Implementing a Download Background Service

Android Services

Stefan Tramm

Patrick Bönzli

Android Experience Day, FHNW

2008-12-03

Agenda

Part I: Business Part

§ the application

§ some screenshots

Part II: Technical Part

§ general introduction

§ our approach

Part I: Business Part

§ the mission
§ customers and usage scenario
§ technical background
§ screenshots

Prototype “Jukebox”

Mission: Jukebox is a mobile application that collects selected information of various types from authenticated feeds – pull and push. Jukebox interacts with other data and applications on the device and allows to perform custom actions based on the information content type.

Prototype "Jukebox"

- Information content types:
  - Invitations to phone conferences
    Actions: set reminder/alarm, one-click dial-in, put to calendar
  - Audio/Video podcasts

- Attachments: viewing accompanying documents (e.g. PDF)
- add personal notes
- instant feedback during event/show

- uses open data format standards

Usage scenario: enlarge your CRM-capabilities

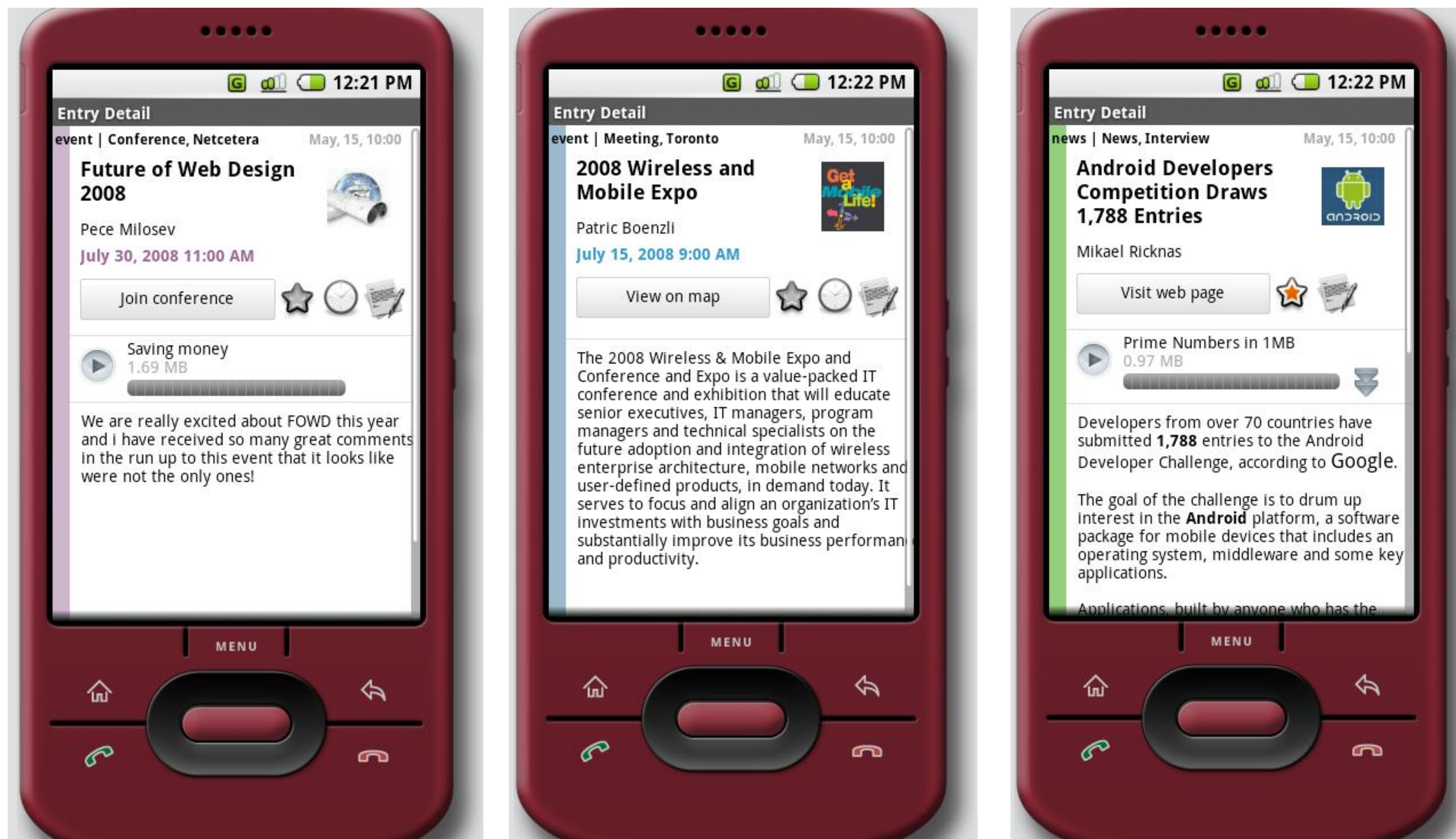
- push *news* to *subscribers* onto *mobile phones*
- provide well known end users with the right information at the right time:
  - invite to telephone conferences (or meetings)
  - allow easy joining at the right time
  - provide attachments (PDF, Word, Excel, Video)
  - link to a specific website
  - provide a podcast afterwards
  - visually indicate updates and changes
  - feedback: reading confirmation, voting

Technical background info

- Android application
- based on standard ATOM file format (**RSS-reader on steroids**)
- asynchronously updating data in background
- uses internal SQLite DB and aggressive caching
  - works without network connection (e.g. airplane)

Jukebox: Listview

Jukebox: Detailview



Part II: Technical Part

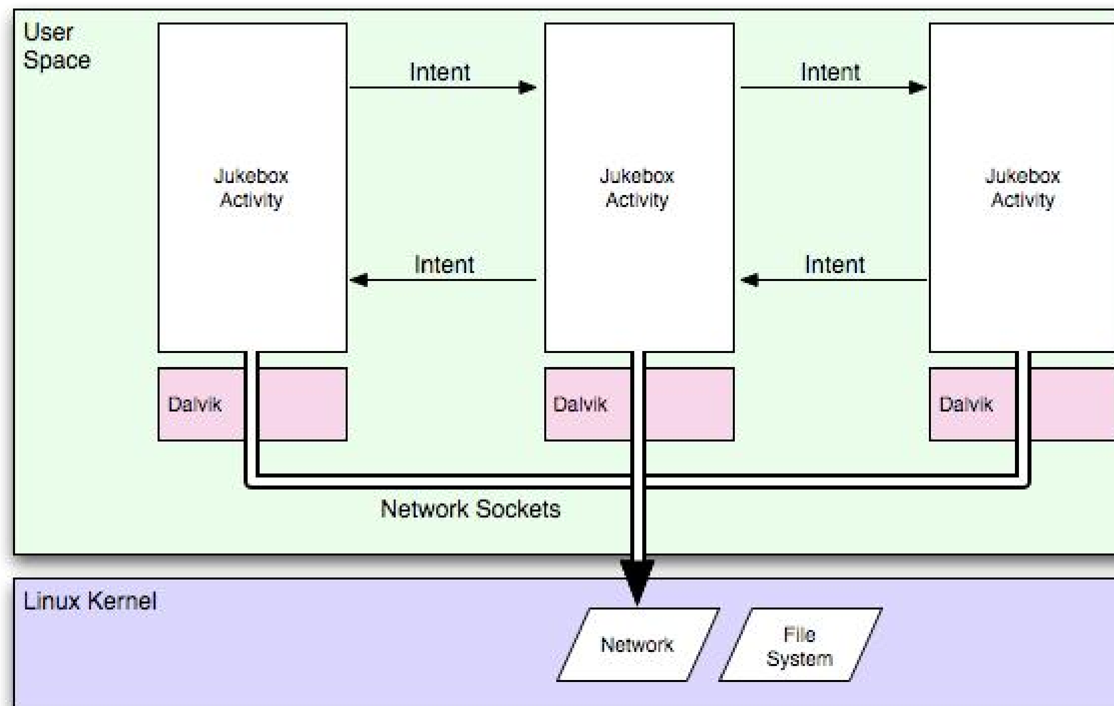
- Android Services
- download manager

Goals

- Understand reasons for background services
- Know the Android Service lifecycle
- See service IPC possibilities by theory and examples
- Understand our download manager approach

Starting Point

- multiple Activities, triggered by Intents
- each Activity needs to download many media entities
- network bandwidth scarce resource ð bottleneck



Managing Network Resources

More reasons to manage network resources carefully:

- optimize **application performance**
- longer **battery lifetime** by minimizing network activity
- conserve **data volume limits**

Boils down to two problems:

- **asynchronous** working
- classic resource **scheduling problem**

Our approach:
Uncoupling network from Activity using a **queue**.

Asynchronous Working Problem

Instruments for solving this problem with a queue:

Parallel processing to remove waiting times

- using threads and services

Indicating finished work

- using events or notifications

Resource Access Scheduling Problem

Instruments for solving this problem with a queue:

Control access to the resource:

- resource allocations only over a central entity
- by limiting number of parallel connections

Prioritize, to determine the order of access:

- using a priority queue

Optimize resource access:

- active caching (memory and disk)

Android Application Building Blocks

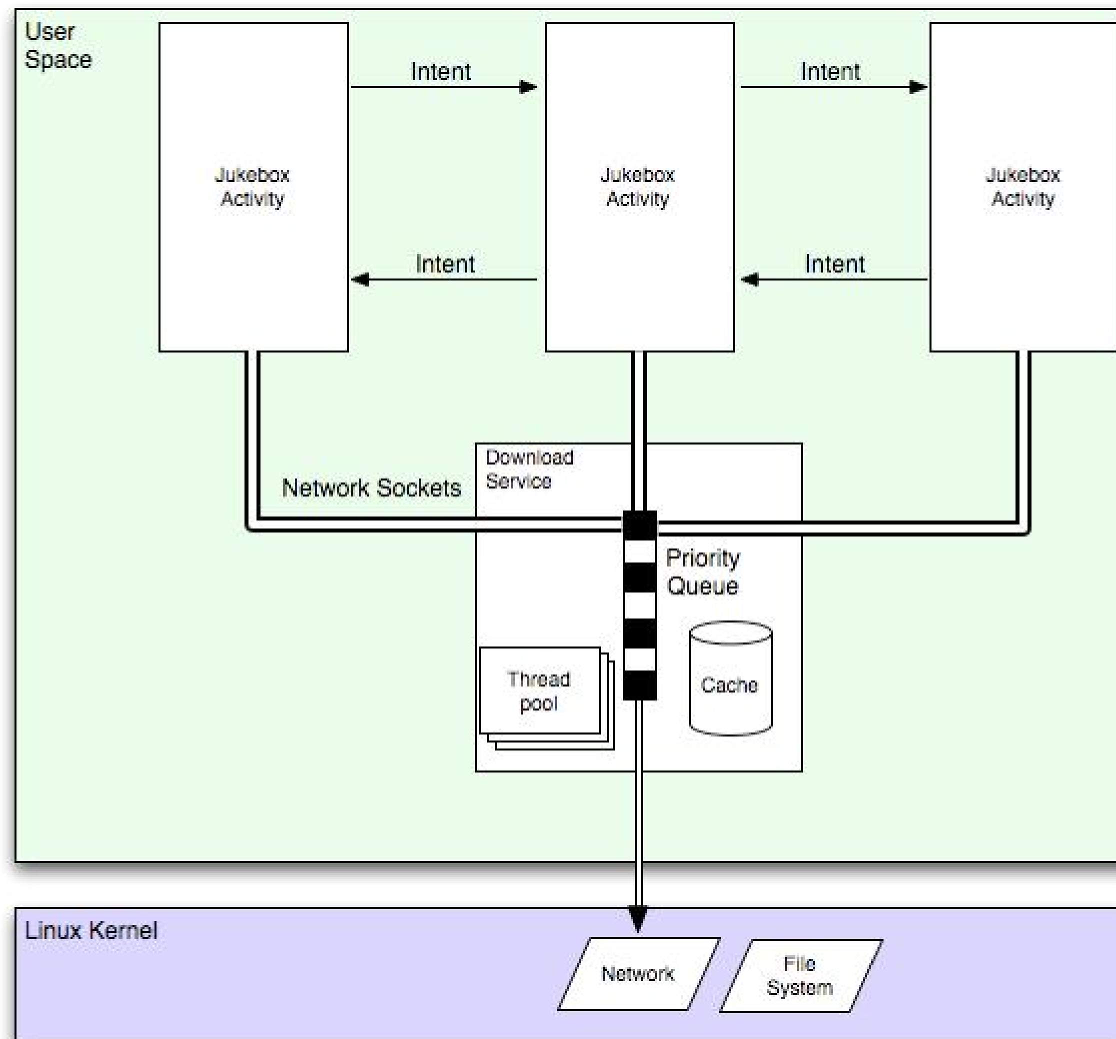
- AndroidManifest.xml
- Activities
- Views
- Layouts
- Intents & IntentReceivers
- Services
- Notifications
- ContentProviders

Android Toolbox

Android offers:

- synchronous downloading: `OpenHTTPConnection`
- parallel processing using threads: `java.lang.Thread`
- thread pools: `java.util.concurrent.ThreadPoolExecutor`
- communication with threads: `android.os.Handler`
- http request queues: `android.net.http.RequestQueue`
- background services: `android.Service`

Architecture Overview



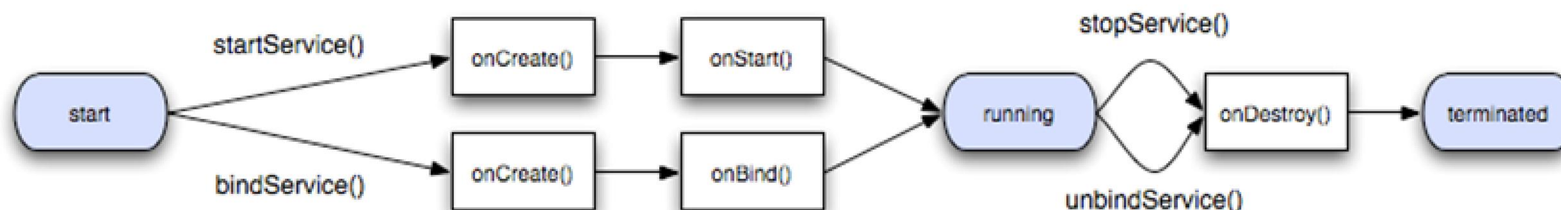
Android Service

Service is an Android application component running in background:

- class extends `android.Service`
- not directly interacting with the user
- started by a hosting process
- runs in the main thread of their hosting process
- uses a different address space than the hosting process
- service can be made publicly available

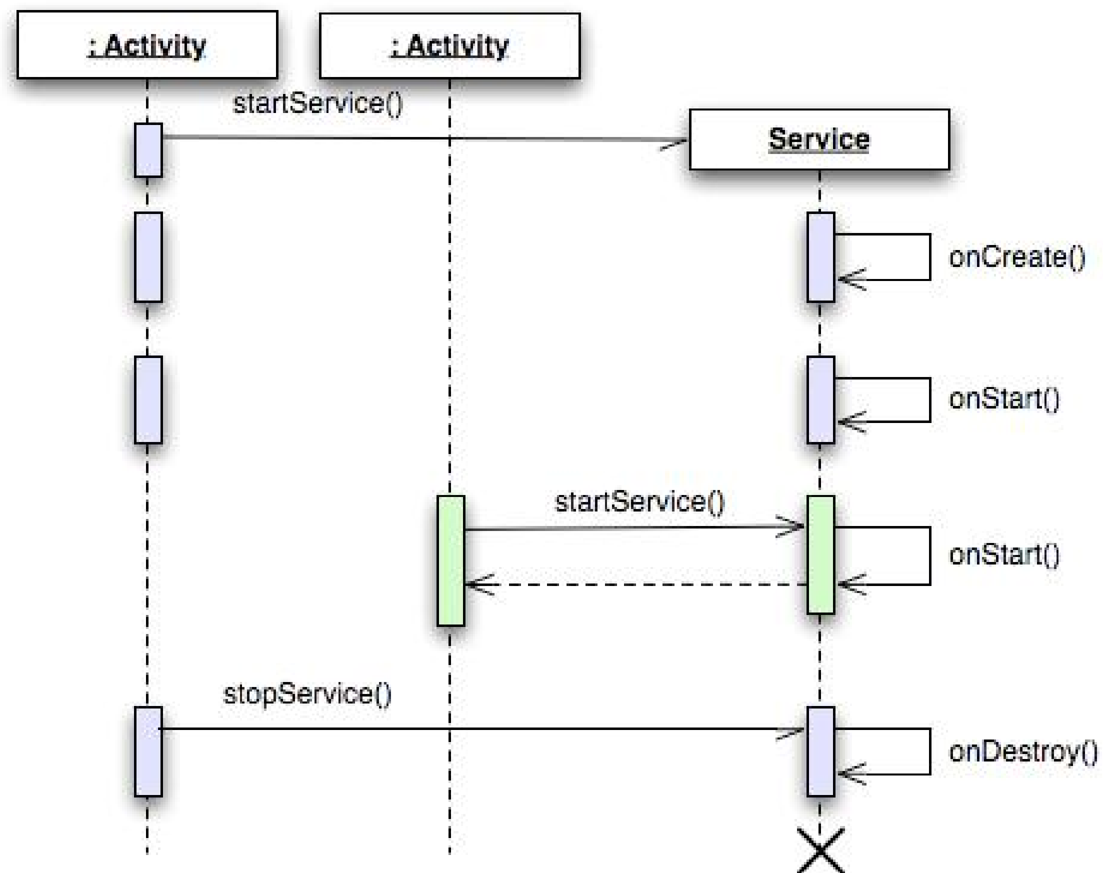
Services Lifecycle I

- services are started by a „hosting“ process
- two possibilities to start a service:
  - `startService()`: starts a service
  - `bindService()`: starts and binds service to host life cycle



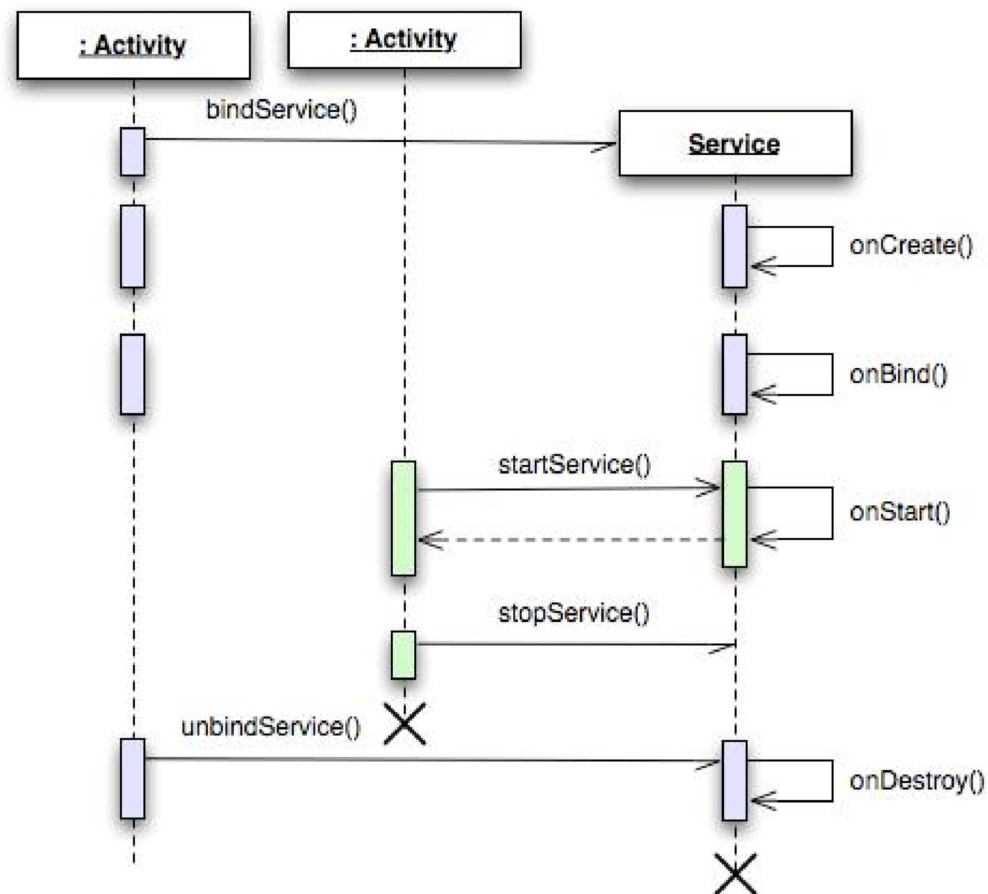
Services Lifecycle II

Starting a service using `startService()`



Services Lifecycle III

Starting a service using `bindService()`



Services IPC

How to share data among multiple concurrent processes?

Persistent:

- files (e.g. flash drive)
- databases (SQLite), Bonus: offers transactions
- Android properties

Non-persistent:

- shared memory
- network sockets
- IDL (interface description language) the Android way: AIDL

Android Interface Description Language - AIDL

§ IDL used to describe interfaces in language-neutral way
§ AIDL is the Android way of doing IPC
§ command line tool available

Important AIDL properties:
§ calls are synchronous
§ objects are reference counted across processes
§ no inter process exception forwarding

Coding Example: The Service

Service definition

```
public class MyService extends Service {

  @Override
  protected void onCreate() {
    super.onCreate();
    Log.i(getClass().getSimpleName(), "Service started.");
  }

  @Override
  protected void onDestroy() {
    super.onDestroy();
    Log.i(getClass().getSimpleName(), "Service stopped.");
  }
}
```

Service start

```
public class SomeActivity extends Activity {
...
startService(new Intent("com.example.android.apis.app.MyService"));
...
}
```

Coding Example: AIDL I

Interface declaration

```
interface IRemoteDownloadService {
  void startDownload(in Uri uri, String type, IRemoteDownloadServiceCallback cb);
}
```

Interface implementation

```
public class RemoteDownloadService extends Service {

private final IRemoteDownloadService.Stub mBinder = new IRemoteDownloadService.Stub() {

  public void startDownload(Uri uri, String type, IRemoteDownloadServiceCallback cb) {
    ResourceHandler handler = new ResourceHandler(uri, type, cb);
    completionService.handleResource(handler);
  }
};

}
```

Coding Example: AIDL II

Communicating with Service

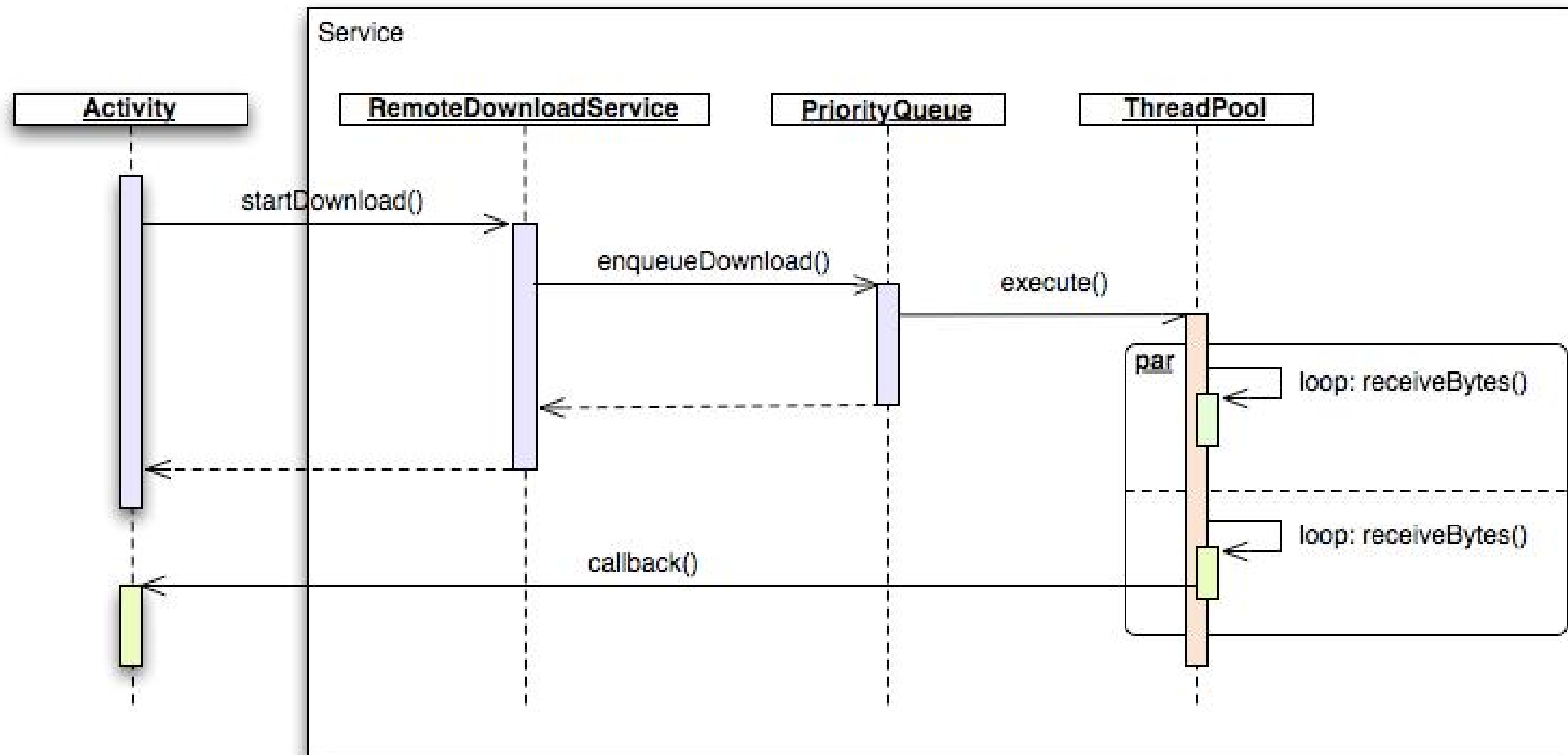
```
public class SomeClass {

  private ServiceConnection connection = new ServiceConnection() {
    public void onServiceConnected(ComponentName name, IBinder service) {
      downloader = IRemoteDownloadService.Stub.asInterface(service);
    }

    public void onServiceDisconnected(ComponentName name) {
    }
  };

...
bindService(new Intent("ch.netcetera.REMOTE_DOWNLOAD_SERVICE"), connection,
BIND_AUTO_CREATE);
...
downloader.startDownload(uri, type, downloadCallback);
}
```

Downloading Manager



Download Manager Error Handling

We need to handle special states separately:

- telephone going to **sleep**
  - save current downloading state persistently
  - resume download on application startup
- **low battery**
  - possibility to switch to offline mode
- **low memory**
  - resize in memory cache
- **network connection down**
  - working in offline mode
  - resume downloads if network comes up again

Q & A

Any questions?

Conclusions

- Android allows background services
- background services must follow a well defined lifecycle
- the implementation of a background data loader is feasible
- experience on real hardware and under day-to-day conditions is needed