Objective

The objective to this lab is to get experience with:

1. Network communication APIs
2. Retrieving and viewing wireless data
3. Threads
4. Develop a custom client application using these APIs

Experiment 1 Develop an application using network communication APIs

In this experiment, you will learn how to write a custom client application that communicates over the wireless network via HTTP. The client application is connected to the Google using HttpConnection object, reads the Google HTML page and displays the content on the BlackBerry device's screen. Network communication in the BlackBerry JDE is implemented according to the MIDP 2.0 specification. Thus, you should import the required package: import javax.microedition.io.*.

To start the experiment, do the following steps:

1. Create a class extended from **net.rim.device.api.ui.UiApplication** class. Name it "Lab9".
2. Write the constructor and main() methods and add the required codes to these methods.
3. Import the required classes.

```
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import javax.microedition.io.*;
import java.io.*;
```

4. Create a new instance of MainScreen in the constructor. Also, create a new instance of RichTextField object and add it to the screen.

```
class Lab9 extends net.rim.device.api.ui.UiApplication {
    MainScreen screen;
   RichTextField textfield;

    public Lab9() {
```

```
        screen  = new MainScreen();
        textfield = new RichTextField("Lab9: Connecting to the Google.ca \n");
        screen.add(textfield);
        pushScreen(screen );
    }
    public static void main(String[] args) {
        Lab9 theApp = new Lab9();
        theApp.enterEventDispatcher();
    }
}
```

5. In order to avoid potential deadlock, network connections should be performed in a different thread. Technically, you can use multiple threads to run different parts of your program simultaneously. In order to implement a thread you have two possible solutions: one is to define an inner class which extends *Thread* class and the other solution is to implement a *Runnable* interface. This experience uses the second solution. For this purpose do the following steps:
    - Implement Runnable interface. This is done by adding the *implements Runnable* to the Lab9 class as follows:

      ```
      class Lab9 extends net.rim.device.api.ui.UiApplication implements Runnable {
      ….
      }
      ```

    - Every thread in Java begins by executing a *run ()* method. The network communication objects are created in this method. You can define this method as follow:

      ```
      public void run() {

      }
      ```

    - In order to invoke the run() method, you need to create and then start a thread. To do this add the following lines to the constructor:

      ```
      Thread t = new Thread(this);
      t.start();
      ```

      Thus, since the thread starts, the run () method executes.

6. In order to connect to the server and read the Google HTML page do the following tasks:
    - Open a connection:
      First of all, your client application should connect to the server. For this purpose, open a connection. This is done using the following command:
      Connector.open("<protocol>:<address>;<parameters>");

      This is a generic form of connection and support various form of connections such as TCP, UDP and socket connections. In this

experiment, you use HTTP connection. Thus, you should type cast the connection as follows:

```
public void run() {
        String url = "http://www.google.ca/";
        HttpConnection hc = null;
        try {
                // Now make a connection to the server.
                hc = (HttpConnection) Connector.open(url);


        } catch (Exception e) {
                System.out.println(e);
        } finally {
                if (hc != null) {
                    try {
                        hc.close();
                    } catch (IOException ioe) {}
                }
        }
```

At the end, you should close the connection.

- Read the data:
  For this purpose, first you should initiate the connection. This is done by invoking the *openInputStream()* method:

```
InputStream in = null;
….
in = hc.openInputStream();
```

Second, you should read the data. This is done by invoking the *read*( ) method.

```
byte[] raw = new byte[1024];

int readLength = in.read(raw);
```

Third, close the inputstream.

```
if (in != null) {
      try {
          in.close();
       }
      catch (IOException ioe) {}
}
```

- Display data on the screen:

To do this, you should first convert the array of byte to string and then add the string to the *RichTextField* instance which you already created in the constructor.

```
String message = new String(raw, 0, readLength);

textfield.setText(message);
```

7. Now, the client application is ready. However, in order you be able to connect to the Internet using HTTP connection, you should enable MDS (Mobile Data Service), otherwise you can not surf the Internet from the handheld device. In order to enable MDS with the simulator, you should go to the "Edit" menu then click on "Preferences…". In the "Simulator" tab, select "General" tab and then select "Launch Mobile Data Service (MDS) with simulator".

   Edit > Preferences…> Simulator > General > Launch Mobile Data Service (MDS) with simulator. **Note** that the MDS simulator package doesn't come with the BlackBerry JDE. It has to be downloaded and installed separately from: http://na.blackberry.com/en/developers/browserdev/devtoolsdownloads.jsp.

8. Demo your work to the TA. [**5 marks**]

Listing 1 – Lab9.java:

```
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import javax.microedition.io.*;
import java.io.*;

class Lab9 extends net.rim.device.api.ui.UiApplication implements Runnable {
   MainScreen screen;
   RichTextField textfield;

   public Lab9() {
      super();
      screen  = new MainScreen();
      textfield = new RichTextField("Google.ca \n");
      screen.add(textfield);
      pushScreen(screen );

      Thread t = new Thread(this);
      t.start();
   }
    public static void main(String[] args) {
      Lab9 theApp = new Lab9();
      theApp.enterEventDispatcher();
```

```
    }

    public void run() { // HTTPMIDlet's run() method connects to a server, retrieves some
data, and shows it on the screen.
        String url = "http://www.google.ca/";

        HttpConnection hc = null;
        InputStream in = null;

        try {
            // Now make a connection to the server.
            hc = (HttpConnection)Connector.open(url);

            in = hc.openInputStream();

            byte[] raw = new byte[1024];
            int readLength = in.read(raw);

            String message = new String(raw, 0, readLength);

            textfield.setText(message);
        }
        catch (Exception e) {
            System.out.println(e);
        }
        finally {
            if (in != null) {
                try { in.close(); }
                catch (IOException ioe) {}
            }
            if (hc != null) {
                try { hc.close(); }
                catch (IOException ioe) {}
            }
        }
    }
}
```

Exercise 1 Revising the Experiment #1

When you run the application developed in the Experiment #1, it will throw an IllegalStateException. This occurs because of the line *textfield.setText(message)*. Technically, the BlackBerry UI components are run from the main event thread. Thus, if you want to access them from a thread other than the main event thread, the exception will be thrown. In the Experiment #1, you try to access them from the thread created for the network connection which is not the main thread. One of the solutions is to place the screen changes in the screen's event queue. For this purpose, use invokeLater(). It adds the screen change to the application's event queue to be executed when it is possible. Calling it repeatedly may cause the event queue to overflow.

In this exercise, you should re-implement the experiment #1, using this method. The following illustrates how to use the invokeLater () method: [**5 marks**]

```
UiApplication.getUiApplication().invokeLater (new Runnable() {
        public void run()
        {
                //Add the screen change methods here
                ……

                //Call the screen's invalidate method to force the screen to redraw itself.

                  screen.invalidate();

        }
});
```