Programming with Android: Network Operations

Federico Montori
Dipartimento di Scienze dell'Informazione
Università di Bologna

Outline

- **Network operations**: *WebView*
- **Network operations**: *WebView* and *WebSettings*
- **Network operations**: *HTTP* Client
- **Network operations**: *Download Manager*
- **Network operations**: *OKHttp*
- **Network operations**: *Volley*
- **Network operations**: *TCP/UDP* Sockets

Android: Network Operations

- In order to perform network operations, specific **permissions** must be set on the **AndroidManifest.xml**.

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

- Failure in setting the permissions will cause the system to throw a **run-time** exception …

Before we start

- Network operations are expensive
  - In terms of battery
  - In terms of time
  - In terms of costs

- You should always care about making network operations (and your app in general) optimized

Lazy first

- Make your app **Lazy first**, by
  - Decreasing redundant operations (cache)
    - If your app needs frequent updates, cache static objects to not download them every time
  - Timing operations (deferring them until better situation)
    - Wait to perform network operations until device is charging, connected to a Wifi, etc.
  - Grouping operations together
    - Instead of performing similar operations at slightly different times, try to perform them together at once

User Preferences

- User preferences for network operations matter a lot
  - User may want to perform network intensive operations only when connected to WiFi
  - Synchronization may be performed at night or at home
  - Updates frequency can be customized
- Network-related user preferences activities should declare a **MANAGE_NETWORK_USAGE** intent filter
- Monitor the **CONNECTIVITY_ACTION** Broadcast Intent

Android: Network Operations

- Before the application attempts to connect to the network, it should check to see whether the active network connection is available using **getActiveNetworkInfo()** and **isConnected()** …

```
ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    if (networkInfo != null && networkInfo.isConnected()) {
        // fetch data
    } else {
        // display error
    }
```

Android: Network Operations

- It is also possible to differentiate between different connections

```
ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
Network[] allNetworks = connMgr.getAllNetworks();

for (Network network : allNetworks) {
   NetworkInfo nInfo = connMgr.getNetworkInfo(network);
   // Do stuff ...
}
```

Android: Network Operations

And you can ask for even more

- getType(): to check the network type
  - ConnectivityManager.TYPE_WIFI, ConnectivityManager.TYPE_MOBILE
- getDetailedState(): to obtain fine grained information
  - IDLE, SCANNING, … [Deprecated from API 29]
- isAvailable(): to check whether the network is available
  - Not necessarily connected
- isRoaming(): if the network is operated abroad

Android: WebView Usage

WebView A **View** that displays web pages, including simple browsing methods (history, zoom in/out/ search, etc).

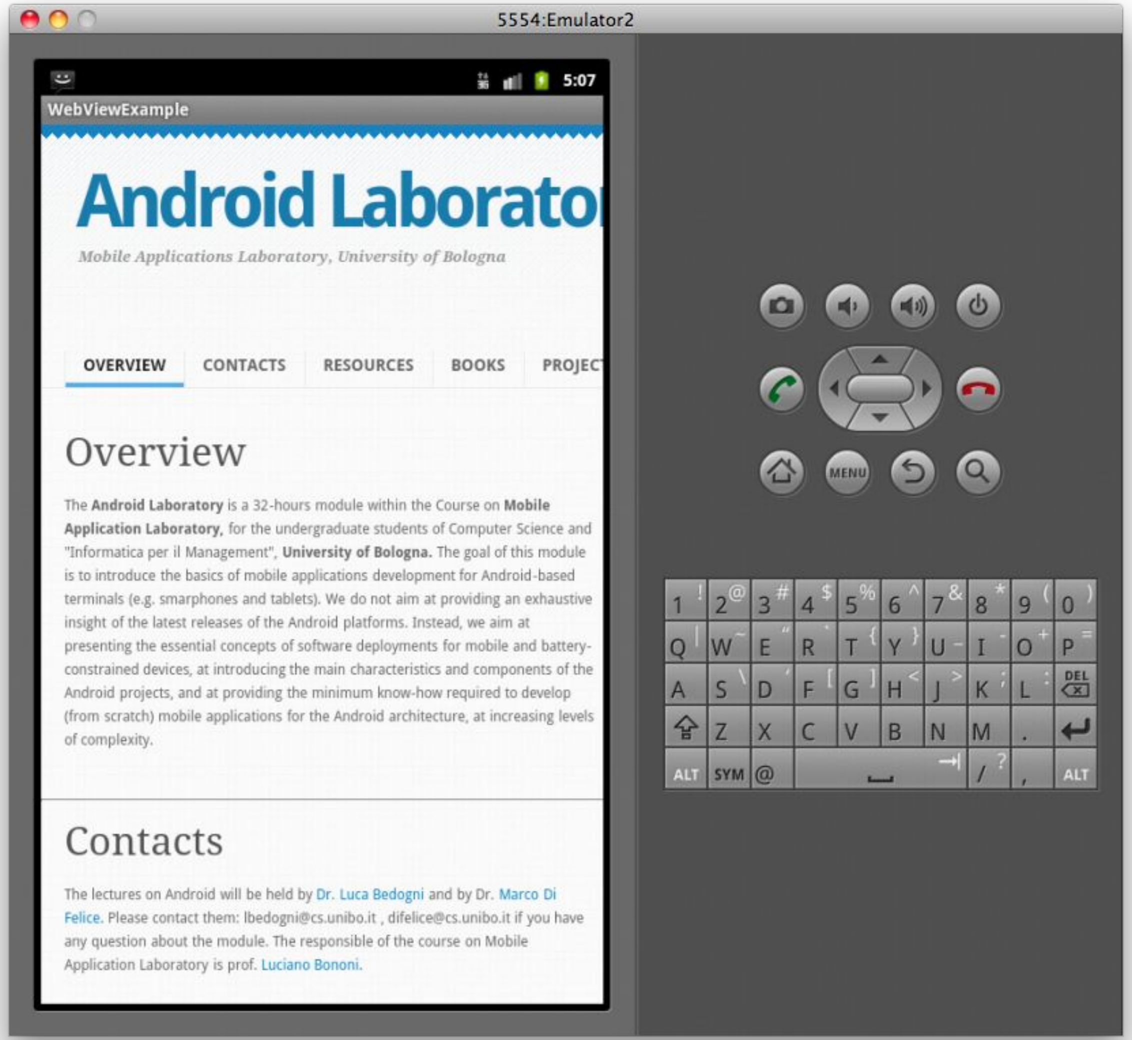
Implemented by the WebView class

```
public WebView(Context context)
```

Main methods:

- public void **loadUrl**(String url) load the HTML page at url
- public void **loadData**(String data, String mimeType, string encoding) load the HTML page contained in data

Android: WebView Usage



Overview

The **Android Laboratory** is a 32-hours module within the Course on **Mobile Application Laboratory**, for the undergraduate students of Computer Science and "Informatica per il Management", **University of Bologna.** The goal of this module is to introduce the basics of mobile applications development for Android-based terminals (e.g. smarphones and tablets). We do not aim at providing an exhaustive insight of the latest releases of the Android platforms. Instead, we aim at presenting the essential concepts of software deployments for mobile and battery-constrained devices, at introducing the main characteristics and components of the Android projects, and at providing the minimum know-how required to develop (from scratch) mobile applications for the Android architecture, at increasing levels of complexity.

Contacts

The lectures on Android will be held by Dr. Luca Bedogni and by Dr. Marco Di Felice. Please contact them: lbedogni@cs.unibo.it , difelice@cs.unibo.it if you have any question about the module. The responsible of the course on Mobile Application Laboratory is prof. Luciano Bononi.

All it does is pretty much showing the content of a Web page. It's <u>NOT</u> a browser.

Useful when you quickly need content that is always up to date.

In some case better than getting data, parsing and displaying in a layout.

Android: WebView Usage

By default, the WebView UI does not include any navigation button …However, **callbacks** methods are defined:

Example:

- public void **goBack**()
- public void **goForward**()
- public void **reload**()
- public void **clearHistory**()

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent
event) {

     // Is there a page in the history?
     if ((keyCode == KeyEvent.KEYCODE_BACK) &&
    myWebView.canGoBack()) {
         myWebView.goBack();
         return true;
     }
    // Otherwise use the normal behavior
     return super.onKeyDown(keyCode, event);
}
```

Android: WebView Usage

It is possible to modify the visualization options of a WebView through the **WebSettings** class.

```
public WebSettings getSettings()
```

Some options:

- void **setJavaScriptEnabled**(boolean)
- void **setBuildInZoomControls**(boolean)
- void **setDefaultFontSize**(int)

Also, bear in mind that cleartext data is not allowed by default. If you really need it then add to your manifest (**application** tag):

```
android:usesCleartextTraffic="true"
```

Android: WebView Usage

Override the behavior for which links in the WebView open in the WebView (they in fact don't throw an intent) with a **WebViewClient**

```
myWebView.setWebViewClient(MyWebViewClient);
```

```
private class MyWebViewClient extends WebViewClient {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
         if ("www.mysite.com".equals(Uri.parse(url).getHost())) {
           // This is my website, so do not override; let my WebView load the page
             return false;
         }
         // The link is not for a page on my site, so throw the intent for browser
         Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
         startActivity(intent);
         return true;
     }
}
```

Android: Download Manager

DownloadManager System service that handles long-run HTTP downloads.

- The client can specify the file to be downloaded through an **URI** (path).
- Download is conducted in **background** (with retries)
- Broadcast Intent action is sent to notify when the download completes.

```
DownloadManager dm = (DownloadManager)
   getSystemService(DOWNLOAD_SERVICE);
```

Android: Download Manager

- The Request class is used to specify a download request to the Download Manager.

```
Request request = new DownloadManager.Request(Uri.parse(address));
```

Main methods of the **DownloadManager**

- long **enqueue**(DownloadManager.Request)
- Cursor **query**(DownloadManager.Query)
- ParcelFileDescriptor **openDownloadedFile**(long)

Android: Download Manager

- long **enqueue**(DownloadManager.Request)

```
long id = dm.enqueue(new DownloadManager.Request(uri)
.setAllowedNetworkTypes(DownloadManager.Request.NETWORK_WIFI |
      DownloadManager.Request.NETWORK_MOBILE)
.setDestinationInExternalPublicDir(Environment.DIRECTORY_DOWNLOADS,
      "output.txt"));
```

- Cursor **query**(DownloadManager.Query)

```
Cursor c = dm.query(
   new DownloadManager.Query().setFilterById(id));
// can use DownloadManager.COLUMN_BYTES_DOWNLOADED_SO_FAR etc...
```

- ParcelFileDescriptor **openDownloadedFile**(long) or better:

```
registerReceiver(myReceiver,
     new IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE));
```

Android: HTTP Classes

HTTP (<u>HyperText Tranfer Protocol)</u>: Network protocol for exchange/transfer data (hypertext)

Request/Response Communication Model

MAIN COMMANDS

- HEAD
- GET
- POST
- PUT
- DELETE
- TRACE
- CONNECT

Android: HTTP Classes

HTTP (HyperText Tranfer Protocol): Network protocol for exchange/transfer data (hypertext)

Two implementations of HTTP Clients for Android historically:

- **`HTTPClient`** → Complete extendable HTTP Client suitable for web browser (not supported starting from 6.0)
- **`HTTPUrlConnection`** → Light-weight implementation, suitable for client-server networking applications (recommended by Google, starting from 2.3)

In both cases, HTTP connections must be managed on a separate thread, e.g. using **Thread Pool** (not the UI thread!).

Android: HTTP Classes

HTTPUrlConnection □ HTTP component to send and receive streaming data over the web.

1. Obtain a new **HttpURLConnection** by calling the **URL.openConnection()**

```
URL url = new URL("http://www.android.com/");
HttpURLConnection urlConnection = (HttpURLConnection)
                        url.openConnection();
```

2. Prepare the request, set the options:
   - session cookies
   - credentials
   - preferred content type (e.g. **setRequestProperty("Content-Type", "text/plain");**)

Android: HTTP Classes

HTTPUrlConnection □ HTTP component to send and receive streaming data over the web.

3. For **POST** commands, invoke **`setDoOutput(true).`** Transmit data by writing to the stream returned by **`getOutputStream().`**

```
try {
     urlConnection.setDoOutput(true);
     urlConnection.setRequestMethod("POST");
     urlConnection.setChunkedStreamingMode(0);
     OutputStream out = new
      BufferedOutputStream(urlConnection.getOutputStream());
     out.write(“YourPostInput”.getBytes());   }
```

For best performance use **setFixedLengthStreamingMode(int)** `instead of setChunkedStreamingMode when the size is known.`

Android: HTTP Classes

HTTPUrlConnection □ HTTP component to send and receive streaming data over the web.

4. Read the response (data+header). The response body may be read from the stream returned by **`getInputStream()`**.

```
InputStream in = new
BufferedInputStream(urlConnection.getInputStream());
// Do what you want with the InputStream
```

5. Close the session when ending reading the stream through **`disconnect`**().

```
urlConnection.disconnect();
```

Android: HTTP Classes

HTTPUrlConnection □ HTTP component to send and receive streaming data over the web.

- use **getErrorStream()** in case of errors
- use the **HttpsURLConnection** in case of HTTPS URLs
  - Can override the default HostnameVerifier
  - Can override the **SSLSocketFactory**
  - Can define a custom **X509TrustManager** to verify certificate chains
- use **HttpResponseCache** if you need to cache replies in order not to waste resources

OKHttp

- HTTP Client for Java applications
- Supports multiplexing of different connections on the same socket
- Lower latency
- Can compress larger downloads transparently
- Repeated requests may be served through cache

OKHttp builder

- Requests are built through the builder paradigm

```
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder()
      .url("https://www.unibo.it/sitoweb/federico.montori2")
      .build();
```

```
Request request = new Request.Builder()
      .header("Authorization", ”your authorization here")
      .url("https://www.unibo.it/sitoweb/federico.montori2")
      .build();
```

OKHttp: network calls

- Synchronous call

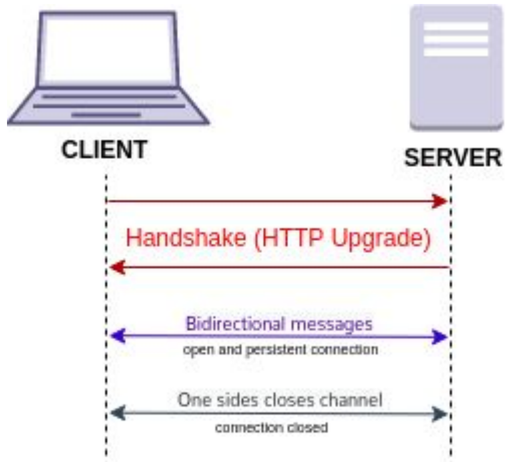
```
Response response = client.newCall(request).execute();
```

- Asynchronous call

```
client.newCall(request).enqueue(new Callback() {
@Override
public void onFailure(Call call, IOException e) {}

@Override
public void onResponse(Call call, final Response response) {
if (response.isSuccessful()) {
    // Here we have the response
}
```

OKHttp: WebSockets

WebSocket is a **full-duplex** communication protocol based on HTTP (ports 80 and 443) and TCP.



It is a more efficient alternative to HTTP polling…

Several libraries available, even OkHttp
https://square.github.io/okhttp/4.x/okhttp/okhttp3/-web-socket/
https://github.com/square/okhttp/blob/d854e6d5ad93da4da9b5d5818ee752477e501b18/samples/guide/src/main/java/okhttp3/recipes/WebSocketEcho.java

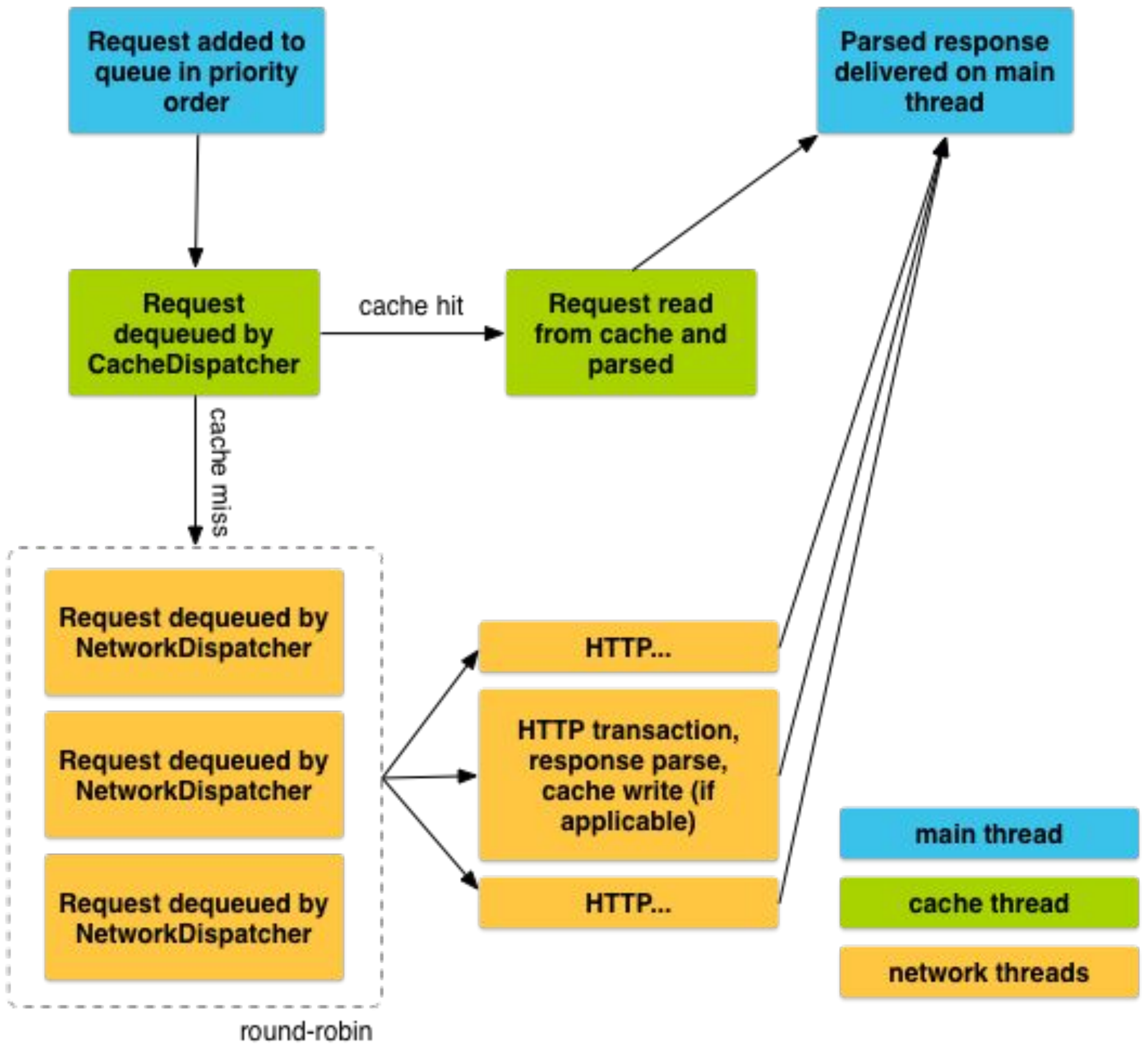
```
public final class WebSocketEcho implements WebSocketListener {
    private void run() { … }
    …
}
```

Volley

- Volley is an HTTP library
- Supports scheduling of network requests
- Can have concurrent connections and handles priorities
- Caching mechanism
- Can cancel requests
- Heavily customizable
- Request ordering
- Not suited for long download operations (keeps in memory all streaming content)

Volley: framework

- Make a request and **add** it.
- Then it moves through the pipeline
- Cache triages it
- If not found it's transferred to a network thread
- Response is sent back


Request added to queue in priority order
Parsed response delivered on main thread
Request dequeued by CacheDispatcher
cache hit
Request read from cache and parsed
cache miss
Request dequeued by NetworkDispatcher
Request dequeued by NetworkDispatcher
Request dequeued by NetworkDispatcher
HTTP...
HTTP transaction, response parse, cache write (if applicable)
HTTP...
main thread
cache thread
network threads
round-robin

Adding Volley to the project

- Add to the build.gradle

```
implementation 'com.android.volley:volley:1.1.1'
```

- Make a request (more on https://developer.android.com/training/volley)

```
RequestQueue queue = Volley.newRequestQueue(this);
StringRequest stringRequest = new StringRequest(Request.Method.GET, baseUrl,
        new Response.Listener<String>() {
                @Override
                public void onResponse(String response) {   // do something   }
        }, new Response.ErrorListener() {
                @Override
                public void onErrorResponse(VolleyError error) {  // do something   }
});
queue.add(stringRequest);
```

Adding Headers

- Custom headers can be added by overriding the getHeaders method like so:

```
{
    @Override
    public Map<String, String> getHeaders() {
        Map<String, String> params = new HashMap<String, String>();
        params.put("x-vacationtoken", "secret_token");
        params.put("content-type", "application/json");
        return params;
}
```

Android: TCP/IP Communication

TCP/UDP Communication Android applications can use java.net.Socket facilities.

 Use socket-based programming like in Java …

Class **DatagramSocket** UDP Socket

Classes **Socket/ServerSocket** TCP socket

Read/Write on Sockets through **InputStream/OutputStream**

Socket example (server)

- Somewhere outside my app...

```
socket=new ServerSocket(10000);

while (true) {
		clientSocket = socket.accept();
		System.out.println("Connected to:"
			+ clientSocket.getInetAddress().toString());
		DataOutputStream outStream =
			new DataOutputStream(clientSocket.getOutputStream());
		double val = rand.nextDouble();
		outStream.writeDouble(val);
		outStream.close();

		clientSocket.close();
	}
```

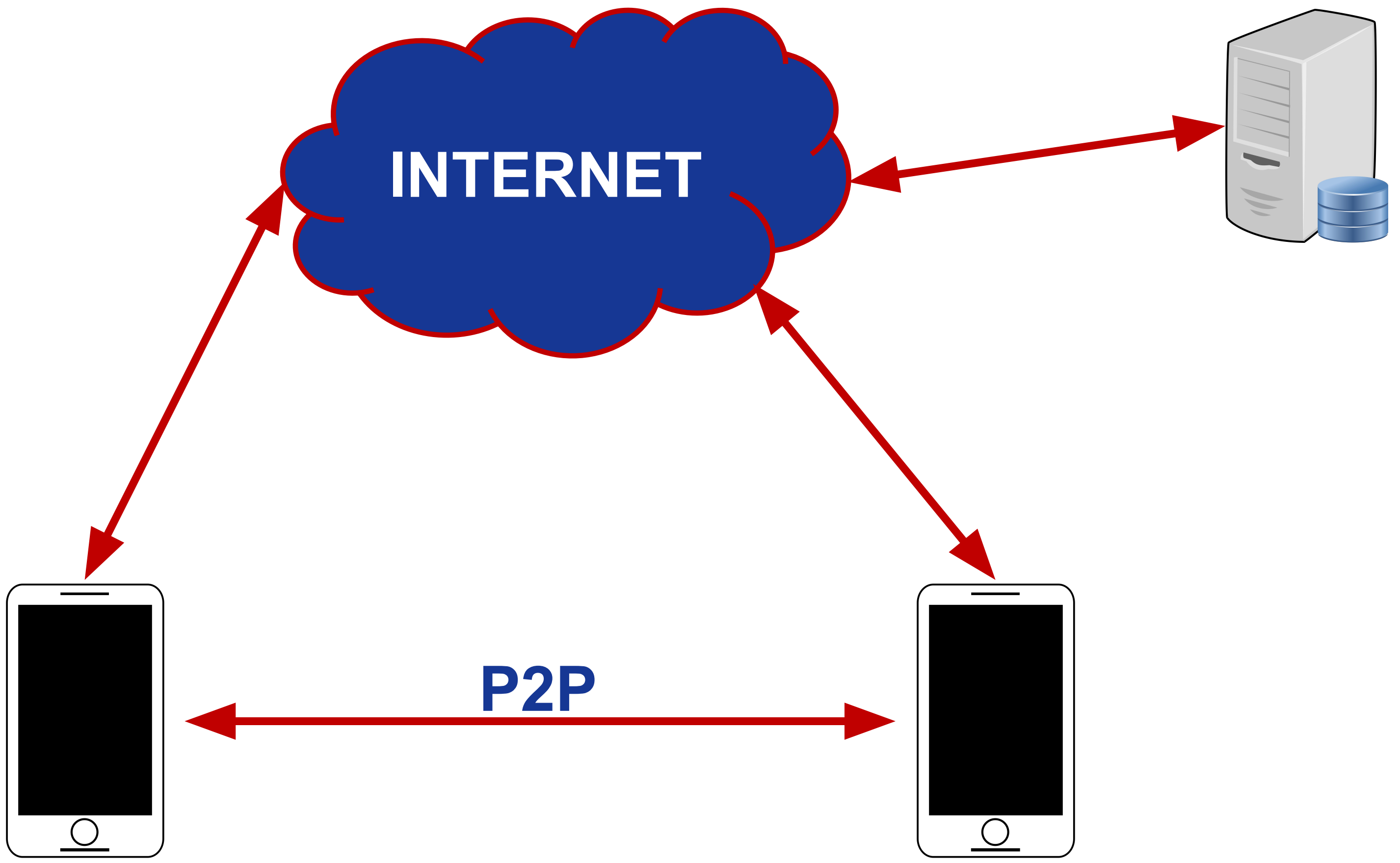
Socket example (Android client)

```
Socket socket = new Socket(serverAddress, 10000);
Message messageToSend = myHandler.obtainMessage(); // Handler in the main thread
messageToSend.what = STATUS_UPDATE;
messageToSend.obj = "Connection Established";
myHandler.sendMessage(messageToSend);

DataInputStream inputStream = new DataInputStream(socket.getInputStream());
double val = inputStream.readDouble();  // The actual data

messageToSend = myHandler.obtainMessage();
messageToSend.what = DATA_UPDATE;
String msg = "Value received "+ val;
messageToSend.obj = msg;
myHandler.sendMessage(messageToSend);

inputStream.close();
socket.close();
```

Other resources

- **Cronet** https://developer.android.com/guide/topics/connectivity/cronet
  - Chromium network stack made available as a set of libraries
  - Support for HTTP, HTTP2 and QUIC
  - Support for resource caching, asynchronous requests and compression
- **gRPC** https://developer.android.com/guide/topics/connectivity/grpc
  - Remote Procedure Calls, so no handling of HTTP protocol syntax
  - Protocol buffer encoding instead of text-based JSON os similar
  - Full duplex streaming

Peer-to-Peer (P2P)



P2P: Why?

- Confidentiality
  - Information is transferred directly between devices
- Speed
  - Data transfer takes the shortest path
- Network relief
  - If 2 devices communicate infrastructure-less, then the infrastructure has more available resources
- Resilience
  - More robust than centralized network (if enough clients)

Wi-Fi Direct

- Standardized by the Wi-Fi Alliance
- Available in popular devices such as smartphones, printers, TVs
- Uses WPA2
- Differences with Bluetooth?
  - Energy efficiency, range, data rate

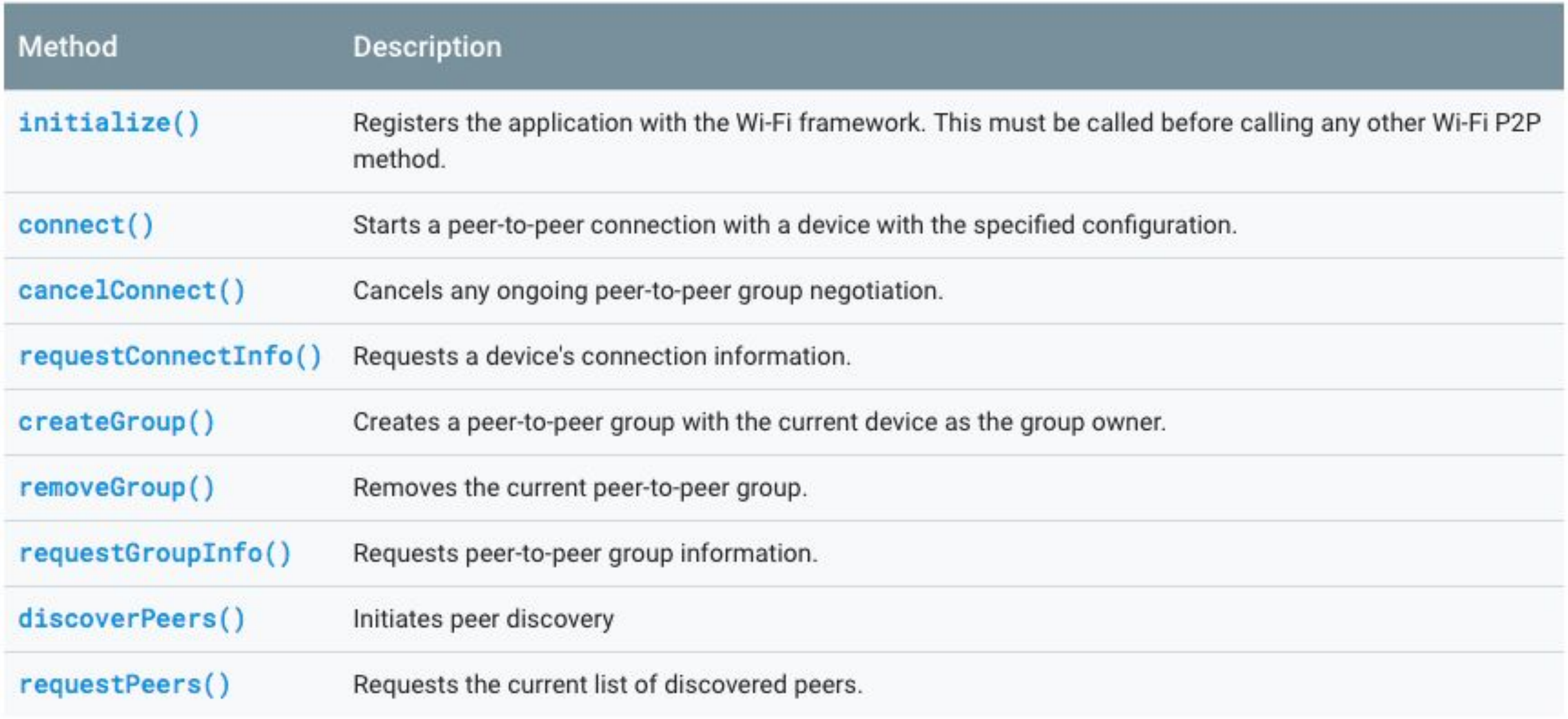
Wi-Fi Direct: building blocks

- Obtain the **WifiP2pManager**
- Discover clients around you
- Connect

- Define **listeners** to be notified about specific events

- **Broadcast Intents** for global events

WifiP2pManager methods

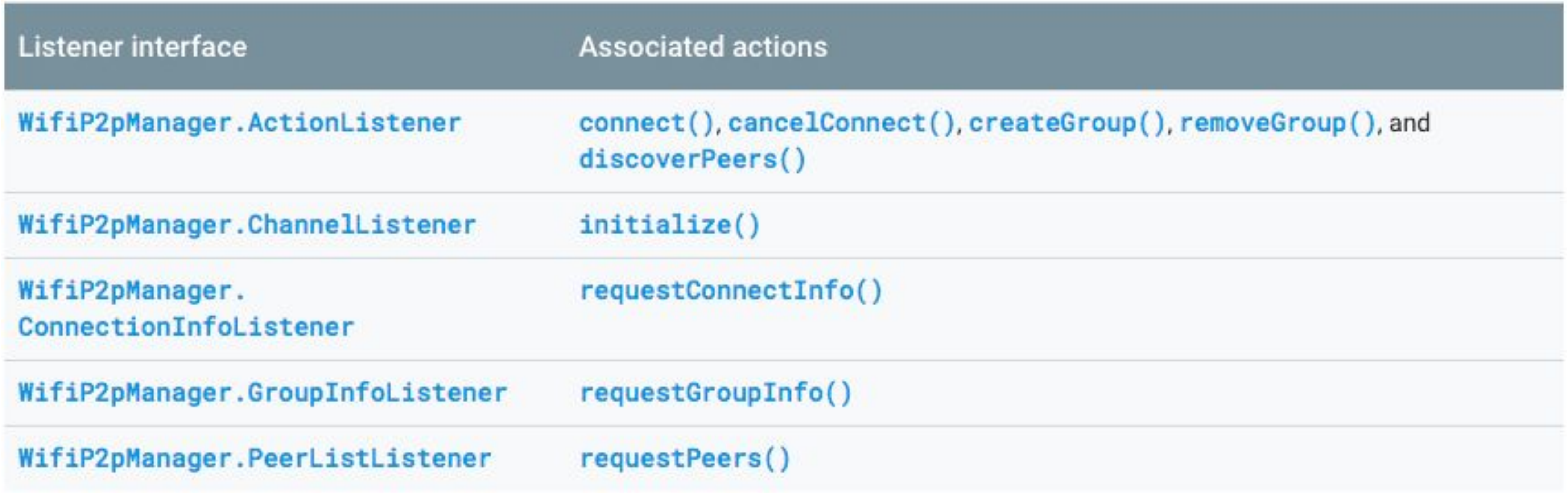


Method	Description
`initialize()`	Registers the application with the Wi-Fi framework. This must be called before calling any other Wi-Fi P2P method.
`connect()`	Starts a peer-to-peer connection with a device with the specified configuration.
`cancelConnect()`	Cancels any ongoing peer-to-peer group negotiation.
`requestConnectInfo()`	Requests a device's connection information.
`createGroup()`	Creates a peer-to-peer group with the current device as the group owner.
`removeGroup()`	Removes the current peer-to-peer group.
`requestGroupInfo()`	Requests peer-to-peer group information.
`discoverPeers()`	Initiates peer discovery
`requestPeers()`	Requests the current list of discovered peers.

WifiP2pManager methods

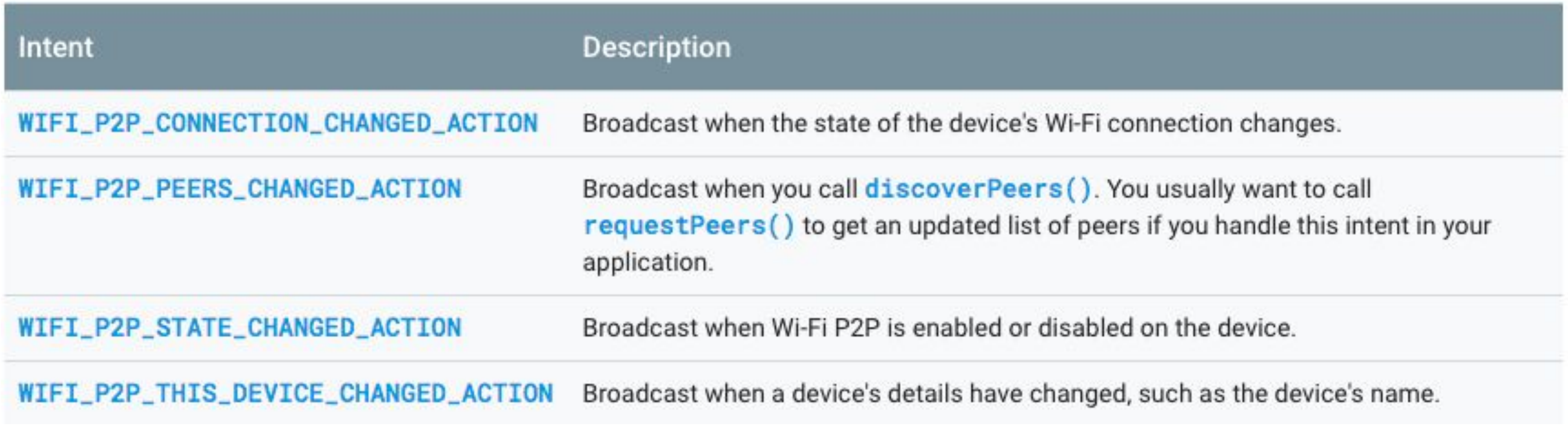
	Method	Description
1	`initialize()`	Registers the application with the Wi-Fi framework. This must be called before calling any other Wi-Fi P2P method.
3	`connect()`	Starts a peer-to-peer connection with a device with the specified configuration.
	`cancelConnect()`	Cancels any ongoing peer-to-peer group negotiation.
	`requestConnectInfo()`	Requests a device's connection information.
	`createGroup()`	Creates a peer-to-peer group with the current device as the group owner.
	`removeGroup()`	Removes the current peer-to-peer group.
	`requestGroupInfo()`	Requests peer-to-peer group information.
2	`discoverPeers()`	Initiates peer discovery
	`requestPeers()`	Requests the current list of discovered peers.

WifiP2pManager listeners



Listener interface	Associated actions
WifiP2pManager.ActionListener	connect(), cancelConnect(), createGroup(), removeGroup(), and discoverPeers()
WifiP2pManager.ChannelListener	initialize()
WifiP2pManager. ConnectionInfoListener	requestConnectInfo()
WifiP2pManager.GroupInfoListener	requestGroupInfo()
WifiP2pManager.PeerListListener	requestPeers()

WifiP2pManager Broadcast Receivers



Intent	Description
WIFI_P2P_CONNECTION_CHANGED_ACTION	Broadcast when the state of the device's Wi-Fi connection changes.
WIFI_P2P_PEERS_CHANGED_ACTION	Broadcast when you call discoverPeers(). You usually want to call requestPeers() to get an updated list of peers if you handle this intent in your application.
WIFI_P2P_STATE_CHANGED_ACTION	Broadcast when Wi-Fi P2P is enabled or disabled on the device.
WIFI_P2P_THIS_DEVICE_CHANGED_ACTION	Broadcast when a device's details have changed, such as the device's name.

Other Connection technologies

- Bluetooth Low Energy
  - https://developer.android.com/guide/topics/connectivity/bluetooth
- NFC
  - https://developer.android.com/guide/topics/connectivity/nfc
- Telephony
  - https://developer.android.com/guide/topics/connectivity/telecom
- WiFi
  - https://developer.android.com/guide/topics/connectivity/wifi-scan