UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Madis Pink

Reloading Android Application Resources at Runtime

Bachelor Thesis (6EAP)

Supervisors:
Jevgeni Kabanov
Margus Niitsoo

Author.. "...." May 2012
Supervisor.. "...." May 2012
Supervisor.. "...." May 2012
Professor.. "...." May 2012

Tartu 2012

Table of Contents

Introduction

In the last four years the adoption of smartphones has risen sharply. This has created a new market of mobile applications for the developers. Android is one of the leading platforms with 50.9% market share at the end of the fourth quarter of 2011 [1].

Android is an open source platform that can be freely licensed by any mobile device manufacturer. This has resulted in availability of a wide variety of Android devices – in addition to mobile smartphones Android has made its way onto tablets, television set-top boxes and tablet-laptop hybrids. Even within a device class the devices have varying screen sizes and resolutions, sensors, keyboard options, etc.

A part of the Android SDK has been devoted to the management of resources across different configuration classes – it is possible for an application to specify different layouts, bitmaps and values for different device configurations. An application may have separate layouts for tablets and smartphones, making optimal use of the different screen sizes. The Android SDK enables the developer to define multiple resource sets from a single codebase but this comes with a cost – the resource system is complex and difficult to grasp for new developers. As the resources are compiled and bundled with the application at build it increases the build and turnaround times when developing Android applications. This is especially evident for applications with large number of resources.

This thesis explores a method to reload Android application resources at runtime. The method is used as a basis to construct a tool for developers called Catalyst that enables an Android developer to see results inflicted by resource changes near realtime and without rebuilding-redploying or restarting the application.

Chapter 1 gives a general overview of Android application development process along with the tools used to build Android applications. Chapter 2 describes the Android resource framework in detail by listing various resource types, their basic usage and the way Android represents and interprets them internally. Chapter 3 is an overview of various technologies and techniques used to implement Catalyst. Chapter 3 also provides a short set of measurements done with and without using Catalyst. The full source code for the utility developed is stored on a compact disc which is included with this thesis as Appendix 1.

1. Development Process of Android Applications

1.1. Android Architecture

The Android operating system is based on a modified fork of Linux kernel. On top of the kernel there are 3 distinct layers - a set of native system libraries, the Android runtime consisting of Dalvik Virtual Machine with its support libraries and the Application Framework which is commonly referred to as the official API [2].

The Dalvik VM is a register-based VM that runs Dalvik bytecode, which is heavily optimized for mobile applications. Java bytecode can be converted to Dalvik bytecode by the dx tool bundled with Android SDK. This makes developing applications with Java possible.

The Application Framework is a collection of frameworks enabling applications to use UI widgets/libraries and to interface with other system components including, but not limited to: applications via IPC, the native system libraries and system hardware. The visible part of the Application Framework is the public API and extensively documented by Google. The framework also provides facilities for application resource retrieval.

1.2. Structure of an Android Application

An Android package consists of four categories of contents [3]:

1. Dalvik bytecode files (optional)
2. An indexed resource table (optional, present if the package has resources)
3. Uncompiled resources bundled with the application (optional)
4. Manifest file (required)

Android SDK contains tools to convert and package source files for these components. Google has provided two toolchains – an Eclipse plugin called ADT (Android Developer Tools) and a number of command line tools enabling the packaging of applications without the Eclipse IDE. Every package is uniquely identified by a package name, a string identifier consisting of latin characters and dot characters. The package name follows Java package naming conventions and is defined in the manifest file.

According to Google the build process of the ApiDemos sample takes roughly 20 seconds [4]. ApiDemos is a medium-sized sample Android application distributed with the Android SDK and has around 500 resource files. Larger projects may have a greater amount of resource files – up to multiple thousands. In these cases build times can take more than 2 minutes.

1.3. The Build Process

The default build procedure for Android applications consists of the following sequence of steps [5]:

1. Android asset packaging tool (*aapt*) crawls through the resource folder and generates an integer ID for every resource. These ID's are written into *R.java* source file under generated sources folder where they can be accessed from application code.
2. Java compiler (*javac*) compiles the application's java source files along with generated source files (including *R.java*).
3. Dexer (*dx*) translates the compiled Java bytecode into Dalvik bytecode (dex) format.
4. Aapt takes resource ID's and the resource folder as input and:
    1. Runs *pngcrunch,* a PNG compression tool, on any bitmaps under resource folder
    2. Adds 9-patch PNG chunks to 9-patch images
    3. Compiles any XML resources into Android XML binary format
    4. Constructs a resource table that contains the identifiers and configurations for all resources. Also contains primitive resources (scalars).
5. *Apkbuilder* zips the resource table, application code and compiled resource into an apk file.

The SDK tools starting from version r14 skip some parts of this process if possible. Lets consider the case where developer makes changes to application code - in that case the tools can simply rerun steps 2., 3. and 5. as no resources were changed. This helps to save some time.

If a single resource is changed, e.g. a bitmap file, then steps 1., 2. and 3. can be omitted - the code hasn't changed and there were no changes in resource ID's (no resource added nor deleted). If a developer adds a new resource file then all of the build steps have to be executed because resource ID's have to be generated again.

1.4. The Impact of Build Times on Development

There are two ways to develop UI for Android applications: one either creates and composes widgets from the code or alternatively uses the standard layouts. Developing user interfaces is usually an iterative process and for that reason the long deployment times are a major hurdle for developers. A brief survey in a small Tartu IT company (Mobi Solutions OÜ) revealed that on average an Android developer makes 25-100 rebuilds every day.

There are a few ways to mitigate this problem:

1. One can use the layout editor view bundled with the Android Eclipse plugin (ADT). The layout editor bundled with ADT is based on a x86 port of the resource framework present on the ARM builds of Android. This fairly straightforward approach works for simpler layouts. It fails however in cases where the visual state is dependent on the business logic of the application, e.g., when displaying a dialog for a specific event or dynamic data inside the layouts. Some developers use stub values when using the graphical layout editor but this approach is error-prone as such stub values may end up in production code due to developer errors.
2. One can write apps in HTML and either test them in Browser or use the UI WebView widget (basically an embedded web browser inside an application). This approach is a tradeoff between development time and the look and feel of the application as all the UI widgets are defined in HTML and do not look native to the platform.

There are a number of frameworks (Phonegap, Titanium) that allow to construct native UI widgets via JavaScript but these solutions come with a heavy performance overhead since the application needs to load a JavaScript virtual machine alongside Dalvik. It is however a viable alternative to achieve cross-platform compatibility with minimal codebase.

The current SDK toolset is designed with the application runtime performance in mind, trading build time length for runtime performance. There are numerous examples of this tradeoff. To name a few:

1. PNG bitmaps are crunched every time during a build
2. XML files are compiled into a binary format for more efficient run-time parsing
3. Integer values are represented as strings in pre-compiled format and translated to 32-bit values during the build

The core concept of Catalyst is to engineer a way around this limitation by compiling the resources periodically and provisioning them via a network stream at runtime. This enables a developer to do incremental updates to resources without suffering the rebuild/redeploy time penalty.

2. Android Resource Framework

2.1. The Structure of the Resource Folder Tree

Application resources are compiled from a source resource folder. By default this folder is named *res*. The *res* folder contains subfolders named with the following mechanism: the name consists of a type name followed by zero or more qualifiers, e.g. *drawable-hdpi* subfolder is a folder of drawables with the high dpi (density) qualifiers. A qualifier is a short string which specifies the device configuration(s) for which the underlying resources should be used. For instance a layout in the *layout-land* folder is only used when the screen orientation in the current configuration is landscape. Only one qualifier of a type can be specified, for instance *values-et-en* would be an invalid name. Qualifiers must be ordered by the precedence specified in the Android SDK documentation [6].

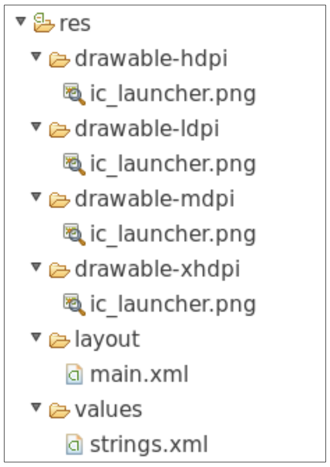
If the conditions set by the folder qualifiers are not met then the next subfolder of that type is picked until a resource is found within a folder for which the qualifiers are not conflicting with the device configuration. The algorithm for this process is the following [6]:

1. Eliminate resource files that contradict the device configuration
2. Pick the (next) highest-precedence qualifier
3. If no resource folders include that qualifier then return to step 2
4. Eliminate resource directories that do not include this qualifier
5. Repeat steps 2-5 until a single qualifying folder remains

The resource types can be split into three core classes - value resources, XML resources and raw resources. Value resources are simple scalars and arrays, XML resources describe complex data structures (e.g. shapes, layouts, menus, animations) and raw resources are raw bitstreams. Bitmaps (PNG, GIF, JPEG) are a special case of raw resources.

One of the simpler examples of res folder structure can be observed when one creates a new Android application in Eclipse. The Eclipse plugin generates a stub “Hello World!” application which has 4 resources. The contents of the application’s resource folder, shown in

Figure 1, are: a PNG bitmap named *ic_launcher*, a layout named *main* (Figure 2) and two strings named *hello* and *app_name* (Figure 3). The bitmap has four different screen density configurations: *ldpi*, *mdpi*, *hdpi* and *xhdpi*. The layout is a simple XML file containing a *TextView* element inside a *LinearLayout* (these correspond to the user interface widget classes in the Android API). The strings are simple scalar resources.



```
res
  drawable-hdpi
    ic_launcher.png
  drawable-ldpi
    ic_launcher.png
  drawable-mdpi
    ic_launcher.png
  drawable-xhdpi
    ic_launcher.png
  layout
    main.xml
  values
    strings.xml
```

Figure 1. Resource folder tree of the "Hello World" project

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical" >
   <TextView
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:text="@string/hello" />
</LinearLayout>
```

Figure 2. The res/layout/main.xml file in a new Android application

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="hello">Hello World, HelloActivity!</string>
   <string name="app_name">Hello</string>
</resources>
```

Figure 3. The res/layout/strings.xml file in a new Android application

2.2. Referencing Resources

Android internally tracks resources with integer ID's. The ID is a 32-bit signed integer consisting of 3 regions: the high 16 bits contain the package namespace and the type (8 bits for each with the package namespace being higher) of the resource whereas the low 16 bits describe the actual ID. For instance application string resources usually have the ID in the range of *0x7f040000 - 0x7f04ffff*. This imposes a theoretical limit on the number of resources within a package-type pair. Different configurations of the same resource share the same ID so the contents of the resource can be fetched on the basis of the ID and the current device configuration.

The integer ID's are generated at build-time and stored in a generated source file (*R.java*) as integer constants. The resulting Java class is placed in the same package as the applications package (as declared in the manifest file). The resource ID's can be accessed from Java code using the generated *R* class, i.e., a resource ID for a string resource with the name *hello* is stored in *R.string.hello* (in the application namespace). XML and value resource types can also reference other resources via the following syntax: *@[package:]type/name*. Package name is optional and may omitted, in this case Android assumes application's own package by default. For instance, in an application with the package name *com.madisp.hello* the identifiers *@string/hello* and *@com.madisp.hello:string/hello* are identical [7].

The platform resources are in a package named android and with an ID of 1. The resulting *R* class (*android.R*) is accessible at runtime, for example, to reference the platform string yes one would use the value stored in *R.android.string.yes* in runtime. The respective XML reference would be *@android:string/yes*.

2.3. Value Resources

Value resources are the simplest form of Android resource types. They can be used to store two types of values: either scalars (strings, integers, booleans, colors, dimensions) or integer to value dictionaries which are called bags. Examples of bags are arrays, styles and quantity strings (plurals).

These values are defined in XML files in resource tree subfolders prefixed with "*values*". For instance, one can provide localized strings in *values-et* subfolder. An example values XML file is given in Figure 4.

The underlying platform implementation is in *TypedValue* class (scalars) and *TypedArray* class (bags). The values are compiled into *TypedValue* and *TypedArray* binary representations and are stored into the resource table file (*resources.arsc*). One can use the *getValue* and *obtainTypedArray* methods from the *android.content.res.Resources* class to access scalars and arrays during the runtime. These methods take the resource ID as an argument.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">Hello</string>
   <string name="hello">Hello World!</string>
   <string name="parametric">Hello, %s</string>
   <color name="white">#ffffff</color>
   <bool name="valTrue">true</bool>
   <integer name="one">1</integer>
   <string-array name="arr">
       <item>one</item>
       <item>two</item>
   </string-array>
</resources>
```

Figure 4. An example XML file defining various value resources

2.4. Raw Resources

Raw resources are essentially bitstreams that can be opened during the runtime. In a way they are similar to standard Java resources as they are copied into the apk file with no modifications. The main difference is that during build-time every raw resource gets a representing string resource in the resource table. The string resource contains the relative path of the raw resource and can be used to resolve the variant of the resource path given current device configuration. This enables the developer to use different files for different configurations.

2.5. XML Resources

XML resources are raw resources that contain XML files compiled into Android XmlBlock format. The XmlBlock format is a binary structure that has a string pool in the beginning of the file followed by constant-size blobs containing the XML events enabling an efficient XML pull parser implementation for the format.

The format supports UTF-16 and UTF-8 string encodings, tags, attributes, namespace delcarations, namespace prefixes for both tags and attributes, text and CDATA sections. Althouth text and CDATA sections are supported they are currently not used by any XML representations of complex resources.

XML resources can be placed in a number of subfolders: *xml* (generic files), *layout*, *drawable folder* (shapes, gradients), *anim* (view animations), etc. Like any other raw resources the XML resources are represented in the resource table as relative path strings (e.g. "*res/layout/main.xml*"). They can be obtained during runtime by using the *getXml()* method from the *Resources* class which returns an *XmlResourceParser* instance. The process of parsing layout files is commonly referred to as inflation. XML files may reference other resources in the format described at the end of Section 2.2. These references are resolved at runtime. Figure 5 is an example of a layout with a string reference.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical">
   <TextView
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:text="@string/hello" />
</LinearLayout>
```

Figure 5. An example XML layout resource with a string resource reference

An interesting notion is the concept of attribute resources. One can define special *TypedValue* resources called attributes by describing the attribute name and value formats via *declare-styleable* tag under the *values* folder. These are given identifiers under the *attr* type and can be accessed by declaring an XML namespace in the format of *http://schemas.android.com/apk/res/<package name>*. This means that a set of attributes for a tag can be interpreted as a bag of scalar resources where the keys are attribute resource IDs and values are either scalar values or references to other resources.

The concept of attribute resources is used extensively throughout Android XML files. An example XML layout given in Figure 5 references attribute resources from the standard framework resource package (*android*) by declaring a namespace prefix *android* and binding namespace URI *http://schemas.android.com/apk/res/android* to said namespace prefix.

The definitions of these attributes are bundled with the Android SDK in the platforms folder. For instance, to see the *android:orientation* attribute one has to seek out the attribute definition in the *platforms/android-15/data/res/values/attrs.xml*, the relevant excerpt is given in Figure 6.

```
<!-- Standard orientation constant. -->
<attr name="orientation">
   <!-- Defines an horizontal widget. -->
   <enum name="horizontal" value="0" />
   <!-- Defines a vertical widget. -->
   <enum name="vertical" value="1" />
</attr>
```
Figure 6. Definition of the orientation attribute

This definition shows that the orientation attribute has two possible integer values - 0 and 1. The values are aliased with the strings *horizontal* and *vertical*. Thus the attribute in the form *android:orientation="vertical"* can be interpreted as an entry in a bag with the key *android.R.attr.orientation* and value 1.

2.6. Bitmap Resources

Bitmaps are another class of raw resources that are placed in the *drawable* family of subfolders. Android supports PNG, JPEG and GIF bitmap files although the use of last two types is discouraged due to their limitations - they do not have a blendable alpha channel, JPEG is a lossy format and GIF does not support more than 256 colors in a single image.

In addition to the listed three types a developer can also provide a Nine-Patch bitmap. When used as container backgrounds Nine-Patch bitmaps will be stretched by the system to accommodate the contents of a container. This is achieved by defining stretchable areas and content areas for the image by adding an extra 1-pixel border for the image. A region is defined by placing black pixels in the border (other pixels must be fully white or transparent). The top and left borders define the stretchable pixel region(s) while the bottom and right borders define the content area [8]. These borders are removed and encoded into PNG text chunks at build-time. The Android SDK tools contain a program, called *draw9patch*, to manipulate uncompiled 9-patch files.

Bitmap files can be accessed during runtime by using the *getDrawable()* method from the system *Resources* class. It is worth noting that the method can also return other, non-bitmap, drawable types such as *GradientDrawable* or *ColorDrawable*. These are defined by an XML file instead of a bitmap.

2.7. The Resource Table

The resource table is a collection of *TypedValue* and *TypedArray* objects serialized in a binary form and distributed inside the Android package file. *TypedValues* and *TypedArrays* are grouped by the packages, resource types and configurations, in that order. In most scenarios there is only one package in a resource table file - the application's package. Resources within a package are grouped by resource type and a resource type may contain a number of configurations. Empty package/type/configuration groups are omitted from the final file. All of this is easier to understand by analyzing the dump of *aapt* tool for the example "Hello World" project created by the "New Android Application" wizard in Eclipse.

For instance, the two strings in a hello world program (*app_name* and *hello*) are stored in the application's package group, with the type *string* and under the default configuration (since the resource subfolder, *values*, does not have any qualifiers). This can be observed with the *aapt* tool's *dump resources* command (Figure 7). For the sake of brevity parts of the output have been omitted.

```
Package Groups (1)
Package Group 0 id=127 packageCount=1 name=com.madisp.hello
 Package 0 id=127 name=com.madisp.hello typeCount=4
   type 3 configCount=1 entryCount=2
     resource 0x7f040000 com.madisp.hello:string/hello: t=0x03
       (string8) "Hello World!"
     resource 0x7f040001 com.madisp.hello:string/app_name: t=0x03
       (string8) "Hello"
```
Figure 7. Two strings in a resource table

The bitmap representing the application's launcher icon, *ic_launcher*, has three distinct configurations, based on the device's display density: *hdpi*, *mdpi* and *ldpi*. It is important to note that although due to this there are 3 entries in the resource table they all share the same resource ID. The *aapt* resource table dump for *ic_launcher* is given in Figure 8.

```
Package Groups (1)
Package Group 0 id=127 packageCount=1 name=com.madisp.hello
 Package 0 id=127 name=com.madisp.hello typeCount=4
   type 1 configCount=3 entryCount=1
     config 0 density=120 sdk=4
       resource 0x7f020000 com.madisp.hello:drawable/ic_launcher: t=0x03
         (string8) "res/drawable-ldpi/ic_launcher.png"
     config 1 density=160 sdk=4
       resource 0x7f020000 com.madisp.hello:drawable/ic_launcher: t=0x03
         (string8) "res/drawable-mdpi/ic_launcher.png"
     config 2 density=240 sdk=4
       resource 0x7f020000 com.madisp.hello:drawable/ic_launcher: t=0x03
         (string8) "res/drawable-hdpi/ic_launcher.png"
```
Figure 8. A drawable with three different configurations in a resource table

It is also evident from the output that qualifiers can imply multiple configuration flags for a resource. For instance support for densities was added with SDK version 4 so the density qualifiers also imply the SDK version 4 qualifier.

2.8. Resources Runtime Library

Android applications do not have a classical main method as an entry point. A notion of components is used instead. Components are implemented by deriving the core component base classes and the implementations must be declared in the application's manifest file. There are 4 possible component types and they are: activities, services, broadcast receivers and content providers. Activity is an abstraction of a fullscreen UI view tied to a specific action - for instance the contact list screen and contact details screen are separate activities in the standard Contacts application provided by Google as part of Android Open Source Project source code. Services are background tasks that do not require any user interaction. Broadcast receivers listen to system events like receiving an SMS or empty battery notification. Content providers are an abstraction over data storage, e.g. the SMS application provides a content provider to query SMS inbox and outbox.

Every component has a reference to a *Context* object. *Context* objects enable application components to interface with system services and device hardware via IPC. *Context* objects hold current device configuration and have an interface for accessing application and system resources via a *Resources* object which can be obtained by calling *Context.getResources()*. The *Resources* class is basically a caching proxy for *AssetManager* - the actual class that is responsible for reading in resources and parsing the resource table. Parts of *AssetManager* are written in C++ and it shares some of its code with the *aapt* tool.

For illustration, consider a short example of behind-the-scenes method call trace with an activity from the hello world project. The *Activity* implementation calls *setContentView(R.layout.main)* (Figure 9).

```
package com.madisp.hello;

import android.app.Activity;
import android.os.Bundle;

public class HelloActivity extends Activity {
   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.main);
   }
}
```
Figure 9. A simple Activity using the resources framework to set the screen layout

The *setContentView(int layoutResId)* is a convenience method to *LayoutInflater.inflate(int resource, ViewGroup root, boolean attachToRoot)* which in turn calls the *getLayout(int id)* method from the *Resources* class to retrieve an *XmlResourceParser* (Figure 10).

```
public View inflate(int resource, ViewGroup root, boolean attachToRoot) {
   if (DEBUG) System.out.println("INFLATING from resource: " + resource);
   XmlResourceParser parser = getContext().getResources().
      getLayout(resource);
   try {
      return inflate(parser, root, attachToRoot);
   } finally {
      parser.close();
   }
}
```
Figure 10. The source code of the *LayoutInflater.inflate* method [9]

The *getLayout(int id)* method, in turn, is a simple wrapper around *loadXmlResourceParser(int id, String type)* (Figure 11). The latter is hidden from the public API.

```
public XmlResourceParser getLayout(int id) throws NotFoundException {
   return loadXmlResourceParser(id, "layout");
}
```
Figure 11. The source code of the *Resources.loadXmlResourceParser(int id)* method [10]

As explained in section 2.5 XML resources are simply a special case of raw resources. Raw resources are not stored in the resource table - a string with the path of the resource is stored instead. This is evident when looking at the *loadXmlResourceParser(int id, String type)* method's source (Figure 12).

```
/*package*/ XmlResourceParser loadXmlResourceParser(int id, String type)
       throws NotFoundException {
   synchronized (mTmpValue) {
       TypedValue value = mTmpValue;
       getValue(id, value, true);
       if (value.type == TypedValue.TYPE_STRING) {
           return loadXmlResourceParser(value.string.toString(), id,
                   value.assetCookie, type);
       }
       throw new NotFoundException(
               "Resource ID #0x" + Integer.toHexString(id) + " type #0x"
               + Integer.toHexString(value.type) + " is not valid");
   }
}
```
Figure 12. The source code of the *loadXmlResourceParser(int id, String type)* method [10]

The *getValue* method retrieves a *TypedValue* from the resource table via *AssetManager*. As can be seen from the code the returned value ought to be a string type (containing the path for the resource, in this case it is "*res/layout/main.xml*". The value is passed on to internal *loadXmlResourceParser(String file, int id, int assetCookie, String type)* method which uses *AssetManager* to open the file as a stream and passes the stream to a *XmlResourceParser* implementation which is later used by *LayoutInflater* to inflate the layout. Relevant parts of the method are given in Figure 13.

```
XmlBlock block = mAssets.openXmlBlockAsset(
       assetCookie, file);
if (block != null) {
   int pos = mLastCachedXmlBlockIndex+1;
   if (pos >= num) pos = 0;
   mLastCachedXmlBlockIndex = pos;
   XmlBlock oldBlock = mCachedXmlBlocks[pos];
   if (oldBlock != null) {
       oldBlock.close();
   }
   mCachedXmlBlockIds[pos] = id;
   mCachedXmlBlocks[pos] = block;
   //System.out.println("**** CACHING NEW XML BLOCK!  id="
   //                   + id + ", index=" + pos);
   return block.newParser();
}
```
Figure 13. The code in *Resources* class that instantiates a new *XmlResourceParser* object [10]

3. Implementation of Catalyst

3.1. General Architecture

The core concept of Catalyst is to enable an application to override resources at runtime. This enables a developer to make changes to an application without the requirement to rebuild the application to see the results. To achieve this, the Catalyst was designed as a two-component application: the server and the runtime client library.

The server is essentially a reimplementation of resource table functionality - it provides an interface to query *TypedValue* and *TypedArray* objects. The device configuration is passed along with a *TypedValue* request to enable device-specific resource configuration just as the native resource table implementation. Additionally the server enables retrieving raw resource files as bitstreams. One core difference between the native resource table implementation and the server is the requirement of updates - the server needs to constantly scan the filesystem for changes and propagate these events to the underlying resource table reimplementation.

The client is a thin wrapper over the *Resources* class and proxies *TypedValue* request to the server instead of using the native *AssetManager* implementation. If the server does not have the resource or an error is encountered then the client uses the value from the system's *Resources* class instead. A local cache on the device is also used to reduce the amount of requests done to the server.

3.2. Server Implementation

3.2.1. Lifecycle and Initialization

Since Android uses build-time generated integer ID’s for resources we cannot start the server before *aapt* generates ID’s for the resources. This means that the lifecycle of the server is closely tied with the build process - the server can be started after *R.java* is generated and must be restarted every time the developer rebuilds the application since the resource table and the resulting ID’s might have changed, leading to inconsistencies between the application’s runtime code and the resource table in the server.

At initialization the server parses the project's manifest file (*AndroidManfiest.xml*), the *R.java* file and crawls the application's resource folder. Any discovered resources are loaded into the internal resource table and a recursive file monitor is started. The file monitor notifies the server if any file within the resource folder has been changed, added or deleted.

3.2.2. Data Model

During the design of the data model the two main events for the server were considered:

1. when a file within the *res* folder changes
2. when the client requests a resource by ID

To cater these requirements the resources are grouped by the files they were read from. In the case of raw resources, the files have a single resource, in case of value resource files, such as *strings.xml*, the file contains multiple resources. The resource files are uniquely identified by their relative path within the main resource folder.

Resource files with multiple resources are referred to as buckets. Every resource file is also assigned a set of qualifiers based on the parent folder's name. Example: *res/values-et/strings.xml* is a bucket since it may contain more than one resource file. Additionally the representing resource file object has the *language=et* qualifier.

Every resource has a matching *TypedValue* or *TypedArray* object and optionally a file location (in the case of raw resources). This mirrors the native resource table implementation where every resource is represented by a *TypedValue*. Since the retrieval of these values is based on the resource ID a map of type *id* -> *{resfile1, resfile2, ..., resfileN}* is created where *id* is the resource ID and *{resfile1, resfile2, ..., resfileN}* represents the list of all resource files containing that resource. If the list has more than 1 elements then the algorithm described in section 2.1. is applied until there is a single remaining qualifying file.

3.2.3. Value Resources

As described in section 3.2.3, value resources are read into buckets – resource files containing multiple resources. This is due to the possibility of defining multiple values within a single XML file. Typically this is used to separate value resources into sensible groups, e.g., strings, booleans, colors, integers, etc. A value resource file is read into memory as a set of *TypedValue* and *TypedArray* objects at startup. If the file monitor observes a change then the resource file is read again and the old set of resources are replaced.

3.2.4. Raw Resources

General raw resources, e.g., anything under the *res/raw* folder are handled differently than value resources. Firstly, there is only a single resource per file. Secondly there is no need to read in the values since they are not compiled by *aapt* at build time in any way - directly piping the file contents over network at request time is sufficient.

As with the native resource table implementation all raw resources (including bitmaps and XML files) are represented as string *TypedValues* where the value is the relative path to the resource. A special internal flag is used to distinguish these *TypedValues* from actual string resources.

3.2.5. Bitmaps

Bitmaps can not be directly piped over the network due to the nature of Nine-Patch bitmaps - the stretchable and content areas are defined as a 1 pixel frame around the image in the precompiled form whereas in the compiled form the areas are encoded in custom PNG chunks along with the pixel data.

The *aapt* tool has an option to compile the files without doing any other resource processing, the *crunch* command. This command takes two folders as arguments – a source folder and a cache folder. If the cache folder does not contain any files then *aapt* crunches all the PNG files in the source folder with the *pngcrunch* tool and additionally compiles any Nine-Patch bitmaps. The latter are indicated with a *.9.png* filename suffix. If the cache folder is not empty, then *aapt* compares the source file and destination file modification timestamps and processes only the files for which the source timestamp is newer than the destination file timestamp.

This effectively means that the Catalyst server can run the *aapt* crunch command over the whole base resource folder every time a bitmap is modified – since only one file is changed the processing takes only a fraction of the time it takes to process all the bitmaps in the source folder.

3.2.6. XML Resources

The straightforward approach would be to transmit XML resources as is in the precompiled form and write a client-side *XmlResourceParser* implementation. *XmlResourceParser* is a Java interface inheriting from both *XmlPullParser* and *AttributeSet* interfaces and for that reason implementing the interface would be an easy task. The way how platform currently inflates layout XML files makes this approach impossible.

As described in section 2.5. the XML tag attributes can be interpreted as a bag of scalar values. Sadly in this process the framework typecasts the *AttributeSet* interface into a *XmlBlock.Parser* type during layout inflation. The *XmlBlock.Parser* class is final and the method wherein the typecasting takes place belongs to a final class. Android source code contains a comment indicating this problem and its reprecussions (Figure 14). It is worth noting that this comment and the typecast has been in the Android Open Source Project source code repository since version 1.0 which was released in the September of 2008.

This effectively means that the XML files have to be compiled to the XmlBlock format either on the server or the device prior to inflation. Catalyst does this process on the server since the average modern developer machine has considerably more processing power than a handheld device.

```
// XXX note that for now we only work with compiled XML files.
// To support generic XML files we will need to manually parse
// out the attributes from the XML file (applying type information
// contained in the resources and such).
XmlBlock.Parser parser = (XmlBlock.Parser)set;
```
Figure 14. Excerpt from the *obtainStyledAttributes* method in the *Resources.Theme* class [10]

Additionally the process of interpreting *AttributeSet* as a *TypedArray* is done in the *AssetManager* class instead of the *Resources* class. This means that if an XML file contains reference to any scalar resources they are always read from the bundled resource table. The *AssetManager* class is final and cannot be extended to alter this behaviour. The server replaces any references to a scalar value with the actual *TypedValue* as a workaround, a process called shadowing. This enables the developer to change the referenced resources and observe the changes inflicted without recompiling the bundled native resource table. The shadowing process reuses the code written for retrieving a *TypedValue* interpretation of a resource by resources ID.

3.3. Transport and Serialization

Initially the HTTP protocol was chosen for resource transport due to a large number of HTTP tools and libraries available. Also the default Oracle Java VM has a HTTP server implementation which was used. It was not a good strategy, however – this imposes the VM limitation on developers and for that reason a custom protocol implementation on top of *ServerSocket* was written instead. *ServerSocket* is the standard Java API for implementing TCP servers. Although UDP offers better performance TCP was chosen because of built-in packet integrity and packet order guarantees.

In addition to the TCP socket interface the server listens on an UDP port for multicast packets. Every time a new Catalyst-enabled application starts on a device a UDP multicast packet is broadcast over the WiFi network with the application package name. If the server has said package's resources in its resource table it sends an empty response packet back to the multicast origin. This enables the client side implementation to discover the server IP. For emulator this step is unnecessary since an emulator can always connect to host machine using the magic IP 10.0.2.2 [11].

The server exposes two endpoints: one for querying *TypedValues*/*TypedArrays* and one for retrieving raw resources. The protocol consists of request (device to server) and response messages (server to device). A request contains application package, unique server tag, device configuration and either a resource ID or a raw resource path. The application package and server tag combination is used to uniquely identify developer's machine – this is necessary if there are multiple developers working on the same project within the same LAN network. The configuration is used to resolve resources according to the algorithm described in section 2.1. For instance if a resource has both landscape and portrait orientation versions the server needs the device configuration to pick the best-matching resource.

The response consists of either a serialized *TypedValue*/*TypedArray*, if the request was by resource ID, or a binary blob, if the request was a specific raw resource path. If a resource was not found or another error occurred than an error flag with the appropriate reason is set in the response.

Both messages are serialized with Google Protocol Buffers [12]. The latter was chosen due to its binary nature, extensive documentation, availability of an official Java implementation and a small serialization overhead both in terms of performance and resulting binary stream length.

3.4. Client Implementation

3.4.1. General Client Architecture

Android Catalyst client library is a relatively lightweight library consisting of three classes and the Google Protocol Buffers Java library. The network code is implemented in *RemoteAssetManager* - a class that mimics the system's *AssetManager* class - that provides facilities to retrieve and deserialize *TypedValue*/*TypedArray* objects and raw resources from the server.

RemoteAssetManager is used by the *CatalystResources* class, the latter is a wrapper around the standard *Resources* class that forwards *getValue*, *getTypedArray*, *loadDrawable* and *loadXmlResourceParser* methods to *RemoteAssetManager*. If fetching remote resources fails for some reason then the resources from native implementation are provided instead as a fallback.

Additionally there is a *CatalystContext* class which extends *ContextWrapper* with the sole purpose of returning a *CatalystResources* class instead of the standard system's *Resources* from the *Context.getResources()* method. This wrapper is used to enable remote resource fetching for any application component.

3.4.2. Resources Proxy Injection

Out of the four possible application components - activities, services, content providers and broadcast receivers - Catalyst focuses on the former since almost all of the resources are used to construct layouts. Layouts are tightly integrated with the activities with the possible exception of system-wide notifications and home screen widgets.

The *Activity* class (baseclass for all activity components) extends the *ContextWrapper* class. *ContextWrapper* is essentially a proxy - it receives the underlying *Context* via a *attachBaseContext(Context base)* method call and forwards all *Context* methods to the underlying base context. Catalyst overrides this method and wraps the base context in a custom *ContextWrapper* that uses a proxy *Resources* class implementation instead of the native one. This proxy *Resources* implementation is used to query resource values from the server instead of the native *AssetManager* class. This enables loading updated resources at runtime only for activity components but a similar approach could be leveraged for services as well.

The process of enabling application components to use Catalyst is automated. Javassist [13], a popular tool to manipulate Java bytecode files, is used to inject *attachBaseContext* method to every activity declared in the application's manifest. These tools need to be run after the application source files have been compiled with *javac* and before the class folder has been converted to Dalvik bytecode by the dexer.

In addition to adding the *attachBaseContext* method to all the activities one must also include a set of Catalyst classes and Google Protocol Buffers Java library to one's application. This is only desirable during development because these libraries make the release build bigger than it needs to be and adding/removing support classes manually is inconvenient for the developer. Catalyst contains tools that automate this process by copying any necessary client classes and libraries to the application's class folder. The classes and libraries are copied only during debug builds, the same applies to the bytecode modifications done by Javassist.

3.4.3. Lifecycle dimension

Android already natively deals with the problem of replacing resources at runtime – one can consider the case where there is a XML layout file in both *layout* and *layout-land* subfolders. In the case where the device is in the portrait mode and user rotates the device to landscape the framework needs to replace the inflated layout from *res/layout* folder with the one from *res/layout-land*. It is not a straightforward task, however. The application code can make changes to the view tree at runtime and such changes would be lost if the layout was simply reinflated.

Android resolves this by adding a lifecycle dimension to the activity component. Activities can be destroyed and recreated at will if the device encounters a configuration change. The persistence of the activity state is up to the developer - the framework provides facilities to serialize the state into a simple map where keys are strings and values are primitives. This map is implemented in the *Bundle* class and is passed on to the new instance of the activity.

Catalyst client is based on the same paradigm - the updated resources are retrieved and loaded at activity instantiation, entangling Catalyst client lifecycle with activity's lifecycle. Usually this means that the developer has to initiate a configuration change to see the updated resources, but this can be worked around by requesting a different orientation from the code and then restoring the orientation when a new activity with the updated configuration is instantiated. This trades performance, as an extra activity create-destroy cycle is required, for increased developer convenience.

3.5. Testing

As Catalyst replaces the *Resources* class during development phase it is critical that the new class behaves exactly as the native one. Since the number of possible qualifier combinations too large for manual ad-hoc testing an automatic testing system on top of standard Android JUnit framework was implemented. The test is exhaustive and follows the following algorithm:

1. Discover the list of all resource ID's from the R class via reflection
2. For every resource:
    1. load the resource using the standard API
    2. load the resource using Catalyst
    3. deep-compare the two resources and pass the test if no inconsistencies are found

This test was successfully executed on a standard Hello World project, the ApiDemos sample project and a small number of real-world Android projects provided by Mobi Solutions OÜ, a small mobile software development company based in Tartu.

3.6. User Interface

Catalyst provides three different user interfaces for the developer: a command line interface for integration with automated tools, mainly used by testing framework, an Apache Ant tasks that interface with the standard Android Ant build tasks for developers who use Ant, and an Eclipse plugin to make Catalyst-enabled builds within Eclipse.

The command line tool provides two separate commands. One command is for processing an Android application to add Catalyst classes and modify the *Activity* classes mid-build, after the javac compiler has produced the Java bytecode and before the *dx* tool has converted it to Dalvik bytecode. The other command is for running the Catalyst server. The tool takes the Android SDK folder path and the application path as an argument. The ant tool is a simple convenience wrapper around the command-line tool – Android Ant build files already provide *post-compile* and *post-build* targets which are overridden with the invocations of the appropriate command-line tools.

The Eclipse plugin introduces a new project nature called *CatalystNature* to any Android projects. This nature can be enabled and disabled at will for any Android projects by opening the project's context menu via right click and selecting either "Enable Catalyst" or "Disable Catalyst" from the submenu named "Catalyst". Only one of the items is displayed at once, depending if the project has the Catalyst project nature or not. Project natures indicate the build commands that are used when building a given project, for instance an Android project has both the *JavaNature* and *AndroidNature* project natures. *CatalystNature* adds a custom build command that runs the aforementioned command-line tools if and only if the developer triggered a debug build.

3.7. Build and Turnaround Times with Catalyst

Two different metrics were measured – build times to estimate the Catalyst processing and server startup overhead and the time it took from a string change to observing the result in the emulator. The ApiDemos was picked for the measurements as it is a good medium-to-large size project – build times are most critical for larger projects. All measurements were done with the Android emulator running on a modern developer machine with a quad-core 2.93GHz processor with 8GB of RAM. The machine was running Ubuntu 12.04 LTS and r19 version of Android SDK was used.

For building the full build time with Apache Ant was measured. It is evident from the results (Table 1) that adding Catalyst to a project adds a small overhead (roughly 6 seconds) to the project. This is the time it takes to process the *Activity* class files and to start the server.

For measuring the deployment times a common scenario was constructed:

1. Developer changes a string
2. Developer saves the *strings.xml* file
3. Developer runs the “Run as Android Application” launcher in Eclipse (omitted in Catalyst measurements)
4. Developer navigates to the screen that has the string and observers the result

As is shown in Table 2 Catalyst provides substantial gain in observing the results even with simple examples. Most of the time spent with Catalyst was for waiting the emulator to refresh the device configuration and to reinstate the activity.

Table 1. Measurement of build times

Attempt	**Standard Android Ant build**	**Ant build with Catalyst processing**
#1	29.6 sec	37.4 sec
#2	29.9 sec	36.1 sec
#3	29.4 sec	36.2 sec
#4	29.9 sec	35.4 sec

Table 2. Measurement of a common developer scenario – a string change

Attempt	**Standard Android Eclipse build**	**String change with Catalyst**
#1	22.9	5.9 sec
#2	22.6	4.8 sec
#3	22.5	4.7 sec
#4	21.4	4.2 sec

3.8. Implementation Limitations

The current implementation of Catalyst imposes certain limitations and risks:

1. Only internal components can retrieve the updated version of the resources, e.g., it is not possible to replace the launcher icon and label with Catalyst since the platform retrieves these resources directly without invoking any application runtime.

2. Themes cannot be reloaded as they typically extend the device's default theme. The latter is inaccessible to the Catalyst server as device's theme differs from the stock Android theme, e.g. Samsung has Touchwiz, HTC has Sense, etc. If one makes a change to a theme one still needs to run the full rebuild/redeploy cycle to see the results, even if using Catalyst.

3. Catalyst is built on information not present in the Android SDK documentation. For instance the internal representations of resources are undocumented. This means that a future build of Android may break Catalyst functionality. It is worth noting that such occurrences should be unlikely as Android still has to provide backwards compatibility for applications built using the older versions of the Android SDK.

4. Catalyst only provides facilities for updating resources at runtime. If one makes any code changes then one still needs to recompile and reinstall the application. Class reloading on-the-fly may be possible but is beyond the scope of this thesis.

Conclusion

The aim of this thesis was to find a way to reduce Android application development time and cost by enabling Android developers to reload application's resources at runtime without requiring them to run the full rebuild/redploy cycle every time a resource is changed.

To achieve this both the official Android SDK documentation and the Android Open Source Project source code for the latest Android version, Ice Cream Sandwich, were investigated. This provided insights into how developers can define Android resources, how they are stored internally within Android application packages and how the standard build tools bundled with the Android SDK compile the developer-defined resources into internal format.

The resulting knowledge was used to implement Catalyst, a developer tool with the sole purpose of providing resource compilation on the developer's machine, delivery to the device and facilities to use the updated version of the resource in the application instead of the bundled resource in the package. Catalyst achieved the goal of drastically reducing mandatory rebuild-redeploy cycles for developers working with Android resources and UI interfaces.

The same methods used by Catalyst may alternatively be used in production systems to achieve silent over-the-air updates of application's resources.

Androidi rakenduse ressursside dünaamiline laadimine

Bakalaureusetöö

Madis Pink

Resümee

Mobiiliplatvormil Android baseeruvaid seadmeid on erinevaid – lisaks nutitelefonidele leidub tahvelarvuteid, televiisoreid, tahvel- ja sülearvuti hübriide ja palju teisi tüüpi seadmeid. Lisaks võib ühe seadmeklassi siseselt riistvaraline konfiguratsioon suuresti erineda. Selle vastu võitlemiseks leiduvad Android-platvormi standardteegis eraldi meetodid ja klassid ressursside käitlemiseks, mis võimaldavad rakendusega kaasa pakendada erinevate konfiguratsioonide jaoks erinevaid ressursse. Pakendamise käigus transleeritakse ressursid inimloetavalt kujult, nagu näiteks XML, madalamale masinloetavale kujule. See protsess võtab aega ja tähendab rakenduse arendaja jaoks, et iga muudatusega kaasneb rakenduse taasehitamisest tulenev lisaajakulu.

Käesoleva töö eesmärk oli uurida, kas ressursside transleerimisprotsess on võimalik viia rakenduse ehitusfaasist (ing.k. *build-time*) käivitusfaasi (ing. k. *run-time*). Uurimise aluseks võeti Android-platvormi uusima versiooni, koodnimega *Ice Cream Sandwich*, lähtekood ning Google poolt avaldatud ametlik Androidi standardteegi dokumentatsioon.

Töö käigus leiti, kuidas arendajad saavad ressursse defineerida ning kuidas rakendustega kaasa pakendatud ressursse esitatakse. Samuti leiti moodus seadme peal ressursside laadimiseks üle võrgu, jättes ära kaasapakendatud ressursside laadimine.

See informatsioon oli aluseks Android-platvormi rakenduste arendajatele suunatud abivahendi nimega Catalyst loomisel, mis võimaldab teha rakenduse ressurssides muudatusi nii, et rakendust ennast uuesti ehitama ja seadmesse installima ei pea. Töö praktiliseks väljundiks ongi nimetatud abivahendi prototüübi loomine. Töös mõõdeti põgusalt keskmistele ressurssimuutmistele kulutatud aega. Tulemused näitasid standardvahenditega võrreldes mitmekordset ajavõitu.

Bibliography

[1] Gartner – Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth

http://www.gartner.com/it/page.jsp?id=1924314; retrieved 14.05.2012

[2] Android Developers – What is Android

http://developer.android.com/guide/basics/what-is-android.html; retrieved 14.05.2012

[3] Android Developers – Application Fundamentals

http://developer.android.com/guide/topics/fundamentals.html; retrieved 14.05.2012

[4] Android Tools Project Site – Build changes in revision 14; "Ant build improvements"

http://tools.android.com/recent/buildchangesinrevision14; retrieved 14.05.2012

[5] Android Developers – Building and Running

http://developer.android.com/guide/developing/building/index.html; retrieved 14.05.2012

[6] Android Developers – Providing Resources

http://developer.android.com/guide/topics/resources/providing-resources.html; retrieved 14.05.2012

[7] Android Developers – Accessing Resources

http://developer.android.com/guide/topics/resources/accessing-resources.html; retrieved 14.05.2012

[8] Android Developers – Drawable Resources

http://developer.android.com/guide/topics/resources/drawable-resource.html; retrieved 14.05.2012

[9] AOSP source code

android/view/LayoutInflater.java; Tag 4.0.3_r1

[10] AOSP source code

android/content/res/Resources.java; Tag 4.0.3_r1

[11] Android Developers – Using the Android Emulator

http://developer.android.com/guide/developing/devices/emulator.html; retrieved 14.05.2012

[12] Google Code – Google Protocol Buffers

http://code.google.com/p/protobuf/; retrieved 14.05.2012

[13] Javassist

http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/; retrieved 14.05.2012

Appendix 1

Appendix 1 is a compact disc bundled with this document. Appendix 1 contains the Catalyst source code and source code for the sample "Hello World" and ApiDemos Android applications.