Docker Engine Services in Android OS

B.E. (CIS) PROJECT REPORT

by

Usama Bin Masood

Department of Computer and Information Systems Engineering

NED University of Engineering & Technology
Karachi-75270

Docker Engine Services in Android OS

B.E. (CIS) PROJECT REPORT

by

Usama Bin Masood

Dr. Muhammad Ali Ismail

Muhammad Faraz Hyder

(Internal Advisors)

Department of Computer and Information Systems Engineering

NED University of Engineering & Technology
Karachi-75270

B.E. (CIS) PROJECT REPORT

Project Group:

Usama Bin Masood	CS-071
Jasim Ahmed	CS-044
Muhammad Hummad	CS-055
Amrat Kumar	CS-069

BATCH: 2012-13

Project Advisors:

Dr. Muhammad Ali Ismail

Muhammad Faraz Hyder

(Internal Advisors)

December 2016

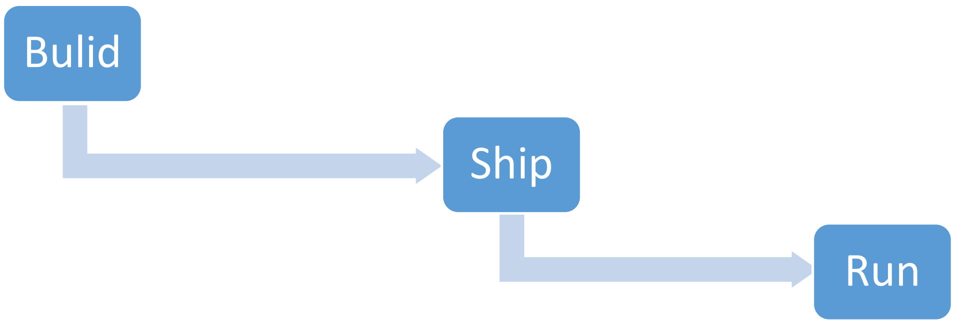
Department of Computer and Information Systems Engineering

NED University of Engineering & Technology

Karachi-75270

ABSTRACT

This project intends to provide the execution of container build applications on Android OS. Building and deployment has been a potential problem due to compatibility and customization issues, since the application development started. Docker engine provides the ease in development and deployment of application due to "all-in-one" run time environment approach that is termed as containers.



Container technology facilitate developer to package up an application with all of the parts it need, such as binary code, runtime, system tools, libraries and other dependencies and ships them all out as one package. The developer can rest assured that his delivered package known as "container", will be able to run on any other 'Docker' regardless of any customized settings that machine might have which could differ from the machine employed for writing and testing the code.

In this way container build applications can run easily on any android device that is Docker supported.

S.No. **CONTENTS** **Page No.**

CHAPTER 1

Introduction

Docker engine is the solution to the problem that demolishes the need for multiple operating system or machine with different operating systems.

The objective of the development of this project is to deploy the Docker engine on android operating system to enhance its capability of executing applications of other platforms (windows, iOS, blackberry etc.) To make application run on Docker, they are built/packed in container format. Containers contain application and programs with their binary files and libraries to support platform dependent applications. After the successful execution of containers over Android Docker, we will implement a light weight Docker on Android with its Master piece on server in synchronous communication to exchange libraries of current use.

1.1 Docker Engine:

Docker Engine was started by Solomon hikes as an internal project at dotCloud, a platform-as-a-service company, it was initially contributed by other engineers of dotCloud including Andrea Luzzardi and Francois-Xavier Bourlet. Docker Engine portrays an evolution of dotCloud's proprietary technology. Docker was released as open source in March 2013.On March 2014 its version 0.9 was released with certain updates interface and filesystem. One of the major changes was dropping of LXC as the default execution environment and replaced it with its own libcontainer library. At the same time Docker now supports a much broader range of isolation tools through the use of execution drivers, which include: systemd-nspawn, libvirt-lxc, libvirt-sandbox, qemu/kvm, BSD Jails, Solaris Zones, and chroot. Docker can now manipulate namespaces, control groups, apparmor profiles, network interfaces and firewalling rules - all in a consistent and predictable way, and without depending on LXC or any other userland package. This drastically reduces the number of moving parts, and insulates Docker from the side-effects introduced across versions and distributions of LXC. In fact, libcontainer delivered such a boost to stability that we decided to make it the default. In other words, as of Docker 0.9, LXC is now optional.

The execution driver API provides an interface to run Docker on non-Linux Systems such as FreeBSD, Solaris, OpenSolaris, illumos, OS X, and Windows.

1.2 What is Linux?

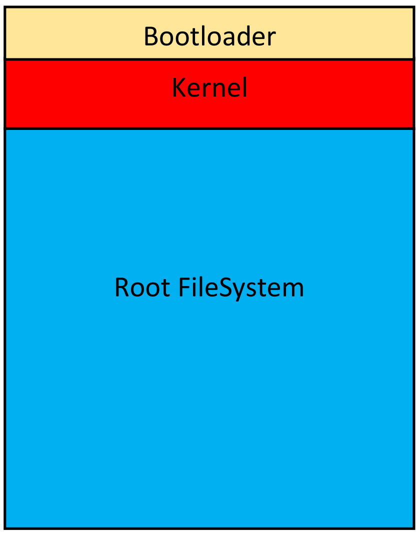
From home appliances to cars, supercomputer and smart phones, the linux operating system is everywhere. It's run most of our internet, the supercomputer breakthroughs, the stock exchange, space shuttle and exploration used by NASA and various other countries in world. But before linux become a accessible to a desktop user, embedded system and server it was the most reliable and worry free operating systems available.

Just like Windows XP, Windows Vista, Windows 7, Windows 10, Mac OS X, Solaris OS X, Linux is an operating system. An operating system is software that manages all of the hardware resources associated with your desktop or laptop. It is an interface that provides control of hardware through software. Without the Operating System there will be no communication between hardware and software. Hence the software won't function. The operating system is comprised of number of pieces:

The OS is comprised of a number of pieces:

1.2.1 The Boot Loader:

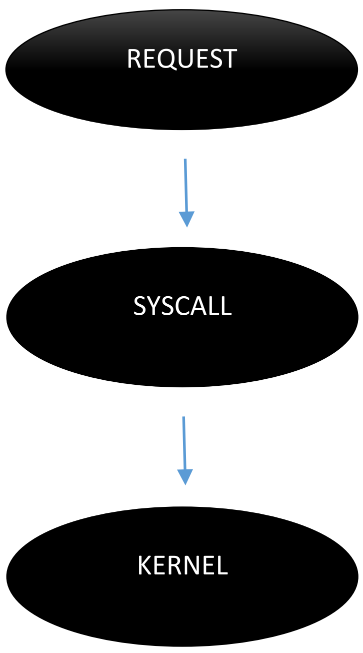
Boot loader is a software that manages the boot process of a computer. For most of the user it will be the splash screen that occurs in starting of computer and eventually goes away till the operating is booted.



Memory Block

1.2.2 The Kernel:

This is one thing in whole system that is actually called the “Linux”. The kernel is the heart of the operating system that manages the system resources(booting up , allocating memory(RAM),assigning CPU ids and process ids, allocation of input/output block(peripheral devices). The kernel is the low level interface that manages the hardware driver such as sound card, video card and graphic card etc. It is responsible for servicing resource requests from application and the management of resources. A kernel facilitates resources reqests by the application and allocate the desired resources. Kernel uses system call (or syscall) interface to handshake with the application.



It is also knows as the Gate-keeper between application and resources.

1.2.3 Daemons:

Daemons are background services such as (printing, sound, scheduling, etc.) that either start up during boot, or after you log into the desktop.

1.2.4 The Shell:

The shell also known as the Linux command line. Shell provides a command line interface to system process, memory, files, folder, partitions and the native application. At one time, scared people away from Linux the most (assuming they had to learn a seemingly archaic command line structure to make Linux work). This is no longer the case. With modern desktop Linux, there is no need to ever touch the command line.

1.2.5 Graphical Server:

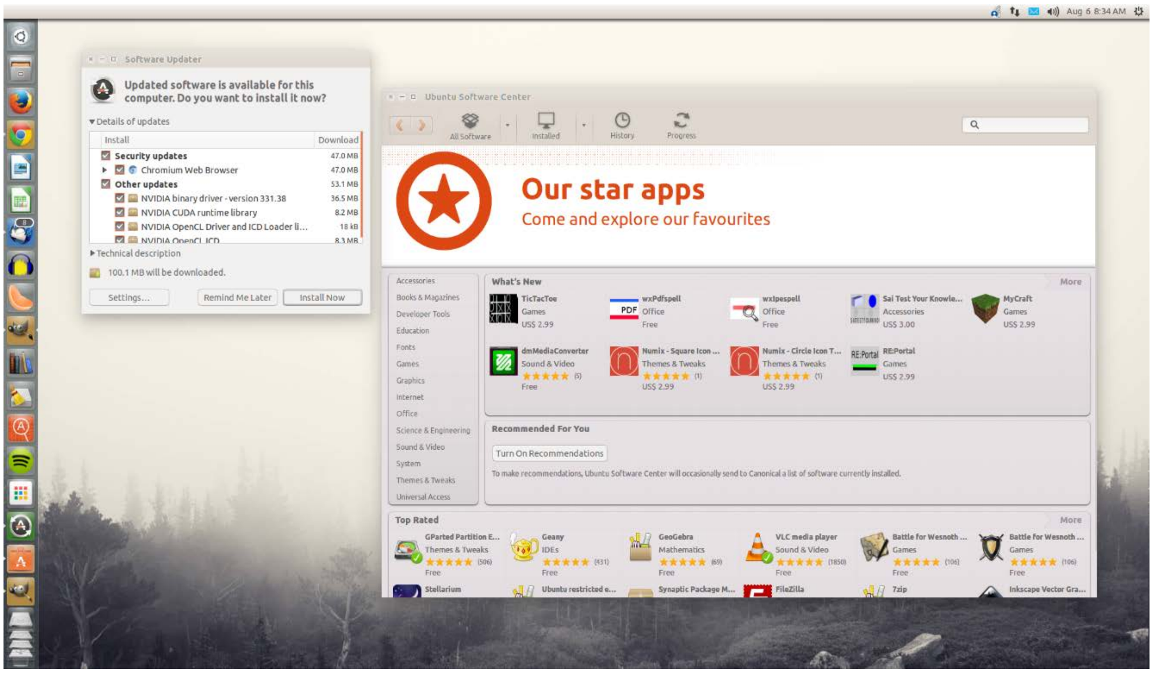
This is the sub-system that displays the graphics on your monitor. It is commonly referred to as the X server or just “X”.

1.2.6 Desktop Environment:

This is the piece of the puzzle that the users actually interact with. There are many desktop environments to choose from (Unity, GNOME, Cinnamon, Enlightenment, KDE, XFCE, etc). Each desktop environment includes built-in applications (such as file managers, configuration tools, web browsers, games, etc).

1.2.7 Applications:

Desktop environments do not offer the full array of apps. Just like Windows and Mac, Linux offers thousands upon thousands of high-quality software titles that can be easily found and installed. Most modern Linux distributions include App Store-like tools that centralize and simplify application installation. For example: Ubuntu Linux has the Ubuntu Software Center (Figure) which allows you to quickly search among the thousands of apps and install them from one centralized location. These applications can be accessed modified directly using UI or command line interface.



Updated software is available for this computer. Do you want to install it now?

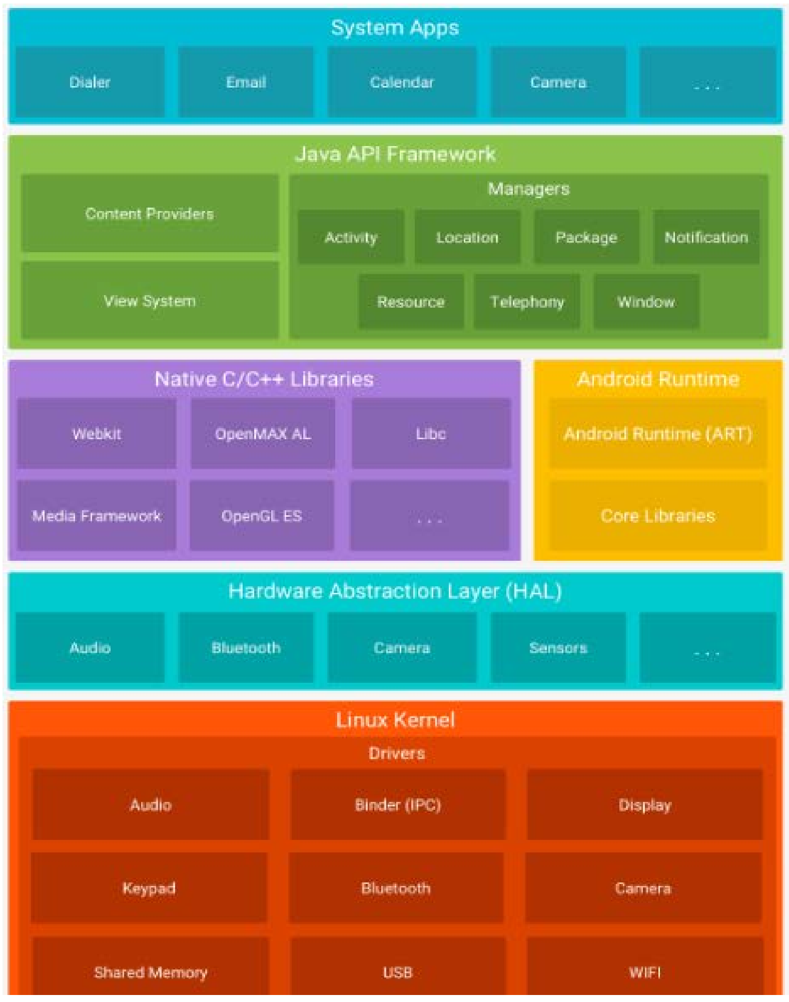
Install	Download
Security updates	47.0 MB
Chromium Web Browser	47.0 MB
Other updates	53.1 MB
NVIDIA binary driver - version 331.38	36.5 MB
NVIDIA CUDA runtime library	8.2 MB
NVIDIA OpenCL Driver and ICD Loader li...	18 kB
NVIDIA OpenCL ICD	8.3 MB

100.1 MB will be downloaded.

To make recommendations, Ubuntu Software Center will occasionally send to Canonical a list of software currently installed.

1.3 Creating Linux Kernel for Android Operating System:

Android software stack.



1.4 The Linux Kernel

Positioned at the bottom of the Android software stack, the Linux Kernel provides a level of abstraction between hardware and the upper layers of the Android software stack. Based on Linux version 2.6, the kernel provides preemptive multitasking, low-level core system services such as memory, process and power management in addition to providing a network stack and device drivers for hardware such as the device display, Wi-Fi and audio.

The original Linux kernel was developed in 1991 by Linus Torvalds and was combined with a set of tools, utilities and compilers developed by Richard Stallman at the Free Software Foundation to create a full operating system referred to as GNU/Linux. Various Linux distributions have been derived from these basic underpinnings such as Ubuntu and Red Hat Enterprise Linux.

It is worth to note, that Android only uses the Linux kernel. That said, it is important to note that the Linux kernel was originally developed for use in traditional computers in the form of desktops and servers. In fact, Linux is now most widely deployed in mission critical enterprise server environments.

1.5 Android Runtime - Dalvik Virtual Machine:

The Linux kernel provides a multitasking execution environment allowing multiple processes to execute concurrently. So it can be assumed that each Android application simply runs as a single process directly on the Linux kernel. In fact, each application running on an Android device does so within its own instance of the Dalvik virtual machine (VM).

Running applications in virtual machines provides a number of advantages. Firstly, applications are essentially sandboxed, in that they cannot detrimentally interfere (intentionally or otherwise) with the operating system or other applications, nor can they directly access the device hardware. Secondly, this enforced level of abstraction makes applications platform neutral in that they are never tied to any specific hardware.

CHAPTER 2

Debian Package in Android Virtual Machine

In this chapter we'll discuss about the Virtualization technique used to deploy Docker on Android system. Android is already an OS, so we can create a virtual instance of Linux on Android OS and provide the services of Docker to the Android User.

2.1 Motivation

The motivation for deploying Linux virtually on Android came to exist because Linux is well supported by Docker officially. After creating an instance of Linux on Android, our next step would be to deploy Docker on Android. The Linux would be installed on android system manually through creating virtual instance of it using different approaches. Then after creating the Linux instance, dependencies of Docker would be taken care of. After successfully accomplishing these steps then Docker would be installed on Linux through official repository.

2.1.1 Advantages

The biggest advantage of this approach is to get rid from the filesystem problem and architecture of Android. Another advantages include is not to worry about the dependencies of android and Docker and would not be necessary to resolve the dependencies between both of them.

2.1.2 Dis-Advantages

The disadvantages here which will be faced is the overhead of a virtual instance over Android. Though Android users prefer their phones not to lag and get slow. But because of this approach the UX of Android users will surely be changed.

2.2 How to Implement

A big question arises here, how to Implement the virtualization technique over ARM architecture. Many possibilities are there to implement and integrate Linux over Android. We will discuss them all in the next section.

2.2.1 Pre-Requisite

Before implementing this phase, we need to root our phone which is a must pre-requisite for solving the problem of Linux and accessing the Linux over Android. Rooting an Android simply means to gain root access of Android system. Firstly, any user of Android does not have any right over Android. You can use it, install applications on it, but you can't make any change to the system files. After gaining root access you could get to access the root folder and can change any system files, plus you can install any app that you can't use on your rooted phone.

2.2.2 Advantages

By rooting our Android, we would get the root access of the Android. Android root access will be beneficial for us in deploying and using Linux on an Android. Many apps which we will be using for installing Linux would be in need of the rooted device and that's why we have to do it more accurately. Another advantage of rooting is we will be

2.2.3 Dis-Advantages

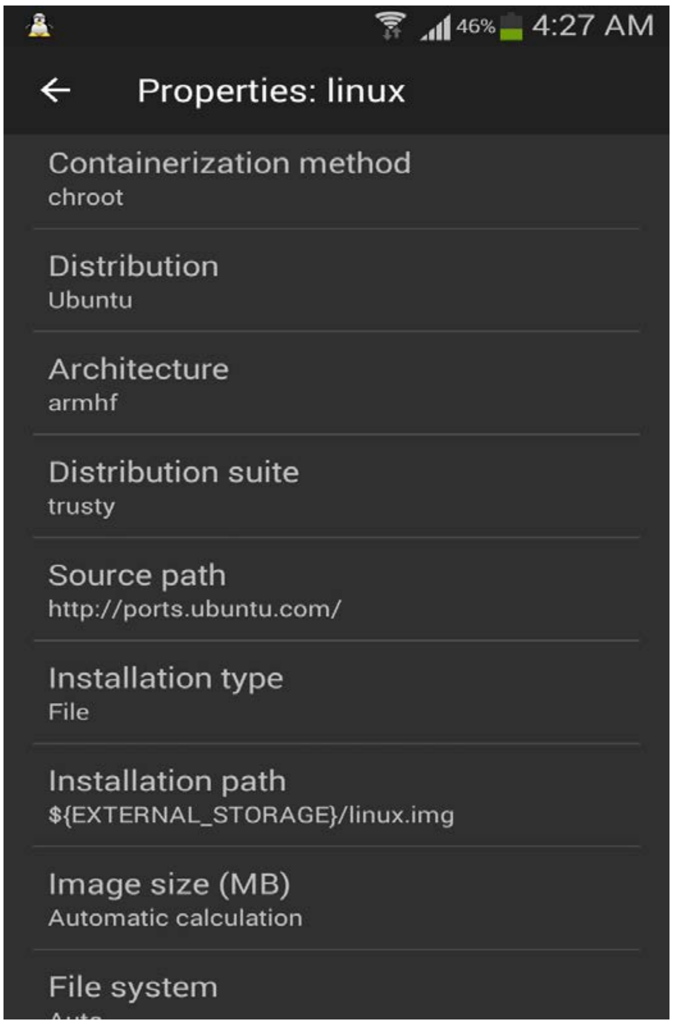
The main dis-advantage of rooting is we would void our warranty. Another dis-advantage is we can get our phone bricked, and there's no turning back then. Rooting is dangerous if one didn't have the proper know-how of the rooting.

2.4 Virtual Environment

Here we'll create a virtual environment of Linux over the Android. For creating virtual environment, we would be using Linux Deploy, Limbo PC Emulator.

2.4.1 Linux Deploy

At first, before using the Linux Deploy we need several other applications too which will be considered as the dependency on which Linux Deploy depend and will use after the correct installation of Linux distribution. These applications include VNC Server and Android Terminal. Android Terminal for accessing the instance of Linux distribution, whereas VNC server for accessing the GUI of desired instance of Linux.



46% 4:27 AM

Properties: linux

Containerization method
chroot

Distribution
Ubuntu

Architecture
armhf

Distribution suite
trusty

Source path
http://ports.ubuntu.com/

Installation type
File

Installation path
${EXTERNAL_STORAGE}/linux.img

Image size (MB)
Automatic calculation

File system

Linux Deploy is an open source app that offers an easy way to install and run a specific Linux Distribution. This Linux distribution will run on a chroot environment. chroot Environment will act as a temporary root directory and also create an isolated environment that doesn’t need to interfere with the rest of the android system.

To install now the specific distribution on Android, launch the app and route to Properties tab. It will show a list of configurable options. Select your desired Linux distribution from the Distribution. We will here be choosing Ubuntu.

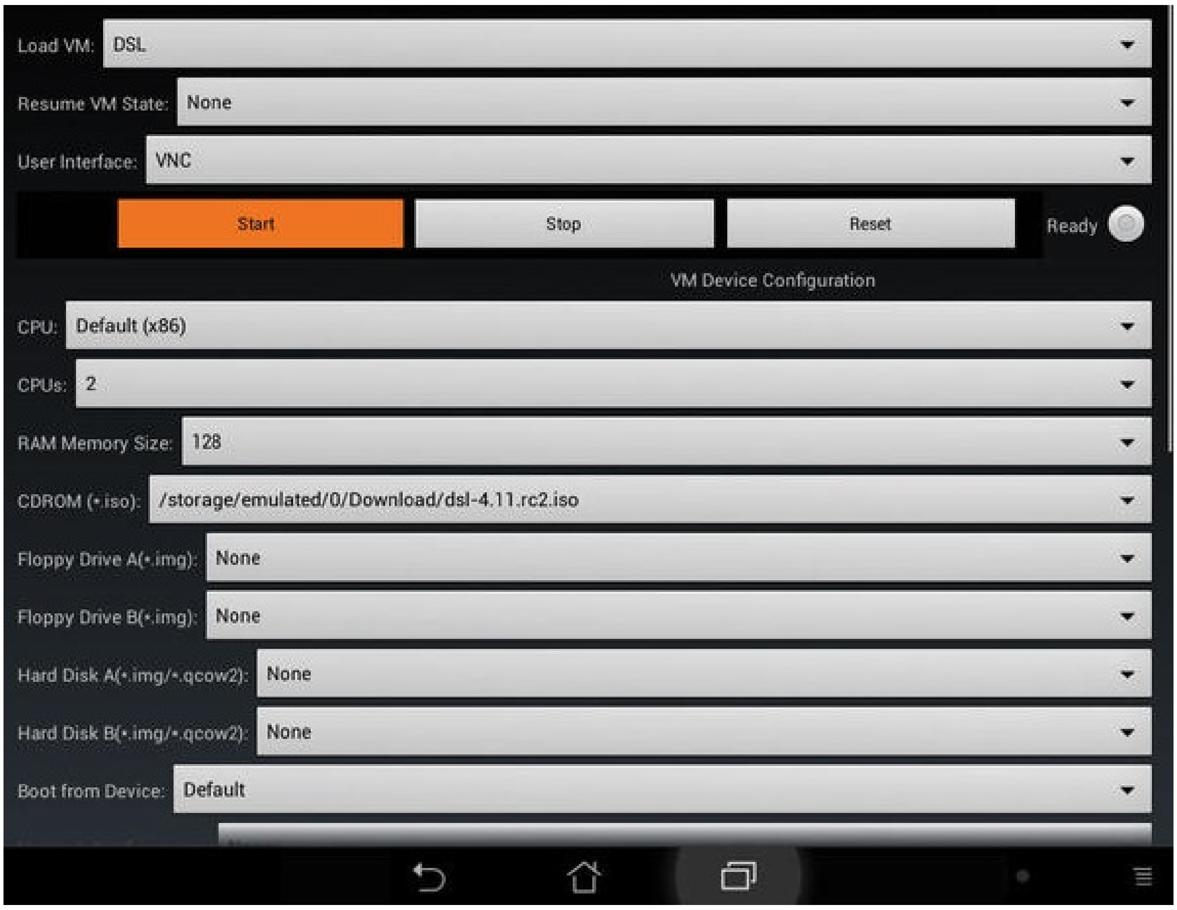
Linux Deploy then install Ubuntu (Linux) in an image file, you can further change the properties of Linux Deploy as your choice. Enable SSH option to be able to connect to the current Linux. Enable GUI configuration for the GUI environment.

Once you are done with these configuration, tap Install and wait Linux Deploy to finish the installation.

Linux Deploy will confirm the Installation once completed. Next thing you have to do is to boot the installed Linux distribution. Establish an SSH connection to the running Linux instance using its IP address, username and password. 7Establish the graphical desktop is equally easy, Launch the VNC server and connect to the running instance with the given set of username and password.

2.4.2 Limbo Pc Emulator

This provides another way to run Linux on your Android phone. This little app allows a lightweight Linux distribution. Limbo is not particularly fast, as it emulates the x86 architecture on the ARM device. Start by using the ISO image of the desired Linux distribution. Launch the Limbo app and make a New VM. The desired image of the Linux distribution which you early opt out for using can be put into CDROM option of it. Enable external VNC option here which will be helpful in accessing the graphical desktop from a remote machine. Tap start to launch the created VM.



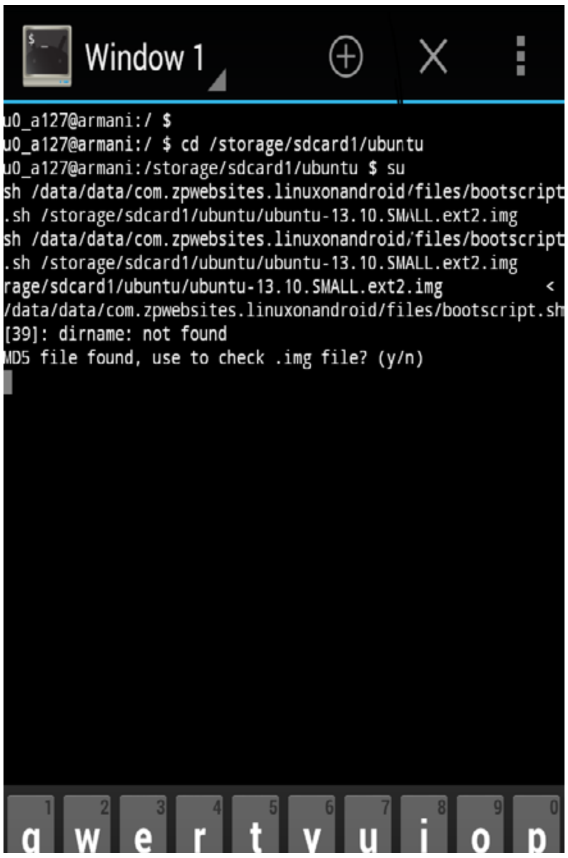
Load VM:	DSL
Resume VM State:	None
User Interface:	VNC
CPU:	Default (x86)
CPUs:	2
RAM Memory Size:	128
CDROM (*.iso):	/storage/emulated/0/Download/dsl-4.11.rc2.iso
Floppy Drive A(*.img):	None
Floppy Drive B(*.img):	None
Hard Disk A(*.img/*.qcow2):	None
Hard Disk B(*.img/*.qcow2):	None
Boot from Device:	Default

2.4.3 COMPLETE LINUX INSTALLER

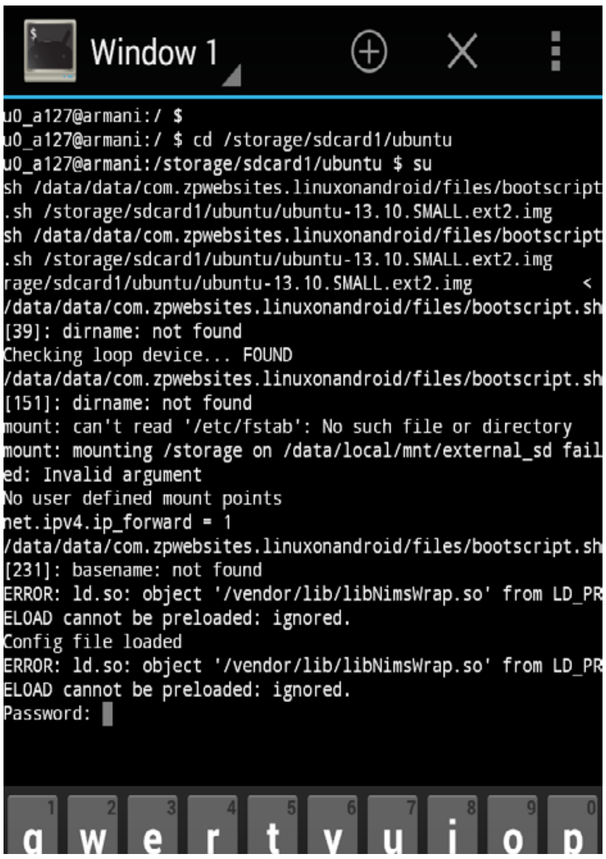
To run Complete Linux installer, we need rooted device in advance, Android VNC and Terminal Emulator.

2.4.4 How to Make Virtual Image of Linux

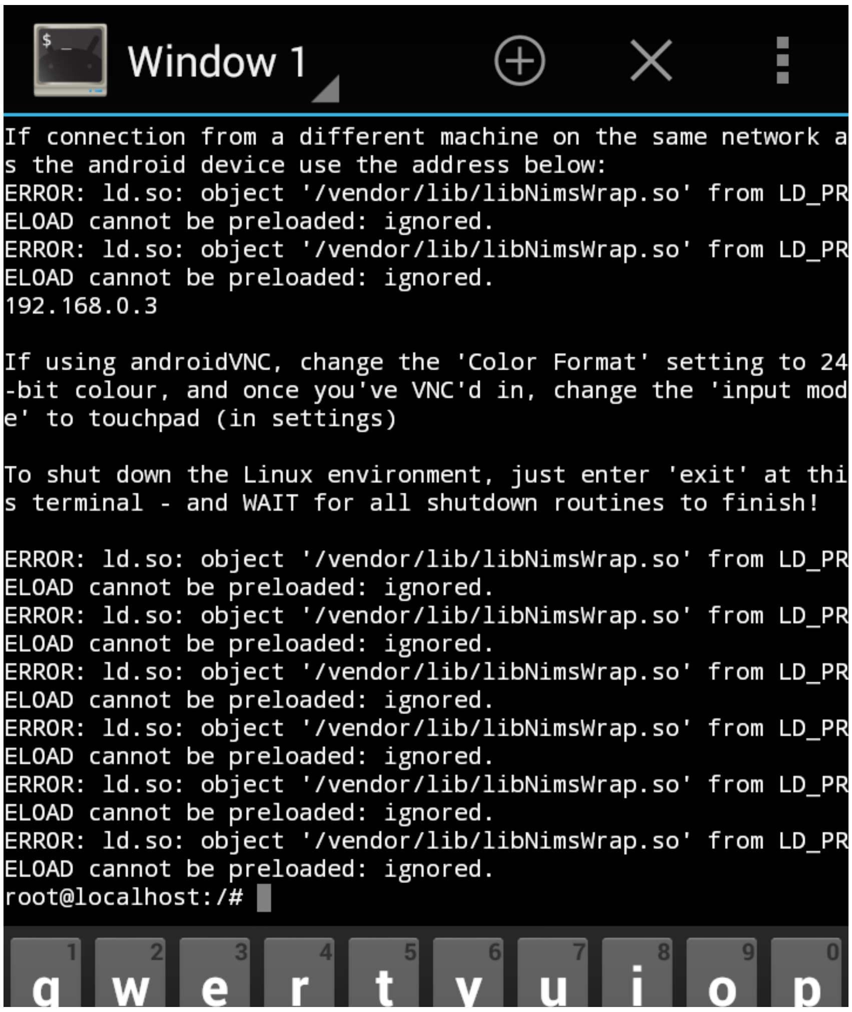
Install the required apk's to the device and launch Complete Linux Installer. Choose the required distribution of Android that is required. Make a new folder 'ubuntu' and extract the downloaded file. Again launch the Complete Linux Installer app and Launch the selected Linux Distributions which you earlier extracted on your sdcard. Save changes and tap "Start Linux". It will open up the terminal app, there you'll be asked to check .img file. Password will be required to enter into Linux, enter 'ubuntu' and tap enter. Terminal will act as a CLI of Linux. For getting GUI of Linux we would be using Android VNC app.



```
u0_a127@armani:/ $
u0_a127@armani:/ $ cd /storage/sdcard1/ubuntu
u0_a127@armani:/storage/sdcard1/ubuntu $ su
sh /data/data/com.zpwebsites.linuxonandroid/files/bootscript
.sh /storage/sdcard1/ubuntu/ubuntu-13.10.SMALL.ext2.img
sh /data/data/com.zpwebsites.linuxonandroid/files/bootscript
.sh /storage/sdcard1/ubuntu/ubuntu-13.10.SMALL.ext2.img
rage/sdcard1/ubuntu/ubuntu-13.10.SMALL.ext2.img             <
/data/data/com.zpwebsites.linuxonandroid/files/bootscript.sh
[39]: dirname: not found
MD5 file found, use to check .img file? (y/n)
```



```
u0_a127@armani:/ $
u0_a127@armani:/ $ cd /storage/sdcard1/ubuntu
u0_a127@armani:/storage/sdcard1/ubuntu $ su
sh /data/data/com.zpwebsites.linuxonandroid/files/bootscript
.sh /storage/sdcard1/ubuntu/ubuntu-13.10.SMALL.ext2.img
sh /data/data/com.zpwebsites.linuxonandroid/files/bootscript
.sh /storage/sdcard1/ubuntu/ubuntu-13.10.SMALL.ext2.img
rage/sdcard1/ubuntu/ubuntu-13.10.SMALL.ext2.img             <
/data/data/com.zpwebsites.linuxonandroid/files/bootscript.sh
[39]: dirname: not found
Checking loop device... FOUND
/data/data/com.zpwebsites.linuxonandroid/files/bootscript.sh
[151]: dirname: not found
mount: can't read '/etc/fstab': No such file or directory
mount: mounting /storage on /data/local/mnt/external_sd fail
ed: Invalid argument
No user defined mount points
net.ipv4.ip_forward = 1
/data/data/com.zpwebsites.linuxonandroid/files/bootscript.sh
[231]: basename: not found
ERROR: ld.so: object '/vendor/lib/libNimsWrap.so' from LD_PR
ELOAD cannot be preloaded: ignored.
Config file loaded
ERROR: ld.so: object '/vendor/lib/libNimsWrap.so' from LD_PR
ELOAD cannot be preloaded: ignored.
Password:
```

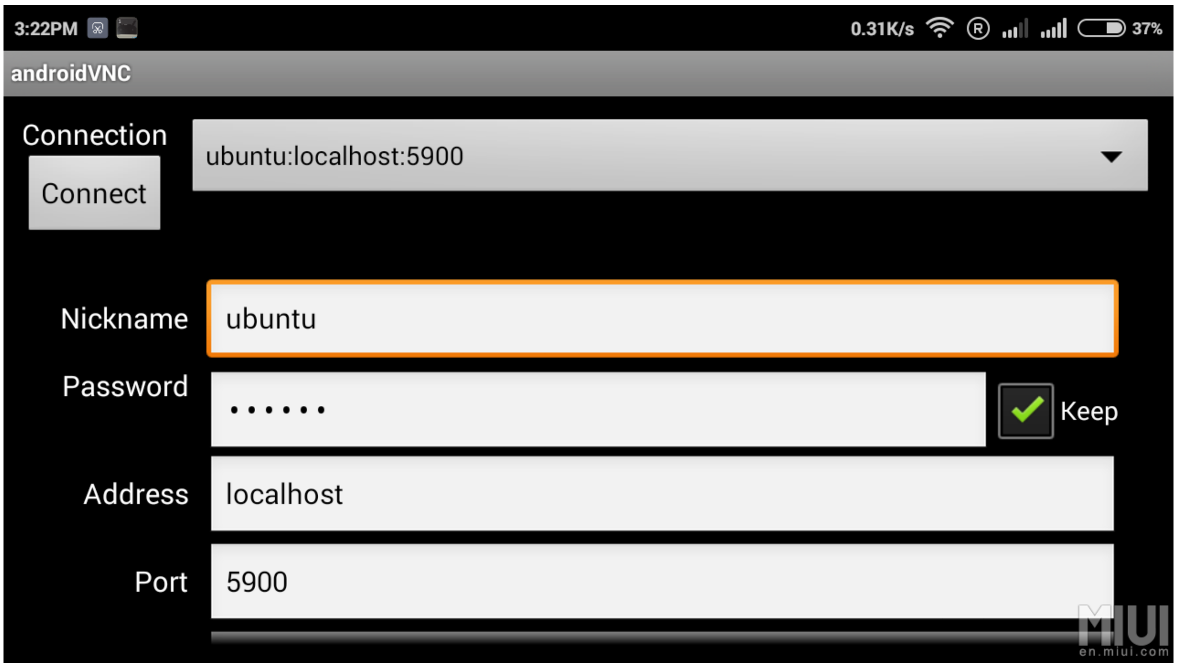


```
If connection from a different machine on the same network a
s the android device use the address below:
ERROR: ld.so: object '/vendor/lib/libNimsWrap.so' from LD_PR
ELOAD cannot be preloaded: ignored.
ERROR: ld.so: object '/vendor/lib/libNimsWrap.so' from LD_PR
ELOAD cannot be preloaded: ignored.
192.168.0.3

If using androidVNC, change the 'Color Format' setting to 24
-bit colour, and once you've VNC'd in, change the 'input mod
e' to touchpad (in settings)

To shut down the Linux environment, just enter 'exit' at thi
s terminal - and WAIT for all shutdown routines to finish!

ERROR: ld.so: object '/vendor/lib/libNimsWrap.so' from LD_PR
ELOAD cannot be preloaded: ignored.
ERROR: ld.so: object '/vendor/lib/libNimsWrap.so' from LD_PR
ELOAD cannot be preloaded: ignored.
ERROR: ld.so: object '/vendor/lib/libNimsWrap.so' from LD_PR
ELOAD cannot be preloaded: ignored.
ERROR: ld.so: object '/vendor/lib/libNimsWrap.so' from LD_PR
ELOAD cannot be preloaded: ignored.
ERROR: ld.so: object '/vendor/lib/libNimsWrap.so' from LD_PR
ELOAD cannot be preloaded: ignored.
ERROR: ld.so: object '/vendor/lib/libNimsWrap.so' from LD_PR
ELOAD cannot be preloaded: ignored.
root@localhost:/#
```



Connection: ubuntu:localhost:5900

Connect

Nickname: ubuntu

Password: •••••• ✓ Keep

Address: localhost

Port: 5900

CHAPTER 3

Android as working Container

Vision: Make base image of android OS and run applications as containers. Theory about Android

3.1 Android (operating system):

Android is a mobile operating system developed by Google Company, Android is based on Linux and designed for touchscreen devices such as smart-phones and tablets. It contains Web browser and end user applications that can be downloaded from the google play store. Android's user interface is mainly based on direct buttons, using touch gestures that correspond to actions, such as swiping, tapping and long taping, to operate the on-screen objects, along with a virtual keyboard for text input which help to enter text. Furthermore, to touchscreen devices, Google has further developed Android TV for televisions, Android Auto for cars, and Android Wear for wrist watches, each with a specialized user interface. Variants of Android are also used on notebooks, game consoles, digital cameras, and other electronic gadgets which are based on visuals and touch screen.

Android's source code is released by Google under open source licenses which is free for everyone.

3.2 Architecture of android

On top of the Linux kernel, there are the middleware, libraries and APIs written in C, and application software running on an application framework which includes Java-compatible libraries. Development of the Linux kernel independently of other Android's source code bases and customize versions are available and can be build depending upon the user requirements and device.

Until version 5.0, Android used Dalvik as a process virtual machine with trace-based just-in-time (JIT) compilation to run Dalvik "dex-code" (Dalvik Executable), this translation is to the Java bytecode. these trace-based JIT principle followed by it, in addition to interpreting the majority of application code of android, Dalvik machine performs the compilation and native execution of select frequently executed code segments ("traces") each time an application is launched to the device, when Android 4.4 was introduced Android Runtime (ART) as a new runtime environment, which uses ahead-of-time (AOT) compilation to entirely compile the application bytecode into machine code upon the installation of an application. In Android 4.4, ART was an experimental feature and not enabled by default; it became the only runtime option in the next major version of Android, 5.0.

For its Java library, the Android platform uses a subset of the now discontinued Apache Harmony project. Next version Java implementation based on OpenJDK has been switched in Android.

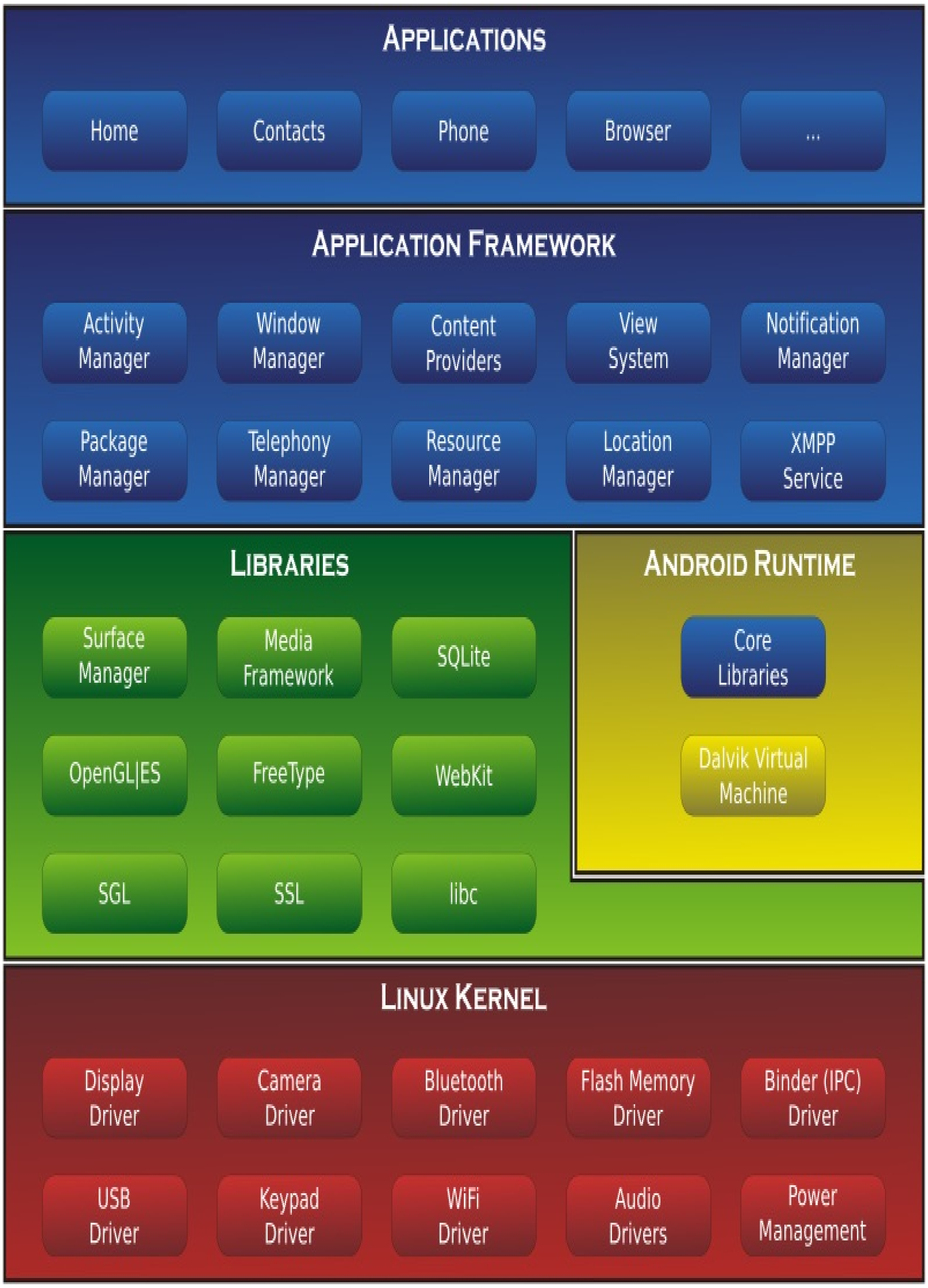
Android's standard C library, Bionic, which was developed by the Google only for the Android, as a variation of the BSD's standard C library code. Bionic itself has been designed with several major features specific to the Linux kernel. The main positive aspect of using Bionic rather than of the GNU C Library (glibc) or uClibc are its smaller runtime footprint, and optimization plus efficient for the low-frequency CPUs. At the same time, Bionic is licensed under the terms of the BSD licence, which Google thinks more suitable for the Android's overall licensing model for open source.

Aiming for a different licensing model for open-source, towards the end of 2012, Google switched the Bluetooth stack in Android operating system from the GPL-licensed BlueZ to the Apache-licensed BlueDroid.

Android don’t have a native X Window System by its default, nor it supports the full set of functioning of standard GNU libraries. This issue made it difficult to port the existing Linux applications or libraries to Android, until version R5 of the Android Native Development Kit brought support for applications written completely in C or C++.Libraries written in C may also be used in applications by injection of a small shim and usage of the JNI and they start working and became compactable with the android.

Since Marshmallow, "Toybox", a collection of command line utilities (mostly for use by apps, as Android doesn't provide a command line interface by default), replaced similar "Toolbox" collection.

Android also has another operating system, Trusty OS, which also contain, as a part of "Trusty" "software components supporting a Trusted Execution Environment (TEE) on mobile devices." "Trusty and the Trusty API are subject to change so its working and implementation depends on the changes and updates. Applications for the Trusty OS can be written in C/C++ (C++ support is limited), and they have access to a small C library. All Trusty applications are single-threaded and the multithreading in Trusty user-space currently is unsupported in it. Third-party application development is not supported in the current version of android operating system, and software running on the Operating system and processor for it, run the "DRM framework for protected content working. There are many other uses for a TEE such as mobile payments, secure banking, full-disk encryption, replay-protected persistent storage, multi-factor authentication, device reset protection, wireless display ("cast") of protected content, secure PIN and fingerprint processing, and even malware detection.


Applications
Home
Contacts
Phone
Browser
...
Application Framework
Activity Manager
Window Manager
Content Providers
View System
Notification Manager
Package Manager
Telephony Manager
Resource Manager
Location Manager
XMPP Service
Libraries
Surface Manager
Media Framework
SQLite
OpenGL|ES
FreeType
WebKit
SGL
SSL
libc
Android Runtime
Core Libraries
Dalvik Virtual Machine
Linux Kernel
Display Driver
Camera Driver
Bluetooth Driver
Flash Memory Driver
Binder (IPC) Driver
USB Driver
Keypad Driver
WiFi Driver
Audio Drivers
Power Management

3.3 Android Version History

Code name	Version number	Initial release date	API level	Support status
Alpha	1.0	September 23, 2008	1	Discontinued
Beta	1.1	February 9, 2009	2	Discontinued
Cupcake	1.5	April 27, 2009	3	Discontinued
Donut	1.6	September 15, 2009	4	Discontinued
Eclair	2.0 – 2.1	October 26, 2009	5–7	Discontinued
Froyo	2.2 – 2.2.3	May 20, 2010	8	Discontinued
Gingerbread	2.3 – 2.3.7	December 6, 2010	9–10	Discontinued
Honeycomb	3.0 – 3.2.6	February 22, 2011	11–13	Discontinued
Ice Cream Sandwich	4.0 – 4.0.4	October 18, 2011	14–15	Discontinued
Jelly Bean	4.1 – 4.3.1	July 9, 2012	16–18	Discontinued
KitKat	4.4 – 4.4.4	October 31, 2013	19	Discontinued
Lollipop	5.0 – 5.1.1	November 12, 2014	21–22	Discontinued

Code name	**Version number**	**Initial release date**	**API level**	**Support status**
Marshmallow	6.0 – 6.0.1	October 5, 2015	23	Supported
Nougat	**7.0 – 7.1.1**	**August 22, 2016**	**24–25**	Supported

3.4 Complex Docker Containers:

So what if you need a Docker container that does need to have a daemon running and listening on a network port for it? It's very great that you are able to run a container for the bash shell, but that's not particularly useful for you. What if you wanted to run a MySQL container, an SSH container, or a web server container?

To do something like that, you'll need to utilize a Docker file. A Docker file is a plain text file that tells Docker exactly how to construct a Docker image. Image contains all the dependencies which the container would require for the lunching and communication with the services and network. Here's an example Dockerfile:

```
FROM ubuntu:12.04
MAINTAINER Amratkumar "amratkumar94@gmail.com"
RUN echo "deb http://archive.ubuntu.com/ubuntu precise main u
niverse" > /etc/apt/sources.list
RUN apt-get update
RUN apt-get install -y nginx
EXPOSE 80
CMD ["nginx"]
```

The syntax for a Docker file is INSTRUCTION arguments, where "INSTRUCTION" refers to commands like FROM, RUN, or CMD. Here are a few quick notes for it:

- The FROM command is what enables Docker containers to be based on other Docker containers. In this case, the Docker container is based on the Ubuntu image; specifically, the Ubuntu 12.04 image.
- The RUN commands execute various commands within the new container.Use RUN instructions to install packages, modify files, etc.
- The EXPOSE command exposes a listening port on a Docker containers.
- The CMD instruction tells what command/daemon to run when the container launches in beginning.

If you want anything more complex than a simple bash shell, you're probably going to need to use a Docker file.

Once you have a working Docker file, you can create a Docker image from that Docker file using the Docker build command. Once you have a working image, you can then base other Docker containers on *that* image (using the FROM instruction in a subsequent Docker file).

3.5 Methodology for Containerization Of The Android.

Android Operating system is basically a Linux distribution. So to Run docker container first need to use base image of the Ubuntu operating system. After using Ubuntu then required to install all the dependencies which android required to run.

After installing dependencies now installing the emulator needs to install for the Graphical user interface which hold the android operating system and gives the complete access to all features over the screen.

Method Steps:

1. Base image
2. Android dependencies
3. Emulator
4. Applications
5. Testing

Researchers working on related project. Following are their Git reprosatories for android emulators and approaches base images and docker files.

1. Softsam/docker-android-emulator (its works using sdk approach)
2. Craigw9292/android-emulator (using No vnc approch)
3. Ksoichiro/android-emulator (android and emulator with gradle cache)
4. Beevelop/android
5. Tracer0tong/android-emulator
6. Thshaw/arc-welder (chrome app approach , app running directing and app)
7. Bitrise-docker/android (with preinstalled android tools)

3.6 Building Android in A Container

Building and maintaining a complex software system over time like Android (AOSP) can be a very daunting task, especially when the system has many dependencies. Perhaps can come accross with one or more these maintenance challenges.

- Need to fix a errors in an Android release which built a months or years ago, but the back traces are very useless because as they no longer have the symbols laying around anywhere.

Maybe have a copy of these symbols, but when go to build and test it, then build broke because of incompatibilities with host tools, or an important patch might be missing.

Or when upgraded the build host's OS, and now Android build get error.

Or someone broke the build recently, but you didn't notice it for weeks because haven't been doing full repo syncs frequently and builds, or you weren't building that particular product for regularly bases as not required by your software.

- This for the reason will be unable to reproduce a build when rolled way back , or maybe just feel guilty for not having a well-controlled release process, and finally decided that now's the time to automate clean, reproducible builds for release process with continuous integration.

A proper release process with continuous integration can solve all these problems we a non-profesional person face during not following the proper steps, and containers are an important part of the solution.

3.7 Why build Android in a container?

There are several healthy reasons for building Android in a container:

- Consolidate all the build dependencies in one place.
- Easily build on other machines, even those running different versions of the host OS (e.g., Android versions older than 2.3 need to be built on a 32-bit Ubuntu 10.04 host OS, which has reached its end of support).
- Expose the unknown dependencies, like uncommitted patches in a developer's source tree.
- Version-controlled, reproducible build environment offered by it (e.g. via Docker file, etc.), which is necessary for it e.g. continuous integration builds
- It's easy very to use.

3.8 How to build Android in a Docker container

AOSP Docker Container makes this easy:

```
$ mkdir -p ~/docker/lollipop
$ export AOSP_VOL=~/docker/lollipop
$ cd ~/docker
$ git clone https://github.com/kylemanna/docker-aosp.git
$ cd docker-aosp/tests
$ bash ./build-lollipop.sh
```

When it's done, you'll find the build output in the usual place, aosp/out/target/product. This is an excellent starting point, but probably want to build AOSP for a different product than the default specified in the script used so choose a different repo and branch by adding the following to Docker file like this

```
ENV TEST_URL ssh://joe@gerrit.example.com:29418/platform/manifest
ENV TEST_BRANCH android-5.0.1_r1    // product version required to build
```

3.9 Pitfalls and Tips

The example above works well when pulling the sources right from Google, but it's a bit painful if you're pulling from a repo that requires authentication.

3.9.1 Ssh Authentication Issues

Kyle's ssh_config turns off StrictHostKeyChecking, which reduces security. If your container is only going to be connecting to known set of machines, you can avoid reducing security while still allowing the container to connect to other machines by exposing it prior ~/.ssh/known_hosts with the right host keys:

1. Copy the known_hosts file into the directory containing the Dockerfile.

Remove the following line from Dockerfile:

```
ADD ssh_config /home/aosp/.ssh/config
```

1. Add the following lines:

```
ADD known_hosts /home/aosp/.ssh/known_hosts
RUN chown aosp:aosp /home/aosp/.ssh/known_hosts
RUN chmod 0600 /home/aosp/.ssh/known_hosts
```

3.9.2 Ssh Agent Forwarding Problems

If the aosp script detects that you're using ssh agent forwarding, it will helpfully bind-mount the ssh agent-forwarding socket inside the container. But if the aosp user inside the container don't have the same userid as the user owning the ssh agent which forwarding socket, authentication inside the container will get fail.

The safest way to get rid of this problem is probably to make the aosp userid inside the container match the userid of the user’s ssh agent forwarding socket. Can do this by changing the line in the Dockerfile that creates the aosp user just add --uid myuserid, as indicated below:

```
RUN useradd --create-home --uid 1234 aosp
```

However, this breaches the general principle of not making containers dependent on host configuration.

3.9.3 Mirror AOSP locally

Sometimes network outages or rate-limiting will cause repo sync to fail which occurs as error and you need to change your firewall security change. If that’s the case, or if you’re running repo sync frequently and want to be a good bandwidth by not flooding the upstream, simply create a local mirror for all the AOSP repos. Syncing against a local copy is very fast (if the machine holding the mirror is reasonably capable) and well worth the effort to get rid of repo sync failures.

3.9.4 Make the build clean

Building from scratch, no build objects are being written to the source tree. AOSP would builds cleanly, but often third-party ports sprinkle .o files hither and yon. Imagine .o files built with one toolchain hiding under the bushes from a previous build, and later getting linked against by a different toolchain or for a different product. Problems like that are not easy to discover and fix.

Often in third-party ports it’s kernel or u-boot builds that get to write to the source tree. Fix that by making O= to specify where build objects should be placed e.g.:

```
make -C u-boot O=$(abspath $(TARGET_OUT_INTERMEDIATES)/uboot)
```

If too hard to prevent build objects from being write to the source tree, an alternative is to make sure that all those build objects be deleted before a build proceed.

CHAPTER 4

ADB (Android Debug Bridge)

Android Debug Bridge (ADB) is a multipurpose command line tool that lets you communicate with the connected Android device or an emulator instance [2]. Any android device or emulator instance can be used to communicate with ADB. It exploits variety of device features, such as debugging an application or installing an application as a system app. One can access complete root file structure through ADB. It enables the user to access Unix shell to run a variety of commands on a connected device. Examples of such commands are listed at the end of this chapter It is a client-server program that includes three components:

- **Client** which executes on a developer machine command terminal sends commands to the ADB demon running on a device.
- **Daemon** that receive commands from client and executes them on a device. The daemon runs as a background process on instance.
- **Server** runs as a background process on developer’s machine. It solely responsible to manage communication between daemon & client.

Developer can find the ADB tool in *android_sdk*/*platform-tools*/.

If only testing is required, it can simply download and install ADB shell as Windows installer package or Linux tarball package.

4.1 How ADB works

When you initiate an ADB client, it first checks if there is an ADB server process already running or not. If there isn't any ADB server process already running, it starts the server process. When the server starts, it binds itself to local TCP port 5307 and listens for commands sent from ADB clients—all ADB clients use port 5307 to communicate with the ADB server.

The server establishes connections to all running device/emulator instances. It searches device/emulator instances by deep scanning over network on odd-numbered ports from the range 5555 to 5585. This range is used by devices/emulator. As soon as the server successfully searched an ADB daemon, it sets up a connection to that port. Note that each device/emulator instance acquires a pair of consecutive ports — an odd-numbered port for ADB connections and an even-numbered port for console terminal connections. For example:

Emulator 1, ADB: 5563	->	Emulator 1, console: 5562
Emulator 2, ADB: 5565	->	Emulator 2, console: 5564

and so on...

As shown, the emulator instance connected to ADB on port 5565 is the same as the instance whose console listens on port 5564.

Once the server establishes connections to all instances, developer can use ADB commands to access those instances. Since the server manages connections to instances and handles commands from multiple ADB clients, therefore developer can control any instance from any client.

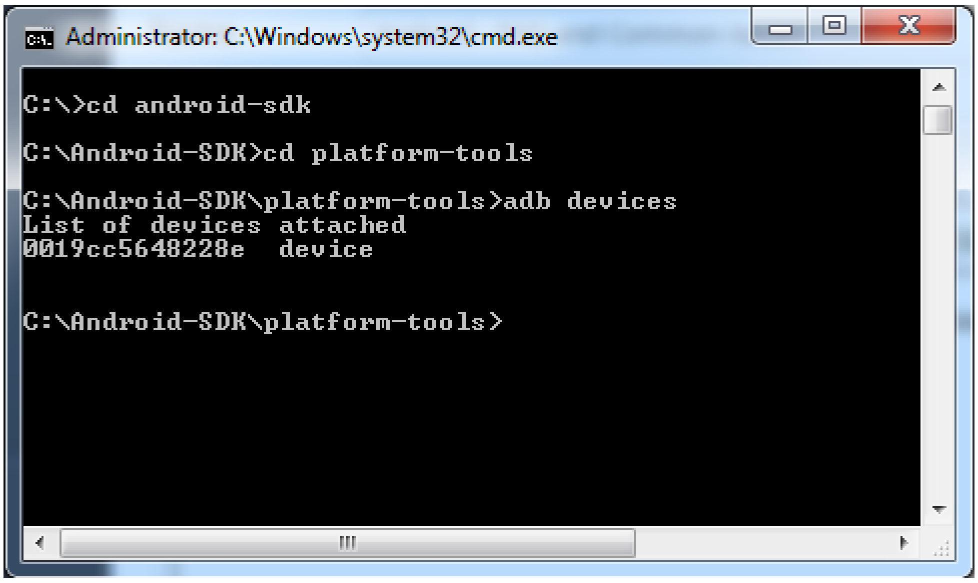
4.2 Enable ADB debugging on your device:

In order to use ADB with a device connected over USB, developer must check **USB debugging** option in the device system settings, under **Developer options**.

On Android 4.2.1 and higher, by default the Developer options screen is hidden. To make it observable, go to **Settings > About phone** and tap **Build number** several (7) times. When done with it. Tap return to land on previous screen. There you will find **Developer options** at the bottom.

On some handsets, the Developer options screen may be found in a different way.

Now developer can connect his device with USB. He can verify if device is connected by executing ADB devices from ADB shell environment or from the *android_sdk*/*platform-tools*/ directory. If connected, developer could see the device name listed as a "device."



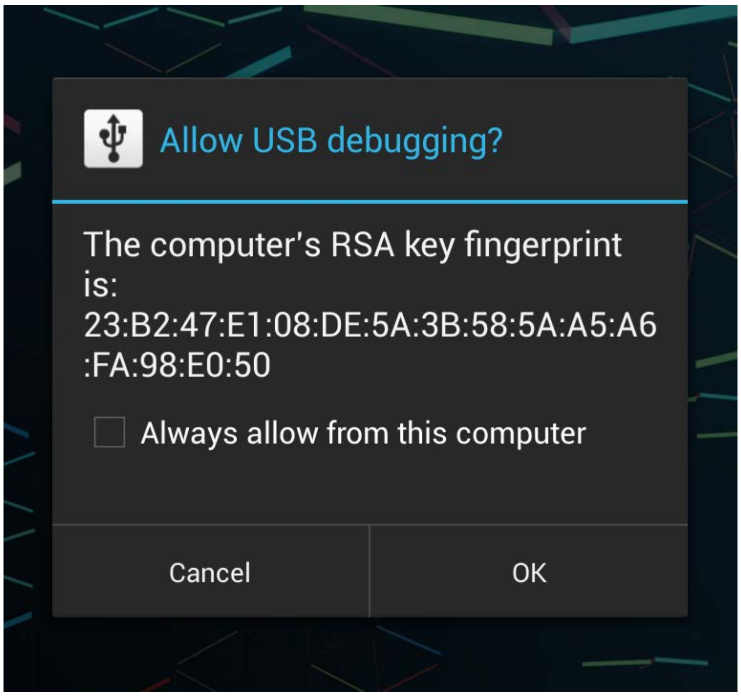
```
C:\>cd android-sdk

C:\Android-SDK>cd platform-tools

C:\Android-SDK\platform-tools>adb devices
List of devices attached
0019cc5648228e  device


C:\Android-SDK\platform-tools>
```

Note: When you connect a device running Android 4.2.2 or higher, the device shows a dialog box asking to accept an RSA key that allows debugging via this computer. This security mechanism shields user devices because it ensures that ADB commands cannot be executed unless you unlock the device and acknowledge the dialog.



Allow USB debugging?

The computer's RSA key fingerprint is:
23:B2:47:E1:08:DE:5A:3B:58:5A:A5:A6:FA:98:E0:50

☐ Always allow from this computer

4.3 Common ADB Commands

1. Adb Devices
2. Adb Push
3. Adb Pull
4. Adb reboot
5. FastBoot Devices

4.4 ADB shell

After successfully establishing ADB shell, you can access complete system files from root access. Linux basic commands like ls, cp -rf, mv, rm behave similarly in ADB shell. But by default the system is mounted in Read Only File system. To edit system files, one must have read-write and execute permission for a file. Therefore root, “/” must be remounted with read, write and execute permissions to edit system files as per the requirement of project approach.

CHAPTER 5

File paths in build scripts

The objective of this approach was to add Docker dependencies into Android OS by installing their modules separately into root mode. In a pre-installed device Android root was accessed and required modules were tried to be added. Considering the device as an active machine executing OS over a processor, we started building required dependencies over a device directly in a root mood.

5.1 Dependencies

The dependencies to install Docker from binaries were:

1. iptables version 1.4 or later
2. Git version 1.7 or later
3. procps (or similar provider of a “ps” executable)
4. XZ Utils 4.9 or later
5. apparmor
6. apt-utils
7. aufs-tools
8. automake
9. bash-completion
10. binutils-mingw-w64
11. bsdmainutils
12. btrfs-tools
13. build-essential
14. clang
15. cmake

16. createrepo
17. curl
18. dpkg-sig
19. gcc-mingw-w64
20. git
21. jq
22. less
23. libapparmor-dev
24. libcap-dev
25. libltdl-dev
26. libnl-3-dev
27. libprotobuf-c0-dev
28. libprotobuf-dev
29. libsqlite3-dev
30. libsystemd-journal-dev
31. libtool
32. libzfs-dev
33. mercurial
34. net-tools
35. pkg-config
36. protobuf-compiler
37. protobuf-c-compiler
38. python-dev
39. python-mock
40. python-pip

41. python-websocket
42. tar
43. ubuntu-zfs
44. vim
45. vim-common
46. zip

5.2 IP Tables:

IP tables served as a firewall and already present in Android. That version however only works with devices built with Linux kernel 2.6.29. Users of retail Android devices cannot access iptables binary. Even Android OS itself cannot access that binary. This is hard-coded in Android. Many devices also don't have iptables at all.

One can access with a rooted phone by using busybox and terminal to run "iptables -L" to list current tables. All one have to do is to root the phone and had Iptables on retail android. Once the device has confirmed iptables you can use the command line through your app to adjust the tables.

5.3 GIT:

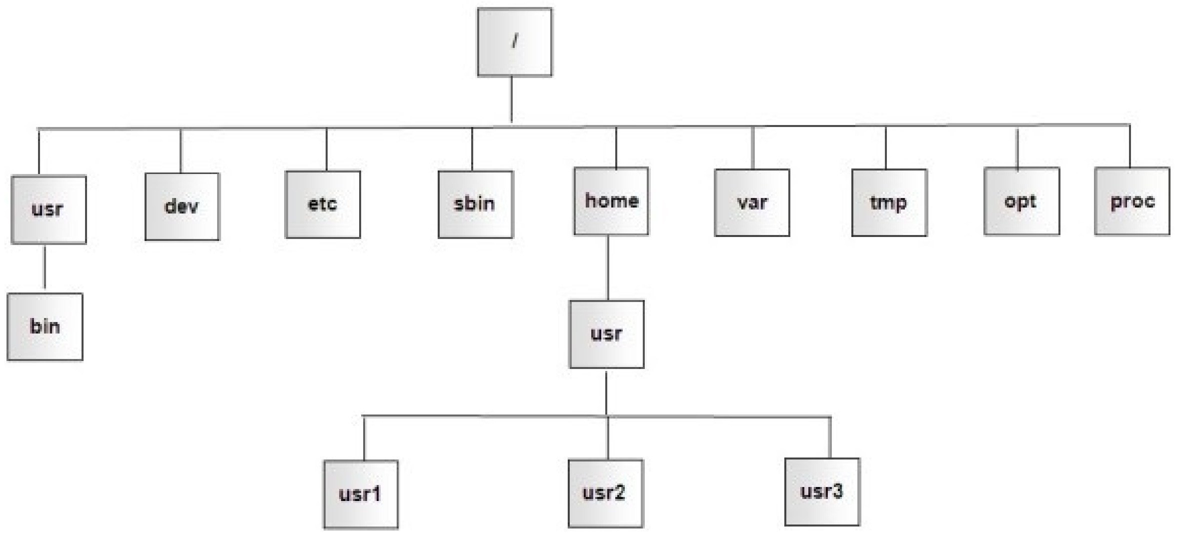
GIT requires debian packages listed above as a core dependency to be installed. Let’s start from the very basic like “tar.gz”. Git is downloaded as a tarball and extracted into a folder. After changing directory path to that folder one can easily locate configure file. It can be easily executed over a debian package but Android root have no such predefined environment variables like of Debian root. More over the configure file is based on scripts that set the values of variables in makefile scripts.

Example is adhered below:

```
./configure --prefix=/usr --bindir=/bin --with-root-prefix="" --enable-elf-shlibs --disable-libblkid --disable-libuuid --disable-uuidd --disable-fsck
```

5.4 File paths:

Here prefix is set to /usr - -bindir=/bin whereas in Android file structure there is no concept of /usr directory. /bin directory is present in /system directory. Thus a major change in file structure was there. The structure was edited accordingly in configure scripts to execute properly along-with the creation of dummy environment variables. Configure script was executed successfully but makefile failed while creating directories at anonymous locations which do not exist in Android file structure. Again it was to study the flow of complete makefile script and edit the paths accordingly. This labour was required for each package from a total of 46. Therefore this approach was halted and efforts were shifted towards another approach.



Linux OS Directory	**Android OS Directory**
/ ➢ bin ➢ boot • vmlinuz ➢ dev • hda • sda • st0 ➢ etc ➢ home • yxz • linux • tux ➢ lib • ld.so ➢ sbin ➢ root ➢ opt • kde • gnome ➢ proc ➢ mnt ➢ tmp ➢ usr • lib • local • sbin ➢ var ➢ srv • ftp • www	/ ➢ ls ➢ acct ➢ cache ➢ cal ➢ charger ➢ config ➢ data ➢ default.prop ➢ dev ➢ etc ➢ fota.rc ➢ init ➢ init.goldfish.rc ➢ init.lge.early.rc ➢ init.lge.rc ➢ init.rc ➢ mnt ➢ proc ➢ res ➢ root ➢ sbin ➢ sdcard ➢ sys ➢ system ➢ ueventd.rc ➢ vendor

CHAPTER 6

Building Android with Docker Engine

Android kernel was built along with Docker engine in a customized built Linux. These are the two main important things to run application on android operating system. For creating Linux there are number of steps involved.

6.1 Preparing the Host system:

Host system is defined as the system where we are building our linux kernel. The host system must have the following software with the minimum versions to work correctly. This should not be an issue for most modern Linux distributions.

- **Bash-3.2** (/bin/sh should be a symbolic or hard link to bash)
- **Binutils-2.17** (Versions greater than 2.27 are not recommended as they have not been tested)
- **Bison-2.3** (/usr/bin/yacc should be a link to bison or small script that executes bison)
- **Bzip2-1.0.4**
- **Coreutils-6.9**
- **Diffutils-2.8.1**
- **Findutils-4.2.31**
- **Gawk-4.0.1** (/usr/bin/awk should be a link to gawk)
- **GCC-4.7** including the C++ compiler, **g++** (Versions greater than 6.2.0 are not recommended as they have not been tested)
- **Glibc-2.11** (Versions greater than 2.24 are not recommended as they have not been tested)

- **Grep-2.5.1a**
- **Gzip-1.3.12**
- **Linux Kernel-2.6.32**
- **M4-1.4.10**
- **Make-3.81**
- **Patch-2.5.4**
- **Perl-5.8.8**
- **Sed-4.1.5**
- **Tar-1.22**
- **Texinfo-4.7**
- **Xz-5.0.0**

6.2 Problems:

- By default /bin/sh point to dash but to make the bash work properly a symlink was added to make it point to bash.
- By default /usr/bin/awk point to mawk but to make it work seamless it should be pointed to gawk.

6.2.1 Swap Partition:

Swap partition should be created so that RAM is not accessed more. The recommendation size of swap partition of swap should be twice of the RAM.

6.2.2 Creation of new Partition:

Create a new partition of size desirable. Mount that new partition.

6.3 Packages:

To make a Linux kernel, its environment application and packages should be downloaded. Total size of these packages are about 414 MB. So for this purpose create a new folder /sources and /tools download all the given packages in /sources of the new partition.

- Acl (2.2.52) - 380 KB:
- Attr (2.4.47) - 336 KB:
- Autoconf (2.69) - 1,186 KB:
- Automake (1.15) - 1,462 KB:
- Bash (4.3.30) - 7,7791 KB:
- Bc (1.06.95) - 288 KB:
- Binutils (2.27) - 25,488 KB:
- Bison (3.0.4) - 1,928 KB:
- Bzip2 (1.0.6) - 764 KB:
- Check (0.10.0) - 752 KB:
- Coreutils (8.25) - 5,591 KB:
- DejaGNU (1.6) - 512 KB:
- Diffutils (3.5) - 1,330 KB:
- Eudev (3.2) - 1,744 KB:
- E2fsprogs (1.43.1) - 6,846 KB:
- Expat (2.2.0) - 405 KB:
- Expect (5.45) - 614 KB:
- File (5.28) - 760 KB:
- Findutils (4.6.0) - 3,692 KB:

- Flex (2.6.1) - 816 KB:
- Gawk (4.1.3) - 2,258 KB:
- GCC (6.2.0) - 97,441 KB:
- GDBM (1.12) - 822 KB:
- Gettext (0.19.8.1) - 7,041 KB:
- Glibc (2.24) - 13,237 KB:
- GMP (6.1.1) - 1,898 KB:
- Gperf (3.0.4) - 960 KB:
- Grep (2.25) - 1,300 KB:
- Groff (1.22.3) - 4,091 KB:
- GRUB (2.02~beta3) - 5,890 KB:
- Gzip (1.8) - 712 KB:
- Iana-Etc (2.30) - 201 KB:
- Inetutils (1.9.4) - 1,333 KB:
- Intltool (0.51.0) - 159 KB:
- IPRoute2 (4.7.0) - 577 KB:
- Kbd (2.0.3) - 1,013 KB:
- Kmod (23) - 440 KB:
- Less (481) - 310 KB:
- LFS-Bootscripts (20150222) - 31 KB:
- Libcap (2.25) - 64 KB:
- Libpipeline (1.4.1) - 787 KB:
- Libtool (2.4.6) - 951 KB:
- Linux (4.7.2) - 88,290 KB:

- M4 (1.4.17) - 1,122 KB:
- Make (4.2.1) - 1,375 KB:
- Man-DB (2.7.5) - 1,471 KB:
- Man-pages (4.07) - 1,445 KB:
- MPC (1.0.3) - 655 KB:
- MPFR (3.1.4) - 1,096 KB:
- Ncurses (6.0) - 3,059 KB:
- Patch (2.7.5) - 711 KB:
- Perl (5.24.0) - 13,825 KB:
- Pkg-config (0.29.1) - 1,967 KB:
- Procps (3.3.12) - 826 KB:
- Psmisc (22.21) - 447 KB:
- Readline (6.3) - 2,411 KB:
- Sed (4.2.2) - 1,035 KB:
- Shadow (4.2.1) - 1,558 KB:
- Sysklogd (1.5.1) - 88 KB:
- Sysvinit (2.88dsf) - 108 KB:
- Tar (1.29) - 1,950 KB:
- Tcl (8.6.6) - 5,731 KB:
- Texinfo (6.1) - 4,416 KB:
- Time Zone Data (2016f) - 306 KB:
- Udev-lfs Tarball (udev-lfs-20140408) - 11 KB:
- Util-linux (2.28.1) - 4,061 KB:
- Vim (7.4) - 9,632 KB:

- XML::Parser (2.44) - 232 KB:
- Xz Utils (5.2.2) - 993 KB:
- Zlib (1.2.8) - 441 KB:
- Bash Upstream Fixes Patch - 15 KB:
- Bc Memory Leak Patch - 1.4 KB:
- Bzip2 Documentation Patch - 1.6 KB:
- Coreutils Internationalization Fixes Patch - 146 KB:
- Glibc FHS Patch - 2.8 KB:
- Kbd Backspace/Delete Fix Patch - 12 KB:
- Readline Upstream Fixes Patch - 8 KB:
- Sysvinit Consolidated Patch - 3.9 KB:

6.4 Setting up the environment and installing of packages:

Set up the environment using the bash_profile script. Install the packages downloaded with the correct headers referring to the folders created inside the new partition. After installing these packages enter the Chroot Environment to begin building and installing the final Linux system. As user root, run the following command to enter the realm that is only populated with the temporary tools:

```
chroot "$LFS" /tools/bin/env -i \
HOME=/root \
TERM="$TERM" \
PS1='\u:\w\$ ' \
PATH=/bin:/usr/bin:/sbin:/usr/sbin:/tools/bin \
/tools/bin/bash --login +h
```

6.4.1 LFS-Bootscripts-20150222

This package contains a set of scripts to start/stop the Linux system at bootup/shutdown. It also contains configuration files and procedures needed to customize the boot process.

6.4.2 Creation of /etc/fstab file

The /etc/fstab file is used by programs to determine where file systems are mounted by default, and in which order, and which must be checked (for integrity errors) prior to mounting.

6.5 Compiling Linux kernel:

The following are few situation where you may have to compile Kernel on your Linux system.

- To enable the experimental features that are not part of the default kernel system.
- To enable the support for a new hardware that is not currently supported by default kernel.
- To debug or modify the kernel
- To learn how kernel works, for this purpose you might want to explore the kernel source code, and compile it on your own.

Download the latest kernel version:

E.g:

```
# cd /usr/src/

# wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.9.3.tar.xz
```

a) Untar the Kernel Source

```
# tar -xvJf linux-3.9.3.tar.xz
```

b) Configuring the Kernel

The kernel consists of 3000 configuration options. To mold the kernel according to one’s need for specific hardware or software experimental it is configured through following commands.

```
# make menuconfig
```

The make menuconfig, will launch a text-based user interface with default configuration options. To launch this properly the “libncurses and libncurses-devel” packages for this command to work. It will give a pop up window with set of configuration setting related power management, Network support, processor types and features, device drivers, kernel hacking, security options etc. Some of the configuration settings that needs to be done are as following:

```
CONFIG_MMU=y

CONFIG_NEED_DMA_MAP_STATE=y

CONFIG_NEED_SG_DMA_LENGTH=y

CONFIG_GENERIC_ISA_DMA=y
```

CONFIG_GENERIC_BUG=y

CONFIG_GENERIC_HWEIGHT=y

Generic Driver Options --->

[ ] Support for uevent helper [CONFIG_UEVENT_HELPER]

[*] Maintain a devtmpfs filesystem to mount at /dev [CONFIG_DEVTMPFS]

After applying these configuration setting kernel can be make and installed.

REFERENCES:

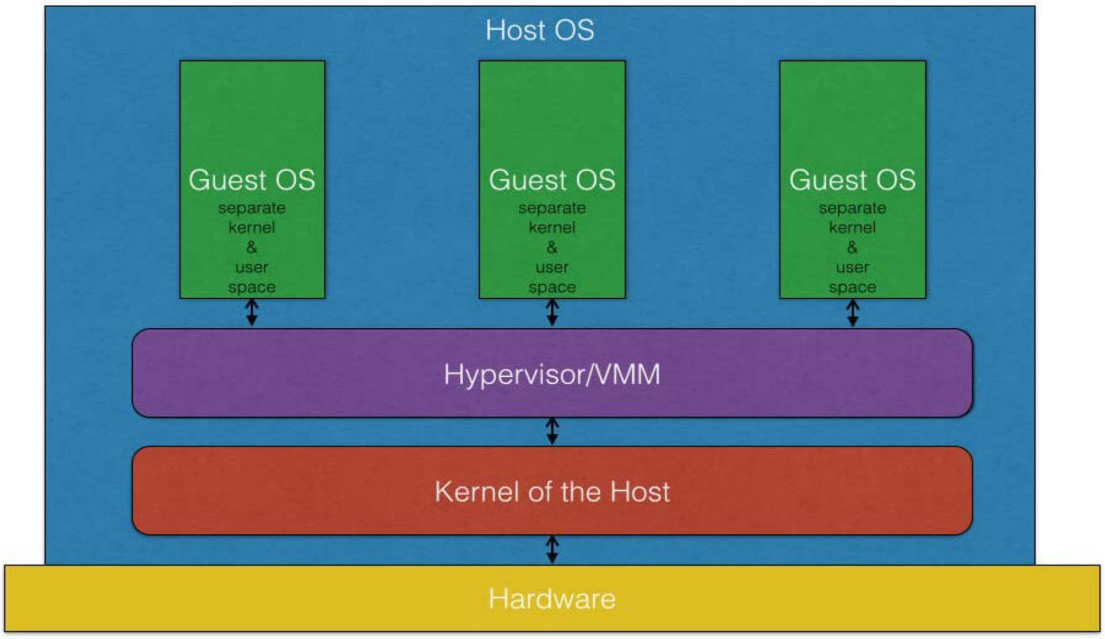
[1] https://sdgsystems.com/blog/building-android-container

[2] https://developer.android.com/studio/command-line/adb.html

APPENDIX:

What are virtual machines (VMs)?

A hypervisor, or a virtual machine monitor, is software, firmware, or hardware that creates and runs VMs. It’s what lies between the OS and hardware and is necessary to virtualize the server. Hypervisor based virtualization technologies have existed for a long time now. Since a hypervisor or full virtualization mechanism emulates the hardware, you can run any operating system on top of any other, Windows on Linux, or the other way around. Both the guest operating system and the host operating system run with their own kernel and the communication of the guest system with the actual hardware is done through an abstracted layer of the hypervisor**.**


Host OS
Guest OS
separate kernel & user space
Guest OS
separate kernel & user space
Guest OS
separate kernel & user space
Hypervisor/VMM
Kernel of the Host
Hardware

Hypervisor based Virtualization

As all communication is through the hypervisor so, it provides a high level of isolation and security between the guest and host This approach is also usually slower and incurs significant performance overhead due to the hardware emulation. To reduce this overhead, another level of virtualization called "operating system virtualization" or "container virtualization" was introduced which allows running multiple isolated user space instances on the same kernel.

What Is Docker?

So then what is Docker good at?

- Docker is great at building and sharing disk images with others through the Docker Index
- Docker is a manager for infrastructure (today's bindings are for Linux Containers, but future bindings including KVM, Hyper-V, Xen, etc.)
- Docker is a great image distribution model for server templates built with Configuration * Managers (like Chef, Puppet, SaltStack, etc)
- Docker uses btrfs (a copy-on-write filesystem) to keep track of filesystem diff's which can be committed and collaborated on with other users (like git)
- Docker has a central repository of disk images (public and private) that allow you to easily run different operating systems (Ubuntu, Centos, Fedora, even Gentoo)

What is LXC (Linux Containers)?

LXC (Linux Containers) is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) on a control host using a single Linux kernel.The Linux kernel provides Cgroups functionality that allows limitation and prioritization of resources (CPU, memory, block I/O, network, etc.) without the need for starting any virtual machines, and also namespace isolation functionality that allows complete isolation of an applications' view of the operating environment, including process trees, networking, user IDs and mounted file systems(AUFS), SELinux, and network namespaces, enabling Linux Containers have emerged as a key open source application packaging and delivery technology, combining lightweight application isolation with the flexibility of image-based deployment methods. Developers have rapidly embraced Linux Containers because they accelerate thee application deployment, while many PaaS platforms are built around Linux container technology, including OpenShift by Red Hat.secure multi-tenancy and reducing the potential for security exploits.

Building on the capabilities of the Linux Containers, application containers are deployed as software packages that include the application and all of its required runtime components. Benefits include:

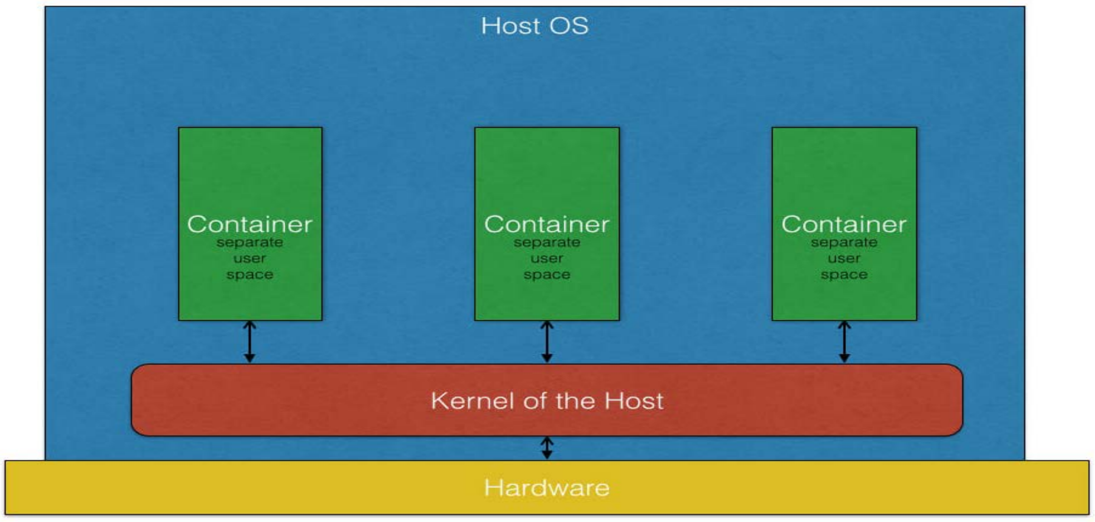
- **Instant portability**, deploy the certified application container across any certified container host
- **Minimal footprint**, avoid the overhead of virtual machine images which include a complete operating system
- **Simplified maintenance**, reduce the effort and risk by patching applications along with all of their dependencies
- **Lowered development costs**, develop, test and certify applications against a single runtime environment

What is Libcontainer?

Libcontainer is a library written in Go programming language that provides direct access for Docker to Linux container APIs. They provide a lightweight virtual environment that exploits the linux kernel virtualization feature specifically the container virtualization feature and isolates a set of processes and resources such as memory, CPU, disk.It enables containers to work with Linux namespaces, control groups, capabilities, AppArmor security profiles, network interfaces and firewalling rules from the host and any other containers. The isolation guarantees that any processes inside the container cannot see any processes or resources outside the container.

It doesn't rely on Linux userspace components such as LXC, libvirt, or systemd-nspawn Docker claims "This drastically reduces the number of moving parts, and insulates Docker from the side-effects introduced across versions and distributions of LXC."

Libcontainer, which is written natively in Google's Go, is also being ported into other languages. Microsoft *may* be porting it to ASP.NET. Parallels' libct, which includes libcontainer's functionality, has native C/C++ and Python bindings.


Host OS
Container separate user space
Container separate user space
Container separate user space
Kernel of the Host
Hardware

Operating System/Container Virtualization

VM VS DOCKER(Containers):

The difference between a container and a full-fledged VM is that all containers share the same kernel of the host system. This gives them the advantage of being very fast with almost 0 performance overhead compared with VMs. They also utilize the different computing resources better because of the shared kernel. However, like everything else, sharing the kernel also has its set of shortcomings.

- Type of containers that can be installed on the host should work with the kernel of the host. Hence, you cannot install a Windows container on a Linux host or vice-versa.
- Isolation and security -- the isolation between the host and the container is not as strong as hypervisor-based virtualization since all containers share the same kernel of the host and there have been cases in the past where a process in the container has managed to escape into the kernel space of the host.
- Size: VMs are very large which makes them impractical to store and transfer.
- Performance: running VMs consumes significant CPU and memory, which makes them impractical in many scenarios, for example local development of multi-tier applications, and large-scale deployment of cpu and memory-intensive applications on large numbers of machines.
- Portability: competing VM environments don't play well with each other. Although conversion tools do exist, they are limited and add even more overhead.

- Hardware-centric: VMs were designed with machine operators in mind, not software developers. As a result, they offer very limited tooling for what developers need most:

building, testing and running their software. For example, VMs offer no facilities for application versioning, monitoring, configuration, logging or service discovery.

OS Containers

OS containers are virtual environments that share the kernel of the host operating system but provide user space isolation. For all practical purposes, you can think of OS containers as VMs. You can install, configure and run different applications, libraries, etc., just as you would on any OS. Just as a VM, anything running inside a container can only see resources that have been assigned to that container.

OS containers are useful when you want to run a fleet of identical or different flavors of distros. Most of the times containers are created from templates or images that determine the structure and contents of the container. It thus allows you to create containers that have identical environments with the same package versions and configurations across all containers.

Container technologies like LXC, OpenVZ, Linux VServer, BSD Jails and Solaris zones are all suitable for creating OS containers.

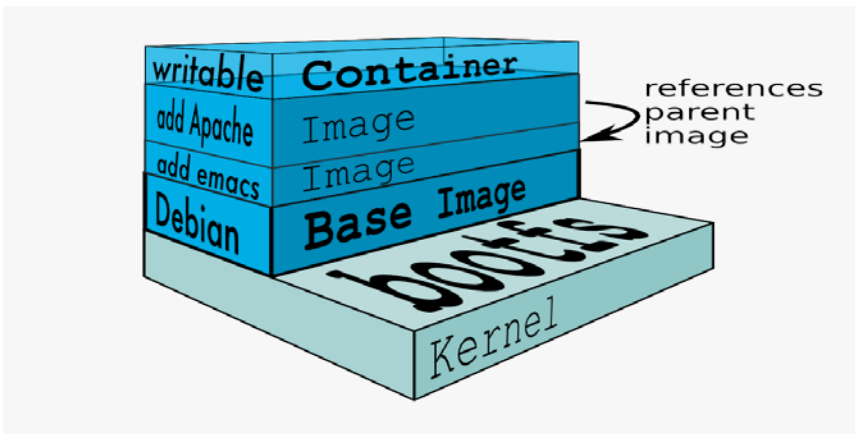
Application containers

While OS containers are designed to run multiple processes and services, application containers are designed to package and run a single service. Container technologies like Docker are examples of application containers. So even though they share the same kernel of the host there are subtle differences make them different, which I would like to talk about using the example of a Docker container:

Run a single service as a container

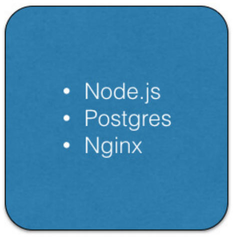
When a Docker container is launched, it runs a single process. This process is usually the one that runs your application when you create containers per application. This very different from the traditional OS containers where you have multiple services running on the same OS.

Layers of containers



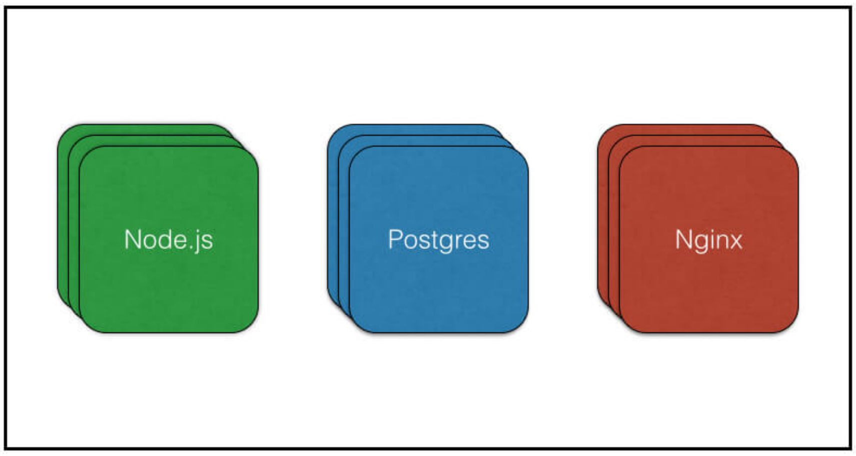
Any RUN commands you specify in the Dockerfile creates a new layer for the container. In the end when you run your container, Docker combines these layers and runs your containers. Layering helps Docker to reduce duplication and increases the re-use. This is very helpful when you want to create different containers for your components. You can start with a base image that is common for all the components and then just add layers that are specific to your component. Layering also helps when you want to rollback your changes as you can simply switch to the old layers, and there is almost no overhead involved in doing so.

In the simplest cases, using the traditional approach, one would put the database, the Node.js app and Nginx on the same machine.



Typical 3-tier architecture in the simplest sense

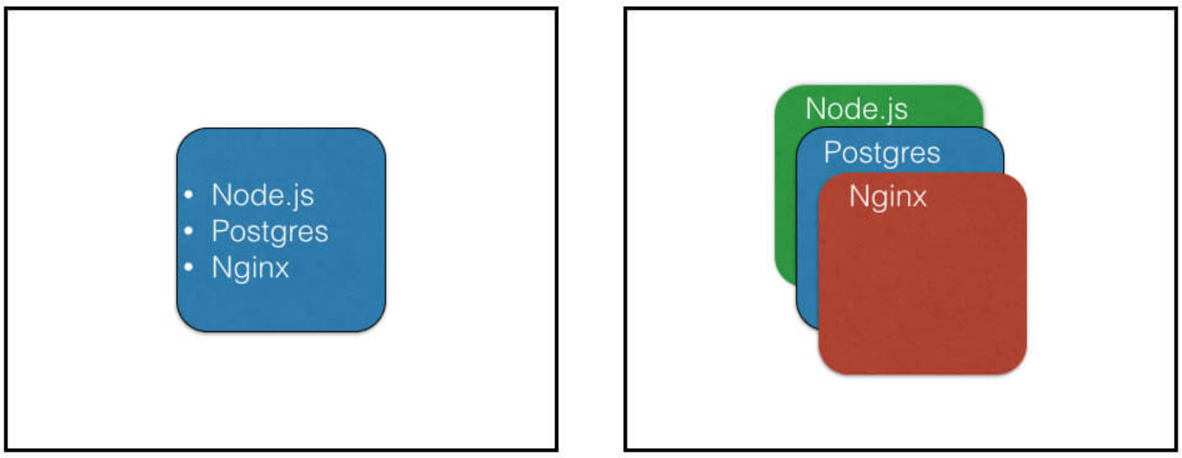
Deploying this architecture as Docker containers would involve building a container image for each of the tiers. You then deploy these images independently, creating containers of varying sizes and capacity according to your needs.



Typical 3-tier architecture using Docker containers

Summary

So in general when you want to package and distribute your application as components, application containers serve as a good resort. Whereas, if you just want an operating system in which you can install different libraries, languages, databases, etc., OS containers are better suited.



OS containers

- Meant to used as an OS - run multiple services
- No layered filesystems by default
- Built on cgroups, namespaces, native process resource isolation
- Examples - LXC, OpenVZ, Linux VServer, BSD Jails, Solaris Zones

App containers

- Meant to run for a single service
- Layered filesystems
- Built on top of OS container technologies
- Examples - Docker, Rocket