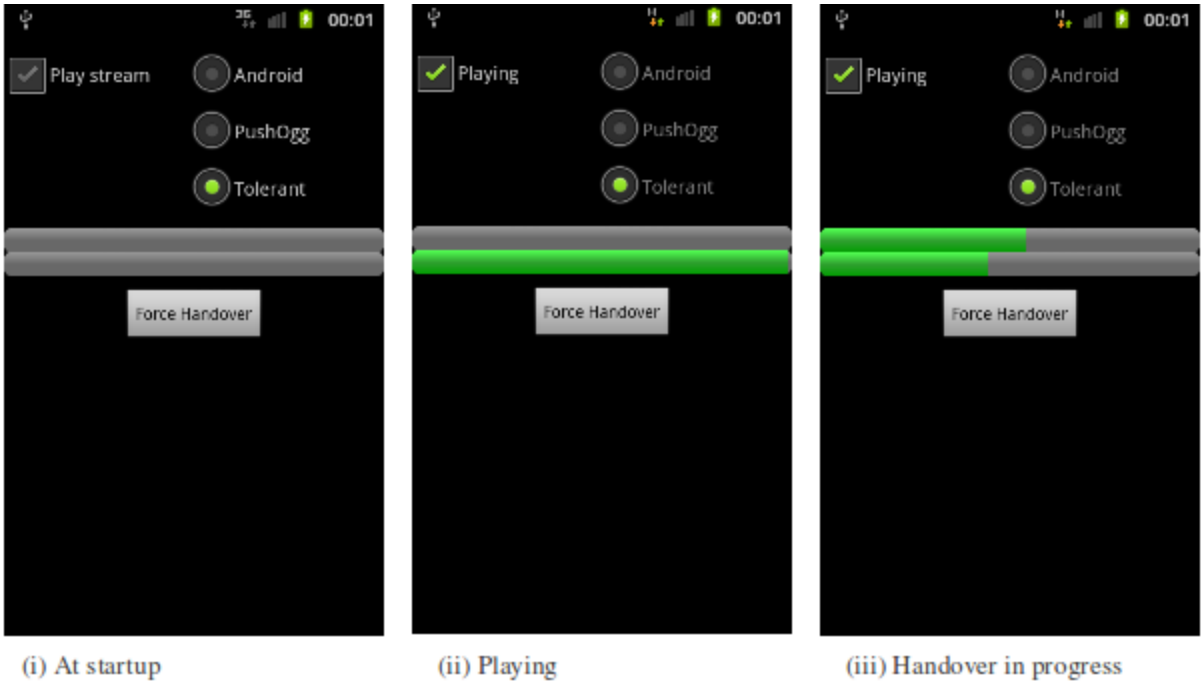
Practical 1: Disconnection-Tolerant Media Player

Application Description

The purpose of the Disconnection-Tolerant Media Player is to stream live audio from a given shoutcast server. The server is located remotely and accessed by http protocol. For this implementation the incoming data is assumed to be an Ogg stream carrying Vorbis audio encoded data, since this is an exercise on how we are handling data, the server URL is hard-coded against a reliable source that repeats a piece of classical music.

The user-interface, shown in figure 1 enables selection between one of three streaming types with various levels of sophistication. The type *Tolerant* is designed to handle continuous streaming in the face of network handovers, two other methods that are not disconnection tolerant are provided for comparison.



(i) At startup (ii) Playing (iii) Handover in progress

Figure 1: Series of screenshots

Android

This streaming type uses the built-in Android MediaPlayer that supports a set of high-level methods that make it convenient to play audio directly from the server. We provide a set of asynchronous listeners that are called on various state-changes (prepared, error and completion). Each of the listeners will start or stop the player as needed and will reflect changes to the user-interface. The most important listener is onPrepared(), after the URL to the server is supplied together with the method prepareAsync(), onPrepared will start the player as soon as a connection is established and enough data has been buffered.

PushOgg

PushOgg is a lower-level approach to the problem. Essentially it achieves the same as the *Android* streamer (both will work well on a stable connection), the difference is that data is handled at the byte level that is eventually written as pulse-code modulated (PCM) audio directly to the hardware. This is done through the Android AudioTrack object which accepts byte-arrays of PCM encoded data.

There are two main areas of concern related to handling the incoming Ogg stream. The first is to query the server as necessary to fetch all the incoming packets one after the other. The second is to extract and convert the data to the PCM encoding

suitable for the Android hardware. Fetching and parsing the data is handled by a collection of objects in the *pushoggdecoder* library. The payload of the Ogg pages are Vorbis encoded, these are each decoded by the Java Vorbis decoder library, JOgg.

The packages are all fetched by our implementation of the OggContainer class. A Thread is required since the advance() method of the container will run continuously until one of the exceptions are thrown (either an error or an end-of-stream). The advance() method will in turn keep calling a nextPage() method that will parse and extract the payload of each of the incoming pages, also called *logical streams*. The data is passed to a user supplied OggPacketReceiver that implements the method

```
packetData(long granulePosition, OggStream stream,
           int length, boolean isStart, boolean isEnd)
```

The OggStream is at this stage conditioned to be read from byte-position zero to length. Our user-supplied PushVorbisPacketReceiver will for each page read, decode and write the conditioned data to the AudioTrack object. If the packet-data is the first one in the current stream the method will additionally setup the audio-track according to the vorbis encoding and characteristics of the hardware. The granulePosition given in the implemented method is a unique number used to track a collection of pages. The value is ignored in this streaming-method because we are assuming a persistent connection which means the packets will be in order and can be processed directly. The granulePosition is important in the next streaming-method to sort out incoming data from two simultaneous connections.

Tolerant

The last of the three streaming-methods is a disconnection-tolerant version of the *PushOgg* method. The main difference is that the OggPacketReceiver (implemented by TolerantPacketReceiver) intermediately writes the PCM data to a buffer instead of directly to the audio-track. The buffer is controlled by the SplicingAudioBuffer class which is instantiated once and can be accessed by different threads as necessary. SplicingAudioBuffer will in a similar way to PushVorbisPacketReceiver in *PushOgg* create an Android AudioTrack object that it writes to, it will create the audio-track only once across potentially more than one streaming thread.

A complication of several threads establishing connections and fetching their own data from the server is that audio data will overlap between them. This is intentionally done by the server to enable seamless playback between connection handovers, but it is the responsibility of the application to check for duplicate data. This is made possible by the value of granulePosition mentioned before. The server sends collections of pages in bursts with the same granulePosition, which is a value that increments between collections. A connection may terminate in the middle of a collection of pages, but the server will always start sending pages at the start of a granulePosition border. With this knowledge a page-counter and granulePosition combined allows the SplicingAudioBuffer to patch data in the audio-track without making assumptions about the controlling thread.

To visualise the process two progress-bars were added to the user-interface, that show the amount of available data between two different streaming threads. Figure 1.(iii) shows a handover in place where the buffer in the first thread is being emptied (and will soon terminate on an EndOfStreamException). The other thread is already fetching data and is writing both to its local buffer (a BufferedOggStream) and the SplicingAudioBuffer, that now will employ the mechanisms to discards any duplicate data.

In order to make the application responsive to network changes a ConnectivityManager is created in each of the streaming threads. It will actively look for changes in connected and dis-connected states between network types. If the active connection becomes unavailable it will force a handover by closing the active thread, and spawning a replacement. The force-handover is also supplied by a button in the user-interface for testing purposes.