**INTRODUCTION:**

As per the COMP 206 course requirement of the Second year first semester of Computer Engineering, we created a Java Bluetooth application using JAVA programming language. Our project “JAVA BLUETOOTH APLLICATION” uses the wireless technology for establishing the connection between the client and the server.

Our software is specifically based on the concept of Bluetooth connection. Bluetooth is a low-cost, short-range wireless technology that has become popular among those who want to create *personal area networks (PANS).*Each PAN is a dynamically created network build around an individual, which enables devices such as cellular phones and personal digital assistants (PDAs) to connect automatically and share data immediately. Bluetooth enabled application can be either a server or a client.

**The Java language:**

Java is a fully functional programming language, more easily compared to C, C++, or Visual Basic in its capabilities than it is to JavaScript, VBScript, or HTML, which might be surprising to anyone who considers Java only an Internet development tool. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities.

Java is *platform independent* because you can run Java programs on any operating system without having to rewrite or recompile them for each system

More specifically, Java is a high-level programming language. A *high-level programming language* uses instructions that more closely resemble a written language (such as English) than machine language. High-level languages are much easier to understand.

**Bluetooth Wireless Technology:**

Bluetooth wireless technology was originally developed as a cable replacement technology for connecting devices such as mobile phone handsets, headsets, and portable computers with each other.

This wireless technology is used 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 adhoc personal area networks (PANs) for small data transfers (or voice communication) between devices such as phones and PDAs. Bluetooth wireless technology supports both voice and data, allowing devices to communicate either type of content.

### *Java APIs for Bluetooth Wireless Technology*

### The Java APIs for Bluetooth do not implement the Bluetooth specification, but rather provide a set of APIs to access and control a Bluetooth-enabled device. JSR 82 concerns itself primarily with providing Bluetooth capabilities to J2ME-enabled devices. ..

JSR 82 consists of two optional packages: the core Bluetooth API and the Object Exchange (OBEX) API. The latter is transport-independent and can be used without the former.

The Java APIs for Bluetooth target devices with the following characteristics:

* 512K minimum of total memory available (ROM and RAM) (application memory requirements are additional)
* Bluetooth wireless network connection
* Compliant implementation of the J2ME Connected Limited Device Configuration (CLDC)

#### Packages

The Java APIs for Bluetooth define two packages that depend on the CLDC javax.microedition package:

* javax.bluetooth: core Bluetooth API
* javax.obex: APIs for the Object Exchange (OBEX) protocol

**MIDlets**

All applications for the *Mobile Information Device Profile* (MIDP) must be derived from a special class, MIDlet. The MIDlet class manages the life cycle of the application. It is located in the package javax.microedition.midlet. MIDlets can be compared to J2SE applets and can exist in four different states: loaded,active, paused, and destroyed.

The Application Manager is responsible to manage the MIDlets' life cycle

### 

### Figure: Bluetooth and J2ME MIDP

### Anatomy of a Bluetooth application

### Stack initialization

### Device management

### Device discovery

### Service discovery

### Communication.

#### Stack Initialization

The Bluetooth stack is responsible for controlling the Bluetooth device, so you need to initialize the Bluetooth stack before you can do anything else. The initialization process comprises a number of steps whose purpose is to get the device ready for wireless communication.

****

**Figure: Bluetooth Stack**

**Device Manangement**

The Java Bluetooth APIs contain the classes LocalDevice and RemoteDevice, which provide the device-management capabilities defined in the Generic Access Profile. LocalDevice depends on the javax.bluetooth.DeviceClass class to retrieve the device's type and the kinds of services it offers. The RemoteDevice class represents a remote device (a device within a range of reach) and provides methods to retrieve information about the device, including its Bluetooth address and name. The following code snippet retrieves that information for the local device:

|  |
| --- |
| ...  // retrieve the local Bluetooth device object  LocalDevice local = LocalDevice.getLocalDevice();  // retrieve the Bluetooth address of the local device  String address = local.getBluetoothAddress();  // retrieve the name of the local Bluetooth device  String name = local.getFriendlyName();  ... |

You can get the same information about a remote device:

|  |
| --- |
| ...  // retrieve the device that is at the other end of  // the Bluetooth Serial Port Profile connection,  // over RFCOMM connection  RemoteDevice remote = RemoteDevice.getRemoteDevice(  javax.microedition.io.Connection c);  // retrieve the Bluetooth address of the remote device  String remoteAddress = remote.getBluetoothAddress();  // retrieve the name of the remote Bluetooth device  String remoteName = local.getFriendlyName(true);  ... |

The RemoteDevice class also provides methods to authenticate (pair and bond), authorize, or encrypt data transferred between local and remote devices.

**Discovery**

The core Bluetooth API's DiscoveryAgent class and DiscoveryListener interface provide the necessary discovery services.

A Bluetooth device can use a DiscoveryAgent object to obtain a list of accessible devices, in any of three ways:

The DiscoveryAgent.startInquiry method places the device into an inquiry mode. To take advantage of this mode, the application must specify an event listener that will respond to inquiry-related events. DiscoveryListener.deviceDiscovered is called each time an inquiry finds a device. When the inquiry is completed or canceled, DiscoveryListener.inquiryCompleted is invoked.

If the device doesn't wish to wait for devices to be discovered, it can use the DiscoveryAgent.retrieveDevices method to retrieve an existing list. Depending on the parameter passed, this method will return either a list of devices that were found in a previous inquiry, or a list of *pre-known devices* that the local device has told the Bluetooth Control Center it will contact often.

The following code snippets demonstrate the various approaches:

|  |
| --- |
| ...  // retrieve the discovery agent  DiscoveryAgent agent = local.getDiscoveryAgent();  // place the device in inquiry mode  boolean complete = agent.startInquiry();  // retrieve the discovery agent  DiscoveryAgent agent = local.getDiscoveryAgent();  RemoteDevice[] devices =  // return an array of pre-known devices  agent.retrieveDevices(DiscoveryAgent.PREKNOWN);  // return an array of devices found in a previous inquiry  agent.retrieveDevices(DiscoveryAgent.CACHED);  … |

#### Service Discovery

Once the local device has discovered at least one remote device, it can begin to search for available *services* - Bluetooth applications it can use to accomplish useful tasks. Because service discovery is much like device discovery, DiscoveryAgent also provides methods to discover services on a Bluetooth server device, and to initiate service-discovery transactions. The API provides mechanisms to search for services on remote devices, but not for services on the local device.

#### Service Registration

Before a service can be discovered, it must first be *registered* - advertised on a *Bluetooth server device*. The server is responsible for:

* Creating a service record that describes the service offered
* Adding the service record to the server's Service Discovery DataBase (SDDB), so it's visible and available to potential clients
* Registering the Bluetooth security measures associated with the service (enforced for connections with clients)
* Accepting connections from clients
* Updating the service record in the SDDB whenever the service's attributes change
* Removing or disabling the service record in the SDDB when the service is no longer available

The following code fragment shows the process involved in registering a service using the Java APIs for Bluetooth:

1. To create a new service record that represents the service, Connector.open is invoked with a server connection URL argument, and the result is casted to StreamConnectionNotifier that represents the service:
2. Obtain the service record created by the server device.
3. Indicate that the service is ready to accept a client connection.
4. When the server is ready to exit, close the connection and remove the service record.

#### Communication

For a local device to use a service on a remote device, the two devices must share a common communications protocol. So that applications can access a wide variety of Bluetooth services, the Java APIs for Bluetooth provide mechanisms that allow connections to any service that uses RFCOMM, L2CAP, or OBEX as its protocol. If a service uses another protocol (such as TCP/IP) layered above one of these protocols, the application *can* access the service, but only if it implements the additional protocol in the application, using the CLDC Generic Connection Framework.

**Serial Port Profile**

|  |
| --- |
| ...  // retrieve the discovery agent  DiscoveryAgent agent = local.getDiscoveryAgent();  // place the device in inquiry mode  boolean complete = agent.startInquiry();  // retrieve the discovery agent  DiscoveryAgent agent = local.getDiscoveryAgent();  RemoteDevice[] devices =  // return an array of pre-known devices  agent.retrieveDevices(DiscoveryAgent.PREKNOWN);  // return an array of devices found in a previous inquiry  agent.retrieveDevices(DiscoveryAgent.CACHED);  … |

The RFCOMM protocol, which is layered over the L2CAP protocol, emulates an RS-232 serial connection. The Serial Port Profile (SPP) eases communication between Bluetooth devices by providing a stream-based interface to the RFCOMM protocol. Some limitations:

* Two devices can share only one RFCOMM session at a time.
* A single Bluetooth device can have at most 30 active RFCOMM services.
* A device can support only one client connection to any given service at a time.

The server must:

1. Construct a URL that indicates how to connect to the service, and store it in the service record
2. Make the service record available to the client
3. Accept a connection from the client
4. Send and receive data to and from the client

The URL placed in the service record may look something like:

btspp://102030405060740A1B1C1D1E100:5

This says that a client should use the Bluetooth Serial Port Profile to establish a connection to this service, which is identified with server channel 5 on a device whose address is 102030405060740A1B1C1D1E100.

|  |
| --- |
| *...*  *// assuming the service UID has been retrieved*  *String serviceURL ="btspp://localhost:"+serviceUID.toString());*  *try {*  *// create a server connection*  *StreamConnectionNotifier notifier =*  *(StreamConnectionNotifier) Connector.open(serviceURL);*  *// accept client connections*  *StreamConnection connection = notifier.acceptAndOpen();*  *// Open Streams for the connection*  *InputStream in = connection.openInputStream();*  *OutputStream out = connection.openOutputStream();*  *// send data to the client*  *out.write(msg.getBytes());*  *// read data from client*  *in.read();*  *connection.close();*  *} catch(IOException e) {*  *e.printStackTrace();*  *}*  *...* |

At the other end, as the next code snippet shows, to set up an RFCOMM connection to a server the client must:

1. Initiate a service discovery to retrieve the service record
2. Construct a connection URL using the service record
3. Open a connection to the server
4. Send and receive data to and from the server

|  |
| --- |
| ...  // (assuming we have the service record)  // use record to retrieve a connection URL  String url =  record.getConnectionURL(  record.NOAUTHENTICATE\_NOENCRYPT, false);  // NOAUTHENTICATE\_NOENCRYPT implies that authentication and encryption //isn’t required  // open a connection to the server  StreamConnection connection =  (StreamConnection) Connector.open(url);  connection.close();  } catch(IOException e) {  e.printStackTrace();  }  … |

**Discussion:**

**Resources Used:**

For the development of our application regarding the project we have used

* NetBeans 6.9.1. as the IDE
* Wireless Toolkit as the simulator.
* JDK 1.6.021 as the Software Development Kit
* Mobility and Mobility End to End as Java ME plugins for providing infrastructure to create, build and manage Mobile projects.

**Project Implementations:**

The Java MIDlet class for the client/server implements DiscoveryListener, CommandListener and Runnable classes. Implementing the above mentioned classes, provide us with some basic functions for discovery of devices and services in the application, handling the commands and implementing the thread. The implemented functions are as follows:

1. *public void startApp() {}*

The function is invoked when the MIDlet is run. In the application it is used to set the display to MIDlet and create a LocalDevice and set it on discoverable mode.

1. *public void pauseApp() {}*

The function is used in order to halt the process of the application. However, this hasn’t been used in our application.

1. *public void destroyApp(boolean unconditional) {}*

The DestroyApp function is called when the user tries to close the application.The function has been implemented in order to exit the application, close the available connection and input/output streams.

1. *public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {}*

The function is invoked each time a remote device is discovered. This function has further been used to add the remote devices in a Vector and display them in a list.

1. *public void servicesDiscovered(int transID, ServiceRecord[] servRecord) {}*

The function is invoked each time a new service is discovered in the selected remote device. Similar to the *deviceDiscovered()* function, it has further been used to add the services found in a vector and enlist them.

1. *public void serviceSearchCompleted(int transID, int respCode) {}*

The function is invoked after a service search is completed. The function uses a switch case depending on how the search has ended. The second parameter of the function represents the mode of completion of a search which may be of following types:

* + - * Search Terminated
      * Search Completed
      * Error in search
      * No records found
      * Device not reachable

After the function is called, if any services are found then the *servicesDiscovered()* function is invoked and the list of services is shown.

1. *public void inquiryCompleted(int discType) {}*

The function is called when the *startInquiry()* function is called by the DiscoveryAgent instance. Similar to the *serviceSearchCompleted()* function, it also has different mode of completions as follows:

* Inquiry Completed
* Inquiry Error
* Inquiry Terminated

If any remote devices are discovered, the *deviceDiscovered()* function is invoked and the device’s names are enlisted.

1. *public void commandAction(Command c, Displayable d) {}*

The *CommandAction()* function is the implementation of the class CommandListener. It is invoked each time a Command instance is selected in the MIDlet.

In our application, we have used commands such as EXIT for terminating the application, SEARCH for initiating search for remote devices and services, BACK for returning to previous menu, CANCEL for terminating a search in process and SELECT\_COMMAND in case of list to selected individual items present in the list­.

1. *public void run() {}*

The *run()* function is the implementation of the class Runnable and is used to run a separate thread in parallel with the main program.

In the application, the function is used in order to run a thread to open a connection and input/output data streams for server to connect.

Besides the pre-implemented functions, we have also defined functions for the easy access in the program.

The functions defined in case of client are as follows:

1. *startdevicesearch(){}*

The function when called initiates a device search by invoking *startInquiry()* function and then maintains a Vector for the found devices and enlists them.

The function calling is controlled by a Boolean data type “devicesearch” and can be called only when the devicesearch parameter is set to false. Once the search is initiated the value is changed to true and when the search ends it is set back to false.



**Figure: Performing device search**

1. *startservicesearch(){}*

The is called in order to search services in a particular remote device and when called it invokes the *searchServices()* function which actually starts the search for services. The *searchServices()* function starts the search for the services with particular attributes which are passed into the function as parameters. Along with the search, the function *Startservicesearch()* also stores the found services in a Vector and enlists them and then displays it to the user

1. *mainmenu(){}*

We have maintained a menu number in order to track which menu we are in. The first menu we see has been called the mainmenu and is designated with no. 1. This menu is displayed when the *mainmenu()* function is called by the  *startApp()* function and shows the list of pre-known devices.

1. *showdevices(){}*

This function is called in order to synchronize the list of devices found and display the list of the found devices.



**Figure: List of devices discovered**

1. *Showservices(){}*

This function is called in order to synchronize the list of services found and display the list of the found services.

1. *Run(){}*

The *run()*function starts the thread which controls the datastreams and opens the connection for the server to connect. It opens the connection by calling the function *Connector.open()* which opens the stream connection in order to connect to the client.

1. *getnames(){}*

This function is used in order to get friendly names of the remote devices.

1. *clear(){}*

This function is called before the *destroyApp()* function is invoked. This function closes the connection if active and the data streams and also stops the thread in case it is active. Doing this prevents a null pointer exception and proper termination of the application.

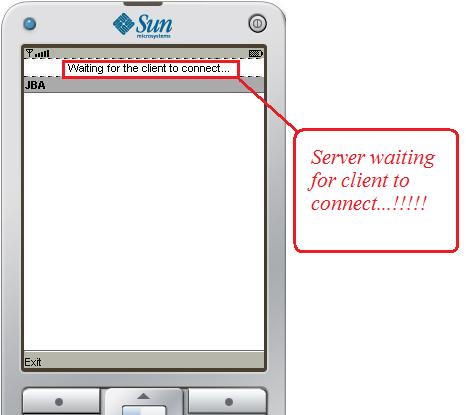
Similarly, the functions defined in the case of server are as follows:

1. *readytheserver(){}*

This function sets up a new localdevice and sets it to discoverable mode. Then it starts a connection through a URL using the function *Connector.open(URL*). It also sets the attributes of the service record (also update the record) and makes service public browseable such that a client can detect it.

1. *startserver(){}*

This function just plays the role of starting the thread which in turn invokes the run() function and the server is able to accept the connections from clients.



**Figure1: Server asking for permission to make connection**

**Figure2: Server waiting for a client to connect**

1. *run(){}*

The *run()*function starts the thread which controls the datastreams and opens the connection for the client to connect. It opens the connection by calling the function *acceptandopen()* which sets the ConnectionNotifier to accept the client connections.

The *acceptandopen()*function holds and blocks the program until a client connects to the server.

1. *clear(){}*

It works the same way as in case of the client and is called before the *destroyApp()* function is invoked.



**Figure: Anatomy of the application**

****

**Figure: Working Process of the application**

**Working mechanism**

* **Client**

The client when initiated shows the list of previously searched devices and provides the user with a search command and exit command. The exit command leads to the termination of the application, but if the user chooses the search command the new search begins and the list of devices found (if any) is then displayed to the user. The user then can initiate a search for the services in any of the devices found. If any services are found in the device, the services are then displayed to the user in the form of a list. Now the user can select the service and the application displays to which user s/he is connected to.

* **Server**

However when a server is initiated, the user is asked if s/he really wants to allow the application to access the connectivity applications. If selected ok, the server initiates and waits for the client to connect to it. Now, when some client connects to the server it displays its friendly name and bluetooth address.

**GANTT CHART**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Work | Week 1 | Week 2 | Week 3 | Week 4 | Week 5 | Week 6 | Week 7 | Week 8 | | Week 9 | Week 10 | |
| Study Phase |  | | |  | | | | | | | | |
|  |  |  | | | | | | | | | |  |
|  |  | | | | |  |  | | | | | |
|  |  |  | | | |  | | | | | | |
| Packaging and testing |  | | | | | | | |  | | | |

Table 6: Gantt char for the project

**Conclusion**

Our project to develop “JBA” helped us learn a great deal about java programming regarding bluetooth and team work. We are very grateful to all those who shared their ideas with us in developing this application, especially to our Supervisor Mr. Samir Kharel for letting us to carry out this project.

This project led us to some serious application development skills that we were totally unfamiliar with. We learnt the importance of planning and organization for building big software. As for a second year computer engineering students, we believe that we have put our best effort in completing this project.

We eagerly look forward for more such future endeavors challenging the limitations of our imaginations.

**REFERENCES**

* Java Programming for absolute beginners - **Joseph P. Russell**
* Java How to Program – **Dietel and Dietel**
* Bluetooth Application Programming with the Java APIs – **C.B. Kumar, P.J.Kine, T.J. Thompson**
* *Wireless Java: Developing with J2ME*, Second Edition - **J. Knudsen**
* Developing Applications with the Java APIs for Bluetooth™ (JSR-82) (PDF)
* <http://www.sunjava.com>
* <http://www.netbeans.org>
* <http://www.eclipse.org>
* <http://java.sun.com/products/j2mewtoolkit/>
* <http://java.sun.com/products/midp/>
* <http://java.sun.com/j2me/>
* <http://java.sun.com/products/cldc/>
* <http://www.bluetooth.org>
* <http://www.bluetooth.com>
* <http://qualweb.bluetooth.org>
* <http://developers.sun.com/techtopics/mobility/>