openBOXware
English documentation

Authors

Lorenz Cuno Klopfenstein lck@klopfenstein.net

Saverio Delpriori saveriodelpriori@gmail.com

Revision

24 April 2013

Contents

Introduction

What is openBOXware?

OpenBOXware is an open-source framework built on top of Android to provide a TV-like experience of multimedia contents taken from heterogeneous sources, while also allowing the end-user to enjoy all the applications installed on the underlying Android device.

To this purpose openBOXware sports a custom user interface conceived to offer a lean-back usage experience by means of three home screens, granting access to: the media library (Figure 1), the list of openBOXware applications (Figure 2), and the list of all other Android applications installed on the device. The media library is the default home screen, which allows the end-user to find media channels and to select the one to watch. Multimedia contents are made available by special add-ins, called media sources.

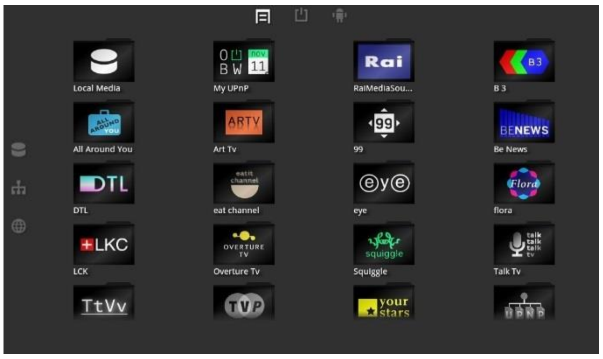

Figure 1: Media library

A media source is a tree of nested multimedia nodes. Leaf nodes are playable, in that they can be forwarded to the media player for playback, while all other nodes are explorable, in that they allow the media library to navigate their content and display the list of children nodes. Examples of media sources include IPTV channels, Internet TV channels, UPNP clients granting access to the multimedia contents made available by the UPNP servers discovered in the LAN, collections of media elements stored in the local file system, and collections of online multimedia contents.

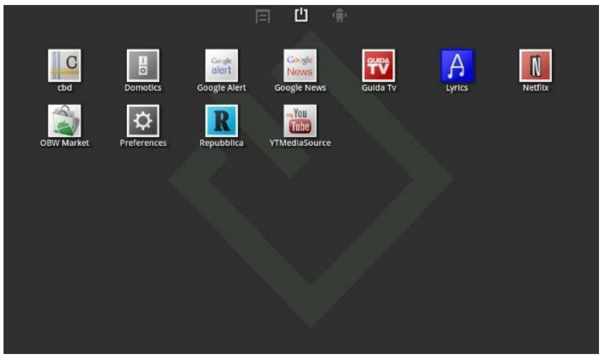

Figure 2: openBOXware applications home screen

Media source nodes may contain metadata, including title, duration, and icon, that can be displayed by the media library to provide a richer and more vivid browsing experience and to help the end-user to decide which content to pick. Figure 1 shows the media library home screen, with the icons of all the media sources installed in the device. When a media source is selected, the media library shows the icons of its children nodes.

The three small icons on the top of the screen can be used to switch among the three homes, while the ones on the left represent filters that can be applied to the media sources based on the location of the contents they link to: local file system, LAN, Internet.

Playable media source nodes provide a TV-like watching experience by offering both linear channels or contents on demand. A content on demand is nothing but a media source node associated with a single file which is played back whenever the media source node is selected. A linear channel, on the contrary, can be either a link to a continuous stream provided by a live streaming server, or a (possibly unlimited) list of disjoint multimedia elements which are glued together by the node to be played back as a continuous stream by the media player, shown in Figure 3.

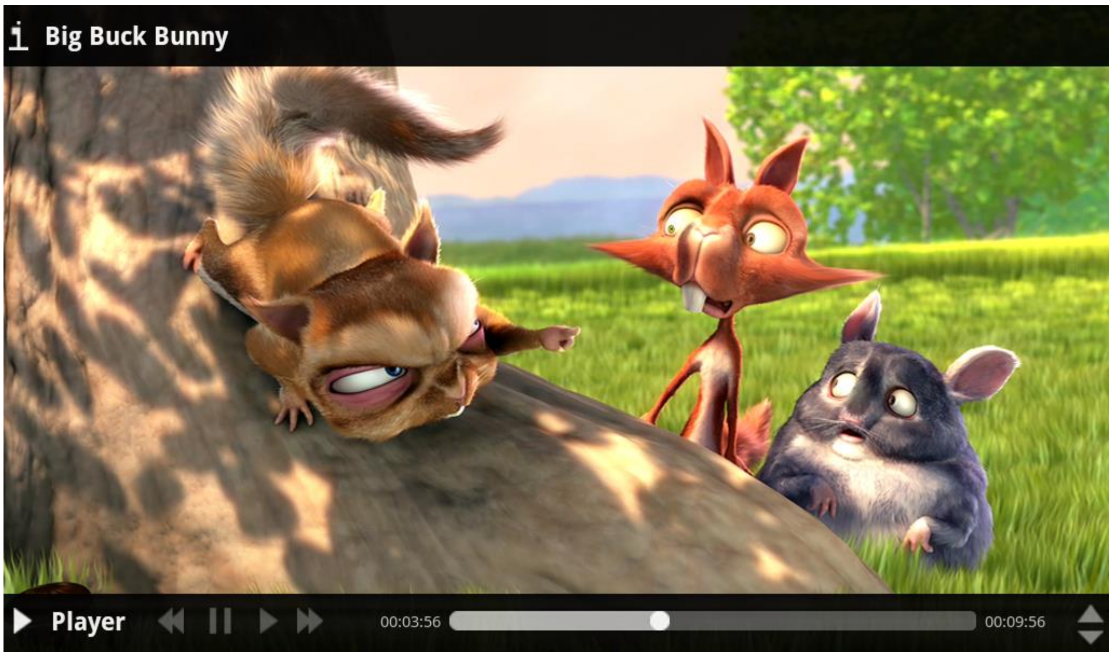

Figure 3: the openBOXware media player

Installing openBOXware

The openBOXware framework can be installed to any Android device from the Google Play application store:

> https://play.google.com/store/apps/details?id=it.uniurb.openboxware.launcher

Additional media source packages can also be installed through Google Play, by searching for *OBW-SOURCE* applications:

> https://play.google.com/store/search?q=OBW-SOURCE&c=apps

Accessing the source code

The full source code of the openBOXware framework is available as a mercurial repository on Google Code:

> http://code.google.com/p/openboxware-android/

The root `trunk` folder is intended to be used as an Eclipse workspace. Each project located inside the subdirectories can be imported into that common workspace.

The following projects are found inside the repository:

- `OpenBoxWareLauncher`: This is the main project, containing the openBOXware launcher and all shared subsystems (including the media player). This project is compiled and packaged to be the openBOXware application (as found on the Google Play store).
- `OpenBoxWareLib`: Library project that is shared among all openBOXware projects. This project contains all API code that is used throughout the system

and that enables communication between media sources and the launcher application.

- `FileSystemMediaSource`, `LastFmFreeTracks`, `UPnPMediaSource`, `YouTubeMediaSource`: Sample projects containing media sources for openBOXware (some of these are published on Google Play).
- `ConfUPnPMediaSource`: A sample configurator project containing a configuration application for the `UPnPMediaSource`.
- `ContentProviderTest`: A sample media source used for testing.

Notice: the `YouTubeMediaSource` represents a well-engineered sample project for accessing an online video service, but is *not* compatible with YouTube's licensing limitations and also was not updated following recent YouTube API changes.
It is not guaranteed to work and cannot be published. It is distributed along with openBOXware only as a media source template.

Implementing the Context

All openBOXware applications, while they may contain several different components, are packaged into one single Android APK and run as a single Android application.

In order to correctly be initialized and to correctly access their context, it is mandatory for openBOXware applications to derive from the `OBWApplication` base class, supplied by the openBOXware SDK.

```
public class MyApplication extends OBWApplication {

      @Override
      public void onObwCreate (){
            //Initialization
      }

}
```

This is needed because the `OBWApplication` class handles some special initialization cases and provides services which might be needed by other parts of the SDK you rely on (which will access the Android application context and expect it to be a class derived from `OBWApplication`).

Your application class must be specified in the Android manifest, as follows:

```
<application
      android:icon="@drawable/icon"
      android:label="@string/app_name"
      android:name="./MyApplication" >
```

Implementing a Media Source

An openBOXware "Media Source" is an Android service that provides access to a set of multimedia resources. The resources can be browsed by any other component (usually the openBOXware launcher, but not necessarily) navigating the hierarchy in which such resources are organized.

The openBOXware API library provides two methods of implementing a media source: either by using the low-level Android constructs (handling AIDL interfaces and binding to background services) or using the high-level interface. Using the latter is suggested, to avoid common programming pitfalls when implementing Android binding.

Low level API are located inside the `it.uniurb.openboxware.lib.media` package, while high level constructs inside `it.uniurb.openboxware.lib.media.tawheed`.

The Android manifest

In order for the media source to be visible to the system, a media source service must be capable of responding to a navigation Intent launched by external components (as before, this is *usually* the openBOXware launcher).

```
<service android:name="com.package.MyMediaSource">
      <intent-filter>
            <action
android:name="it.uniurb.openboxware.media.source.ACCESS" />
            <category
android:name="it.uniurb.openboxware.media.source.INTERNET" />
      </intent-filter>
</service>
```

A media source *must* be registered with an intent filter that responds to the `it.uniurb.openboxware.media.source.ACCESS` intent.

The `category` element is optional and specifies what kind of media access the media source requires. Giving no category tells the system that the media source is compatible with any network access. The following values are supported:

- `it.uniurb.openboxware.media.source.INTERNET`: The media source requires access to the Internet in order to access the media resources it exposes to the system.

Notice: adding this category only marks the media source as dependent on an Internet connection in order to work propertly, but does not assign Internet connection permissions. In order to gain access to the Internet in your media source's code, the permission `android.permission.INTERNET` must be added to the manifest as well.

- `it.uniurb.openboxware.media.source.LAN`: Used for media sources that require access to the local network, but not to the Internet (for instance, UPnP clients).
- `it.uniurb.openboxware.media.source.LOCAL`: The media source does not require any network connection.

Implementing the media source service

Your main media source class should implement the low-level media source API provided by openBOXware.

```
public class MyMediaSource extends MediaSource {

	@Override
	protected MediaSourceBinder createMediaSourceBinder() {
		return new MyTahweedMediaSource();
	}
}
```

This simple class works as the entry point to your media source (the name of this class must be the same of the service class exposed in your manifest file). The `createMediaSourceBinder` method must simply return a valid binder, using the facilities of the high-level API as seen below:

```
public class MyTahweedMediaSource extends MediaSource {

      public MyTahweedMediaSource(){
      }

      @Override
      protected MediaSourceNode getRootNodeTahweed() {
            //Return root source node here…
            return new MediaSourceNode();
      }

      @Override
      protected String getNameCore() {
            return "Test media source";
      }

      @Override
      protected it.uniurb.openboxware.lib.media.MediaSourceNode
getNodeFromCodeCore(String code) {
            return null;
      }

}
```

The high-level `MediaSource` class exposes a simple set of methods that allow callers to navigate the media source hierarchy for media elements to play back.

`getRootNodeTahweed()` returns a representation of the *root* node of the media source. Since each node can have any number of children nodes, the root node is used as the main access point for the user when navigating your media source. The root node should provide an access point that makes sense, depending on the kind of multimedia resources you are exposing.

For instance, the file system media source provides a *root* node that matches the root of the file system (since the file system as well is a hierarchical representation of files, the hierarchical structure of the media source can be matched directly). On the other hand, a media source providing access to YouTube may return a virtual *root* node that contains several child nodes, representing channels, searches, tags and so on.

`getNameCore()` returns the name of the media source, as displayed by the media library and the media player.

`getNodeFromCodeCore()` re-creates a node from a textual code. For further information about this mechanism, see “Node serialization to string codes” (page 16).

Implementing the nodes

The *root* node returned by the `MediaSource.getRootNodeTahweed()` method (and in fact every other node returned by the *root* node itself) must be an instance of a class extending `MediaSourceNode`, as shown below.

```
class MyNode extends MediaSourceNode {

	public MyNode(){
		super(MediaSourceNodeProperties.PLAYABLE |
			MediaSourceNodeProperties.PINNABLE |
			MediaSourceNodeProperties.EXPLORABLE);
	}

	@Override
	public MediaSourceNode[] getChildren() {
		//Create array of children nodes and return
	}

	@Override
	public MediaElementSource getMediaElementSource() {
		return new MyElementSource();
	}

	@Override
	public String getName() {
		return "My node";
	}

	@Override
	public String getIdentifyingCode() {
		return null;
	}
}
```

A media source implementation may contain several `MediaSourceNode` implementations (for instance, one for every kind of media element you are exposing).

The constructor of classes deriving from `MediaSourceNode` must specify the type of node they are representing, using the following flag values:

- `MediaSourceNodeProperties.PLAYABLE`: a playable node can be played back by the media player. When this flag is set, the node *must* provide a correct implementation of the `getMediaElementSource` method (see page 14) *and* an implementation of the `getIdentifyingCode` (see below).
- `MediaSourceNodeProperties.PINNABLE`: a *pinnable* node can be stored as a channel by the launcher and the user will be able to browse directly to this node, without passing through the *root* node. If set, the node *must* provide a valid `getIdentifyingCode` implementation (see page 16).
- `MediaSourceNodeProperties.EXPLORABLE`: if set, the node contains any number of child nodes. Those child nodes must be returned by the `getChildren` method.

All methods mentioned above may return `null` if unneeded (for instance, nodes without children can return `null` in their `getChildren` method and ignore the `EXPLORABLE` flag).

Warning: *root* nodes of a media source, even if declared as `PINNABLE` and/or `PLAYABLE`, will never be played back nor pinned by the openBOXware launcher. Only children of the *root* node (at any level) can be pinned and played back.

Media element playback

Nodes that can be played back must have the `PLAYABLE` flag set in their constructor. The media library will acknowledge the flag by allowing the user to select the "play" option and send the node to the media player.

The media layer uses a so-called `MediaElementSource` in order to enumerate all media resources contained by a node. Specifics of how this works depend on your implementation: a node may return a single media element (a single video), an infinite stream of resources (a video channel) or a finite set of elements (a directory containing a list of pictures).

The media element source API works by providing an implementation of the `MediaElementSource` class and a matching `MediaElementSourceEnumerator`:

```
class MyMediaElementSource extends MediaElementSource {

      @Override
      protected MediaElementSourceEnumerator createEnumerator() {
            return new MyMediaElementSourceEnumerator();
      }

}

class MyMediaElementSourceEnumerator extends MediaElementSourceEnumerator {

      boolean _valid = false;

      @Override
      public boolean moveToNext() {
            _valid = true;
            return true;
      }
```

```
	private final String URL =
"http://download.blender.org/peach/bigbuckbunny_movies/BigBuckBunny_320x180
.mp4";

	@Override
	public MediaElement getCurrent() {
		if(!_valid)
			return null;
		return new MediaElement(URL, "video", "mp4");
	}
}
```

The `MediaElementSource` implementation, at its core, must implement a single `createEnumerator` method, which will return the actual enumerator. In this sample, the class only forwards to the enumerator, but in your implementation it may need to access remote data, prepare a cache or do anything else to find out the actual source of media resources.

The `MediaElementSourceEnumerator` provides two simple methods to override: the `moveToNext` method moves the enumerator forward to the next element (this is usually called when the user skips tracks or when the player reaches the end of the current element) and the `getCurrent` method returning a `MediaElement` instance.

Notice that a newly instantiated enumerator starts in an “invalid” state (i.e. pointing *before* the first element of the list it is enumerating) and only after the first call to `moveToNext` it should be ready to return the current `MediaElement` instance. This behavior is taken care of by the `_valid` field in the sample above, which is turned to `true` only after the first access to `moveToNext`.

The sample enumerator will simply return an infinite list of Big Buck Bunny videos, thus representing a (very boring) TV channel.

Helper methods

Since enumerating a finite set of resources from a media source node is a very common scenario, implementing a full `MediaElementSource` and its enumerator would be very time consuming.

The `MediaSourceNode` class has a helper method that allows implementors to very easily enumerate the items of an array:

```
@Override
public MediaElementSource getMediaElementSource() {
      return this.enumerateArray(new MediaElement[] {
            //Populate MediaElement instances here...
      });
}
```

Your code will simply have to create an array of `MediaElement` items and call the `enumerateArray` method: the openBOXware API will take care of the enumeration, without the need of writing a custom `MediaElementSource` implementation.

Node serialization to string codes

Nodes of a media source are always accessed through the *root* node of a media source: that is, the user sees the media source, navigates to it from the media library, and then hierarchically explores the tree of nodes. This works in a way that is very similar to exploring a file system, starting from the root folder.

However, there are some cases in which direct access to a specific node of a media source is needed and desirable: the user has the ability to “pin” nodes to a channel number, thus accessing them directly without passing through the media source entry point.

This feature (which is enabled if a node of your media source provides the `PINNABLE` flag, as described in section "Implementing the nodes") requires that each node provides an identifying code. A code is a simple string value that uniquely identifies each instance of a media source's node tree.

The `MediaSourceNode` class must return this code from the `getIdentifyingCode` method. When calling the `getNodeFromCodeCore` method of the `MediaSource` class with that same string value, the corresponding node should be re-instantiated. Each string value should represent an unique token that allows users of a media source to identify a node, store it and restore it in a later moment (even if the media source service was terminated and restarted in the meantime).

Of course the identifying string value may contain any value the programmer finds useful to re-instantiate a valid media source node. For instance, the node representing a specific YouTube video may contain the hash ID of the video (since that is the only identifying value used to access that video). File system nodes may contain the full path of the folder they represent. Other nodes may contain any set of data, encoded in any way chosen by the programmer.

Implementing an Application

An "openBOXware application" provides an enhanced TV viewing experience by integrating with the openBOXware system and by providing a focused user interaction. Such applications can work in different modes.

Fullscreen activities

An openBOXware fullscreen activity takes over the whole screen (also hiding the Android system bars) and handles gestures and commands provided by the openBOXware system (to interact with features such as the sidebar and the channel list).

In order to be features as an openBOXware fullscreen application, the application manifest must include a special category:

```
<activity
	android:name="MyFullScreenActivity"
	android:label="@string/app_name">
	<intent-filter>
		<action android:name="android.intent.action.MAIN" />
		<category android:name="android.intent.category.DEFAULT" />
		<category android:name="it.uniurb.openboxware.gui.FULLSCREEN"
/>
	</intent-filter>
</activity>
```

Also, the activity must derive from `it.uniurb.openboxware.lib.gui.fullscreen.FullscreenActivity`, which implements much of the integration internally. The derived class must override the `onObwCreate` method instead of the Android `onCreate` method.

Background activity

The framework provides means for openBOXware applications to specify services that should run in the background and control their lifecycle from the integrated control bar. Such background services need to specify the following intent filter in their manifest:

```
<service
      android:name="MyBackgroundService" >
      <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="it.uniurb.openboxware.gui.BACKGROUND"
/>
      </intent-filter>
</service>
```

The class implementing the service must also extend the openBOXware class `it.uniurb.openboxware.lib.gui.background.BackgroundActivity`. Like fullscreen activities seen above, classes implementing background activities should override the `onObwCreate` method instead of `onCreate`.

A background activity associated with your application can be started and stopped by the user using the openBOXware launcher.

Additional services

The framework exposes some additional services to simplify development of applications.

Shared data

It is usually needed to share data between different activities of an application, especially if both a foreground and a background activity are running and operating on the same data set (e.g. the background activity is updating some data the user is interested in).

This can be achieved very easily in Android by using a hash table or a database in shared memory. However, for very simple use cases, you may want to use the `SharedDataService` provided by openBOXware. The service is accessed by calling the service manager:

```
OBWServiceManager m = OBWServiceManager.getOBWServiceManager(this);
SharedDataService sds =
      (SharedDataService)m.getOBWService(OBWServiceType.SHARED_DATA);
```

(Notice that the this pointer used at line 1 is a reference to the activity or the application context.)

The `SharedDataService` instance returned can act as a simple associative table for keys and values, with methods such as `addSharedData`, `updateSharedData` and `removeSharedData`.

Notifications

The openBOXware system, when running, replaces the default Android launcher and effectively hides the notification bar from the user. This is intended, in order to reduce visual clutter and distractions for the end user.

However, openBOXware applications may need to deliver urgent notifications to the user in a way that is not too intrusive to the TV watching experience: this is handled by the openBOXware notifications manager. You can get a reference to the service using the following code:

```
NotificationService s = (NotificationService)OBWServiceManager
      .getOBWServiceManager(this)
      .getOBWService(OBWServiceType.NOTIFICATION);
```

The openBOXware notification manager accepts any `Notification` instance that could normally be sent to the default Android notification manager. You may want to use the Notification.Builder facility in order to construct a valid Notification instance, and then register it calling the following method:

```
s.OBWNotification(NOTIFICATION_ID, notification,
      (NotificationManager)getSystemService(Context.NOTIFICATION_SERVICE));
```

Notice that you need to pass in a notification ID (any numeric value that helps you identify the notification) and the default Android notification service as the third parameter. This is needed since openBOXware notifications are also routed through the standard notification mechanism for consistency (they will also appear in the notification bar, if the user switches back to the default launcher).

References

Official openBOXware website:

```
http://www.openboxware.net/
```

Official openBOXware code repository:

```
http://code.google.com/p/openboxware-android/
```

L. Klopfenstein, S. Delpriori, G. Luchetti, A. Seraghiti, E. Lattanzi, and A. Bogliolo, “OpenBOXware on Android”, NEUNET Technical Report No. 12.004, 2012. DOI: 10.4459/12.004.TRP.EN

```
http://www.neunet.it/neunet-publications/12-004-trp/
```

L. Klopfenstein, S. Delpriori, G. Luchetti, A. Seraghiti, E. Lattanzi, and A. Bogliolo, “Introducing openBOXware for Android: The Convergence between Mobile Devices and Set-Top Boxes”, International Journal On Advances in Internet Technology 5.1 and 2 (2012): 44-53.

```
http://www.neunet.it/wp-content/uploads/2012/04/IARIAintech-12.pdf
```

Contact

Please contact Alessandro Bogliolo alessandro.bogliolo@uniurb.it for any enquiry and further information on the project.