Using Android devices within a Jenkins Continuous Ingetration Environment

André Boddenberg
student at the Beuth University of Applied Sciences Berlin
Matr. Nr. 791997
andre@blobb.me

Abstract

Is it possible to implement Android devices into a Continuous Integration Server like Jenkins? So we would gain the possibility of running Android tests on real devices and even vendor specific Android versions instead of emulators.

Key words

androidFarm: Jenkins slave instance which acts like a master for all androidSlaves.

androidSlave: Android device connected to the androidFarm via the android debug bridge.

1. Introduction

First of all, I want to point out that this paper is about running automated Android tests on devices managed by a Continuous Integration[1] Server like Jenkins[2]. So all available solutions of running tests, e. g. Android JUnit[3], Espresso[4] or Monkey[5] on an Android device via an integrated development environment, e. g. Eclipse or AndroidStudio are not in the scope of this work. Secondly, I want to give a brief overview of the possibility for testing Android applications via Jenkins. At the moment you can use the following Plugins[6]:

Android Emulator Plugin[7]	(Christopher Orr)
AppThwack Plugin[8]	(Andrew Hawker)

The first Plugin provides Monkey and Android JUnit testing on an emulator. The Plugin itself handles all necessary steps to setup, start and delete an emulator.
The second solution provides Appium, Calabash, Android JUnit, Espresso, MonkeyTalk, Robotium, Selendroid and Monkey tests on Android devices in a cloud. So you basically just have to upload your apk[9], which includes the tests. Or you upload two apks, the apk under test and the test apk and you will get back all results.

If we already have such possibilities, why would we want to run tests on our own devices?

Basically, because we have them already. If you develop Android applications you have at least one device and most companies have at least the main devices on the market. Especially if you have an own test department. Sometimes you even must buy a vendor specific device, because

1Continuous Integrations means to integrate new code as much as possible to the main branch. Another part of CI is testing code, which ensures it's quality by running automated tests every day and publish the results.
2 Jenkins is a build and CI Server. http://jenkins-ci.org/ (last visit: 3:49 PM, January 3, 2015 (GMT+1)
3 Unit testing for Android applications.
4 Android UI testing framework by Google, you write all events.
5 Automated UI Testing for android, it creates random events

6 Plugins are installed to Jenkins to add new functionalities.
7 https://wiki.jenkins-ci.org/display/JENKINS/Android+Emulator+Plugin (last visit: 3:49 PM, January 3, 2015 (GMT+1)
8.https://wiki.jenkins-ci.org/display/JENKINS/AppThwack+Plugin (last visit: 3:49 PM, January 3, 2015 (GMT+1)
9 Android application package file (apk), distributes installing an app.

you want to develop for their implementations, e. g. Samsung Knox[10].
Although running tests on a cloud sounds brilliant, we don't need to buy hardware and pay only on-demand. But running tests on devices owned by someone else means that your apks must be obfuscated[11]. In fact you don't want to provide the possibility to gain your source code in clear text by decompiling your apks. But often you obfuscate your apks right before a release. So using AppThwack for daily testing would imply to deal with the obfuscation every day as well. Furthermore the pricing of AppThwack seems quite expensive for daily testing purposes. But it should be used for massive device specific tests right before a release to the Google Playstore.
Although running tests on an emulator is reliable and well implemented for Jenkins by Christopher Orr, it always frightens me to run performance tests on an emulator. Because mostly you run an x64 or x86 machine, which emulates the ARM-Architecture. So I always ask myself which performance are we testing right now. Additionally it will be faster to run tests on a real device instead of an emulator. In fact starting an emulator can take several minutes, if you don't use the Intel HAXM[12] and the x86 emulator images, which aren't available for every Android version.

Considering all the facts above it would be an improvement for the Android Continuous Integration, if we could implement Android devices into Jenkins. So we gain another way of testing Android applications closer to their actual environment without any restrictions with our own devices.

10 “Samsung KNOX Workspace provides hardware and software integrated security for Samsung mobile devices .” (Samsung http://www.samsung.com/global/business/mobile/platform/mobile-platform/knox/) (last visit: 3:49 PM, January 3, 2015 (GMT+1)

11 Obfuscation replaces all names of variables,classes,methods, etc pp to a short term like “aab”, so it will be much harder to understand the code after decompiling it.

12 “Intel HAXM is a hardware-assisted virtualization engine that uses Intel Virtualization Technology to speed up Android app emulation on a host machine.” (Intel: https://software.intel.com/en-us/android/articles/intel-hardware-accelerated-execution-manager) (last visit: 3:49 PM, January 3, 2015 (GMT+1)

2. Problem

Before facing the problems of implementing an Android device into Jenkins I want to explain a bit about Jenkins functionality of distributed builds. Basically, you have one Jenkins server, which is called master and multiple machines called slaves. The master instance handles almost everything related to users or administrators, e. g. user database, build-job configuration, Winstone web server. But the actual build-job, e. g. running a test or building an apk will be delegated via ssh[13] to a slave to ensure the accessibility of the Jenkins master by passing the actual workload to another machine.
The interested reader is referred to the book “Jenkins: The Definitive Guide”[14] by John Ferguson Smart.

Unfortunately, we cannot create a Jenkins slave on top of an Android device, because the master instance needs to install the slave.jar[15] after connecting via ssh. This cannot be done with android's DalvikVM[16], which cannot execute compiled Java byteCode. At this point I want to mention that porting the slave.jar into a slave.apk is not considered as a good solution, because from my point of view it would result in massive work while losing the actual aim. Because even if we ported the code to a Dalvik executable, we would have to compile every Plugin specific code to a Dalvik Executable after installing it on the Jenkins master, before updating the slave.

Furthermore, I figured out, while starting Android JUnit and Monkey tests via an ssh connection to my Nexus 5, that Google does not provide the possibility to run tests via shell for reasonable security reasons. You just get the following message:

13 Secure shell is a cryptographic network protocol to access another machine and execute shell commands.

14 http://it-ebooks.info/book/576/ ISBN: 978-1-4493-0535-2

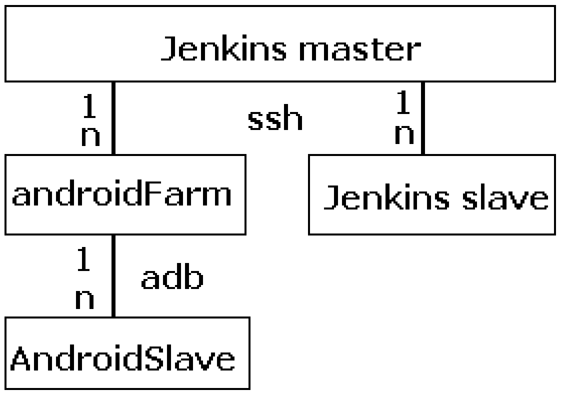
15 Java runnable, which has to be execute after establishing the connection to the slave via ssh in order to fully connect the slave to Jenkins.

16 Dalvik is the process virtual machine (VM) in the Android operating system.

"java.lang.SecurityException: Permission Denial: startInstrumentation ask to run as user -2 but is calling from user 0; this requires android.permission.INTERACT_ACROSS_USERS_FULL"

I added this permission to the manifest, but the error still remained. Additionally, it turns out that we cannot use adb command directly via an ssh connection. I was able to start an adb[17] daemon[18], but no device was listed.

So I decided to implement a slave which acts like a master for all connected Android devices via adb and delegates such commands device specific, which will be called androidFarm.



At this point I want to mention the androidCluster of Joshua Drake, which gave me a lot of inspiration and will be discussed in the Related Work section.

Furthermore we have to take care that this setup is not only capable of executing adb commands on a specific device. It should also block devices for build-jobs if their already used by another build-job[19], pass an exception to Jenkins if the device is not connected to the androidFarm and ensure that the Jenkins build-job fails if an adb command e. g. installing an apk won't be successful.

17 An Android debug bridge daemon communicates with your USB connected Android device

18 Program that runs as a background process and not in the context of an interactive user.

19 A build-job is a predefined task e. g. building a apk or run tests, which consists of several different build-setps and can be simply started by a user via the "build now" button.

3. Detailed Work

In this section we will take a closer look at a possible implementation of such setup mentioned above. First, I want to explain the setup I used to implement the Android Farm to Jenkins and how the Android specific build-jobs are handled. Afterwards I will discuss how those build-jobs are realized via shell scripts. Finally, I will evaluate my tests with three different kinds of tests (Monkey, Espresso, Android JUnit) on rooted[20] and non-rooted devices connected via adb to the androidFarm.

3.1 androidFarm setup

The hardware of the androidFarm slave is a Raspberry Pi (Model A, B+, Pi 2) running Rasbian (Debian Wheezy build 2014-09-09). To connect multiple Android devices I used the D-Link (DUB-H7) USB hub as recommended by Joshua Drake.

In fact that a ssh daemon and a Java runtime-environment is already installed on Rasbian, we can directly connect to the RaspberryPi from the Jenkins master via ssh and establish the connection as a Jenkins slave by copying and running the slave.jar. In order to run adb commands on an ARM-Architecture I needed to cross-compile the adb, this will be explained later in the related work section.
Even so any adb command will start a server, if none is running. I had to start the adb server as root to see devices. So I wrote a script to start such server, which will be executed automatically after booting.

I use environment variables of the Android slave to provide user-friendly names for the connected devices instead of their

20 If you root an Android device you gain super user rights from your normal user context. This isn't provided for security reasons by Google.

serial number. Additionally, I define a variable called androidFARM, which references to the directory where all the necessary files, executables and shell scripts are based.

3.2 Used Android devices

I used 7 devices for the evaluation.

Samsung Galaxy Y(oung) - rooted
Samsung S3
Samsung Note 2
Samsung S4
Nexus 5
Nexus 7 (2012)
Nexus 10

3.3 Build-job setup

First of all the build-job has to be executed on the androidFarm via the “Restrict where this project can be run” option.
The user-friendly name of the specific device has to be passed as a parameter to the build-job in order to bind it.
I copied all necessary apks from a previous ran build-job via the Copy Artifact Plugin[21].
Finally you have to create a “execute shell” build-step[22] and copy the androidFarm bash script. Alternatively you can call the script and pass the “user friendly device” name.

3.4 androidFarm bash script

```
#!/bin/bash
1) eval dvc='$'$device

2)adb=$($androidFARM/androidFarm.sh
       "$dvc" $androidFARM)

3)if [ "$adb" = "adb -s $dvc" ]; then

   # run adb commands directly
4) $adb shell input keyevent 26

   # to pass script as parameter
5) bash -ex $script $adb

6) rmdir $androidFARM/androidPID/$dvc

else
      echo $adb
      exit 1
fi
```

Let's have a closer look to the shell script of the Jenkins execute shell build-step.
In line 1 we evaluate the user-friendly name of the specific device via the environment variables of the Android slave to get its serial.
In line 3 we compare the output of the androidFarm.sh script, which will be discussed soon with the expected output. If they are equal we can choose between writing adb commands directly inside the Jenkins shell via “$adb” (line 4) or reference to a adb shell script (line 5).
After executing the desired commands the directory, used for something like a PID will be deleted (line 6) and the shell terminates.
Othoerwise the variable adb will be echoed to log the error and the shell terminates with a exit code.

3.5 androidFarm shell script

Before describing the androidFarm.sh script I want to explain the prerequisites to execute adb commands on a specific Android device. To do so you specify the -s flag with the device's serial as a parameter when executing adb command.

```
adb -s <serial>  <some command>
```

So we basically just have to wrap the underlined part of the above line and save it into the adb variable.

```
#!/bin/bash
######### functions #######


connected(){
```

21 https://wiki.jenkins-ci.org/display/JENKINS/Copy+Artifact+Plugin (last visit: 3:49 PM, January 3, 2015 (GMT+1)

22 A build-step is part of a build-job, which provides predefined functionalities. So there are different build-steps available e. g. “invoke ant”, “publish JUnit report”.

```
  OUTPUT=`adb devices | grep $1`

  If [ ${#OUTPUT} -lt 1 ];
    then
      echo "false"
    else
      echo "true"
  fi
}
available() {

cd  $2/androidPID
OUTPUT=`ls | grep $1`

if [ ${#OUTPUT} -lt 1 ];
      then
            mkdir $1
            echo "true"
      else
            echo "false"
fi
cd ..
}
######### script ##########

dvc=$1
LEN=$(echo ${#dvc})

1) if [ $LEN -lt 6 ]; then
    echo ”serial seems to short”
    exit 1
else

    deviceConnected=$(connected $dvc)
2) if [ $deviceConnected = "false" ];
then
        echo "not connected”
        exit 1
    else

deviceAvailable=$(available $dvc $2)
3) if [ $deviceAvailable = "false" ];
then
      echo "device already used”
      exit 1
   else
             echo adb -s $1
        fi
    fi
fi
```

At 1 we check if a corresponding serial of user-friendly name of the device is available by the environment variables of the androidSlave by evaluating the length of the string.
At 2 we check if the Android device is connected to the androidFarm by using the function connected().
Afterwards at 3 we check whether the desired device is free to use, by checking if a directory named after the serial of the device exists already in the androidPID directory. If so we create a directory name after the serial to block this device for other build-jobs.
When everything is fine we echo “adb -s <serial>” back to the Jenkins shell script.
In case the desired device has no reference to its serial, is not connected or not available a corresponding error message will be echoed to pass some kind of log to Jenkins.

I want to point out that the handling of the PIDs[23] could be even handled simply in a file, but for the rough test implementation I decided to keep it as simple as possible.

It should be clear now why the shell script inside the Jenkins build-step deletes a directory before terminating if the desired Android device was free to use and not outside the if case. Because we would cause an error in fact of making a device available, which haven't been available for us, in fact it was and probably still is used. This would allow another build-job to bind a already binded device.

3.6 Evaluating the test setup

I used three kinds of test. A simple Monkey, Android JUnit and an Espresso UI test. I want to mention that the scope of this work is not writing perfect tests, the interested reader is referred to the Android Testing Fundamentals.[24] It's more about being able

23 Process identification number
24 http://developer.android.com/tools/testing/testing_android.html (last visit: 3:49 PM, January 3, 2015 (GMT+1)

to run them on Android devices integrated into Jenkins. Here's how I started the test via adb inside the “execute shell” buildstep.

Android Junit and Espresso UI tests:

```
$adb shell am instrument -w
<testpackage>/<testRunner>
```

Pulling the Zutubi[25] result XML[26] file:

```
$adb pull /data/data/<package under
test>/files/junit-report.xml
```

Monkey:

```
$adb sehll monkey -p <package>
-v <amount of events>
```

Alternatively you can take a look at the adb shell scripts, which are called from the Jenkins shell in the Related work section.

After running a couple of tests with this setup, I positively figured out that the device handling works quite nice. Although I could only use 6 devices simultaneously connected to a Pi 2 or B+ (2 on A). I was able to start all described tests and get the directly echoed results. Those results actually provide all necessary information but not in a Jenkins-friendly XML valid way. To do so you should use the Zutubi Android testrunner to provide a XML valid report to Jenkins. First I was only able to pull the zutubi result.xml from the androidSlave to the androidFarm, if the device was rooted. In order to pull them from a non-rooted device I needed to add the “WRITE_EXTERNAL_STORAGE” permission to the manifest, so I could save the result via the -e flag of the Zutubi runner on the local storage, which then can be pulled.

Furthermore a build-job directly fails if a specific device was used already, which is good but needs improvement. And I have to come up with a “unlockScreen” script, because all UI test ran, but such test are useless if they run on the lockscreen. The build-job should wait a defined time-out and try periodically to acquire the desired device. Another solution which will implement such functionality will be discussed in the Additional Work section.

Finally the build-job doesn't terminate with an exit code, if an adb command wasn't successful.

4. Related Work

In this section I want to discuss projects of others, which helped and inspired me while writing this paper. Additionally, I will discuss the adb shell script used for evaluating more detailed.

4.1 androidCluster

The android-cluster-toolkit project by Joshua Drake can be found on github[27]. It handles adb and fastboot commands “on single devices, a selected subset, or all connected devices at once”(github). It's written in Ruby and only needs to access the adb and fastboot commands installed explicitly by the user. To get more details about his work and why he build such an androidCluster view his presentation[28].

Although I didn't want to use his toolkit directly in fact of ruby, which I'm not familiar with. I adopt the “user-friendly device name” feature and I followed his recommendation of using the D-Link USB hub. Additionally it was great to try this toolkit in order to see that handling multiple devices connected to one host is possible.

4.2 adb and fastboot on RaspberryPi

25 http://zutubi.com/source/projects/android-junit-report/ (last visit: 3:49 PM, January 3, 2015 (GMT+1)

26 Extensible Markup Language http://www.w3.org/XML/ (last visit: 3:49 PM, January 3, 2015 (GMT+1)

27 https://github.com/jduck/android-cluster-toolkit (last visit: 3:49 PM, January 3, 2015 (GMT+1)

28.https://www.blackhat.com/docs/us-14/materials/us-14-Drake-Researching-android-Device-Security-With-The-Help-Of-A-Droid-Army.pdf (last visit: 3:49 PM, January 3, 2015 (GMT+1)

I needed to cross-compile the adb from source to be able to run adb commands on the ARM-Architecture of the RaspberryPi.
I basically just followed the steps of the guide[29] by Andres Rudolf. First I wanted to use some already cross-compiled files, which can be found in the xda-developers[30] forum. But those executables were too old, so they couldn't handle the RSA-handshake of an adb connection on devices running Android 4.2.x.

4.3 adb shell scripts

A first try to write an adb shell script, which installs all necessary apks, start the test itself, copy back the report and even uninstall the previously installed apks shows that the amount of passed parameters increases a lot. The following parameters have to be passed:

- $1 workspace of build-job
- $2 adb with wrapped serial
- $3 fastboot with wrapped serial
- $4 apk under test
- $5 test apk
- $6 package name of apk under test
- $7 package name of test apk

The actual shell script used for evaluating:

```
cd $1

$2 install $4
$2 install $5

# start test
$2 shell am instrument -w $7/com.
zutubi.android.junitreport.
JUnitReportTestRunner

# copy report
$2 pull /data/data/$6/files/junit-
      report.xml

$2 uninstall $4
$2 uninstall $5
```

Everything worked out except the error handling of adb commands. This couldn't be achieved by simply evaluting the $? variable after executing an adb command.
So we need wrapper commands for all necessary adb commands.

5. Resume

There is a way to access Android devices from Jenkins and run tests on them. It's even possible by using a low-cost RaspberryPi as a host for the Android devices, some shell scripts and a quite manually setup of Jenkins configurations. Basically we could use such setup for daily testing. But there are still some issues to be resolved.

The handling of the androidSlave was handy for the test evaluation, but should be improved. Especially the fact that a build-job fails, if a device is already used or not connected instead of waiting for a timeout before doing so.

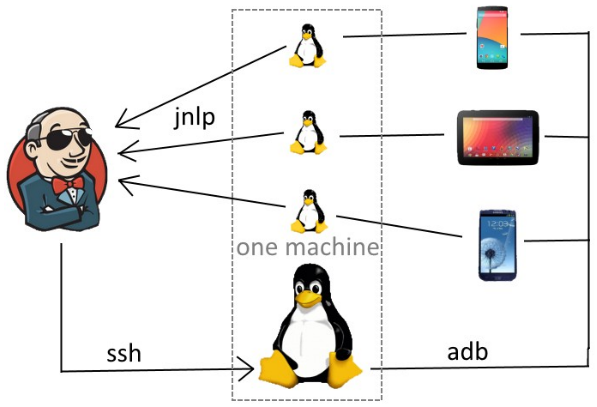
Additionally, I need to implement a wrapper for all adb commands, e. g. installing an apk to be able to verify if they succeeded or mark the build-job's result correspondingly otherwise.

Especially the issue mentioned first could be resolved by creating a slave instance for each Android device connected to the androidFarm, which results in multiple connection. But then the build-job queue for each slave, in our case an Android device would be handled by Jenkins automatically.

29 http://android.serverbox.ch/?p=1217 (last visit: 3:49 PM, January 3, 2015 (GMT+1)
30 http://www.xda-developers.com/
(last visit: 3:49 PM, January 3, 2015 (GMT+1)

6. Additional Work

To get more Transparency of the androidSlaves I decided to create a slave instance for every Android device. So it appears as a slave itself in the jenkins environment.

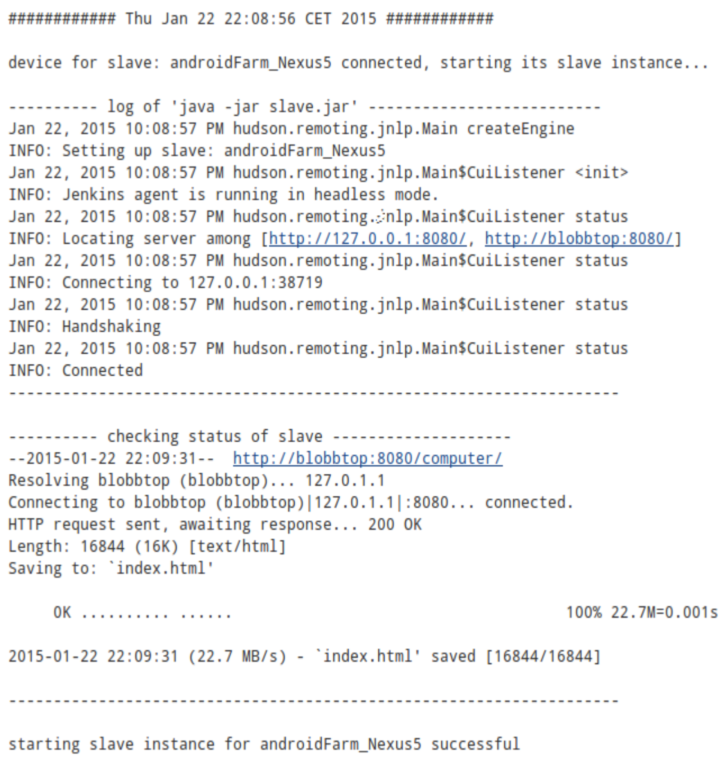


Therefore you must create a slave instance per android device manually and pass the slave name, the android device serial and the jnlp secret via one of the three androidFarm jobs to the devices.conf file in order to start a slave if its corresponding android device is connected. Those jobs are:

- “add a device”
- “run androidFarm daemon”
- “list devices”

The androidFarm script is able to start a slave instance via JNLP for every known device and it shuts down the slave instance if its corresponding android device is disconnect. So every android device can be still used by developers or quality engineers. Additionally all androidSlave jobs will first lighten the screen and darken it again if the build-job ends. So it's even possible to provide information about usage of an android device without accessing Jenkins.

Furthermore I tried to get some feedback of the slave's status by parsing the node's overview page. It works but takes so much time, that I decided to not use it in practice and only print the feedback of starting a slave instance to the log file.



```
############ Thu Jan 22 22:08:56 CET 2015 ############

device for slave: androidFarm_Nexus5 connected, starting its slave instance...

---------- log of 'java -jar slave.jar' --------------------------
Jan 22, 2015 10:08:57 PM hudson.remoting.jnlp.Main createEngine
INFO: Setting up slave: androidFarm_Nexus5
Jan 22, 2015 10:08:57 PM hudson.remoting.jnlp.Main$CuiListener <init>
INFO: Jenkins agent is running in headless mode.
Jan 22, 2015 10:08:57 PM hudson.remoting.jnlp.Main$CuiListener status
INFO: Locating server among [http://127.0.0.1:8080/, http://blobbtop:8080/]
Jan 22, 2015 10:08:57 PM hudson.remoting.jnlp.Main$CuiListener status
INFO: Connecting to 127.0.0.1:38719
Jan 22, 2015 10:08:57 PM hudson.remoting.jnlp.Main$CuiListener status
INFO: Handshaking
Jan 22, 2015 10:08:57 PM hudson.remoting.jnlp.Main$CuiListener status
INFO: Connected
-------------------------------------------------------------------

---------- checking status of slave --------------------
--2015-01-22 22:09:31--  http://blobbtop:8080/computer/
Resolving blobbtop (blobbtop)... 127.0.1.1
Connecting to blobbtop (blobbtop)|127.0.1.1|:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 16844 (16K) [text/html]
Saving to: `index.html'

     0K .......... ......                                  100% 22.7M=0.001s

2015-01-22 22:09:31 (22.7 MB/s) - `index.html' saved [16844/16844]

-------------------------------------------------------------------

starting slave instance for androidFarm_Nexus5 successful
```

All in all I could improve the Android device handling, so now a test multijob can run on multiple Android devices simultaneously and if one device is not connected while starting the job, its build-job will wait until this device will be connected again.

It's even possible to create device groups by using multiple slave labels for each androidSlave. As an example let's use the following labels for a Nexus 10:

- android
- android_tablet
- android_Nexus
- android_Nexus_tablet
- android_Nexus_10

Such label handling provides an easy and effective way of running tests for specific android device groups like all device, all tablets, all Nexus devices or one specific device.

Although this setup could only handle 6 devices on a RaspberryPi (B+ and 2). On a x64, 4 GB, dual core machine I was able to launch all seven androidSlaves and run tests on them. Each androidSlaves consumes 20 to 45 MB of RAM depending on the job. To save some storage the androidFarm Script, which can be found on github[31] creates a soft link to the tools directory used by the androidFarm instance to not copy it over and over again from Jenkins to the slave when creating a androidSlave. Therefor a specific directory structure must be set up. This can be read in the doc at the github project.

I made an unlock screen job, which fires random UI event to open the lock screen and then enters the passed pin code to unlock the lock screen. So it's possible to run UI tests on the devices without unlocking them manually.

Finally i've come up with a setup which let's you handle android devices for testing. By starting an own slave instance for every device. All devices are still usable by employees without crashing Jenkins or a build-job, such will just wait until the device is connected again.

I recently found an “android device connector” plugin, which offers the possibility to deploy an apk to an android device connected to any slave or even the master. I consider to merge my androidFarm Script functionalities of running tests to this plugin. But this will be discussed in a different paper.

31 https://github.com/blobbsen/androidFarm