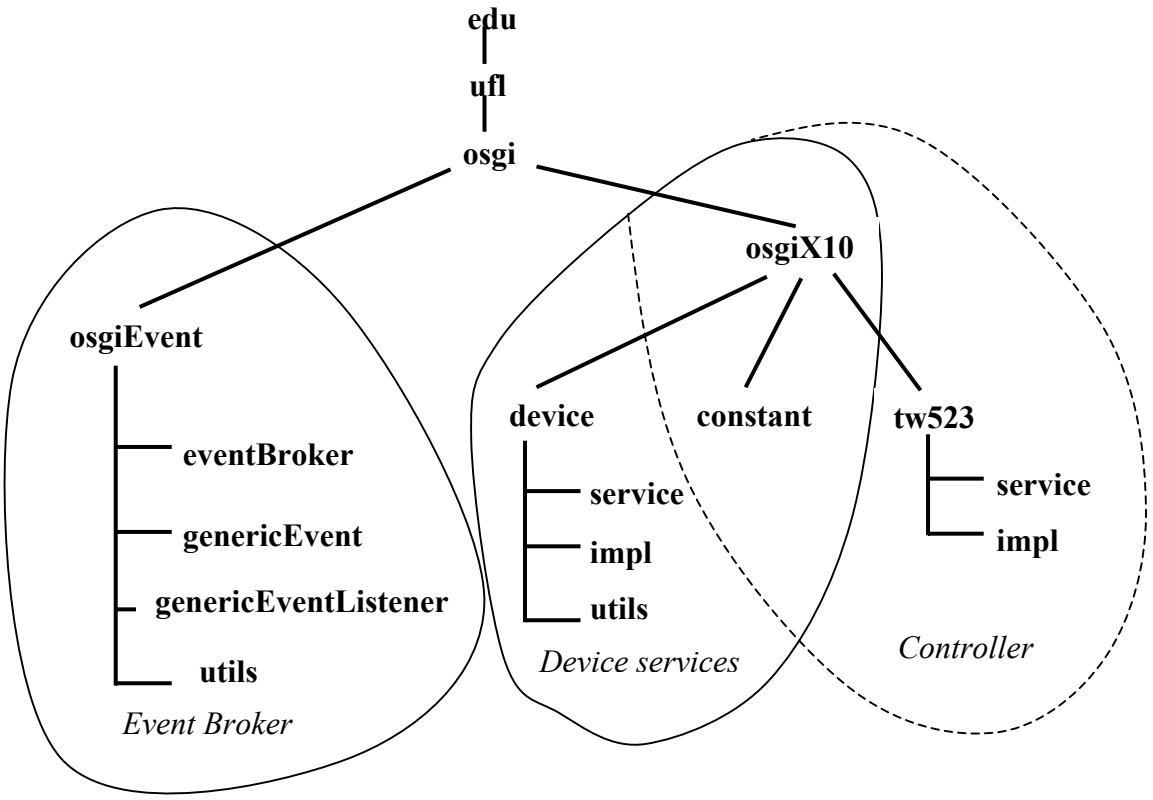


Figure 5.1: Package tree of the software infrastructure

The anatomy of a typical OSGi bundle is explained in section 3.3.1. The software infrastructure package tree is shown in figure 5.1. As it can be seen, both the controller and device services components share the package *edu.ufl.osgiX10.constants*.

The next three sections discuss the implementation details of the three components in detail. Two applications are developed to demonstrate the utility of this infrastructure and these are discussed in chapter 6.

5.1 Event Broker

In this section, the package structure of event broker, which is shown in figure 5.1, is explained. As it can be seen from figure 5.1, all the classes in this component are under *edu.ufl.osgiEvent* package.

5.1.1 Package Details

edu.ufl.osgiEvent.genericEventListener

As explained in section 4.1, every application interested in a particular event, needs to subscribe to the event broker with the event name and a listener object. This package defines a standard listener interface called *GenericEventListener*. The event broker accepts only the listener objects that implement this listener interface. The interface defines just one method called *handleEvent()*. The event broker calls this method while dispatching events. The method *handleEvent()*, expects the event information in an object of type *GenericEvent.* While dispatching events, the event broker creates a new thread of execution on each listener object.

edu.ufl.osgiEvent.GenericEvent

The information about any event is passed in the form of an event object. When an event source generates an event, it signals the event broker by passing the event information in the form of an event object.

The event broker dispatches the event by passing the same event object to all subscribed listeners. All the event objects should be of type *GenericEvent* or its *derived* classes.

edu.ufl.osgiEvent.eventBroker.service

This package has the interface definition of the event broker service. The interface is named *EventBroker.*

edu.ufl.osgiEvent.eventBroker.implementation

This package has the implementation of event broker service i.e., it has a class called *EventBrokerImpl*, which implements the *EventBroker* interface.

edu.ufl.osgiEvent.eventBroker.utils

This package has event table data structures used by the event broker implementation class.

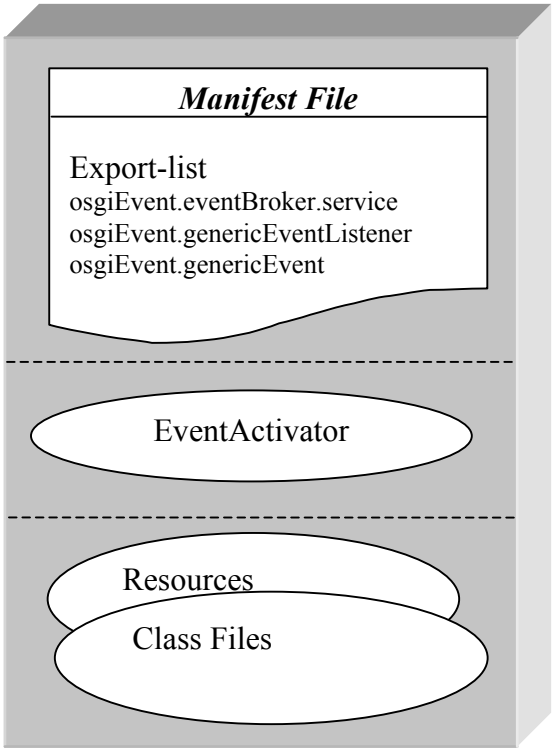


Figure 5.2: Structure of *osgiEvent.jar*

5.1.2 Bundle Structure

The bundle is named as *osgiEvent.jar*. The structure of the bundle is shown in figure 5.2. As it can be seen, the bundle exports the event broker interface, generic listener and the object classes.

5.2 Controller Component

The classes in this component are packaged in *edu.ufl.osgiX10.Tw523* as shown in figure 5.1. This component has the classes that implement a driver service for the Tw523 X10 controller. In this section, the connection details of TW523, the package details and the structure of this component's bundle are explained

5.2.1 Connecting Tw523 to Computer's Serial Port

Tw523 controller comes with two interfaces: RJ11 and a power line socket interface. Since RJ11 interface cannot be directly connected to the computer's serial port, the scheme shown in figure 5.3 is used for connections.

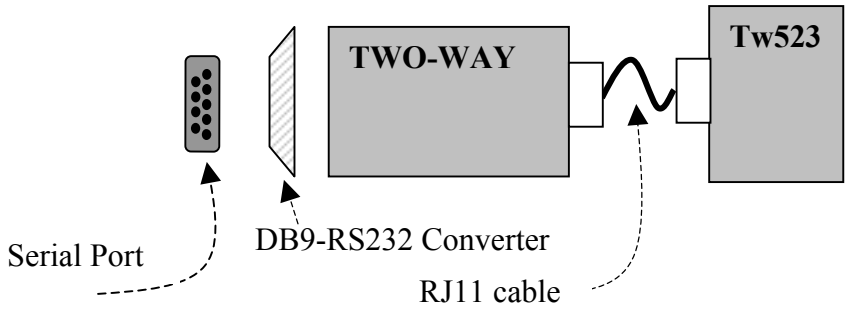


Figure 5.3: Connection details of Tw523

A device, called TWO-WAY, is acts as a RJ11 to DB9 converter. Apart from that, TWO-WAY also accepts inputs from serial port in the form of ASCII text, buffers them and communicates with TW523 taking care of all timing details.

Use of TWO-WAY greatly simplifies the task of developing a driver for Tw523.The DB9 interface of TWO-WAY is connected to the serial port using a DB9/RS232 converter.

5.2.2 Package Details

edu.ufl.osgiX10.constants

This package defines all the constants that are used by X10 applications. The device properties, device types, event names and event types are defined.

edu.ufl.osgiX10.Tw523.service

This package defines the standard Tw523 device service interface. The interface is named *Tw523*.

edu.ufl.osgiX10.Tw523.implementation

This package has a class *Tw523Impl* that implements the *Tw523* interface. This is the core class implementing the driver service for Tw523 device. This class corresponds to the Tw523 software service component explained in section 4.2.3

edu.ufl.osgiX10.Tw523

This package has two classes *Tw523Activator* and *SerialListener*, which, correspond to the serial port scanner and serial port listener components explained in sections 4.2.1 and 4.2.2 respectively.

5.2.3 Bundle Structure

The bundle is named as *osgiTw523.jar*. As it can be seen from figure 5.4, this bundle exports the Tw523 interface and the constants package. Since the classes in this component use the event broker, this bundle imports the event broker packages.

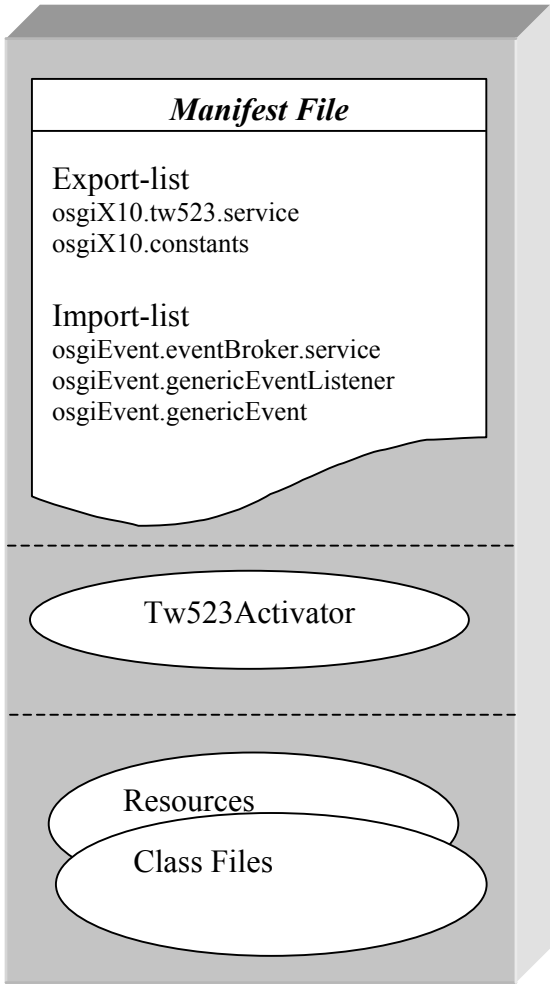


Figure 5.4: Structure of *osgiTw523.jar*

5.3 Device Services Component

The classes in this component are packaged in *edu.ufl.osgiX10.device* and *edu.ufl.osgiX10.constants* packages. The design details are given in section 4.3.

This component registers device services for all devices in a smart home environment. The information about the devices is initially obtained from a configuration file named *X10Mapfile.conf*.

Device services for lamp and small electrical appliance are developed as a part of this thesis. In this section, the standard properties defined by this component and the

configuration file is explained in detail followed by a brief description of packages and bundle structure.

5.3.1 Standard Device Properties Defined in this Component

In section 3.2.1 it was seen that every OSGi service should have some properties associated with it and that these properties help other applications to search for services based on these properties. Since this component creates device services, it should also associate them with some properties. Device services component defines a standard set of properties for every smart home device. These properties are shown in table 5.1.

Table 5.1 Standard properties of devices in smart home

Property Name	**Explanation**
X10_ADDRESS	The device X10 address
ROOM_NUMBER	The room number in which the device is present[1].
FLOOR_NUMBER	The floor number in which the device is present
DEVICE_NUMBER	The device number
DEVICE_TYPE	The type of the device as defined in efu.ufl.osgiX10.contants.DeviceType package

For example, the property 5-tuple <A10, 2, 1, 2, 1>, conveys the following information:

1. It represents an X10 device with address “A10”.
2. The device is located in room number 2, first floor.
3. The device is a lamp, since the device type is 1 (device types are defined in edu.ufl.osgiX10.constants package).
4. The device is the second lamp in the room, since the device number is 2.

It has to be noted that, in addition to the properties listed in table 5.1, the device services may have other properties defined by the user.

5.3.2 X10Mapfile.conf Structure

The configuration file is a text file where each line represents the properties of a particular device. The format of each line is shown in figure 5.5

<X10> <RN> <FN> <DN> <DT> [<p1> <v1> <p2> <v2> …<pn> <vn>]

Legend:

X10 : X10 Address
RN : Room Number
FN : Floor Number
DN : Device Number

Note:

1. The fields are separated by delimiters which can be one of these <,;: \t> If there is no value for the field a "*" has to be entered
2. First five tokens represent the mandatory properties. Hence the property name is not specified (the property names are implicit)
3. The subsequent tokens should be present in (property name, property value) pairs.
4. Device type is according to the values defined in the package *edu.ufl.osgiX10.constants.DeviceTypes*

Figure 5.5: *X10Mapfile.conf* file structure

[1] All the rooms in a smart home should be logically numbered. Similarly all devices in a room should also be logically numbered.

5.3.3 Package Details

edu.ufl.osgiX10.device

This package has the class *DeviceActivator* that corresponds to the device service initiator module described in section 4.3.1. It reads X10Mapfile.conf file and creates the device services.

edu.ufl.osgiX10.device.utils

This package has a utility class called *MapfileLoader* that is used to parse the configuration file i.e., *X10Mapfile.conf* file. *MapfileLoader* has a method called *loadMapfile()* that reads the configuration file and returns a hash table containing the device X10 address and properties.

edu.ufl.osgiX10.device.service

This package defines the interfaces of lamp and general electrical appliance devices.

edu.ufl.osgiX10.device.impl

This package defines the classes that implement the device interfaces defined in the *edu.ufl.osgiX10.device.service* package.

5.3.4 Bundle Structure

The bundle containing the device services component is named *osgiDevice.jar*

Since the device services component uses both the eventing module and the controller module, it imports those packages. The bundle exports the device service package.

The structure of osgiDevice.jar is shown in figure 5.6.

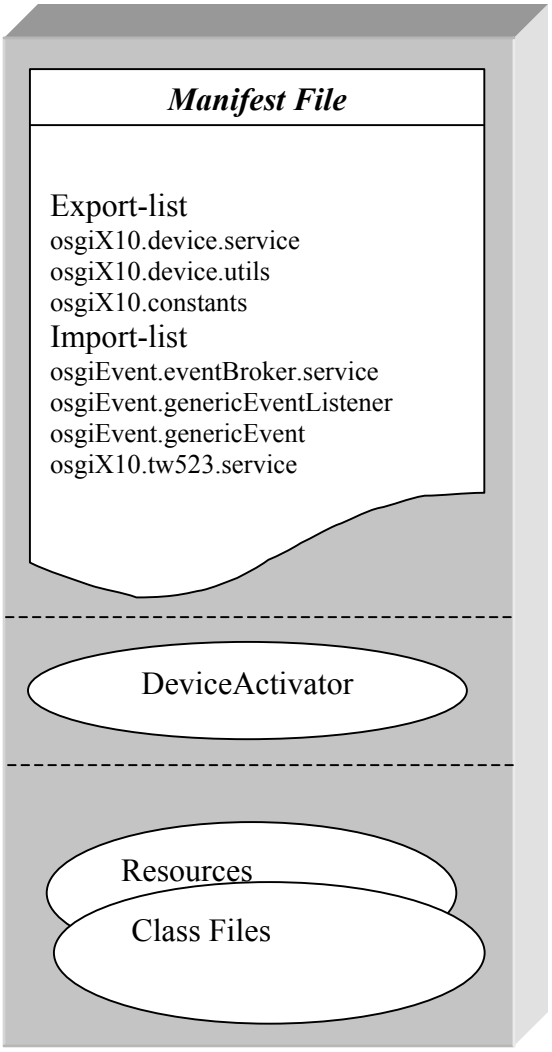


Figure 5.6: Structure of *osgiDevice.jar*

CHAPTER 6
APPLICATIONS USING THE SOFTWARE INFRASTRUCTURE

The advantages of software infrastructure that was explained in chapters 4 and 5 are more evident by considering the applications that are built on the infrastructure.

In this chapter, two such applications developed as a part of this thesis are explained. One of these applications, named *Magic Wand*, is developed for cell phones and the other application, named *Smart App* is for desktops. Both the applications allow the user to remotely control the devices in a smart home. It has to be noted that these applications are developed for demonstration purposes only.

6.1. Design of "Smart App"

Smart App is designed for desktops and it provides a way to remotely control the devices. Figure 6.1 illustrates the design of the application.

6.1.1 Server Side Component of *Smart App*

The server side component of Smart App is called main proxy and it executes on OSGi framework. Main proxy reads the X10mapfile.conf[1] file, creates proxy device services[2] for each device, and creates a table indexed by the devices' X10 addresses. The proxy device services created by main proxy, search for the corresponding device services and associate with them.

The devices that can be controlled using *Smart app* are lamp and radio.

---

[1] Structure of X10Mapfile.conf is explained in section 5.3.2

[2] The design and implementation of device services component was already explained in sections 4.3 and 5.3

Requests from client side of *Smart App* are of the following three types.

1. To get a list of all devices in smart home and their properties (see table 5.1)
2. To control a device
3. To register for events

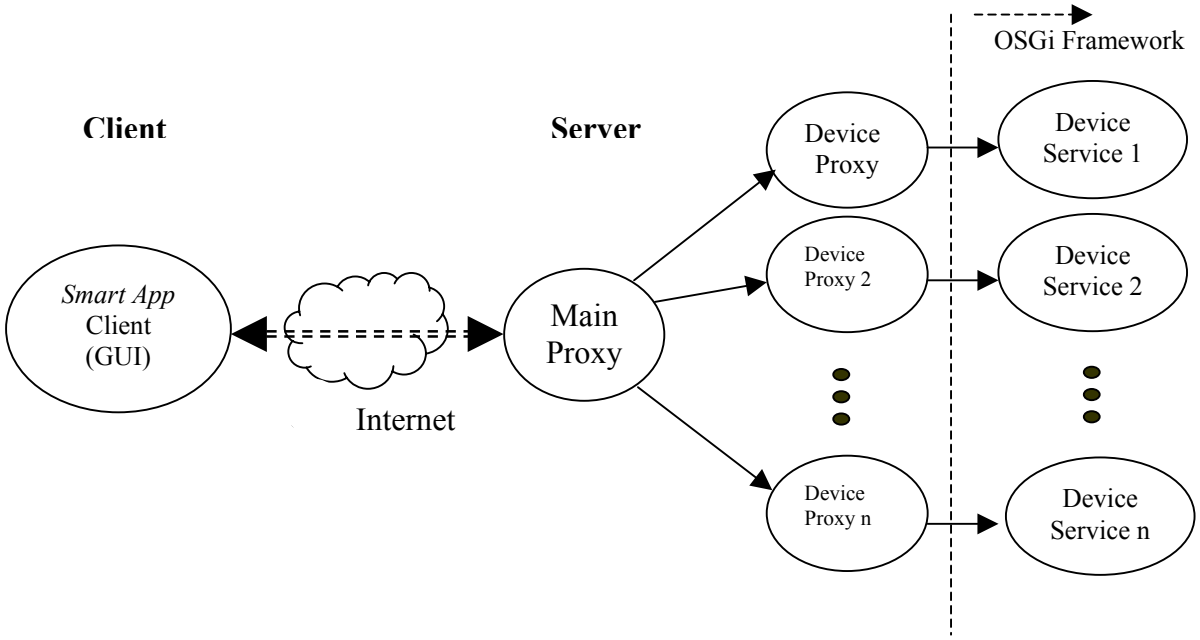


Figure 6.1: Architecture of *Smart App*

Request to get a list of all devices is received by main proxy just once i.e., when the proxy starts. All other requests to control a device or to register for an event have two fields. The first field has the X10 address of the device to be controlled, second field has the appropriate function code and the third field has the function specific data.

When main proxy receives such request packets, it looks at the X10 address field, consults its table to find the corresponding device proxy and forwards the request packet to it. The request is then handled accordingly. Events are discussed in section 6.3.

6.1.2 Client Side Component of *Smart App*

The client side component is designed to execute on any remote computer and it provides a nice graphical user interface (GUI) to control the devices. The screen shot of the GUI is shown in figure 6.2.

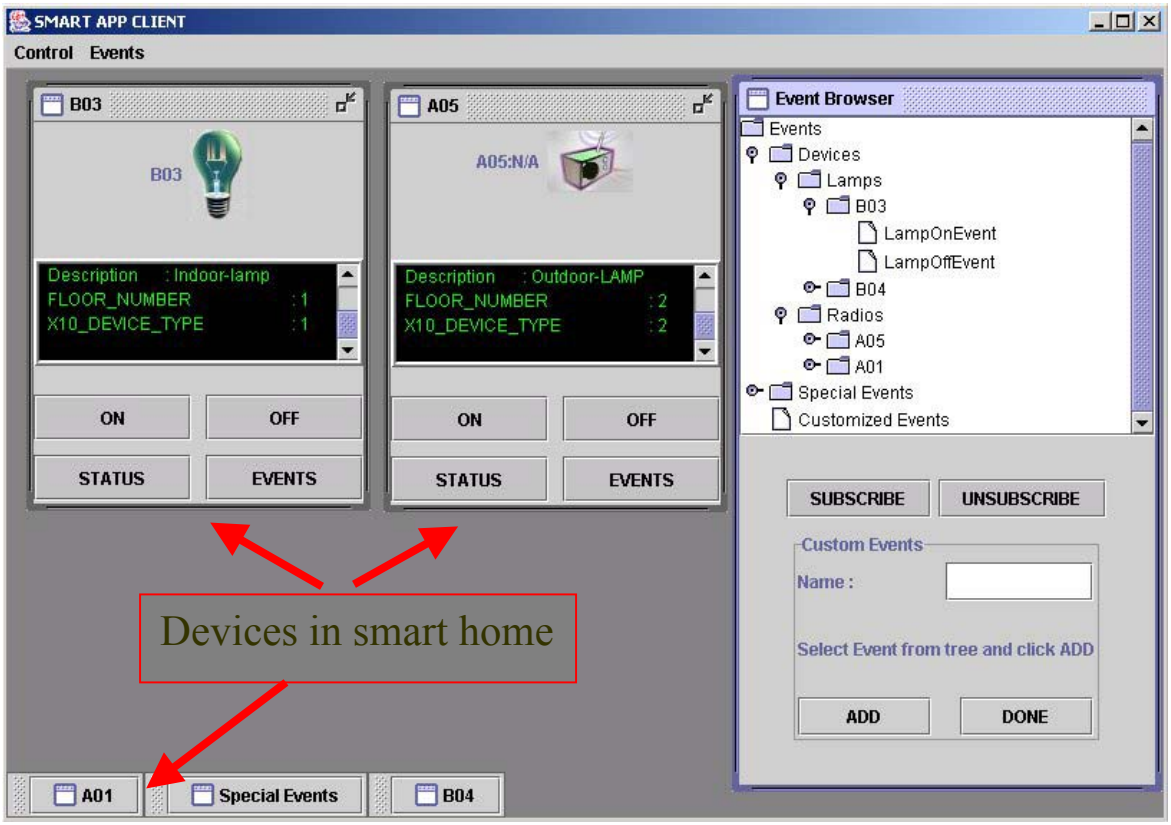


Figure 6.2: *Smart App* GUI

6.2. Design of Magic Wand

Magic Wand is an application designed to control devices in a smart home using Motorola iDEN phones. *Magic Wand* is designed using client-server architecture. The server component executes on the OSGi framework while the client component executes on Motorola iDEN phones. Magic wand application is a simple application and controls only two lamps and a radio.

6.2.1 Server Side Component of *Magic Wand*

When the server side component of Magic Wand is started, it searches the OSGi framework and gets references to device services (two lamps and a radio). The component also registers for *new mail* and *system critical* events, which are explained in section 6.3.

6.2.2 Client Side Component of *Magic Wand*

The client component is designed for Motorola iDEN phones and it provides graphical user interface. Figure 6.3 shows screen shots of the interface.

Figure 6.3: *Magic Wand* GUI

6.3. Events

Though *Magic Wand* and *Smart App* are developed as two independent applications, they control the same set of devices. This is made possible by the software infrastructure.

The advantages of software infrastructure become more evident when the events are considered. The event broker, which is a part of the infrastructure, allows different applications to communicate using events. To demonstrate this feature, the following events are defined

- *Lamp On* event: Generated by the lamp device service when the lamp is turned on
- *Lamp Off* event: Generated by the lamp device service when a lamp is turned off
- *Appliance On* event: Generated by the appliance device service when the appliance turned on
- *Appliance Off* event: Generated by the appliance device service when the appliance turned off
- *System Critical* event: Generated when Tw523 controller is plugged out or plugged into the serial port. This event is generated by the serial port component explained in section 4.2.2
- *New mail* event: Generated when a new mail is kept in the mailbox. This event is generated by the Tw523 software interface component that was explained in section 4.2.3
- *Someone at door* event: Generated when someone rings the door bell. This event is generated by the Tw523 software interface component that was explained in section 4.2.3

By default *Magic wand* is registered to *New mail*, *Someone at door* and *System critical* events. *Smart app* provides interface that displays a list of all events available and lets user choose the events to subscribe/unsubscribe to. The screen shot shown in figure 6.2 shows this event browser interface.

It can be seen that when the lamp or appliance is turned on/off using *Magic Wand*, the events can be seen on *Smart App.* Similarly when *New mail, Someone at door and System critical events* are generated, both *Smart App* and *Magic Wand* receive the notifications. Figure 6.4 shows the screen shots of the critical event when Tw523 controller is unplugged.

(a) Alert message displayed on Smart App GUI when TW523 controller is unplugged on the home gateway

(b) Alert message displayed on Magic Wand GUI when TW523 controller is unplugged on the home

Figure 6.4: Critical events on *Smart App* and *Magic Wand*

CHAPTER 7
CONCLUSION AND FUTURE WORK

The previous chapters explained in detail, the motivation behind this thesis, the proposed software infrastructure and also the applications developed to demonstrate its benefits. This chapter summarizes the achievements of this thesis and concludes with a note on future work that can be done to improve it.

7.1. Achievements and Contributions of this Thesis

This thesis is a modest attempt to capture the essence of application development for smart homes. The major achievement is the software infrastructure it proposes. Though all the components (excluding the event broker) in the infrastructure are developed for X10 technology, the infrastructure essentially suggests a component-based model for application development. The event broker model that is developed provides a simple and elegant way for different applications to coexist and co-operate. The choice of OSGi was obvious as the standard was developed with a similar philosophy – to provide generic framework for application development.

The following are the major contributions made by this thesis

1. Generic event broker component
2. APIs to generate complex events
3. X10 driver to communicate with TW523 interface
4. Device service component to control devices
5. Standardized properties to be associated with every device service (see table 5.1)

Other achievements include a detailed study on existing technologies like X10, CEBus, Lonworks etc., and identifying X10 as the appropriate choice under the present circumstances

7.2 Future Work

As it was said in the previous section, the main purpose of this thesis was to propose a model that would make application development easy. The software infrastructure developed in this thesis focused primarily on the class of applications to control devices in smart homes. While the components are developed to provide a rich set of APIs, there is still a lot of scope for future work. The following list proposes some enhancements than can be done in future

Improving X10 modules: As it was mentioned in section 2.5.2, X10 modules suffer form a lot of reliability issues. X10 provides no means to check if a command was successfully executed or not. This feature can be improved suggested in section 2.6.2. This change was proposed in Microsoft's Aladdin project [WAN00].

Device discovery in X10: X10 modules provide no way to convey information about the type of device plugged. This prevents any device and service discovery mechanism to be implemented using X10. However, the work around suggested by Microsoft's Aladdin project could be used. This work around is explained in section 2.6.1.

Providing a generic "Serial port" device service: Any Java™ based application that communicates with the serial port uses the javax.comm. API to communicate with the serial port. To use a serial port, the application has to own the port, which prevents any other application from using the serial port. For example, the Tw523 software interface module developed as a part of this thesis owns all the serial ports and prevents any other applications from using them. Developing a "serial port" device service on OSGi can solve this. The service would own the port and cater to multiple applications in an OSGi environment.

Event broker: The event broker can be made more robust by enabling it to dispatch events even over the network. This requires defining APIs to register remote listeners. RMI technology can be used to implement this feature.

7.3 Conclusion

The range of applications that can be developed is infinite as it depends on ones imagination and creativity. This is a first step in this direction and the author envisions that the near future would see a gamut of applications to make our lives more pleasurable and independent.

LIST OF REFERENCES

[CEB00] CEBus, *CEBus Industrial Council*, http://www.cebus.org, 03/2002

[CHE02] Kirk Chen, Li Gong, *Programming Open Service Gateways with Java Embedded Server™ Technology*, Addison-Wesley, Boston, 2002

[EVA01] Gray Evans, *CEBus Demystified: The ANSI/EIA 600 User Guide,* McGraw Hill, Columbus, 2001

[HOR00] Cay S. Horstmann, Gary Cornell, *Core Java*, Prentice Hall India, New Delhi 2000

[KIN02] Phil Kingery, *X10-Technical Series*, http://www.hometoys.com/resources.htm, 03/2002

[OSG01] OSGi, *OSGi Service Platform: Release 2 Specification*, October 2001 www.osgi.org, 04/2002

[SCH01] Herbert Schildt, *Java™ Complete Reference*, Tata McGraw Hill, New Delhi, 2001

[WAN00] Yi-Min Wang, Wilf Russell, Anish Arora, *A Toolkit for Building Dependable and Extensible Home Networking Applications,* USENIX technical program, Winsys, August 2000, http://www.usenix.org/events/usenix-win2000/wang.html, 03/2002

BIOGRAPHICAL SKETCH

Sree Charan Kuchibhotla was born on November 20th, 1976, in Visakhapatnam, India. He received his bachelor's degree in engineering from Karnataka Regional Engineering College, Surathkal, India, in June 1999. Soon after graduating, he worked for IBM, India as a software developer until December 2000. His area of work in IBM was on network security and network file systems.

He joined the University of Florida in January 2001, to pursue a master's degree in computer and information sciences. He served as a teaching assistant for Mr. William B. Noffsinger till May 2002.

Sree Charan became a part of Harris Communications laboratory in August 2001 and his major achievements include winning first place in Motorola killer app. competitions in December 2001. He has a passion for new technology and his interests are in the field of pervasive computing.

After graduation, Sree Charan will be moving to Madison, Wisconsin, where he will be working as a full time software developer in EPIC systems corp.