AirSync

Robert Durham
`durhamrj@gmail.com`

Master's Project
Syracuse University

May 7, 2013

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

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

AirSyncServer source is an Eclipse Java project. Eclipse is also used for the Android Development Toolkit(ADT) and was the logical choice for an IDE for this project.

Lines of Code by file:
76 ./flyinpig/sync/service/Listener.java
297 ./flyinpig/sync/service/Main.java
17 ./flyinpig/sync/service/structures/DeviceInfo.java
14 ./flyinpig/sync/service/structures/ParsingException.java
224 ./flyinpig/sync/service/structures/CommandResponse.java
6 ./flyinpig/sync/service/structures/Config.java
148 ./flyinpig/sync/service/CommandHandler.java
293 ./flyinpig/sync/service/CommandExecutor.java
1075 total

AirSyncServer Source	
flyinpig.sync.service	
CommandExecutor.java	Handles CommandResponse objects and executes the associated commands. CommandExecutor is a singleton but uses state information from CommandHandler as needed for some functions.
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 CommandHandlers are registered with Main for display in command interface and UI in the future.
Main.java	Singleton class. Includes the application main and fields/structures to track connection application state information and commandline interface implementation.
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 is all manually coded 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.
Config.java	Class used for XML serialization of AirSyncServer configuration.

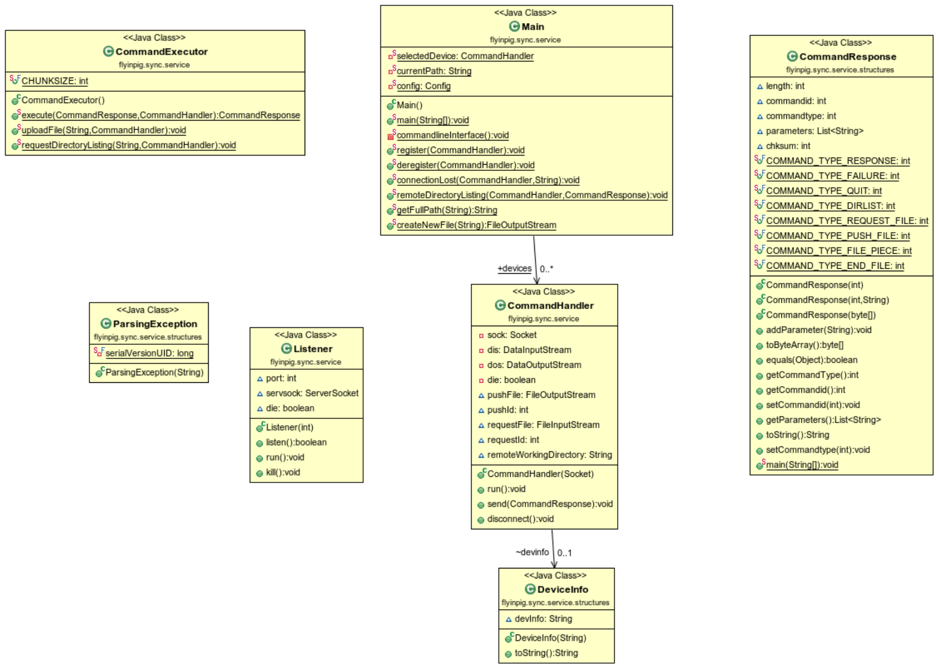


Figure 1: AirSyncServer Class Diagram

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 AirSync-Server project and the RemoteSync[7] Android project.

Lines of Code by file:

255 ./flyinpig/sync/RemoteSyncActivity.java
261 ./flyinpig/sync/ACommandExecutor.java
14 ./flyinpig/sync/io/ParsingException.java
199 ./flyinpig/sync/io/ClientThread.java
224 ./flyinpig/sync/io/CommandResponse.java
953 total

[7]AirSync name was taken already for Android Market

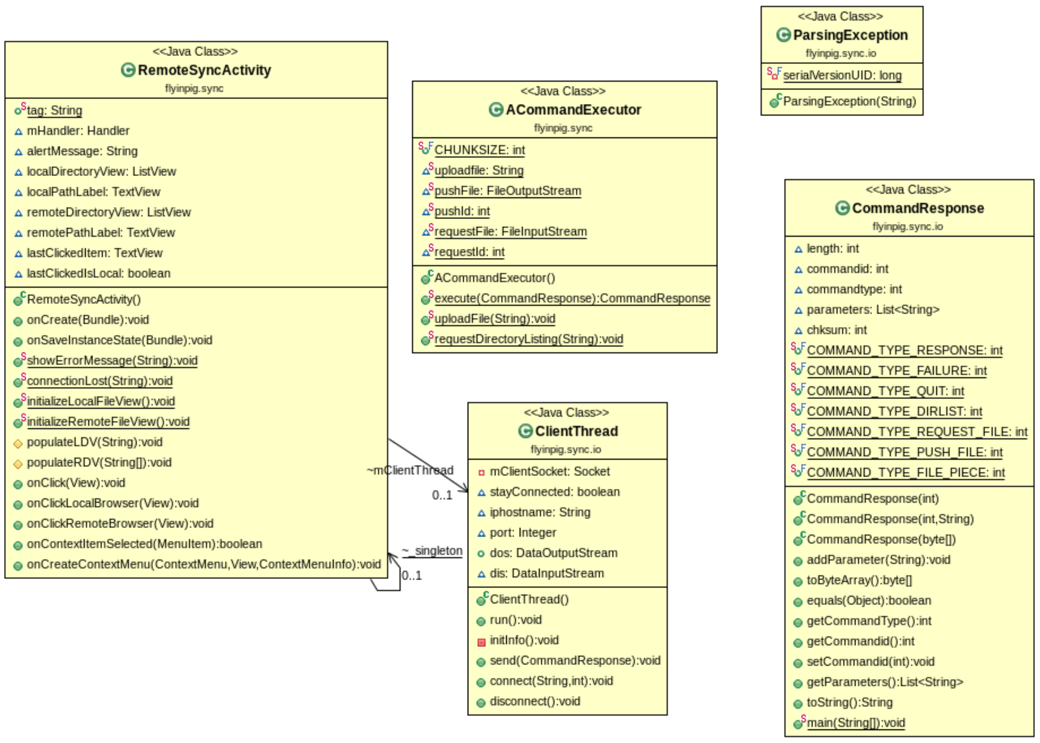


Figure 2: RemoteSync Class Diagram

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.
ACommandExecutor.java	Handles CommandResponse objects and executes the associated commands. Acts as an abstraction Layer between CommandHandlers networkIO and the UI components.
flyinpig.sync.io	
ClientThread.java	Handles Client Socket IO. Similar to CommandHandler but more specific to android threading and IO paradigm.
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 is all manually coded in this file and uses not use additional system libraries.
ParsingException.java	Exception thrown by CommandResponse when errors occur parsing command & responses.

5 Using AirSync

Two files are required to run AirSync.

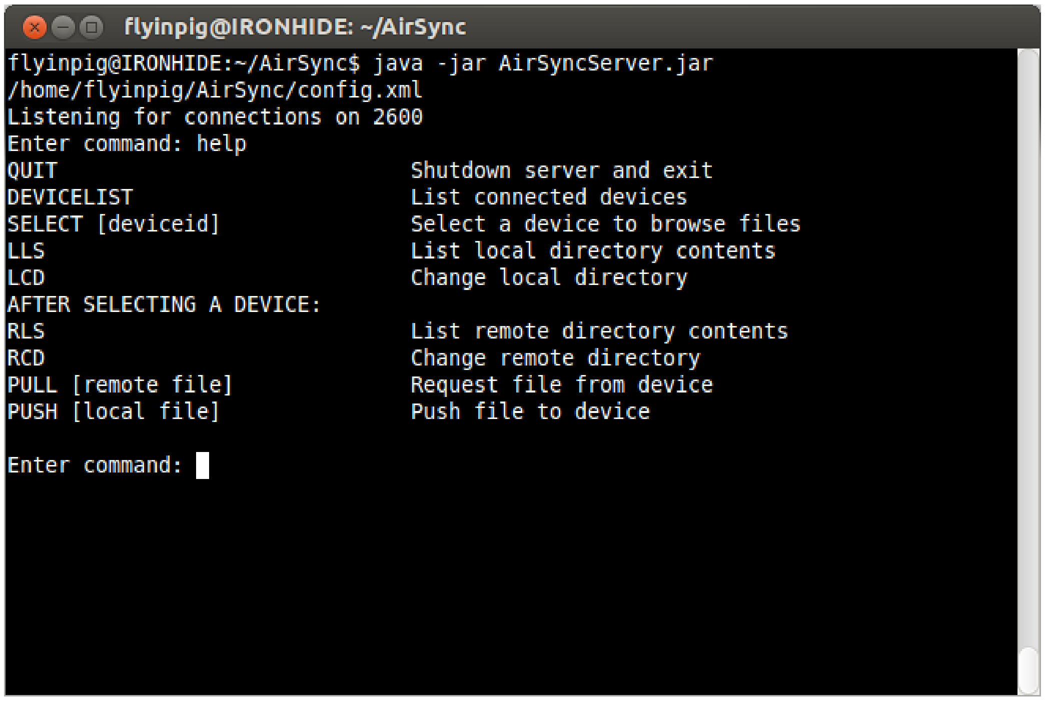
- **RemoteSyncActivity.apk** Android applicaton. This can be copied on any android device v2 or newer and installed. Acts as a client connecting to the AirSyncServer allowing for management and transfer of files on both the android device and the server.

- **AirSyncServer.jar** This jar should be executable on any system with a properly installed Java Runtime Environment(JRE). I have tested this jar with both, openjre v7 and Oracle JRE v7. Though, I believe this jar will execute properly with JRE version on 6 as well, although it has not been tested.

5.1 AirSyncServer

To run the AirSyncServer, copy the .jar file to the system you wish to run it on and ensure that a version of the java runtime is properly installed. Double clicking the jar file will execute the jar and a command prompt will open with the jar running. The jar file requires a file in the same directory *config.xml*. Below is once example with a relative folder path. The server prevents uploading or downloading of any file outside of this directory. If the path doesn't not exist, the server attempts to create it.

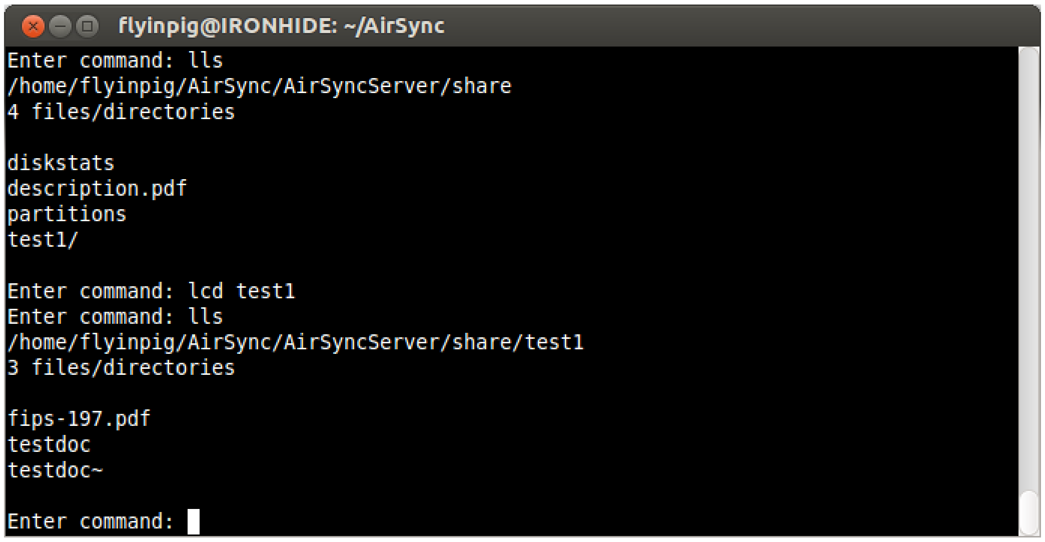
```
<config>
  <port>2600</port>
  <rootpath>AirSyncServer/share</rootpath>
</config>
```



```
flyinpig@IRONHIDE:~/AirSync$ java -jar AirSyncServer.jar
/home/flyinpig/AirSync/config.xml
Listening for connections on 2600
Enter command: help
QUIT                            Shutdown server and exit
DEVICELIST                      List connected devices
SELECT [deviceid]               Select a device to browse files
LLS                             List local directory contents
LCD                             Change local directory
AFTER SELECTING A DEVICE:
RLS                             List remote directory contents
RCD                             Change remote directory
PULL [remote file]              Request file from device
PUSH [local file]               Push file to device

Enter command: 
```

Figure 3: Starting AirSyncServer and Help Menu

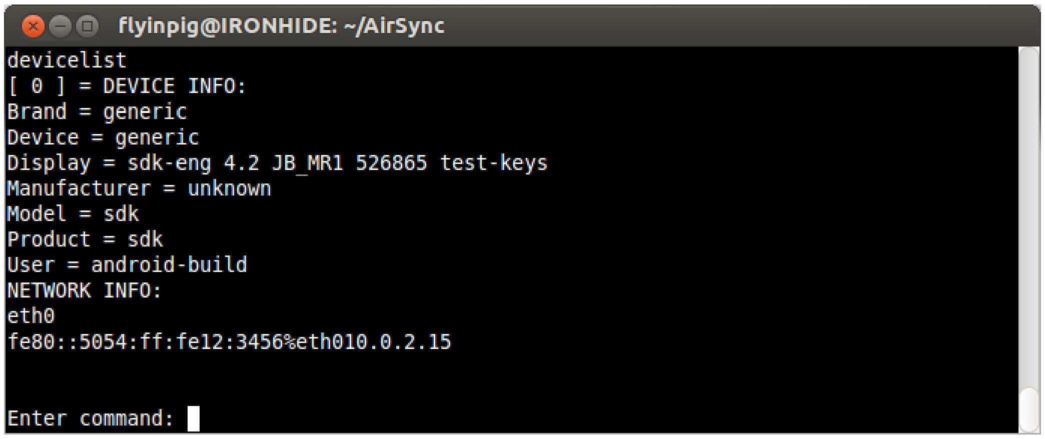
Without any devices connected, through the commandline interface you can browse the local directory structure and see what files and folders will be visible to connected applications. In the screenshot below is an example of how folders and subfolders look in the display. *diskstats* and *partitions* are two files that were transferred from the Android emulator */proc* folder while testing the application.



```
Enter command: lls
/home/flyinpig/AirSync/AirSyncServer/share
4 files/directories

diskstats
description.pdf
partitions
test1/

Enter command: lcd test1
Enter command: lls
/home/flyinpig/AirSync/AirSyncServer/share/test1
3 files/directories

fips-197.pdf
testdoc
testdoc~

Enter command: 
```

Figure 4: Local Browsing

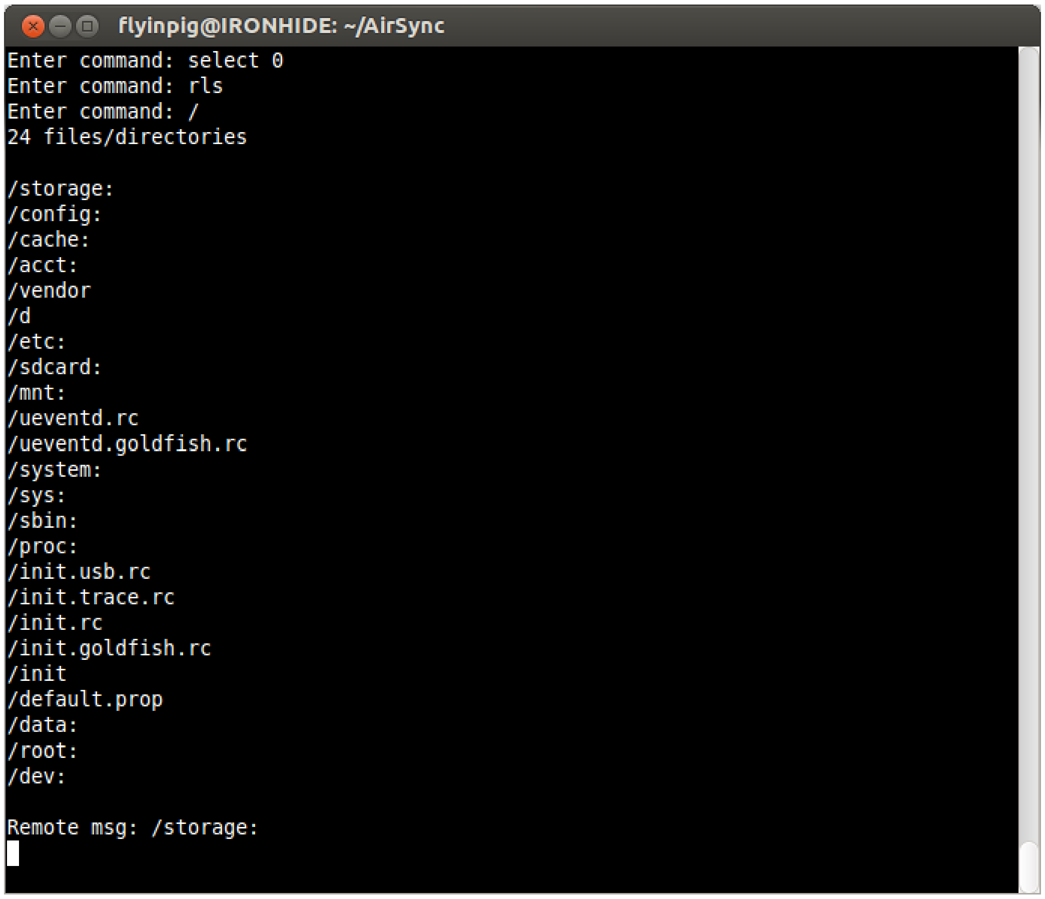
Once AirSyncServer has been launched client devices can connect to it. In the following screenshot the emulator has been connected and device information is displayed when the *devicelist* command is entered.



```
devicelist
[ 0 ] = DEVICE INFO:
Brand = generic
Device = generic
Display = sdk-eng 4.2 JB_MR1 526865 test-keys
Manufacturer = unknown
Model = sdk
Product = sdk
User = android-build
NETWORK INFO:
eth0
fe80::5054:ff:fe12:3456%eth010.0.2.15


Enter command: 
```

Figure 5: Device List

At this point the connected device can remotely upload and download files from the shared directory. The Android App section will walk through those steps. In order to browse the files on the connected device, the device must first be selected through the commandline interface. In the screenshot above there is only one device connected so this is device 0. In the following screenshot, entering *select 0* will select the connected to device in order to interact with it. By executing the *rls* command the root folder listing is retrieved for the device.



```
flyinpig@IRONHIDE: ~/AirSync
Enter command: select 0
Enter command: rls
Enter command: /
24 files/directories

/storage:
/config:
/cache:
/acct:
/vendor
/d
/etc:
/sdcard:
/mnt:
/ueventd.rc
/ueventd.goldfish.rc
/system:
/sys:
/sbin:
/proc:
/init.usb.rc
/init.trace.rc
/init.rc
/init.goldfish.rc
/init
/default.prop
/data:
/root:
/dev:

Remote msg: /storage:
```

Figure 6: Remote Folder Listing

Changing directories on the remote device is performed by issuing the *rcd* command. Many of the root system files and directories on android devices have restricted access. Few folders will allow read/write access at the root level and these are not present on the emulator. The *pull* and *push* commands are used to request files from the remote device and push files to the remote device. At this stage there are no progress indicators for file transfers.

5.2 RemoteSync Android App

In order to install the user must go into the devices system settings. Typically, varying with device vendor, in either Security or Application settings there is an option to allow installation of applications of unknown source. This means that you are allowing the device to install applications that did not come from Google's android market or 'play store'.

Once the application has been installed, looking in the application tray on your device you should find the RemoteSync application. If it doesn't not appear in the application then the application was not properly installed. The following screen will appear on your device. All of these screenshots were taken within the android development emulator and the controls on the right hand side of the device screen can be disregarded.

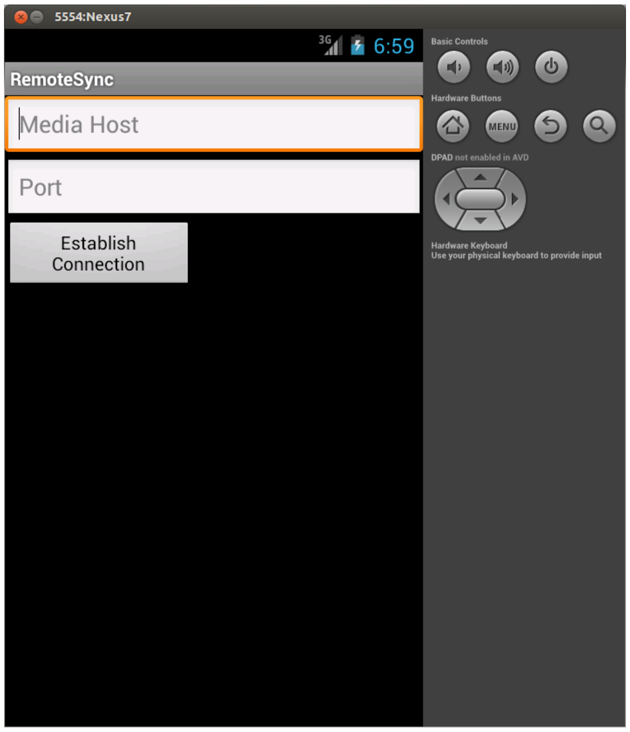
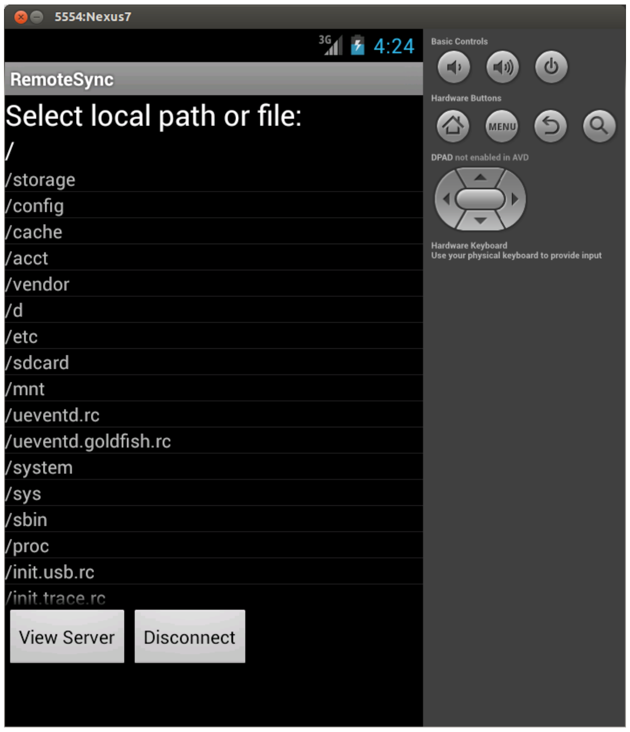


Figure 7: RemoteSync Initial View

In the first field 'Media Host' fill in the IP or host URL where your the server application is running. In the second field, fill in the tcp port that the server is configured to run on. Tap 'Establish Connection' to connect to the server. If the application fails to connect check your networking configuration, server firewall setting s and try again. Once the connection has been established the screen will change to a local device browser.



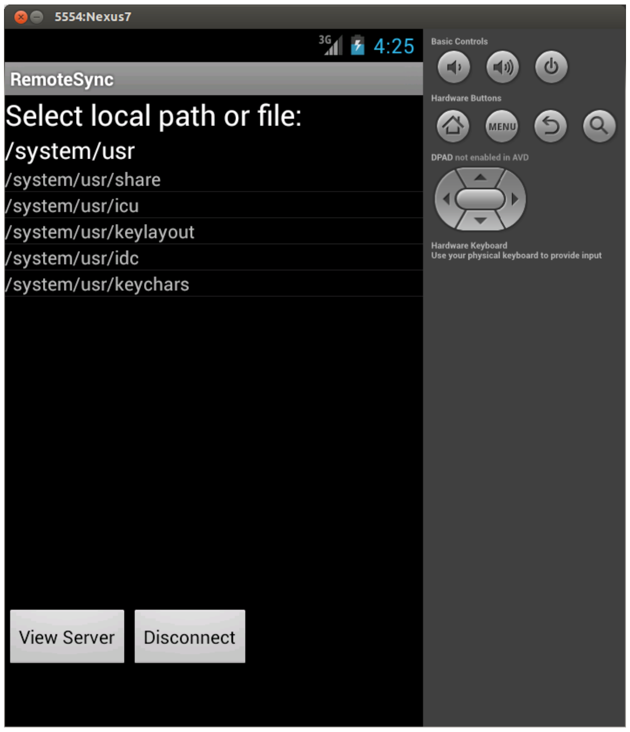
Select local path or file:

/

/storage
/config
/cache
/acct
/vendor
/d
/etc
/sdcard
/mnt
/ueventd.rc
/ueventd.goldfish.rc
/system
/sys
/sbin
/proc
/init.usb.rc
/init.trace.rc

Figure 8: RemoteSync Device File Browser

Tapping a folder will browse into that folder. Tapping a file will bring up a menu prompting to upload that file to the server. The upload will push the file the folder that the server is navigated to. On initial connection that folder is the root folder that the server is configured to allow access to. Tapping 'View Server' will switch to a remote view of the servers file system. Tapping 'Disconnect' will close the connection to the server and change the display back to the initial view. Tapping on the larger text that displays the current path that the browser is listing will browse to the parent folder.



Select local path or file:
/system/usr
/system/usr/share
/system/usr/icu
/system/usr/keylayout
/system/usr/idc
/system/usr/keychars

Figure 9: RemoteSync Browser Subfolders

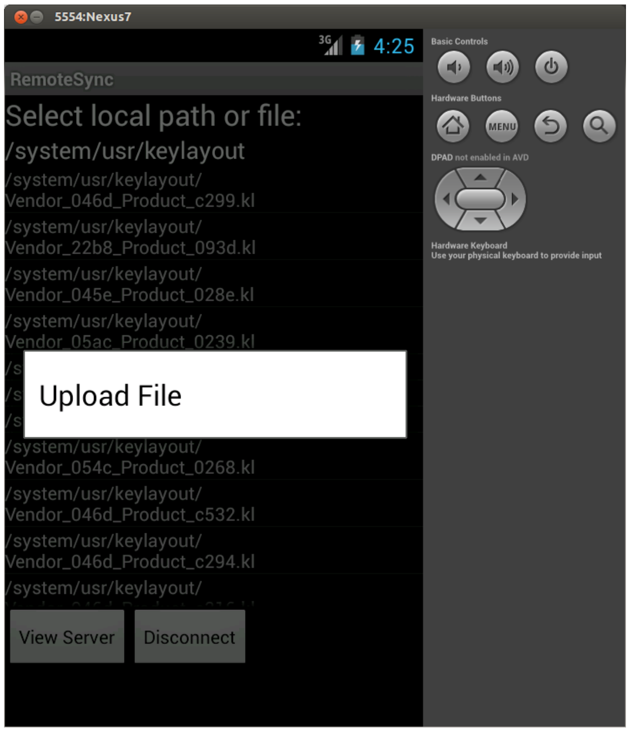
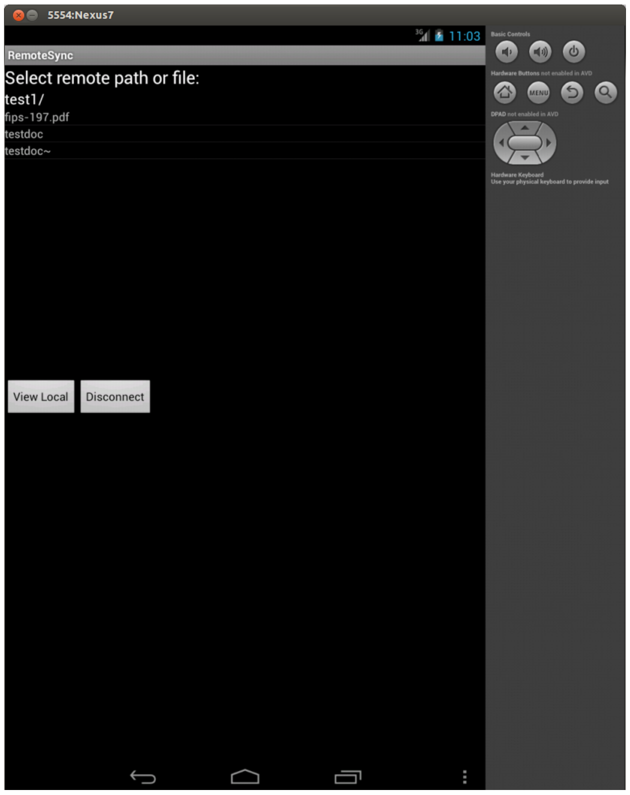

5554:Nexus7
4:25
RemoteSync
Select local path or file:
/system/usr/keylayout
/system/usr/keylayout/Vendor_046d_Product_c299.kl
/system/usr/keylayout/Vendor_22b8_Product_093d.kl
/system/usr/keylayout/Vendor_045e_Product_028e.kl
/system/usr/keylayout/Vendor_05ac_Product_0239.kl
Upload File
/system/usr/keylayout/Vendor_054c_Product_0268.kl
/system/usr/keylayout/Vendor_046d_Product_c532.kl
/system/usr/keylayout/Vendor_046d_Product_c294.kl
/system/usr/keylayout/
View Server
Disconnect
Basic Controls
Hardware Buttons
MENU
DPAD not enabled in AVD
Hardware Keyboard
Use your physical keyboard to provide input

Figure 10: RemoteSync Upload File Prompt

Below the Server Browser display list the available folder contents on the connected server. Similar to the local browser listing, tapping a folder will browse into that folder and tapping file will prompt to Download the file.



Select remote path or file:
test1/
fips-197.pdf
testdoc
testdoc~

Figure 11: RemoteSync Server Browser

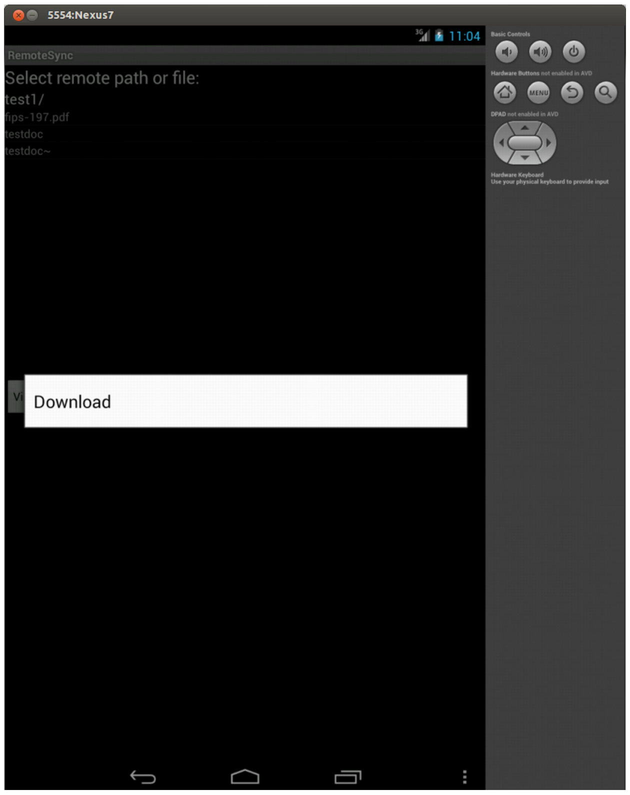
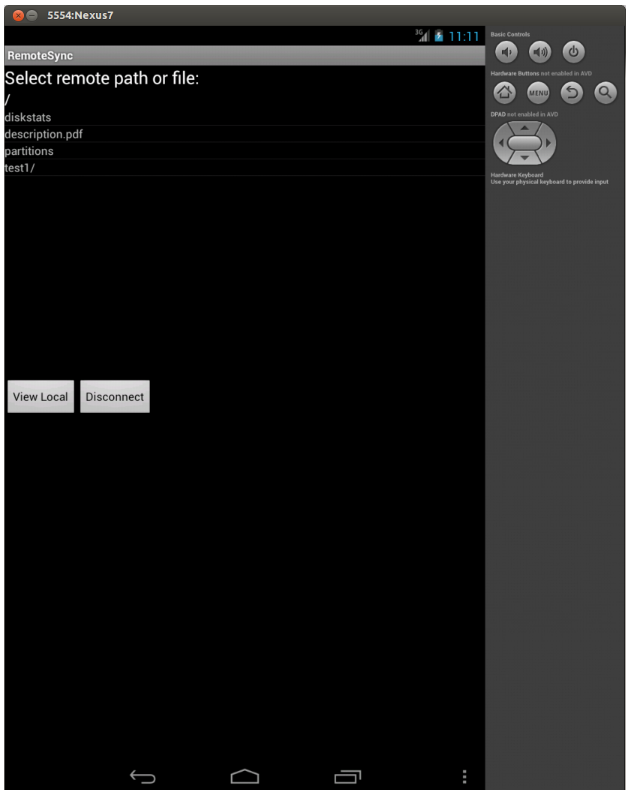


Figure 12: RemoteSync Download Menu

The following caption shows two files from the android device that have been upload from the */proc* folder, *partitions* and *diskstats*. This is one of the few readable folders on the emulator. The files in this folder contain system information similar to any UNIX system, as android as a UNIX based embedded operating system.



RemoteSync

Select remote path or file:

/
diskstats
description.pdf
partitions
test1/

Figure 13: RemoteSync After Uploading Files

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.

In order to ensure the backend transfer code was stable and working well I have put off the user interface for the desktop version. My second reason for putting off the UI is that the publicly addressable server I use as an intermediary to transfer files doesn't not have a graphical interface. It is only accessible through SSH. I have been using this server for the majority of my testing and would be unable to test the UI on this system. All of the screen shots of shown in the 'Using AirSync' section are of the commandline interface I created to perform utilize all of the components. I intend to complete the UI component before releasing AirSync to the public. The commandline option will still be usable through a '-nogui' commandline flag when executing the jar.

6.1 Desired Additional Features

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

- **Visual Indicators & Progress Bars:** Looking at it initially it would seem that adding progress bars to show on the android what is occuring. This will give the user confidence in knowing what exactly the application is doing and that a file transfer is actually happening and how long it will take.

- **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.[8]

- **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.[9] 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

Before diving into the project development I want to discuss the documentation. This is the first time I have used LaTeXfor documentation. I have very impressed by the versatility and consistency. Any time that I have lost learning its nuances while generating this documentation has likely made up for the time that I would have spent in Microsoft Word or OpenOffice trying to perfect the documents formatting. It allows you to focus on writing the

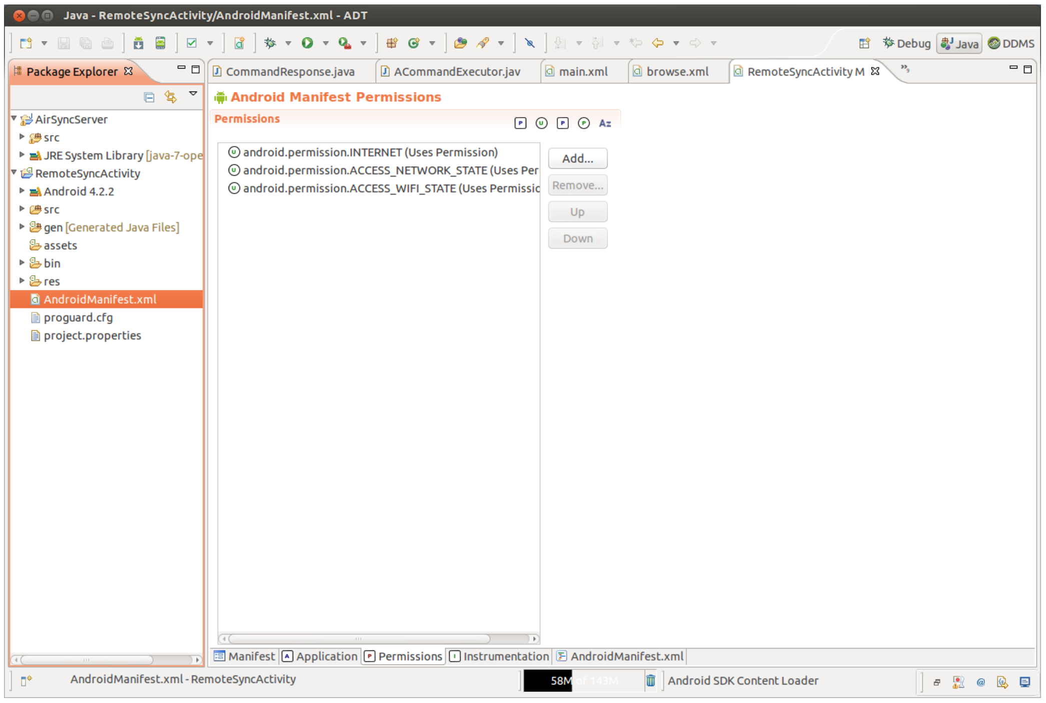
[8]http://src.nist.gov/publications/fips/fips197/fips-197.pdf
[9]http://svnkit.com/

text and generating the content without being distracted by issues like inconsistent spacing, images that are centered quite correctly. Once you have your format and layout set in latex, then it is set. Adding text, figures, lists, tables do not affect the layout before or after your additions. I have spent seven years working for DoD generating documents so large that many times we had to divide sections into separate documents because they would cause Word to crash. The largest of which was in excess of 700 pages. Having become familiar with LaTeXI have begun negotiating its use with my DoD program management team.

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 do not 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.

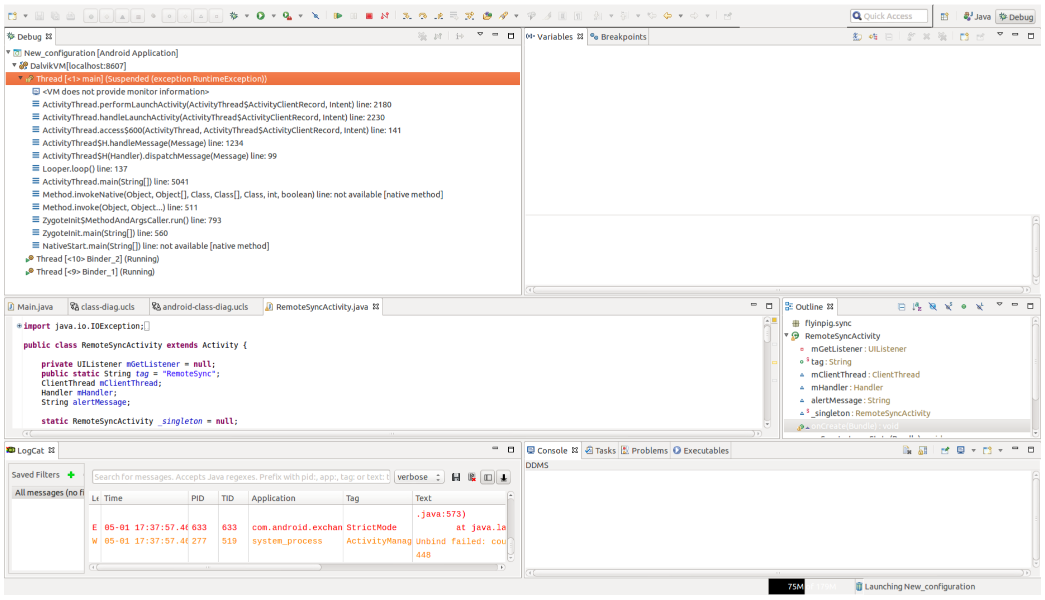
- Android has permissions that will prevent an application from accessing certain system calls. The permissions that your application use must be specified in the AndroidManifest.xml in the base directory of your Android project. You can edit this file manually or use the graphical editor provided in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



android.permission.INTERNET (Uses Permission)
android.permission.ACCESS_NETWORK_STATE (Uses Per
android.permission.ACCESS_WIFI_STATE (Uses Permissic

Figure 14: Android Activity Permissions Editor

- Android does not allow some SystemIO calls to occur from the primary thread of an Activity. I ran into this problem with Socket IO. Similarly, when attempting to change the current application view from 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.



```
New_configuration [Android Application]
  DalvikVM[localhost:8607]
    Thread [<1> main] (Suspended (exception RuntimeException))
      <VM does not provide monitor information>
      ActivityThread.performLaunchActivity(ActivityThread$ActivityClientRecord, Intent) line: 2180
      ActivityThread.handleLaunchActivity(ActivityThread$ActivityClientRecord, Intent) line: 2230
      ActivityThread.access$600(ActivityThread, ActivityThread$ActivityClientRecord, Intent) line: 141
      ActivityThread$H.handleMessage(Message) line: 1234
      ActivityThread$H(Handler).dispatchMessage(Message) line: 99
      Looper.loop() line: 137
      ActivityThread.main(String[]) line: 5041
      Method.invokeNative(Object, Object[], Class, Class[], Class, int, boolean) line: not available [native method]
      Method.invoke(Object, Object...) line: 511
      ZygoteInit$MethodAndArgsCaller.run() line: 793
      ZygoteInit.main(String[]) line: 560
      NativeStart.main(String[]) line: not available [native method]
    Thread [<10> Binder_2] (Running)
    Thread [<9> Binder_1] (Running)
```

```java
import java.io.IOException;

public class RemoteSyncActivity extends Activity {

    private UIListener mGetListener = null;
    public static String tag = "RemoteSync";
    ClientThread mClientThread;
    Handler mHandler;
    String alertMessage;

    static RemoteSyncActivity _singleton = null;
```

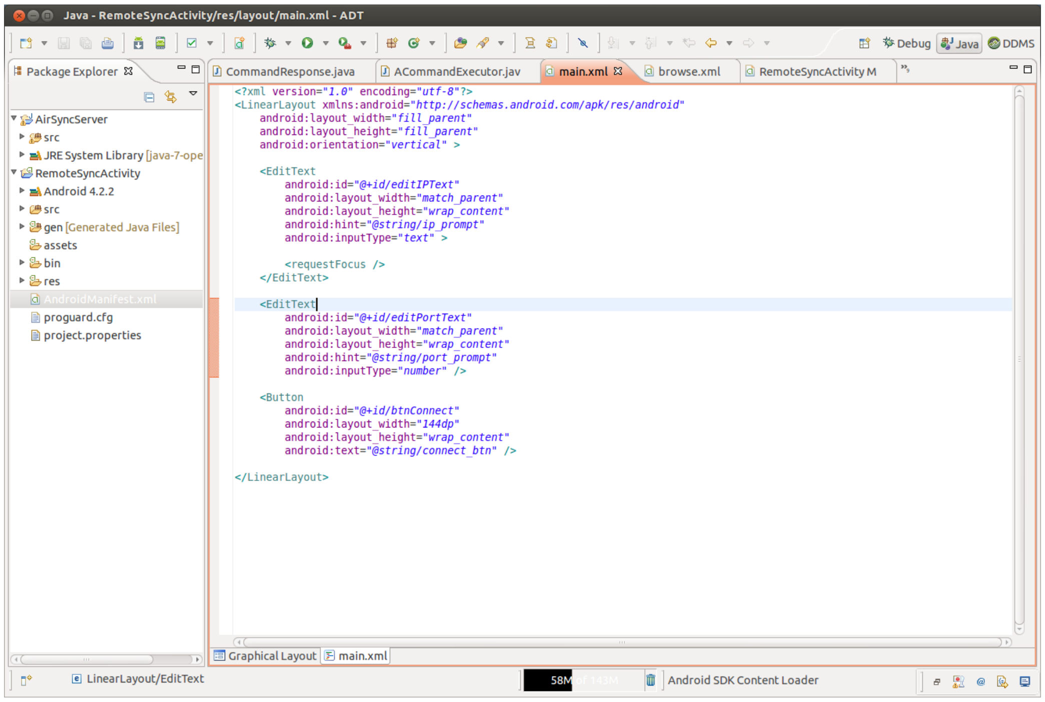
```
flyinpig.sync
  RemoteSyncActivity
    mGetListener : UIListener
    tag : String
    mClientThread : ClientThread
    mHandler : Handler
    alertMessage : String
    _singleton : RemoteSyncActivity
    onCreate(Bundle) : void
```

```
Lε Time                PID  TID  Application        Tag            Text
                                                                   .java:573)
E  05-01 17:37:57.46  633  633  com.android.exchan StrictMode            at java.la
W  05-01 17:37:57.46  277  519  system_process     ActivityManag  Unbind failed: cou
                                                                   448
```

Figure 15: Non-descriptive RuntimeException

- 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.

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



```xml
<?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" >

    <EditText
        android:id="@+id/editIPText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/ip_prompt"
        android:inputType="text" >

        <requestFocus />
    </EditText>

    <EditText
        android:id="@+id/editPortText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/port_prompt"
        android:inputType="number" />

    <Button
        android:id="@+id/btnConnect"
        android:layout_width="144dp"
        android:layout_height="wrap_content"
        android:text="@string/connect_btn" />

</LinearLayout>
```

Figure 16: Android View/Layout XML

- 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

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 hope to share the source code and allow open development. All project contents and source code are available on GitHub. I currently am not allowing anyone besides myself to edit the source. In the future with the addition of access control restrictions and encrypted communication this is a viable document and media sharing solution that can provide corporations and individuals with the benefits of cloud sharing without the privacy concerns. Corporations will be able to own and control the server assets and allow employees to access work documents on mobile devices securely. Because this projects source code is open, companies will also be able add additional security features as needed, i.e. deleting important documents within a certain time frame of download to protect it.