Lec #16: Networking

Tamer Nadeem
Dept. of Computer Science
Objective

- **Bluetooth**
  - Managing Bluetooth Properties
  - Device Discovery
  - Bluetooth Communication

- **WiFi**

- **Presentation**
  - Calling All Cars: Cell Phone Networks and the Future of Traffic
    - Presenter: Jeremiah Dunn
  - Micro-Blog: Sharing and Querying Content Through Mobile Phones and Social Participation
    - Presenter: Zhonglu Wang
Bluetooth
Overview

• Bluetooth is a specification for the use of low power wireless communications over short distance.

• Although Bluetooth standard utilizes the same 2.4 GHz range of Wi-Fi

• Compared to Wi-Fi, Bluetooth networking is slower, a bit more limited in range, and supports many fewer devices
Bluetooth States

STANDBY

Page
master response
Page Scan
Inquiry

Inquiry Scan
Inquiry response
slave response

CONNECTION

Active
Hold
Pack
Sniff
Android Bluetooth

• Classes support Bluetooth devices and connections:
  ➤ **BluetoothAdapter** represents the local Bluetooth device on which your application is running.
  ➤ **BluetoothDevice** Each remote device with which you wish to communicate.
  ➤ **BluetoothSocket** let you make a connection request to the remote device, and then initiate communications.
  ➤ **BluetoothServerSocket** use it on your local Bluetooth Adapter to listen for incoming connection requests from Bluetooth Sockets on remote devices.
Bluetooth Device Adapter

• To access the default Bluetooth adapter on the host device call `getDefaulAdapter`.

```java
BluetoothAdapter bluetooth = BluetoothAdapter.getDefaultAdapter();
```

• To read any of the local Bluetooth Adapter properties, initiate discovery, or find bonded devices: include the `BLUETOOTH` manifest permission.

• To modify any of the local device properties: include the `BLUETOOTH_ADMIN` manifest permission.

```xml
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
```
Managing Bluetooth Properties

• Reading/Changing properties needs Bluetooth adapter to be on
• Access the Bluetooth Adapter’s *friendly name* (an arbitrary string that users can set) and hardware address

```java
BluetoothAdapter bluetooth = BluetoothAdapter.getDefaultAdapter();

String toastText;
if (bluetooth.isEnabled()) {
    String address = bluetooth.getAddress();
    String name = bluetooth.getName();
    toastText = name + " : " + address;
} else 
    toastText = "Bluetooth is not enabled";

Toast.makeText(this, toastText, Toast.LENGTH_LONG).show();
```

• Change the friendly name of the Bluetooth Adapter using:

```java
bluetooth.setName("Blackfang");
```
Enabling Bluetooth Adaptor

- By default the Bluetooth adapter will be turned off.

- Enable the Bluetooth Adapter thru system sub-Activity using the `ACTION_REQUEST_ENABLE`
  ➤ Use the result code parameter returned in the `onActivityResult` handler to determine the success of this operation.

```java
String enableBT = BluetoothAdapter.ACTION_REQUEST_ENABLE;
startActivityForResult(new Intent(enableBT), 0);
```

- You can turn the Bluetooth Adapter on and off directly, using the `enable` and `disable` methods.
Enabling Bluetooth Adaptor

- Enabling/disabling are time-consuming, asynchronous operations.
  ➤ Register a Broadcast Receiver that listens for `ACTION_STATE_CHANGED`.
  ➤ The broadcast Intent will include two extras, `EXTRA_STATE` and `EXTRA_PREVIOUS_STATE`, the current and previous states.

```java
BluetoothAdapter bluetooth = BluetoothAdapter.getDefaultAdapter();

BroadcastReceiver bluetoothState = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String prevStateExtra = BluetoothAdapter.EXTRA_PREVIOUS_STATE;
        String stateExtra = BluetoothAdapter.EXTRA_STATE;
        int state = intent.getIntExtra(stateExtra, -1);
        int previousState = intent.getIntExtra(prevStateExtra, -1);

        String tt = "";
        switch (state) {
            case BluetoothAdapter.STATE_TURNING_ON : {
                tt = "Bluetooth turning on"; break;
            }
        }
    }
};
```
Enabling Bluetooth Adaptor

case (BluetoothAdapter.STATE_ON) : {
    tt = "Bluetooth on";
    unregisterReceiver(this);
    break;
}

case (BluetoothAdapter.STATE_TURNING_OFF) : {
    tt = "Bluetooth turning off"; break;
}

case (BluetoothAdapter.STATE_OFF) : {
    tt = "Bluetooth off"; break;
}

default: break;

Toast.makeText(this, tt, Toast.LENGTH_LONG).show();

if (!bluetooth.isEnabled()) {
    String actionStateChanged = BluetoothAdapter.ACTION_STATE_CHANGED;
    String actionRequestEnable = BluetoothAdapter.ACTION_REQUEST_ENABLE;
    registerReceiver(bluetoothState,
        new IntentFilter(actionStateChanged));
    startActivityForResult(new Intent(actionRequestEnable), 0);
}
Device Discovery

• The process of two devices finding each other in order to connect is called *discovery*.
  ➤ Before you can establish a Bluetooth Socket for communications, the local Bluetooth Adapter must bond with the remote device.
  ➤ Before two devices can bond and connect, they first need to discover each other.

• In order for remote Android Devices to find your local Bluetooth Adapter during a discovery scan, you need to ensure that it is discoverable.
Managing Device Discoverability

• The Adapter’s discoverability is indicated by its scan mode.
• Call `getScanMode` on the BluetoothAdapter object. It returns:
  ➤ `SCAN_MODE_CONNECTABLE_DISCOVERABLE` Inquiry Scan and Page Scan are both enabled → device is discoverable.
  ➤ `SCAN_MODE_CONNECTABLE` Page Scan is enabled but Inquiry Scan is not → devices that have previously connected and bonded to the local device can find it during discovery, but new devices can’t.
  ➤ `SCAN_MODE_NONE` Discoverability is turned off → No remote devices can find the local adapter during discovery.

• By default, discoverability is disabled. To turn on discovery use `start activity` with `ACTION_REQUEST_DISCOVERABLE`:

```java
String aDiscoverable = BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE;
startActivityForResult(new Intent(aDiscoverable), DISCOVERY_REQUEST);
```
Managing Device Discoverability

• To handle user response, override `onActivityResult` handler.
  ➤ The returned `resultCode` indicates the duration of discoverability.

```java
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == DISCOVERY_REQUEST) {
        boolean isDiscoverable = resultCode > 0;
        int discoverableDuration = resultCode;
    }
}
```

• Monitor `ACTION_SCAN_MODE_CHANGED` broadcast action.
  ➤ Broadcast Intent has current and previous scan modes as extras.

```java
registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String prevScanMode = BluetoothAdapter.EXTRA_PREVIOUS_SCAN_MODE;
        String scanMode = BluetoothAdapter.EXTRA_SCAN_MODE;
        int scanMode = intent.getIntExtra(scanMode, -1);
        int prevMode = intent.getIntExtra(prevScanMode, -1);
    }
}, new IntentFilter(BluetoothAdapter.ACTION_SCAN_MODE_CHANGED));
```
Discovering Remote Devices

- The discovery process can take some time to complete (up to 12 seconds).
- To check if the Adapter is already performing a discovery scan, use the `isDiscovering` method.
- To initiate the discovery process call `startDiscovery` on the Bluetooth Adapter. To cancel a discovery in progress call `cancelDiscovery`.
- The discovery process is asynchronous. Android uses broadcast Intents to notify you of the start (`ACTION_DISCOVERY_STARTED`) and end (`ACTION_DISCOVERY_FINISHED`) of discovery as well as remote devices discovered (`ACTION_FOUND`) during the scan.
Monitoring discovery

```java
BroadcastReceiver discoveryMonitor = new BroadcastReceiver() {
    String dStarted = BluetoothAdapter.ACTION_DISCOVERY_STARTED;
    String dFinished = BluetoothAdapter.ACTION_DISCOVERY_FINISHED;

    @Override
    public void onReceive(Context context, Intent intent) {
        if (dStarted.equals(intent.getAction())) {
            // Discovery has started.
            Toast.makeText(getApplicationContext(),
                            "Discovery Started . . . ", Toast.LENGTH_SHORT).show();
        }
        else if (dFinished.equals(intent.getAction())) {
            // Discovery has completed.
            Toast.makeText(getApplicationContext(),
                            "Discovery Completed . . . ", Toast.LENGTH_SHORT).show();
        }
    }
};

registerReceiver(discoveryMonitor, new IntentFilter(dStarted));
registerReceiver(discoveryMonitor, new IntentFilter(dFinished));
```
Discovering remote Bluetooth Devices

• Each broadcast Intent includes the name of the remote device in an extra `BluetoothDevice.EXTRA_NAME`, and representation of the remote device under `BluetoothDevice.EXTRA_DEVICE` extra.

```java
BroadcastReceiver discoveryResult = new BroadcastReceiver() {  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        String remoteDeviceName = intent.getStringExtra(BluetoothDevice.EXTRA_NAME);  
        BluetoothDevice remoteDevice;  
        
        remoteDevice = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);  
        Toast.makeText(getApplicationContext(), "Discovered: " + remoteDeviceName,  
                        Toast.LENGTH_SHORT).show();  
        // TODO Do something with the remote Bluetooth Device.  
    }  
};

registerReceiver(discoveryResult, new IntentFilter(BluetoothDevice.ACTION_FOUND));

if (!bluetooth.isDiscovering())
    bluetooth.startDiscovery();
```
Bluetooth Communications

• The Bluetooth communications APIs are wrappers around RFCOMM, the Bluetooth radio frequency communications protocol.
  ➤ RFCOMM supports RS232 serial communication over the Logical Link Control and Adaptation Protocol (L2CAP) layer.

• You must have a **client** and a **server**. Used classes are:
  ➤ **BluetoothServerSocket**: Used to establish a listening socket at the **server** for initiating a link between devices.
  ➤ **BluetoothSocket**: Used in creating a new **client** socket to connect to a listening Bluetooth Server Socket, and returned by the Server Socket once a connection is established.

• Once the connection is made, Bluetooth Sockets are used on both the **server** and **client** sides to transfer data streams.

• Connection is done in **separate thread**
RFCOMM (Radio Frequency Communication)

- The Bluetooth protocol RFCOMM is a simple set of transport protocols.
- RFCOMM is sometimes called Serial Port Emulation.
- The Bluetooth Serial Port Profile is based on this protocol.

- In the protocol stack, RFCOMM is bound to L2CAP
- RFCOMM provides a simple reliable data stream to the user, similar to TCP. It is used directly by many telephony related profiles as a carrier for AT commands.
Bluetooth Server

• Use **BluetoothServerSocket** to **listen** for incoming connection requests get by calling **listenUsingRfcommWithServiceRecord**
  ➤ Passing: String “name” to identify your server, and
  ➤ UUID (universally unique identifier) to be used by clients to connect

• To start listening, call **accept** on this Server Socket
  ➤ Optionally passing in a timeout duration.

• The Server Socket will **block** until a remote Bluetooth Socket client with a matching UUID attempts to connect.

• If a remote device is not yet paired, the user will be prompted to accept a pairing request before the accept call returns.

• If an incoming connection request is successful, accept will return a **Bluetooth Socket** connected to the client device.
private class AcceptThread extends Thread {
  private final BluetoothServerSocket mmServerSocket;

  public AcceptThread() {
    BluetoothServerSocket tmp = null;
    try {
      tmp = mBluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME, MY_UUID);
    } catch (IOException e) { ... }
    mmServerSocket = tmp;
  }

  public void run() {
    BluetoothSocket socket = null;
    while (true) {
      try {
        socket = mmServerSocket.accept();
      } catch (IOException e) {
        break;
      }
      if (socket != null) {
        // You must implement this method
doSomethingWithTheConnection(socket);
        mmServerSocket.close();
        break;
      }
    }
  }

  public void cancel() {
    try {
      mmServerSocket.close();
    } catch (IOException e) { ... }
  }
}
Bluetooth Client

• Use **BluetoothSocket** to create a new **connection** to the server by calling **createRfcommSocketToServiceRecord**
  ➤ Passing: UUID of the Bluetooth Server Socket accepting requests

• If you attempt to connect to a Bluetooth Device that has not yet been paired (bonded) with current host, you will be prompted to accept the pairing before the connect call completes

• The user must accept the pairing request on both the host and remote devices for the connection to be established.

• The returned Bluetooth Socket can then be used to initiate the connection with a call to connect.
Bluetooth Client

```java
private class ConnectThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final BluetoothDevice mmDevice;

    public ConnectThread(BluetoothDevice device) {
        BluetoothSocket tmp = null;
        mmDevice = device;

        try {
            tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
        } catch (IOException e) { ... }
        mmSocket = tmp;
    }

    public void run() {
        mBluetoothAdapter.cancelDiscovery();

        try {
            mmSocket.connect();
        } catch (IOException connectException) {
            try {
                mmSocket.close();
            } catch (IOException closeException) { }
            return;
        }

        manageConnectedSocket(mmSocket);
    }

    public void cancel() {
        try {
            mmSocket.close();
        } catch (IOException e) { }
    }
}
```
Questions?