12-20-2002

A Service Discovery-Enabled LCD Projector Device

Jeevan Kale
University of New Orleans

A SERVICE DISCOVERY-ENABLED LCD PROJECTOR DEVICE

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
In partial fulfillment of the
Requirements for the degree of

Master of Science
in
The Department of Computer Science

by

Jeevan Kale

B.E., Pune Institute of Computer Technology, Pune, India, 1999

December 2002

ACKNOWLEDGEMENT

I would like to express my deepest appreciation towards my thesis advisor Dr. Golden Richard III, for sharing his knowledge and providing me with many helpful comments. He was very patient and cheerful throughout the duration of my thesis. I would also like to thank him for providing me his book on 'Service and Device Discovery'. It was a great help towards clarifying the concepts and writing the report as well.

I would like to thank Dr. Markus Montigel and Dr. Ming-Hsing Chiu for being on my thesis committee. It was a great honor having them both. I would like to give my special thanks to Dr. Montigel for his valuable help in pointing out the mistakes in the report and help making it more understandable, and a worthwhile one.

I would like to thank my family members for being patient, for all their support and love. I am grateful to have Amit, Pushkar, Sanjay, Ashwini and Urmila in my life. They really make my world a better place.

I am grateful to the city of New Orleans for all the good times, wonderful people, great experiences and all the wonderful memories it has engraved onto my heart.

I would also like to add that student life in the University of New Orleans was both challenging and rewarding. It is really a memorable experience of my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF CODE LISTS

ABSTRACT

The widespread deployment of inexpensive communications technology, computational resources in the networking infrastructure and network-enabled end devices pose a problem for end users: how to locate a particular network service or device out of those accessible. Service providers use Service Discovery Services (SDS) to advertise the descriptions of available or already running services, while clients use SDS to compose queries for locating these services. Service descriptions and queries use the eXtensible Markup Language (XML[9]) to encode vendor specific information and device- or service-specific capabilities as well as the actions addressed to the device or service. This report presents the architecture and implementation of a SDS used to locate enabled LCD projectors and use them for presentation. The presentation service provides all the capabilities to the end user so that he can choose the projector device of his interest and use the graphical user interface to navigate thorough the presentation. The presentation service also has the capability to use more than one projector at a time. We use the Universal Plug and Play[7] suite of protocols to establish the communication between client and the projector device.

CHAPTER 1. INTRODUCTION

Service discovery[1] allows clients to perform discovery in order to find needed services. Clients do so by multicasting their discovery request on the network. In order for clients to perform discovery, they need to know very little about their environment. On the other hand, when a service enters the network, it performs service advertisement, either directly to clients or to service catalogs. The advertisement has necessary contact information, and will also include either descriptive attributes or information that will allow these attributes to be discovered. After a client has found the service of their interest they can make use of all the services provided by the service.

In my application, I have implemented a virtual projector device that allows operating an LCD projector remotely. The projector device acts as the device driver for the LCD projector. It provides all the basic function of the LCD projector and makes is convenient for the user to perform the slide show or presentation. The set of functionalities provided by the projector device are broadly categorized into three service groups:

- Power Service
- Presentation Service
- Slide-Control Service

The power service allows the user to turn on / off the projector. A presentation can only be performed if the projector is on.

The presentation service allows the user to choose the presentation. In our application the slides used for the presentation are JPEG image file named serially like '1.jpg', '2.jpg', …'n.jpg'. And they are stored in a directory, the name of which is used as the presentation name. We use 'Electric Eyes', an image displaying software to display our slides.

The slide-control service allows all the basic functions required to perform the presentation. It is absolutely necessary to set the slide-range, i.e. how many slides are present in the presentation. The functions provided are: viewing next slide, previous slide, jump to a particular slide, the slide number for which is accepted from the user and display a blank slide.

The LCD projector application achieves the following service discovery capabilities, which are discussed more in detail in the section 2.1.3.

- Discovery of services[1]

The client can know all the projector devices on the network. The devices are discovered using the device type ('Projector').

- Service Advertisement[1]

Whenever the projector device becomes available, it advertises its presence and the service offered. Each advertisement has a timer associated with it. If it expires without further advertisement the device is assumed to be unavailable.

- Service browsing[1]

A Graphical User Interface is created on the client side, which can show all the available services and the user can select the one of interest.

- Eventing[1]

Eventing provides asynchronous notification of the interesting conditions like changes in the state of a service; e.g. a printer needs a change of toner. Eventing makes the developer's job easier, eliminating the use of polling to watch for service state changes. Whenever there are changes in the device state, e.g. "projector power turned on", the client is notified of the change. We will discuss this in more detail in section 4.2.

The interesting feature implemented in this application is that one can simultaneously operate more than one projector at a time. A 'queue' is implemented which keeps track of all the projector devices. Using the GUI user can switch between the available projectors.

The entire system is developed under the Red Hat Linux platform with Intel UPnP SDK v1.0.3[4]. All the coding was written in C except the GUI was implemented in Java to make it portable across various platforms. The Java Native Interface[2] (JNI) is used to communicate between the GUI and client side. We will discuss more about JNI in chapter 2.

The rest of the chapters are organized as follows: In chapter 2 we talk in detail about service discovery technology. We will also talk about UPnP[4], the technology we use to achieve the service discovery paradigm and in detail about JNI technology provided by Java, which we use to talk back and forth between the GUI and the client. Chapter 3 talks about the design and

architecture of the projector service we have implemented. It also describes a working scenario of the projector service. Chapter 4 explains the implementation of the service. Finally chapter 5 concludes the work and talk about the future work to improve the scalability of the application.

CHAPTER 2. BACKGROUND

2.1 Service Discovery[1]

A service discovery framework is a collection of protocols for developing dynamic client/server applications. Enabled services announce their presence when they enter the network and (if possible) announce their demise while leaving the network. Service catalogs are used to track the available services on the network. Garbage collection facilities clean up the system by removing outdated information. Service discovery technologies generalize and standardize the environments in which client/server applications are developed and used, and provide software tools that make the development effort much easier and the interactions between clients and services more dynamic.

In the service discovery framework, clients perform discovery in order to find needed services. Clients either may directly seek the needed service themselves or else they may contact one or more service catalogs. Clients do so by multicasting their discovery request on the network. Clients need to do very little or no static configuration prior to dynamically locate the needed service, and clients need very little information about their environment.

When a service enters the network, it performs service advertisement, either directly to clients or to service catalogs. The advertisement has necessary contact information, and will also

include either descriptive attributes or information that will allow these attributes to be discovered.

A major problem one has to deal with is the installation, configuration and management of peripherals. The complexity of this problem is becoming more serious as laptops, handheld computers, printers, scanners and other peripherals are integrated into networked environments. It is quite possible that a new user might get frustrated with all the installation routines. For experienced users it is a waste of valuable time. Service discovery technologies take an important step toward eliminating manually installed device drivers, relying instead on standard interfaces to put devices in touch. Service discovery technologies allow services introduced in the network to be discovered, configured and used with a minimum manual intervention.

In a networked environment with services like printing, storage, scanning etc., it is inconvenient to manually determine the services available to users. Service discovery can make things more convenient by allowing the types of available services to be discovered easily. And since the interfaces between client and server are standardized, the client can get to work quickly without manual configuration.

2.1.1 Service Discovery Technologies

In this section we briefly discuss several leading service discovery technologies including Jini[10], UPnP[4] and SLP[11].

2.1.1.1 Jini[10]

Jini is a service discovery technology based on Java, developed by Sun Microsystems. Because of the platform-independent nature of Java, Jini can rely on mobile code to control services. Lookup services provide catalogs of available services to clients in a Jini network. Jini services register their availability by uploading proxy objects to one or more of these lookup services. The proxy objects are essentially "device drivers" written in Java – they allow interaction with the service. Clients can dynamically discover lookup services, search for interesting services, and then download these proxy objects. Searching is based on the type of proxy objects and on sets of descriptive attributes. Jini requires that at least one lookup service is accessible. Clients and services in Jini cannot discover each other directly.

2.1.1.2 Service Location Protocol[11] (SLP)

SLP is an IETF standards-track protocol for IP-based networks. It is language neutral, although APIs are defined for C and Java. In SLP, User agents (UAs) search for needed services on behalf of clients. Service agents (SAs) advertise the availability of services, either directly to UAs or to Directory agents (DAs), if at least one DA is available. Directory agents serve some of the purposes of the lookup servers in Jini by cataloging available services, but they store only contact information – not code. Service location information in SLP is encoded in service URLs, which contain all the information necessary for contacting a service. In contrast to Jini, SLP is concerned primarily with putting clients in touch with services – the specific protocol spoken between clients and services are outside the scope of SLP. Service templates are used to standardize particular service types (such as print services). They also provide the syntax of service URLs for a specific type and also standardize the set of descriptive attributes. UAs locate

interesting services by service type, further narrowing searches by the values of descriptive attributes.

2.1.1.3 Universal Plug and Play[4] (UPnP)

Universal Plug and Play is a set of protocols for service discovery under development by the Universal Plug and Play forum, an industry consortium by Microsoft. UPnP standardizes the protocols used to communicate between clients and services. Device and service descriptions are coded in XML and a number of protocols for auto configuration, discovery, advertisement, client/server interaction and eventing-Auto-IP[4], SSDP[4], SOAP[4] and GENA[4]- are included in the specification. These protocols are based on existing standards like HTTP. Unlike Jini and SLP, UPnP does not support service directories. The communication between devices and clients is always direct.

We use Universal Plug and Play technique for our implementations. One of the most significant reasons for using UPnP is that it gives the choice of language and operating system to be used for the implementation. Also, UPnP provides the flexibility to leverage lower layers of the protocol stack where this makes sense, thereby keeping UPnP designs simple and effective. A detailed description of this technology is given in the following chapter.

2.1.2 Common Characteristics

Despite their differences, current service discovery technologies like Jini, UPnP, and SLP share many characteristics. Most include the following capabilities:

- Discovery of services

Needed services may be discovered on demand, with minimal prior knowledge of network. Typically, clients can search for services by type ("projector") or by descriptive attributes ("manufacturer name") or by both.

- Service "subtyping"

A client may be interested in a very specific type of service; for example, a high-resolution color laser printer with duplex capability. A client needing services like these must make these needs known. On the other hand, a client who is searching for a basic printer should be able to discover one without knowing many details. Service subtyping allows the detailed attributes of the printer to be defined. So the client needing to print can be as specific as he wishes to be.

- Service insertion and advertisement

Services can enter into the network with minimal manual intervention and advertise their availability directly either to clients or to servers maintaining service catalogs. Conversely, services leaving a network can advertise their demise. The main difference between service discovery technologies and static information services such as DHCP[12] is that service discovery technologies allow highly dynamic updates – services appearing or disappearing result in immediate updates. Service discovery technologies also provide more powerful searching capabilities than the static information services. For example, static information systems such as DHCP can search for an IP address for a machine from a specified range of addresses. This range is predefined and DHCP has to choose an address out of this range. Service discovery searches

are based on dynamic input from the client who is looking for a particular service. Clients can search for services based upon their type, unique names or specific descriptive attributes. Service discovery technique supports this wide range of inputs, which make its searching capabilities very powerful.

- Service browsing

A client can browse the list of available services and can choose the one of interest. A GUI can be used to give the list of available services that a client can use.

- Catalogs of available services

Some service discovery technologies use service catalogs to keep track of available services. Services register their availability with the catalogs and clients can obtain contact information for the service of interest directly from the catalog. The service catalogs reduce the multicast traffic, but add another component to be administered. Jini uses service catalogs, while in UPnP, clients and services can directly address each other using multicast for the purpose of discovery or advertisement. SLP supports catalogs but their deployment is optional.

- Eventing

Eventing provides asynchronous notification of the interesting conditions like changes in the state of a service; e.g. printer needs a change of toner. Eventing makes the developer's job easier, eliminating use polling to watch for service state changes.

- Garbage collection

Garbage collection mechanisms are normally included in a service discovery framework to expire the service availability information, to remove the outdated information from catalogs of available services, and to terminate client initiated eventing. Leases are a popular way of garbage collection, used in Jini. In SLP and UPnP, garbage collection facility is implemented using a timer associated with the advertisement requests. If the timer expires without receiving an advertisement requests from the service, the service is assumed to be inaccessible.

2.2 Universal Plug and Play

Among all the available service discovery suites we use UPnP technology to achieve service discovery. In this section we will talk about UPnP in depth. The following sections describe the responsibilities of UPnP, the steps involved in UPnP networking, components of the UPnP network and the protocol stack of UPnP.

2.2.1 Overview

Universal Plug and Play (UPnP) is the architecture for pervasive peer-to-peer network connectivity of PCs of all form factors, intelligent appliances and wireless devices. UPnP is a distributed, open networking architecture. With the addition of Device Plug and Play (PnP) capabilities to the operating system it became a great deal easier to setup, configure and add peripherals to a PC. Universal Plug and Play (UPnP) extends this simplicity to include the entire network, enabling discovery and control of networked devices and services, such as network-attached printers, Internet gateways, and consumer electronics equipment.

UPnP is more than just a simple extension of the Plug and Play peripheral model. It is designed to support zero-configuration, “invisible” networking and automatic discovery for a breadth of device categories from a wide range of vendors.

2.2.2 How Universal Plug and Play Works

With UPnP, a device can dynamically join a network, obtain an IP address, convey its capabilities, and learn about the presence and capabilities of other devices – all automatically, truly enabling zero configuration networks. Devices can subsequently communicate with each other directly, thereby further enabling peer to peer networking.

The varieties of device types that can benefit from the UPnP enabled network are large and include intelligent appliances, wireless devices, and PCs of all form factors. UPnP can be used in scenarios like home automation, printing audio/video entertainment, and automobile networks.

UPnP uses standard TCP/IP and Internet protocols. Using these standardized protocols allows UPnP to benefit from the use of existing technology and thereby making interoperability easier. Because UPnP is a distributed, open network architecture, defined by the protocols used, it is independent of any particular operating system, programming language, or physical medium. UPnP does not specify the APIs applications will use, allowing programmers to create the APIs that will meet their requirements.

2.2.3 The Responsibilities of Universal Plug and Play

UPnP provides support for communication between control points; i.e. clients and devices. The network media, the TCP/IP protocol suite and HTTP provide basic network connectivity and addressing needed. On top of these open, standard, Internet based protocols, UPnP defines a set of HTTP-based protocols to handle discovery, description, control, events, and presentation.

This section describes how the protocols defined earlier in this article are used to provide for these needs.

2.2.4 Steps Involved in UPnP Networking

Universal Plug and Play networking involves following six steps:

1. Addressing
2. Discovery
3. Description
4. Control
5. Eventing
6. Presentation

2.2.4.1 Addressing

The foundation for UPnP networking is the TCP/IP protocol suite and the key to this suite is addressing. Each device must have a Dynamic Host Configuration Protocol (DHCP) client and search for a DHCP server when the device is first connected to the network. If a

DHCP server is available, the device must use the IP address assigned to it. If no DHCP server is available, the device must use Auto IP to get an address.

In brief, Auto IP defines how a device intelligently chooses an IP address from a set of reserved private addresses, and is able to move easily between managed and unmanaged networks.

A device may implement higher layer protocols outside of UPnP that use friendly names for devices. In these cases, it becomes necessary to resolve friendly host (device) names to IP address. Domain Name Services[18] (DNS) are usually used for this. A device that requires or uses this functionality may include a DNS client and may support dynamic DNS registration for its own name to address mapping.

2.2.4.2 Discovery

Once devices are attached to the network and addressed appropriately, discovery can take place. Discovery is handled by Simple Service Discovery Protocol (SSDP), which will be discussed in the following section. When a device is added to the network, SSDP allows this device to advertise its services to control points on the network. When a control point is added to the network, SSDP allows that control point to search for devices of interest on the network.

The fundamental exchange in both cases is a discovery message containing a few, essential specifics about the device or one of its services, for example its type, identifier, and a pointer to its XML device description document. This document provides general information about the device such as manufacturer information, unique device name etc. and lists all the services provided by this device.

2.2.4.3 Description

The next step in UPnP networking is description. After a control point has discovered a device, the control point still knows very little about the device. For the control point to learn more about the device and its capabilities, or to interact with the device, the control point must retrieve the device's description from the URL provided by the device in the discovery message.

Devices may contain other logical devices and services. The UPnP description for a device is expressed in XML and includes vendor-specific manufacturer information, including the model name and number, serial number, manufacturer name, URLs to vendor-specific Web sites, and so forth. The description also includes a list of any embedded devices or services, as well as URLs for control, eventing, and presentation.

2.2.4.4 Control

After a control point has retrieved a description of the device, the control point has the essentials for device control. To learn more about the service, a control point must retrieve a detailed UPnP description for each service. The description for a service is also expressed in XML and includes a list of the commands, or actions, the service responds to, and parameters or arguments for each action. The description for a service also includes a list of variables; these variables model the state of the service at run time, and are described in terms of their data type, range, and event characteristics.

To control a device, a control point sends an action request to a device's service. To do this, a control point sends a suitable control message to the control URL for the service (provided

in the device description). Control messages are also expressed in XML using Simple Object Access Protocol (SOAP), which will be discussed in the following section.

In response to the control message, the service returns action specific values or fault codes.

2.2.4.5 Eventing

The UPnP description for a service includes a list of actions the service responds to and a list of variables that model the state of the service at run time. The service publishes updates when these variables change, and a control point may subscribe to receive this information.

The service publishes updates by sending event messages. Event messages contain the names of one or more state variables and the current value of those variables. These messages are also expressed in XML and formatted using General Event Notification Architecture (GENA), which will be discussed in the following section.

A special initial event message is sent when a control point first subscribes; this event message contains the names and values for all evented variables and allows the subscriber to initialize its model of the state of the service.

To support multiple control points, all subscribers are sent all event messages, subscribers receive event messages for all evented variables, and event messages are sent no matter why the state variable changed (in response to an action request or due to a state change).

2.2.4.6 Presentation

If a device has a URL for presentation, then the control point can retrieve a page from this URL, load the page into a browser and depending on the capabilities of the page allow a user to control the device and/or view device status. The degree to which each of these can be accomplished depends on the specific capabilities of the presentation page and device.

2.2.5 Components of the UPnP Network

Figure 2.1 shows the basic building blocks of the UPnP network. They are devices, services and control points.

2.2.5.1 Devices

The UPnP device is a container of services and nested devices. For instance, a VCR device may consist of a tape transport service, a tuner service, and a clock service. A TV/VCR combo device would consist not just of services, but of nested sets of devices as well.

Different categories of UPnP devices will be associated with different sets of services and embedded devices. For instance, services within a VCR will be different than those within a printer. Consequently, different working groups will standardize the set of services that a particular device type will provide. This information is captured in an XML device description document that the device must host. In addition to the set of services, the device description also lists the properties (such as device name and icons) associated with the device.

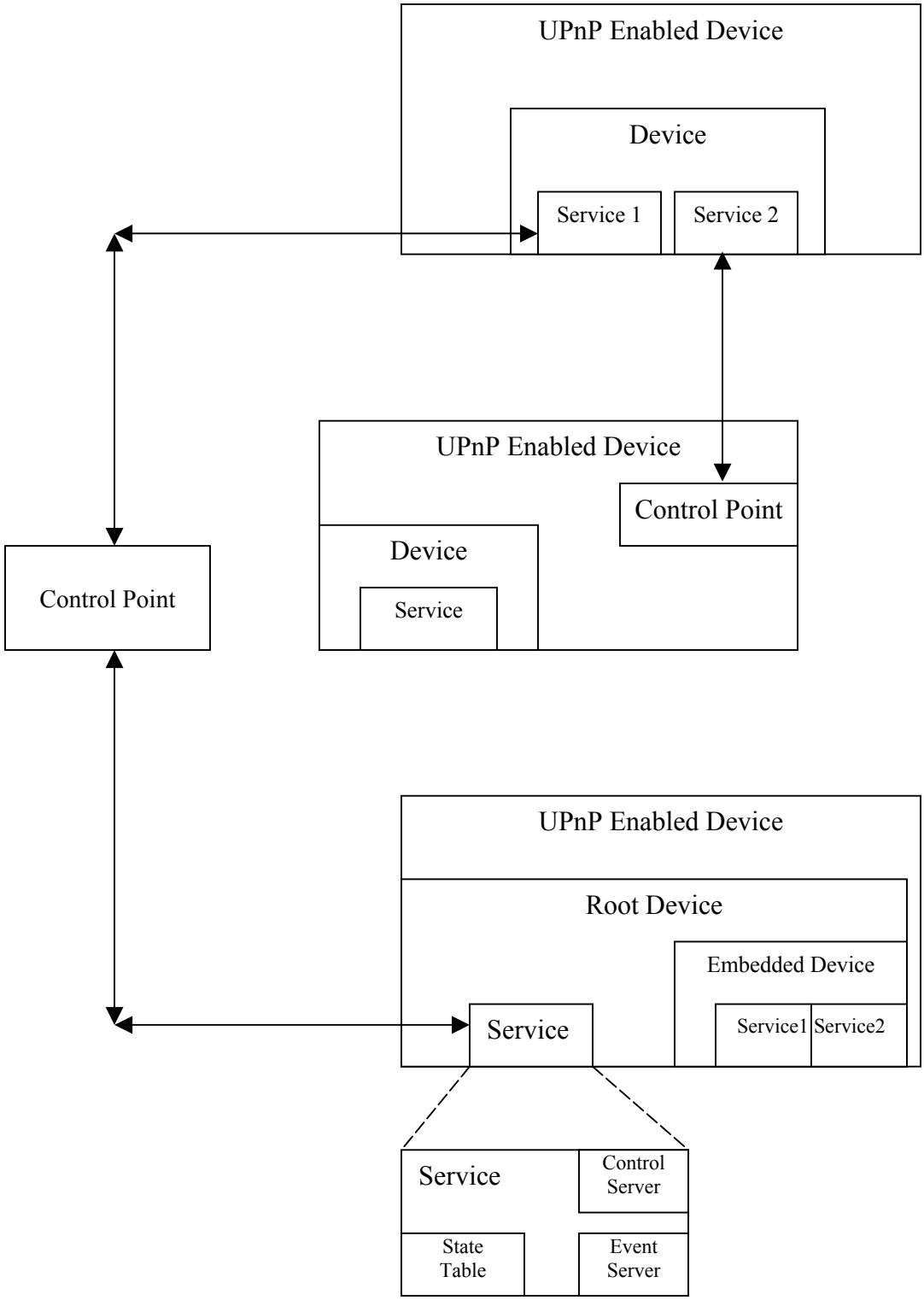


Fig 2.1 Components of the UPnP network[5]

2.2.5.2 Services

The smallest unit of control in the UPnP network is a service. A service exposes actions and models its state with state variables. For instance, a clock service could be modeled as having a state variable, current_time, which defines the state of the clock, and two actions, set_time and get_time, which allow users to control the service. Similar to the device description, this information is part of an XML service description standardized by the UPnP forum. A pointer (URL) to these service descriptions is contained within the device description document. Devices may contain multiple services.

A service in the UPnP device consists of a state table, a control server and an event server. The state table models the state of the service through state variables and updates them when the state changes. The control server receives action requests (such as set_time), executes them, and updates the state table and returns responses. The event server publishes events to interested subscribers anytime the state of the service changes. For instance, a fire alarm service would send an event to interested subscribers when its state changes to “ringing.”

2.2.5.3 Control Points

A control point in the UPnP network is a controller capable of discovering and controlling other devices. After discovery, a control point could:

- Retrieve the device description and get a list of associated services.
- Retrieve service descriptions for interesting services.
- Invoke actions to control the service.

- Subscribe to the service's event source. Anytime the state of the service changes, the event server will send an event to the control point.

It is expected that devices will incorporate control point functionality (and vice-versa) to enable true peer-to-peer networking.

2.2.6 The UPnP protocol stack

UPnP leverages many existing, standard protocols. Using these standardized protocols aids in ensuring interoperability between vendor implementations. The protocols used to implement UPnP are found in use on the Internet and on local area networks everywhere. This prevalence ensures that there is a large pool of people knowledgeable in implementing and deploying solutions based on these protocols. Since the same protocols are already in use, little would need to be done to make UPnP devices work in an existing networked environment. Some of the protocols used to implement UPnP are summarized in the rest of this section.

The UPnP Device Architecture defines a schema or template for creating device and service descriptions for any device or service type. Individual working committees subsequently standardize on various device and service types and create a template for each individual device or service type. Finally, a vendor fills in this template with information specific to the device or service, such as the device name, model number, manufacturer name and URL to the service description. This data is then encapsulated in the UPnP-specific protocols defined in the UPnP Device Architecture document (such as the XML device description template).

The required UPnP specific information is inserted into all messages before they are formatted using SSDP, GENA, and SOAP and delivered via HTTP, HTTPU, or HTTPMU, which are discussed in the following section.

Figure 2.2 depicts how the UPnP protocol stack is organized. We explain each of the protocols in the following section.

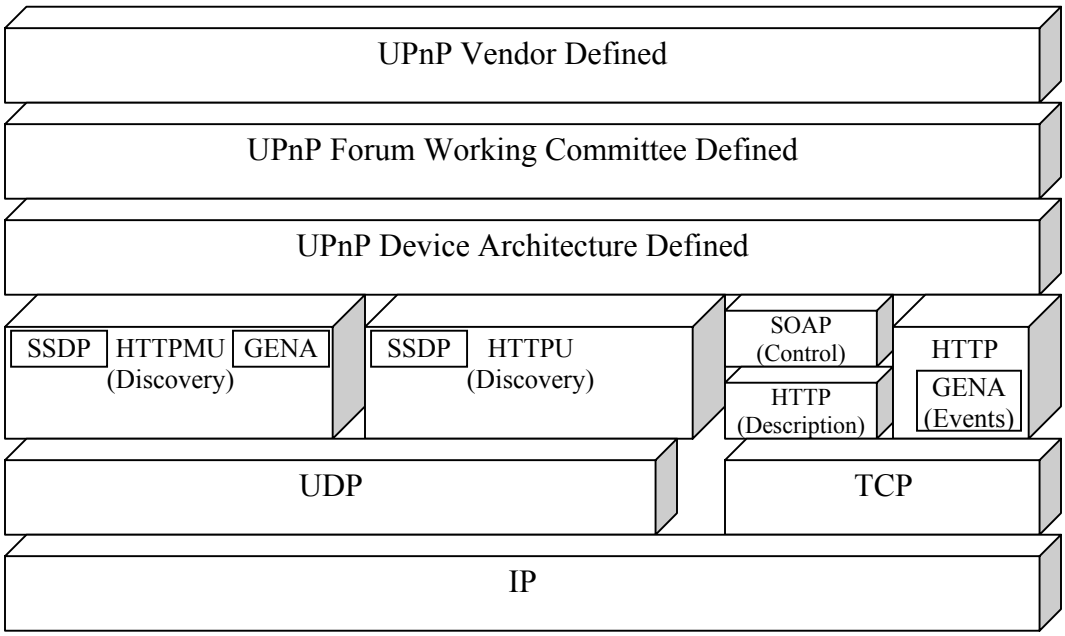


Fig 2.2 The UPnP Protocol Stack[5]

2.2.6.1 UPnP Specific Protocols

UPnP vendors, UPnP Forum Working Committees and the UPnP Device Architecture document define the highest layer protocols used to implement UPnP. Based on the device architecture, the working committees define specifications specific to device types such as VCRs, HVAC systems, dishwashers, and other appliances. Subsequently, UPnP Device Vendors add the data specific to their devices such as the model name, URL, etc.

2.2.6.1.1 TCP/IP[5]

The TCP/IP networking protocol stack serves as the base on which the rest of the UPnP protocols are built. By using the standard, prevalent TCP/IP protocol suite, UPnP leverages the protocol's ability to span different physical media and ensures multiple vendor interoperability.

UPnP devices can use many of the protocols in the TCP/IP stack including TCP, UDP, ARP and IP. How these protocols and services are used to provide what is required for UPnP to work will become clear as we discuss the other protocols in this section and discuss how UPnP works in subsequent sections.

Since TCP/IP is one of the most ubiquitous networking protocols, locating or creating an implementation for the UPnP device that is tuned for footprint and/or performance is relatively easy.

2.2.6.1.2 HTTP[5], HTTPU[5], HTTPMU[5]

TCP/IP provides the base protocol stack to provide network connectivity between UPnP devices. HTTP, which is hugely responsible for the success of the Internet, is also a core part of UPnP. All aspects of UPnP build on top of HTTP or its variants.

HTTPU (and HTTPMU) are variants of HTTP defined to deliver messages on top of UDP/IP instead of TCP/IP. These protocols are used by SSDP, described next. The basic message formats used by these protocols adheres with that of HTTP and is required both for multicast communication and when message delivery does not require the overhead associated with reliability.

2.2.6.1.3 SSDP[5]

Simple Service Discovery Protocol (SSDP), as the name implies, defines how network services can be discovered on the network. SSDP is built on HTTPU and HTTPMU and defines methods both for a control point to locate resources of interest on the network, and for devices to announce their availability on the network. By defining the use of both search requests and presence announcements, SSDP eliminates the overhead that would be necessary if only one of these mechanisms is used. As a result, every control point on the network has complete information on network state while keeping network traffic low.

Both control points and devices use SSDP. The UPnP control point, upon booting up, can send an SSDP search request (over HTTPMU), to discover devices and services that are available on the network. The control point can refine the search to find only devices of a particular type (such as a VCR), particular services (such as devices with clock services) or even a particular device, using its universally unique ID.

UPnP devices listen to the multicast port. Upon receiving a search request, the device examines the search criteria to determine if they match. If a match is found, a unicast SSDP (over HTTPU) response is sent to the control point. Similarly, a device, upon being plugged into the network, will send out multiple multicast SSDP presence announcements advertising the services it supports. Both presence announcements and unicast device response messages contain a pointer to the location of the device description document, which has information on the set of properties and services supported by the device. In addition to the discovery capabilities provided, SSDP also provides a way for a device and associated service(s) to gracefully leave the

network (bye-bye notification) and includes cache timeouts to purge stale information for self healing.

2.2.6.1.4 GENA[5]

Generic Event Notification Architecture (GENA) was defined to provide the ability to send and receive notifications using HTTP over TCP/IP and multicast UDP. GENA also defines the concepts of subscribers and publishers of notifications to enable events.

GENA formats are used in UPnP to create the presence announcements to be sent using Simple Service Discovery Protocol (SSDP) and to provide the ability to signal changes in service state for UPnP eventing. A control point interested in receiving event notifications will subscribe to an event source by sending a request that includes the service of interest, a location to send the events to and a subscription time for the event notification.

The subscription must be renewed periodically to continue to receive notifications, and can also be canceled using GENA.

2.2.6.1.5 SOAP[5]

Simple Object Access Protocol (SOAP) defines the use of Extensible Markup Language (XML) and HTTP to execute remote procedure calls. It is becoming the standard for RPC based communication over the Internet. By making use of the Internet's existing infrastructure, it can work effectively with firewalls and proxies. SOAP can also make use of Secure Sockets Layer (SSL) for security and use HTTP's connection management facilities, thereby making distributed communication over the Internet as easy as accessing web pages.

Much like a remote procedure call, UPnP uses SOAP to deliver control messages to devices and return results or errors back to control points.

Each UPnP control request is a SOAP message that contains the action to invoke along with a set of parameters. The response is a SOAP message as well and contains the status, a return value and any return parameters.

2.2.6.1.6 XML

The eXtensible Markup Language (XML), to use the W3C[5] definition, is the universal format for structured data on the Web. In other words, XML is a way to place nearly any kind of structured data into a text file.

XML looks a lot like HTML in that it uses tags and attributes. Actually, it is quite different in that these tags and attributes are not globally defined as to their meaning, but are interpreted within the context of their use. These features of XML make it a good fit for developing schemas for various document types. The use of XML as a schema language is defined by the W3C[5]. XML is a core part of UPnP used in device and service descriptions, control messages and eventing.

2.3 Java Native Interface[2]

In our LCD projector application, we have implemented the GUI in Java. Implementing GUI in Java makes it portable as well as easier to code because of the various AWT[14] libraries available. The backend client application is implemented in C, which a problem of communication between the GUI and the client application. The solution to this problem is the Java Native Interface (JNI), a native programming interface for Java.

2.3.1 Overview

The Java Native Interface (JNI) is the native programming interface for Java that is part of the JDK. The JNI allows Java code that runs within a Java Virtual Machine[14] (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly. In addition, the *Invocation API* allows programmers to embed the Java Virtual Machine into the native applications. Programmers use the JNI to write native methods to handle those situations when an application cannot be written entirely in the Java programming language. For example, one may need to use native methods and the JNI in the following situations:

- The standard Java class library may not support the platform-dependent features needed by the application.
- One may already have a library or application written in another programming language and wishes to make it accessible to Java applications.
- One may want to implement a small portion of time-critical code in a lower-level programming language, such as assembly, and then have a Java application call these functions.

Programming through the JNI framework lets the programmer use native methods to do many operations. Native methods may either originate from legacy applications or they may be written explicitly to solve a problem that is best handled outside of the Java programming environment. The JNI framework lets a native method utilize the Java objects in the same way that Java code uses these objects. A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code. A native method can even update Java objects that it created or that were passed to it, and these updated objects are available to the Java application. Thus, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them. Native methods can also easily call Java methods. Often, one will already have developed a library of Java methods. The native method does not need "re-invent the wheel" to perform functionality already incorporated in existing Java methods. The native method, using the JNI framework, can call the existing Java method, pass it the required parameters, and get the results back when the method completes.

JNI makes available the advantages of the Java programming language to native methods. In particular, one can catch and throw exceptions from the native method and have these exceptions handled in the Java application. Native methods can also get information about Java classes. By calling special JNI functions, native methods can load Java classes and obtain class information. Finally, native methods can use the JNI to perform runtime type checking.

It is easy to see that the JNI serves as the glue between Java and native applications. Figure 2.3 shows how the JNI ties the C side of an application to the Java side.

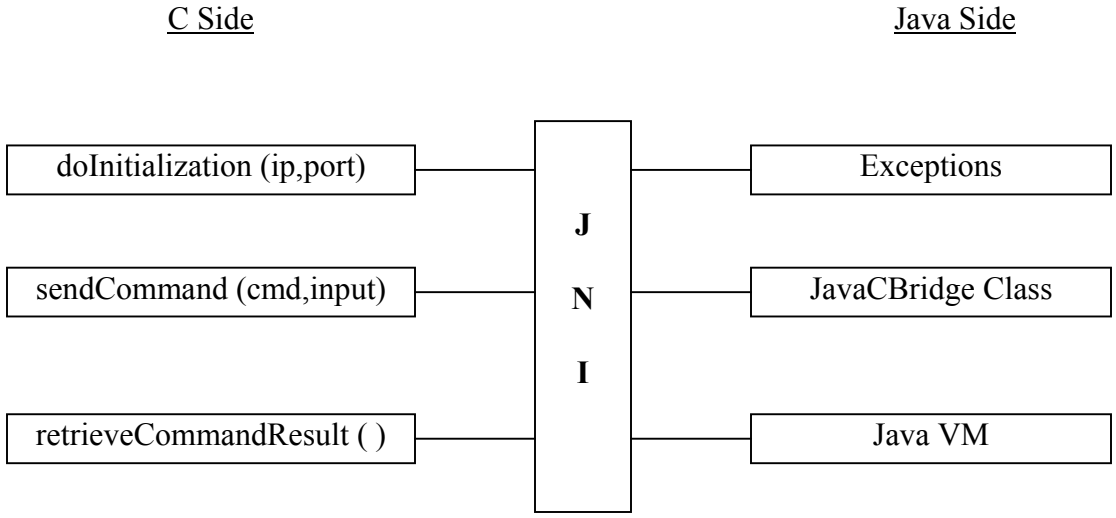


Fig 2.3 GUI and Client application interaction using JNI[2]

2.3.2 Writing a Java Program with Native Methods

1. Begin by writing the Java program. Create a Java class that declares the native method; this class contains the declaration or signature for the native method. It also includes a main method, which calls the native method.
2. Compile the Java class that declares the native method and the main method.
3. Generate a header file for the native method using javah with the native interface flag -jni. Once the header file is generated one can have the formal signature for his native method.
4. Write the implementation of the native method in the programming language such as C or C++.
5. Compile the header and implementation files into a shared library file.
6. Run the Java program.

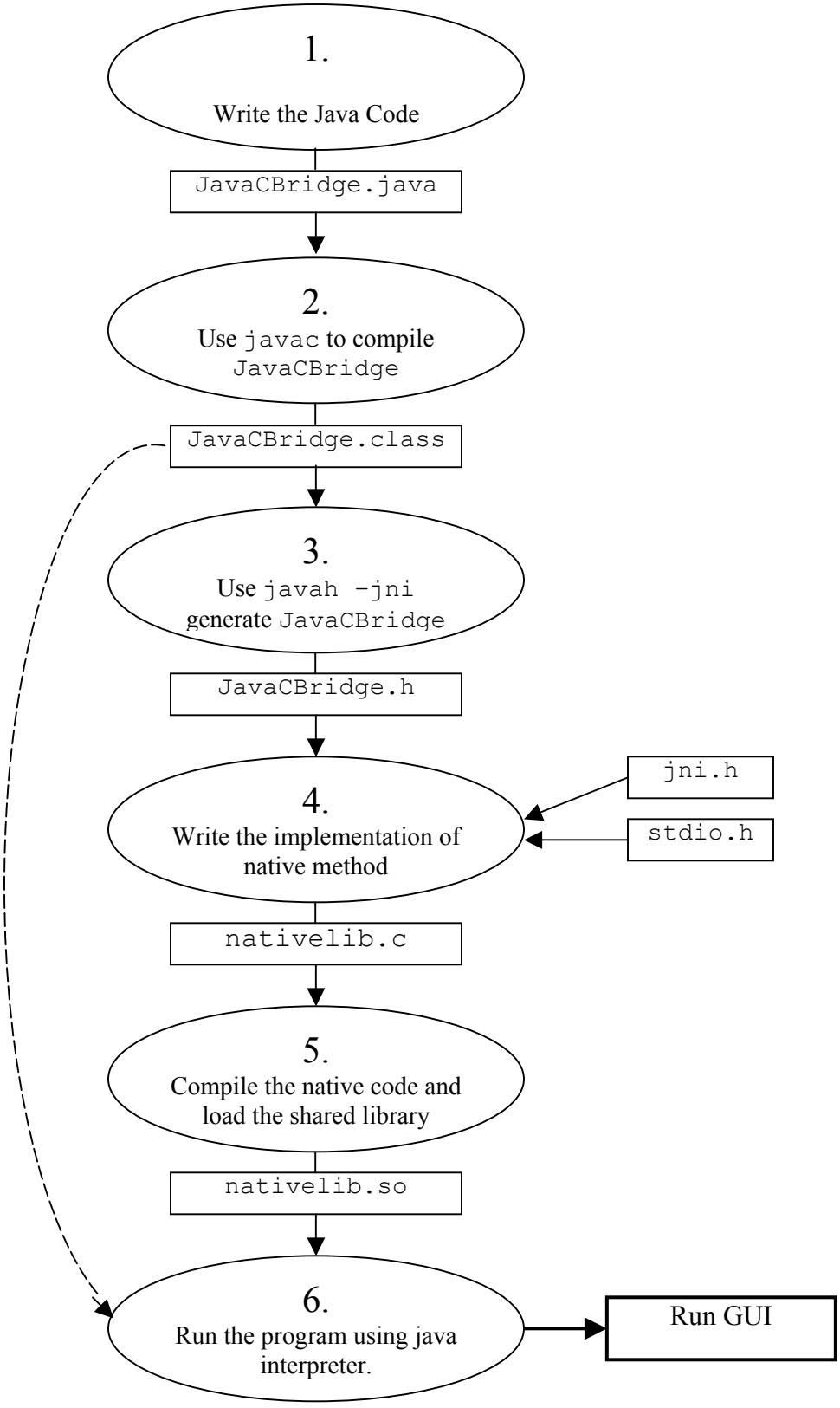


Fig 2.4 Steps writing native methods for Java programs[8]

2.3.3 Integrating Java and Native Programs

The Java Native Interface defines a standard naming and calling convention so that the Java Virtual Machine (VM) can locate and invoke the native methods. This section shows how to follow the JNI naming and calling conventions so that one can use JNI functions from a native method. It also explains how to declare types so that both the Java program and the native method can correctly recognize them.

2.3.4 Declaring Native Methods

This section illustrates how to declare a native method in Java and how to generate the corresponding C/C++ function prototype.

2.3.4.1 The Java Side

Our example, JavaCBridge.java, contains a native method that accepts a Java string, sends it to the native implementation and performs respective control operation. The program calls the native method, which waits for user input and then returns the result of the control operation.

The GUI class accepts an input from the user if necessary, which invokes the JavaCBridge object, and intern a native method named sendCommand, which is declared as follows:

```
public native int sendCommand(String cmd, String input);
```

Notice that the declarations for native methods are almost identical to the declarations for regular, non-native Java methods. However, we declare native methods differently, as follows:

- First, native methods must have the `native` keyword. The `native` keyword informs the Java compiler that the implementation for this method is provided in another language.
- Secondly, the native method declaration is terminated with a semicolon (the statement terminator symbol) because the Java class file does not include implementations for native methods.

2.3.4.2 The Native Language Side

One must declare and implement native methods in a native language, such as C or C++. Before doing this, it is helpful to generate the header file that contains the function prototype for the native method implementation.

Compile the *Prompt.java* file and then generate the `.h` file.

javac JavaCBridge.java

Once successfully compiled *JavaCBridge.java* and have created the *JavaCBridge.class* file, one can generate a JNI-style header file by specifying a *-jni* option to *javah*:

javah -jni JavaCBridge

Examine the JavaCBridge.h file. Note the function prototype for the native method sendCommand that is declared in JavaCBridge.java

```
JNIEXPORT jint JNICALL
Java_JavaCBridge_sendCommand (JNIEnv *, jobject, jstring, jstring);
```

The native method function definition in the implementation code must match the generated function signature in the header file. `JNIEXPORT` and `JNICALL` must always be included in the native method function signatures. `JNIEXPORT` and `JNICALL` ensure that the source code compiles on platforms such as Microsoft Windows that require special keywords for functions exported from dynamic link libraries.

Native method names are concatenated from the following components:

- the prefix `Java_`
- the fully qualified class name
- an underscore "_" separator
- the method name

Fig 2.5 shows how it looks like graphically,

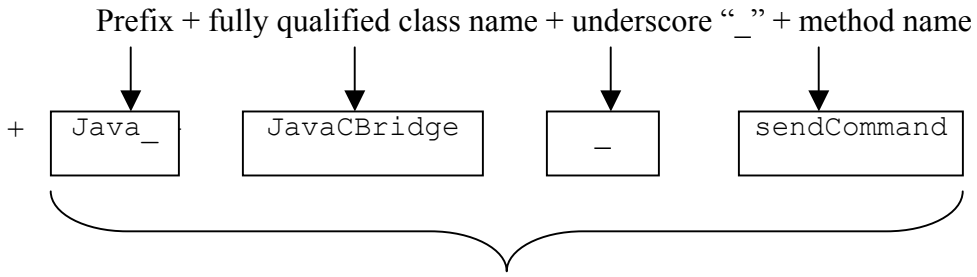


Fig 2.5 Graphical representation of native method names[8]

Thus, the native code implementation for the *JavaCBridge.sendCommand* method becomes *Java_JavaCBridge_sendCommand*. (Remember that no package name component appears because the JavaCBridge class is in the default package.)

Each native method has two parameters in addition to those parameters that is declared on the Java side. The first parameter, *JNIEnv* *, is the JNI interface pointer. This interface pointer is organized as a function table, with every JNI function at a known table entry point. The native method invokes specific JNI functions to access Java objects through the *JNIEnv* * pointer. The *jobject* parameter is a reference to the object itself (it is like the '*this*' pointer in Java).

2.3.5 Mapping between Java and Native Types

In this section, we will see how to reference Java types from the native method. One needs to do this when he wants to:

- Access arguments passed in to a native method from a Java application.
- Create Java objects in the native method.
- Have the native method return results to the caller.

2.3.5.1 Java Primitive Types

A native method can directly access Java **primitive** types such as booleans, integers, floats, and so on, that are passed from programs written in Java. For example, the Java type boolean maps to the native language type jboolean (represented as unsigned 8 bits), while the Java type float maps to the native language type jfloat (represented by 32 bits). The following table describes the mapping of Java primitive types to native types.

2.3.5.2 Primitive Types and Native Equivalents[8]

Java Type	Native Type	Size in bits
`boolean`	`jboolean`	8, unsigned
`byte`	`jbyte`	8
`char`	`jchar`	16, unsigned
`short`	`jshort`	16
`int`	`jint`	32
`long`	`jlong`	64
`float`	`jfloat`	32
`double`	`jdouble`	64
`void`	`void`	n/a

2.3.5.3 Java Object Types

Java objects are passed by reference. All references to Java objects have the type `jobject`. For convenience and to avoid programming errors, the JNI implements a set of types that are conceptually all sub classed from (or are "subtypes" of) `jobject`, as follows:

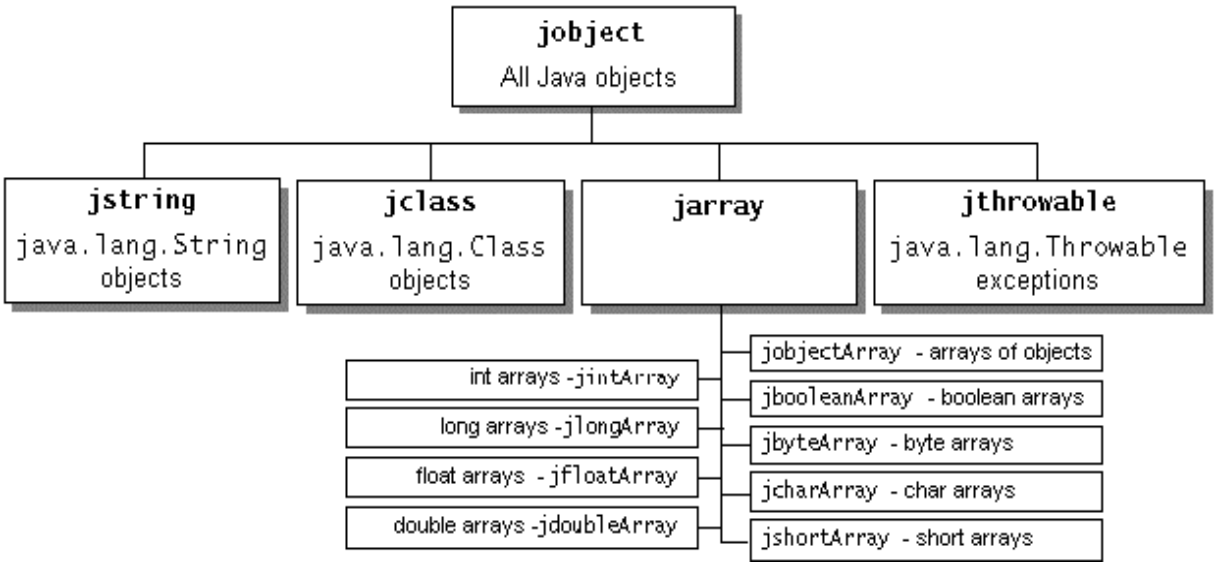


Fig 2.6 Java Object types[8]

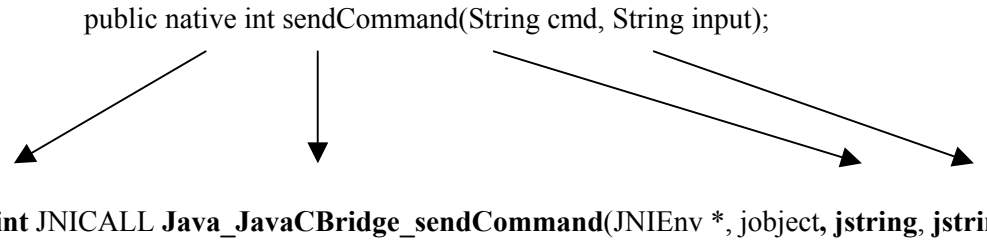
In our JavaCBridge.java example, the native method sendCommand takes two Java strings as arguments and returns a Java int:

```
public native int sendCommand (String cmd, String input);
```

Its corresponding native implementation has type jstring for the arguments and jint for the return value:

```
JNIEXPORT jint JNICALL
Java_JavaCBridge_sendCommand (JNIEnv *, jobject, jstring, jstring);
```

Graphically, this looks as follows:



As mentioned above, *jstring* corresponds to the Java type String and *jint* corresponds to the Java type int. Notice that the second argument to *Java_JavaCBridge_sendCommand*, which is the reference to the object itself, has type *jobject*.

CHAPTER 3. SYSTEM DESCRIPTION AND DESIGN

3.1 System Description

We use the LCD Projector device system to demonstrate the service discovery paradigm. It is a real life projector, which is used to display the presentation. This system consists of two major units:

- Virtual Projector Device
- Client Unit

The virtual projector device advertises its presence over the network, informs its capabilities to the interested control point, performs the duties mentioned in the control actions it receives from the control points, and sends out the events as a result of change in the control variables. It consists of the following three services:

- Power Management Service
- Slide Control Service
- Presentation Service.

The power management service is used to power on / off the selected projector from the list of available projectors at the control point. The projector can be used to display the slides only if it is powered on.

The slide control service provides all the slide presentation related functions. Prior to using any slide control functions one must set the slide range first; it indicates the number of slides in the presentation. The functions include set slide range, go to next slide, see the previous slide, go to a particular slide, and display blank slide.

The presentation related service allows to select the presentation that user wants to show. It is the name of the directory where all the slides are stored. Slides are the JPEG image files numbered '1.jpg', '2.jpg', '3.jpg', …, 'n.jpg'. The presentation must be selected prior to starting the slide show. 'n' represents the number of slides in the presentation; the slide range.

The Client Unit, which discovers the projector device and uses the services described above, consists of two major components:

- Graphical User Interface
- Virtual Control Point.

The GUI provides the user interface for the entire system. The GUI can be used to search for available projector devices over the network, select the device of interest out of the available devices, select the presentation, set the slide range and perform a slide show using available controls to view next slide, go to previous slide, jump to a slide and display blank slide.

The main function of the virtual control point is keep track of available projector devices over the network. When a user selects the device of interest via GUI, it can download the necessary information about the device. It can subscribe to receive interesting events, send control actions (e.g. set presentation, advance slide etc.) to the device of interest and track the changes in the device state.

3.2 System Design

The LCD projector system is designed in such a way that the users of the system can make use of all the services, which are provided by the projector device. The complete system can be broadly categorized into three units:

- Graphical User Interface
- Virtual Control Point
- Virtual Projector Device

3.2.1 Graphical User Interface

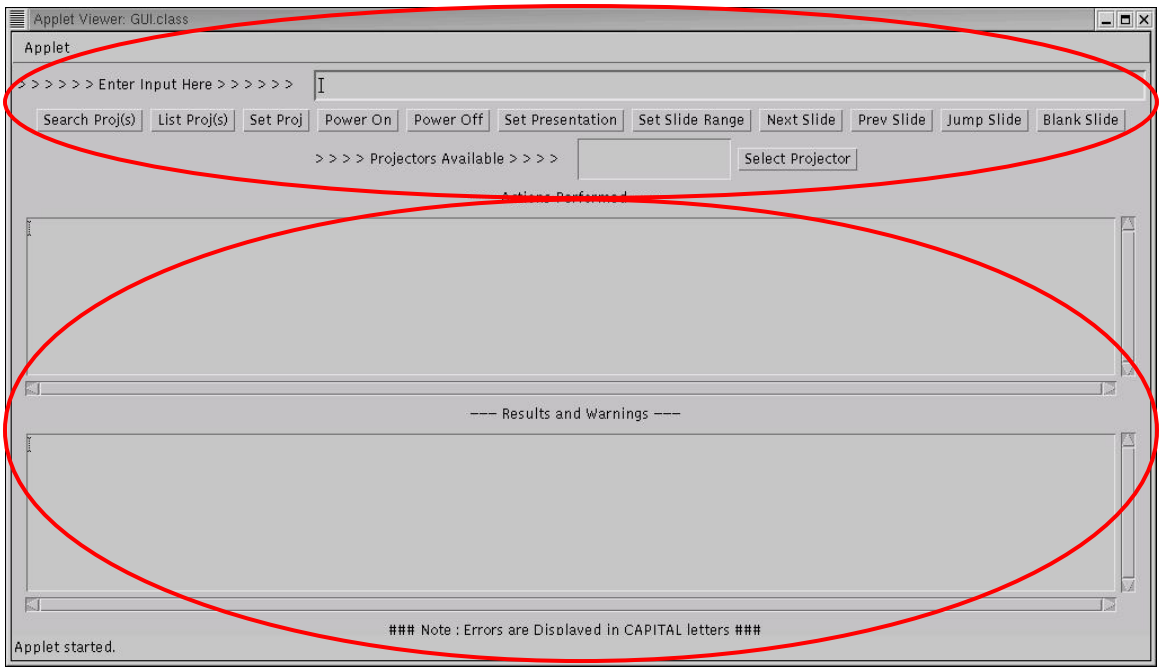
It consists mainly of two parts:

- Operational Part
- Display Unit

Figure 3.1 shows how the actual Graphical User Interface is organized showing the operational part and the display unit.

The operational part is used to operate the LCD Projector. With the help of the operational part of the user interface, one can make use of the services provided by the projector device such as power management service, slide control service and the presentation service.

Operational Part



Result Display Unit

Fig 3.1 Graphical User Interface

The second part is a display unit, which displays the results and warnings that are generated as the result of using the various services provided by the projector device. The GUI is created using basic Java AWT classes. To be able to talk to the control point code, which is written in C, we make use of the Java Native Interface provided by Java.

Figure 3.2 shows the interaction between the GUI and the control point. A class named JavaCBridge is created, which acts as an interface between the GUI and the control point. JavaCBridge defines three native functions, which can be called from the GUI in order to communicate with the virtual control point. One is used to send the user commands with any input data to the control point and the other used to retrieve the results from the control point.

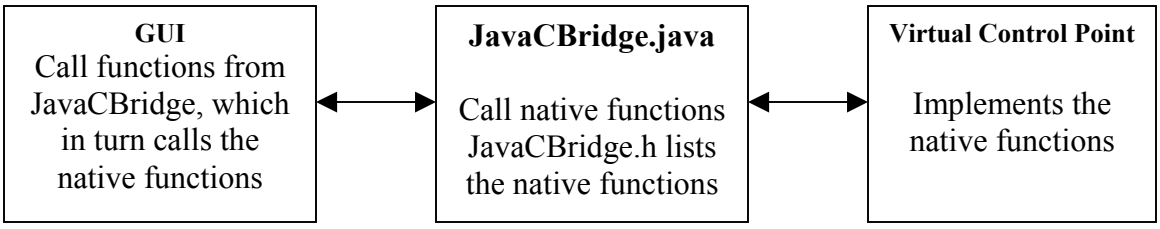


Fig 3.2 Interaction between GUI and Control Point

3.2.2 Virtual Control Point

This is the client-side application, which takes user commands from the GUI. It can discover and control projector devices available over the wireless network. Once discovered the control point can retrieve the device description and get a list of associated services, retrieve service descriptions for interesting services, invoke actions to control the service and can subscribe to the service's event source to keep track of the changes in the state of the projector device. The virtual control point is implemented in C and the name of the implementation is 'nativelib.c'. It implements native functions, which will be used by the GUI to pass the user requests.

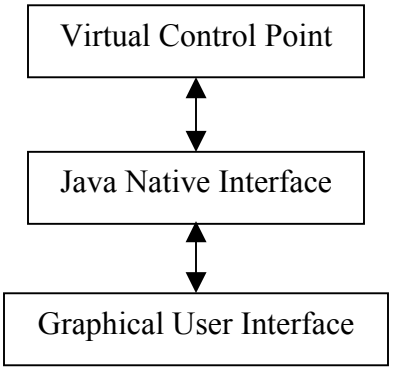


Fig 3.3 Virtual Control Point

3.2.3 Virtual Projector Device

The virtual projector device is a container of three different services, namely a power management service, a slide control service, and a presentation service. The device description is maintained in the XML description document. The device description document not only lists the set of services, but also lists the properties (such as device name) associated with the device. The device description for the LCD Projector device is maintained in an XML file named 'ProjectorDevDesc.xml'. There is a Service Control Protocol Description file associated with each of the service the projector device provides. 'ProjectorPowerSCPD.xml' is for the power service of the device. 'ProjectorPresSCPD.xml' is for the presentation service and the slide control service has 'ProjectorSlideCtrlSCPD.xml' as the Service Control Protocol Description. The virtual projector is implemented in C and the name of the implementation is 'projector.c'.

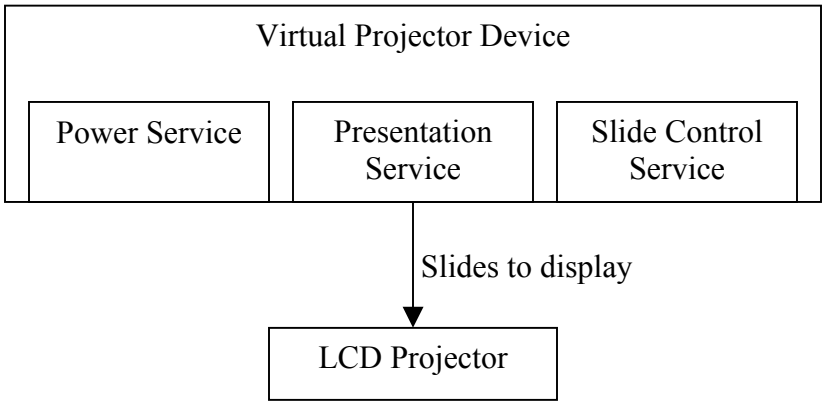


Fig 3.4 Virtual Projector Device

3.3 System Architecture

Figure 3.5 depicts the system architecture. It shows the interaction between various components involved in the system. The projector device and the control point communicate with each other via the UPnP paradigm. The various protocols provided by the UPnP paradigm are used to pass the messages back and forth between the projector device and the control point. The communication that takes place between the various components of the system is explained using the following steps. These steps also explain which messages are passed between the control point and the projector device. The format of all exchanged messages and actions is also explained in these steps. The steps are:

- Discovery
- Description
- Control
- Eventing, and
- Presentation

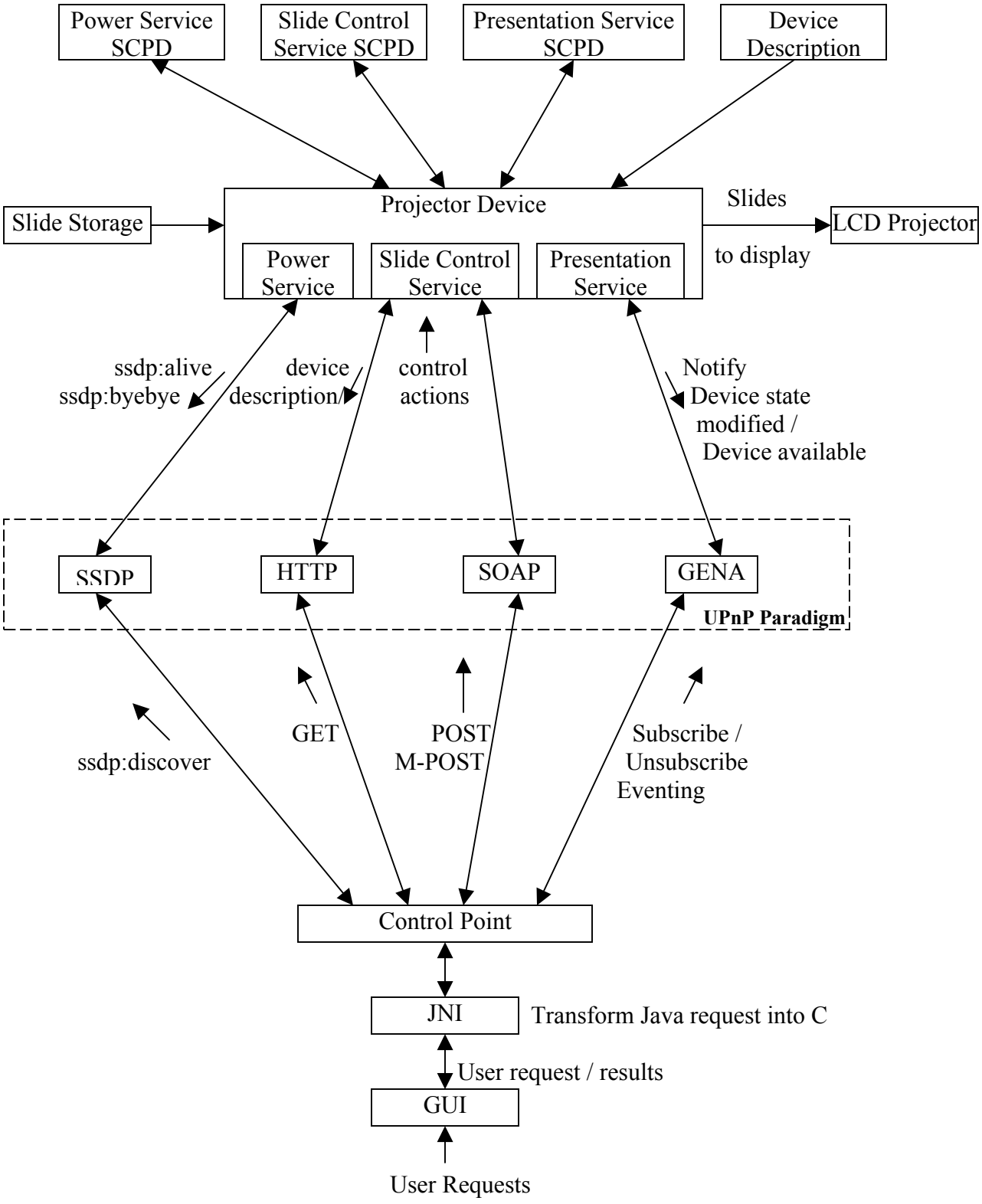


Fig 3.5 System Architecture

3.3.1 Discovery

From a device perspective, discovery governs the process of periodically advertising services offered by the device and responding to queries for service by control point. For control points, discovery allows interesting services to be discovered and used. When a device initializes and joins a UPnP network, a series of advertisement messages are multicast that provide information about device, any embedded devices and the services offered. The advertisements are sent using "ssdp:alive" messages. When a UPnP device wishes to leave the network, it sends one "ssdp:byebye[4]" message for each advertisement that was sent when it entered the network.

An example of "ssdp:aalive[4]" message, which is used to advertise the root projector device is illustrated below:

NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age=1800
LOCATION: http://137.30.123.107:5431/ProjectorDevDesc.xml
NT: upnp:rootdevice
NTS: ssdp:alive
SERVER: Linux/2.4.2-2 UPnP/1.0 Intel UPnP SDK/1.0
USN: uuid:Upnp-Projector-1_0-1234567890001::upnp:rootdevice

The HOST header always contains 239.255.255.250:1900, which is a multicast address and port reserved for the SSDP. The max-age field in the CACHE-CONTROL[1] header is the advertisement duration, in seconds. When this number of seconds passes without an additional advertisement, control points should assume that the device has become unavailable. The NT[1] (Notification Type) header contains a single Uniform Resource Identifier that identifies the entity being advertised. The NTS[1] (Notification Sub Type) header always contains "ssdp:alive" for advertisement messages. Finally, USN[1] (Unique Service Name) header corresponds to the NT header, providing a unique name of a device or service to match the type in the NT header.

An example of a “ssdp:byebye” message, which is used to announce the demise of the root projector device is illustrated below:

NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age=1800
LOCATION: http://137.30.123.107:5431/ProjectorDevDesc.xml
NT: upnp:rootdevice
NTS: ssdp:byebye
USN: uuid:Upnp-Projector-1_0-1234567890001::upnp:rootdevice

UPnP devices respond to control points explicitly searching for services. To search for the service control points send “ssdp:discover” message. This message forces the appropriate device to respond with the location of their description document. These protocols fall under the Simple Service Discovery Protocol (SSDP) portion of the UPnP specification.

An example of a “ssdp:discover” message, which is used to discover the projector device(s) is illustrated below.

M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: “ssdp:discover”
MX: 10
ST: urn:schemas:upnp-org:device:projector:1

The MX[1] field specifies the number of seconds the control point is willing to wait for a response. The ST header defines the scope of the search. The value “urn:schemas:upnp-org:device:projector:1” specifies the device type being searched.

3.3.2 Description

The information provided to a control point during the discovery is very minimal. The control point only learns that the specific device exists but knows very little about it. Description in

UPnP allows control points to obtain additional information about the device. This information is contained in the XML description document for the root device, whose URL is discovered by the control points at the time of discovery phase.

To obtain a device's description document, a control point issues a standard HTTP GET, using TCP over IP. The format of the message that is used in the projector system is shown below.

GET ./web/ProjectorDevDesc.xml HTTP/1.1
HOST: 137.30.123.107:5431
ACCEPT-LANGUAGE: text/xml
<<BLANK LINE>>

After "GET", the path name component of the device's description document is specified. The host and port components are specified in the "HOST" component. The device's is due within 30 seconds. The response contains the XML description document in its body.

3.3.3 Control

Once a control point has obtained a device's description document, it will typically want to interact with the device, both to issue commands and to investigate an interesting device state. Device interaction in UPnP is handled with Remote Procedure Call protocol called the Simple Object Access Protocol. SOAP is based on extensions to HTTP, with actions specified in XML.

The SCPD for a service describes the actions that may be executed to interact with the service. To cause a device to execute an action, control point sends a "POST" or "M-POST" message. Control point may also request the value of specific state variables associated with a service using a "POST" message.

An example of the POST message for the projector system is illustrated below.

```
POST /upnp/control/power1 HTTP/1.0
CONTENT-TYPE: text/xml
SOAPACTION: “urn:schemas-upnp-org:service:PowerSwitch:1#PowerOn”
CNOTENT-LENGTH: 221
HOST: 137.30.123.107:5431

<s:Envelope xmlns:s=”http://schemas.xmlsoap.org/soap/envelope/”
s:encodingStyle=”http://schemas.xmlsoap.org/soap/encoding”>
<s:Body><u:SetSlideRange
        xmlns:u=”urn:schemas-upnp-org:service:SlideControl:1”/>
        <SlideRange>21</SlideRange>
        </u:SetSlideRange>
</s:Body>
</s:Envelope>
```

In the POST header, the path name component of the control URL for the associated service is passed. The HOST[1] header supplies the IP address and port of the control URL. CONTENT-LENGTH[1] tracks the number of bytes in the in the body of HTTP message. CONTENT-TYPE[1] is always fixed, specifying that XML and utf-8 character encoding used in the body. The SOAPACTION[1] header contains a fixed prefix (“urn:schemas-upnp-org:service:”) and then *serviceType* and *actionName* components. The serviceType defines the type of service to which this action is addressed. This must match the serviceType appearing in the description document for the root device and the actionName must match an action in the SCPD[1] of the service.

The body of the message contains XML, which is enclosed in the tags <s:Envelope> and </s:Envelope>. Two initial lines define the SOAP envelope schema and SOAP encoding schema. In the <s:Body>, the name of the action and the service type are provided. These should match the values provided in the message header. Subsequently, <argumentName>, </argumentName> pairs enclose the value of the “in” parameter to the action.

3.3.4 Eventing

The eventing mechanism in UPnP allows control points to receive asynchronous notifications about interesting state changes in UPnP services. During control operations, control points explicitly issue commands to change the state of UPnP devices and to query the values of state variables. Eventing adds the ability to subscribe to a service and to learn of changes in the values of state variables as they occur.

To subscribe to receive event messages, a control point sends a SUBSCRIBE[1] messages. To cancel a subscription, either a control point may let the subscription duration pass without issuing a request for renewal, or it may explicitly cancel the subscription using an UNSUBSCRIBE[1] message. Requests to subscribe and unsubscribe to the service's event notifications are made to the event subscription URL in the device's description document. All subscriptions are leased, and they must be renewed periodically or they will expire. The protocol that supports subscribing, unsubscribing and notifications in UPnP is called the General Event Notification Architecture. GENA is an HTTP based protocol whose messages are sent over TCP.

An example of the SUBSCRIBE message in the projector system is illustrated below.

SUBSCRIBE /upnp/event/power1 HTTP/1.1
HOST: 137.30.123.107:5431
CALLBACK: http://137.30.123.107:5432
NT: upnp:event
TIMEOUT: Second-1800

The SUBSCRIBE message header follows the event subscription URL for the associated service. CALLBACK provides URL on the control point where the service can send event messages. The TIMEOUT[1] is the requested duration in seconds. The response to the SUBSCRIPTION request is due within 30 seconds and it contains the SID[1] (Subscription ID) header. The UUID[1] provided

in the SID header contains a unique subscription ID associated with this subscription, which will be used by control points to cancel or renew this subscription.

The format of the renew message is given below. It is similar to the SUBSCRIBE message except the CALLBACK header is replaced by the SID header, which contains the subscription UUID.

SUBSCRIBE publisherpath **HTTP/1.1**
HOST: publisherhost:publisherport
SID: uuid:subscription UUID
TIMEOUT: Second- requested subscription duration in seconds
<<BLANK LINE>>

The publisherpath corresponds to the event subscription URL (/upnp/event/power1), which is obtained from the description document. The publisherhost and publisherport portion of the message contain the host name and port components of the event subscription URL (137.30.123.107 and 5431). The UUID contains the unique subscription ID.

The format of UNSUBSCRIBE message is given below. It is similar to the renew message except it does not contain the TIMEOUT header.

SUBSCRIBE publisherpath **HTTP/1.1**
HOST: publisherhost:publisherport
SID: uuid:subscription UUID
<<BLANK LINE>>

Once a control point has subscribed, UPnP services are responsible for sending NOTIFY event messages to the control point whenever state variables change. The format of the NOTIFY message is given below:

NOTIFY deliverypath **HTTP/1.1**
HOST: deliveryhost:deliveryport
CONTENT_TYPE: text/xml
CONTENT-LENGTH: number of bytes in the body
NT: upnp:event
NTS: upnp:propchange
SID: uuid:subscription- UUID
SEQ: event identifier
<e:propertyset xmlns:e="urn:schemas-upnp-org:event-1-0">
<e:property>
<variableName> new value </variableName>
</e:property>

<e:property>
<variableName> new value </variableName>
</e:property>
<e:propertyset>

The deliverypath component of the NOTIFY header us the path name component of the delivery URL provided when this control point subscribed. The deliveryhost and deliveryport components of the HOST header are the host name and port on which the control point is listening for event messages. These correspond to the values specified in the CALLBACK header during the control point subscription (137.30.123.107:5432). SEQ provides the event number corresponding to this event-each event is tagged with a value higher than the previous, with the initial event message tagged with 0. Each control value whose new value is being reported is enclosed in the <e:property> </e:property> pair. The variable name corresponds to the name of the control variable (e.g. Power), and the value is the new value for the variable (e.g. true).

3.3.5 Presentation

Client can request the HTML UI that is provided by the projector device and use it to control the device. In our project we use our own GUI designed in Java. As the control point is programmed in C and GUI in Java we use JNI to communicate between them.

3.4 Working Scenario

A service discovery world in our project is depicted in Figure 3.6. At time (0), the projector device and the control point talk to the DHCP[12] server to obtain an IP address. After the initialization, at time (1), advertisement messages are multicast that provide information about the device, any embedded devices and the services offered. At the same time control points multicast discovery messages to search for the device of interest. After the device of interest is discovered, the control point gets the detailed description about the device at time (2). After getting the description, at time (3), the control point issues control actions to interact with the device. As a result of issuing control actions, the state of device is modified. This is notified to the control point by sending event messages at time (4).

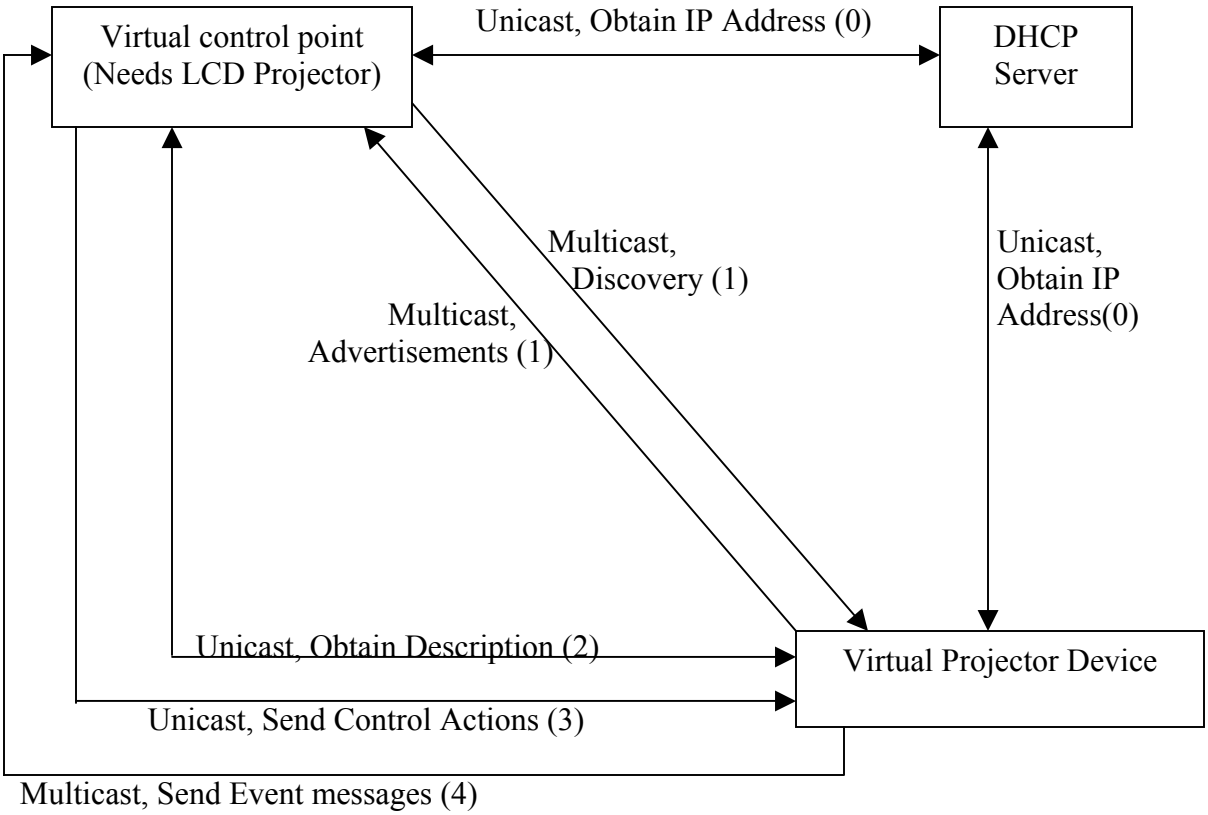


Fig 3.6 Working Scenario

CHAPTER 4. IMPLEMENTATION

4.1 The API Selections

As discussed previously, the main goal of this project was to design and implement a wireless application for showing a presentation using a LCD projector. The system framework is based on UPnP. The system achieves Automatic Discovery, Plug and Play and Zero Configuration. Once the control point is initialized, the user can determine the available projectors over the wireless network. He can choose one of them and start the slide show. If there is more than one projector present in the network, the user can switch between the projectors and use them interchangeably.

According to the system requirements and the study of system architecture, we decided to use the APIs listed below:

- JDK1.3[14]. We needed to use JNI for communication between GUI and the virtual control point. And JDK1.3 has the support for JNI.
- Intel UPnP SDK v1.0.3[4] Starter kit provided by UPnP forum.

4.2 Implementation Overview

Fig 4.1 shows the pictorial representation of the overview of implementation. The client side consists of a GUI developed in Java and the client application called ‘Control Point’. The GUI accepts all the commands from the user with relevant input parameters. Based on which function the user wants to perform, the functions from the ‘JavaCBridge’ class are called. ‘JavaCBridge’ is an interface implemented in Java, which calls JNI functions corresponding to the user request from the control point. The control point implements three native functions so as to communicate with the GUI. The native functions are:

- ‘doInitialization(ip, port)’
- ‘sendCommand(cmd, input)’
- ‘retrieveCommandResult()’

‘doInitialization’ is called when the user chooses to search for LCD projector device. It will in turn issue a discover message over the network and look for devices of type ‘Projector’. The ‘ip’ and ‘port’ are the IP address and port on which the client is running. They are used to register the control point with the UPnP environment.

‘sendCommand(cmd, input)’ is called when the user chooses to perform the control actions provided by the services. The ‘cmd’ contains the action specified by the user and the ‘input’ parameter contains the relevant input data.

‘retrieveCommandResult()’ is called after every user action that the GUI sends to the control point.

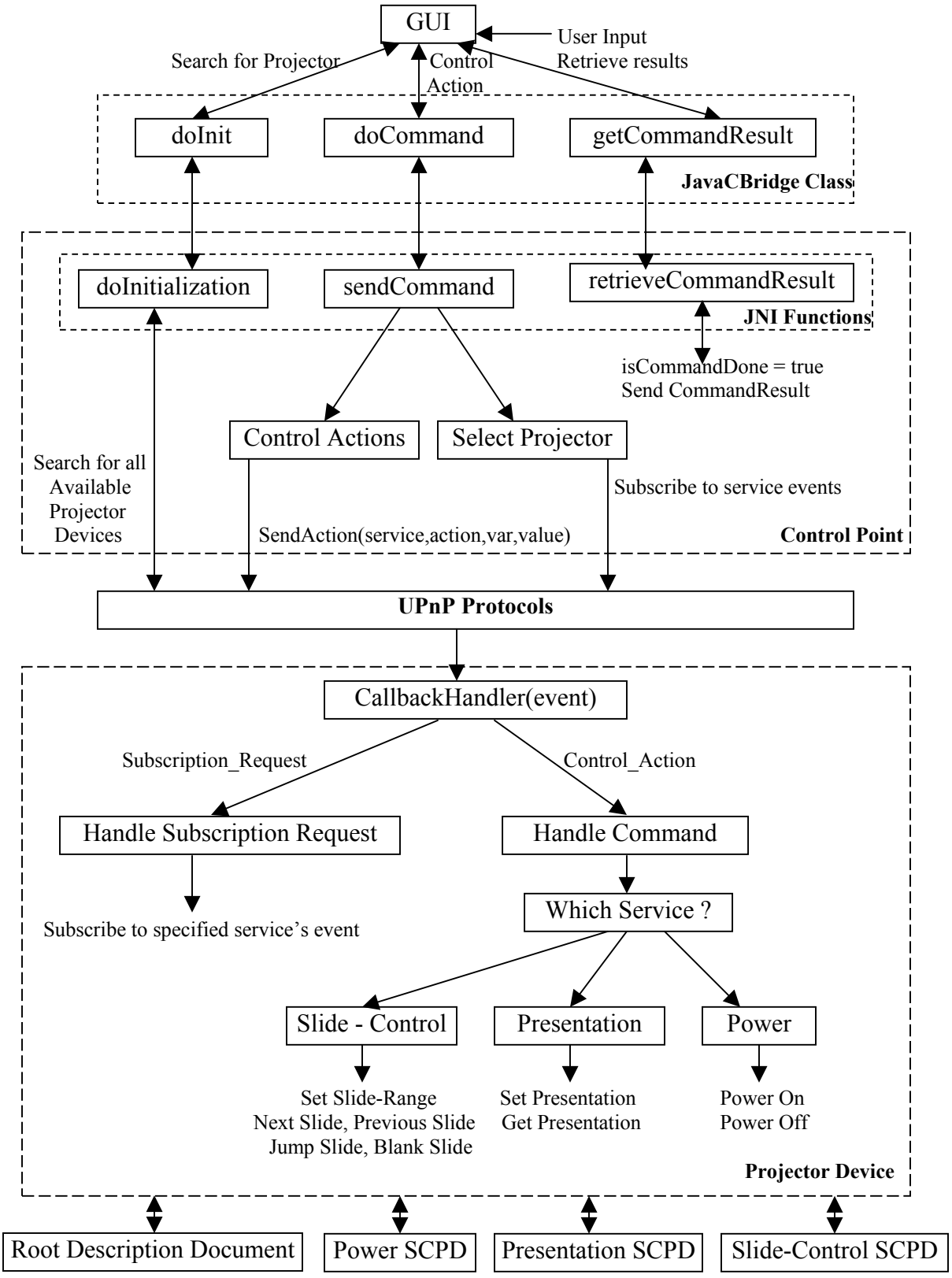


Fig 4.1 Implementation Overview

The control actions requested by the user can be of two types.

- Control action provided by the services
- Select the Projector

All control actions are handled calling the 'SendAction()' function which takes the following input parameters:

- Service, to which the control action is addressed
- Action, the actual action. For e.g. 'PowerOn'
- Variable, the name of the state variable that is going to be affected
- Value, the new value that the state variable will contain

If the user wants to select a particular projector, the device state variable maintained in the control point code is updated with the Unique Device Name (UDN) of the projector. And the subscription requests are send to subscribe to all the services. A failure to subscribe with any of the services results in the device not being selected.

For every command issued by the control point, the UPnP environment generates an event, which categorizes the command into two types.

- 'Subscription_Request'
- 'Control_Action'

This differentiation is done at the 'projector device' side of the application. When the projector device becomes alive it registers itself with the UPnP environment specifying the IP address and port it is running on, the name of the description document, the directory containing

the description document and the name of the function that should be called if any control action is addressed to this device. The name of this function is 'CallbackHandler(event)'. The 'event' specifies the event generated by the UPnP environment upon receiving the action from the control point. If it is a 'Subscription_Request' it extracts the name of the service for which this subscription request is addressed and processes the request. If it is a 'Control_Action', it extracts which service this action is addressed to and accordingly processes the request.

On the 'Projector Device' side, a 'device state' is managed which keeps track of whether the projector is on or off, of the name of the presentation, of the number of slides in the presentation and of the identifier of the currently viewed slide. Maintaining the device state is absolutely necessary to perform all the functions. The XML documents that are maintained to track all the above parameters are called description documents. These description documents also list all services the projector device offers. The projector device provides the power service, the presentation service and the slide-control service. For each of these services, we define a Service Control Protocol Description (SCPD) document, which lists all actions that the user can perform with the service. Each of the service documents contains state variables, which contribute to the state of the device. In our application, a power service contributes to the device state by keeping track of whether the projector is on or off. A presentation service contributes to the device state by keeping track of the name of the selected presentation. Finally, the slide-control service contributes to the device state by keeping track of the total number of slides in the presentation and the identifier of the currently viewed slide.

4.2 Implementations of the Description Documents

There are in total four description documents defined, which are divided into two categories: One is the root device description document and the three others are called Service Control Protocol Description documents.

4.2.1 Device Description Document

The device description document describes the root device, all its embedded devices and the services provided. It defines a set of URLs, which are used by the control point to learn more about the device. The presentation URL allows control points to download the HTML-based GUI for the device. The Control URL of the service serves as entry point for all user commands. The Event subscription URL is used for subscribing to the service's eventing. The relative URL is defined which points to the Service Control Protocol Description, an XML document that describes the actions and control variables associated with that service.

```
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <URLBase>http://137.30.123.22:5431</URLBase>
  <device>
    Descriptive attributes for 'Projector device'. Important  ones are 'deviceType' and 'UDN'
    <deviceType>urn:schemas-upnp-org:device:projector:1</deviceType>
    <friendlyName>UPnP LCD Projector</friendlyName>
     <manufacturer>University of New Orleans Dept. of Computer
                  Science</manufacturer>
    <manufacturerURL>http://www.cs.uno.edu/</manufacturerURL>
     <modelDescription>UPnP LCD Projector to display Power-Point
                       Presentations</modelDescription>
    <modelName>Plug-N-Project</modelName>
    <modelNumber>1.0</modelNumber>
    <modelURL>http://www.cs.uno.edu/</modelURL>
    <serialNumber>9876543210</serialNumber>
    <UDN>uuid:Upnp-Projector-1_0-1234567890001</UDN>
    <UPC>123456789</UPC>
    <serviceList>
```

```
      ‘Power Service’ with its ‘Service Type’ and all the required URLs
      <service>
        <serviceType>urn:schemas-upnp-org:service:PowerSwitch:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:PowerSwitch1</serviceId>
        <controlURL>/upnp/control/power1</controlURL>
        <eventSubURL>/upnp/event/power1</eventSubURL>
        <SCPDURL>/ProjectorPowerSCPD.xml</SCPDURL>
      </service>
      ‘Presentation Service’ with its ‘Service Type’ and all the required URLs
      <service>
        <serviceType>urn:schemas-upnp-org:service:Present:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:Present1</serviceId>
        <controlURL>/upnp/control/Presentation1</controlURL>
        <eventSubURL>/upnp/event/Presentation1</eventSubURL>
        <SCPDURL>/ProjectorPresSCPD.xml</SCPDURL>
      </service>
      ‘Slide-Control Service’ with its ‘Service Type’ and all the required URLs
      <service>
        <serviceType>urn:schemas-upnp-org:service:SlideControl:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:SlideControl1</serviceId>
        <controlURL>/upnp/control/SlideControl1</controlURL>
        <eventSubURL>/upnp/event/SlideControl1</eventSubURL>
        <SCPDURL>/ProjectorSlideCtrlSCPD.xml</SCPDURL>
      </service>
    </serviceList>
   <presentationURL>/ProjectorDevicePres.html</presentationURL>
</device>
</root>
```

Note: The BOLD lines in between the scripts are the comments and not part of the script or code. Similar commenting style is used in all further code lists.

Code List 4.1: ProjectorDevDesc.xml; Root Device Description Document

4.2.2 Power Service Description Document

The SCPD for the power service lists the various control actions this service offers. It also contains the control variables, which keep track of the device state. There are two actions provided by the power service, ‘PowerOn’ and ‘PowerOff’, and there is one state variable named ‘Power’ for which the eventing is enabled by setting ‘sendEvents=”yes”’ in the stateVariable tag. Whenever there is a change in the control variable, ‘Power’, it will be notified to the control points who are subscribed to receive the events from this service will be notified.

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
```

```
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>

  <actionList>
    Sets the 'Power' state variable to True
    <action>
      <name>PowerOn</name>
    </action>
    Sets the 'Power' state variable to False
    <action>
      <name>PowerOff</name>
    </action>
  </actionList>

  <serviceStateTable>
    Control variable 'Power' for which eventing is made on
    <stateVariable sendEvents="yes">
      <name>Power</name>
      <dataType>boolean</dataType>
      <defaultValue>false</defaultValue>
    </stateVariable>
  </serviceStateTable>
</scpd>
```

Code List 4.2: ProjectorPowerSCPD.xml, Power Service Description Document

4.2.3 Presentation Service Description Document

The SCPD for the presentation service lists the various control actions this service offers. It also contains the control variables, which keep track of the device state. There are two actions provided by the presentation service, 'SetPresentation' and 'GetPesentation', and there is one state variable named 'Presentation' for which the eventing is enabled by setting 'sendEvents="yes"' in the stateVariable tag. An action may have a variable associated with it. If it has one, relatedStateVariable tag is used to specify which state variable is associated with this action. It also specifies the direction of the variable, whether it is a 'in' or 'out' variable.

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
```

```
  </specVersion>

  <actionList>
    Sets the 'Presentation' state variable to the value provided by the user
    <action>
      <name>SetPresentation</name>
      <argumentList>
        <argument>
        <name>P</name>
          <relatedStateVariable>Presentation</relatedStateVariable>
          <direction>in</direction>
        </argument>
      </argumentList>
    </action>
    Retrieves the value in 'Presentation' state variable
    <action>
      <name>GetPresentation</name>
      <argumentList>
        <argument>
        <name>P</name>
          <relatedStateVariable>Presentation</relatedStateVariable>
          <direction>out</direction>
        </argument>
      </argumentList>
    </action>
  </actionList>

  <serviceStateTable>
    Control variable 'Presentation' for which eventing is made on
    <stateVariable sendEvents="yes">
      <name>Presentation</name>
      <dataType>string</dataType>
      <defaultValue>""</defaultValue>
    </stateVariable>
  </serviceStateTable>
</scpd>
```

Code List 4.3: ProjectorPresSCPD.xml; Presentation Service Description Document

4.2.4 Slide Control Service Description Document

The SCPD for the slide-control service lists the various control actions this service offers. It also contains the control variables, which keep track of the device state. There are seven actions provided by the power service, 'SetSlideRange', 'GetSlideRange', 'GetSlide', 'NextSlide', 'PrevSlide', 'JumptoSlide', and 'DisplayBlank'. There are two state variables named 'CurrentSlide' and 'SlideRange', for which the eventing is enabled by setting

‘sendEvents=”yes”’ in the stateVariable tag. The state variables in the slide-control service also specify their minimum and maximum values as well as their default value.

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>

  <actionList>
    Sets the ‘SlideRange’ state variable to the value specified by the user
    <action>
      <name>SetSlideRange</name>
       <argumentList>
        <argument>
        <name>S</name>
          <relatedStateVariable>SlideRange</relatedStateVariable>
          <direction>in</direction>
        </argument>
      </argumentList>
    </action>
    Retrieves the value in the ‘SlideRange’ state variable
    <action>
     <name>GetSlideRange</name>
       <argumentList>
        <argument>
        <name>S</name>
          <relatedStateVariable>SlideRange</relatedStateVariable>
          <direction>out</direction>
        </argument>
      </argumentList>
    </action>
    Retrieves the value in the ‘CurrentSlide’ state variable
    <action>
     <name>GetSlide</name>
      <argumentList>
        <argument>
        <name>S</name>
          <relatedStateVariable>CurrentSlide</relatedStateVariable>
          <direction>out</direction>
        </argument>
      </argumentList>
    </action>
    Increments ‘CurrentSlide’ state variable by one
    <action>
      <name>NextSlide</name>
    </action>
    Decrements ‘CurrentSlide’ state variable by one
    <action>
      <name>PrevSlide</name>
    </action>
    Updates ‘CurrentSlide’ state variable with the value specified by the user
    <action>
```

```
      <name>JumptoSlide</name>
       <argumentList>
        <argument>
        <name>S</name>
          <relatedStateVariable>CurrentSlide</relatedStateVariable>
          <direction>in</direction>
        </argument>
      </argumentList>
    </action>
    Displays blank slide, 'CurrentSlide' variable initialized to one
    <action>
        <name>DisplayBlank</name>
    </action>
  </actionList>

  <serviceStateTable>
    Control Variable 'CurrentSlide' for which eventing is made on
    <stateVariable sendEvents="yes">
      <name>CurrentSlide</name>
       <dataType>i1</dataType>
        <allowedValueRange>
          <minimum>1</minimum>
          <maximum>500</maximum>
          <step>1</step>
        </allowedValueRange>
      <defaultValue>1</defaultValue>
    </stateVariable>
    Control variable 'SlideRange' for which eventing is made on
    <stateVariable sendEvents="yes">
      <name>SlideRange</name>
       <dataType>il</dataType>
        <allowedValueRange>
          <minimum>1</minimum>
          <maximum>500</maximum>
        </allowedValueRange>
    </stateVariable>
  </serviceStateTable>
</scpd>
```

Code List 4.4: ProjectorSlideCtrlSCPD.xml; Slide-Control Service Description Document

4.3 Implementations of Device and Control Point

In this section we explain all important code snippets of our control point and the projector device files. Towards the end, we also provide a few important supporting functions contained in the files 'common.c' and 'JavaCBridge.java'. The latter serves as an interface for the GUI to call the native functions defined in 'nativelib.c', which is the control point implementation.

4.3.1 Virtual Projector Device

This is the implementation of the virtual projector device. This code runs on a machine that connects physically to the LCD projector. This code registers the projector device and provides various functions to handle the subscription requests and requests to operate the various services provided by the projector device such as power, slide control and presentation.

Defining constants

```
#define AD_DURATION     1800  // duration between device advertisements
#define POWER_INDEX      0    // index into variables/values arrays for power
#define SLIDE_INDEX      1    // index into variables/values arrays for slide
#define PRESENT_INDEX    2    // index into variables/values arrays for
                              // presentation
```

Projector device type

```
char PROJECTOR_TYPE[] = "urn:schemas-upnp-org:device:projector:1";
```

Projector service types

```
char POWERSERVICE[] ="urn:schemas-upnp-org:service:PowerSwitch:1";
char SLIDESERVICE[] ="urn:schemas-upnp-org:service:SlideControl:1";
char PRESENTSERVICE[] ="urn:schemas-upnp-org:service:Present:1";
```

Unique ids for services

```
char POWERSERVICE_NAME[] ="urn:upnp-org:serviceId:PowerSwitch1";
char SLIDESERVICE_NAME[] ="urn:upnp-org:serviceId:SlideControl1";
char PRESENTSERVICE_NAME[] ="urn:upnp-org:serviceId:Present1";
```

Projector type

```
typedef struct projector_state {
  char UDN[NAME_SIZE];     // unique name for projector
  char ***variables;       // array of state variable names for each service
  char ***values;          // array of corresponding variable values
} ProjectorState;
```

A semaphore used to lock the global state before touching it!

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;  // this is that semaphore

UpnpDevice_Handle handle=-1;            // projector's device handle
ProjectorState state;                   // state table for projector
```

The main() function takes four arguments IP address, port, name of projector's description document and directory where description document and SCPDs for projector services are stored. It initializes and register device, start advertisements, and deal with shutdown via control-C.

```
int main(int argc, char** argv) {

  int port;               // port on which we listen for messages
  int code;               // return value from UPnP functions
  char *ip=NULL;          // IP address for this device
  char *desc_doc=NULL;    // location of description document
```

```
  char *web_dir=NULL;     // root path for web server
  char doc_url[NAME_SIZE]; // absolute URL for our description document
     // pthread_t blend_thread;

  ip=argv[1];
  sscanf(argv[2],"%d",&port);
  desc_doc=argv[3];
  web_dir=argv[4];
```

Absolute description document URL is created. This URL will be relative to the root directory specified for web server.

```
  sprintf(doc_url, "http://%s:%d/%s", ip, port, desc_doc);
```

UpnpInit() must be called before any other UPnP functions are used. It initializes the UPnP library. Diagnose() function from common.c examines return code from a UPnP function and outputs a human-readable description of error condition.

```
  code = UpnpInit(ip, port)
```

UpnpSetWebServerRootDir() initializes the web server provided in the Intel SDK, supplying it with the root directory where description documents are stored.

```
  code = UpnpSetWebServerRootDir(web_dir)
```

UpnpRegisterRootDevice() accepts URL describing the root device's description document, a callback function that will be called when interesting things happen. All the interactions with control points are initiated by SDK through this function. Third parameter provides a cookie that will be passed back when callback function is called. Forth parameter is address of a UpnpDevice_Handle. This handle is our key to all future interactions with UPnP library. Callback function directly or indirectly calls all the other functions.

```
  code = UpnpRegisterRootDevice(doc_url, CallbackHandler, &handle, &handle)
```

This function initializes all the state variables of the projector services.

```
  InitializeStateTable(doc_url);
```

UpnpSendAdvertisement() sends advertisements for the root devices and all the services.

```
  code = UpnpSendAdvertisement(handle, AD_DURATION)
```

Initialization is almost complete. This keeps the Projector to run in the loop waiting for further actions from the control point. The signal handler function ensures proper functions are called before shutting down the projector. This function handles control-C—it is the only way to shut down the projector.

```
  code = pthread_create(&projector_thread, NULL, ProjectLoop, NULL);

  SignalHandler();
  return 0;
}
```

Initialize the projector's state table, which contains bits of data that clients are interested in. Default values for variables, etc. are hard-coded--this means that the projector implementation will

need tweaking if these values are modified in the description document. Could parse the description document to obtain default values, etc. to increase abstraction. We do parse the description document to obtain the unique device name of this projector.

```
void InitializeStateTable(char *doc_url) {

  Upnp_Document doc=NULL;    // our description document
  int code;                  // return value from UPnP functions
  char *udn=NULL;            // unique projector name
  int i,j;


  // lock device state
  pthread_mutex_lock(&mutex);

   // The structure ProjectorState is initialized. The name and default
   // values for state variables and are hard-coded.

  strcpy(state.variables[POWER_INDEX][0], "Power");
  strcpy(state.values[POWER_INDEX][0], "false");
  strcpy(state.variables[SLIDE_INDEX][0], "CurrentSlide");
  strcpy(state.values[SLIDE_INDEX][0], "1");
  strcpy(state.variables[SLIDE_INDEX][1], "SlideRange");
  strcpy(state.values[SLIDE_INDEX][1], "1");
  strcpy(state.variables[PRESENT_INDEX][0], "CurrentPresentation");
  strcpy(state.values[PRESENT_INDEX][0], "EMPTY");

   // UpnpDownloadXmlDoc() function downloads the description document and
   // extracts the unique device name for the projector. ParseItem() function
   // from common.c is used to extract the UDN from the document.

  code = UpnpDownloadXmlDoc(doc_url, &doc);
  udn = ParseItem(doc, NULL, "UDN");
  strcpy(state.UDN, udn);

   // Always use appropriate UpnpXXX_free() functions to free the memory
   // instead of just free() function. At the end the lock on the device
   // state is released.

  if (doc) {
    UpnpDocument_free(doc);
  }

  pthread_mutex_unlock(&mutex);

}
```

All asynchronous notifications of interesting events come through this handler. This function is the reason, why a semaphore has to be used to lock global device state before accessing it! For each event type, hand off processing to an appropriate projector function.
type = type of event that occurred
event = the event description
cookie = data we specified during registration of the callback

```
int CallbackHandler(Upnp_EventType type, void *event, void *cookie) {
```

Prints bunch of information about the event which is useful for debugging. Reportevent() function is in the common.c.

```
  ReportEvent(type,event,true);

  switch (type) {

  // A control point is subscribing to know about interesting state changes
  case UPNP_EVENT_SUBSCRIPTION_REQUEST:
    HandleSubscriptionRequest((struct Upnp_Subscription_Request *)event);
    break;

  // A control point is interested in knowing the current value of a state
  // variable
  case UPNP_CONTROL_GET_VAR_REQUEST:
    HandleVariableRequest((struct Upnp_State_Var_Request *)event);
    break;

  // A control point is issuing a command to Projector
  case UPNP_CONTROL_ACTION_REQUEST:
    HandleCommand((struct Upnp_Action_Request *)event);
    break;

  default:
    printf("Projector: Do not know why I received that event!!\n");
    break;
  }

  return 0;
}
```

This function is called from CallbackHandler() as a result of the control point attempting to subscribe to a service's eventing. A Upnp_Subscription_Request structure, which is passed as a parameter contains 3 fields, a ServiceId, UDN, and Subscription ID.

```
void HandleSubscriptionRequest(struct Upnp_Subscription_Request *event) {

  // lock device state
  pthread_mutex_lock(&mutex);

   // The first step is to see if the subscription request is directed to us?
   // If it is, we find out service the control point wish to subscribe to.

    if (! strcmp(POWERSERVICE_NAME, event->ServiceId)) {
      printf("Projector: Subscription is for power service.\n");
      index = POWER_INDEX;
    }
    else if (! strcmp(SLIDESERVICE_NAME, event->ServiceId)) {
      printf("Projector: Subscription is for slide control service.\n");
      index = SLIDE_INDEX;
    }
    else if (! strcmp(PRESENTSERVICE_NAME, event->ServiceId)) {
      printf("Projector: Subscription is for projector contents service.\n");
```

```
      index = PRESENT_INDEX;
    }

    // Accept subscription for the specified service. The following accepts
    // the subscription and provides the service's state variable names and
    // current values.  The sixth argument specifies the number of variables;
    // all of the projector's services have a single state variable, except
    // for the slide service, which has two.

    code = UpnpAcceptSubscription(handle,
    event->UDN,
    event->ServiceId,
    (const char **)state.variables[index],
    (const char **)state.values[index],
    (index == SLIDE_INDEX ? 2 : 1),
    event->Sid);
  }

  // unlock
  pthread_mutex_unlock(&mutex);
}
```

A control point needs the value of a specific control variable. The Upnp_State_Var_Request event has number of components. ErrCode, which provides status code for request. ErrStr, set by the device in case of error. Unique device name; DevUDN, ServiceID, StateVarName and CurrentVal, that is used to return the value of the required variable.

```
void HandleVariableRequest(struct Upnp_State_Var_Request *event) {

  // lock device state
  pthread_mutex_lock(&mutex);

  if (! strcmp(event->DevUDN, state.UDN)) {

      // Could look at service ID in the event, but since all our state
      // variables have distinct names, can just examine variable names

      event->ErrCode = UPNP_E_SUCCESS;
      event->CurrentVal = (Upnp_DOMString)
            malloc(strlen(state.values[service][variable])+1);
      strcpy(event->CurrentVal, state.values[service][variable]);
   }
  // unlock
  pthread_mutex_unlock(&mutex);

}
```

This functions handles actions attempted by control points. We determine which service the action is directed at and then hand off the processing to the appropriate function.

```
void HandleCommand(struct Upnp_Action_Request *event) {
  // lock device state
  pthread_mutex_lock(&mutex);
```

```
  if (! strcmp(event->ServiceID, POWERSERVICE_NAME)) {
      HandlePowerChanges(event);
  }
  else if (! strcmp(event->ServiceID, SLIDESERVICE_NAME)) {
      HandleSlideChanges(event);
  }
  else if (! strcmp(event->ServiceID, PRESENTSERVICE_NAME)) {
      HandlePresentChanges(event);
  }

  // unlock
  pthread_mutex_unlock(&mutex);

}
```

This function is called when the client tries to use the power service provided by the projector device. It allows to power on / off the selected projector.

```
void HandlePowerChanges(struct Upnp_Action_Request *event) {
```

Determine which power service function is invoked

```
  if (! strcmp(event->ActionName, "PowerOn")) {
    printf("Projector: PowerOn.\n");
    strcpy(state.values[POWER_INDEX][0], "true");
  }
  else if (! strcmp(event->ActionName, "PowerOff")) {
    printf("Projector: PowerOff.\n");
    strcpy(state.values[POWER_INDEX][0], "false");
  }
```

construct response—we are responsible for putting some info into the event structure
The format of the XML response looks like this:

<s:Envelope
xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

<s:Body>
<u:actionNameResponse xmlns:u="urn:schemas-upnp-org:service:serviceType:v">
<argumentName>out arg value</argumentName>
other out args and their values go here, if any
</u:actionNameResponse>
</s:Body>
</s:Envelope>

We are only responsible for dealing with the portion between the <s:Body> and </s:Body>. The UPnP library will add the rest. If there are no out parameters, the <argumentName> ... </argumentName> stuff can be omitted. None of the power-related actions have out parameters.

```
    sprintf(response, "<u:%sResponse xmlns:u=\"%s\"> </u:%sResponse>",
    event->ActionName, POWERSERVICE, event->ActionName);
    event->ActionResult = UpnpParse_Buffer(response);
```

Inform the control points of changes to power settings

```
  code = UpnpNotify(handle, state.UDN, POWERSERVICE_NAME,
    (const char **)state.variables[POWER_INDEX],
    (const char **)state.values[POWER_INDEX],
    1);
}
```

This function is called when the client tries to use the slide-control service provided by the device. This function handles various commands like 'show the next slide', 'go to previous slide', 'jump to a particular slide', 'display a blank slide' etc. When command is next slide, the current value of the slide in the state variables is incremented by one, when the command is previous slide, its decremented by one. In case of jump to a particular slide, slide number is accepted from the user and the current value of the slide in the state variable is set as the value provided by the user. When displaying the blank slide, a blank slide is displayed and the slide variable is reset to the first slide.

```
void HandleSlideChanges(struct Upnp_Action_Request *event) {

  // ------ Show Next Slide ------
  if (! strcmp(event->ActionName, "NextSlide")) {
    slide++;
    sprintf(state.values[SLIDE_INDEX][0], "%1d", slide);
  }

  // ------ Show Previous Slide ------
  else if (! strcmp(event->ActionName, "PrevSlide")) {
    slide--;
    sprintf(state.values[SLIDE_INDEX][0], "%1d", slide);
  }

  //------ Jump To a Slide -----
  else if (! strcmp(event->ActionName, "JumpToSlide")) {
    value = ParseItem(event->ActionRequest, NULL, "CurrentSlide");
    strcpy(state.values[SLIDE_INDEX][0],value);
  }

  //------ Display Blank Slide ------
  else if(! strcmp(event->ActionName, "DisplayBlank")) {
    slide = 0;
    sprintf(state.values[SLIDE_INDEX][0], "%1d", slide);
  }

  //-------- Set Slide Range -----
  else if (! strcmp(event->ActionName, "SetSlideRange")) {
    value = ParseItem(event->ActionRequest, NULL, "SlideRange");
    strcpy(state.values[SLIDE_INDEX][1], value);
  }

  //------ Get Slide Range ------
  else if (! strcmp(event->ActionName, "GetSlideRange")) {
    // create "inner" portion of response.  See the detailed comment below
    // about the format of a response to understand this.
    value = ParseItem(event->ActionRequest, NULL, "SlideRange");
    slide = atoi(value);
    sprintf(inner, "<Slide>%1d</Slide>", slide);
```

```
  }

  // construct response—we are responsible for putting some info into the
  // event structure

    sprintf(response, "<u:%sResponse xmlns:u=\"%s\">%s</u:%sResponse>",
    event->ActionName, SLIDESERVICE, inner, event->ActionName);
    event->ActionResult = UpnpParse_Buffer(response);

    code = UpnpNotify(handle, state.UDN, SLIDESERVICE_NAME,
      (const char **)state.variables[SLIDE_INDEX],
      (const char **)state.values[SLIDE_INDEX],
      2);
}
```

This function is called when the client tries to use the presentation service provided by the projector device. This is used to set the presentation state variable in the SCPD to the one provided by the user.

```
void HandlePresentChanges(struct Upnp_Action_Request *event) {

  if (! strcmp(event->ActionName, "SetPresentation")) {
    value = ParseItem(event->ActionRequest, NULL, "Presentation");
    strcpy(state.values[PRESENT_INDEX][0], value);
  }

  else if (! strcmp(event->ActionName, "GetPresentation")) {
      value = ParseItem(event->ActionRequest, NULL, "Presentation");
      sprintf(inner, "<Contents>%s</Contents>",
      state.values[PRESENT_INDEX][0]);
  }

    sprintf(response, "<u:%sResponse xmlns:u=\"%s\">%s</u:%sResponse>",
    event->ActionName, PRESENTSERVICE, inner, event->ActionName);
    event->ActionResult = UpnpParse_Buffer(response);
    printf("Projector:  Response to action is\n\"%s\"\n", response);

    code = UpnpNotify(handle, state.UDN, PRESENTSERVICE_NAME,
      (const char **)state.variables[PRESENT_INDEX],
      (const char **)state.values[PRESENT_INDEX],
      1);
}
```

This function runs in a thread. It keeps the virtual projector running and help the projector to listen to the commands from client. Every five seconds, it checks to see if the projector is on / off.

```
void *ProjectLoop(void *args) {

 while (1) {
    printf("Projector: Project thread sleeping.\n");
    sleep (5);
    printf("Projector: Project thread awake.\n");

    // lock device state
```

```
    pthread_mutex_lock(&mutex);

    // is the projector on?
    if (! strcmp(state.values[POWER_INDEX][0], "true")) {
     printf("Projector is On !!!");
    }
    else {
      // nothing to do if projector is off
      printf("Projector: Projector is off.\n");
    }
    // unlock, then back to sleep
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}
```

Catches if user presses control-C and calls Shutdown() function.

```
void SignalHandler() {

  int sig;
  sigset_t sigs_to_catch; // signal handling stuff (for handling shutdown)
  sigemptyset(&sigs_to_catch);
  sigaddset(&sigs_to_catch, SIGINT);
  sigwait(&sigs_to_catch, &sig);
  Shutdown("Shutting down.");
}
```

Do a clean shutdown. Unregister the root device and then call UpnpFinish() before dying. Shutdown is also called at other points during the execution as a result of fatal errors.

```
void Shutdown(char *reason) {

  printf("Wait...\n");
  if (handle >= 0) {
      UpnpUnRegisterRootDevice(handle);
  }
  UpnpFinish();
  printf("%s\n", reason);
  exit(0);
}
```

Code List 4.5: Projector.c; Projector Device implementation

4.3.2 Virtual Control Point

A client side application. Helps the user to discover the available projector devices, select the one of interest and use all the services offered by the device. There are in total three JNI functions defined, which are called from the GUI. The first one used to initialize the control point and search for the available projector devices on the network. The second one is used to receive user commands from the GUI. And the GUI, to retrieve the results, as a result of the user actions, uses the third one.

Defining constants

```
#define REG_TIMEOUT     1800         // registration duration
#define true 1
#define false 0
```

Define to display info about ALL callbacks

```
#undef SHOW_EVENT_INFO
```

UPnP devices send lots of events that can be ignored--for example, advertisements are sent not only for the root device, but for all services, etc. If the following is defined, a "we are ignoring this" message is printed for such events, otherwise such events are silently ignored.

```
#undef  MENTION_IGNORED_EVENTS
```

Define projector device type we will search for…

```
char PROJECTOR_TYPE[] = "urn:schemas-upnp-org:device:projector:1";
```

Define types of projector device's services

```
char POWERSERVICE[] ="urn:schemas-upnp-org:service:PowerSwitch:1";
char SLIDESERVICE[] ="urn:schemas-upnp-org:service:SlideControl:1";
char PRESENTSERVICE[] ="urn:schemas-upnp-org:service:Present:1";
```

Maintains state of Projector services

```
typedef struct projector_service {
  char service_id[NAME_SIZE];           // id of service
  char service_type[NAME_SIZE];         // type of service
  char eventURL[NAME_SIZE];             // event subscription URL
  char controlURL[NAME_SIZE];           // control URL for service
  char SID[NAME_SIZE];                  // subscription ID
  int timeout;                          // actual timeout for registration
} ProjectorService;
```

Maintains the state of Projector device

```
typedef struct projector_device {
  char UDN[NAME_SIZE];              // unique name of device
  char descriptionURL[NAME_SIZE];   // URL for description document
  char friendly[NAME_SIZE];         // human-readable name of device
  char presentationURL[NAME_SIZE];  // URL for presentation document
  char presentationName[NAME_SIZE]; // Name of the presentation to view
  int  expiration;                  // timeout in device advertisement
  ProjectorService power;           // power switch for projector
  ProjectorService slide;           // slide control service
  ProjectorService present;         // presentation service
} ProjectorDevice;
```

Since callbacks are used to inform the projector of interested events (e.g., power on/off), a semaphore must be used to protect important state. The following semaphore should be used to lock global state before touching it!

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

UpnpDevice_Handle handle=-1;    // projector control point's device handle
Queue device_list;              // list of discovered projector devices
ProjectorDevice *current=NULL;  // pointer to current selected proj. device
int done=false;                 // controls death of command loop thread
int isCommandAvailable = false; // checks if command has arrived from GUI
int isCommandDone = false;      // checks if the command handling is done
char *CommandResult;  // Contains output of the command to be send to the GUI
char cmd_val=-1;
char *inputFromGUI;
```

The GUI calls this function at the very beginning. It initializes and registers the control point, start a thread to handle interactive commands, and deals with shutdown via control-C.

```
JNIEXPORT jint JNICALL Java_JavaCBridge_doInitialization
    (JNIEnv * env, jobject jobj, jstring ipadd, jstring ipport) {

    init_queue(&device_list,sizeof(ProjectorDevice),false,CompareProjectors);

     str1=(*env)->GetStringUTFChars(env,ipadd,&iscopy);
     strcpy(ip,str1); // IP address for this device

     str2= (*env)->GetStringUTFChars(env,ipport,&iscopy);
     sscanf(str2,"%d",&port);  //Port for the device

     // UpnpInit() must be called before any other UPnP functions are used
     code = UpnpInit(ip, port)

     // Register with UPnP
     code = UpnpRegisterClient(CallbackHandler, &handle, &handle)

     // start threads to handle timer events and interaction with user
     code = pthread_create(&timer_thread, NULL, TimerLoop, NULL);
     code = pthread_create(&cmdloop_thread, NULL, CommandLoop, NULL);

     SearchForProjectors();

     // handle control-C
     code = pthread_create(&sigHandler_thread,NULL,CallSignalHandler,NULL);

     isCommandDone = true;
     strcpy(CommandResult,"Initialization Done !");
     return 0;
}
```

This function is used to receive the user commands send via GUI.

```
JNIEXPORT jint JNICALL Java_JavaCBridge_sendCommand
      (JNIEnv * env, jobject jobj, jstring cmd,jstring cmd1) {

   const char *command=(*env)->GetStringUTFChars(env,cmd,&iscopy);
   const char *input=(*env)->GetStringUTFChars(env,cmd1,&iscopy);

   cmd_val = command[0];
   isCommandAvailable = true;

   strcpy(inputFromGUI,input);

   return 0;
}
```

The GUI calls this function to retrieve the results resulted from each user action. After each function is successfully completed the 'isCommandDone' is set to 'true'. And the result is copied into the 'CommandResult' buffer. This function is called immediately after the user action is issued. The application waits till the 'isCommandDone' becomes true and then sends the result to the user.

```
JNIEXPORT jstring JNICALL Java_JavaCBridge_retrieveCommandResult
    (JNIEnv * env, jobject jobj) {

   while(isCommandDone==false)
   {
   }

   isCommandDone = false;

   CommandResult = (*env)->NewStringUTF(env,CommandResult);
   return CommandResult;
}
```

Call the Signal Handler Function this function runs in a different thread

```
void *CallSignalHandler(void *args) {

   SignalHandler();

}
```

Handle commands from the user No arguments expected, no meaningful return value. This function is run in a separate thread. As soon as the command from the GUI becomes available, the 'isCommandAvailable' flag is turned on. This is how the functions know that the command from the GUI has arrived. Then it is further processed.

```
void *CommandLoop(void *args) {

  char cmd;
```

```
   while (! done) {
      cmd = cmd_val;
      isCommandAvailable = false;
      CallProjectorFuncntion(cmd);
   }
}
```

Search for projectors. We use the asynchronous search function UpnpSearchAsync(). Notifications for discovered projectors will come through our callback function CallbackHandler().

```
void SearchForProjectors() {

  // forget about projectors we already know about.
  ForgetAllProjectors();
```

We are looking for all root devices of type Projector and are willing to wait 10 seconds for a response. The last argument to UpnpSearchAsync() allows a 'cookie' (a bit of data) to be specified that will be passed back to us in a callback when devices are discovered--we do not need this here, so NULL is passed. Nothing thread-unsafe about this one, since SearchForProjectors is called only once, and the search does not touch global state.

```
      code = UpnpSearchAsync(handle, 10, PROJECTOR_TYPE, NULL);

}
```

Add a discovered projector, whose identity is encapsulated in 'event', to our list (if the advertised device is in fact a projector!).
event = discovery event from callback function

```
void AddProjector(struct Upnp_Discovery *event) {

  int code;                // return value from Upnp functions
  ProjectorDevice dev;       // new node in device list
  Upnp_Document doc=NULL;  // description document of discovered projector
  int addit=TRUE;          // projector passes enough tests to be useful?
  char *friendly_name=NULL, *baseURL=NULL, *relative_presentation_URL=NULL;
   Upnp_NodeList service_node_list=NULL;   // list of service nodes in
                                           // description doc
  Upnp_NodeList tmp=NULL;

  // lock global data
  pthread_mutex_lock(&mutex);
```

We are only interested in projectors. It is possible we will see advertisements for other sorts of devices, so do not check first—do not do a lot of work if it is not a projector! Also react ONLY to advertisements for root Projector devices--UPnP devices send LOTS of advertisement messages, corresponding to the root device, embedded devices, services, etc. If the event does not identify a projector (i.e., device type is specified) with a device ID, then ignore. It is a projector...do we already know about it? If so, just need to update the expiration time, so we can skip all the heavy duty stuff (e.g., downloading the description document, parsing service information, etc.)

```
  strcpy(dev.UDN, event->DeviceId);
  if (element_in_queue(&device_list, &dev)) {
```

We already know about this projector, updating expiration. Current position in queue is now the matching projector; download its complete description...

```
  peek_at_current(device_list, &dev);
  dev.expiration = event->Expires;
  update_current(&device_list, &dev);
}
else {
```

Interesting data about the projector is in its description document; the first step is to download the document, then we will examine it.

```
  code=UpnpDownloadXmlDoc(event->Location, &doc)
```

Extract all the interesting stuff about the projector.

```
  strcpy(dev.descriptionURL, event->Location);
  friendly_name = ParseItem(doc, NULL, "friendlyName");
  baseURL = ParseItem(doc, NULL, "URLBase");
  relative_presentation_URL = ParseItem(doc, NULL, "presentationURL");
  sprintf(dev.presentationURL,"%s%s",baseURL,relative_presentation_URL);
  strcpy(dev.friendly, friendly_name);
```

Now parsing service information. First retrieve a list of serviceList nodes from the description document. UpnpDocument_getELementsByTagName() returns a bunch of subtrees, the root of each being a serviceList. This includes nested service information for more complicated devices.

```
  service_node_list=UpnpDocument_getElementsByTagName(doc,"serviceList");
```

Narrow the list to *only* root device level services; a UpnpNodeList_item() call with an index of '0' gets us this far. Note that for projectors, there are not any nested services, so 0 should be the only valid index.

```
  tmp = UpnpNodeList_item(service_node_list, 0);
```

Then get a set of subtrees, the root of each being a description of a single available service at this root level.

```
  service_node_list = UpnpElement_getElementsByTagName(tmp, "service");
```

Deal with power, slide control, and present contents services. Failure to get info on any of these services means that we can not use the projector.

```
  printf("ProjectorCP: Parsing individual service descriptions.\n");

  addit = addit &&
  GetServiceInfo(service_node_list,baseURL,POWERSERVICE, &(dev.power)) &&
  GetServiceInfo(service_node_list,baseURL,SLIDESERVICE, &(dev.slide)) &&
 GetServiceInfo(service_node_list,baseURL,PRESENTSERVICE,&(dev.present));
```

Initialize subscription id and timeout for services - these would not be filled in until the projector is selected for use

```
      dev.power.SID[0] = 0;
      dev.slide.SID[0] = 0;
      dev.present.SID[0] = 0;
      dev.power.timeout = 0;
      dev.slide.timeout = 0;
      dev.present.timeout = 0;
   }
```

finally, add the projector if enough stuff went well!

```
if (addit) {
    add_to_queue(&device_list, &dev, 0);
}

  // unlock
  pthread_mutex_unlock(&mutex);
}
```

Read service information for a particular service from a list of service information nodes.
services = list of service nodes, extracted from description document
name = name of target service (e.g., "urn:schemas-upnp-org:service:SlideControl:1")

```
int GetServiceInfo(Upnp_NodeList services, char *baseURL, char *name,
   ProjectorService *service) {

  Upnp_Node service_node=NULL;   // used to iterate through services in list
  char *service_type=NULL;       // type for service under examination
  char *service_id=NULL;         // id of service under examination
  char *url=NULL;     // temporary used to build absolute event, control URLs

  for (i=0; i < UpnpNodeList_getLength(services) && ! ok; i++) {
    service_node = UpnpNodeList_item(services, i);
    // Getting service type.
    service_type = ParseItem(NULL, service_node, "serviceType");

    if (service_type && strcmp(service_type, name) == 0) {
        strcpy(service->service_type, service_type);

     // Getting serviceID
      service_id = ParseItem(NULL, service_node, "serviceId");
      strcpy(service->service_id, service_id);

    // Getting control URL.
    url = ParseItem(NULL, service_node, "controlURL");
    sprintf(service->controlURL, "%s%s", baseURL, url);

    // Getting event subscription URL.
    url = ParseItem(NULL, service_node, "eventSubURL");
    sprintf(service->eventURL, "%s%s", baseURL, url);

   return ok;
}
```

Subscribe to a service's events.

```
int SubscribeService(ProjectorService *service) {
```

Attempt to subscribe, providing the device handle, the event URL for the service, and a timeout. We will get a subscription ID service->SID if all goes well.

```
  service->timeout = REG_TIMEOUT;
  code=UpnpSubscribe(handle,service->eventURL,&(service->timeout),service-
>SID);

}
```

Remove a projector whose identity is encapsulated in 'event', if it is a projector! As for device advertisements, we want to react only when we get the root device's 'byebye' message.

```
void RemoveProjector(struct Upnp_Discovery *event) {

  ProjectorDevice removeit, peek;

  // lock global data
  pthread_mutex_lock(&mutex);
```

Got a root projector device's departure message. Create a ProjectorDevice node with UDN set to departing projector's unique name--the rest does not matter, since that uniquely identifies the service to the queue package. We use this to see if we know about this projector.

```
  strcpy(removeit.UDN, event->DeviceId);

  if (element_in_queue(&device_list,  &removeit)) {

      peek_at_current(device_list, &peek);

      // unsubscribe to all services
      UpnpUnSubscribe(handle, peek.power.SID);
      UpnpUnSubscribe(handle, peek.slide.SID);
      UpnpUnSubscribe(handle, peek.present.SID);
```

See if currently selected projector device is the one we are removing; if it is, notify user and then there is no currently selected projector

```
      if (! CompareProjectors(&peek, current)) {
          current = NULL;
      }
      delete_current(&device_list);
    }
  // unlock
  pthread_mutex_unlock(&mutex);

}
```

Remove all the Projectors

```
void ForgetAllProjectors() {

  ProjectorDevice dev;   // temp used in deletion

  // lock global data
  pthread_mutex_lock(&mutex);

  // Delete each element in the device list
  rewind_queue(&device_list);

  while (! end_of_queue(device_list)) {
    peek_at_current(device_list, &dev);

    // attempt to unsubscribe from services
    UpnpUnSubscribe(handle, dev.power.SID);
    UpnpUnSubscribe(handle, dev.slide.SID);
    UpnpUnSubscribe(handle, dev.present.SID);
    delete_current(&device_list);
  }

  // unlock
  pthread_mutex_unlock(&mutex);
}
```

Turn current projector on/off.

```
void SetPower(int power) {

  // lock global data
  pthread_mutex_lock(&mutex);

  // PowerOn / PowerOff functions take no arguments
  response = SendAction(POWERSERVICE, current->power, power ? "PowerOn" :
              "PowerOff", NULL, NULL, 0);

   isCommandDone = true;
   strcpy(CommandResult,"Projector is ");
   if(power == true)
       strcat(CommandResult,"On");
   if(power == false)
       strcat(CommandResult,"Off");

   // unlock
   pthread_mutex_unlock(&mutex);
}
```

Set count of slides for the presentation.

```
void SetSlideRange(int range) {

  // lock global data
  pthread_mutex_lock(&mutex);

  strcpy(var, "SlideRange");
```

```
  sprintf(val, "%1d", range);
  response = SendAction(SLIDESERVICE, current->slide, "SetSlideRange",
     &var, &val, 1);

  isCommandDone = true;
  strcpy(CommandResult,"Slide Range set to : ");
  strcat(CommandResult,val);

  // unlock
  pthread_mutex_unlock(&mutex);

}
```

Return current slide range current projector.

```
int GetSlideRange() {

  // lock global data
  pthread_mutex_lock(&mutex);

  // No "in" arguments for GetSlideRange
  response = SendAction(SLIDESERVICE, current->slide, "GetSlideRange",
     NULL, NULL, 0);

  // Parse response to determine sliderange. "SlideRange" is an out
  parameter.
  slidestr = ParseItem(response, NULL, "SlideRange");
  slideRange = atoi(slidestr);

  // unlock
  pthread_mutex_unlock(&mutex);

  return slideRange;
}
```

Return current slide # of projector.

```
int GetSlide() {
  // lock global data
  pthread_mutex_lock(&mutex);

  // No "in" arguments for GetSlide
  response = SendAction(SLIDESERVICE, current->slide, "GetSlide",
     NULL, NULL, 0);

  // Parse response to determine slide "CurrentSlide" is an out parameter.
  slidestr = ParseItem(response, NULL, "CurrentSlide");
  slide = atoi(slidestr);
  // unlock
  pthread_mutex_unlock(&mutex);
  return slide;
}
```

Increment the slide variable to go to next slide

```
int ShowNextSlide(void)
{
    slide = GetSlide();
    slide = slide+1;

  // lock global data
  pthread_mutex_lock(&mutex);

  strcpy(var, "CurrentSlide");
  sprintf(val, "%1d", slide);

  response = SendAction(SLIDESERVICE, current->slide, "NextSlide",
     &var, &val, 1);

   isCommandDone = true;
   strcpy(CommandResult,"Displaying Next Slide");

   // unlock
   pthread_mutex_unlock(&mutex);

  return slide;
}
```

Decrement the slide variable to go to previous slide

```
int ShowPrevSlide(void)
{
    slide = GetSlide();
    slide = slide-1;

  // lock global data
  pthread_mutex_lock(&mutex);

  strcpy(var, "CurrentSlide");
  sprintf(val, "%1d", slide);
  response = SendAction(SLIDESERVICE, current->slide, "PrevSlide",
     &var, &val, 1);

   isCommandDone = true;
   strcpy(CommandResult,"Displaying Previous Slide");

  // unlock
  pthread_mutex_unlock(&mutex);

  return slide;
}
```

Jump a particular slide

```
void JumpToSlide(int slide) {

  // lock global data
```

```
  pthread_mutex_lock(&mutex);

  strcpy(var, "CurrentSlide");
  sprintf(val, "%1d", slide);
  response = SendAction(SLIDESERVICE, current->slide, "JumpToSlide",
     &var, &val, 1);

   isCommandDone = true;
   strcpy(CommandResult,"Displaying Slide # : ");
   strcat(CommandResult,val);

  // unlock
  pthread_mutex_unlock(&mutex);

}
```

Displays a blank Slide

```
void ShowBlankSlide(int slide)
{
   // lock global data
   pthread_mutex_lock(&mutex);

   strcpy(var, "CurrentSlide");
   sprintf(val, "%1d", slide);

   response = SendAction(SLIDESERVICE, current->slide, "DisplayBlank",
     &var, &val, 1);

   isCommandDone = true;
   strcpy(CommandResult,"Displaying Blank Slide");

   // unlock
   pthread_mutex_unlock(&mutex);

}
```

Allow user to select a current projector, then subscribe to its services. Unsubscribe to services of any previously selected projector.

```
void SetCurrentProjector(void) {

  ProjectorDevice temp;

  // lock global data
  pthread_mutex_lock(&mutex);

  strcpy(selected,inputFromGUI);

  strcpy(temp.UDN, selected);

  // unsubscribe to previously selected projector, if there is one
  if (current != NULL) {
      // Unsubscribing to services of previous projector
```

```
      UpnpUnSubscribe(handle, current->power.SID);
      UpnpUnSubscribe(handle, current->slide.SID);
      UpnpUnSubscribe(handle, current->present.SID);
  }

  // have a new current projector
  current = (ProjectorDevice *)pointer_to_current(device_list);

  // Subscribing to power service
  SubscribeService(&(current->power));

  // Subscribing to Slide control service
  SubscribeService(&(current->slide));

  // Subscribing to Presentation service
  SubscribeService(&(current->present));

  isCommandDone = true;
  strcpy(CommandResult,"Projector is selected ... UDN : ");
  strcat(CommandResult,selected);

  // unlock
  pthread_mutex_unlock(&mutex);

}
```

List all known projectors. Returns the number of projectors displayed.

```
int ListProjectors(void) {
  // lock global data
  pthread_mutex_lock(&mutex);

  if (queue_length(device_list) == 0) {
    printf("\n**LIST IS EMPTY**\n");
  }
  else {
    rewind_queue(&device_list);

    while (! end_of_queue(device_list)) {
       peek_at_current(device_list, &dev);

       isCommandDone = true;
       strcpy(CommandResult, dev.UDN);
       next_element(&device_list);
    }
    len = queue_length(device_list);
  }
  // unlock
  pthread_mutex_unlock(&mutex);
  return len;

}
```

Set the Current Presentation. Sets the Directory in which all the slides are present. To display all the slide, slides are picked up from that directory

```
void SetCurrentPresentation(void)
{
   // lock global data
   pthread_mutex_lock(&mutex);

   strcpy(var, "Presentation");
   sprintf(val, "%s", inputFromGUI);

   response = SendAction(PRESENTSERVICE, current->present, "SetPresentation",
     &var, &val, 1);


   isCommandDone = true;
   strcpy(CommandResult,"Presentation Selected : ");
   strcat(CommandResult, val);

  // unlock
  pthread_mutex_unlock(&mutex);

}
```

Returns the Presentation name

```
char *GetPresentation() {

  // lock global data
  pthread_mutex_lock(&mutex);

  response = SendAction(PRESENTSERVICE, current->present, "GetPresentation",
     NULL, NULL, 0);

  // parse response to determine slide(unless response is NULL, which is bad)
  // "Presentation" is an out parameter.

  currentPresentation = ParseItem(response, NULL, "Presentation");

  // unlock
  pthread_mutex_unlock(&mutex);

  return currentPresentation;
}
```

Query a state variable. This function allows the control point to peek into the guts of the currently selected projector, bypassing the usual actions that support setting/retrieving state variable values. The projector model actually exposes almost all state variables through Set/Get methods. This function serves as an example of HOW to do state variable queries.

```
void QueryStateVariable(void) {

  char c;                 // identifier for associated service ('0'-'2')
```

```
  char var[NAME_SIZE];    // variable to query
  char *val=NULL;         // returned value

  // lock global data
  pthread_mutex_lock(&mutex);

  // figure out which control URL to try

  switch (c) {
  case '0': controlURL = current->power.controlURL;
    break;
  case '1':  controlURL = current->slide.controlURL;
    break;
  case '2': controlURL = current->present.controlURL;
    break;
  }

  // Gets the value of the specified control variable
  code = UpnpGetServiceVarStatus(handle, controlURL, var, &val);
  // unlock
  pthread_mutex_unlock(&mutex);

}
```

A Upnp_Event contains three interesting fields: a subscription ID, a sequence number (so events can be processed in order, if this is important) and a Upnp_Document which outlines the state changes reported by this event.

```
void HandleGENAEvent(struct Upnp_Event *event) {

  Upnp_NodeList props=NULL;          // pieces of ChangedVariables document
  Upnp_NodeList children=NULL;
  Upnp_Node varnode=NULL;
  Upnp_Node varnodechild=NULL;
  Upnp_Node prop=NULL;
  Upnp_DOMException err;
  char *variable=NULL, *value=NULL;  // name, value of changed state variable

  // lock global data
  pthread_mutex_lock(&mutex);


  // Find out which service the GENA event corresponds to

  if (! strcmp(current->power.SID, event->Sid)) {
    printf("ProjectorCP: Power-related event.\n");
  }
  else if (! strcmp(current->slide.SID, event->Sid)) {
    printf("ProjectorCP: Slide-related event.\n");
  }
  else if (! strcmp(current->present.SID, event->Sid)) {
    printf("ProjectorCP: Contents-related event.\n");
  }
```

Parse supplied document to see what is new. The changed variables are represented as follows in the ChangedVariables XML document:

<e:propertyset ...>
<e:property>
<variablename>new value</variablename>
</e:property>
<e:property>
<variablename>new value</variablename>
</e:property>
... …
</e:propertyset>

Find all of the "e:property" tags...

```
  props=UpnpDocument_getElementsByTagName(event->ChangedVariables,
  "e:property");


   for (i=0; i < UpnpNodeList_getLength(props); i++) {

      prop=UpnpNodeList_item(props, i);
      children = UpnpNode_getChildNodes(prop);
      varnode=UpnpNodeList_item(children, 0);
      variable = UpnpNode_getNodeName(varnode);
      varnodechild = UpnpNode_getFirstChild(varnode);
      value = UpnpNode_getNodeValue(varnodechild, &err);
```

Free all the variables using appropriate UpnpXXX_free() functions

```
      if (children) { UpnpNodeList_free(children); }
      if (varnode) { UpnpNode_free(varnode); }
      if (varnodechild) { UpnpNode_free(varnodechild); }
      if (prop) { UpnpNode_free(prop); }
  }

  if (props) { UpnpNodeList_free(props); }

  // unlock
  pthread_mutex_unlock(&mutex);

}
```

Perform an action on the currently selected projector. Returns the response document; the calling code is responsible for freeing associated storage.
servtype == type of service (e.g., "urn:schemas-upnp-org:service:PowerSwitch:1")
service == description of service
func == name of action (e.g., "SetPower")
args == argument names
values == argument values
numargs == number of elements in args, values string arrays

```
Upnp_Document SendAction(char *servtype, ProjectorService service, char
*func, char **args, char **values, int numargs) {
  Upnp_Document doc=NULL;       // XML representation of our requested action
  Upnp_Document response=NULL;  // return value from action
```

```
  int code;                     // return value from UPnP functions
  int j;


  // create initial document to describe this action
  doc = UpnpMakeAction(func, servtype, 0, NULL, NULL);

  for (j=0; j < numargs && code == UPNP_E_SUCCESS; j++) {
      code = UpnpAddToAction(&doc, func, servtype, args[j], values[j]);
  }

  code = UpnpSendAction(handle, service.controlURL, servtype,
      current->UDN, doc, &response);

  return response;
}
```

All asynchronous notifications of interesting events come through this handler. This guy is the reason that a sempaphore has to be used to lock global device state before accessing it! For each event type, hand off processing to an appropriate function unless only a few lines of code are required.
type == type of event that occurred
event == the event description
cookie == data we specified during registration of the callback

```
int CallbackHandler(Upnp_EventType type, void *event, void *cookie) {


 // the following are the only events of interest for a control point;
 // devices will respond to other events

switch (type) {
```

New device discovered, either through a search or receipt of an advertisement. We do use the asynchronous version of the Upnp search function, UpnpSearchAsync(). For most other things, such as changing state variables, etc., we do synchronous calls.

```
   case UPNP_DISCOVERY_ADVERTISEMENT_ALIVE:
   case UPNP_DISCOVERY_SEARCH_RESULT:
          AddProjector((struct Upnp_Discovery *)event);
   break;
```

Nothing found within specified timeout

```
   case UPNP_DISCOVERY_SEARCH_TIMEOUT:
          // nothing to do here
   break;
```

A device is saying goodbye - forget about the device

```
   case UPNP_DISCOVERY_ADVERTISEMENT_BYEBYE:
         RemoveProjector((struct Upnp_Discovery *)event);
   break;
```

Asynchronous notification of the status of an earlier action; generally only care about this if an error occurred. Code for checking the error return is provided here as an example, but we do

not expect this kind of event, since we do actions synchronously using UpnpSendAction() rather than UpnpSendActionAsync().

```
case UPNP_CONTROL_ACTION_COMPLETE:
{
   struct Upnp_Action_Complete *a_event =
       (struct Upnp_Action_Complete *) event;
   if (a_event->ErrCode != UPNP_E_SUCCESS) {
        // An asynchronous action completed but returned an error.
        Diagnose("CallbackHandler()", __LINE__, a_event->ErrCode);
   }
   // Not expecting asynchronous action notifications?
}
break;
```

Asynchronous notification of the completion of a state variable query; generally only care about this if an error occurred. Code for checking the error return is provided here, but we do not expect this kind of event, since we do queries synchronously using UpnpGetServiceVarStatus() rather than UpnpGetServiceVarStatusAsync(). If the latter is used, the important variables in the UpnpState_Var_Complete event, other than the error code, are StateVarName (name of the variable) and CurrentVal (its value).

```
case UPNP_CONTROL_GET_VAR_COMPLETE:
{
   struct Upnp_State_Var_Complete *sv_event =
     (struct Upnp_State_Var_Complete *) event;
   if (sv_event->ErrCode != UPNP_E_SUCCESS){
       // An asynchronous state var query returned an error
       Diagnose("CallbackHandler()", __LINE__, sv_event->ErrCode);
   }
   // Not expecting asynchronous state var query notifications
}
break;
```

This is a GENA event that alerts us that state variables have changed value

```
case UPNP_EVENT_RECEIVED:
 HandleGENAEvent((struct Upnp_Event *)event);
 break;
```

Asynchronous notification of completion of subscribe/unsubscribe. As for control variable queries, etc. we use the synchronous versions of subscribe and unsubscribe, so we do not expect to receive these events. If UpnpSubscribeAsync()/UpnpUnSubscribeAsync() are used, then this is the place to know that the subscribe/unsubscribe attempt completed. The error code evaluation code below is provided for illustrative purposes only.

```
case UPNP_EVENT_SUBSCRIBE_COMPLETE:
case UPNP_EVENT_UNSUBSCRIBE_COMPLETE:
{
     struct Upnp_Event_Subscribe *es_event =
         (struct Upnp_Event_Subscribe *)event;

     if (es_event->ErrCode != UPNP_E_SUCCESS) {
          // An asynchronous subscribe/unsubscribe returned an error
          Diagnose("CallbackHandler()", __LINE__, es_event->ErrCode);
     }
     // Not expecting asynchronous subscription notifications
}
break;
```

Asynchronous notifiction of renewal completion. Again, we use the synchronous versions so do not expect to see these events.

```
    case UPNP_EVENT_RENEWAL_COMPLETE:
    {
       struct Upnp_Event_Renewal *er_event = (struct Upnp_Event_Renewal
       *)event;

       if (er_event->ErrCode != UPNP_E_SUCCESS) {
             // An asynchronous renewal returned an error
             Diagnose("CallbackHandler()", __LINE__, er_event->ErrCode);
       }
       // Not expecting asynchronous renewal notifications
    }
    break;
```

We receive these events when autorenewal of service subscriptions fails. The autorenewal of service subscriptions by the UPnP SDK makes our life very easy. The default autorenewal period is 35 seconds, which is much less than our requested expiration time, so these events should never occur unless the service is being shut down. Note that autorenewal is not mandated by the UPnP specification, so it is possible that some SDKs will not do the work for you!

```
    case UPNP_EVENT_AUTORENEWAL_FAILED:
    case UPNP_EVENT_SUBSCRIPTION_EXPIRED:
           printf("!!!!! Service subscription expired !!!!!!\n");
    break;

    default:
           printf("ProjectorCP: Do not know why I received that event!!\n");
    break;
   }
   return(0);
}
```

Deal with control-C

```
void SignalHandler() {

  int sig;
  sigset_t sigs_to_catch; // signal handling stuff (for handling shutdown)
  sigemptyset(&sigs_to_catch);
  sigaddset(&sigs_to_catch, SIGINT);
  sigwait(&sigs_to_catch, &sig);
  Shutdown("Shutting down.");
}
```

Do a clean shutdown. Unregister (if we got that far) and then call UpnpFinish() before dying.

```
int Shutdown(char *reason) {

  printf("Wait...\n");
  if (handle >= 0) {
    UpnpUnRegisterClient(handle);
  }
```

```
  UpnpFinish();
  printf("%s\n", reason);
  done=true;
  exit(0);
  return;
}
```

Compare projectors; projectors are the same if their unique names are the same – do not care about anything else

```
int CompareProjectors(void *b1, void *b2) {
  ProjectorDevice *projector1 = (ProjectorDevice *)b1;
  ProjectorDevice *projector2 = (ProjectorDevice *)b2;

  return strcmp(projector1->UDN, projector2->UDN);
}
```

Called in a thread. May be used to check if 'done = true'

```
void *TimerLoop(void *args) {  return NULL;    }
```

Code List 4.6: nativeLib.c; Control Point implementation

4.3.3 ParseItem, a function from “common.c”.

Given a Upnp_Document or a UpnpElement, return the corresponding string value for a specific tag. Uses the DOM (document object model) functions provided in the UPnP SDK. The caller is responsible for freeing storage associated with the return value using free().
doc == UPnP Document
element == UPnP Element
tag == tag to search for

```
char *ParseItem(Upnp_Document doc, Upnp_Element element, Upnp_DOMString tag)
{

  Upnp_NodeList node_list=NULL;
  Upnp_Node node=NULL;
  Upnp_Node tmp=NULL;
  Upnp_DOMException err;
  char *ret=NULL;
  Upnp_DOMString val=NULL;

  // depending on whether a document or element was passed, call the
  // appropriate function to retrieve matches

  if (doc) {
    node_list = UpnpDocument_getElementsByTagName(doc, tag);
  }
  else {
    node_list = UpnpElement_getElementsByTagName(element, tag);
  }

  tmp = UpnpNodeList_item(node_list, 0);
  node = UpnpNode_getFirstChild(tmp);
  val = UpnpNode_getNodeValue(node, &err);
  ret = malloc(strlen((char *)val)+1);
  strcpy(ret, (char *)val);

  UpnpNode_free(tmp);
  UpnpNodeList_free(node_list);
  UpnpNode_free(node);
  UpnpDOMString_free(val);

  return ret;
}
```

Code List 4.7: Function ParseItem() from common.c

4.3.4 JavaCBridge.java

Program used as a bridge between the GUI and the Control point

```
public class JavaCBridge{

    public native int doInitialization(String ip, String port);
    public native int sendCommand(String cmd, String input);
    public native String retrieveCommandResult();

    int retVal;
    String cmdResult=new String();
```

Called from GUI at the very start. Registers the Control point and searches for the available Projector devices over the network.

```
    public int doInit(String ip, String port, GUI gui)
    {
        try{
           retVal=doInitialization(ip,port);
        }
        catch(Exception e)
        {
           e.printStackTrace();
        }
        return 0;
    }
```

Called from the GUI to perform all the service related actions

```
    public int doCommand(String cmd, String input, GUI gui)
    {
        try{
            retVal=sendCommand(cmd,input);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        return 0;
    }
```

Called from GUI to collect the results of the action requested by user; e.g. Power On/Off, Next slide etc.

```
    public String getCommandResult()
    {
        try{
            cmdResult=retrieveCommandResult();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        return cmdResult;
    }
}
```

Code List 4.8: JavaCBridge Class implementation; Interface between GUI and Control Point

CHAPTER 5. CONCLUSION AND FUTURE WORK

5.1 Conclusion

Service discovery plays an important role in networks. It lets mobile users who have no clear understanding of their environment find service providers that are physically close and accessible. In our project, an enabled LCD Projector becomes available as soon as it is plugged in. The control point that wants to use it, downloads the configuration information directly from the projector device, and then starts using the projector device almost immediately.

We have implemented a working prototype of the LCD Projector System. It is a distributed system that allows mobile users to look for available projector devices on the network and let them select the one of their interest for doing their presentation. Thus, the mobile users will not be burdened with extraneous information for services in remote locations. In order to search for the service the client needs to know very little about the environment. Knowing the 'service type' would be quite sufficient to track down the services available. The LCD Projector System allows service providers to supply dynamic service updates about the changes in the device state. This eliminates continuous polling for the interesting state changes. This dramatically improves the value of the service for the providers and gives users more refined service information.

There are many service discovery technologies available, e.g., Jini[10], Universal Plug and Play[4], Service Location Protocol[11] etc. One of the most significant reasons for using UPnP is that it gives the choice of language and operating system to be used for the implementation. UPnP also provides the flexibility to leverage lower layers of the protocol stack where this makes sense, thereby keeping UPnP designs simple and effective. UPnP uses open, standard protocols such as TCP/IP, HTTP and XML, which makes it easy to program.

We have implemented a virtual control point and a virtual projector device in C on a Linux platform with a GUI in Java. We use the Java Native Interface to communicate between the GUI and the control point. The range of available AWT classes in Java makes the programming of the GUI much easier than programming it in C. Also, since Java is platform independent, the GUI is portable across various platforms.

An interesting feature that has been achieved is the ability to use multiple projectors at a time. It gives the user more freedom and choices for their sophisticated presentations.

5.2 Future Work

The current system was designed to support projector devices only. In the future, the system can be enhanced to support various other devices like printers, scanners, digital cameras, etc. Though this will substantially increase the implementation effort, as we will have to keep track of different device states, it will give the user of the system a great range of devices to choose from.

For displaying the slides we use an image displaying software called 'Electric Eyes' developed for Linux systems. Electric Eyes does not support opening multiple images in the same window. Therefore, whenever there is a need to display a new slide, a new window of electric eyes is opened and the previously open window is closed. This operation may not be smooth if the processor is not fast enough. Instead it gives a flickering effect. A solution to this problem could be embedding a JPEG image displaying code into the existing program on the device side for displaying the slides.

REFERENCES

[1] Service and Device Discovery Protocols and Programming; McGraw-Hill; Golden G. Richard III; 2002

[2] Essential JNI: Java Native Interface; Rob Gordon; 1998

[3] XML; McGraw-Hill; Solomon H. Simon; 2001

[4] Universal Plug and Play Specification, v1.0; www.upnp.org

[5] Universal Plug and Play Technology; www.microsoft.com/hwdev/tech/nonpc/upnp/default.asp

[6] Universal Plug and Play; msdn.microsoft.com/library/en-us/wceupnps/htm/_wcecomm_Universal_Plug_and_Play_UPnP.asp

[7] UPnP Newsletter; http://www.upnp.org/newsletters/newsletterII/tech.htm

[8] Java Native Interface; http://java.sun.com/docs/books/tutorial/native1.1/

[9] Understanding XML; http://java.sun.com/webservices/docs/1.0/tutorial/doc/IntroXML.html

[10] Jini Network Technology; http://wwws.sun.com/software/jini/

[11] SLP White Pages; http://playground.sun.com/srvloc/slp_white_paper.html

[12] Resources for DHCP; http://www.dhcp.org/

[13] RedHat Linux official website; http://www.redhat.com/

[14] Java official website; http://java.sun.com/

[15] DNS Resource Directory; http://www.dns.net/dnsrd/

[16] The World Wide Web Consortium; http://www.w3.org/

VITA

Jeevan Kale was born in Pune, India on 1978. His younger years were spent in a small town near Pune called Bhosari. Then he moved to Pune for his high school and college education. He entered Pune Institute of Computer Technology on 1995, one of the best colleges for Computer Engineering in Pune, India. After completing an intense graduation program of 4 years in Computer Engineering he earned a Bachelor of Engineering degree on 1999.

On Fall 2000 he entered University of New Orleans and enrolled in the Masters program at the department of Computer Science. During his studies he was working as a Student consultant–II at University Computing and Communications, UNO for 18 months. He also worked at Center for Society, Law and Justice as a Graduate Assistant for 9 months. His graduate studies were concentrated on software development in distributed systems.

EXAMINATION AND THESIS REPORT

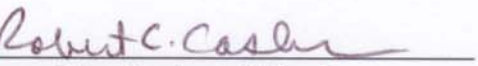
Candidate: Jeevan Kale

Major Field: Computer Science

Title of Thesis: A Service Discovery-Enabled LCD Projector Device

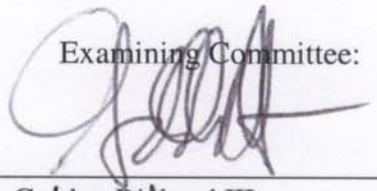
Approved:

Major Professor & Chair: Dr. Golden Richard III

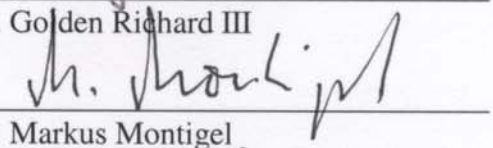


Dean of the Graduate School

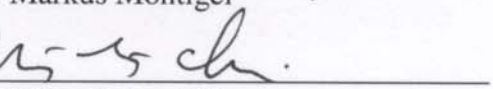
Examining Committee:



Dr. Golden Richard III



Dr. Markus Montigel



Dr. Ming-Hsing Chiu

Date of Examination:

December 5, 2002