



**DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING**

Voting on Mobile Devices

Hannah Fagan

50006165

June 2004

B.Eng. in ELECTRONIC SYSTEMS

Supervised by Dr. Jennifer McManis

and Dr. Sean Murphy

Acknowledgements

I would like to thank my supervisor Dr. Sean Murphy for his guidance, enthusiasm and commitment to this project. Thanks are also due to Mr. Jim Dowling.

Declaration

I hereby declare that, except where otherwise indicated, this document is entirely my own work and has not been submitted in whole or in part to any other university.

Signed:

Date:

Abstract

The purpose of this document is to share the research that has been done by the writer into Java 2 Micro Edition and to share the attempted implementation of a voting application using J2ME over GPRS.

Contents

	Page
1. Introduction.....	5
2. Application Description.....	6
3. J2ME and J2SE.....	9
4. Implementation.....	16
5. Results.....	27
6. Conclusion and Future Work.....	31
7. References.....	32

Introduction-

This fourth year project aims to research J2ME and learn about its usage and to implement an application capable of running a voting service on a mobile phone over GPRS. It will be implemented on the server side in J2SE and on the client side in J2ME. First an attempt will be made to implement the application using only J2SE, if this is successful the client side of the application will be attempted.

Application Description

The current situation: SMS

The current voting system uses a SMS (Short Messaging Service) mechanism. Short message service (SMS) is a globally accepted wireless service that enables the transmission of alphanumeric messages between mobile subscribers and external systems such as electronic mail, paging, and voice-mail systems. SMS provides a mechanism for transmitting short messages to and from wireless devices. The service makes use of an SMSC, which acts as a store-and-forward system for short messages. The wireless network provides the mechanisms required to find the destination station and transports short messages between the SMSCs and wireless stations. For voting using SMS the user must first see a list of options and send their choice to the number for their specific choice. One of the problems with the SMS voting system is the cost of sending the messages, no matter how short they are it still costs the same amount for sending a message. GPRS on the other hand charges for the size of the information sent. Therefore GPRS is a better mechanism for running a voting application on.

General Packet Radio Service (GPRS)

GPRS is a nonvoice value added service that allows information to be sent and received across a mobile telephone network. Theoretical maximum speeds of up to 171.2 kilobits per second (kbps) are achievable with GPRS using eight timeslots at the same time. This is about three times as fast as the data transmission speeds possible over today's fixed telecommunications networks. By allowing information to be transmitted more quickly, immediately and efficiently across the mobile network, GPRS is a relatively less costly mobile data service compared to SMS.

GPRS based solution-

The purpose of this application is to replace the current SMS voting mechanism with an application that allows the user to send the name or number of a voting scheme and in return they will receive a list of voting options for this scheme. The user will then send their vote and will receive a vote confirmation in return. (See Figure 1)

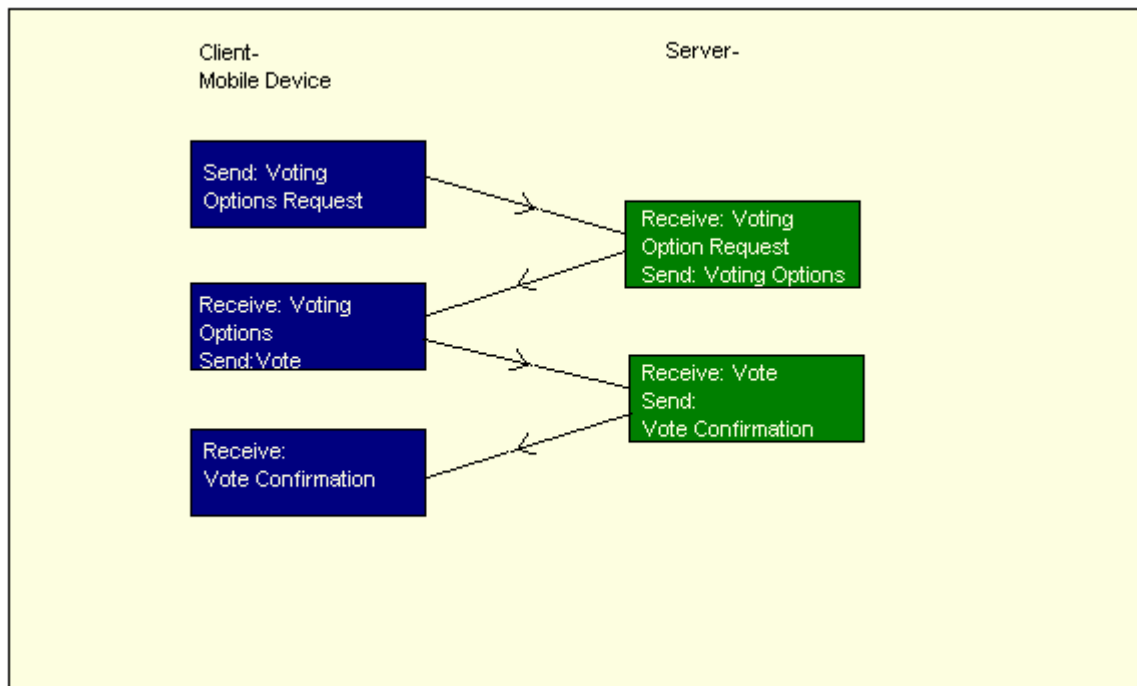


Figure 1

J2ME

And

J2SE

Java:

Java is an object oriented language developed in 1991 by Sun Microsystems. It shares many similarities with C and C++ but it is not based on either of these languages. It was originally developed because C++ was inadequate for dealing with certain tasks. Java was designed from the ground up to allow for secure execution of code across a network, even when the source of that code was untrusted and possibly malicious.

The Java platform is based on the power of networks and the idea that the same software should run on many different kinds of computers, consumer and other devices. Since its initial release in 1995, Java has grown in popularity and usage because of its portability. The Java platform allows the user to run the same Java application on lots of different kinds of computers.

Any Java application can easily be delivered over the Internet, or any network, without operating system or hardware platform compatibility issues. A Java technology based application can be run on a PC, a Macintosh computer, a network computer, or even new technologies like Internet screen phones. Furthermore, the Java platform was designed to run programs securely on networks, which means that it integrates safely with the existing systems on your network.

The Editions

The Java platform is divided into 3 different editions (J2SE, J2EE and J2ME).

Java 2 Platform, Enterprise Edition (J2EE)

J2EE is used to develop server based applications, it simplifies enterprise applications by basing them on standardised, modular and re-usable components

Enterprise JavaBeans, providing a complete set of services to those components, and handling many details of application behaviour automatically. By automating many of the time-consuming and difficult tasks of application development, J2EE technology allows enterprise developers to focus on adding value and therefore enhancing business logic, rather than building infrastructure.

Java 2 Platform, Standard Edition (J2SE)

The J2SE platform could be considered a subset of the J2EE platform. It is used for desktop-based applications and is a fast and secure foundation for building and deploying client-side enterprise applications. J2SE technology provides speedy performance and high functionality that is demanded by Web users. J2SE includes options for objects, strings, threads, numbers, input and output, data structures, system properties, date and time, applets, networking, and security on networks.

Java 2 Platform, Micro Edition (J2ME)

J2ME is used in handheld and embedded devices. J2ME technology enables device manufacturers, service providers, and content creators to gain a competitive advantage and capitalise on new revenue streams by rapidly and cost-effectively developing and deploying compelling new applications and services to their customers worldwide.

J2ME Classes

The difference between the editions is the set of class libraries defined by each edition. It is possible to run the same program in all 3 editions but the classes referred to by the programs must be available in all editions for this to work. J2ME devices have fewer classes than those that J2SE and J2EE provide especially the

smaller devices. The smaller the amount of space available on the devices the fewer classes it can implement.

In J2ME the Java runtime environment is adapted for devices that have limitations on them compared to what a standard computer can do. For low-end devices these limitations are extremely limited memory, small screen sizes, alternative input methods and slow processors.

Configurations in J2ME

A configuration is a complete Java runtime environment consisting of three things-

- A Java virtual machine (VM)
- Native code to interface to the underlying system
- A set of core Java runtime classes

To use a configuration a device must meet certain minimum requirements as defined in the configuration's formal specification. In J2ME there are two configurations the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC)

CLDC

The CLDC is for very constrained devices, those with small amounts of memory and/or slow processors. The virtual machines used by the CLDC omit important features like finalisation and the set of core classes is just a tiny fraction of the J2SE core classes, the basics from the `java.lang`, `java.io` and `java.util` packages with some additional classes from the `javax.microedition.io` package. Only selected classes from the J2SE packages are included: the `java.util.Vector` and `java.util.Hashtable` classes are included, but none of the collection classes are. The largest package is the `java.lang` package, which defines the classes that are fundamental to any Java

application, classes like `java.lang.Object` or `java.lang.Integer`. The `java.io` subset only includes abstract and memory-based classes and interfaces like `java.io.DataInput` or `java.io.ByteArrayInputStream`. The `java.util` subset only includes a few utility classes. The K virtual machine (KVM) is available to support CLDC. It is the smallest VM and supports handheld consumer devices with 128K to 512 k of available memory for the Java technology stack. However CLDC does not require the use of this specific VM, only the use of a VM that adheres to the requirements of the specification in question. The VM is restricted in the following ways when compared to a full-featured J2SE VM. These restrictions allow the VM to fit the memory and power constraints of the small devices that the CLDC target: the VM and classes can fit in 128K of memory.

For CLDC 1.0 the primary restrictions on the VM were:

- No floating point types.
- No object finalisation or weak references.
- No JNI or reflection (hence no object serialisation).
- No thread groups or daemon threads (threads are supported, just not thread groups).
- No application-defined class loaders.

CLDC 1.1 relaxes some of these restrictions, in particular re-enabling support for floating-point types and weak references. CLDC also requires class verification to be done differently. An off-device class verifier, in a process called preverification, processes class files. At runtime, the VM uses information inserted into the class files by the preverifier to perform the final verification steps. Files that have not been processed by the preverifier are not loaded since they cannot be verified.

The CLDC also defines a new set of Application Programming Interfaces (APIs) for input/output called the Generic Connection Framework. J2SE includes many classes for performing input and output; classes that are found in the `java.io` and the `java.net` packages. Unfortunately, there are a large number of I/O classes and they tend to encapsulate I/O models that are not necessarily found on all devices. For example,

some handheld devices do not have file systems. Socket support is not universal, either. CLDC defines a new set of APIs for I/O called the Generic Connection Framework. The GFC, part of the new javax.microedition.io package, defines interfaces for the different kinds of I/O that are possible and a factory class for creating objects that implement those interfaces. The type of object to create is specified in the protocol part of the URL (universal resource locator) passed to the factory class.

CDC

The CDC unlike the CLDC supports a full Java VM and a much larger set of core classes so it requires more memory than the CLDC and a faster processor.

Profiles

A profile adds domain-specific classes to a configuration to fill in the missing functionality and to support specific uses of a device. Most profiles define user interface classes for building interactive applications.

Mobile Information Device Profile (MIDP)

MIDP was the first profile to be released for J2ME. It is a CLDC based profile for running applications on mobile phones and interactive pagers with small screens wireless connectivity and limited memory. All applications in the MIDP profile 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, except that their state is more independent from the display state. A MIDlet can exist in four different states: loaded, active,

paused, and destroyed. When a MIDlet is loaded into the device and the constructor is called, it is in the loaded state. This can happen at any time before the program manager starts the application by calling the startApp() method. After startApp() is called, the MIDlet is in the active state until the program manager calls pauseApp() or destroyApp(); pauseApp() pauses the MIDlet, and desroyApp() terminates the MIDlet. Figure 2 shows the life cycle of a MIDlet.

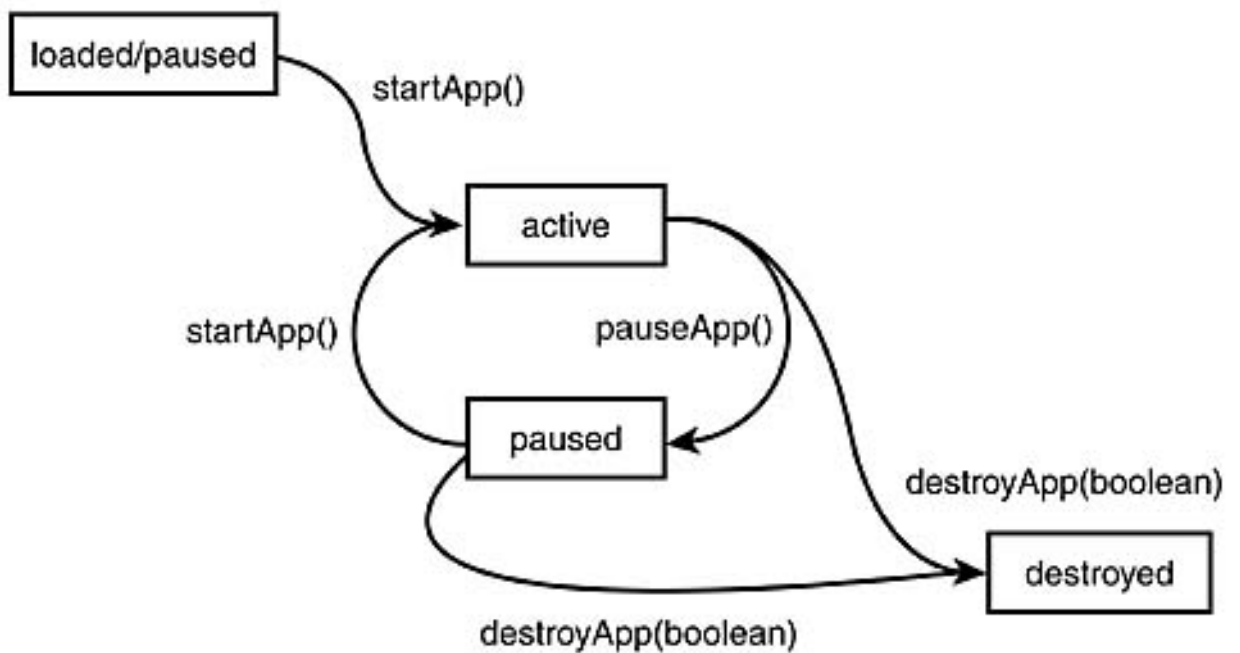


Figure 2

Implementation:

The voting application in both J2SE and J2ME consists of two main parts: the client and the server.

The client handles the user interaction and sends the requests for information and the vote cast to the server. The server returns the information requested and updates the appropriate voting results file.

4.1 J2SE Implementation-

The Client

The client is a normal basic client application; it works using a number of functions. The uses of these functions are explained below.

The first function that is called is `chooseSystem()`. It asks the user to input their choice of voting system and using the `send()` function it send the choice to the server.

```
private void chooseSystem()
{
    String cho;
    System.out.println("Please input name of voting system:");
    cho = Console.readString();
    this.send(cho);
}
```

The next two functions that are called are `getNumOptions()` and `getOptions()`. `getNumOptions()` sends a request to the server for the number of options in the voting system previously specified and prints the number of options received to the screen. `getOptions()` sends a request to the server for the names of these options.

```
private void getNumOptions()
{
    String NumOptions;
```

```

System.out.println("GetNumOptions");
this.send("GetNumOptions");
NumOptions = (String)receive();
if (NumOptions != null)
{
    System.out.println(NumOptions);
}
}

```

```

private void getOptions()
{
    String Options;
    System.out.println("GetOptions");
    this.send("GetOptions");
    Options = (String)receive();
    if (Options != null)
    {
        System.out.println(Options);
    }
}

```

The `getVote()` asks the user to input the number corresponding to their choice and send this to the server.

```

private void getVote()
{
    System.out.println("Please input the number corresponding to your choice:");
    int myInt;
    myInt = Console.readInt();
    String Vote = Integer.toString(myInt);
    this.send(Vote);
}

```

The thankYou() functions prints the vote confirmation received from the server after a vote has been sent.

```
private void thankYou()
{
    String thanks;
    thanks = (String)receive();
        if (thanks != null)
        {
            System.out.println(thanks);
        }
    }
}
```

The send function sends an object to the server using the ObjectOutputStream.

```
private void send(Object o)
{
    os.writeObject(o);
    os.flush();
}
}
```

The receive function reads objects from the ObjectInputStream and returns the object received to the function using the receive function.

```
private Object receive()
{
    Object o = null
    o = is.readObject();
    return o;
}
}
```

After this has been completed the client closes the connection.

The Server

The server is made up of 3 parts-

VoteServer.java

The voteserver program handles opening a connection with the client. It then passes control onto the handleconnection.java. The main function of voteserver.java is to listen out on the port for a client connection. It does this as follows-

```
while (true)  
{  
    Socket clientSocket = null;  
    clientSocket = serverSocket.accept();  
    System.out.println("Server has just accepted socket connection from a  
client");  
}
```

It then create a new HandleConnection to interact with the client-

```
HandleConnection con = new HandleConnection(clientSocket);
```

HandleConnection.java

The HandleConnection program first creates a new Vote service to handle the voting.

```
theVoteService = new Vote();
```

HandleConnection then associates the input and output stream is and os with the incoming client input and output streams.

```
this.is = new ObjectInputStream(clientSocket.getInputStream());  
this.os = new ObjectOutputStream(clientSocket.getOutputStream());
```

It then runs `readCommand()` which reads whatever is on the input stream and converts this to a string. `readCommand` compares this string with a set of strings. If the string sent by the client corresponds with “GetNumOptions” `readCommand` runs `getNumOptions()`. If it corresponds with the string for sending options it runs `getOptions()`. If it corresponds with a number, this number is sent to `Vote.java` and `thankYou()` send a vote confirmation. If it does not correspond to anything `HandleConnection` sends an invalid input message back to the client.

The functions for sending and receiving objects are the same as those for the client.

Vote.java

The vote class is made up of the functions concerned with the voting process. The first function called is used to associate the correct string array with the `voteOption` string array. It uses `getOptions()` from `HandleConnection` which sends an integer to `getArray` this integer corresponds with an array. The second function that is called is `getFileName()`. It works in the same way as `getArray` associating a string, this time containing the names of the results file specified, with the `fileName` variable.

```
public String [] getArray(int x)
{
    if(x==0)
    {
        voteOption = Names;
    }
    else if(x==1)
    {
        voteOption = Transport;
    }
    else if(x==2)
    {
        voteOption = Animals;
    }
}
```

```

        return voteOption;
    }

```

The getResultData function uses a buffered reader to read in each line of the results file and store the numbers in sequential indexes of an integer array.

```

public void getResultData(String fileName)
{
    BufferedReader record = new BufferedReader(new FileReader(fileName));
    for(int i=0;i<voteOption.length;i++)
    {
        String text = record.readLine();
        result[i] = Integer.parseInt(text);
    }
    record.close();
}

```

The getVoteOption and getOption functions are used to send the number of options and the options themselves to the client.

```

public String getVoteOption()
{
    int numOptions;
    numOptions = getNumOptions();
    String VoteOption = "The number of options is: " + numOptions;
    return VoteOption;
}

public String getOption()
{
    int numOptions;
    numOptions = getNumOptions();
    String Option = "The options are: ";
    for (int i=0;i<numOptions;i++)

```

```

        {
            Option = Option + voteOption[i];
        }
        return Option;
    }

```

The castVote function increments the index of the result integer array for the vote specified by the client.

```

    public void castVote(int index)
    {
        index--;
        String Name = fileName;
        this.getResultData(Name);
        result[index]++;
        this.getResult();
        this.writeData(Name);
    }

```

The getResult function prints out the results to the server as follows-

```

    public void getResult ()
    {
        int numOptions;
        int res;
        numOptions = getNumOptions();
        String Results = "The results are: \n";
        System.out.print(Results);
        for (int i=0;i<numOptions;i++)
        {
            String Option = voteOption[i] + ":" + result[i];
            String Result = Option + "\n";
            System.out.print(Result + "\n");
        }
    }

```

```
}
```

The writeData function updates the appropriate results file after a vote has been cast.

```
public void writeData(String fileName)  
{  
    FileWriter fw = new FileWriter(fileName);  
    PrintWriter fileOutput = new PrintWriter(fw);  
    for(int i=0;i<voteOption.length; i++)  
    {  
        String Res = "" + result[i];  
        fileOutput.println(Res);  
    }  
    fw.close();  
}
```

4.2 J2ME Implementation-

The client

The client for a J2ME application is different from a J2SE client. This client has three parts- the actual client class, the Socket MIDlet and the sender class.

Client.java

The client opens a form called Socket Client. This form has a box at the top which is a String Item for displaying messages. It also has a text field for inputting messages and a two command buttons. See figure 3.

```
f = new Form("Socket Client");  
si = new StringItem("Status:", " ");  
tf = new TextField("Send:", "", 30, TextField.ANY);  
f.append(si);  
f.append(tf);  
f.addCommand(exitCommand);
```

```
f.addCommand(sendCommand);
```



Figure 3

The run function opens connection with the server and passes control to the sender class if a message is being sent. If a message is being received it creates a string buffer and converts the information being received to a string to be printed on the screen.

```
public void run() {  
    sender = new Sender(os);  
  
    while (true) {  
        StringBuffer sb = new StringBuffer();  
        int c = 0;  
        while (((c = is.read()) != '\n') && (c != -1))  
        {  
            sb.append((char) c);  
        }  
        si.setText("Message received - " + sb.toString());  
    }  
}
```

The commandAction function tells the client what to do if one of the command buttons is pressed depending on which one has been pressed. If send is pressed it sends the string in the text field to the sender class to be sent. If exit is pressed the client will close the application.

```
public void commandAction(Command c, Displayable s) {
```

```

    if (c == sendCommand && !parent.isPaused()) {
        sender.send(tf.getString());
    }
    if ((c == Alert.DISMISS_COMMAND) || (c == exitCommand)) {
        parent.notifyDestroyed();
        parent.destroyApp(true);
    }
}

```

Sender.java

The main purpose of send messages to the server. It takes whatever is in message (from the client) and sends it to the server.

```

    public synchronized void run() {

        while(true)
        {
            os.write(message.getBytes());
        }
    }
}

```

SocketMIDlet.java

The purpose of SocketMIDlet.java is to control the MIDlet itself it is able to start pause and destroy the application. It is SocketMIDlet that creates a new client.java.

```

    public void startApp() {
        isPaused = false;
    }
}

```

```

public void pauseApp() {
    isPaused = true;
}

public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
        destroyApp(true);
        notifyDestroyed();
    } else if (c == startCommand) {
        String name = "client";
        client = new Client(this);
        client.start();
    }
}
}

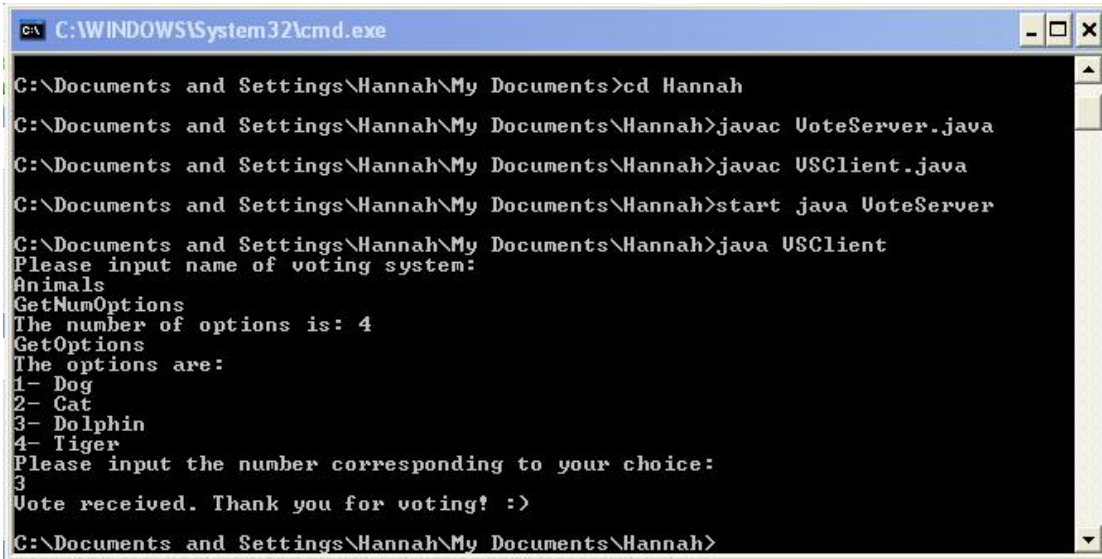
```

The server

For the J2ME application the J2SE server was used with a few changes to the VoteServer and HandleConnection classes. The J2SE server is unable to interpret information sent if the InputStream is defined as an ObjectInputStream so this must be changed to a DataInputStream. It is however difficult to interpret the information sent and I have been unable to implement this properly.

Results:

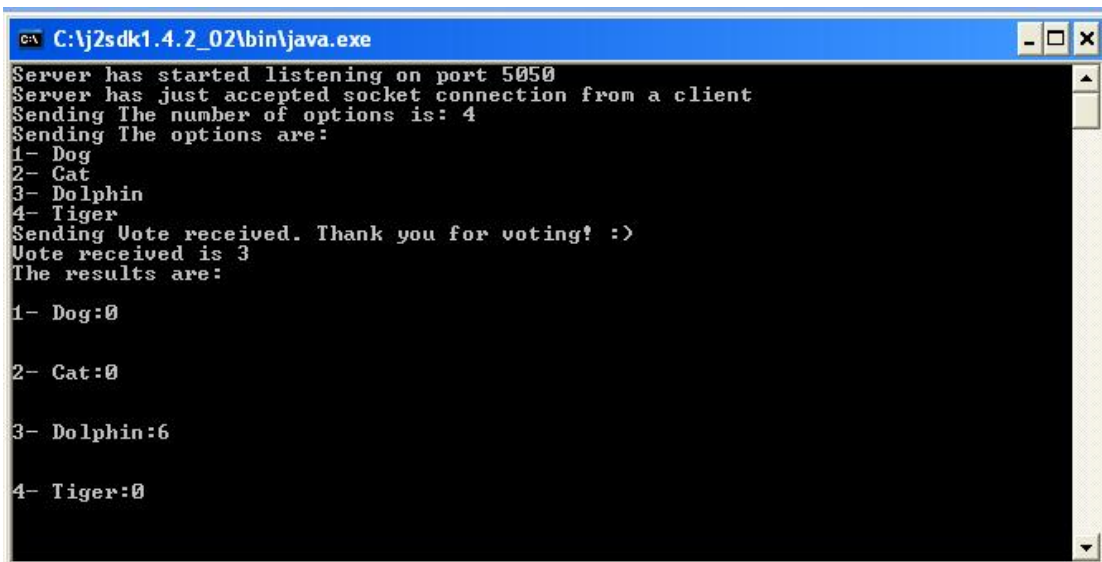
This screen shot shows the results for the client side of the J2SE application.



```
C:\WINDOWS\System32\cmd.exe
C:\Documents and Settings\Hannah\My Documents>cd Hannah
C:\Documents and Settings\Hannah\My Documents\Hannah>javac VoteServer.java
C:\Documents and Settings\Hannah\My Documents\Hannah>javac USClient.java
C:\Documents and Settings\Hannah\My Documents\Hannah>start java VoteServer
C:\Documents and Settings\Hannah\My Documents\Hannah>java USClient
Please input name of voting system:
Animals
GetNumOptions
The number of options is: 4
GetOptions
The options are:
1- Dog
2- Cat
3- Dolphin
4- Tiger
Please input the number corresponding to your choice:
3
Vote received. Thank you for voting! :>
C:\Documents and Settings\Hannah\My Documents\Hannah>
```

As can be seen from the above the user is first asked to enter the name of the voting system they wish to vote on. They are then sent a numbered list of these options and they must input the number corresponding to their choice, which is then sent to the server. If the server receives this vote correctly a vote confirmation message is then sent.

This screenshot shows the results for the server side.



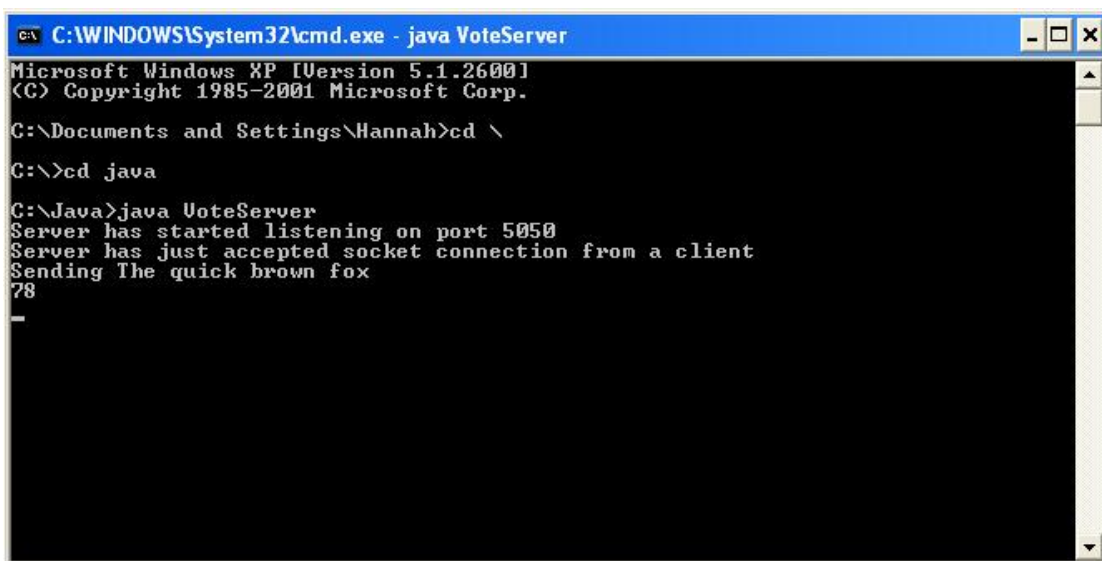
```
C:\j2sdk1.4.2_02\bin\java.exe
Server has started listening on port 5050
Server has just accepted socket connection from a client
Sending The number of options is: 4
Sending The options are:
1- Dog
2- Cat
3- Dolphin
4- Tiger
Sending Vote received. Thank you for voting! :>
Vote received is 3
The results are:
1- Dog:0
2- Cat:0
3- Dolphin:6
4- Tiger:0
```

As can be seen from the above image the server indicates when it starts to listen to for a client connection and when the client actually connects. It also indicates what it is sending to the client as well as printing the results of the voting on the screen.

The screen shots below show a message “The quick brown fox” being sent by the server and the client attempting to send “Names” back to the server. Unfortunately as can be seen from the server screen shot, the server cannot interpret this properly.



Client screenshots



Server screen shot

Conclusion:

I have been unsuccessful in my attempt to implement this application using J2ME, however I feel that with more time it would be possible to have the application working successfully as it is only a minor problem that needs to be worked through.

Future Work:

This application is quite simple and there are many things that could be added to it to make it user-friendlier. The following are some of the possibilities-

Adding more voting systems-

For the application all voting systems must be hard-coded into the program. To make it easier for more voting systems to be added it would be better to be able to add text files with the voting options to a voting system database and these could be read into the program as they are requested by the client

Web site-

With the current application it is only possible to see votes by looking at the individual results files. A web site would make it easier for everyone to view the results of the different votes and it could also be possible to include the application for adding more voting systems in it. The home page of the web site would be universally accessible and include the names of all the voting schemes. There may also be a facility on the web page to vote. The results of the votes could either be only publicly accessible when a vote has been cast or not publicly accessible at all until the time allowed for voting has elapsed and the final results could be put on the

home page. The application for adding more votes would only be available to authorised users.

Dealing with more than one client simultaneously-

The server is able to deal with only one client at a time. If a second client attempts to connect it will have to wait until the previous client has terminated its connection. If the vote service is very busy this leads to the possibility of long waiting times for users. Also if the votes have to be cast within a specific amount of time not all the users who wish to vote will be able to in the time allowed. To overcome this problem it may be possible to a separate `HandleConnection` opened for each client that connects.

References

1. Computing concepts with Java 2 essentials (2nd Edition) by Cay Horstmann
2. Sun Microsystems Inc., <http://java.sun.com> (10 June 2003).
3. GSM world <http://www.gsmworld.com> (10 June 2003)