CLIENT SERVER APPLICATION FOR MOBILE DEVICES USING J2ME

BY

AKUL GARG (061403)

GURDIP SINGH (061418)

ANKUR BROOTA (061447)

DEPARTMENT OF CS AND IT

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY-WAKNAGHAT

MAY-2010

Waknaghat, Solan-173215, Himachal Pradesh

CERTIFICATE

This is to certify that the work entitled, **"Client Server Application For Mobile Devices Using J2ME"** submitted by **Akul Garg(061403), Gurdip Singh(061418), Ankur Broota(061447)** in partial fulfillment for the award of degree of Bachelor of Technology in Information Technology of Jaypee University of Information Technology has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Mr P.K.Gupta

(Project Supervisor)

Department of Computer Science and Engineering

Jaypee University of Information Technology

Waknaghat, Solan-173215

ACKNOWLEDGMENT

Apart from the efforts by us, the success of this project depends largely on encouragement and guidelines of many others. We take this opportunity to express our gratitude to the people who have been instrumental in the successful completion of this project. We would like to show our greatest appreciation to our supervisor Mr.P.K Gupta. We feel motivated and encouraged every time we get his encouragement. For his coherent guidance throughout the tenure of the project, we feel fortunate to be taught by him, who gave us his unwavering support. Besides being our mentor, he has taught us that there's no substitute for hard work. Being a dynamic personality himself, he has a practical approach towards a profession. Also, the guidance and support received from all the team members including Akul Garg, Gurdip Singh,Ankur Broota who contributed, which was vital for the success of the project. We feel grateful working together as a group. We owe our heartiest thanks to Brig. (Retd.) S.P.Ghrera(H.O.D.-CSE/IT Department) who've always inspired confidence in us to take initiative. He has always been motivating and encouraging. Finally, thanks to all our family members who supported us in our every grim phase.

Project Group No.-12

Akul Garg(061403) Akul Garg

Gurdip Singh(061418) Gurdip Singh

Ankur Broota(061447) AnkurBroota

B.TECH (Computer Science and Engineering)

Jaypee University of Information Technology

CONTENTS

1. Introduction.....1

1.1 Definition.....2

1.2 Purpose.....3

1.3 Intended Audience.....3

2. Java 2 Micro Edition (J2ME)..... 4

2.1 J2ME Configuration.....4

2.1.1. Connection Limited Device Configuration (CLDC).....5

2.1.2. CLDC Security.....6

2.1.3. Connected Device Configuration.....7

2.2 Profile.....8

2.2.1. K java.....8

2.2.2. MIDP.....8

2.3 The MIDlet Life Cycle10.....9

2.4 MIDlet Suite 10

2.5 Display Hierarchy.....11

3. Bluetooth.....12

3.1 Piconet and Scatternet.....13

3.2 Bluetooth Architecture.....16

4. Client Server Application using Bluetooth.....18

4.1 Server..... 19

4.2 Server Initialization.....19

4.3 Connecting To the Client.....21

4.4 Device Discovery.....22

4.5 Service Search.....24

4.6 Applications Developed.....25

4.6.1 Dictionary.....25

4.6.2 PC Control.....25

4.6.3 File Search.....25

4.7 Graphical Representation of Project.....26

4.8 Data Flow Diagram (DFD).....28

4.8.1 DFD Level 0.....28

4.8.2 DFD Level 1.....28

4.8.3 DFD Level 2.....29

5 Testing Technologies To be Used.....30

5.1 Black Box Testing.....30

5.1.1 Step 1.....31

5.1.2 Step 2.....32

5.1.3 Step 3.....33

5.1.4 Step 4.....33

5.1.5 Step 5.....34

5.1.6 Step 6.....34

5.2 Integration Testing.....35

5.3 Unit Testing.....36

Appendix Sample Code.....37

Server.....37

Remote Discovery.....42

Logger.java.....45

Service Search.....46

SearchInfo.java.....50

LIST OF FIGURES

Fig 1.1 J2ME Architecture.....4

Fig 2.1 CLDC And CDC.....7

Fig 2.2 MID Architecture.....9

Fig 2.3 Display Class Hierarchy.....11

Fig 3.1 A Typical Piconet.....13

Fig 3.2 Scatternet.....14

Fig 3.3 Piconet With 2 Nodes.....15

Fig 3.4 Scatternet With 3 Nodes.....15

Fig 3.5 Piconet With 3 Nodes.....16

Fig 3.6 Bluetooth stack.....17

Fig 4.1 Client Server Model.....18

Fig 4.2 Device Discovery State Diagram.....23

Fig 4.3 Service Search State Diagram.....24

Fig 4.4 Graphical Representation.....26

ABBREVIATIONS

1. **J2ME** Java To Micro Edition for building the application.
2. **JABWT** The Java APIs for Bluetooth Wireless Technology
3. **UUID** Is a unique Bluetooth ID
4. **CLDC** Connected Limited Device Configuration
5. **CDC** Connected Device Configuration

Abstract

We have developed some applications which can run on mobile phones due to its hardware or processing constraints. It will send its input (taken on mobile, referred to as client) to laptop connected via a Bluetooth (referred to as a work station) for processing, and will receive back the output on same mobile through which input was sent.

We have developed 3 applications

- Dictionary
- PC control
- File Search

1. INTRODUCTION

Wireless technologies are becoming more and more popular around the world. Consumers appreciate the wireless lifestyle, relieving them of the well known "cable chaos" that tends to grow under their desk. Nowadays, the world would virtually stop if wireless communications suddenly became unavailable. Both our way of life and the global economy are highly dependent on the flow of information through wireless mediums like television and radio. Mobile phones have become highly available during the last decade.

Now virtually everyone owns a mobile phone, making people available almost wherever they are. Many companies are highly dependent on their employees having mobile phones, whereas some companies have even decided not to employ stationary phone systems but instead use mobile phones exclusively throughout the organization. The Bluetooth wireless technology is one of these technologies which are being implemented.

New wireless technologies are being introduced at an increasing rate in the organizations. During the last few years the IEEE 802.11 [1] technologies have started to spread rapidly, enabling consumers to set up their own wireless networks. This constitutes an important change in how wireless communications are made available to consumers. Wireless networks are no longer provided by big corporations alone, they can just as well be implemented by individuals. Our society is becoming more and more dependent on wireless communications as new areas of use are introduced.

Java enabled mobile phones have already been on the market for some years. Due to the very resource constrained mobile phones available a few years ago, Java applications were not very sophisticated and did not hit the mass-market the way many had hoped. As seen in the rest of the software and hardware industry, games play an important role in driving the development of both hardware and software forward. It is therefore interesting to see that a large market has emerged lately for Java games targeting mobile devices. Processing power, available memory, screen size, and screen resolution are increasing as new Java enabled mobile devices enter the market. Newly

released Java applications are accordingly sophisticated, and will help to spread the Java technology usage even further.

The Java APIs for Bluetooth Wireless Technology (JABWT) ties the Java technology and the Bluetooth technology together. One can easily imagine different scenarios where JABWT would be useful, e.g. the functionality of existing Java games is extended to support multi-player games using Bluetooth connectivity. Other interesting scenarios emerge as well, such as a consumer using a Java Bluetooth enabled mobile phone to pay for a soda by connecting to a Bluetooth enabled soda vending-machine

J2ME applications have in the recent years gained considerable popularity in global scale, and have attracted a significant part of the mobile user community. The above observations can be explained by the fact that these applications make it possible for users to manage and share various kinds of information among them.

By doing research on this topic we found out that the J2ME applications do not use databases and prefer only limited internal database. So the main objective of taking up this project is to develop an application which has synchronization between Mobile devices and external databases. These applications permit only restricted users to access the databases within the organization.

1.1 Definition

Sun Microsystems define J2ME as "a highly optimized Java run-time environment targeting a wide range of consumer products, including pagers, cellular phones, screen-phones, digital set-top boxes and car navigation systems."

1.2 Purpose

J2ME application allows the user to perform various tasks on his mobile phones that were not earlier possible like cross functionality in mobile phones.

1.3 Intended Audience

This document is intended for different types of readers, such as developers, project managers, management staff, users, testers, and documentation writers.

2. JAVA 2 MICRO EDITION (J2ME)

This gives an overview of the J2ME technology. The J2ME architecture is described in general before the components in the J2ME technology are introduced. J2ME applications are also discussed in general, and it is explained how they are made available to end users. Finally, JABWT is discussed, showing where it has its place in the J2ME architecture. J2ME is a highly optimized Java runtime environment. J2ME is aimed at the consumer and embedded devices market. This includes devices such as cellular telephones, Personal Digital Assistants (PDAs) and other small devices.

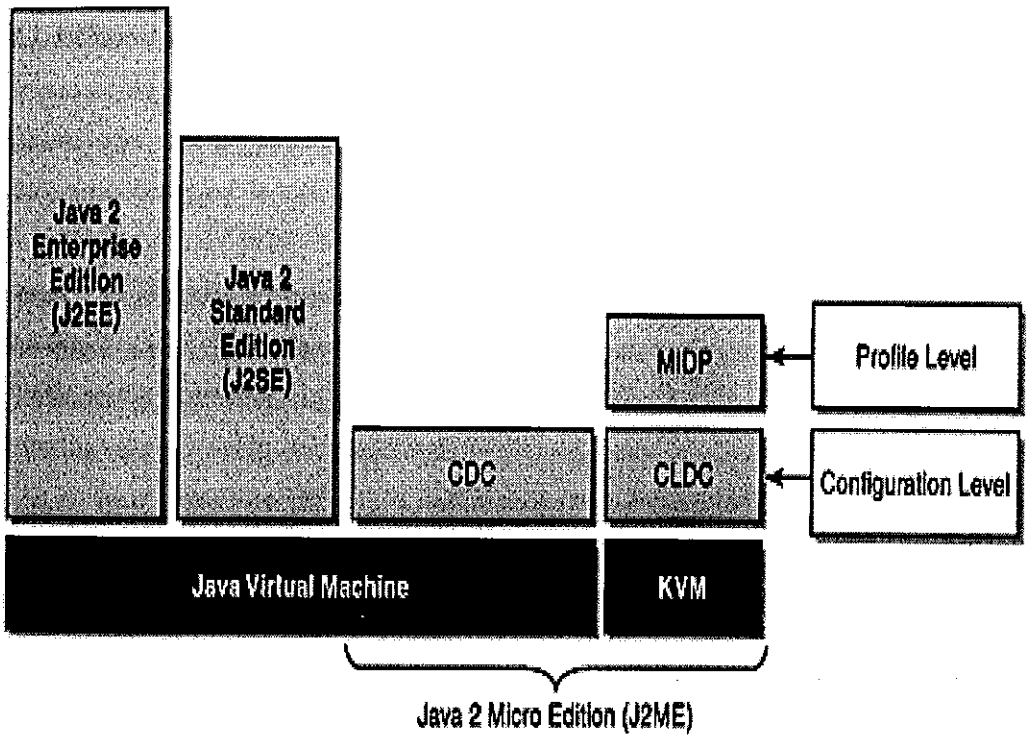


Fig-1.1 J2ME architecture

2.1 J2ME Configurations

The configuration defines the basic run-time environment as a set of core classes and a specific JVM that run on specific types of devices. You also learned that the two types of configurations

for J2ME are CLDC and CDC. Sun provides J2ME configurations that are suitable for different market segments – CLDC for small devices and CDC for larger devices. A J2ME environment can be configured dynamically to provide the environment needed to run an application, regardless of whether or not all Java technology-based libraries necessary to run the application are present on the device. The core platform receives both application code and libraries. Configuration is performed by server software running on the network. In the next few panels, you will learn more about CLDC and CDC and which profiles they are associated with.

2.1.1 Connected Limited Device Configuration (CLDC)

CLDC was created by the Java Community Process, which has standardized this "portable, minimum-footprint Java building block for small, resource-constrained devices," as defined on Sun Microsystems' Web site. The J2ME CLDC configuration provides for a virtual machine and set of core libraries to be used within an industry-defined profile. As mentioned, a profile defines the applications for particular devices by supplying domain-specific classes on top of the base J2ME configuration.

The K virtual machine (KVM), CLDC's reference implementation of a virtual machine, and its KJava profile run on top of CLDC. CLDC outlines the most basic set of libraries and Java virtual machine features required for each implementation of J2ME on highly constrained devices. CLDC targets devices with slow network connections, limited power (often battery operated), 128 KB or more of non-volatile memory, and 32 KB or more of volatile memory. Volatile memory is non-persistent and has no write protection, meaning if the device is turned off, the contents of volatile memory are lost.

With non-volatile memory, contents are persistent and write protected. CLDC devices use non-volatile memory to store the run-time libraries and KVM, or another virtual machine created for a particular device. Volatile memory is used for allocating run-time memory.

2.1.1.1 CLDC Security

The security model of the CLDC is defined at three different levels, low-level security, application-level security and, end-to-end security [24]. Low-level security ensures that the application follows the semantics of the Java programming language. It also ensures that an ill-formed or maliciously encoded class file does not crash or in any other way harm the target device. In a standard Java virtual machine implementation this is guaranteed by a class file verifier, which ensures that the byte codes and other items stored in class files cannot contain illegal instructions, cannot be executed in an illegal order, and cannot contain references to invalid memory locations or memory areas outside the Java object memory.

However, the conventional J2SE class verifier takes a minimum of 50 kB binary code space and typically at least 30-100 kB of dynamic Random Access Memory (RAM) at runtime. This is not ideal for small, resource constrained devices. Because of this, a different approach is used for class file verification in CLDC. Class files are pre-verified off device, usually on the workstation used by the developer to compile the applications. The pre-verification process will add some information to the classes, making runtime verification much easier. The result is that the implementation of the class verifier in Sun's KVM requires about 10 kB of Intel x86 binary code and less than 100 bytes of dynamic RAM at runtime for typical class files.

Application-level security means that the application will run in the CLDC sandbox model. The application should only have access the resources and libraries permitted by the Java application environment. This means that the application programmer must not be able to modify or bypass the standard class loading mechanisms of the virtual machine. The CLDC sandbox model also requires that a closed, predefined set of Java APIs is available to the application programmer, defined by the CLDC, profiles (e.g. MIDP) and manufacturer-specific classes. The application programmer must not be able to override, modify, or add any classes to the protected java.*, javax.microedition.*, profile-specific or manufacturer-specific packages.

End-to-end security usually requires a number of advanced security solutions (e.g. encryption and authentication). The CLDC expert group decided not to mandate a single end-to-end security

mechanism. Therefore, all end-to-end security solutions are assumed to be implementation dependent and outside the scope of the CLDC specification.

2.1.2 *Connected Device Configuration (CDC)*

Connected Device Configuration (CDC) has been defined as a stripped-down version of Java 2 Standard Edition (J2SE) with the CLDC classes added to it. Therefore, CDC was built upon CLDC, and as such, applications developed for CLDC devices also run on CDC devices. CDC, also developed by the Java Community Process, provides a standardized, portable, full-featured Java 2 virtual machine building block for consumer electronic and embedded devices, such as smart phones, two-way pagers, PDAs, home appliances, point-of-sale terminals, and car navigation systems. These devices run a 32-bit microprocessor and have more than 2 MB of memory, which is needed to store the C virtual machine and libraries. While the K virtual machine supports CLDC, the C virtual machine (CVM) supports CDC.

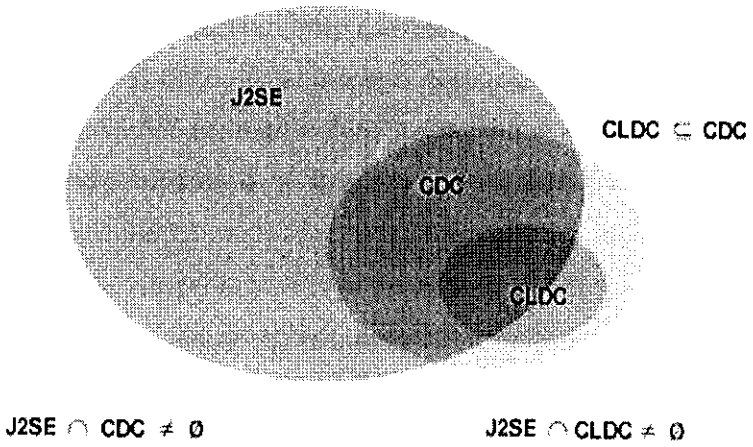


Fig-2.1 CLDC and CDC

2.2 J2ME Profile

As we mentioned earlier in the previous chapter, a profile defines the type of device supported. The Mobile Information Device Profile (MIDP), for example, defines classes for cellular phones. It adds domain-specific classes to the J2ME configuration to define uses for similar devices. Two profiles have been defined for J2ME and are built upon CLDC: KJava and MIDP. Both KJava and MIDP are associated with CLDC and smaller devices. Profiles are built on top of configurations. Because profiles are specific to the size of the device (amount of memory) on which an application runs, certain profiles are associated with certain configurations. A skeleton profile upon which you can create your own profile, the Foundation Profile, is available for CDC. However, for this tutorial and this section, we will focus only on the KJava and MIDP profiles built on top of CLDC.

2.2.1 *Profile 1: KJava*

KJava is Sun's proprietary profile and contains the KJava API. The KJava profile is built on top of the CLDC configuration. The KJava virtual machine, KVM, accepts the same byte codes and class file format as the classic J2SE virtual machine. KJava contains a Sun-specific API that runs on the Palm OS. The KJava API has a great deal in common with the J2SE Abstract Windowing Toolkit (AWT). However, because it is not a standard J2ME package, its main package is com.sun.kjava

2.2.2 *Profile 2: MIDP*

MIDP is geared toward mobile devices such as cellular phones and pagers. The MIDP, like KJava, is built upon CLDC and provides a standard run-time environment that allows new applications and services to be deployed dynamically on end-user devices. MIDP is a common, industry-standard profile for mobile devices that is not dependent on a specific vendor. It is a complete and supported foundation for mobile application development.

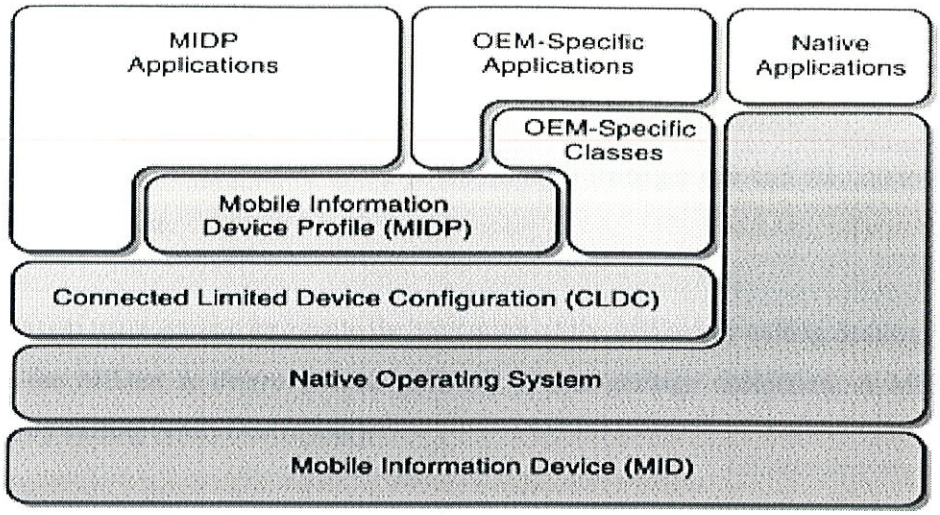


Fig-2.2 Mobile information device architecture

2.3 The MIDlet Life Cycle

MIDP applications are represented by instances of the javax.microedition.midlet.MIDlet class. MIDlet has a specific life cycle, which is reflected in the methods and behaviour of the MIDlet class. A piece of device-specific software, the application manager, controls the installation, execution, and life cycle of MIDlets. MIDlets have no access to the application manager. A MIDlet is installed by moving its class files to a device. The class files will be packaged in a Java Archive (JAR), while an accompanying descriptor file (with a .jad extension) describes the contents of the JAR.

A MIDlet goes through the following states:

1. When the MIDlet is about to be run, an instance is created. The MIDlet's constructor is run, and the MIDlet is in the *Paused* state.

2. Next, the MIDlet enters the *Active* state after the application manager calls startApp().

3. While the MIDlet is Active, the application manager can suspend its execution by calling pauseApp(). This puts the MIDlet back in the Paused state. A MIDlet can place itself in the paused state by calling notifyPaused().

4. While the MIDlet is in the Paused state, the application manager can call startApp() to put it back into the Active state.

5. The application manager can terminate the execution of the MIDlet by calling destroyApp(), at which point the MIDlet is *destroyed* and patiently awaits garbage collection. A MIDlet can destroy itself by calling notifyDestroyed().

2.4 MIDlet suites

MIDlets are usually available through MIDlet suites. It consists of two files, a .jar and a .jad file. The Java Archive (JAR) file contains compiled classes in a compressed and pre-verified format. Several MIDlets may be included in one MIDlet suite. Hence, the JAR file will contain all these MIDlet classes. This enables multiple them to share resources, like common libraries included in the MIDlet suite or data stored on the device. Because of security constraints, a MIDlet may only access the resources associated with its own MIDlet suite.

This applies to all resources, such as libraries it may depend on or data stored on the MID. The Java Application Descriptor (JAD) file is a plain text file containing information about a MIDlet suite. All MIDlets must be named in this file, the size of the JAR file must be included (and be correct!) and the URL to the JAR file must be present. In addition, the MIDlet suite version number is included here. This is essential information for a MID. The MID will always download the JAD file first and inspect its contents.

If the MIDlet suite is already installed, it will know if a newer version is available. The size of the JAR file is important information, the MID can determine if there is enough memory

available to install the MIDlet suite. If all is well the MID can go to the supplied URL and download the JAR file. Other attributes may be included as well.

2.5 Display Hierarchy

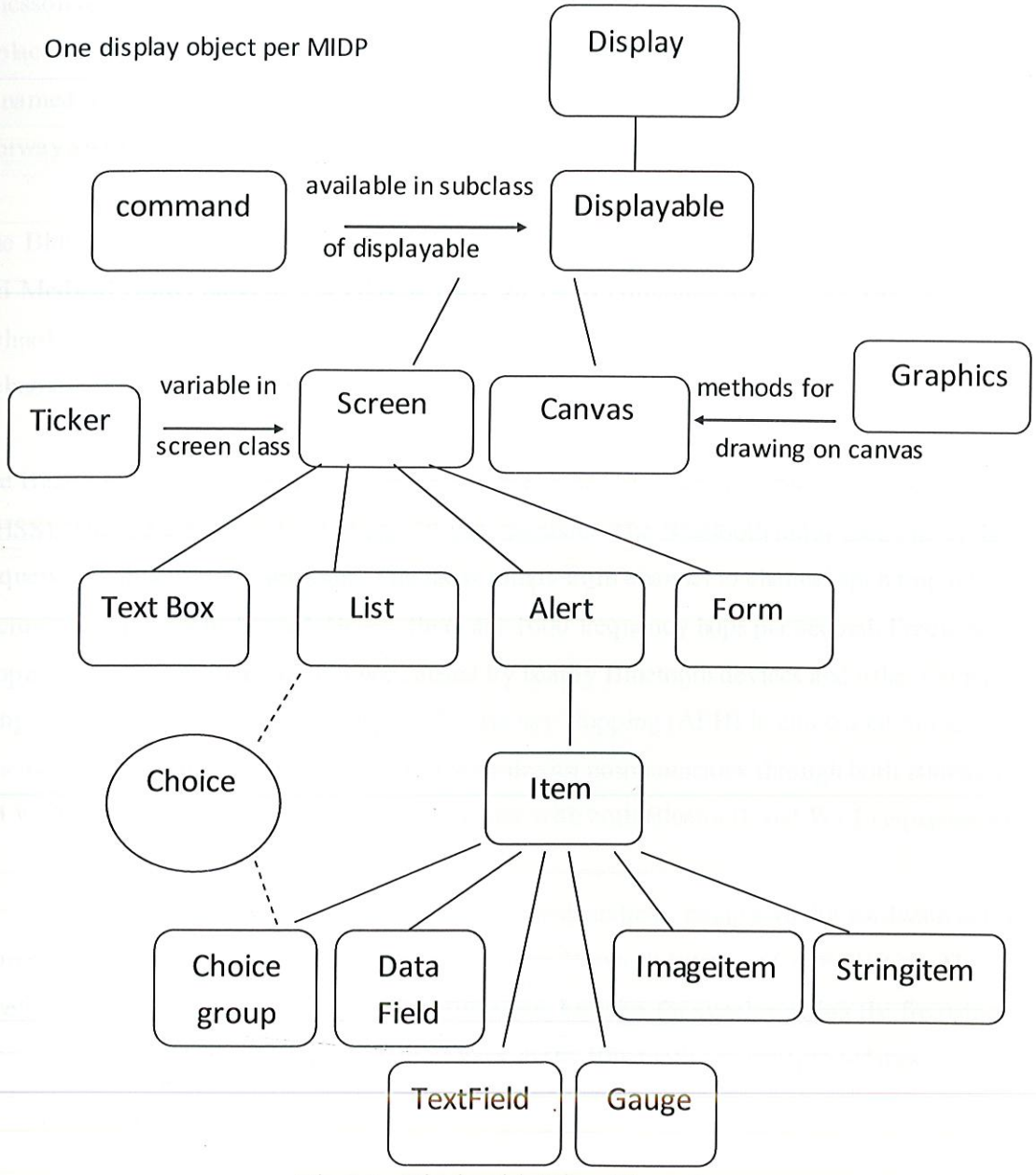


Fig-2.3 Displayable Class Hierarchy

3. Bluetooth

Bluetooth is a low cost, low power; short-range radio technology intended to replace cable connections between mobile phones, PDAs and other portable devices. It can clean up your desk considerably, making wires between your workstation, mouse, laptop computer etc. obsolete. Ericsson Mobile Communications started developing the Bluetooth system in 1994, looking for a replacement to the cables connecting mobile phones and their accessories. The Bluetooth system is named after a tenth-century Danish Viking king, Harald Blåtand, who united and controlled Norway and Denmark. The first Bluetooth devices hit the market around 1999.

The Bluetooth radio is the lowest layer of Bluetooth communication. The Industrial, Scientific and Medical (ISM) band at 2.4 GHz is used for radio communication. Note that several other technologies use this band as well. Wi-Fi technologies like IEEE 802.11b/g and kitchen technologies like microwave ovens may cause interference in this band.

The Bluetooth radio utilizes a signalling technique called Frequency Hopping Spread Spectrum (FHSS). The radio band is divided into 79 sub-channels. The Bluetooth radio uses one of these frequency channels at a given time. The radio jumps from channel to channel spending 625 microseconds on each channel. Hence, there are 1600 frequency hops per second. Frequency hopping is used to reduce interference caused by nearby Bluetooth devices and other devices using the same frequency band. Adaptive Frequency Hopping (AFH) is introduced in the Bluetooth 1.2 specification and is useful if your device communicates through both Bluetooth and Wi-Fi simultaneously (e.g. a laptop computer with both Bluetooth and Wi-Fi equipment).

Every Bluetooth device is assigned a unique Bluetooth address, being a 48-bit hardware address equivalent to hardware addresses assigned to regular Network Interface Cards (NICs). The Bluetooth address is used not only for identification, but also for synchronizing the frequency hopping between devices and generation of keys in the Bluetooth security procedures.

3.1 Piconet and Scatternet

A piconet is the usual form of a Bluetooth network and is made up of one master and one or more slaves. The device initiating a Bluetooth connection automatically becomes the master. A piconet can consist of one master and up to seven active slaves. The master device is literally the master of the piconet. Slaves may only transmit data when transmission-time is granted by the master device, also slaves may not communicate directly with each other, and all communication must be directed through the master. Slaves synchronize their frequency hopping with the master using the master's clock and Bluetooth address.

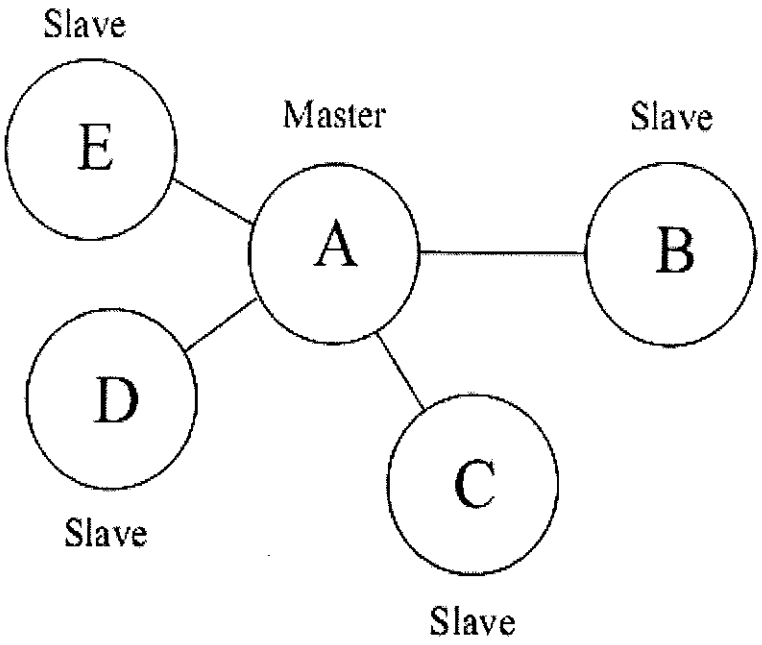


Fig-3.1 A typical piconet

Piconets take the form of a star network, with the master as the centre node; shown in Figure 3.1 Two piconets may exist within radio range of each other. Frequency hopping is not synchronized between piconets, hence different piconets will randomly collide on the same frequency. When

connecting two piconets the result will be a scatternet. Figure 3.2 shows an example, with one intermediate node connecting the piconets.

The intermediate node must time-share, meaning it must follow the frequency hopping in one piconet at the time. This reduces the amount of time slots available for data transfer between the intermediate node and a master; it will at least cut the transfer rate in half. It is also important to note that neither version 1.1 nor version 1.2 of the Bluetooth specifications define how packets should be routed between piconets. Hence, communication between piconets cannot be expected to be reliable.

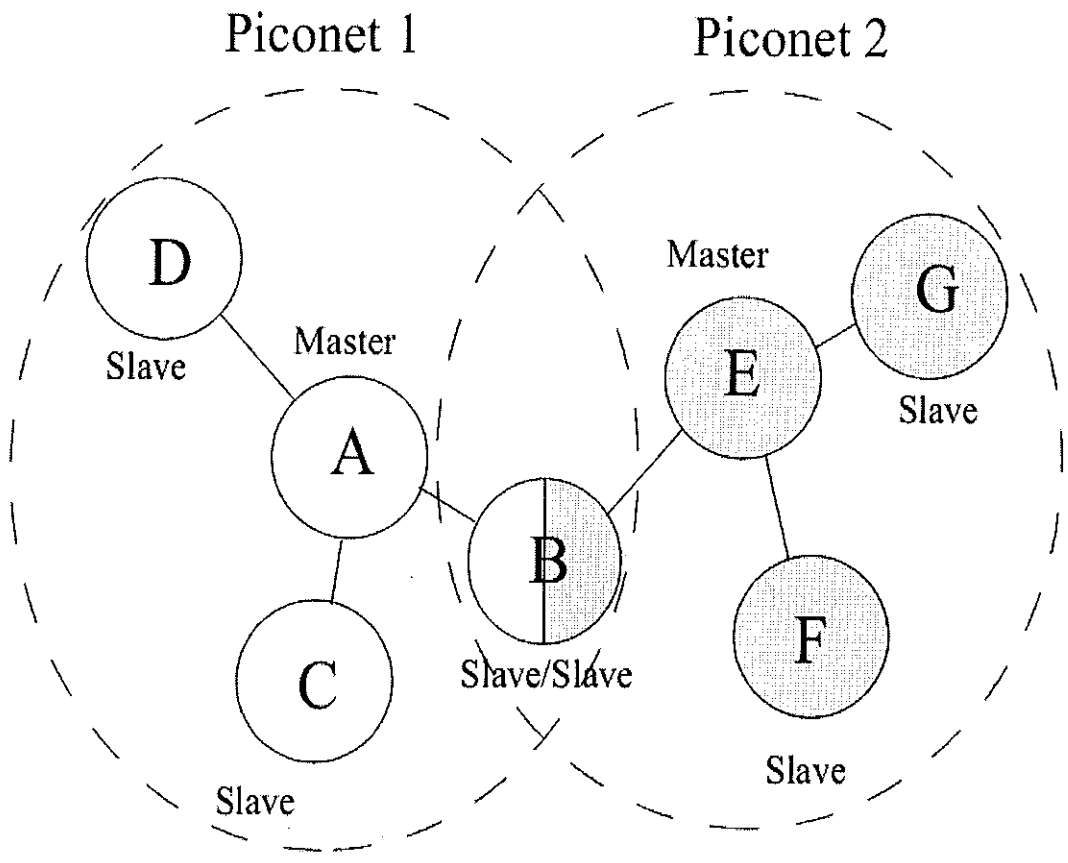


Fig-3.2 Scatternet

Role-switching enables two devices to switch roles in a piconet. Consider the following example: You have two devices A and B. Device A connects to device B, hence, device A becomes the master of the piconet consisting of devices A and B as shown in Figure 3.3.

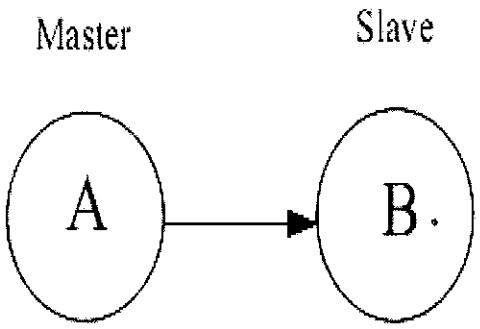


Fig-3.3 Piconet with 2 nodes

Then a device C wants to join the piconet. Device C connects to the master device, A. Since device C initiated the connection it will automatically become the master of the connection between device C and device A. We now have two masters; hence, we have two piconets. Device A is the intermediate node between these piconets, being the master for device B and the slave for device C, as seen in Figure 3.4

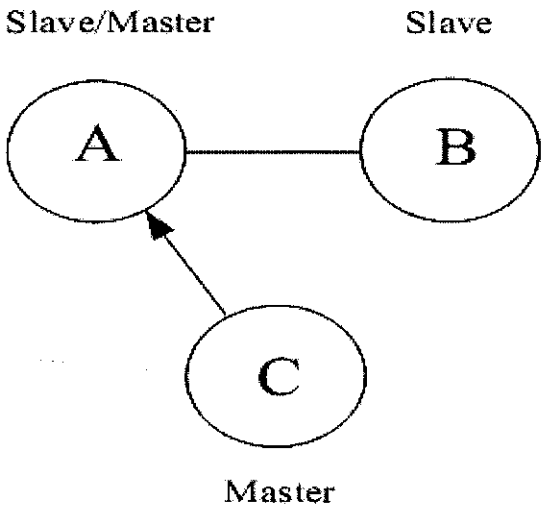


Fig-3.4 scatternet with 3 nodes

Figure 3.5 shows that a role-switch between device A and device C will give us one piconet where A is the master and both B and C are slaves. We see that when a new device wants to be part of a piconet we actually need a role-switch to make this happen, else we get a scatternet.

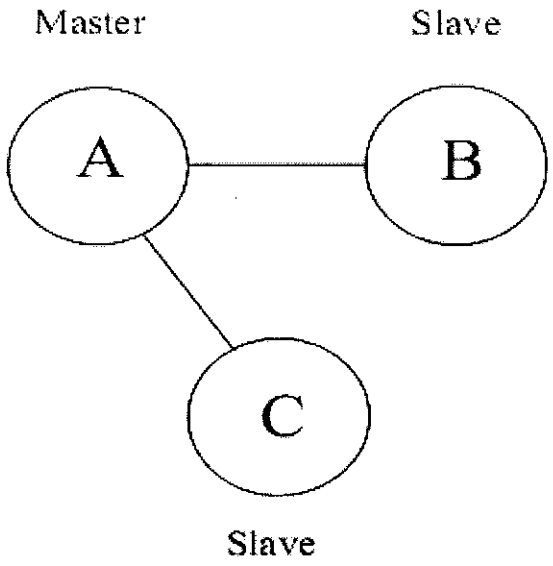


Fig-3.5 Piconet with 3 nodes

3.2 Bluetooth Architecture

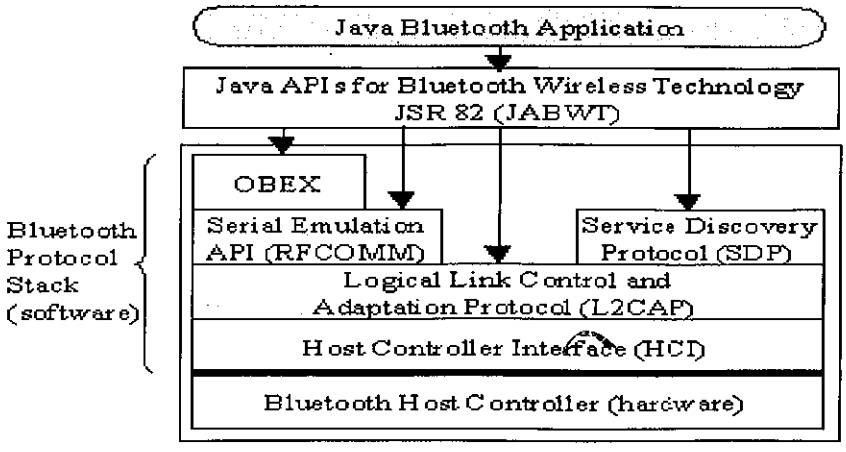


Fig-3.6 Bluetooth Stack

The Bluetooth stack is made up of many layers, as shown in Figure 3.6 The HCI is usually the layer separating hardware from software and is implemented partially in software and hardware/firmware. The layers below the HCI are usually implemented in hardware and the layers above the HCI are usually implemented in software.

L2CAP segments data into packets for transmission, and re-assembles received packets into larger messages.

RFCOMM provides functionality similar to a standard serial communications port, which is utilized as a stream connection at the Java level. I employ RFCOMM as the communications layer in this chapter's echoing client/server example.

OBEX (the Object Exchange protocol) provides a means for exchanging objects between objects (such as images and files), and is built on top of RFCOMM

SDP allows a server to register its services with the device, and is also employed by clients looking for devices and services. The Java Bluetooth API hides SDP behind a discovery API supported by a Discovery Agent class and Discovery Listener interface.

4. Client/Server Application Using Bluetooth

Bluetooth is a wireless technology for communication over distances of up to 10m, offering reasonably fast data transfer rates of around 1 Mb/s, principally between battery-powered devices. Bluetooth's primary intent is to support the creation of personal area networks (PANs) for small data transfers (or voice communication) between devices such as phones and PDAs.

shows how a server 'registers' itself as a Bluetooth server, and processes client connections and messages. A client searches for Bluetooth devices and services, connects to a matching server, and sends messages to it. Several clients can be connected to the server at once, with the server using a dedicated thread for each one

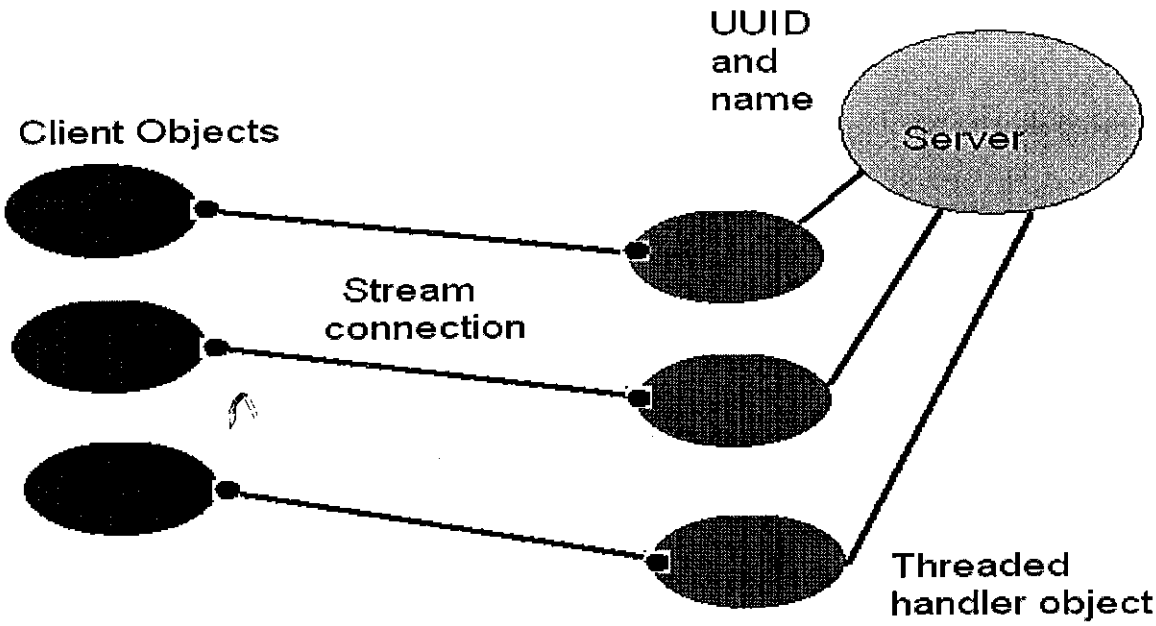


Fig 4.1 Client Server Model

As explained later, the echo server is identified by a Bluetooth UUID (a Universally Unique Identifier) and a service name. The stream connection between a client and handler is created using Bluetooth's RFCOMM protocol

4.1 Server

Server is a threaded RFCOMM-based service, identified by a UUID (a unique Bluetooth ID) and service name ("hacker service"). The waiting for client connections is done in a thread so that it doesn't cause the top level MIDlet to block. When a client does connect, a Threaded Echo Handler thread is spawned to deal with it

4.2 Service Initialization

```
private static final UUID MY_SERVICE_ID =
  new UUID("BAE0D0C0B0A000955570605040302010", false);


  public void run() {


    try {
      try{
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
          con = DriverManager.getConnection("jdbc:odbc:BlueServer");
        }
          catch(Exception ex){
            System.out.println(ex.getMessage());
          }
      // get local BT manager
```

```
            mLocalBT = LocalDevice.getLocalDevice();

            // set we are discoverable
            mLocalBT.setDiscoverable(DiscoveryAgent.GIAC);
            String url = "btspp://localhost:" + MY_SERVICE_ID.toString() +
            ";name=Hacker Service;authorize=false";

            // create notifier now
            mServerNotifier = (StreamConnectionNotifier) Connector.open(
            url.toString());
            //System.out.println(" get notifier ");

        } catch (Exception e) {
            System.err.println("Can't initialize bluetooth: " + e);
        }

mLocalBT=LocalDevice.getLocalDevice();
mLocalBT.setDiscoverable(DiscoveryAgent.GIAC);
```

The DiscoveryAgent.GIAC (General/Unlimited Inquiry Access Code) constant means that all remote devices (i.e. all the clients) will be able to find the device. There's also a DiscoveryAgent.LIAC constant which limits the device's visibility.

The RFCOMM stream connection offered by the server requires a suitably formatted URL. The basic format is:

btspp://<hostname>:<UUID>;<parameters>

I use localhost as the hostname, but any Bluetooth address can be employed.

The UUID field is a unique 128-bit identifier representing the service; I utilize a 32 digit hexadecimal string (each hex digit uses 4 bits).
The URL's parameters are "<name>=<value>" pairs, separated by semicolons. Typical <name> values are "name" for the service name (used here), and security parameters such as "authorize".

The creation of the StreamConnectionNotifier instance, server, by the Connector.open () call also generates an implicit service record. The record is a description of the Bluetooth service as a set of (id, value) attributes. It can be accessed by calling LocalDevice.getRecord():

ServiceRecord record = local.getRecord(server);

The ServiceRecord class offers get/set methods for accessing and changing a record'sattributes.

4.3 Connecting to the Client


```
DataInputStream dis = null;
    DataOutputStream dos=null;
```

```
try {
        dis=conn.openDataInputStream();
        dos=conn.openDataOutputStream();
        String read=dis.readUTF();
```

It's possible to map a Data Input Stream and Data Output Stream to the Stream Connection instance, so that basic Java data types (e.g. integers, floats, doubles, and strings) can be read and written. I've used Input Stream and Output Stream because their byte-based read() and write() methods can be easily utilized as 'building blocks' for implementing different forms of message processing

4.4 Devices Discovery

Device discovery is the first step required when browsing nearby Bluetooth devices. When we have discovered nearby devices we can find out which services they offer.

The state diagram lists three methods that play important roles in the device search:

DiscoveryAgent.startInquiry(),

DiscoveryListener.deviceDiscovered(),

DiscoveryListener.inquiryCompleted()

The vector is used during the search to hold devices information in the form of RemoteDevice objects. DiscoveryAgent.startInquiry() is non-blocking, so the system communicates its progress by calling DiscoveryListener.deviceDiscovered(), and DiscoveryListener.inquiryCompleted(). ServiceFinder implements the DiscoveryListener interface, and a reference to it (this) is passed

to startInquiry() as its second argument. This means that ServiceFinder's deviceDiscovered() and inquiryCompleted() methods will be called during the search.

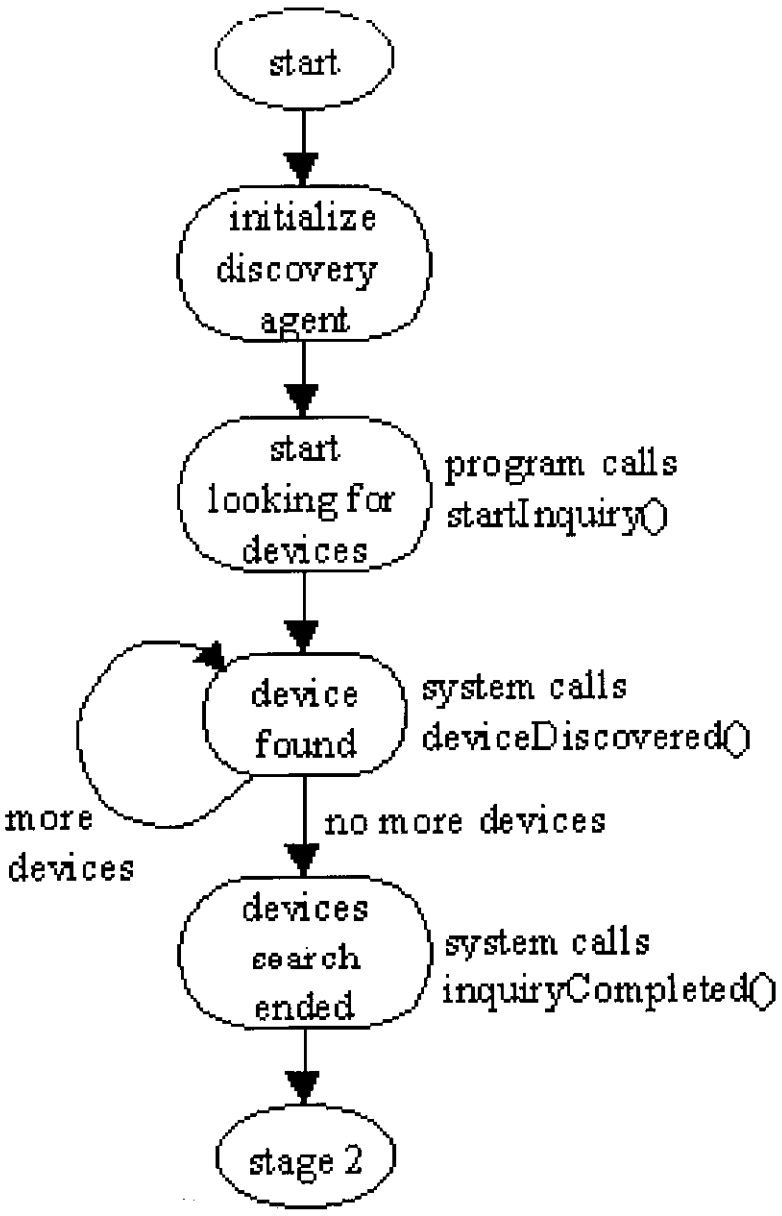


Fig-4.2 Device Discovery State Diagram

4.5 Service Search

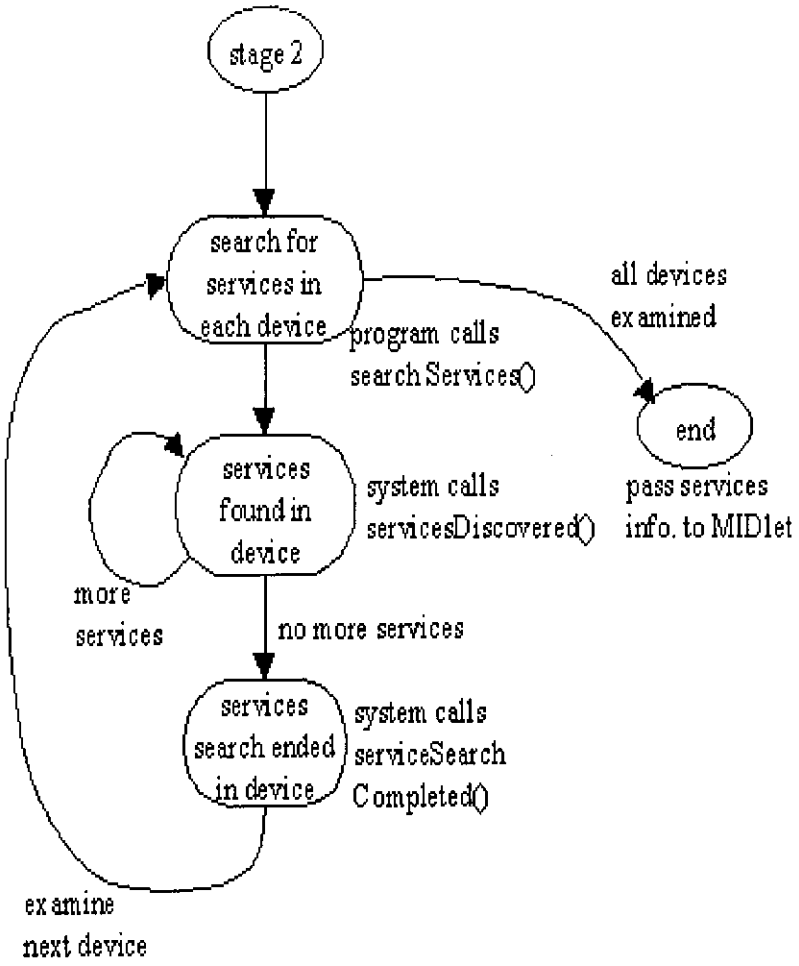


Fig-4.3 Services Search State Diagram

The state diagram consists of two loops: the outer loop cycles through each device found during the devices search, while the inner loop checks each service offered by a particular device. The figure mentions three methods that play important roles in services search:

DiscoveryAgent.searchServices()
DiscoveryListener.servicesDiscovered(), and
DiscoveryListener.serviceSearchCompleted().

4.6 Applications Developed

4.6.1 *Dictionary*

User will type the word (of which meaning is required), on the GUI of the application installed on mobile phone. Database is searched, after Bluetooth connection is established, for the meaning of that particular word. If found the meaning will be sent back to mobile screen and if not, a sorry message will flash on the screen.

4.6.2 PC Control

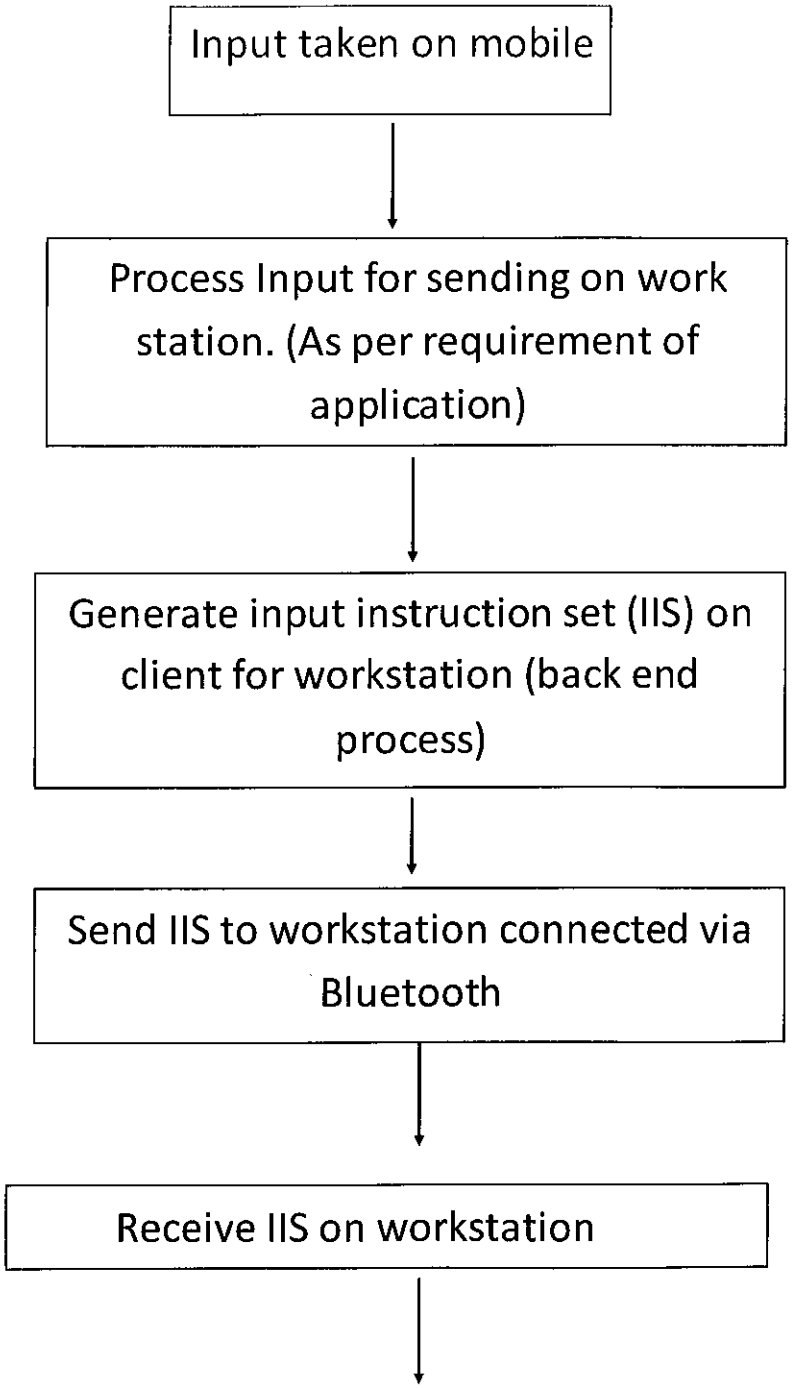
User will choose from the set of certain applications or processes to be invoked at workstation end. We have implemented notepad, shut down , restart.

4.6.3 *File Search*

User will type the name of the file (be it jpeg, mp3, wave file anything and everything that can run on mobile end), on the GUI of application installed on mobile phone and that file will be searched on the workstation. It exits, it will be sent back to the mobile and if not sorry message will flash on mobile phone

In all the three applications the basic step of establishing a connection will remain the same.

4.7 Flowchart of the project



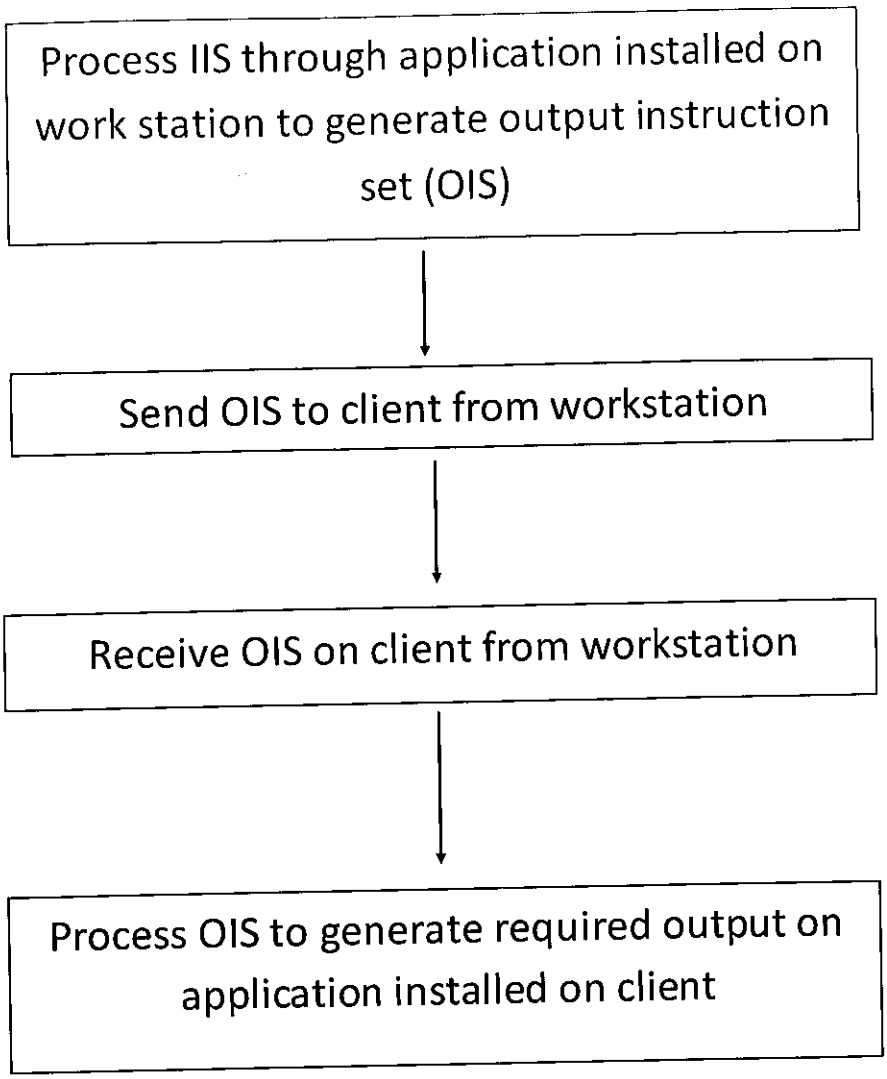
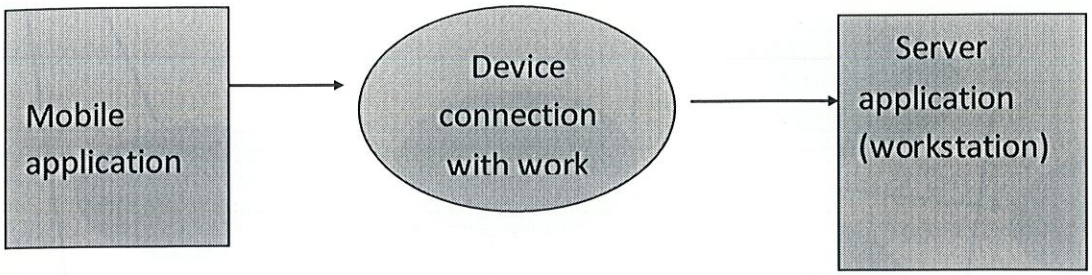


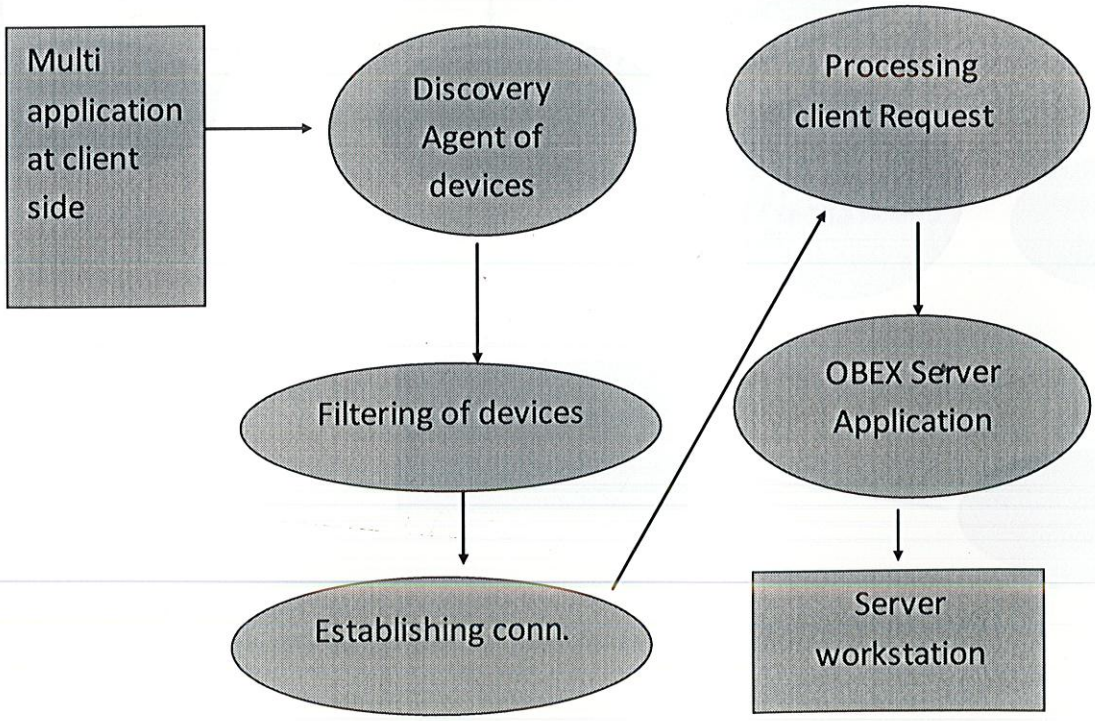
Fig-4.4 Graphical Representation

4.8 Data Flow Diagram

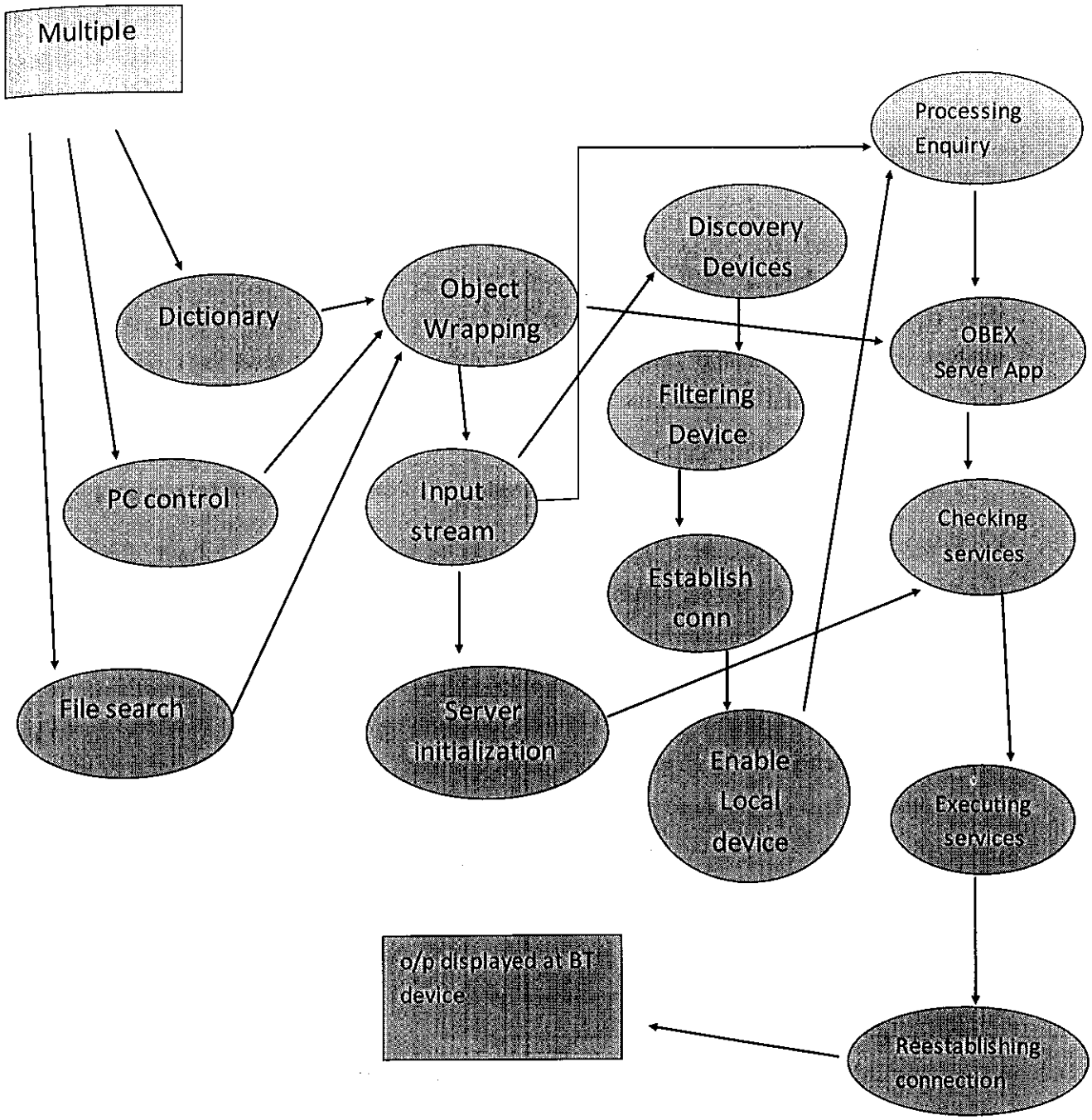
4.8.1 DFD level 0



4.8.2 DFD level 1



4.8.3 DFD level 2



5. Testing technologies to be used

5.1 Black Box Testing

Black box testing assumes the code to be a black box that responds to input stimuli. The testing focuses on the output to various types of stimuli in the targeted deployment environments. It focuses on validation tests, boundary conditions, destructive testing, reproducibility tests, performance tests, globalization, and security-related testing.

Risk analysis should be done to estimate the amount and the level of testing that needs to be done. Risk analysis gives the necessary criteria about when to stop the testing process. Risk analysis prioritizes the test cases. It takes into account the impact of the errors and the probability of occurrence of the errors. By concentrating on the test cases that can lead to high impact and high probability errors, the testing effort can be reduced and the application block can be ensured to be good enough to be used by various applications.

Preferably, black box testing should be conducted in a test environment close to the target environment. There can be one or more deployment scenarios for the application block that is being tested. The requirements and the behavior of the application block can vary with the deployment scenario; therefore, testing the application block in a simulated environment that closely resembles the deployment environment ensures that it is tested to satisfy all requirements of the targeted real-life conditions. There will be no surprises in the production environment. The test cases being executed ensure robustness of the application block for the targeted deployment scenarios.

For example, the CMAB can be deployed on the desktop with Windows Forms applications or in a Web farm when integrated with Web applications. The CMAB requirements, such as performance objectives, vary from the desktop environment to the Web environment. The test cases and the test environment have to vary according to the target environments. Other application blocks may have more restricted and specialized target environments. An example of an application block that requires a specialized test environment is an application block that is deployed on mobile devices and is used for synchronizing data with a central server.

Black Box Testing Steps

Black box testing involves testing external interfaces to ensure that the code meets functional and nonfunctional requirements. The various steps involved in black box testing are the following:

1. **Create test plans:** Create prioritized test plans for black box testing.

2. **Test the external interfaces:** Test the external interfaces for various type of inputs using automated test suites, such as NUnit suites and custom prototype applications.

3. **Perform load testing:** Load test the application block to analyze the behavior at various load levels. This ensures that it meets all performance objectives that are stated as requirements.

4. **Perform stress testing:** Stress tests the application block to analyze various bottlenecks and to identify any issues visible only under extreme load conditions, such as race conditions and contentions.

5. **Perform security testing:** Test for possible threats in deployment scenarios. Deploy the application block in a simulated target environment and try to hack the application by exploiting any possible weakness of the application block.

6. **Perform globalization testing:** Execute test cases to ensure that the application block can be integrated with applications targeted toward locales other than the default locale used for development. The next sections describe each of these steps.

5.1.1 Step 1: Create Test Plans

The first step in the process of black box testing is to create prioritized test plans. You can prepare the test cases for black box testing even before you implement the application block. The test cases are based on the requirements and the functional specification documents. The requirements and functional specification documents help you extract various usage scenarios and the expected output in each scenario.

The detailed test plan document includes test cases for the following:

- Testing the external interfaces with various types of input
- Load testing and stress testing
- Security testing
- Globalization testing

5.1.2 Step 2: Test the External Interfaces

You need to test the external interfaces of the application block using the following strategies:

Ensure that the application block exposes interfaces that address all functional specifications and requirements. To perform this validation testing, do the following:

1. Prepare a checklist of all requirements and features that are expected from the application block.

2. Create test harnesses, such as NUnit, and small "hello world'" applications to use all exposed APIs of the test application block.

3. Run the test harnesses. Testing for various types of inputs. After ensuring that the application block exposes the interfaces that address all of the functional specifications, you need to test the robustness of these interfaces. You need to test for the following input types:

- Randomly generated input within a specified range
- Boundary cases for the specified range of input
- The number zero testing if the input is numeric
- The null input
- Invalid input or input that is out of the expected range

5.1.3 Step 3: Perform Load Testing

Use load testing to analyze the application block behavior under normal and peak load conditions. Load testing allows you to verify that the application block can meet the desired performance objectives and does not overshoot the allocated budget for resource utilization such as memory, processor, and network I/O. The requirements document usually lists the resource utilization. You can measure metrics related to response times, throughput rates, and so on, for the load test. In addition, you can measure other metrics that help you identify any potential bottlenecks.

To load test an application block, you need to develop a sample application that is an accurate prototype of applications that will be used in the target environment. In the case of the CMAB, and because one of the deployment scenarios is the Web environment, a simple Web application can be developed that uses the application block for reading and writing configuration information. Preferably, this application block should be tested in clustered and non-clustered environments because deploying in a Web farm is one of the deployment scenarios. On budget for the application block and the workload it should be able to support.

5.1.4 Step 4: Perform Stress Testing

Use stress testing to evaluate the application block's behavior when it is pushed beyond the normal or peak load conditions. The expectation from the system beyond load conditions is to either return expected output or return meaningful error messages to the user without corrupting the integrity of any data. The goal of stress testing is to discover bugs that surface only under high load conditions, such as synchronization issues, race conditions, and memory leaks.

The data that is collected in stress testing is based on the input from load testing and the code review. The code review identifies the potential areas in code that may lead to the preceding issues. The metrics collected in load testing also provides input for identifying the scenarios that need to be stress tested. For example, if during load testing, you observe that the application

starts to show increased response times for increased load conditions when writing to SQL Server, you should check for any potential issues because of concurrency.

5.1.5 Step 5: Perform Security Testing

Black box security testing the application block identifies security vulnerabilities within the application block by treating it as an independent unit. The testing is done at run time. The purpose is to forcefully break the interfaces of the application block, intercept sensitive data within the block, and so on. Sample test harnesses can be used to create a deployment scenario for the application block.

Depending on the functionality the application block provides, test cases can be identified. Examples of test cases and tests can be the following:

If the application block accepts data from a user, make sure it validates the input data by creating test cases to pass different types of data, including unsafe data, through the application block's interfaces and confirming that the application block is able to stop it and handle it by providing appropriate error messages. If the application block accesses any secure resources, such as the registry or file system, identify test cases that can test for threats resulting from elevated privileges. If the application block handles secure data and uses cryptography, scenarios can be developed for simulating various types of attacks to access the data. This tests and ensures that the appropriate algorithms and methods are used to secure data.

5.1.6 Step 6: Perform Globalization Testing

The goal of globalization testing is to detect potential problems in the application block that could inhibit its successful integration with an application that uses culture resources different than the default culture resources used for development. Globalization testing involves passing culture-specific input to a sample application integrating the application block. It makes sure that the code can handle all international support and supports any culture or locale settings without breaking functionality that would cause data loss.

To perform globalization testing, you must install multiple language groups and set the culture or locale to different cultures or locales, such as Japanese or German, from the local culture or Software Testing Phase of Current Project

Software testing is the process used to assess the quality of computer software. Software testing is an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test , with respect to the context in which it is intended to operate. This includes, but is not limited to, the process of executing a program or application with the intent of finding software bugs. Quality is not an absolute; it is value to some person.

With that in mind, testing can never completely establish the correctness of arbitrary computer software; testing furnishes a criticism or comparison that compares the state and behaviour of the product against a specification. An important point is that software testing should be distinguished from the separate discipline of Software Quality Assurance (S.Q.A.), which encompasses all business process areas, not just testing. Over its existence, computer software has continued to grow in complexity and size. Every software product has a target audience. For example, a video game software has its audience completely different from banking software.

Therefore, when an organization develops or otherwise invests in a software product, it must assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. Software testing is the process of attempting to make this assessment.

5.2 Integration testing

Integration testing (sometimes called Integration and 'resting, abbreviated I&T) is the phase of software testing in which individual software modules are combined and tested as a group. It follows unit testing and precedes system testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for

system testing The purpose of integration testing is to verify functional, performance and reliability requirements placed on major design items.

These "design items", i.e. assemblages (or groups of units), are exercised through their interfaces using black box testing, success and error cases being simulated via appropriate parameter and data inputs. Simulated usage of shared data and inter-process communication is tested and individual subsystems are exercised through it input interface. Test cases are constructed to test that all components within assemblages interact correctly, for example across procedure calls or process activations, and this is done after testing individual modules, i.e. unit testing. The overall idea is a "building block" approach, in verified assemblages are added to a verified base which is then used to support the ration testing of further assemblages. Some different types of integration testing are big , top-down, and bottom-up

.

5.3 Unit-testing

Computer programming, unit testing is a procedure used to validate that individual units of :e code are working properly. A unit is the smallest testable part of an application. In Aura] programming a unit may be an individual program, function, procedure, etc., while in :t-oriented programming, the smallest unit is a method, which may belong to a base/super , abstract class or derived/child class. Ideally, each test case is independent from the others) objects like stubs, mock or fake objects as well as test harnesses can be used to assist in a module in isolation. Unit testing is typically done by software developers to ensure that they have written meets software requirements and behaves as the developer intended.

Appendix

Sample codes:

I. Server

```
package bluetoothserver;

import java.io.*;

import javax.bluetooth.*;

import javax.microedition.io.*;

import java.sql.*;

import java.util.StringTokenizer;

import java.util.List;

public class Server implements Runnable {

    private Thread mServer = null;

    java.sql.Connection con;

    private LocalDevice mLocalBT;

    private boolean mEndNow;

    public String messageToBeSent="";

    private StreamConnectionNotifier mServerNotifier;

    private static final UUID MY_SERVICE_ID =

    new UUID("BAE0D0C0B0A000955570605040302010", false);
```

```
    public void run() {

        try {
            try{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                    con =
DriverManager.getConnection("jdbc:odbc:BlueServer");
                }
                    catch(Exception ex){
                        System.out.println(ex.getMessage());
                    }
            // get local BT manager
            mLocalBT = LocalDevice.getLocalDevice();

            // set we are discoverable
            mLocalBT.setDiscoverable(DiscoveryAgent.GIAC);
            String url = "btspp://localhost:" +
MY_SERVICE_ID.toString() +
            ";name=Hacker Service;authorize=false";

            // create notifier now
```

```
            mServerNotifier      =      (StreamConnectionNotifier)
Connector.open(
            url.toString());
            //System.out.println(" got notifier ");
        } catch (Exception e) {
            System.err.println("Can't initialize bluetooth: " +
e);
        }
        StreamConnection conn = null;
        while (!mEndNow) {
            conn = null;
            try {
                conn = mServerNotifier.acceptAndOpen();
            } catch (IOException e) {

                continue;
            }
            if (conn != null)
                processRequest(conn);
        }
    }
```

```
public  void startServer() {
    if (mServer != null)
        return;
    // start receive thread
    mServer = new Thread(this);
    mServer.start();

}
private void processRequest(StreamConnection conn) {
    DataInputStream dis = null;
    DataOutputStream dos=null;
    try {
            dis=conn.openDataInputStream();
            dos=conn.openDataOutputStream();
            String read=dis.readUTF();
            if(read.toUpperCase().startsWith("STUDENT")){
                read=read.substring(read.indexOf(":")+1);
                Statement st=con.createStatement();
                ResultSet  rs=st.executeQuery("select  *  from
students where admno='" + read + "'");
                if(rs.next()){
                    String info="";
```

```
                    info="Name: " + rs.getString("sname") +
", Class: "+ rs.getString("Class") + ", Roll: " +
rs.getString("Roll");
                    dos.writeUTF(info);
                }
                else{
                    dos.writeUTF("Sorry, no result found!");
                }
            }
            dos.flush();
            dos.close();
            dis.close();
                //conn.close();
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }

    }
}
```

II. Remotediscovery.java

```
package bluetoothserver;

import java.io.IOException;

import java.util.Vector;

import javax.bluetooth.*;


/**

 * Minimal Device Discovery example.

 */

public class RemoteDeviceDiscovery {


    public static final Vector/*<RemoteDevice>*/
devicesDiscovered = new Vector();


    public static void main(String[] args) throws IOException,
InterruptedException {


        final Object inquiryCompletedEvent = new Object();


        devicesDiscovered.clear();

        DiscoveryListener listener = new DiscoveryListener() {
```

```
        public void deviceDiscovered(RemoteDevice btDevice,
DeviceClass cod) {
                System.out.println("Device " +
btDevice.getBluetoothAddress() + " found");
                devicesDiscovered.addElement(btDevice);
                try {
                    System.out.println("name " +
btDevice.getFriendlyName(false));
                } catch (IOException cantGetDeviceName) {
                }
            }
            public void inquiryCompleted(int discType) {
                System.out.println("Device Inquiry completed!");
                synchronized(inquiryCompletedEvent){
                    inquiryCompletedEvent.notifyAll();
                }
            }
            public void serviceSearchCompleted(int transID, int
respCode) {
            }
            public void servicesDiscovered(int transID,
ServiceRecord[] servRecord) {
            }
```

```
        };
synchronized(inquiryCompletedEvent) {
            boolean                    started                    =
LocalDevice.getLocalDevice().getDiscoveryAgent().startInquiry(Di
scoveryAgent.GIAC, listener);
            if (started) {
                System.out.println("wait  for  device  inquiry  to
complete...");
                inquiryCompletedEvent.wait();
                System.out.println(devicesDiscovered.size() +   "
device(s)  found");
            }
}
}
```

III. Logger.java

```
package bluetoothserver;

public class Logger {

static void debug(String message) {

        System.out.println(message);}

       static void debug(String message, Object o) {

       System.out.println(message + " " + o);

        }

        static void debug(String message, Throwable e) {

         System.out.println(message + " " + e.getMessage());

          e.printStackTrace();

        }

        static void debug(Throwable e) {

                System.out.println(e.getMessage());

                e.printStackTrace();

        }

        static void error(String message) {

                System.out.println(message); }

        static void error(String message, Throwable e) {

                System.out.println(message + " " + e.getMessage());

                e.printStackTrace()    }
```

IV. Servicesearch.java

```
package bluetoothserver;

import java.io.IOException;

import java.util.Enumeration;

import java.util.Vector;

import javax.bluetooth.*;


public class ServicesSearch {


    static final UUID OBEX_FILE_TRANSFER = new UUID(0x1106);


    public static final Vector/*<String>*/ serviceFound = new
Vector();


    public static void main(String[] args) throws IOException,
InterruptedException {


        // First run RemoteDeviceDiscovery and use discoved
device

        RemoteDeviceDiscovery.main(null);
```

```
        serviceFound.clear();
        String deviceName="";
        UUID serviceUUID = OBEX_FILE_TRANSFER;
        if ((args != null) && (args.length > 0)) {
            //serviceUUID = new UUID(args[0], false);
            deviceName=args[0];
        }

        final Object serviceSearchCompletedEvent = new Object();

        DiscoveryListener listener = new DiscoveryListener() {
            public void deviceDiscovered(RemoteDevice btDevice,
DeviceClass cod) {
            }
            public void inquiryCompleted(int discType) {
            }
            public void servicesDiscovered(int transID,
ServiceRecord[] servRecord) {
                //for (int i = 0; i < servRecord.length; i++) {
                for (int i = 0; i < 1; i++) {

Stringurl=servRecord[i].getConnectionURL(ServiceRecord.NOAUTHENT
ICATE_NOENCRYPT, false);
```

```
                    if (url == null) {
                        continue;
                    }
                    serviceFound.add(url);
                    DataElement serviceName =
servRecord[i].getAttributeValue(0x0100);
                    if (serviceName != null) {
                        System.out.println("service " +
serviceName.getValue() + " found " + url);
                    } else {
                        System.out.println("service found " +
url);
                    }
                }
            }
            public void serviceSearchCompleted(int transID, int
respCode) {
                System.out.println("service search completed!");
                synchronized(serviceSearchCompletedEvent){
                    serviceSearchCompletedEvent.notifyAll();
                }
            }
        };
```

```
        UUID[] searchUuidSet = new UUID[] { serviceUUID };
        int[] attrIDs =  new int[] {
                0x0100 // Service name
        };
        for(Enumeration en =
RemoteDeviceDiscovery.devicesDiscovered.elements();
en.hasMoreElements(); ) {
            RemoteDevice btDevice =
(RemoteDevice)en.nextElement();
            synchronized(serviceSearchCompletedEvent) {

if(btDevice.getFriendlyName(false).equals(deviceName)){

                    System.out.println("search services on " +
btDevice.getBluetoothAddress() + " " +
btDevice.getFriendlyName(false));

LocalDevice.getLocalDevice().getDiscoveryAgent().searchServices(
attrIDs, searchUuidSet, btDevice, listener);
                    serviceSearchCompletedEvent.wait()}}}}}
```

v. SearchInfo.java

```
import java.io.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;
import javax.microedition.io.*;

public class SearchInfo extends MIDlet implements
CommandListener {

    private boolean midletPaused = false;
    private Thread mClientThread = null;
    private boolean mEndNow = false;
    private DiscoveryAgent mDiscoveryAgent = null;
    private String mConnect = null;
    private String lastMessage="";
    StreamConnection conn  = null;
    DataInputStream dis=null;
    DataOutputStream dos=null;
    private static final UUID MY_SERVICE_ID =
        new UUID("BAE0D0C0B0A00095557060504030201O", false);
```

```
    //<editor-fold   defaultstate="collapsed"   desc="   Generated
Fields ">
    private Form form;
    private TextField txtSearch;
    private StringItem lblResult;
    private Command okCommand;
    private Command exitCommand;
    //</editor-fold>
    /**
     * The SearchInfo constructor.
     */
    public SearchInfo() {
        while(mDiscoveryAgent==null){
        try {
            mDiscoveryAgent                                    =
LocalDevice.getLocalDevice().getDiscoveryAgent();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
        }
    }
```

```
    //<editor-fold defaultstate="collapsed" desc=" Generated
Methods ">
    //</editor-fold>

    //<editor-fold defaultstate="collapsed" desc=" Generated
Method: initialize ">
    /**
     * Initilizes the application.
     * It is called only once when the MIDlet is started. The
method is called before the <code>startMIDlet</code> method.
     */
    private void initialize() {
        // write pre-initialize user code here

        // write post-initialize user code here
    }
    //</editor-fold>

    //<editor-fold defaultstate="collapsed" desc=" Generated
Method: startMIDlet ">
    /**
```

```
     * Performs an action assigned to the Mobile Device - MIDlet
Started point.
     */
    public void startMIDlet() {
        // write pre-action user code here
        switchDisplayable(null, getForm());
        // write post-action user code here
    }
    //</editor-fold>

    //<editor-fold defaultstate="collapsed" desc=" Generated
Method: resumeMIDlet ">
    /**
     * Performs an action assigned to the Mobile Device - MIDlet
Resumed point.
     */
    public void resumeMIDlet() {
        // write pre-action user code here

        // write post-action user code here
    }
    //</editor-fold>
```

```
    //<editor-fold defaultstate="collapsed" desc=" Generated
Method: switchDisplayable ">

    /**

     * Switches a current displayable in a display. The
<code>display</code> instance is taken from
<code>getDisplay</code> method. This method is used by all
actions in the design for switching displayable.

     * @param alert the Alert which is temporarily set to the
display; if <code>null</code>, then <code>nextDisplayable</code>
is set immediately

     * @param nextDisplayable the Displayable to be set

     */

    public void switchDisplayable(Alert alert, Displayable
nextDisplayable) {

        // write pre-switch user code here

        Display display = getDisplay();

        if (alert == null) {

            display.setCurrent(nextDisplayable);

        }

else

{

            display.setCurrent(alert, nextDisplayable);

        }
```

```
        // write post-switch user code here
    }
    //</editor-fold>

    //<editor-fold defaultstate="collapsed" desc=" Generated Method: commandAction for Displayables ">
    /**
     * Called by a system to indicated that a command has been invoked on a particular displayable.
     * @param command the Command that was invoked
     * @param displayable the Displayable where the command was invoked
     */
    public void commandAction(Command command, Displayable displayable) {
        // write pre-action user code here
        if (displayable == form) {
            if (command == exitCommand) {
                // write pre-action user code here

                // write post-action user code here
                this.exitMIDlet();
            } else if (command == okCommand) {
```

```
                // write pre-action user code here

                // write post-action user code here
                searchDict();
            }
        }
        // write post-action user code here
    }
    //</editor-fold>

    //<editor-fold defaultstate="collapsed" desc=" Generated Getter: form ">
    /**
     * Returns an initiliazed instance of form component.
     * @return the initialized component instance
     */
    public Form getForm() {
        if (form == null) {
            // write pre-init user code here
            form = new Form("Search Student Info.", new Item[] { getTxtSearch(), getLblResult() });
            form.addCommand(getOkCommand());
```

```
            form.addCommand(getExitCommand());
            form.setCommandListener(this);
            // write post-init user code here
        }
        return form;
    }
    //</editor-fold>

    //<editor-fold defaultstate="collapsed" desc=" Generated Getter: txtSearch ">
    /**
     * Returns an initiliazed instance of txtSearch component.
     * @return the initialized component instance
     */
    public TextField getTxtSearch() {
        if (txtSearch == null) {
            // write pre-init user code here
            txtSearch = new TextField("Admsn No:", null, 32, TextField.ANY);
            // write post-init user code here
        }
        return txtSearch;
```

```
    }
    //</editor-fold>
    //<editor-fold defaultstate="collapsed" desc=" Generated
Getter: okCommand ">
    /**
     * Returns an initiliazed instance of okCommand component.
     * @return the initialized component instance
     */
    public Command getOkCommand() {
        if (okCommand == null) {
            // write pre-init user code here
            okCommand = new Command("Ok", Command.OK, 0);
            // write post-init user code here
        }
        return okCommand;
    }
    //</editor-fold>
    //<editor-fold defaultstate="collapsed" desc=" Generated
Getter: lblResult ">
    /**
     * Returns an initiliazed instance of lblResult component.
     * @return the initialized component instance
```

```
     */
    public StringItem getLblResult() {
        if (lblResult == null) {
            // write pre-init user code here
            lblResult = new StringItem("Result:", "");
            // write post-init user code here
        }
        return lblResult;
    }
    //</editor-fold>
    //<editor-fold defaultstate="collapsed" desc=" Generated
Getter: exitCommand ">
    /**
     * Returns an initiliazed instance of exitCommand component.
     * @return the initialized component instance
     */
    public Command getExitCommand() {
        if (exitCommand == null) {
            // write pre-init user code here
            exitCommand = new Command("Exit", Command.EXIT, 0);
            // write post-init user code here
        }
```

```
        return exitCommand;
    }
    //</editor-fold>
    /**
     * Returns a display instance.
     * @return the display instance.
     */
    public Display getDisplay () {
        return Display.getDisplay(this);
    }
    class ShowMessage implements Runnable {
        Display disp = null;
        String message = null;
        public ShowMessage(String mess) {
            try{
                message = mess;
            }
            catch(Exception ex){
                //ignore
            }
        }
```

```
        public void run() {
            if(message!=null)
                lblResult.setText(message);
        }
    }
    void searchDict(){
        try  {
                mConnect =
mDiscoveryAgent.selectService(MY_SERVICE_ID,
                ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);

                String readMessage="";
                if (mConnect != null)  {
                    conn = (StreamConnection)
Connector.open(mConnect);
                    dis=conn.openDataInputStream();
                    dos=conn.openDataOutputStream();
                    dos.writeUTF("STUDENT:" +
txtSearch.getString());
                    dos.close();
                    readMessage=dis.readUTF();
                    dis.close();
```

```
                }
                else{
                    lblResult.setText("Unable to connect!");
                }
                conn.close();
                lastMessage=readMessage;
                getDisplay().callSerially(new
ShowMessage(lastMessage));
            } catch (Exception ex) {
                System.err.println(ex.getMessage());
            }
    }
    /**
     * Exits MIDlet.
     */
    public void exitMIDlet() {

        switchDisplayable (null, null);
        destroyApp(true);
        notifyDestroyed();
    }
    /**
```

```
     * Called when MIDlet is started.
     * Checks whether the MIDlet have been already started and
initialize/starts or resumes the MIDlet.
     */
    public void startApp() {
        if (midletPaused) {
            resumeMIDlet ();
        } else {
            initialize ();
            startMIDlet ();
        }
        midletPaused = false;
    }
    /**
     * Called when MIDlet is paused.
     */
    public void pauseApp() {
        midletPaused = true;
    }
    /**
     * Called to signal the MIDlet to terminate.
```

```
     * @param unconditional if true, then the MIDlet has to be
unconditionally terminated and all resources has to be released.
     */
    public void destroyApp(boolean unconditional) {}
}
```

OPERATING ENVIRONMENT

Software Interface

- Technologies to be used:
- J2ME for designing the application
- Database

  -MS sql

Hardware Interface

Processor: P I at 233 MHz

RAM: 128 MB

Disk Space: 512MB

PC/Laptop: bluetooth enabled

Mobile Device- S40 series bluetooth enabled

BIBLIOGRAPHY

- Andrew Davison , Java Programming Techniques for Games 2005.
- Robert Virkus, J2ME Polish 2005
- Sing Li and Jonathan Knudsen, Beginning J2ME from novice to professional, 3rd Edition, Apress, 2005
- John W. Muchow , Core J2ME Technology & MIDP, Sun microsystems, 2001
- David Kammer, Gordon McNutt, Brian Senese ,Bluetooth Application Developer's Guide, Syngress, 2002
- Qusay Mahmoud, Learning Wireless Java, O'Reilly, 2001
- Tremblett,Paul, Instant Wireless Java With J2ME, First Edition, Osborne,2002
- foroum.java.sun.com
- wireless.java.sun.com
- http://www.java2s.com/Code/Java/Class/CatalogClass.htm
- http://www.roseindia.net/
- http://netbeans.org/kb/trails/mobility.html