AirSync

Robert Durham

April 15, 2012

1 Introduction & Motivation

Many portable devices have become a part of our everyday lives impacting work as well as our home lives and the way we travel. Most of these devices have their own unique means of transferring media and documents. All of these devices can access the internet via some means, e.g. WiFi or cellular network. I will use my situation as an example, between myself and my wife we have at least eight different devices that we like to access music libraries, documents, photos and videos from:

Device	**Description**	**Connectivity**
Desktop PC	Linux	Wired(Home)
Home Theater PC	Linux	Wired(Home)
Laptop(Mine)	Windows 7/Linux	WiFi
Laptop(Spouse)	Windows XP	WiFi
Motorola Droid 4(Mine)	Android 4 Phone	4G & WiFi
Motorola Droid 3(Spouse)	Android 2.3 Phone	3G & WiFi
Nexus 7	Android 4 Tablet	WiFi
Kindle Fire HD	E-Reader/Tablet (Android 4 based)	WiFi

Several 'Cloud' services have offered means to maintain and provide access to media and documents via the internet, e.g. Apple iCloud, DropBox, Asus WebStorage, Ubuntu One. All of these have limited storage, cost money to expand that storage, and don't support all necessary devices. These issues are outside of the fact that these require trusting your personal data to a 3^{rd} party. DropBox and Ubuntu One support almost all of these devices, but transferring large media files through internet is slow. What if there were a solution to easily synchronize media/files through your home network, yet

still allow access to these files through the internet without relying on the privacy policy of a 3^{rd} party provider? Some open source solutions can help in desktop/laptop situations, but not with mobile devices.

2 Goal

The goal of this project is to provide a means to easily synchronize files between multiple types without relying on 3^{rd} party services to protect sensitive data. In order provide a replacement option for services like dropbox, AirSync must provide a means to access files remotely through the internet.

3 Use Cases

Their are three types of devices and two types of connections used to depict each of the use cases. The three types of devices are static PC, mobile PC and mobile device. The two types of connections are local and internet connected. A localized connection refers to the situation where the devices do not need to utilize internet bandwidth to transfer data. Local connections are typically at least 54 mb/s where as most internet connections in 2010 averaged 3.7 mb/s[1]. 54 mb/s is the theoretical limit of 802.11g wireless networking protocol. Hotels have notoriously slow internet connections.

1. Synchronize files between server and devices on local network
2. Manage Android file system from static or mobile PC on local network
3. Download/Upload files between server and Android Device or Mobile PC via internet connection

4 Design

AirSync will provide the means to synchronize files between any Windows, Linux and Android devices. It will include the following three software components:

[1]http://arstechnica.com/telecom/news/2010/01/us-broadband-still-lagging-in-speed-and-penetration.ars

Component	Description
Service	Runs as system service or daemon on server and client PCs. Acts as Server and/or Client depending configuration. *A client needs to be capable of becoming a server...*
Service Configuration UI	Graphical User Interface that will monitor status and change settings with AirSync Service on PCs. Will also provide remote management of files on Android devices.
Android Client Application	Activity App that will provide the user with a choice of two operating modes. One mode will allow remote management of the android devices memory space from the Service Configuration UI. The second mode will allow the Android user to browse the files available from the server and select files to download.

All application components will be developed in Java to maximize code re-usability and portability between platforms. *Android*[2] is essentially any embedded Linux environment with a specialized Java Virtual Machine. If Android specific code sections are properly abstracted into separate Java Classes, the large majority of the android client code will be re-usable with the PC application. The Integrated Development Environment (IDE) will be *Eclipse*[3] with the Android Developer Tools(ADT) addons. *Git*[4] will be used for source versioning control with the git server being hosted on *github.com*[5]. This project will not include the development of an AirSync application for *Apple IOS*[6]. The development environment for IOS only supports Mac OS X. The IOS development language is a managed C++, much of the code would need to be ported and I do not have access to a Mac, so I have exclude support, for now.

[2]http://www.android.com/developers/
[3]http://eclipse.org/
[4]http://git-scm.com/
[5]https://github.com/durhamrj/AirSync
[6]https://developer.apple.com/devcenter/ios/index.action

4.1 Service Design

- Server side component, provides remote access to files based on system configuration
- Tracks initialized devices and associated unique ids
- Communicate transfer files and status

4.2 Service Configuration UI Design

- Graphical interface to service for configuration and status monitoring
- Enable access to certain files and folders. Potentially on a per device basis(Access Restrictions).
- Log remote transfers

4.3 Android Client Application Design

- Provide client file synchronization with service
- Allow remote file selection for download
  - when not on localized network
  - user selectable
- Provide remote file system management to service

4.4 Source Code Organization

This section is being updated as the project progresses.

The source code and build files for this project can be found at *http://github.com/durhamrj/AirSync* The Service and Service Configuration UI share a common Eclipse/Java project. The components are separated by the package structure within this project. The folder structure of the project is as follows:

http://github.com/durhamrj/AirSync	
AirSyncServer/	Cross Platform Java application for Desktop and Laptop computers
RemoteSync/	Android application source code and project files
docs/	Contains documentation for project include latex source for this PDF
README.md	An autogenerated file created by github

4.4.1 AirSyncServer Project Structure

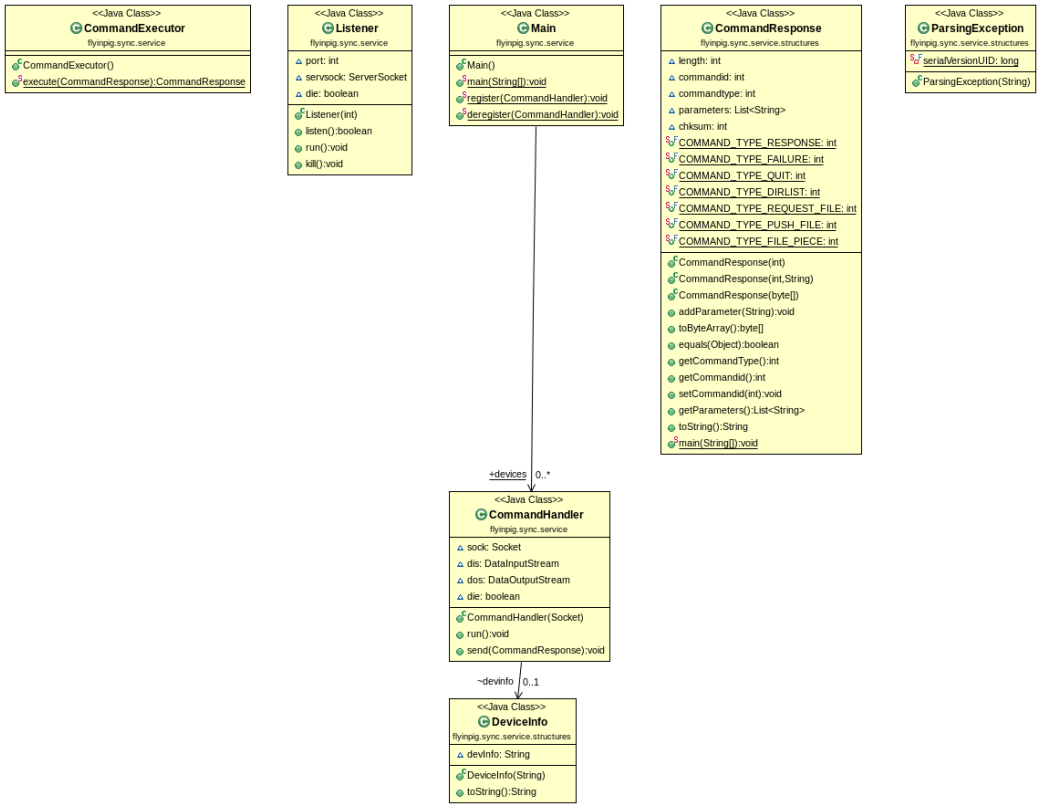

Figure 1: AirSyncServer Class Diagram

AirSyncServer Source	
flyinpig.sync.service	
CommandHandler.java	Handles socket IO with client connections as a separate thread
Listener.java	Opens listening port in a thread and spawns new CommandHandler threads for each new incoming connection. New connections are tracked for display in the UI.
Main.java	Singleton class. Includes the application main and fields/structures to track connection application state information.
flyinpig.sync.service.structures	
CommandResponse.java	Handles conversion of commands & responses between Java objects and binary data to be transmitted over Socket IO. This was initially intended to use XML serialization, but due to a lack of android support the conversion code is all in this file and uses not additional system libraries.
DeviceInfo.java	Contains specific information about a device sent on initial connection. Used to hold information displayed within the UI to uniquely identify each device.
ParsingException.java	Exception thrown by CommandResponse when errors occur parsing command & responses.

4.4.2 RemoteSync Project Structure

The file structure in the Android project is largely made of generated files. I will only identify the files that I have manually modified as well as source code I have written. Many of the source files are identical between the AirSyncServer project and the RemoteSync Android project.

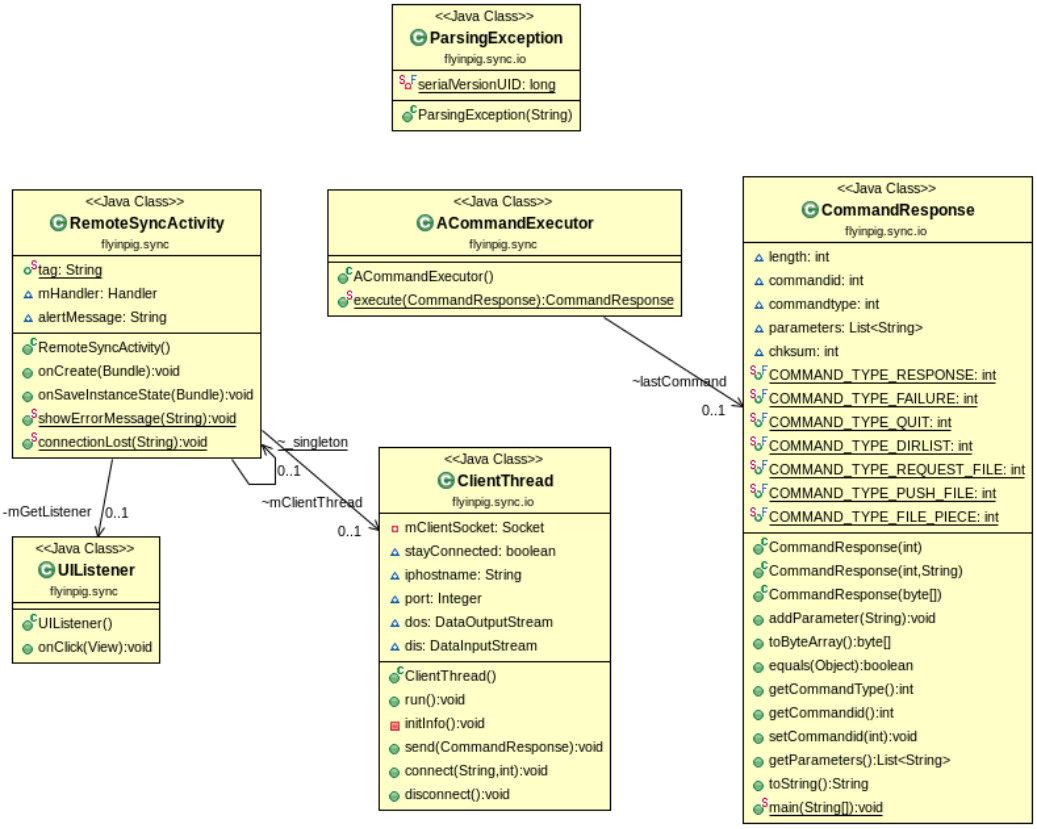

Figure 2: RemoteSync Class Diagram

RemoteSync Source	
AndroidManifest.xml	This file is most importantly used to set application permissions.
flyinpig.sync	
RemoteSyncActivity.java	Main class for application. Monitors/Controls application view and state. Views are generated with a graphical interface in the android developement kit.
UIListener.java	OnClickListener for Views.
flyinpig.sync.io	
ClientThread.java	Handles Client Socket IO. Similar to CommandHandler but more specific to android threading and IO paradigm.

5 Using AirSync

Two files are required to run AirSync.

apk jar

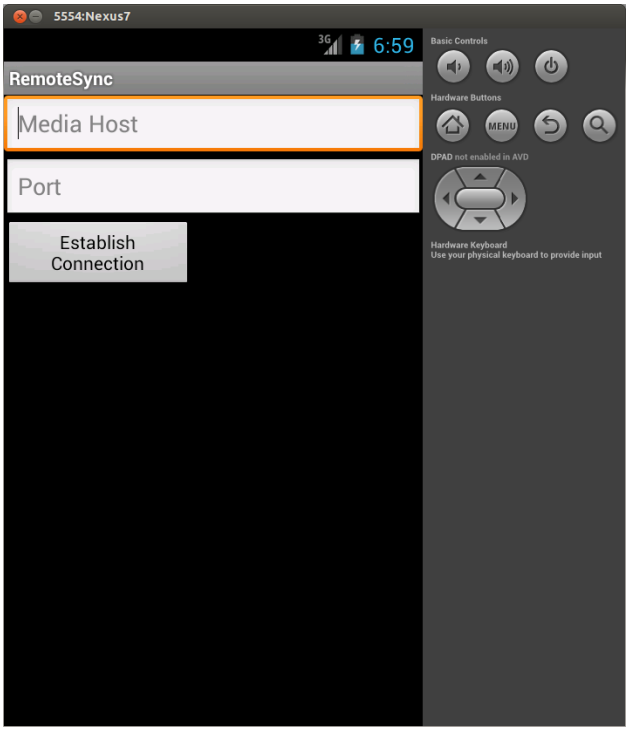

Figure 3: RemoteSync Initial View

6 Current State

The project can be demonstrated to meet use cases 2 and 3. Use case 1 has not been implemented because it will require an additional android project to run as a service and allow continuous synchronization. This use case is far more complicated than I original expected, not only because of the addition project / application required but that it also requires continuous monitoring of the mobile devices power and networking state. Also, services cannot provide user interaction, the interaction needs to occur through an Activity. All configuration and status monitoring will also need to occur

through an Activity. In the Lessons Learned section I describe some specific pitfalls that can occur with a inexperienced android developer. This Activity / Service paradigm along with strict permissions is an example as well. I believe it would require double the effort already expended to develop the proper service and activity interactions to meet the desired usability.

6.1 Desired Additional Features

Besides the use case described above there are some features that have not been implemented in the project.

- **SharedPreferences:** Applications have a defined method for save instance state information when an application loses focus, but this is intended for running information. If an application is "Force Closed", by a user, the saved instance information will no longer exist. SharedPreferences provides a mechanism to store configuration information in a persisted Hash Table using key/value pairs. Similar to how the Windows Registry works the values can be stored as Boolean, String, or Integer values. Windows, unlike Android, also allows binary data and does not restrict access by other applications.

- **MediaStore:** Android provides specifc folder structures and tracking mechanisms for media files. The folder structure and location can vary from device to device. Using MediaStore provides a mechanism to manage media files and playlists from an application while abstracting the file system structure. The current implementation lets the user browse the file structure and select the media directories manually.

- **Batch File Transfer:** Currently files can be transferred one at a time, but the user must wait for a file to transfer before queueing more file transfers. A queued transfer list and progress display would solve this problem.

- **Access Control:** In order to prevent any device or any person from connecting to any server and accessing/manipulating files the software needs to implement user and/or device based access control. This feature should implement the Bell-LaPadula confidentiality focused access control model.

- **Encrypted Communication:** Encrypting communications channels is necessary to protect the privacy of the files in transit across a network, whether wireless or wired. SSL is a very common standard implementation to enable encryption in most applications. SSL can be configured to utilize AES 256 bit encryption, which is NIST FIPS compliant.[7]

- **Historical Versioning:** Historical versioning would be useful in recovering overwritten files. The most robust way to implement this feature is to incorporate a version control library for the server portion of the project.[8] Subversion is widely implemented, and most suited for this task. The merge functionalities of subversion will need to be disabled for this feature.

7 Lessons Learned

During the development of this project I focused primarily on the android application. Prior to this project I had no android development experience. There are several things that anyone should be aware of when starting an android project. I am experienced with Java and therefore have fewer lessons learned with the Java development portion of the project.

- Apps don't shutdown or close in a traditional sense. As the applications developer you must save/restore application state information. Anytime your application is not focused the operating system may choose to "close" your app and re-initialize it when it regains focus. The developer is responsible for handling the state information properly.

- Android has permissions that will prevent an application from accessing certain system calls. The permissions that your application use must be specified in the AndroidManifest.xml in the base directory of your Android project. You can edit this file manually or use the graphical editor provided in the ADT.

- Android does not allow some SystemIO calls to occur from the primary thread of an Activity. I ran into this problem with Socket IO.

[7] http://src.nist.gov/publications/fips/fips197/fips-197.pdf
[8] http://svnkit.com/

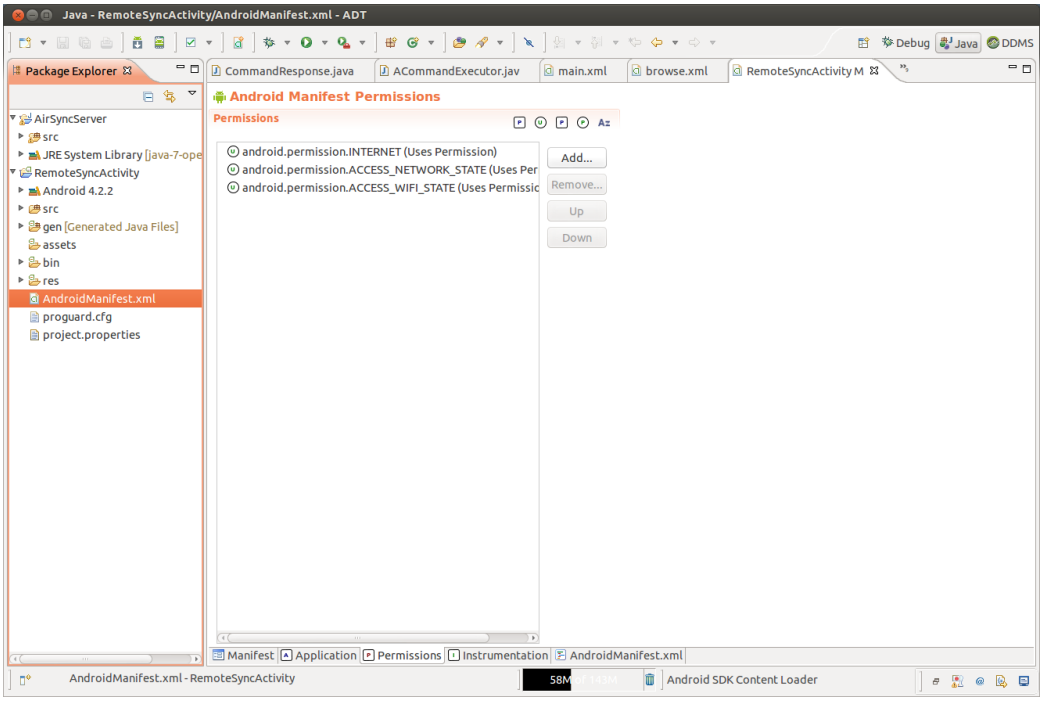

Figure 4: Android Activity Permissions Editor

The non-descriptive RuntimeException below was being thrown whenever I tried. I was unable to find documentation of this on androids development site, but with trial and error discovered that it worked fine in a separate thread. My assumption as that Android does this to avoid causing the interface to hang or lock up while waiting for SystemIO.

- Android also uses XML files to define graphical user interface(GUI) layout. In Java GUIs are defined in the source code, which is typically defining buttons,text labels, fields, etc. and there locations. By moving this definition to XML files called Views or Layouts it significantly cleans up the associated code. In this project I only need to attached Listeners to the Views that I created. I only used the onClick Listener for this project. View object likes buttons are clickable by default, but other view objects like TextView (The equivalent to a JLabel in java) must be specified as follows within its XML definition.

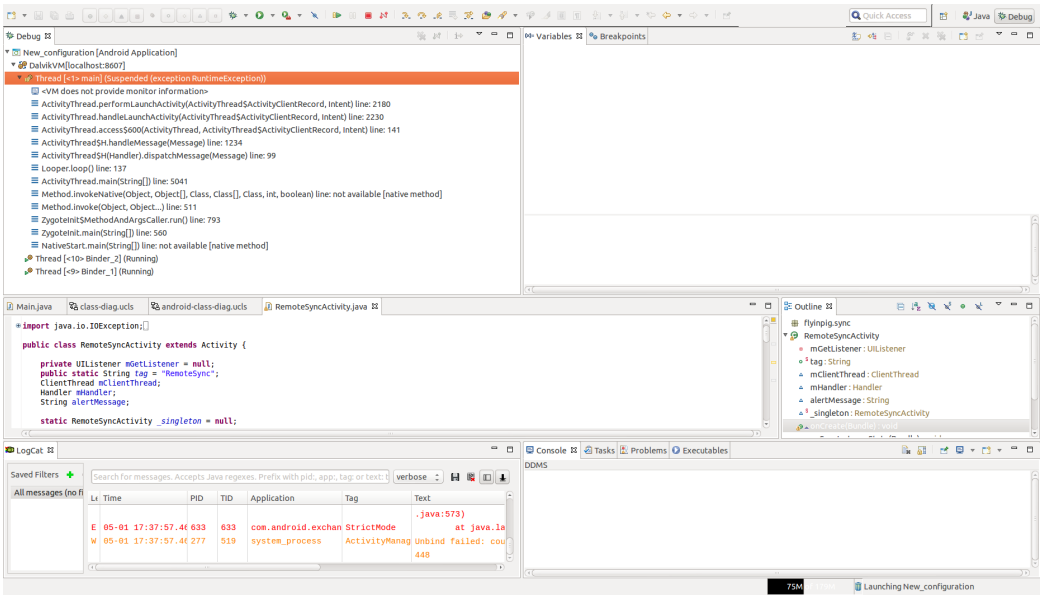

Figure 5: Non-descriptive RuntimeException

android:onClick="onClick"
android:clickable="true"

- The ADT provides a set of emulators that can be configured to matched the desired device specifications and Android versions. This provides an simple method for a developer to try an application on a broad set of configurations. During my development I noticed that system calls that collected information about the device did not behave consistently in the emulator and was causing errors. I was unable to isolate the exact problem, but the same code that failed in the emulator worked well on both physical devices I tried. Some forum posts I read indicated other developers were having similar finicky issues with the emulated devices.

Android applications are strictly structured as *Activities* or *Services*. I developed this application as an activity, which provides the user interaction performs actions when the the application is active.

Although the android development libraries are very similar to the standard Oracle Java Runtime libraries, I found a few discrepencies in how Oracle

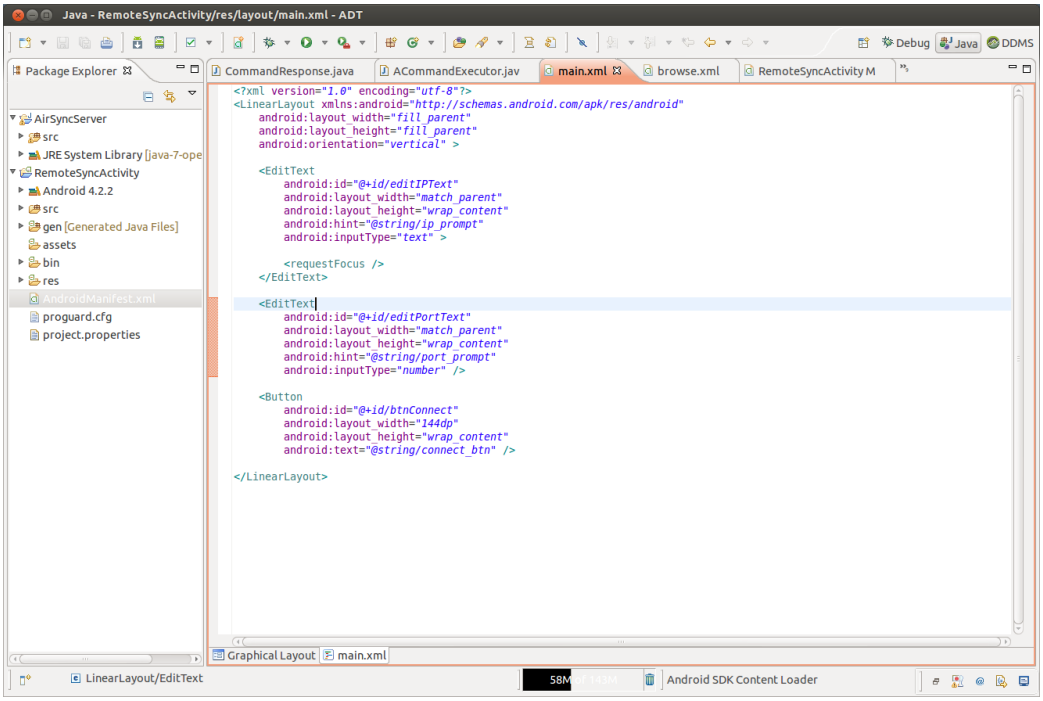

Figure 6: Android View/Layout XML

Java libraries when compared to Android. One in particular is related to Input/Output. DataInputStream and DataOutputStream functions to read and write UTF Strings to the network do not seem to be compatible when attempting to use the Android and Oracle versions together. In this case I wrote compatible replacement code using readChar() as a work-around.

8 Conclusion

I intend to continue developing this project for personal use and with the intent to share the source code and allow open development. With the adition of access control restrictions and encrypted communication this is the a document and media sharing solution that can provide corporations and individuals with the benefits of cloud sharing without the privacy concerns. Where corporations can own the assets and allow employees to access work documents on mobile devices securely.