Chapter 8 Network Communication

8.1 Introduction

Devices connected to the Internet use networking protocol TCP/IP to communicate. The protocol is divided into layers as shown in Figure 8-1 below. At the top of the model is the application layer, which can be any Internet applications such as creating graphics, streaming videos, and searching information. At the bottom is the physical layer, which deals with the actual transfer of the data bits. Below *application* is the transport layer that controls network traffic and the TCP/IP model provides the *Transmission Control Protocol (TCP)* and the *User Datagram Protocol (UDP)* for applications to communicate. The *network* layer is responsible for the routing of a packet, which consists of the source and the destination addresses. The *data link* layer is responsible for handling errors when transferring the data. In the model, a layer is only allowed to communicate with its adjacent layers directly. For example, the *network* layer is only allowed to 'talk' to the *transport* layer and the *data link* layer.

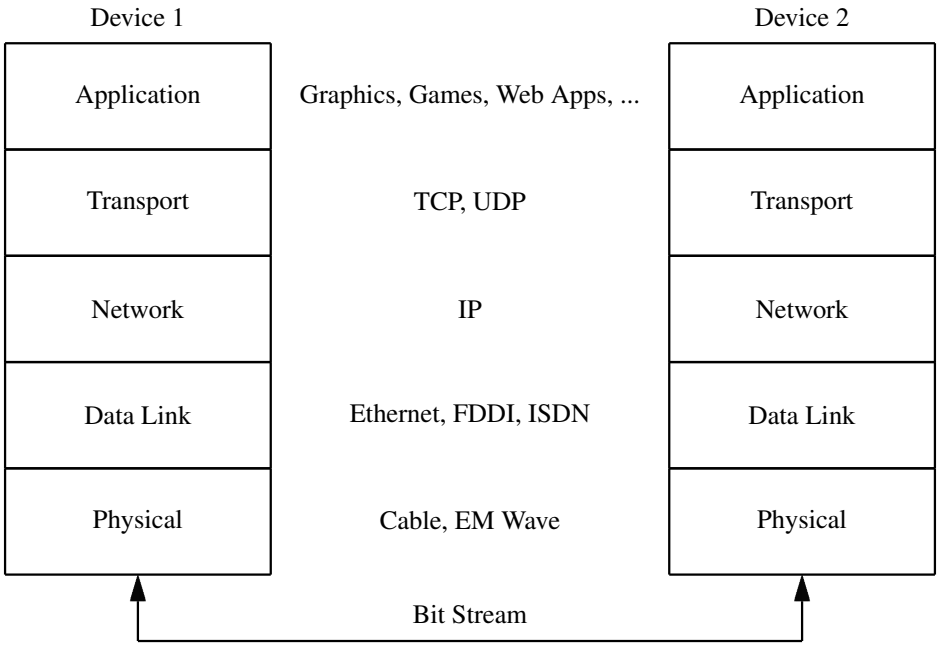


Figure 8-1 TCP/IP Layering Model

Normally we program at the application level and do not need to know the details of the low level layers. Traditionally, people write Internet communication programs in C/C++, calling several *socket* functions to establish a communication channel, which is fairly complicated to program. However, Java has largely simplified the coding of such applications. Developing a simple client-server application may just involve a few lines of codes. We explain a few terms commonly used in TCP/IP data communication below.

TCP

TCP (Transmission Control Protocol) is a connection-oriented protocol, where data packets are transmitted along a fixed established route between the sender and the receiver. It provides a reliable flow of data between two computers by requiring the receiver to send an acknowledgment to the sender, after it has received a data packet.

The protocol is at the *Transport Layer* as shown in Figure 8-1. When two applications communicate to each other reliably, they first establish a connection route and send data back and forth over that route. This is analogous to making a telephone call to a friend, in which a connection is frist established by your dialing a phone number and your friend picking up her phone. Data are sent back and forth over the connection when both of you speak to one another over the phone lines. TCP guarantees that data received are in the same order as they were sent.

The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that are built on top of TCP, which provides a point-to-point reliable communication channel. In many applications, the order in which the data are sent and received over the network is critical, otherwise the information received will be invalid.

UDP

UDP (User Datagram Protocol) is a connectionless-oriented protocol that transmits independent blocks of data, called **datagrams**, from one computer to another with no guarantees about arrival. No fixed route is established during the transmission. It is possible that different data blocks take different routes to get to the receiver from the sender. This is in analogous to sending letters through the postal service, in which the order of delivery is not important and first-come-first-served is not guaranteed, and each letter is independent of any other. The protocol is also at the *Transport Layer*.

For the applications that do not require strict standards of reliability and order of delivery, the UDP can provide faster service as it does not need the extra overhead to meet the strict requirements. For example, a clock server that sends the current time to a client that requests it does not really need to resend a packet that the client misses as the client can make another request later. If the client makes two requests and receives the packets out of order, it does not matter much because the client will find the packets out of order and could make another request. By neglecting reliability requirements of TCP, the server can improve its performance. Another example of a service that does not need reliable communication channel is the *ping* command, which is to test the connection between two machines over the network. Actually *ping* needs to know about out-of-order or dropped packets to determine how good a connection is. A reliable protocol would invalidate this service altogether. Also, in a Local Area Network (LAN) the communication medium is very reliable. Packets rarely lose. So it is desirable to use an unreliable protocol such as UDP in many LAN applications. On the other hand, the physical communication links in the Internet are not very reliable as they extend over very large areas. So it makes sense to use a reliable protocol such as TCP in many Internet applications.

Sockets

When we write Android network applications, we program at the application level, often utilizing the *socket* API to send and receive data. A **socket** is a software endpoint that establishes bidirectional communication between a server program and one or more client programs. The socket associates the server program with a specific hardware port on the machine where it sends and receives data. A client socket has to associate with the same port if it wants to communicate with the server. The socket API was first developed by UC Berkeley in the 1980s for UNIX systems and

is open-source. Later it was adopted by other organizations and platforms. The original Berkeley sockets are referred to as BSD sockets.a As the API has evolved with little modification from a de facto standard into part of the POSIX specification, POSIX sockets are basically BSD sockets.

A server program typically provides services or resources to a number of client programs. A client sends requests to the server, which responds to them. To handle multiple requests from multiple clients, the server can create one thread dedicated to servicing each client. A multi-threaded server program can accept a connection from a client, create and start a thread for that communication, and continue to listen for requests from other clients.

Ports

Physically, a computer connects to a network through a single medium. How can a process (a running program) communicate with several different processes at the same time? The trick is to use ports, which are virtualised endpoints. The virtualisation makes multiple concurrent connections on a single network interface possible. Each process that communicates with another process identifies itself to the TCP/IP protocol suite by one or more ports. A port is specified by a 16-bit number. The purpose of specifying ports is to differentiate multiple endpoints on a given network address. Strictly speaking, an endpoint (socket) is identified by an IP address and a port number. The TCP and UDP protocols use ports to associate incoming data to a particular process running on a computer as shown in Figure 8-2 below.

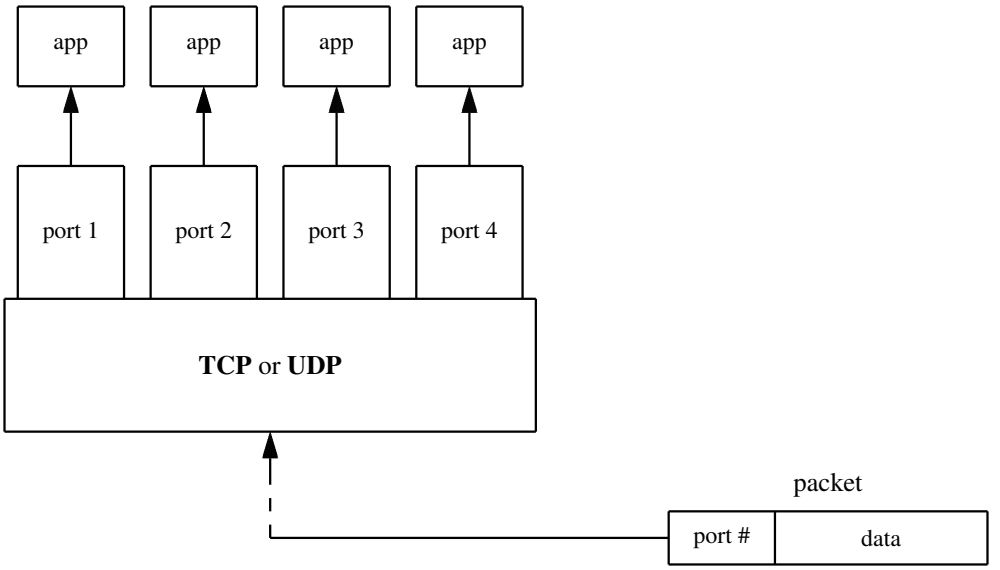


Figure 8-2 TCP/IP Ports

In the following sections, we will present a few examples to illustrate how Android devices communicate in a network.

8.2 Connecting to a Server

The first example that we will discuss is about connecting an Android device to a server program running in a desktop machine and transmitting a message using a socket. Such a server program is very simple if written in Java. It can be run in any common platform such as Linux, MS Windows, or Mac OS/X, though we run it in a 64-bit Linux machine. The following is the complete code for

this program.

Program Listing 8-1 *DemoServer.java*

```
// A simple TCP server for Demo

import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class DemoServer {
  public static void main(String[] args) throws IOException {

    if (args.length != 1) {
      System.err.println("Usage: java Server <port number>");
      System.exit(1);
    }

    int portNumber = Integer.parseInt(args[0]);
    try {
      ServerSocket serverSocket = new ServerSocket(portNumber);
      Socket clientSocket = serverSocket.accept();
      BufferedReader input = new BufferedReader (
         new InputStreamReader(clientSocket.getInputStream()));

       String inputLine = null;
       while ( ( inputLine = input.readLine() ) != null ) {
         System.out.println ( inputLine );
       }
    }  catch (IOException e) {
         System.out.println("Exception caught on listening on port "
                + portNumber );
         System.out.println(e.getMessage());
    }
  }
}
```

In the code, the statement

ServerSocket serverSocket = new ServerSocket(portNumber);

creates an endpoint at the specified port number, which is provided by the user as an input argument to the program. *ServerSocket* is a *java.net* class that provides a system-independent implementation of the server side of a client/server socket connection. If the server successfully binds to the specified port, the *ServerSocket* object (*serverSocket*) is successfully created.

The next statement,

Socket clientSocket = serverSocket.accept();

tells the server process to listen to the port for any request from a client and accepts the connection if everything has been done properly. If no request has been detected, it just continues to listen. TCP is used but Java has hidden all the handshaking and house keeping details from the user,

and has simplified the coding of using sockets. When a connection is successfully established, the **accept** method returns a new *Socket* object (*clientSocket*), which is bound to the same local port and has its remote address and remote port set to that of the client. The server process can communicate with the client process over this new *clientSocket* and continue to listen for client connection requests on the original *serverSocket*. However, this simple program can only handle one client at a time. If we want to handle multiple clients, we need to modify the program.

The subsequent few lines of code simply tell the server read in data from the client and print them out on the console.

We can compile the program with the command *javac DemoServer.java*, which generates the class *DemoServer.class*. Suppose we choose the port number to be 1989. We can execute the program using the following command:

```
$ java DemoServer 1989
```

The process then waits and listens for requests at port 1989.

To communicate with the server, the client also needs to know the IP address of the server, which can be found out by executing the command *ifconfig* in the console of a Linux machine or *ipconfig* in MS Windows. The IP of the machine we used in this example is *192.168.1.69*.

The Android client program that sends messages to the server is also simple. The following are the steps of developing this client program in the Eclipse IDE. Suppose we call the project and application *Client*.

1. As usual, in Eclipse IDE, create the project and application *Client* with package name *comm.client*.
2. Grant Internet access permissions to the application by adding a couple of *uses-permission* tags in the file *AndroidManifest.xml* like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
.....
<uses-permission android:name="android.permission.INTERNET" >
</uses-permission>

<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE">
</uses-permission>
.....
</manifest>
```

3. Modify the file *res/layout/activity_main.xml* to the following, which defines the UI layout to display a button and to accept text input.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
  <EditText
        android:id="@+id/EditText1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Hello, Friend." >
  </EditText>
  <Button
        android:id="@+id/myButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:onClick="onClick"
        android:text="Send" >
  </Button>
</LinearLayout>
```

4. Modify the program *MainActivity.java* to the following.

```
--------------------------------------------------------------------------
package comm.client;

import android.app.Activity;
import android.os.Build;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.BufferedWriter;
import java.io.OutputStreamWriter;
import java.net.Socket;
import java.net.InetAddress;
import java.net.UnknownHostException;

public class MainActivity extends Activity {
  private Socket socket;
  private static final int PORT_NO = 1989;
  // Need to change IP address to the IP of your server
  private static final String SERVER_IP = "192.168.1.69";
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    new Thread(new ClientThread()).start();
  }

  public void onClick(View view) {
    try {
      EditText editText = (EditText) findViewById(R.id.EditText1);
      String str = editText.getText().toString();
      PrintWriter out = new PrintWriter(new BufferedWriter(
      new OutputStreamWriter(socket.getOutputStream())), true);
      out.println(str);
    } catch (UnknownHostException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  // inner class
  class ClientThread implements Runnable {
    @Override
    public void run() {
      try {
```

```
            InetAddress serverAddr=InetAddress.getByName(SERVER_IP);
            socket = new Socket(serverAddr, PORT_NO );
          } catch (UnknownHostException e) {
            e.printStackTrace();
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
      }
}  //class MainActivity
```

In the code, the port number and the IP address of the server are hard-coded. The main class *MainActivity* defines a *Socket* class variable called *socket* as a data member:

private Socket socket;

The actual creation of a *Socket* object to communicate with the server is done by the thread *ClientThread*. The **onCreate**() method of *MainActivity* creates and starts this thread by the statement:

new Thread(new ClientThread()).start();

The thread class *ClientThread* is implemented as an inner class of *MainActivity*. Inner classes have full access to the class in which they are nested, and in our case, the inner class *ClientThread* can access all the data members, *socket, PORT_NO*, and *SERVER_IP* defined in *MainActivity*. It creates the communication channel to the server with the statements:

InetAddress serverAddr=InetAddress.getByName(SERVER_IP);
socket = new Socket(serverAddr, PORT_NO);

The method **onClick**() of *MainActivity* defines an editable text field and a button. When the button is clicked, the text in the text field is sent to the server through the communication channel, which is achieved by the statements:

EditText editText = (EditText) findViewById(R.id.EditText1);
String str = editText.getText().toString();
PrintWriter out = new PrintWriter(new BufferedWriter(
new OutputStreamWriter(socket.getOutputStream())), true);
out.println(str);

When we run the program, the Android UI shows a *Send* button and an editable text field as shown in Figure 8-3 below.

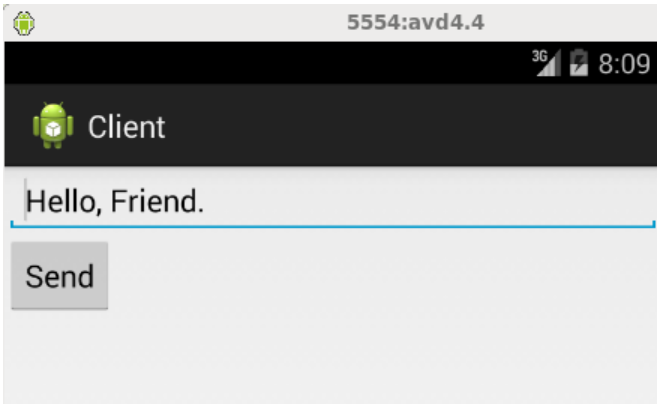


Figure 8-3 Client UI

We can type in any text message in the text area. Upon clicking the *Send* button, the message is sent to the server, which then reads in the text and prints it to the console. The following

shows the operation on the server side and sample text received from the Android client.

```
$ java DemoServer 1989
Hello, Friend.
Android the Beautiful!
```

8.3 Communication Between Android Devices

We have discussed how to send a message from an Android device to a server running on a PC in the previous section. Suppose now we want to send messages from an Android client to an Android server and we want to do the test in AVD emulators. In this case, the client program is the same as the one presented above except that the hard-coded IP address needs to be changed to the one that an AVD emulator runs, which is 10.0.2.2. That is, in the program *MainActivity*.java of the *Client* project discussed above, we need to modify the statement specifying the IP address to:

```
private static final String SERVER_IP = "10.0.2.2";
```

We will still use the port number 1989 for the client.

The Android server program is different from the server program running on a PC as it is run as an Android application. Suppose we develop this program in the Eclipse IDE and name the project and application *Server* and the package *comm.server*. Like what we did to the *Client* application described above, we have to grant Internet access permission to the *Server* application; this is done by adding <uses-permission> statements in the file *AndroidManifest.xml* (see the *Client* project description in the previous section). The listing below shows the complete Java code of *MainActivity.java* of the server program.

Program Listing 8-2 *MainActivity.java* of Server

```
package comm.server;
import android.os.Build;
import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.widget.TextView;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class MainActivity extends Activity {

  private ServerSocket serverSocket;
  private String str;
  Handler textHandler;
  Thread serverThread = null;
  private TextView textView;
  public static final int portNumber = 2014;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    setContentView(R.layout.activity_main);
    textView = (TextView) findViewById(R.id.text1);
    textHandler = new Handler();
    new Thread(new ServerThread()).start();
  }

  @Override
  protected void onStop() {
    super.onStop();
    try {
      serverSocket.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  //Inner class
  class ServerThread implements Runnable {
    public void run() {
      Socket commSocket = null;
      try {
        serverSocket = new ServerSocket(portNumber);
      } catch (IOException e) {
        e.printStackTrace();
      }
      while (!Thread.currentThread().isInterrupted()) {
        try {
          commSocket = serverSocket.accept();
          BufferedReader input = new BufferedReader (
              new InputStreamReader( commSocket.getInputStream() ));
          while ( ( str = input.readLine() )  != null )
            textHandler.post(new ShowText ());
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
  } //ServerThread

  // Inner class
  class ShowText implements Runnable {
    @Override
    public void run() {
      textView.setText(textView.getText().toString() +
          "Client said: " + str + "\n");
    }
  } //ShowText
} //MainActivity
```

In the code, the server port number is hard-coded to be 2014. We created a *Handler* object, which is referred to by the variable *textHandler*, to process any incoming message. A *Handler* allows us to send and process *Message* and *Runnable* objects associated with the *MessageQueue* of a thread. Normally, the main thread of a process is dedicated to running a message queue that manages

the top-level application objects such as activities, and broadcast receivers, and any windows they create. The application can create other threads, and communicate back with the main thread through a *Handler*. A newly created *Handler* is bound to the thread/message queue of the creating thread, delivering messages and runnables to the message queue and executing them as they come out of the message queue. Its method **post**() takes a thread as an argument and adds the thread to the message queue. In our code, *ShowText* is an inner class that displays text on the defined text area, and an object of it is passed as an argument to **post**().

ServerThread is another inner class that creates a socket with the specified port number, listening to incoming messages. When it reads a line, it uses *textHandler* to pass the information to the main thread. The statement

```
textHandler.post(new ShowText ());
```

creates a *ShowText* thread and adds it to the message queue. This thread then displays the message on the screen while *ServerThread* continues to listen to other messages.

To test the application, we first use **Eclipse** to run with one AVD the *Server* program, which uses port 2014. The client program will use port 1989. The AVD emulator running the server would use port 5554. We want to access this emulator and redirect client messages to its port. This can be done using **telnet** and its **redir** command. So the next step is to issue the telnet command in the machine that runs the AVD emulator like the following:

```
$ telnet localhost 5554
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Android Console: type 'help' for a list of commands
OK
redir add tcp:1989:2014
OK
```

The command *redir add tcp:1989:2014* redirects messages from port 1989 to port 2014. Also an AVD emulator runs on the alias IP 10.0.2.2.

Lastly, we run the client program using another AVD emulator (different from the one used to run the server program). When these are done, we see two emulators running on our PC, one for *Client* and one for *Sever*. When we enter text in the client and click the button, the text is sent to the server for display. Figure 8-3 below shows two emulators that run the client and server programs.

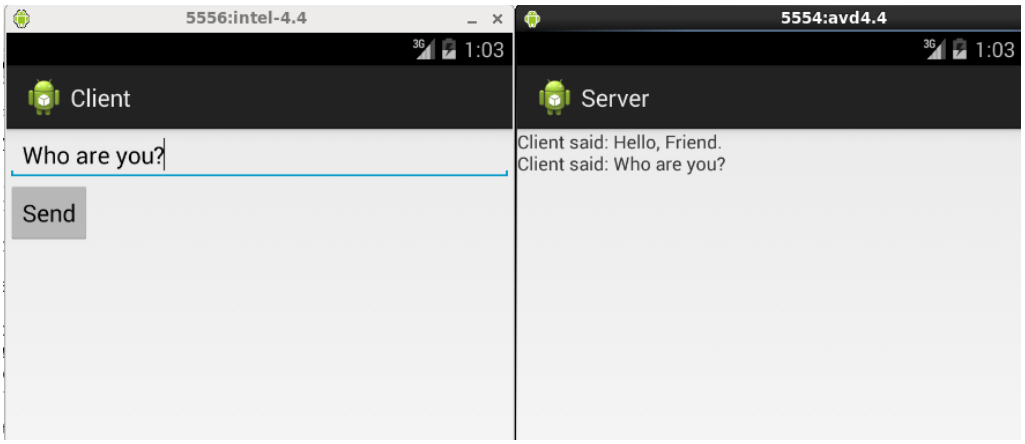


Figure 8-3 Running *Client* and *Server* with Two Emulators

8.4 A Remote Calculator

In the example of this section, we present a simple remote calculator, which works like a typical client-server application. The client is an Android program, managing the UI and interacting with the user. The server is a Java program running in a PC, performing the actual arithmetic calculations. The client accepts inputs from the user and sends them to the remote server, which carries out the calculations and returns the result to the client. Obviously, there are many ways to write such an application.

To introduce more programming techniques, we define a client thread using an independent external class instead of using an inner class like what we did did before.

8.4.1 JSON

We also introduce here the usage of **JSON**, which stands for *JavaScript Object Notation*, to pass data between the client and the server. JSON is a lightweight data-interchange format, easy for humans to read and write, and easy for machines to parse and generate. It is a languate independent text format based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition deployed in 1989, using conventions that are familiar to programmers of common languages such as C/C++, Java and Python. JSON is open-source, making it an ideal data-interchange language. It is built on two structures:

1. **A collection of name/value pairs**, which can be realized as an *object, record, struct, dictionary, hash table, keyed list*, or *associative array*.
2. **An ordered list of values**, which can be realized as an *array, vector, list*, or *sequence*.

As these are universal data structures, supported by virtually all modern programming languages, it makes sense to have a data format, which is interchangeable with programming languages, to be based on these structures.

In JASON, they take on one of the forms: *object, array, value*, and *string*. For details, please refer to its official web site at:

http://www.json.org/

In our application, we will only use the *object* form. An object is an unordered set of *name/value* pairs, starting with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the *name/value* pairs are separated by , (comma) as shown in Figure 8-4 below.

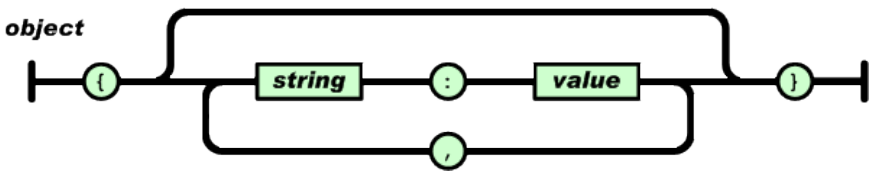


Figure 8-4 Object Form of JSON

We will use this form to pass data structures between the client and the server. For example, we can express the calculation of the sum of two numbers, $2 + 1$, by the text,

$$\{\text{“}calcOp\text{”} : \text{“}+\text{”}, \text{“}op1\text{”} : \text{“}2\text{”}, \text{“}op2\text{”} : \text{“}1\text{”}\}$$

where the string $calcOp$ indicates the calculation operator; the string $op1$ indicates the first operand, and $op2$, the second operand. The JSON package that we have used, *json-simple-1.1.1.jar*, is a simple version, which was downloaded from the site,

http://code.google.com/p/json-simple/

8.4.2 Server Running on PC

The application's server, which runs on a PC, carries out the calculations. The code of reading in a message from a socket is similar to the server code *DemoServer.java* presented in Listing 8-1 of Section 8.2. However, here the code has to parse the input line to a JSON object, extract the operator and the operands, perform the calculation and send the result back to the client. Suppose we call our server program *CalcServer.java*, and we assume that we will use port 1989 to communicate. The following lists the complete server code:

Program Listing 8-3 *CalcServer.java*

```
// Remote Calculator Server running on PC
import java.io.*;
import java.net.*;
import java.util.Arrays;
import org.json.simple.*;
import static java.lang.System.in;

public class CalcServer
{
  public static void main(String[] args) throws IOException
  {
    if (args.length != 1) {
      System.err.println("Usage: java CalcServer <port number>");
      System.exit(1);
    }
    int portNumber = Integer.parseInt(args[0]);
    JSONObject  jsonObject;
    try {
      ServerSocket serverSocket =  new ServerSocket ( portNumber );
      while ( true ) {
        System.out.println( "Waiting for Client!" );
        Socket socket = serverSocket.accept();
        String inputLine = null;
        // get input from socket
        BufferedReader input = new BufferedReader (
           new InputStreamReader(socket.getInputStream()));
        double result = 0.0;

        while  ( (inputLine = input.readLine() )  != null ) {
          System.out.print( "  Received request: " );
          System.out.println ( inputLine );
          jsonObject = (JSONObject) JSONValue.parse ( inputLine );
          result = calculate ( jsonObject );
          PrintWriter ow=new PrintWriter(socket.getOutputStream(),true);
          ow.println ( result );
          System.out.println( "  Result sent to client!");
        }
        socket.close();
      }  // while ( true )
    }  catch (IOException e) {
         System.out.println("Exception caught on listening on port "
                   + portNumber );
         System.out.println(e.getMessage());
```

```
            System.exit( 1 );
        }
    }

    public static double calculate ( JSONObject jsonObject )
    {
        double result = 0;
        // Extract operator and operands.
        String calcOp = (String) jsonObject.get("calcOp");
        double op1=Double.parseDouble((String)jsonObject.get("op1"));
        double op2=Double.parseDouble((String)jsonObject.get("op2"));

        // Do the actual calculation
        if (calcOp.equals( "+" ))
            result = op1 + op2;
        else if (calcOp.equals( "-"))
            result = op1 - op2;
        else if (calcOp.equals( "*" ))
            result = op1 * op2;
        else if (calcOp.equals( "/" ))
        if ( op2 != 0 )
            result = op1 / op2;
        System.out.printf( "  %f %s %f = %f : ",op1,calcOp,op2,result);

        return result;
    }
}
-------------------------------------------------------------------------
```

In the code, the **main** method creates a *ServerSocket* object with the specified port and listens to the port by creating a *Socket* object and using its **accept**() method to accept any incoming message as a string, and form a JSON object from the string by the statement,

jsonObject = (JSONObject) JSONValue.parse (inputLine);

It then calls the method **calculate**(), which takes a JSON object as the input parameter, to do the calculation. The **calculate** method parses the JSON object using the **get**() method of the JSON simple package to extract the operator and the two operands. It then does the arithmetic calculation and returns the result as a **double** to **main**, which sends the result to the client through the socket by the two statements:

PrintWriter ow = new PrintWriter (socket.getOutputStream(), true);
ow.println (result);

After the result has been sent, it closes the current *Socket* object using the statement

socket.close();

and goes back to the beginning of the while-loop to create another socket for listening and accepting another message.

Suppose we put the simple JSON package *json-simple-1.1.1* in the same directory as the server program, *CalcServer.java*. We can compile the server program in a PC with the following command,

```
$ javac  -cp .:./json-simple-1.1.1.jar CalcServer.java
```

and run it with the command,

```
$ java  -cp .:./json-simple-1.1.1.jar CalcServer 1989
```

where the input parameter 1989 is the port number that will be used by the server. The running program will respond with the message:

```
Waiting for Client!
```

while waiting for any request from a client.

8.4.3 Client Running on Android

The application's client program runs on Android. Suppose we call the project *CalcClient* and the package *comm.calcclient*. We divide the application into two classes. The main class *MainActivity* is for UI, accepting inputs and displaying results. The other class, named *Client* is for communicating with the server, sending the operation strings to and accepting results from the server. The following lists the code of the class *MainActivity*, omitting the import and package statements.

Program Listing 8-4 *MainActivity.java* of CalcClient Project

```
public class MainActivity extends Activity
                              implements View.OnClickListener
{
  public static double result = 0;
  public static String oper = "";
  public static String nums1;
  public static String nums2;

  private String str = "";
  private int displayCount = 0;
  EditText t1, t2;
  ImageButton plusButton, minusButton, multiplyButton, didvideButton;
  TextView displayResult;

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // find elements defined in res/layout/activity_main.xml
    t1 = (EditText) findViewById(R.id.t1);
    t2 = (EditText) findViewById(R.id.t2);
    plusButton = (ImageButton) findViewById(R.id.plusButton);
    minusButton = (ImageButton) findViewById(R.id.minusButton);
    multiplyButton = (ImageButton) findViewById(R.id.multiplyButton);
    didvideButton = (ImageButton) findViewById(R.id.divideButton);
    displayResult = (TextView) findViewById(R.id.displayResult);

    // set listeners
    plusButton.setOnClickListener(this);
    minusButton.setOnClickListener(this);
    multiplyButton.setOnClickListener(this);
    didvideButton.setOnClickListener(this);
  }

  // @Override
```

```
  public void onClick(View view) {
    // check if the fields are empty
    if (TextUtils.isEmpty(t1.getText().toString())
          || TextUtils.isEmpty(t2.getText().toString()))
      return;

    // read numbers from EditText
    nums1 = t1.getText().toString();
    nums2 = t2.getText().toString();

    // determine which image button has been clicked
    switch (view.getId()) {
      case R.id.plusButton:
        oper = "+";
        break;
      case R.id.minusButton:
        oper = "-";
        break;
      case R.id.multiplyButton:
        oper = "*";
        break;
      case R.id.divideButton:
        oper = "/";
        break;
      default:
        break;
    }
    // Create thread to send request
    Client client = new Client ( nums1, nums2, oper );
    Thread sendThread = new Thread ( client );
    sendThread.start();
    try {
      result = client.getResult();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    // form the output line, display at most 4 lines on screen
    if ( displayCount < 4 ){
      str += nums1 + " " + oper + " " + nums2 + " = " + result + "\n";
      displayCount++;
    } else {
      displayCount = 1;
      str = nums1 + " " + oper + " " + nums2 + " = " + result + "\n";
    }
    displayResult.setText( str );
  }
}
-------------------------------------------------------------------------
```

The UI part of the code is straight forward and easy to understand. The UI is defined in the layout file of *res/layout/activity_main.xml*, which is not presented here as it is similar to some of the layout files we have explained in previous chapters. (The complete code of this book can be downloaded from the site *hppt://www.forejune.com*.) In the UI, the user enters two numbers in the two *EditText* fields. Upon clicking an image button, the method **onClick**() is called; the method reads in the two numbers and chooses the corresponding operator of the image button. It

then creates a *Client* thread to send the operation string to the server, which returns the result of operation to the *Client* object. The **onClick** method calls the **getResult**() method of *Client* to get the operation result. It then displays the operation string and the result on the screen using the **setText**() method of *TextView*. (See Figure 8-5 below.)

The code of the *Client* class is fairly simple and is presented in Listing 8-5 below. Its method **remoteCalculation**() is responsible for sending the operation string in the JSON object form to the server, which returns the result of operation; the method saves the result in the **double** variable *result*.

The only task of the method **getResult**() is supposed to return the value of *result* as a double to whatever calls it. However, we cannot simply return *result* without checking the situation. This is because the method may be called before the result is available. That is, it is called before **remoteCaculation** has obtained the result from the server, and thus returning an invalid value. To prevent this from happening, we can use the **guard** concept we discussed in Chapter 7 to force the method to wait until the result is available. We define a *Condition* variable, *resultReady*, to enforce the waiting with the statement

```
resultReady.await();
```

It suspends the execution of the method until **resultReady.signal**() is executed by the method **remoteCalculation** or some other routines, after the result has been obtained from the server. Consequently, **getResult** always returns a valid result. However, we need to address one more issue – there could be situations that the **signal**() command is executed before **await**(), leaving it waiting forever. To guard against this situation, we define an integer variable, *count*. The method waits only if the value of *count* is 0, which is incremented by **remoteCalculation** when the result is available. This is illustrated in Figure 8-5 below:

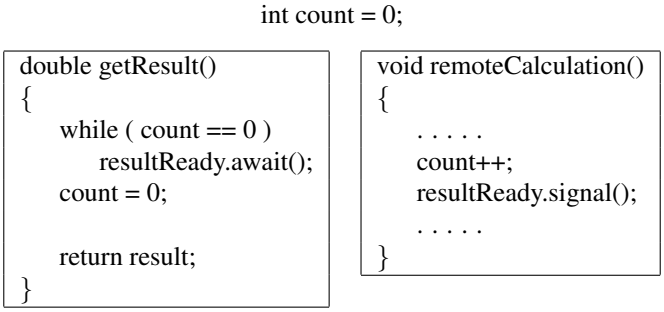


Figure 8-5 Enforcing Valid Result with Condition Variable

In our actual code, the *count* increment and the execution of **signal** are done in the **run**() method. The complete code of the *Client* class is listed below with import and package statements omitted.

Program Listing 8-5 *Client.java* of CalcClient Project

```
public class Client implements Runnable {
  private static Socket socket;
  private static String nums1;
  private static String nums2;
  private static String calcOp;
  private static double result = 0;
  final Lock mutex = new ReentrantLock();
  final Condition resultReady = mutex.newCondition();
  private int count = 0;
```

```
public Client ( String n1, String n2, String op )
{
  nums1 = n1;
  nums2 = n2;
  calcOp  = op;
}
@Override
public void run() {
  // TODO Auto-generated method stub

  String serverAddr = "192.168.1.69";
  int portNumber = 1989;

  mutex.lock();
  try {
    //socket = initiateContact(serverAddr, portNumber);

    remoteCalculation( serverAddr, portNumber );
    count++;
    resultReady.signal();
  } catch (UnknownHostException e1) {
      e1.printStackTrace();
  } catch (IOException e1) {
      e1.printStackTrace();
  } finally{
    mutex.unlock();
  }
}

public  void remoteCalculation(String serverAddr, int portNumbe)
 throws IOException
{
  Socket socket = null;
  socket = new Socket(serverAddr, portNumber);

  BufferedReader stdIn = new BufferedReader(new
                                 InputStreamReader( System.in));
  // Form JSON object string
  String jsonString = "{" + "\"calcOp\":\""
      + calcOp + "\"," + "\"op1\":\""
      + nums1 + "\"," + "\"op2\":\""
      + nums2 + "\"}";

  System.out.println( jsonString );
    sendMessage( socket, jsonString);
  String serverReply = receiveMessage( socket );
  System.out.println("Result: " + serverReply);
  result = Double.parseDouble((String) serverReply);
  stdIn.close();
  socket.close();
}

public double getResult()  throws InterruptedException
{
  mutex.lock();
```

```
    System.out.println("getResult wait");
    // The count is to ensure the function won't wait forever
    //   if signal has been issued before  await.
    while ( count == 0 )
      resultReady.await();   // Condition wait
    count = 0;
    mutex.unlock();

    return result;
  }

  // Receive message from a socket
  public static String receiveMessage(Socket socket)
                          throws IOException {
    String inputLine = null;
    BufferedReader inputBuffer = null;
    inputBuffer = new BufferedReader( new InputStreamReader
                                  ( socket.getInputStream()));
    inputLine = inputBuffer.readLine();

    return inputLine;
  }

  // send message to socket
  public static void sendMessage(Socket socket, String outputLine)
        throws IOException {
    PrintWriter outputWriter = null;
    outputWriter = new PrintWriter(socket.getOutputStream(), true);
    outputWriter.println(outputLine);
  }
}
-----------------------------------------------------------------------
```

When we run the code in Android, we will see an UI like the one shown in Figure 8-6. The text below the image buttons are the calculation expressions we have entered and the results returned by the server.

Also, correspondingly, the server will print out some messages. While the client is running in Android, the server running in a PC will display output messages similar to the following, which tells the user it receives requests from the client, does the calculations, and returns the results.

```
Waiting for Client!
  Received request: {"calcOp":"+","op1":"1989","op2":"64"}
  1989.000000 + 64.000000 = 2053.000000 :   Result sent to client!
Waiting for Client!
  Received request: {"calcOp":"-","op1":"1989","op2":"64"}
  1989.000000 - 64.000000 = 1925.000000 :   Result sent to client!
Waiting for Client!
  Received request: {"calcOp":"*","op1":"1989","op2":"64"}
  1989.000000 * 64.000000 = 127296.000000 :   Result sent to client!
Waiting for Client!
  Received request: {"calcOp":"/","op1":"1989","op2":"64"}
  1989.000000 / 64.000000 = 31.078125 :   Result sent to client!
Waiting for Client!
```

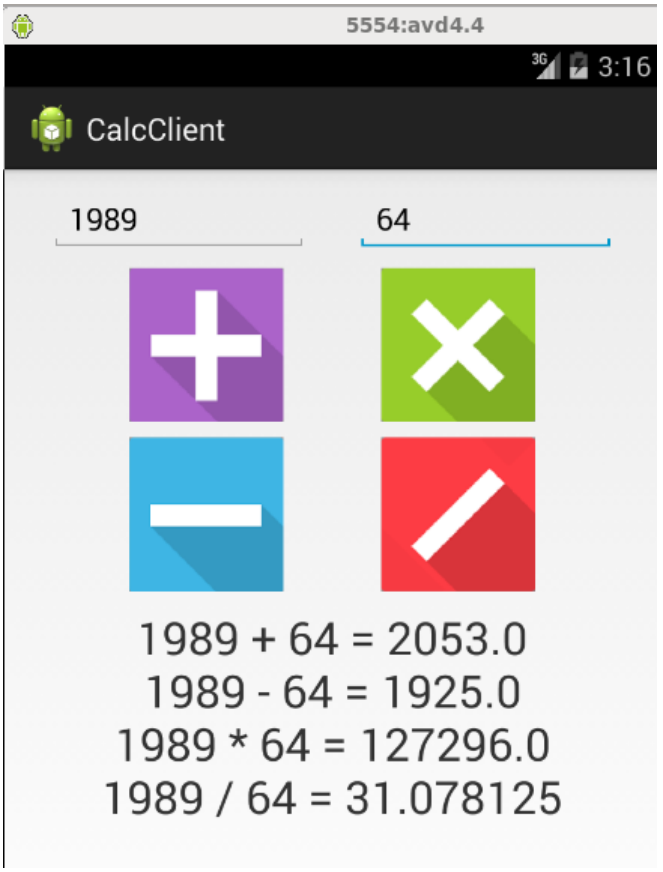


Figure 8-6 Sample I/O Display of Client Calculator

8.5 Broadcast Receiver

A broadcast receiver (or receiver for short) is an Android component for detecting system or application events, responding to system-wide broadcast announcements, such as low battery level or file downloading done. A broadcast receiver is implemented as a subclass of *BroadcastReceiver* and each broadcast is delivered as an *Intent* object. *BroadcastReceivers* are one of Androids four standard app component types (activities, services, content providers, and broadcast receivers), each of which serves a distinct purpose and has a distinct life cycle that defines how the component is created and destroyed. Details of the *BroadcastReceiver* class can be found at

http://developer.android.com/reference/android/content/BroadcastReceiver.html

A broadcast receiver must first be registered before it will receive any announcement. All registered receivers for an event are notified by the Android runtime once the event occurs. For example, an application can register for the ACTION_BOOT_COMPLETED system event, which occurs when the Android system has completed the boot process.

We can register a receiver statically via the *AndroidManifest.xml* file or dynamically via the *Context*.**registerReceiver**() method.

Implementation of a broadcast receiver consists of two steps:

1. declaring a subclass of *BroadcastReceiver*, and
2. implementing the **onReceive**() method.

The following sample code shows the form of the implementation:

```
import android.content.Context;
import android.content.Intent;
import android.content.BroadcastReceiver;

public class MyReceiver extends BroadcastReceiver
{
  @Override
  public void onReceive(Context context, Intent intent) {
    // Response to an  event
  }
}
```

The Android runtime calls the **onReceive**() method on all registered receivers whenever the event occurs. This method takes two parameters:

Parameter	
context	The *Context* object on which the receiver is running. We can use it to access additional information or to start services or activities.
intent	The *Intent* object being received. It contains additional information that we can use in our implementation.

System-wide Events

A lot of system events are defined as final static fields of the *Intent* class. Furthermore throughout the API there are many more classes that offer specific broadcast events themselves. Some examples are BluetoothDevice or TextToSpeech.Engine and nearly all the Manager classes like UsbManager or AudioManager. Android really offers plenty of events that we can make use of in our apps.

The following list is only a small sample of all available events.

Table 8-1 System-Wide Events Samples

Event	**Broadcast Action**
Intent.ACTION_BATTERY_LOW	The battery level is low.
Intent.ACTION_BATTERY_OKAY	The battery level is good again.
Intent.ACTION_BOOT_COMPLETED	System has finished booting. Requires android.permission.RECEIVE_BOOT_COMPLETED.
Intent.ACTION_DEVICE_STORAGE_LOW	Storage space on the device is low.
Intent.ACTION_DEVICE_STORAGE_OK	Storage space on the device is good again.
Intent.ACTION_HEADSET_PLUG	A headset has been plugged in or removed.
Intent.ACTION_INPUT_METHOD_CHANGED	An input method has been changed.
Intent.ACTION_LOCALE_CHANGED	The language of the device has been changed.
Intent.ACTION_MY_PACKAGE_REPLACED	The app has been updated.
Intent.ACTION_PACKAGE_ADDED	A new app has been added to the system.
Intent.ACTION_POWER_CONNECTED	Has connected to external power.
Intent.ACTION_POWER_DISCONNECTED	The device has been upplugged from power.
KeyChain.ACTION_STORAGE_CHANGED	The keystore has been changed.
BluetoothDevice.ACTION_ACL_CONNECTED	Has established a Bluetooth ACL connection.
AudioManager.ACTION_AUDIO_BECOMING_NOISY	The internal audio speaker will be used, not other output means like a headset.

Registration in Manifest File

We can statically register and configure a broadcast receiver in the manifest file, *AndroidManifest.xml* using the <receiver> element and we can specify what event the receiver should react to using the nested element <intent-filter>.

Registration in Java Program

Alternatively, we can register a *BroadcastReceiver* object dynamically in our Java program by calling the **registerReceiver**() method on the corresponding *Context* object.

The following sample code shows the form of the implementation, using the system event *ConnectivityManager*.CONNECTIVITY_ACTION, which responds to a change in network connectivity, as an example:

```
import android.content.BroadcastReceiver;
import android.content.IntentFilter;
import android.net.ConnectivityManager;

public class MainActivity extends Activity
{
  BroadcastReceiver receiver;
  IntentFilter intentFilter;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    receiver = new MyReceiver();
    intentFilter =
       new IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION);
    ....
  }
  @Override
  protected void onPause() {
    unregisterReceiver(receiver);
    super.onPause();
  }

  @Override
  protected void onResume() {
    registerReceiver( receiver, intentFilter );
    super.onResume();
  }
}
```

The method **registerReceiver**() takes two parameters:

Parameter	
receiver	The *BroadcastRegister* we want to register.
intentFilter	The *IntentFilter* object specifying the event that the *receiver* should listen to.

If in our example we just simply print a message to the *Log* in response to a network change, our receiver would like like the following:

```
public class MyReceiver extends BroadcastReceiver
{
  @Override
  public void onReceive(Context context, Intent intent) {
    Log.v("MyReceiver", "Network connectivity changed!");
  }
}
```

8.6 Fetch Data From a Web Site

We have discussed *services* and the usage of services for process communications in Chapter 3. In this section, we present an example that demonstrates the usage of a service to fetch data from the Internet. We call the project and application *FetchData* and the package *comm.fetchdata*.

An *Activity* of the application displays a button for the user to click on to download a file from a Web site. When it is done, the *Service* notifies the *Activity* through a broadcast receiver. Upon receiving the notification, the receiver flashes a *Toast* message on the screen.

In the example, the file name and the URL of the Web site are hard-coded. The data downloaded are saved in an external storage using a method discussed in Chapter 5. After we have used the Eclipse IDE to create the project and default files, we need to modify the manifest file *AndroidManifest.xml*, the layout file *res/layout/activity_main.xml* and the main Java program *MainActivity.java*. In addition, we will create two class files, *FetchData.java*, which is an *Activity* responsible for downloading and saving the data in the background, and *DataReceiver.java*, which is a receiver that flashes a finishing message on the screen using the *Toast* class.

8.6.1 Using a Broadcast Receiver

We will use a broadcast receiver as we have explained in Section 8.5 to communicate with the main activity. The *MainActivity* creates a broadcast receiver, *DataReceiver* and dynamically register it for the event of finishing download. The service *FetchData* generates a signal when the event occurs, and broadcast the signal to all the registered receivers using the method **sendBroadcast**(). The following figure shows this communication scheme.

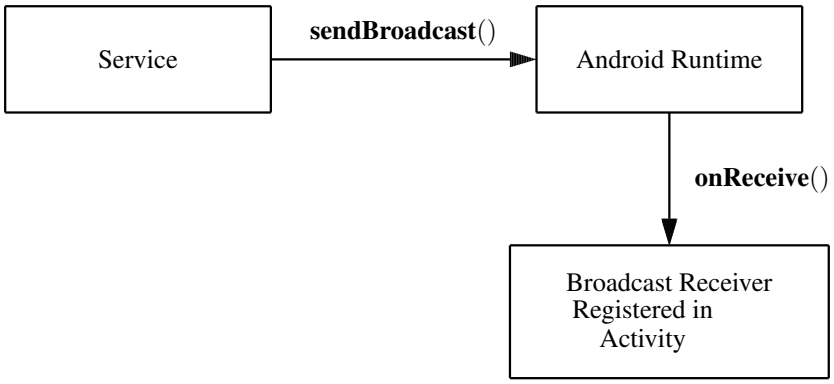


Figure 8-7 Communication Using Broadcast Receiver

The listing below shows the implementation of *DataReceiver* of the project.

Program Listing 8-6 *DataReceiver.java* of *FetchData* Project

```
package comm.fetchdata;

import android.app.Activity;
import android.content.*;
import android.os.Bundle;
import android.widget.Toast;

public class DataReceiver extends BroadcastReceiver
{
 @Override
 public void onReceive(Context context, Intent intent) {
   Bundle bundle = intent.getExtras();
   if (bundle != null) {
     String string = bundle.getString(FetchData.FPATH);
     int resultCode = bundle.getInt(FetchData.RESULT);
     if (resultCode == Activity.RESULT_OK) {
       Toast.makeText(context, "Fetch complete. Store URI: "
                    + string, Toast.LENGTH_LONG).show();
     } else {
       Toast.makeText(context, "Fetch failed",
             Toast.LENGTH_LONG).show();
    }
   }
 }
}
```

The method **onReceive**(*Context*, *Intent*) of this class is invoked when *FetchData* calls **sendBroadcast**(*Intent*). *FetchData* also creates an *Intent* to pass data to *DataReciever* through the *Intent* parameter of **sendBroadcast** to **onReceive**, which extracts the data using the method **getExtras**() of the *Bundle* class. The other parameter, *Context* of **onReceive** is the *Context* of the class (*MainActivity* in this example) that registers *DataReceiver*. Through this *Context* parameter, the receiver can exchange data with *MainActivity*.

8.6.2 Connecting to and Accessing a URL

In this example, the *FetchData* project, we make use of Java's *URL* class of the *java.net* package to connect the Android device to an Internet site. In general, URL is the acronym for *Uniform Resource Locator*, an address that references a resource on the Internet. We provide URLs to a Web browser to locate files on the Internet. The term URL may be ambiguous. It may refer to an Internet address or a URL object in a Java program. To avoid this ambiguity, we follow the convention used by the Java official site. When the meaning of URL needs to be specific, we use "URL address" to refer to an Internet address and "URL object" to an instance of the URL class in a program. Moreover, if we use italics, a *URL* is referring to a URL object or URL class.

Creating a URL

A URL object can be created with a URL constructor, which may take one argument as a string that specifies a URL address. Or it may take two arguments, one of which is a URL object referring to the base URL address of a web site; the other is a string specifying an address relative to the base address. The general forms of the constructors are:

1. URL (String *url_address*);
2. URL (URL *baseURL*, String *relative_url*);

The following are some examples:

1. URL myURL = new URL (“http://www.forejune.com/index.html”);

2. ULR baseURL = new URL (“http://www.forejune.com/”);
   URL url1 = new URL (baseURL, ”index1.html”);
   URL url2 = new URL (baseURL, ”index2.html”);

The class *URL* provides a few methods to parse a URL. For example, methods **getHost**(), **getPort**(), and **getPath**() return the hostname, the port number, and the path components of the URL respectively.

Connecting to and Accessing a URL

We can connect to a URL by first using method **openConnection**() of the class *URL* to open and start a connection. The method returns an object of the class *URLConnection*, which has several method for establishing and initializing a communication link between our Java program and the URL address over the network. We will use the method **connect**() of the class *URLConnection* to establish a connection. If the connection is established successfully, we can treat it as a data stream and use one of the many Java I/O streaming methods to process the data of the remote Web site. The following simple code is a typical example of establishing a link to and reading data from a URL address; actually, it is a complete Java program that saves the downloaded data in the file *downloaded.txt*.

```
// ReadURL.java: Compile: javac ReadURL.java
// Execute: java ReadURL
import java.io.*;
import java.net.URL;

public class ReadURL {
  public static void main ( String args[] )
  {
    String urlAddress = "http://www.forejune.com/";

    InputStream is = null;
    FileOutputStream fos = null;
    try {
      URL url = new URL( urlAddress );
      is = url.openConnection().getInputStream();
      InputStreamReader isr = new InputStreamReader ( is );
      fos = new FileOutputStream( "downdloaded.txt" );
      int next = -1;
      while ((next = isr.read()) != -1)
        fos.write(next);
    } catch (Exception e) {
       e.printStackTrace();
    }
  }
}
```

The class file *FetchData.java* of our *FetchData* project uses the same technique to connect to a hard-coded URL, and read the data, saving them in an external storage with a specified filename.

Program Listing 8-7 below shows the code of *FetchData.java*.

Program Listing 8-7 *FetchData.java* of *FetchData* Project

```
package comm.fetchdata;

import java.io.*;
import java.net.URL;
import android.os.*;
import android.net.Uri;
import android.util.Log;
import android.widget.Toast;
import android.app.Activity;
import android.content.Intent;
import android.app.IntentService;

public class FetchData extends IntentService
{
  private int result = Activity.RESULT_CANCELED;
  public static String URL_ADDRESS = "http://www.forejune.com";
  public static String FNAME = "downloaded.html";
  public static String FPATH = "./";
  public static String RESULT = "result";
  public static  String NOTIFICATION = "comm.fetchdata.receiver";

  public FetchData() {
    super("FetchData");
  }

  // will be called asynchronously by Android
  @Override
  protected void onHandleIntent(Intent intent) {
    String urlUsed = intent.getStringExtra( URL_ADDRESS );
    String fileName = intent.getStringExtra(FNAME);

    File output = new File(Environment.getExternalStorageDirectory(),
       fileName);
    if (output.exists()) {
      output.delete();
    }

    InputStream is = null;
    FileOutputStream fos = null;
    Toast.makeText(FetchData.this, "Testing FetchData",
                        Toast.LENGTH_LONG).show();
    try {
     URL url = new URL ( urlUsed );
     is = url.openConnection().getInputStream();
     InputStreamReader isr = new InputStreamReader( is );
     fos = new FileOutputStream(output.getPath());
     int next = -1;
     while ((next = isr.read()) != -1)
       fos.write(next);
     result = Activity.RESULT_OK;
```

```
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      if ( is != null) {
        try {
          is.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
      if (fos != null) {
        try {
          fos.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
    notifying(output.getAbsolutePath(), result);
  }

  private void notifying( String outputPath, int result ) {
    Intent intent = new Intent(NOTIFICATION);
    intent.putExtra(FPATH, outputPath);
    intent.putExtra(RESULT, result);
    sendBroadcast(intent);
  }
}
```

This class, *FetchData*, extends *IntentService* so that it can receive data from or send data to the main class that starts it. *IntentService* is a base class for *Service* objects that handle asynchronous requests, which are expressed as *Intents*, on demand. Typically, clients send requests with **startService**(*Intent*) calls to start the service when needed. The method handles each *Intent* object in turn using a worker thread, and stops itself when it runs out of work. These usually work in the background. When the work is done, it signals the *DataReceiver* using the **sendBroadcast**() method, which is called inside the **notifying**() method of *FetchData*, where another *Intent* object is created to pass some relevant information to the receiver, *DataReceiver*.

In the code, the *String* variables *URL_ADDRESS*, and *FNAME* are initialized when they are declared as data members in the class *FetchData*. However, their values are not used and the actual values are supplied by the main class, *MainActivity* in the **onHandleIntent**() method through the statements,

```
String urlUsed = intent.getStringExtra( URL_ADDRESS );
String fileName = intent.getStringExtra ( FNAME );
```

To supply the values, the *MainActivity* class should have corresponding statements of **putStringExtra**() like the following:

```
intent.putExtra( FetchData.FNAME, "downloaded.txt");
intent.putExtra( FetchData.URL_ADDRESS,
    "http://www.forejune.com/index.html");
```

The class *FetchData* implements the **onHandleIntent** (*Intent*) method of *IntentService*. The method is invoked on the worker thread with a request to process. Only one *Intent* object is processed at one time, but each worker thread runs independently, not relying on other requests. When all requests have been handled, the *IntentService* stops itself. The parameter of **onHandleIntent**() is the *Intent* object passed to the method **startService**(*Intent*), which is a method of the class *Context*, and is used for starting services. In our example, this method is called in method **onClick**() of the *MainActivity* class, and the *Intent* parameter is a *FetchData* object. In other words, *FetchData* will work in the background to fetch data from a URL address and save the data in an external storage.

8.6.3 UI Handled by *MainActivity*

In this project, *MainActivity* handles the UI and let *FetchData* do the downloading and storing. It also creates a *DataReceiver* object and registers it as we have discussed before. It starts the *FetcData* activity in background by calling the method **startService**(*Intent*) and passes data to *FetchData* through the *Intent* parameter. The following lists the complete code of this class:

Program Listing 8-8 *MainActivity.java* of *FetchData* Project

```
package comm.fetchdata;

import android.app.*;
import android.os.*;
import android.view.*;
import android.widget.*;
import android.content.*;

public class MainActivity extends Activity
{
  public TextView textView;
  DataReceiver receiver;
  @Override
  protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  receiver = new DataReceiver ();
  textView = (TextView) findViewById(R.id.status);
}

@Override
protected void onResume() {
  super.onResume();
  registerReceiver(receiver,new IntentFilter(FetchData.NOTIFICATION));
}

@Override
protected void onPause() {
  super.onPause();
  unregisterReceiver(receiver);
}

public void onClick(View view) {
  Intent intent = new Intent(this, FetchData.class);
```

```
   // add info for the service of fetching file
   intent.putExtra(FetchData.FNAME, "downloaded.txt");
   intent.putExtra(FetchData.URL_ADDRESS,
        "http://www.forejune.com/index.html");
   // Start FetchData to run as background
   startService(intent);
   textView.setText("Service started");
 }
-----------------------------------------------------------------------
```

After we have written these three Java programs, *MainActivity.java*, *FetchData.java* and *DataReceiver.java*, we have to modify the manifest file, *AndroidManifest.xml* to grant access permissions to the Internet and the external storage to the application. We also need to declare the *FetchData* service in the file as shown in the Listing below.

Listing 8-9 *AndroidManifest.xml* of *FetchData* Project

```
<?xml version="1.0" encoding="utf-8"?>
 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
  .....
  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission
      android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

  <application
    .....
    <activity
      .....
    </activity>
    <service android:name="comm.fetchdata.FetchData" >
    </service>
  </application>
</manifest>
-----------------------------------------------------------------------
```

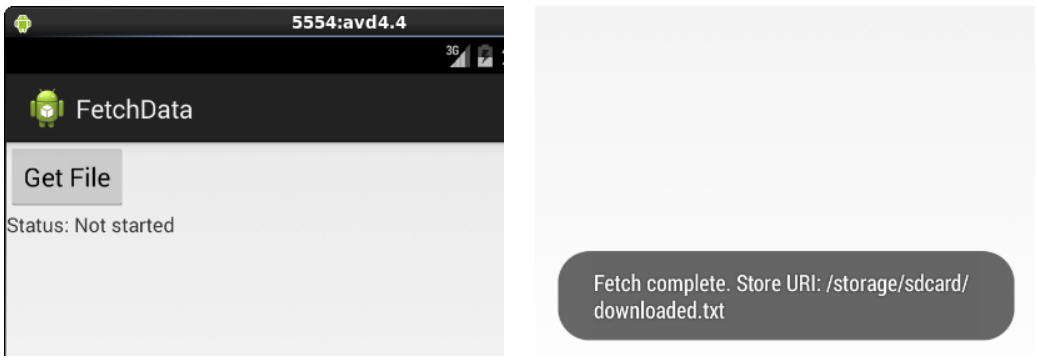
To incorporate the UI features of our project, we also need to modify the layout file, *activity_main.xml* to Listing 8-10. reflect the UI

Listing 8-10 *activity_main.xml* of *FetchData* Project

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical" >

   <Button
      android:id="@+id/button1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="onClick"
      android:text="Get File" />

   <LinearLayout
```

```
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" >

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Status: " />

   <TextView
      android:id="@+id/status"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Not started" />
 </LinearLayout>
</LinearLayout>
-------------------------------------------------------------------------
```

After we have done all these, we can compile and run the application. Figure 8-8 below shows portions of the output display. Figure 8-8 (a) shows the screen when the application has started but the service *FetchData* has not been started. One can start the service in the background by clicking on the *Get File* button. When the *FetchData* service has finished its task, it broadcasts a signal to *DataReceiver*, which displays a *Toast* message as shown in Figure 8-8 (b).



(a) (b)

Figure 8-8 UI of *FetchData* Project

8.7 AIDL (Android Interface Definition Language)

8.7.1 AIDL Interface

As we mentioned in Chapter 3, Android provides AIDL (Android Interface Definition Language) to ease interprocess communication (IPC). It is similar to the traditional interface definition language (IDL) that describes the interface between components in remote procedure calls (RPC) for C/C++ applications. The language allows a user to define a common programming interface for a client to communicate with a server, which runs a service. Normally, a process is not allowed to access the memory of another process and the objects created by one process may not be understood by the other. For two processes to communicate seamlessly, the exchanging objects must be decomposed into primitives and marshalled. Android provides AIDL tools to do the marshalling.

An AIDL interface is defined in a file ending with *.aidl* using the Java programming language syntax. The file should be saved in the source code directories (*src/*) of both the application hosting the service and any other application that binds to the service.

When we build an application that contains the *.aidl* file, the AIDL tools generate a corresponding *IBinder* interface, which is saved in the project's *gen/* directory. The service program has to implement an appropriate *IBinder* interface so that client applications can bind to the service and call the *IBinder* methods to communicate with it.

In general, three steps are involved in building an AIDL service:

1. **Create a file with** *.aidl* **extension** to define the interface to be used by client processes.
2. **Implement the interface** based on the Java program generated from the *.aidl* file by the Android SDK tools.
3. **Expose the interface** to clients by implementing a *Service*, where we override the **onBind**() method.

8.7.2 The .aidl File

We have to create a *.aidl* file inside the project's *src/* directory. If Eclipse IDE is used in the development, it automatically generates an *IBinder* interface file in the *gen* directory when we save the *.aidl* file that has no syntax error. The Eclipse IDE also indicates any syntax error with a red dot.

The syntax of a *.aidl* file is simple. We can declare an interface with one or more methods that can take parameters and return values. Each *.aidl* can only define a single interface. If we need to define more than one interface, we have to create multiple *.aidl* files inside the *src* directory. AIDL supports all primitive types such as **int, long, char**, and **boolean** of the Java programming language. It also suppots built-in Java classes including *String*, *CharSequence*, *List*, and *Map*. The following shows an example of a simple *.aidl* file, named *IRemoteService.aidl*; the method **multiply** is supposed to multiply two numbers and to return the product.

```
// IRemoteService.aidl
package comm.aidlcalc;

// Declare any non-default types here with import statements

/** Example service interface */
interface IRemoteService {
  // You can pass values in, out, or inout.
  // Primitive datatypes (e.g. int, char) can only be passed in.
  float multiply (in float num1, in float num2);
}
```

A corresponding Java program, *IRemoteService.java* is generated inside the *gen* directory, which looks like the following

```
package comm.aidlcalc;
// Declare any non-default types here with import statements
/** Example service interface */
public interface IRemoteService extends android.os.IInterface
{
  /** Local-side IPC implementation stub class. */
  public static abstract class Stub extends android.os.Binder
                      implements comm.aidlcalc.IRemoteService
  {
    private static final java.lang.String DESCRIPTOR =
```

```
                                  "comm.aidlcalc.IRemoteService";
    /** Construct the stub at attach it to the interface. */
    public Stub()
    {
      this.attachInterface(this, DESCRIPTOR);
    }
    .....
  }
  .....
  public float multiply(float num1, float num2)
                       throws android.os.RemoteException;
}
```

8.7.3 Implement AIDL Interface

The AIDL tools generate a Java interface from a *.aidl* file. The interface includes a subclass named *Stub*, which is an abstract implementation of its parent interface and declares all the methods as shown in the above example. To implement the interface, we have to extend the generated *Binder* interface, *Stub* and the methods defined in the *.aidl* file.

Here is an example implementation of the *IRemoteService* of the above example, using an anonymous instance:

```
@Override
public IBinder onBind(Intent intent) {

  return new IRemoteService.Stub() {
     // Implement  multiply()
    public float multiply (float a, float b)
     throws RemoteException {
      return a * b;
    }
  };
}
```

Note that by default, RPC calls are synchronous, meaning the the client waits for the server's result before executing the next instruction. Therefore, if the service takes a relatively long time to finish the task of a request, we should not call the service from the main thread of the client's activity, which might lead to the Android system displaying a dialog of *Application is Not Responding*. To avoid this, we should typically call the service from a separate thread in the client.

8.7.4 Expose AIDL Interface to Clients

After implementing the remote service interface, we need to expose it to the clients, which will bind to it. This is done by extending the *Service* class, and implement its **onBind**() method, which will return an object of the class that implements *Stub*. The following is an example service that exposes the *IRemoteService* interface discussed above to clients:

```
public class RemoteService extends Service
{
  @Override
  public void onCreate() {
    super.onCreate();
  }
```

```
  @Override
  public IBinder onBind(Intent intent) {

    return new IRemoteService.Stub() {
      public float multiply (float a, float b)
       throws RemoteException {
        return a * b;
      }
    };
  }
}
```

When a client activity calls **bindService**() to connect to this service, the client's **onService-Connected**() callback receives the *IBinder* object returned by the service's **onBind**() method. The client also needs to access the interface class. Therefore if the client is not in the same application of the service, the client application must also have a copy of the *.aidl* file inside its *src/* directory, which will be used to generate the same Java program inside its *gen/* directory.

When the client receives the *IBinder* object in the **onServiceConnected**() callback, it must call the interface's *Stub*.**asInterface**() to cast the returned parameter to the same interface type as the service. The following is an example of such a callback:

```
class RemoteServiceConnection implements ServiceConnection
{
  IRemoteService remoteService;

  // Called when the connection with the service is established
  public void onServiceConnected (ComponentName name,
                                  IBinder boundService) {
    remoteService =
      IRemoteService.Stub.asInterface((IBinder) boundService);
      Toast.makeText(MainActivity.this, "Service connected",
           Toast.LENGTH_LONG).show();
  }

  // Called when connection with service disconnects unexpecteldy
  public void onServiceDisconnected(ComponentName name) {
    remoteService = null;
  }
}
```

8.7.5 A Remote Multiplier

The Android developer site presents a couple of detailed and complex examples on remote service communication. Here, to give readers a quick start of writing remote service applications, we present a very simple example, in which the remote service simply accepts two numbers from a client, multiplies them, and returns the product, and the client is responsible for the UI, accepting two numbers from the user, and displaying the result on the screen. A remote service here means a service that runs in a different process from that of the client. Actually, part of the code has already been presented in the examples of the previous two sections.

Again, we use Eclipse IDE to create the project of this example. We call the project *AidlCalc*, the application *AidlCalcServer*, and the package, *comm.aidlcalc*. The followig are the steps of creating and implementing this project.

1. As usual, we click on **File** and subsequent menus to create the project *AidlCalc* with package name *comm.aidlcalc*, along with default files, including *MainActivity.java*.

2. Create the file *IRemoteService.aidl*: click **File** > **New** > **File**. Enter *AidlCalc/src/comm/aidlcalc* for parent folder, and *IRemoteService.aidl* for **File name**. Click **Finish** to create the file. Edit the file to the following:

```
// IRemoteService.aidl
package comm.aidlcalc;
interface IRemoteService {
   float multiply (in float num1, in float num2);}
```

3. Implement the service class *RemoteService.java*: Click **File** > **New** > **Class**, and enter the apropriate names. Edit the file to the following:

```
//RemoteService.java
package comm.aidlcalc;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.RemoteException;
import android.util.Log;

public class RemoteService extends Service
{
  @Override
  public void onCreate() {
    super.onCreate();
  }
  @Override
  public IBinder onBind(Intent intent) {
    return new IRemoteService.Stub() {
      // Implement  multiply()
      public float multiply (float a, float b)
             throws RemoteException {
        return a * b;
      }
    };
  }
  @Override
  public void onDestroy() {
    super.onDestroy();
  }
}
```

This class implements our remote service, which returns an *IBinder* object from the **onBind**() method. The AIDL-defined method **multiply**() is implemented as a method in the inner class. This code exposes the remote service.

4. Modify the file *MainActivity.java* to the following:

```
// MainActivity.java
package comm.aidlcalc;

import android.app.Activity;
import android.content.*;
import android.os.Bundle;
import android.os.IBinder;
import android.os.RemoteException;
import android.text.TextUtils;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.*;

public class MainActivity extends
             Activity implements View.OnClickListener
{
  IRemoteService remoteService;
  RemoteServiceConnection remoteConnection;
  EditText t1;
  EditText t2;
  Button multiply;
  TextView displayResult;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    bindActivityToService();  // starts the service

    // Setup the UI
    t1 = (EditText) findViewById(R.id.t1);
    t2 = (EditText) findViewById(R.id.t2);
    multiply = (Button) findViewById(R.id.multiply);
    displayResult = (TextView) findViewById(R.id.displayResult);
    multiply.setOnClickListener( this);
  }

  public void onClick ( View view ) {
    float num1 = 0, num2 = 0, product = 0;
    // check if the fields are empty
    if (TextUtils.isEmpty(t1.getText().toString())
         || TextUtils.isEmpty(t2.getText().toString())) {
       return;
    }

    // read EditText and fill variables with numbers
    num1 = Float.parseFloat(t1.getText().toString());
    num2 = Float.parseFloat(t2.getText().toString());
```

```
    try {
       product = remoteService.multiply(num1, num2);
    } catch (RemoteException e) {
         e.printStackTrace();
    }
    // form the output line
    displayResult.setText( num1 + " * " + num2 + " = " + product );
  }

  /**
   * This class implements the actual service connection, casting
   * bound stub implementation of the service to AIDL interface.
   */
  class RemoteServiceConnection implements ServiceConnection
  {
    //Called when the connection with the service is established
    public void onServiceConnected(ComponentName name,
                                          IBinder boundService){
      remoteService =
        IRemoteService.Stub.asInterface((IBinder) boundService);
      Toast.makeText(MainActivity.this, "Service connected",
                                      Toast.LENGTH_LONG).show();
    }

    //Called when connection with service disconnects unexpected
    public void onServiceDisconnected(ComponentName name) {
      remoteService = null;
      Toast.makeText(MainActivity.this, "Service disconnected",
                                      Toast.LENGTH_LONG).show();
    }
  }

  /** Binds this activity to the service. */
  private void bindActivityToService() {
    remoteConnection = new RemoteServiceConnection();
    Intent intent = new Intent();
    intent.setClassName("comm.aidlcalc",
                    comm.aidlcalc.RemoteService.class.getName());
    bindService(intent,remoteConnection,Context.BIND_AUTO_CREATE);
  }

  // Unbinds this activity from the service.
  private void releaseService() {
    unbindService(remoteConnection);
    remoteConnection = null;
  }

  // Called when the activity is about to terminate
  @Override
  protected void onDestroy() {
```

```
    releaseService();
  }
```

As we have implemented the **onBind**() method in *RemoteService*, we need to establish a connection between the service and our client (*MainActivity*). This is done by implementing the *ServiceConnection* class in *RemoteServiceConnection*, where **onServiceConnected**() and **onServiceDiconnected**() methods are implemented. These callbacks will get the stub implementation of the remote service upon connection or disconnection.

Besides binding the activity to the remote service, the method **bindActivityToService**() also starts the service.

The UI of the application is very simple. There are two *EditText* fields and a *Button*, which represents multiplication. The *MainActivity* class has implemented the *OnClickListener* class. When the *Button* is clicked, the **multiply**() method is invoked on the service as if it were a local call.

5. Modify the file *res/layout/activity_main.xml* for our UI as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
        "http://schemas.android.com/apk/res/android"
 android:orientation="vertical" android:layout_width="fill_parent"
 android:layout_height="fill_parent">
 <TextView android:layout_width="fill_parent"
   android:layout_height="wrap_content" android:text="AidlCalc"
   android:textSize="22sp" />
 <LinearLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:id="@+id/linearLayout1"
   android:layout_marginLeft="12pt"
   android:layout_marginRight="12pt"
   android:layout_marginTop="4pt">

   <EditText
     android:layout_weight="1"
     android:layout_height="wrap_content"
     android:layout_marginRight="6pt"
     android:id="@+id/t1"
     android:layout_width="match_parent"
     android:text="1989"
     android:inputType="numberDecimal">
   </EditText>
   <Button
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="*"
     android:textSize="10pt"
     android:id="@+id/multiply">
   </Button>
   <EditText
```

```
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_marginLeft="6pt"
        android:id="@+id/t2"
        android:layout_width="match_parent"
        android:text="64"
        android:inputType="numberDecimal">
      </EditText>
   </LinearLayout>
   <TextView
      android:layout_height="wrap_content"
      android:layout_width="match_parent"
      android:layout_marginLeft="6pt"
      android:layout_marginRight="6pt"
      android:textSize="12pt"
      android:layout_marginTop="4pt"
      android:id="@+id/displayResult"
      android:gravity="center_horizontal">
   </TextView>
</LinearLayout>
```

6. Add the service to the file *res/AndroidManifest.xml* as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="comm.aidlcalc"
    .....
    <application
       .....
       <activity
         .....
       </activity>
       <service android:name=".RemoteService" />
    </application>
</manifest>
```

7. Run the application. When we run the application, the multiplier service is started in the background and the client activity presents us a UI like the one shown in Figure 8-9 below. When the activity has successfully connected to the service, it displays a *Toast* message, saying *Service connected* as shown in the lower part of the figure. We can enter two numbers and click the multiplication button, which calls the remote service to do the multiplication. When the client receives the result, it displays the multiplication operation and the result as shown in the upper half of the figure.

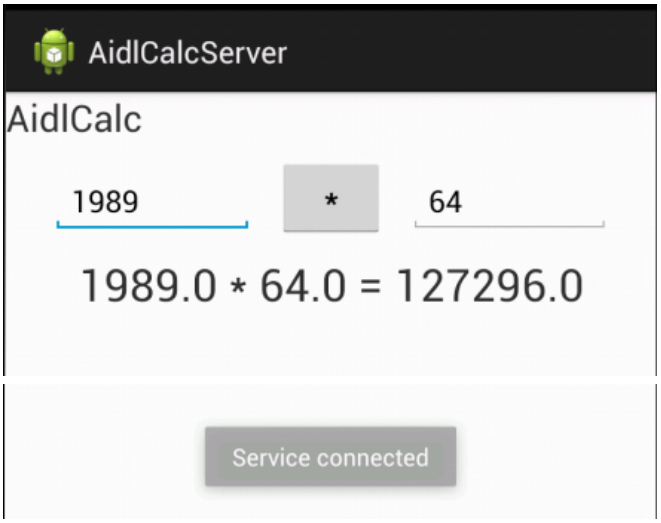


Figure 8-9 UI of *AidlCalc* Project