Columbia University

Unified Heterogeneous Networking Middleware

Spring 2015 Project Report

Lingyuan He / Dhruv Kuchhal
5-19-2015

CONTENTS

1 INTRODUCTION

The goal of the project, is to build a unified policy-based networking middleware for a Linux mobile node. The previous focus of the project is Linux host, but concentration is currently shifting to Android. Over the year, the project has two nicknames: Open Multiple Network Interface (OMNI) and Smart Internet (SINE). These names, together with the title of this report, often appear interchangeably throughout the project.

The middleware aims to support a seamless handover across multiple connections, including WiFi, Ethernet and Mobile Broadband (LTE), base on a custom policy that is written in a specific syntax, where the policy covers factors like location, cost, connection type, bandwidth and signal strength. Obviously, we usually have a combination of two network interface, namely Ethernet/WiFi for Linux and WiFi/Broadband for Android. However, it is possible to add Broadband to Linux using a LTE dongle.

The basic structure of the project is consist of four components: (1) a middleware overseeing interface and switching, based on a defined policy; and also take incoming traffic and wrap them as custom socket object (SOCK server), (2) a protocol (Media Independent Handover) that monitors network interfaces and listen to network events (3) protocol(s) (Host Identity Protocol and Mobile IPv6) to provide a universal presence in terms of network address (tag), and (4) code that handles switching (in C for Linux, and could reside in Java framework for Android).

The project started in 2012 in IRT lab. From 2012 to 2013, the basis of the project has been solidly founded. After a period of inactivity, in Spring 2015, the project was restarted and brought up to date.

Currently, the project mainly include a Linux implementation. It has multiple network interface (and multi-homing) support, socket wrapping / SOCK server, network handover functionality, security / web authentication support, and a simple policy engine (for now, it primarily use hand-input location to determine which interface to use).

Android development is relatively in early stage. There are porting scripts to port the Linux components to Android, as well as documentation on Android Framework.

In this report, we will first cover software architecture, code repository structure and development environment, before we continue on to individual topics in Android and Linux.

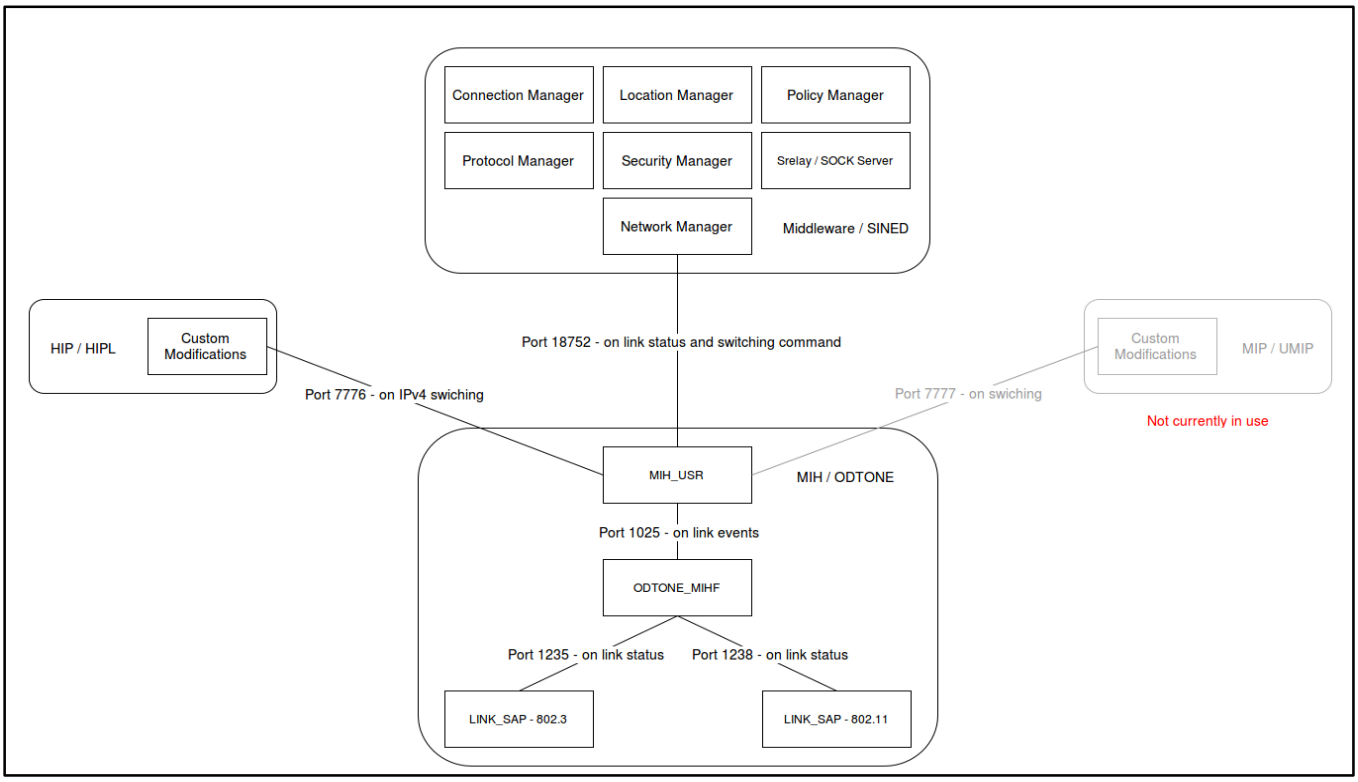
2 ARCHITECTURE

2.1 LINUX

On the following page is an illustration of the software components and their socket communication via UDP ports, on Linux:

Summary:

- Middleware is the center of the action that support the core functionalities (policy analysis, interface decision, network security/authentication etc.).
- MIH oversees network interface, answers interface inquiry (from Network Manager), and pass interface switch command to HIP (and MIP in the past) to conduct switch.



- HIP (and MIP in the past) are our protocols to handle multi-homing, by supplying a unique/stable presence for each host (HIT for HIP and IPv6 for MIP). Currently, custom modification in HIP is applied when compile in Linux, which handles IPv4 interface switch in a separate process forked from the main process.
- MIP is currently disabled due to lack of update and related kernel patches, so an alternative implementation is needed.

Note:

- Although normally there will be two interfaces, on Linux or Android. There can be multiple link_sap configured on a single host, so by adding link_sap configuration and execute a third link_sap instance, a host can monitor a third network interface.
- The reason we put IPv4 switching in HIP instead of MIH is to preserve universality for MIH to work on both Android and Linux host. In this way, we can instead use Android Framework modification plus clean HIP to work with MIH and middleware.

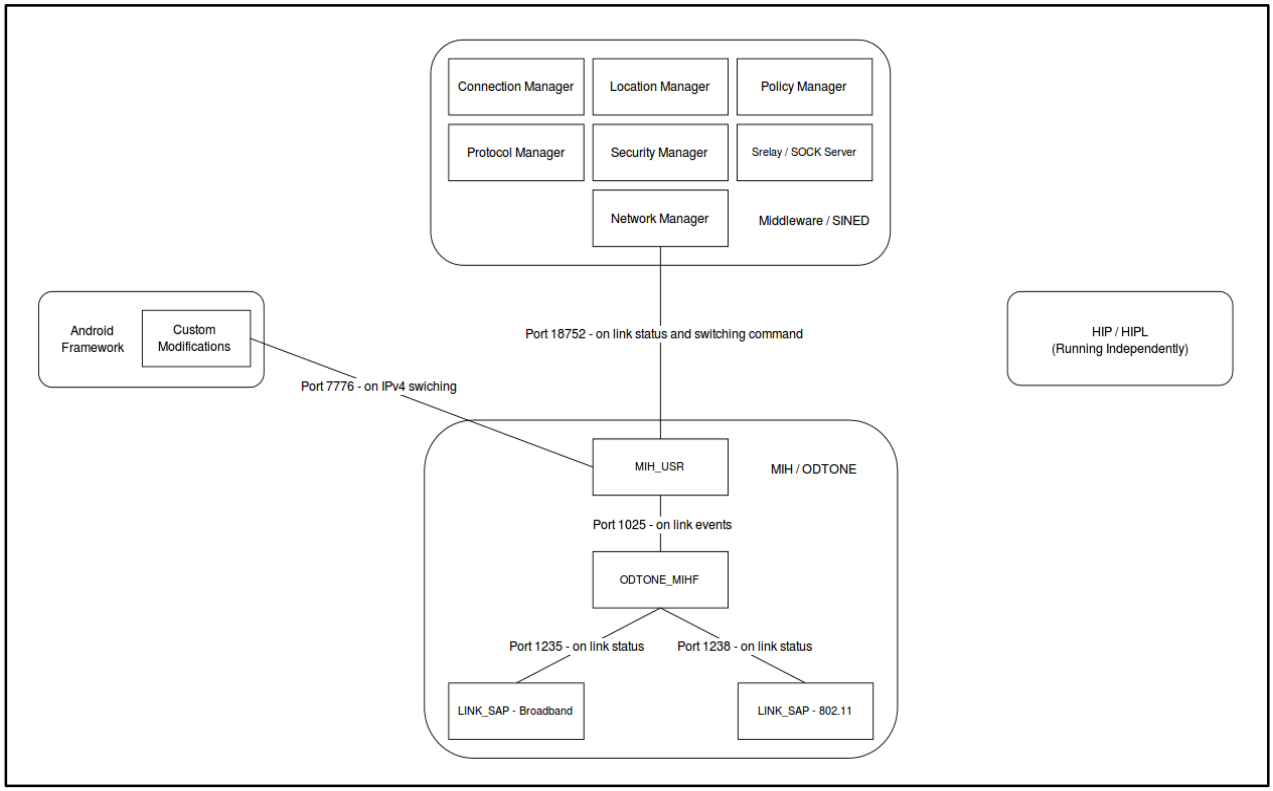
Previous architecture documentation see:
https://wiki.cs.columbia.edu/display/sine/Implementation+Design

2.2 Android (Proposed)

We don't currently have a working Android architecture, but we have a proposed architecture that we believe will be well suited:

Summary:

- Middleware and MIH could largely (or entirely) stay the same.
- Use Android Framework's native methods to perform switching, leaving HIP running independently.



Note that a MIP implementation may be be able to work in Android.

Alternatively, the method of switching in Linux could work in Android (as the ip command is provided in Android, and many missing Linux utilities can be installed using busybox), but the way it works may not be preferable comparing to the method in Android framework. The method on Linux is untested as for now on Android.

3 PROTOCOLS

3.1 MIH - ODTONE

Media Independent Handover (802.21), we use ODTONE / Open 802.21 as our implementation.

MIH is used as a mean to monitor network interface event (link up and down) to help middleware to make decision on switching. In our project, we modified the mih_usr code significantly, so it can be queried on interface status by the middleware, and also issue command to switch (for now to our modification in HIP).

ODTONE website and mailing list:

http://atnog.github.io/ODTONE/

https://atnog.av.it.pt/cgi-bin/mailman/listinfo

Documentation:

http://atnog.github.io/ODTONE/documentation/index.html

Components:

ODTONE consists of three components, executables in protocol/mih/odtone/odtone-0.6/dist

- link_sap: three link_sap's monitor ethernet/wireless/LTE (LTE is not used for Linux, Ethernet is not used for Android)
- odtone-mihf: main ODTONE program
- mih_usr: the user program to listen to events, heavily modified to provide interface monitoring and switch issuing, see protocol/mih/mih_usr.c
- A shell script (protocol/mih/mih) is used to streamline the execution of these components.

3.2 HIP - HIP FOR LINUX

Host Identity Protocol, we use HIP for Linux as our implementation

HIP is used to provide a universal address presence (multi-homing) of a mobile node by supplying a global unique address (Host Identity Tag, HIT). HIT is similar to IPv6, and it is in fact globally unique, but without a DNS-like service to look up HIT, we always need an initial IP mapping before using HIT as an network address to work (e.g. streaming video).

In our project, we modified HIP and added a separate process on Linux. This piece of code listens to command issued by mih_usr in MIH, and perform IPv4 switching. See below for details.

HIPL website and mailing lists:

http://infrahip.hiit.fi/

http://www.freelists.org/list/hipl-users

http://www.freelists.org/list/hipl-dev

How HIP works:

http://infrahip.hiit.fi/index.php?index=how

http://infrahip.hiit.fi/data/archive/InfraHIP_overview_slides.pdf

How to check your HIT:

When hipd is running, execute

```
$ hipconf daemon get hi default
```

Or look for dummy0 interface IPv6 address (global one) in:

```
$ ifconfig
```

How to add an initial mapping of HIT to IP (to get the "server" to initiate connection to "client"):

When hipd is running, execute:

```
$ hipconf daemon add map <PEER_HIT> <PEER_IP>
```

Linux modification of HIP:

A separate forked process is when HIP is used on Linux host, this process listens for command from mih_usr (at port 7776) to switch IPv4 interface. This part of modification include patches and hip_omni.c/.h in protocol/hip. This modification is currently only applied on Linux.

Unlike old version, there is no several routing table. We keep all interface alive, so all the subnet routes stay the same, so the routing table will be the same except the default route, which we will modify.

The modification only supports interface switching (e.g. "pref wlan0"), encryption mode change is not supported ("eon", "eoff").

The reason why we choose to include this piece of code with HIP is to keep middleware universal in case Android and Linux will use different native way to switch interface in the future.

3.3 MIPv6 - UMIP

MIPv6 is an alternative way of providing multi-homing (unique presence).

UMIP was the implementation choice, but it is not currently in use, reason:

- UMIP has not been updated, repository unreachable: http://umip.org/
- Kernel patch required is unavailable for newer kernel (www.mobile-ipv6.org unavailable, and other archives found not up-to-date)

Viable alternative to look into:

http://www.mip6d-ng.net/

3.4 MIDDLEWARE

Middleware largely remain unchanged, other than stability changes and compile warnings fixes.

Some features not implemented yet are commented out in the code, namely:

- Code about interface cost and bandwidth are commented out, around 410th-440th line in PolicyModel.cpp:
  https://github.com/lingyuan-he/OMNI-Columbia-IRT/blob/master/middleware/policyMgr/PolicyModel.cpp
- In network manager, 97th to 126th line, where bandwidth and cost are fetched from MIH. Currently mih_usr does not respond to these request.
  https://github.com/lingyuan-he/OMNI-Columbia-IRT/blob/master/middleware/networkMgr/nm.cpp

4 DEVELOPMENT ENVIRONMENT

4.1 LINUX DEVELOPMENT

- Operating System: a mixture of Ubuntu 12.04 64-bit and 32-bit environments, demo is performed on 32-bit machines in the lab.
- Compiler: gcc/g++ 4.6.3 that come with Ubuntu 12.04
- Development Dependencies: fetched through **`apt-get`** in configure script
- Note:
    - Upgrading to Ubuntu 14.04 may cause compiling warnings and errors in some of the older software components
    - Running some of the components on a 64-bit machine may require 32-bit libs on Ubuntu 12.04, which is not automatically installed:
      `$ sudo apt-get install ia32-libs`

4.2 ANDROID DEVELOPMENT

Android development we did focused on cross-compilation and executable testing. Native interface switching has not been implemented, and the Linux method of switching has not been fully tested.

- Host: A Nexus 7 Device with Stock Android 4.4 Kitkat on a custom Kernel version 3.4
- Toolchain: arm-linux-androideabi-4.9 provided by Android NDK 10d (32-bit) version for cross-compile, and arm-gcc-eabi-4.6 from Google repository for Kernel compilation
- Note:
    - Dependencies on Linux can be obtained by **`apt-get`**, simply run **android_porting/dependency.sh**

5 CODE REPOSITORY

Code repository is located at:

https://github.com/lingyuan-he/OMNI-Columbia-IRT

Relevant folders:

- android_porting: android porting scripts and files
    - install: installation folder of cross compilation
    - kernel: instruction and utility on custom kernel of Android
    - patches: the patches to applied to libraries and NDK header for cross-compilation
    - test: place for test programs, now only include a short static query test on MIH
    - work: working folder that will be filled with software downloads and builds
- middleware: control middleware and policy engine
    - connectionMgr: manage a table of different connection on the host
    - locationMgr: location manager module
    - policyMgr: policy engine module
    - networkMgr: manage network interfaces
    - protocolMgr: socket wrapper
    - securityMgr: authentication module

    - srelay: socket relay / SOCKS server, and middleware main function
    - locationSwitch: location switching utility locsw, used to manually assign location
- protocols: protocol components
    - hip: Host Identity Protocol (HIP for Linux) and related modifications
    - mih: Media Independent Handover Protocol (ODTONE) and related modifications
    - umip: old Mobile IPv6 protocol (UMIP), not in use now due to lack of kernel patch and new software update

Legacy folders:

- include
- experiments
- libsine
- sined
- ssm
- middleware/sine

See old documentation: https://wiki.cs.columbia.edu/display/sine/Code+Repository, for detailed information on the legacy folders.

Previous code repository can be found at: https://github.com/columbia-irt/manhattan.

6 INSTALLATION

6.1 LINUX

```
$ ./configure
```

- Need sudo to install required libraries
- Will take a while to complete in the first run

```
$ make
```

```
$ sudo make install
```

After installation, you have those action available:

- Start HIPL
  `$ sudo hipd`
- Start ODTONE
  `$ sudo mih`
- Start location swither
  `$ locsw`
- Start middleware
  `$ sudo sined -f`

Note that you will always need MIH running first to launch middleware, otherwise middleware will run briefly and halt. It is not really important on when HIPL should be launched.

6.2 ANDROID

While there is no working demo or example on Android, we have guide to device preparation, kernel configuration and cross-compilation to start with.

Prepare Device

Before continue to cross-compile, the device need to be prepared:

- Get adb and fastboot working with your device, here are some reference link if you are unfamiliar with the process:
  http://developer.android.com/tools/help/adb.html#Enabling

  http://developer.android.com/tools/device.html
- Root the device, an example tutorial: http://www.ibtimes.co.uk/root-nexus-5-nexus-7-2012-wi-fi-android-5-1-lmy47d-lollipop-firmware-1491474
  (You also need to enable **`su`** in adb shell)
- Install busybox to have a full set of utilities, such as **`awk`** and **`patch`**:
  https://play.google.com/store/apps/details?id=stericson.busybox&hl=en

  (Note that there might exist a complete Android version of BusyBox, which can work better)

Kernel Compilation

You will need your device to boot with a custom kernel (and kernel modules) in order to experiment with HIP (or other component in the future).

Instructions on how to prepare and compile a custom kernel, and information on booting Android with custom kernel, can all be found in 'Android Kernel Compilation' below.

Cross Compilation

Cross-compilation is done entirely with script, using toolchain **arm-linux-androideabi** from NDK. We will cover the work of cross-compilation in details later in the report.

On Linux:

```
$ cd android_porting
```

```
$ ./dependency.sh
```

This will install standalone adb and fastboot, and also libncurses needed by kernel compilation.

```
$ ./compile
```

This script will download Google NDK, build toolchain inside toolchain folder, and use the toolchain to cross-compile all software components (except the switching support on Linux). It will take a period of time to cross-compile everything.

Before pushing files, change interface MAC addresses inside **install/odtone/lte/link_sap.conf** and **install/odtone/802_11/link_sap.conf** to addresses of LTE and WiFi interface on Android device.

Note that MAC addresses can be checked on Android using **`netcfg`**.

Then push all relevant files. The following script will push the executables onto android on rightful place

```
$ ./push.sh
```

Now you should have the same contents of odtone/hipl/sine folders on Android in **/data/misc/install**, with a few exception that configuration files being uploaded to somewhere else.

7 LINUX DEMO CONSTRUCTION

Below we will cover a demo based on Linux: **Video streaming between two hosts using VLC**

The setup:

server is located on Wireless, client on Ethernet and wireless, they are connected using HIT identifier (initial HIT-IP mapping needed at server side). Client moves between eth0 and wlan0 according to the location policy (triggered by a switcher), and the video streaming will not disconnect (continue with a delay).

7.1.1.1 Server: on Ethernet (eth0)

Step 0: install HIPL

You can either install the whole stack of the middleware, or simply install HIP for Linux, since it is all we need.

Step 1: run HIPL

```
$ sudo hipd
```

Step 2: initial IP mapping of the client

```
$ hipconf daemon add map <client HIP identifier - HIT> <client wireless IPv4 address>
```

We use wireless (wlan0) address here since our control middleware will default to wlan0 (location 'columbia') when starting up.

Note that initial mapping could also be done by adding entry to **/usr/local/etc/hip/host,** but since IP is dynamic, we do it manually each time. On the other hand, HIT should stay the same for one host all the time.

Step 3: stream video to the client

In VLC, stream a video to RTP via port 5000:

rtp://[<client HIT>]:5000

Follow the wiki here to do it:

https://wiki.videolan.org/Documentation:Streaming_HowTo_New/

In short, you have to follow this sequence:

Media Menu -> Stream -> Select File -> Select Protocol -> Input Address and Port -> Stream

7.1.1.2 Client: on both eth0 and wlan0

Step 0: install the whole stack

This is covered in 'Installation', in short:

```
$ ./configure
$ make
$ sudo make install
```

Step 1: setup wireshark

Open two wireshark instances to monitor wlan0 and eth0.

```
$ sudo wireshark
```

For ease of discriminating UDP packets (RTP uses UDP), on both wireshark instances, using View -> Coloring Rules to color UDP to pink.

Step 2: start the server, as we stated above

```
$ sudo kill_sined
```

Just in case the previous one is still running.

Step 3: start MIH in terminal tab 1

```
$ sudo mih
```

Step 4: start HIPL in terminal tab 2

```
$ sudo hipd
```

Step 5: start location switcher in terminal tab 3

```
$ locsw
```

It will start with default location 'columbia' (wlan0).

Step 6: starts control middleware in terminal tab 4

```
$ sudo sined -f
```

Wait a few moment for everything to stabilize. And you will be on wlan0 if you check:

```
$ netstat -ren.
```

Step 7: use VLC to play video streaming

In VLC, play network video stream from:

rtp://[client's own HIT]:5000

You can do that at:

Media Menu -> Open Network Stream

Now you should have the video playing.

Step 8: use locsw to switch location and interface

(1) switch to home (eth0):

At terminal tab 3, type "home", which means eth0 is preferred.

Video will lag for around 20 seconds, then it resumes on eth0.

To verify this, on wireshark you will find now a lot of pink UDP packets (carrying RTP payload) flowing on eth0. Alternatively, **`$ netstat -ren`** can also help you to find the default interface.

(1) switch to columbia (wlan0):

At terminal tab 3, type "columbia", which means wlan0 is preferred.

Video will lag for around 20 seconds, then it resumes on wlan0.

To verify this, on wireshark you will find now a lot of pink UDP packets (carrying RTP payload) flowing on wlan0. Alternatively, **`$ netstat -ren`** can also help you to find the default interface

8 Linux Interface Switching

A reliable and efficient way of switching network interface on Linux has been the top topic for this project in Spring 2015, this section is a summary of all the experiments around this issue.

Current solution is described in Method 1 with sub-solution 1 on obtaining router's address, all other methods are listed here for reference, with red font indicating failure reason.

8.1 Method 0: Bring Interface Up and Down

We call it method 0 because it is obvious.

```
$ sudo ifconfig <interface> <up/down>
```

Of course this does the job, but with two significant shortcomings:

- You cannot continue to monitor an interface via MIH if it is turned off
- Bring an interface online takes seconds

8.2 Method 1: Modify Routing Table Default Route - Current Solution

Syntax (in terminal):

```
$ sudo ip route replace default via <gateway ip> dev <interface name>
```

An example:

```
$ netstat -rne
Kernel IP routing table
Destination     Gateway       Genmask       Flags Metric Ref    Use Iface
0.0.0.0         128.59.16.1   0.0.0.0       UG    0      0        0 eth0
1.0.0.0         0.0.0.0       255.255.255.0   U   0      0        0 dummy0
```

```
128.59.16.0     0.0.0.0       255.255.248.0   U   1      0      0 eth0
160.39.192.0    0.0.0.0       255.255.254.0   U   2      0      0 wlan0
169.254.0.0     0.0.0.0       255.255.0.0   U     1000   0      0 eth0
```

```
$ sudo ip route replace default via 160.39.192.2 dev wlan0
```

```
$ netstat -rne
Kernel IP routing table
Destination     Gateway       Genmask       Flags Metric Ref    Use Iface
0.0.0.0         160.39.192.2  0.0.0.0       UG    0      0      0 wlan0
1.0.0.0         0.0.0.0       255.255.255.0   U   0      0      0 dummy0
128.59.16.0     0.0.0.0       255.255.248.0   U   1      0      0 eth0
160.39.192.0    0.0.0.0       255.255.254.0   U   2      0      0 wlan0
169.254.0.0     0.0.0.0       255.255.0.0   U     1000   0      0 eth0
```

This method easily redirect all IP traffic through a new device by using a new default routing route, react speed is also good, but how to get gateway/router IP is the main issue here.

How to get gateway/router IP address?

1. **Current solution: one-hop ping**
   The workaround is to (1) switch default gateway using only interface name (2) do a one-hop ping to fetch router's address (3) change default gateway again, supplying the gateway IP.

   This solution works well for now at both home and lab setup, but will be troublesome if the first-hop router does not reply to such ping (similar to traceroute where multiple routers may not respond).

   If we leave gateway IP as 0.0.0.0, the host will issue an ARP request for every IP we want to reach (and answers will always be our router address). Technically the network still works, but rather in an inefficient way.

   Example (ping a google server):

   ```
   $ sudo ip route replace default dev <wlan0/eth0>
   $ ping -t 1 -c 1 74.125.226.78
   PING 74.125.226.78 (74.125.226.78) 56(84) bytes of data.
   From 192.168.1.1 icmp_seq=1 Time to live exceeded
   $ sudo ip route replace default via 192.168.1.1 dev <wlan0/eth0>
   ```

2. **Using local ARP result**
   Not working, reason: arp result may contain more than the router's address in a corporate/school network setting, since the machine may not behind a typical DHCP router and thus have several direct peers.

   On home router:

   ```
   $ sudo ip route replace default dev <wlan0/eth0>
   ```

```
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   c8:d7:19:33:52:95   C                     wlan0
```

But in the lab, there are several result for each interface, we have no one to determine which is the router:

```
$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
160.39.192.2             ether   b0:fa:eb:63:00:3f   C                     wlan0
74.125.224.48            ether   00:00:0c:07:ac:00   C                     wlan0
128.59.16.38             ether   00:50:56:82:5d:ef   C                     eth0
74.125.224.48            ether   d0:c7:89:a9:c7:40   C                     eth0
160.39.192.3             ether   7c:ad:74:68:5e:8f   C                     wlan0
128.59.21.104            ether   00:26:ab:bb:c8:24   C                     eth0
128.59.16.1              ether   d0:c7:89:a9:c7:40   C                     eth0
```

3. **Using local DHCP lease**

Not working, reason: not all machine is behind DHCP (no lease), and the lease may not be active.

On home router:

```
$ sudo cat /var/lib/dhcp/dhclient.leases
lease {
  interface "wlan0";
  fixed-address 192.168.1.115;
  server-name "ecosystem.home.cisco.com";
  option subnet-mask 255.255.255.0;
  option routers 192.168.1.1;
  option dhcp-lease-time 86400;
  option dhcp-message-type 5;
  option domain-name-servers 192.168.1.1;
  option dhcp-server-identifier 192.168.1.1;
  option dhcp-renewal-time 43200;
  option broadcast-address 192.168.1.255;
  option dhcp-rebinding-time 75600;
  option host-name "lingyuan-HP";
  option domain-name "nyc.rr.com";
  renew 5 2015/04/10 10:51:15;
  rebind 5 2015/04/10 22:21:30;
  expire 6 2015/04/11 01:21:30;
```

```
}
```

But in the lab, the lease has long expired, or there is no leases:

One lab machine:

```
$ sudo cat /var/lib/dhcp/dhclient.leases
lease {
  interface "wlan0";
  fixed-address 192.168.1.66;
  option subnet-mask 255.255.255.0;
  option dhcp-lease-time 7200;
  option routers 192.168.1.1;
  option dhcp-message-type 5;
  option dhcp-server-identifier 192.168.1.1;
  option domain-name-servers 128.59.59.70;
  option broadcast-address 192.168.1.255;
  option domain-name "mip6test.com";
  renew 3 2013/04/24 19:01:28;
  rebind 3 2013/04/24 19:57:47;
  expire 3 2013/04/24 20:12:47;
}
```

On another lab machine:

```
$ sudo cat /var/lib/dhcp/dhclient.leases
(nothing...)
```

8.3 METHOD 2: MODIFY INTERFACE METRIC

The basic thinking behind this method is to: (1) choose the order of preference of the interfaces (2) refresh or restart network service to let it pick the new preference.

Interface Preference

By modifying interface metric, we can change the "order" of the interface used. In routing table, the default interface (gateway) comes with metric 0, with others larger that 0.

A handy tool to change interface metric is ifmetric:

http://0pointer.de/lennart/projects/ifmetric/

There is a bug in ifmetric, download the source, change line 47 in src/nlrequest.c to "char replybuf[4096];" before make and sudo make install.

Example of ifmetric:

```
$ netstat -ren
```

```
Kernel IP routing table
Destination     Gateway       Genmask       Flags Metric Ref    Use Iface
0.0.0.0         128.59.16.1   0.0.0.0       UG     0      0      0 eth0
1.0.0.0         0.0.0.0       255.255.255.0   U    0      0      0 dummy0
128.59.16.0     0.0.0.0       255.255.248.0   U    1      0      0 eth0
160.39.192.0    0.0.0.0       255.255.254.0   U    2      0      0 wlan0
169.254.0.0     0.0.0.0       255.255.0.0   U      1000   0      0 eth0
$ sudo ifmetric wlan0 0
$ sudo ifmetric eth0 10
$ netstat -ren
Kernel IP routing table
Destination     Gateway       Genmask       Flags Metric Ref    Use Iface
0.0.0.0         128.59.16.1   0.0.0.0       UG     10     0      0 eth0
128.59.16.0     0.0.0.0       255.255.248.0   U    10     0      0 eth0
160.39.192.0    0.0.0.0       255.255.254.0   U    0      0      0 wlan0
169.254.0.0     0.0.0.0       255.255.0.0   U      10     0      0 eth0
```

One thing noted is that after changing metrics, network often begin to act weirdly and result in losing connection. It will go back to normal if you restore the old metrics.

We refer to the previous way the soft/temporary way to set interface metric.

Alternatively, by assigning an interface up event in /etc/network/interfaces, we can set a hard/long-term metric.

```
$ cat /etc/network/interfaces
auto lo
iface lo inet loopback
iface eth0 inet dhcp
  up ifmetric eth0 10
iface wlan0 inet dhcp
  up ifmetric wlan0 0z
```

Obviously, we need restart (not only refresh) network service afterwards to put these into effect.

And it is also not a good practice to assume that the configuration file is empty (default only with the first two lines) and write it on the fly.

Put the Method into Trial

1. **Soft/temporary ifmetric setup + restarting network daemon**
   Restarting network daemon is deprecated and does not re-pick interface (nor changing the metrics):

   (Assume changing metrics like we showed above - wlan0 0, eth0 10)

```
$ sudo /etc/init.d/networking restart

 * Running /etc/init.d/networking restart is deprecated because it may not enable
again some interfaces

 * Reconfiguring network interfaces...                                    [ OK ]
```

(Routing table stays the same, no switching interface)

2. Soft/temporary ifmetric setup + bring interface down and immediately up

Bringing interfaces down and up will reset metric:

(Assume changing metrics like we showed above - wlan0 0, eth0 10)

```
$ sudo ifconfig eth0 down && sudo ifconfig wlan0 down

$ sudo ifconfig eth0 up && sudo ifconfig wlan0 up

$ netstat -ren

Kernel IP routing table

Destination     Gateway       Genmask       Flags Metric Ref    Use Iface

0.0.0.0         128.59.16.1   0.0.0.0       UG     0      0      0 eth0

128.59.16.0     0.0.0.0       255.255.248.0   U    1      0      0 eth0

160.39.192.0    0.0.0.0       255.255.254.0   U    2      0      0 wlan0

169.254.0.0     0.0.0.0       255.255.0.0   U      1000   0      0 eth0
```

3. Soft/temporary ifmetric setup + network-manager restart
network-manager service restart is the new preferred way to restart network service

This method will also reset the metrics:

(Assume changing metrics like we showed above - wlan0 0, eth0 10)

```
$ sudo service network-manager restart

$ netstat -ren

Kernel IP routing table

Destination     Gateway       Genmask       Flags Metric Ref    Use Iface

0.0.0.0         128.59.16.1   0.0.0.0       UG     0      0      0 eth0

128.59.16.0     0.0.0.0       255.255.248.0   U    1      0      0 eth0

160.39.192.0    0.0.0.0       255.255.254.0   U    2      0      0 wlan0

169.254.0.0     0.0.0.0       255.255.0.0   U      1000   0      0 eth0
```

4. Hard/long-term ifmetric setup + restart network-manager

This method does work, but not in a preferable way:

(Assume changing metrics like we showed above - wlan0 0, eth0 10)

```
$ sudo service network-manager restart

network-manager stop/waiting

network-manager start/running, process 3205
```

```
$ netstat -ren
Kernel IP routing table
Destination     Gateway       Genmask       Flags Metric Ref    Use Iface
0.0.0.0         160.39.192.1  0.0.0.0       UG    0      0        0 wlan0
160.39.192.0    0.0.0.0       255.255.254.0   U   2      0        0 wlan0
169.254.0.0     0.0.0.0       255.255.0.0   U     1000   0        0 wlan0
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:21:86:52:7b:3e
          inet6 addr: fe80::221:86ff:fe52:7b3e/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:57173 errors:0 dropped:98 overruns:0 frame:0
          TX packets:1501 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:7026984 (7.0 MB)  TX bytes:499503 (499.5 KB)
          Interrupt:20 Memory:fe000000-fe020000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:904 errors:0 dropped:0 overruns:0 frame:0
          TX packets:904 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:78798 (78.7 KB)  TX bytes:78798 (78.7 KB)

wlan0     Link encap:Ethernet  HWaddr 00:1f:3c:73:51:11
          inet addr:160.39.192.46  Bcast:160.39.193.255  Mask:255.255.254.0
          inet6 addr: fe80::21f:3cff:fe73:5111/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:17264 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1992 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1846034 (1.8 MB)  TX bytes:900656 (900.6 KB)
```

The main interface does switch to WiFi, but Ethernet remains in a unconnected stage without an IP address. This will makes MIH unable to monitor Ethernet.

Also, it takes a couple of seconds to reset network and re-connect.

Overall, this method remains unfavoured.

Summary

As all the above observations, the current solution is default gateway switching with one-hop ping.

Default gateway switching is currently favoured, because:

- There is no time wasted on bringing interface up/down or restarting network services
- All connections remain active and open, ready for next switch
- No need to touch network configuration, like /etc/network/interfaces

However, more work could be done into the preference of interfaces:

- There could be a way to soft set metric setting and then refresh network
- There could be more way to set interface preference, other than metric
- Maybe a better metric changing tool is available

9 ANDROID KERNEL COMPILATION

In order for HIP to work on Android, several kernel module need to be in place, which are not included in Android normally. Therefore we need to compile a custom kernel for Android.

Below we will show how to build custom kernel for HIP on Android. While kernel configuration is specifically for HIP, the instructions are generally good for building custom kernel and install it on Android.

All instructions are also detailed in **android_porting/kernel/README**.

Step 1: prepare for the Work

First install the dependencies in android_porting, including adb and fastboot:

```
$ cd android_porting
$ ./dependency
```

Then Add device USB rules, clone the repo and follow the Readme to add rules to udev:

```
$ git clone https://code.google.com/p/51-android/
$ cd 51-android
```

Make sure your device is connected to your host with both adb and fastboot. Remember to enable USB debug in Developer option, and plug in your device.

Accept RSA fingerprint if prompted (on the device), and device should show:

```
$ adb devices
```

Proceed to fastboot mode:

```
$ adb reboot bootloader
```

Should show device:

```
$ fastboot devices
```

Step 2: download correct kernel source

First check your Android device codename:
http://www.droidviews.com/list-of-android-device-codenames/
(Example: Nexus 7 is flo)

Reference Google 'Building Kernel' guidelines:
https://source.android.com/source/building-kernels.html

Here is an example of Nexus 7 (flo):
```
$ cd ../
```
(back to 'kernel')
```
$ git clone https://android.googlesource.com/kernel/msm
```
(kernel/msm is in 'source location')
```
$ cd msm
```
(initially empty, need to checkout a branch)
```
$ git branch -a
```
(list all branches)
```
$ git checkout android-msm-flo-3.4-kitkat-mr2
```
(choose the branch of the correct Android version kitkat and codename flo)

Step 3: obtain Toolchain

We will need the bare-metal arm-eabi toolcahin, which Google prebuilt for us.

```
$ cd ../
```
(now you are back to 'kernel' folder)
```
$ git clone https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6/
```
(toolchain in 'arm-eabi-4.6' folder)

In the development, a newer toolchain 4.8 does not work with the kernel version, emitting errors.

Step 4: environment Variables

Note you need to use the full path for CROSS_COMPILE:
```
$ cd msm
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=/path/to/OMNI-Columbia-IRT/android_porting/kernel/arm-eabi-4.6/bin/arm-eabi-
```

These are essential for kernel configuration and building.

Step 5: configure kernel

We need to select configuration, and add kernel modules as required.

```
$ make flo_defconfig
```

(kernel pre-config, flo_defconfig is indicated in 'build configuration' on Google webpage)
```
$ make menuconfig
```
(this will start interactive kernel configuration)

In the configuration menu, enable the following (taken from HIPL document) as modules (click 'm'):
- Enable loadable module support
- Networking support > Networking options > IP: IPsec BEET mode
- Device drivers > Network device support > Dummy net driver support
- Cryptographic API > Null algorithms

Select Exit and answer Yes to save configuration.

Step 6: make kernel

```
$ make
```
Kernel image will be compiled to: msm/arch/arm/boot/zImage

Step 7: prepare boot image tools

Before we head into the steps, first we would like to explain why we leverage boot image.

Android image (ROM if you are familiar with it) consists of five components:

- **boot.img**: kernel and ramdisk (initial file system)
- **recovery.img**: kernel and recovery system
- **system.img**: Android framework
- **userdata.img**: userdata
- **cache.img**: user cache

fastboot utility, which is used in Android bootloader, is cable to replace one of these five images on demand. By substituting the kernel of a boot image (unpacking and repacking), we can boot the device with a custom kernel without modifying any user content and the OS.

A side note is that by substituting system image with an image with custom Java code, it should be possible to boot a device with modified Android OS.

Then, compile boot image packing tool from Google. Many commands reference to: https://gist.github.com/jberkel/1087757.

```
$ cd ../
```
(to 'kernel' folder)
```
$ git clone https://android.googlesource.com/platform/system/core.git
```
(getting core android lib and utils)
```
$ cd core/libmincrypt/
$ gcc -c *.c -I../include
$ ar rcs libmincrypt.a  *.o
$ cd ../mkbootimg
$ gcc mkbootimg.c -o mkbootimg -std=c99 -I../include
../libmincrypt/libmincrypt.a
```

Now you have mkbootimg utility in core/mkbootimg

We include an unpack tool, 'unmkbootimg' from a third-party source, which is pre-packed in boot_img folder. The source of this tool: http://whiteboard.ping.se/Android/Unmkbootimg

Step 8: extract original boot image

Nexus factory image can be found at:
https://developers.google.com/android/nexus/images

In the archive, open the zip image file, and extract boot.img to 'boot_img' folder.

Then use unmkbootimg to extract it.

```
$ cd ../boot_img
$ ./unmkbootimg boot.img
```

Now you will have 'initramfs.cpio.gz' (ramdisk) and zImage (kernel), zImage it the one we will replace.

In the process of unpacking, you will see information similar to the following :

```
unmkbootimg version 1.2 - Mikael Q Kuisma <kuisma@ping.se>
Kernel size 6722240
Kernel address 0x80208000
Ramdisk size 492556
Ramdisk address 0x82200000
Secondary size 0
Secondary address 0x81100000
Kernel tags address 0x80200100
Flash page size 2048
Board name is ""
Command line "console=ttyHSL0,115200,n8 androidboot.hardware=flo user_debug=31
msm_rtb.filter=0x3F ehci-hcd.park=3"

*** WARNING ****
This image is built using NON-standard mkbootimg!
OFF_RAMDISK_ADDR is 0x02000000
Please modify mkbootimg.c using the above values to build your image.
**********

Extracting kernel to file zImage ...
Extracting root filesystem to file initramfs.cpio.gz ...
All done.
---------
To recompile this image, use:
  mkbootimg --kernel zImage --ramdisk initramfs.cpio.gz --base 0x80200000 --cmdline
'console=ttyHSL0,115200,n8 androidboot.hardware=flo user_debug=31 msm_rtb.filter=0x3F ehci-
hcd.park=3' -o new_boot.img
---------
```

By using the command it provides, plus consideration of the WARNING (despite it saying we need to recompile, there is a command option of --ramdisk_offset 0x02000000), we can have the full command we will show.

Note that there are many similar tools to unpack boot.img, e.g. on XDA forum, but this is the only one emit the warning message and instruction to repack.

Step 9: pack new boot image

Since the boot.img is split into kernel and ramdisk, we need to replace the kernel (zImage), which located in 'msm' folder, and repack boot.img.

```
$ ../core/mkbootimg/mkbootimg --kernel ../msm/arch/arm/boot/zImage --ramdisk
initramfs.cpio.gz --base 0x80200000 --ramdisk_offset 0x02000000 --cmdline
'console=ttyHSL0,115200,n8 androidboot.hardware=flo user_debug=31
msm_rtb.filter=0x3F ehci-hcd.park=3' -o new_boot.img
```

Now you have new_boog.img in 'boot_img' folder.

Step 10: boot the new image

Boot the device into bootloader:
```
$ adb reboot bootloader
```
(boot the device into bootloader)
```
$ fastboot boot new_boot.img
```
(boot with the new boot image)

If the device boots, and in Setting-About you can see your kernel version with your computer name and recent kernel build time, you have the newly built kernel running.

Additional Information

(1) Load Kernel Modules Manually

While there should be a way for HIPL to correctly utilize 'modprobe' to load kernel modules dynamically, this section is on how to load a kernel module manually.

Back in the step of kernel compilation, you can recall modules being compiled like this:

```
  CC      crypto/crypto_null.mod.o
  LD [M]  crypto/crypto_null.ko
```

You need to push the .ko modules to Android, and then load them using 'insmod' utility.

Example

On Linux:

```
$ adb push crypto_null.ko /data/misc
```

On Android

```
$ su
# insmod /data/misc/crypto_null.ko
```

(2) HIPL on Android

Documentation on Android is available on HIPL website:

http://infrahip.hiit.fi/hipl/manual/HOWTO.html#android

For now, HIPL keeps complaining on "The modprobe tool is not installed, will not load modules", although there is a version installed in busybox.

But modprobe in busybox is also emitting error, "modprobe: can't change directory to '/lib/modules': No such file or directory", indicating further problem.

A possible solution is to find an Android specific 'modprobe', one reference link:

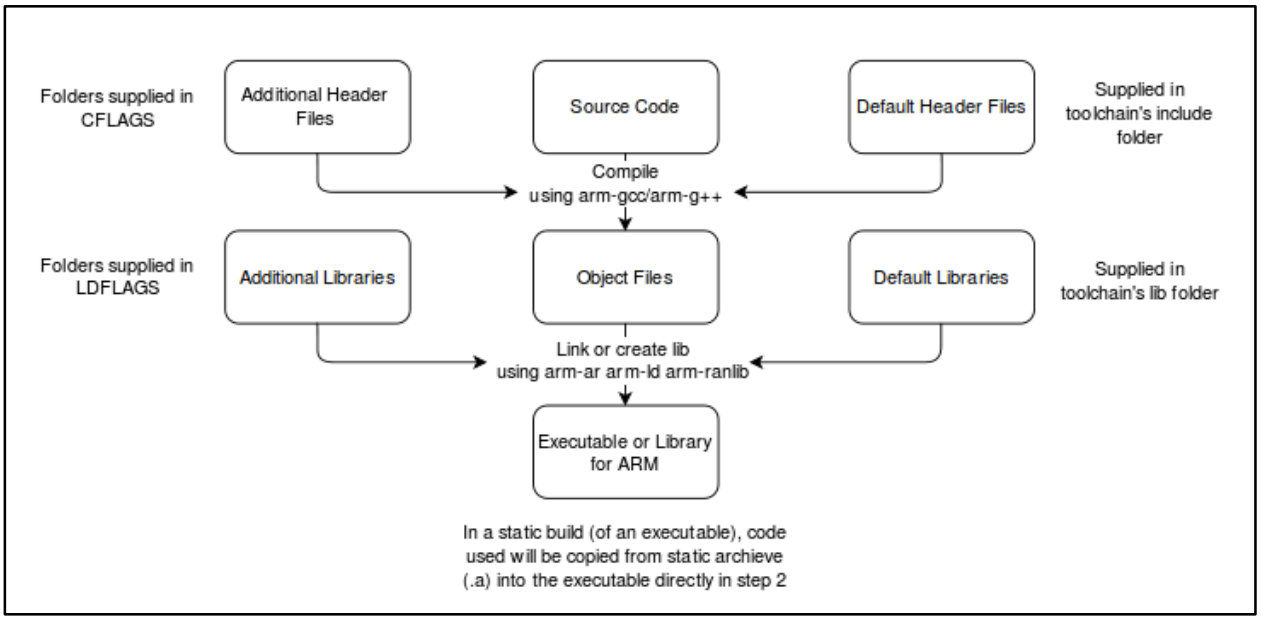
https://github.com/sherpya/android-busybox/blob/master/modutils/modprobe.c

Alternatively, it is also possible that we need a complete Android version of BusyBox afterall.

10 Android Cross Compilation

General

Here is a graph illustrating the basic idea of cross-compilation, in case you are not familiar with it:



In short, cross-compilation is the process of using a cross-compile toolchain turn all code into executables run on that different architecture (e.g. compile on x86 Linux to create executables for arm).

This is just like a process of compilation/linking on Linux. But without utility to automatically take care the dependency structure for you (like in **apt-get**), you will need to discover and tackle the dependencies,

i.e. adding a software on the top of the stack when a dependency is discovered, and work downwards until the stack is empty.

Toolchain Selection

We have gone through a process of picking and experimenting with different toolchain, before we were able to finish cross compilation.

First, **arm-none-linux-gnueabi**, an universal ARM-Linux toolchain. It was the choice in 2013. Although it is last updated in 2010, and no longer being maintained by Mentor Graphics / Codesourcery, it still receives wide coverage among developers. However, it is not a good future-proof choice.

Second, **arm-linux-gnueabi**, another ARM-Linux toolchain. It is offered in Ubuntu software repository (apt-get), and has been used earlier in the term. But a later discovery show that it hardcodes Linux interpreter, and thus cannot work in Android.

In readelf:

"[Requesting program interpreter: /lib/ld-linux.so.3]"

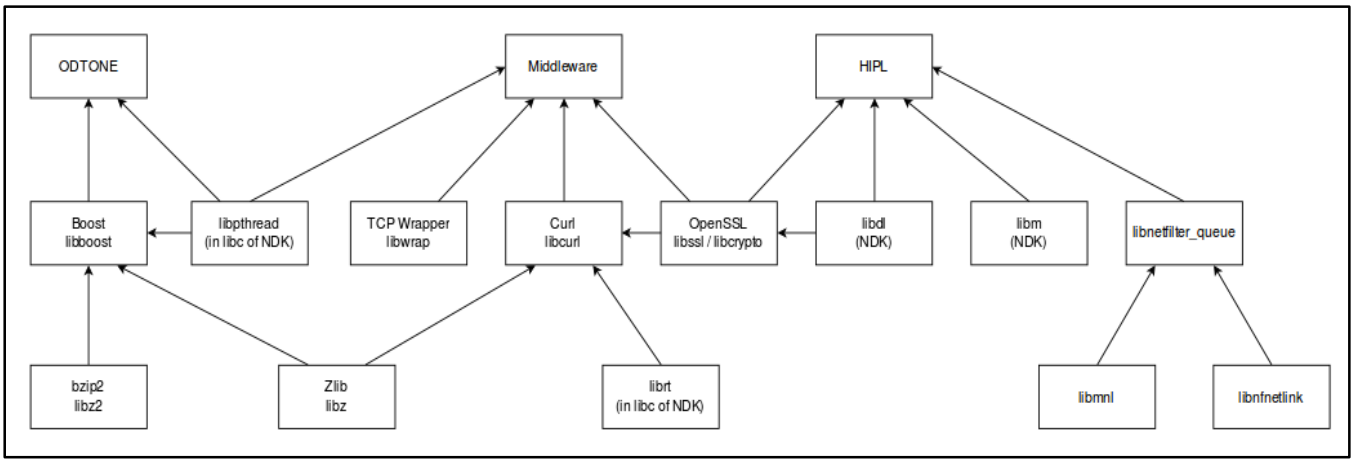
Both these two general purpose toolchains requires static linking, since they rely on a different set of libraries (glibc) and does not include the correct library path (/system/lib and /vendor/lib in Android, not /usr/lib). When static linking is used, the linker in the toolchain will include any library code required in the executable by literally copy them from archive files (.a libraries). On the other hand, in dynamic linking, the linker in the toolchain will assume that the shared libraries (.so libraries) will exist in execution environment to be dynamically linked, therefore only naming the libraries required in the executables.

Third, **arm-linux-androideabi**, toolchain offered by Google in NDK, along with Android library. It is the natural and native toolchain to work with when dealing with Android, however there are major factors that increase the amount of patching work:

- Android uses a strip-down version of C library (Bionic) with different headers
- Android default libraries are different, lack Linux-like support

Dependencies

Here is a graph showing all the dependencies and their relationships.



List of cross compile scripts:

- **compile.sh**: main script to cross-compile
- **dependency.sh**: install dependencies for cross-compilation and kernel building
- **push.sh**: script to push the compield executables and configuration files
- **scripts/common_build.sh**: compile shared dependencies, OpenSSL and Libz
- **scripts/ndk_config.sh**: download and patch NDK, prepare toolchain
- **scripts/middlware_build.sh**: build middleware
- **scripts/hipl_build.sh**: build HIPL
- **scripts/odtone_build.sh**: build ODTONE

See 'Installation' on how to use these scripts.

Executable Testing

The result of executable testing on Android is not quite good as for now:

- **ODTONE**: mih_usr stuck somewhere in the code without useful information, possibly missing kernel and library support.
- **HIPL**: cannot successfully load kernel modules although they exist.
- **Middleware**: cannot test since ODTONEis not working

11 ANDROID JAVA FRAMEWORK

11.1 INTRODUCTION

We started with the notion to research how the Android makes the decision to switch the data networks and how it sets priority of wifi over the usual data network. Our strategy was to look into the source code of the wifi and java framework packages to look for the source code that makes the decision to do so. This would enable us to modify the source code to query our middle ware binaries instead and then port it back to the device. In the next few sections is described the work that has been done to try and achieve this goal.

Challenges faced: During the research on the Java framework it was realized that there were a lot more interconnected instances between several classes of wifi than originally envisaged. To map out all such instances and figuring out the entire structure of the flow of the information to decide when to bring up a certain connection and turn down the other one would take a lot more time than originally envisaged and could be allocated. Also, the absence of a ready documentation made it quite arduous to map out the exact functionalities and the relations between the different classes and we had to resort to go through many lines of code in the java framework. In this report we have tried to be as explanatory as possible in the field of Java framework in Android networking.

11.2 Downloading the source code of Android

The first step to looking into the source code of Android was to download it onto the PC. The following steps are followed to do so:

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH
$ curl https://storage.googleapis.com/git-repo-downloads > ~/bin/repo
$ sudo chmod a+x ~/bin/repo
$ mkdir WORKDIR
```

The directory WORKDIR can be at any location of your choice.

```
$ cd WORKDIR
$ repo init –u https://Android.googlesource.com/platform/manifest
$ repo sync
```

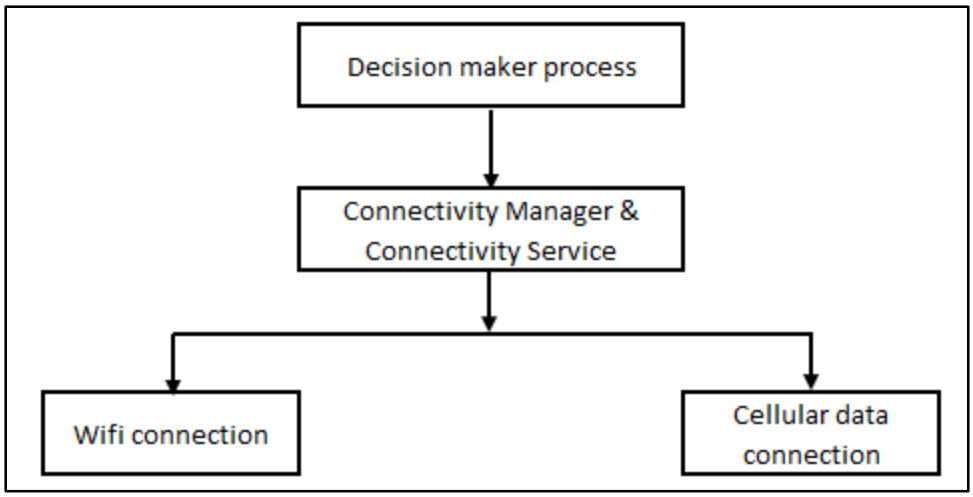
This command is finally used to download the entire source code from the git repo. A portion of the code can also be downloaded (if that is all is needed) by modifying the previous command but we are not following that. After the download is over we need to build the code on the machine to work in it. The following command details how to (assuming we are already in WORKDIR):

```
$ source build/envsetup.sh
```

This command, along with a few subsequent ones can also be used to build the source code after the modifications are made to it and the code needs to be ported back to the device. But since we have not been able to reach that stage, we are not enlisting those commands in this document.

11.3 Java Framework

The following are the crucial Java framework codes that have been looked into. Since there is not a single instance of network switch in a single code, the following descriptions contain the important properties and methods describing as to what their purpose is. This is meant to make it easier for a future researcher in this project to find the code required to be modified.



1. **ConnectivityManager**

   It is used to keep the record for the state of network connectivity for Android. Applications use this class' API to check and track the changes in network connectivity. It monitors the wireless connections of Android (GPRS, Wifi etc) and broadcasts the intents notifying the change in network connectivity. It also allows the applications to select the networks for the data traffic. It defines several variables that can be used to keep a track of all the different kinds of data networks that are supported and the one (i.e. the interface) which is active. The default network attributes are defined in the networkAttributes array in the config.xml file (/frameworks/base/core/res/res/values/config.xml). It maintains a list which should be overwritten by the device to present a list of network attributes. This list can be used by the connectivity manager to decide which networks can coexist based on the hardware. The restore-time in this array for wifi is '-1' which indicates that it will not wait for any other connection to be set up and the preference is set to '1'. If we change these values in this particular array, it will change the network preferences.

   Location: /frameworks/base/core/java/android/net/

   Important features:

   (i) Includes the method setNetworkPreference() to specify the preferred network type to use the network when more than one are available.

   (ii) Defines a method setRadio() to turn the radio on for a particular network.

   (iii) Includes the method called requestNetworkTransitionWakelock() to ensure the device does not go to sleep until it connects to the next available default network. It holds a wakelock for a specified duration in seconds.

   (iv) This class also has a method to check if an active particular network is metered or not. This can be particularly useful in developing the policy language in which the user can also specify the switch to the alternate data network depending upon the data usage statistics. The method is called isActiveNetworkMetered() and returns a boolean true/false.

   (v) It uses the method getActiveLinkQualityInfo() of the return type LinkQualityInfo.java to get the information of the active network link. This information can also be useful while defining the policy of network switch. This particular class maintains the link attributes viz. upload and download throughput, packet error rate and signal strength.

   (vi) Another method called getAllLinkQualityInfo() to return the link quality of all the network links and stores the information in an array of the type LinkQualityInfo.java.

2. **ConnectivityService**

   It is the service responsible for tracking the data connections and establish and maintain connections. It receives the instance from the wifi state tracker that the wifi is down, so it tries to establish the data connection. It checks for the various network parameters to do so like signal strength (and if the data speeds are good or not) assign preferences to the networks. It utilizes an internal wakelock and clears it when a transition is being made from one state to another. It inherits variables and methods from IConnectivityManager.

   Location: /frameworks/base/core/java/android/server/

Important features:

(i) It defines the duration after which the device will switch back to its default network.

(ii) It stores the added the routes to an arraylist called "Collection<>". It is used to check to see if any routes are left in the list. If not, then the routes are deleted from the route table.

(iii) Its constructor ConnectivityService() sets up the connection by setting up the device's hostname and reading the default DNS server's IP.

(iv) It reads in the values from the config.xml file, mentioned in the previous part and load the values (network attributes) into a local array and is used to determine if the device is a wifi only device or not.

(v) It creates an interface called NetworkFactory which is used to create NetworkStateTracker instances. NetworkStateTracker provides the ConnectivityService with services like change notifications, an API for controlling the network and storage service for network specific information.

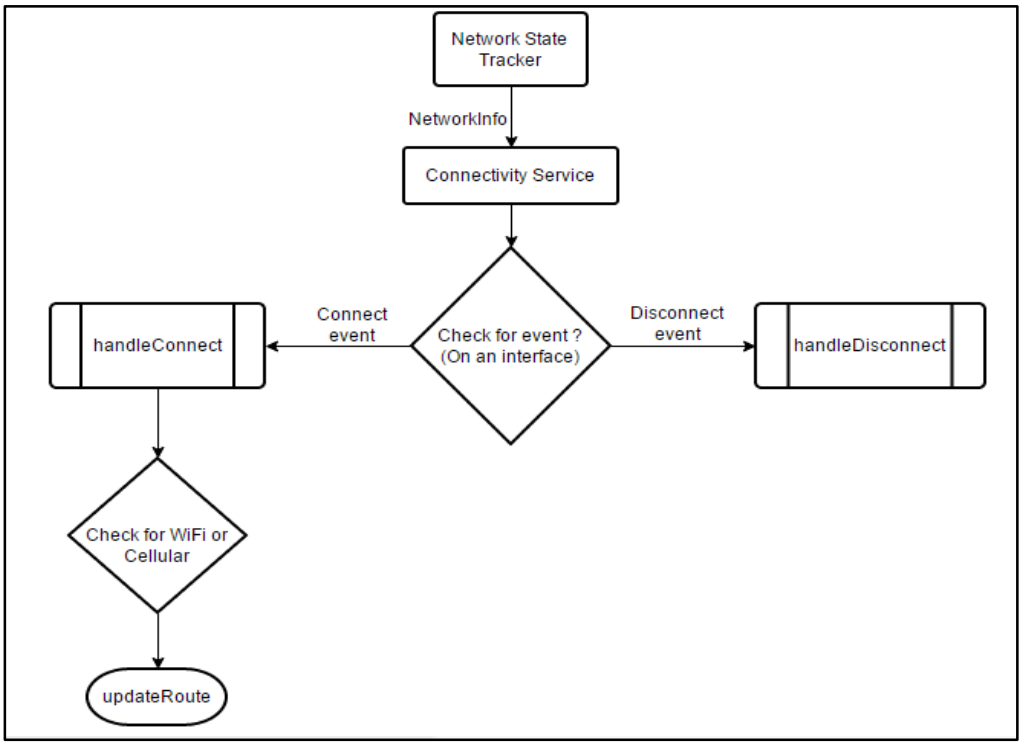
(vi) It takes the network configuration information from the class called NetworkConfig.java (/frameworks/base/core/java/android/net/). The constructor of this class is used to read the values in from the config.xml file that was mentioned before and stores every value in a separate string. The objects of this class can then be used in the ConnectivityService.

(vii) It checks for a network route's existence to the host via the specified network interface using the method requestRouteToHostAddress().

(viii) This class modifies, add and deletes the routes from the device's route table using the methods: addRoute(), removeRoute(), addrouteToAddress(), removeRouteToAddress(), modifyRouteToAddress() and modifyRoute().

(ix) Its handleDisconnect() method is used to handle a disconnect event. For a non-active network, it can be ignored (this may happen because of explicitly disabling a network in accordance with network preference policies) but for an active network a broadcast is sent first and then checked to see if any other network can be connected to. If not, then it goes ahead with disconnection of an interface. So it clears the routing table and DNS server entries.

(x) In the event of a new network attempting to connect to the device, the method named handleConnect() will first check if it is one of the default networks. If yes, then the other default network that was running is checked for preference and the one with higher preferences is connected to and the other one is killed. To connect to the new network, it first has to release the transition wakelock. This process causes the device to temporarily disconnect from all networks. It then uses a method handleConnectivityChange() to ensure the device is then connected to the correct DNS servers and the correct routing table entries exist.



3. WiFiManager

It provides the primary API for managing the WiFi connectivity. It keeps a list of configured networks which can be viewed and updated as and when required and settings for the currently active wifi network. Connectivity to this network can be established and torn down and the network state information can be queried using this class. It also includes methods to return detailed information about access point scans to decide which one to connect to.

Location: /frameworks/base/wifi/java/android/net/wifi/

Important Features:

(i) It uses an internal method named getConfiguredNetworks() to fetch the list of all the wifi networks (and their properties) configured in the supplicant. The list is in the form of WifiConfiguration objects.

(ii) Contains the API methods addNetwork() to add a new network description to the list of configured networks and returns a network ID; and updateNetwork() to update the network description for a particular configured network. These methods take as input parameter the object of class WifiConfiguration which holds the data to be updated for the particular network. The method removeNetwork() is used to remove a network from the list of configured networks. Its input parameter is the integer network ID of that particular network to the supplicant.

(iii) setWifiEnabled() is used to enable the or disable the wifi based on a boolean input parameter to it and the method enableNetwork() is used to enable the association with a particular previously configured network by initiating connection to it. It takes as input the network ID from the list of the configured networks.

(iv) The method disconnect() is used to dissociate from the current active network. To connect to an already configured network, the sequence of function calls is addNetwork(), enableNetwork(), saveConfiguration() and reconnect(). But for a

new network, instead of this sequence, the standalone method called connect() can be used which uses the configuration object (of class WifiConfiguration) as a parameter input.

(v) It has method getDhcpInfo() to return the addresses assigned by the DHCP from the last successful request for a connection.

(vi) The API can also request the signal strength of a network if it has to be shown to the user using the calculateSignalLevel() function and can also compare the signal strengths of two different networks in the vicinity by calling another method compareSignalLevel().

(vii) The class WifiStateTracker in turn uses boolean inputs from the methods startWifi() and stopWifi(), which are used to start or stop the wifi driver itself and therefore connect or disconnect from a network. These methods override the WifiLock and the idle state of the device. The driver will stay disabled after calling stopWifi() until the method startWifi() is not called.

(viii) If an application needs the wifi to stay on (possibly for data transfer that needs longer time), it needs to acquire a WifiLock over the radio (using acquire() of the nested class WifiLock). Otherwise the radio will turn off if the device goes into idle state (even if the wifi driver is running). The lock is held until the release() is called.

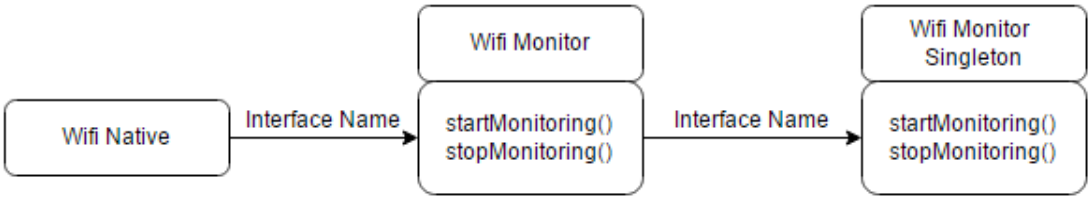
4. WiFiMonitor

This class is used to listen to the events from the wpa_supplicant server and passes them onto the StateMachine for handling. It runs in its own thread.

Location: /frameworks/base/wifi/java/android/net/wifi/

Important Features:

(i) It has methods viz startMonitoring(), stopMonitoring() which in turn call the methods of the same name through a constructor of the nested class WifiMonitorSingleton. The interface name is taken as input by WifiMonitor's parametrized constructor with the parameter of the type WifiNative. This name is then passed through the aforementioned functions to the functions of the same name of class WifiMonitorSingleton.



(ii) WifiMonitor's stopSupplicant() method calls the method of the same name of the WifiMonitorSingleton class which in turn uses the object of the WifiNative class to call its method of the same name.

(iii) It has a nested class MonitorThread whose method run() runs infinitely to keep monitoring the events from the supplicant. It has method handleSupplicantStateChange() to handle the events related to the change of state of the supplicant. It takes as input the new state of the supplicant.

(iv) MonitorThread has the method handleNetworkStateChange() which takes as input from the NetworkInfo class the new state of the network and then checks for the network ID of the old state. It then calls the notifyNetworkStateChange() by sending the new state and the old network ID to it to send the notification of change in network to the state machine.

5. WiFiNative

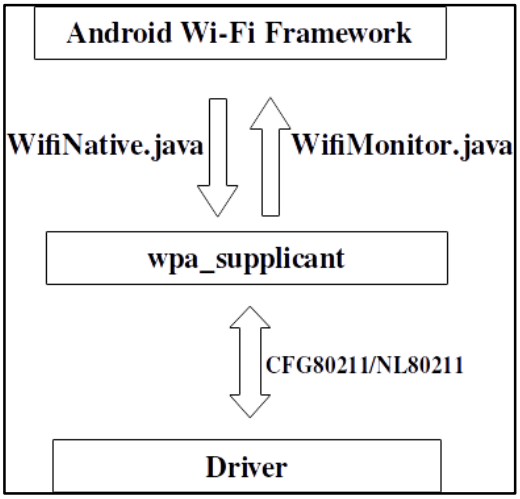
This class is used to bring up or down the wpa_supplicant daemon and for sending instructions and requests to the wpa_supplicant.

Location: /frameworks/base/wifi/java/android/net/wifi/

Important Features:

(i) Contains the methods startSupplicant(), stopSupplicant() to control the operations of the wpa supplicant.

(ii) removeNetwork() removes a network with a particular network ID from the supplicant network list.



6. WifiStateMachine

It tracks the state of wifi connectivity. All the event handling related to wifi is done in this class and all changes to connectivity are initiated in here. It supports the simple client mode of operation and the access point mode of operation, wherein the device itself is acting as an access point. It extends its properties and methods from the StateMachine class. It handles the results of the wifi scans and checks if the underlying chipset supports background scanning.

Location: /frameworks/base/wifi/java/android/net/wifi/

Important Features:

(i) It defines the interval (in msecs) between the device polling for the signal strength (in RSSI) and the linkspeed information.

(ii) It maintains the WakeLock that needs to be held while the wifi is starting (or stopping) and the driver is loading (or unloading).

(iii) It keeps a log entry of the addresses that have been updated or removed using the methods addressUpdated() and addressRemoved().

(iv) Its nested class DriverStartedState tracks the state of the supplicant and network connection state events from the monitor connection to help the framework synchronize with the current supplicant state.

7. WifiStateTracker

It tracks the state of wifi for connectivity service and sends the events to connectivity service using the handler of that class.

Location: /frameworks/base/wifi/java/android/net/wifi/

Important Features:

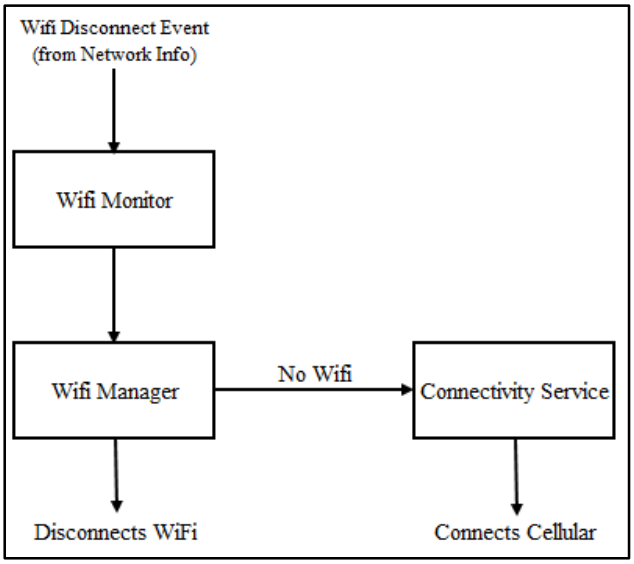
(i) It uses startMonitoring() to monitor the wifi connectivity. It takes instance input from the WifiManager.

(ii) It uses teardown() and reconnect() to disable and re-enable connectivity to a network, respectively. They in turn call the methods stopWifi() and startWifi() of the class WifiManager to handle these events. It also utilizes the function setRadio() to call setWifiEnabled() of WifiManager to turn the wireless radio off for a network.

(iii) The method isAvailable() calls the method of the same name of NetworkInfo class to check the connectivity of the supplicant daemon and thereby return the availability of the wifi as true or false.

(iv) It uses the LinkProperties class inside the method getLinkProperties() to fetch the link properties of the network.

(v) The method getLinkQualityInfo() of the type LinkQualityInfo is used to check for the quality of the network link. It then fetches the Tx packet count and Tx packet error count from the methods of class SamplingDataTracker and the signal level from the calculateSignalLevel() method of WifiManager.



8. DataConnection

This class represents a single cellular data connection. It extends from the state machine class.

Location: /frameworks/opt/telephony/src/java/com/android/internal/telephony/dataconnection

Important Features:

(i) The nested class ConnectionParams is used to initialize the connecting parameters and save internally. The parameters include the APN (access point name) details and the RIL daemon's radio technology. Similarly the nested DisconnectParams saves the disconnecting parameters such as APN which it is disconnecting from and the reason for disconnect.

(ii) As above, it uses the LinkProperties class to fetch the link properties through its method getCopyLinkProperties().

(iii) The method onConnect() is used to start setting up the data connection by calling the setupDataCall() method of the class PhoneBase and tearDownData() is used to disconnect the data connection. It uses the method getRadioState() of the class PhoneBase to check if the radio is still on or not for this purpose. On connection, the method initConnection() is used to initialize the settings for the particular network. It will check for the APN settings for doing so and if they happen to be incompatible, will not let the connection proceed.

(iv) The notify methods viz. notifyConnectCompleted() and notifyDisconnect Completed() are used to send the respective event notifications and the entries are logged with a timestamp. They have the input parameters the objects of the nested classes ConnectionParams and DisconnectParams respectively.

(v) Its nested classes maintain the different states of the data connection. DcDefaultState is the parent state for all the other states. It uses the object of classes PhoneBase and DcController to either register for DRS and add the connection to the list of connections through their respective methods getServiceStateTracker() and addDc() or vice versa.

(vi) The nested class DcInactiveState which maintains the information that the state machine is currently inactive and is waiting for a connect event. It also informs all other context that there was a failure in connecting to a data network and the reason for it. Mainly it is because the device is unable to associate itself with an APN. Another class DcActivatingState on the other hand maintains the information that the state machine is trying to activate a connection and the class DcActiveState tells the contexts that the state machine is connected and is waiting for a disconnect event.

9. DcController

The DC controller is used to control multiple data connections. It can demultiplex any incoming unexpected notification message to the appropriate data connection.

Location: /frameworks/opt/telephony/src/java/com/android/internal/telephony/dataconnection

Important Features:

(i) It maintains an arraylist of the object DataConnection type.

(ii) It defines variables used to track the physical link connection activity/inactivity of the data connection by the class DcTracker.

(iii) It runs a separate thread for every data connection. The handler of every thread will then handle the individual runnables and message queues for each individual dc controller thread.

(iv) It has methods addDc(), removeDc() to add and remove the data connections to its list.

(v) Its nested class DccDefaultState registers the handler of the thread for RIL connect events and state change of the data network using the method registerForRilConnected() of the class CommandsInterface.

10. DcTracker

It is used to track any changes to the APN. It extends its methods and properties from the class DcTrackerBase.

Location: /frameworks/opt/telephony/src/java/com/android/internal/telephony/dataconnection

Important Features:

(i) Its method isApnTypeActive() is used to check if an APN is active or not by querying the get() method of the class ApnContext.

(ii) It checks if the data flow is possible on a connection using a method isDataPossible() in which it firstly checks if the APN context is enabled or not and retrieves its current state. It then uses the method isDataAllowed() of the class DcTrackerBase (which it can directly use since it is extending from that class) and then combines the result of both to check and return a boolean value stating if the data flow is possible or not.

(iii) It also needs to check the link properties (DNS, gateways etc) and the link capabilities (bandwidth required and available) for the data connection for the particular APN type. It does so using the methods of the classes LinkProperties and LinkCapabilities.

(iv) To ensure that the device is connected to the APN of the specified type, we can use the method enableApnType(). If it returns the status as "request started", the instance is broadcasted to the ConnectivityManager when the connection gets established.

(v) We can check if the data connectivity has been enabled for an APN by using the method getAnyDataEnabled().

(vi) The state of change in the data connection can be checked using the method onDataStateChanged(). It takes the input parameter from the RIL daemon. It then stores the changes to the details of the data connection in a separate arraylist. It checks for the dormant or other active connections by starting polling.

(vii) The phone can check I multiple data connections can be supported (by carrier policies, radio technology etc.) using the method isOnlySingleDcAllowed().

(viii) Data connection when roaming in another telecom circle can be enabled (and configured) or disabled using the methods onRoamingOn() and onRoamingOff().

(ix) When the event for disconnect is received by the class, the method onDisconnectDone() is called to disconnect from the APN (and turn the radio off if airplane mode request was called originally).

11. DataCallResponse

It is used in ril.h. It defines the variables used to store the details of a data connection.

Location: /frameworks/opt/telephony/src/java/com/android/internal/telephony/dataconnection

Important Features:

(i) It enumerates the different kinds of responses (and errors/failure causes) that are returned on attempting a data connection by onSetupConnectionCompleted() like "success", "bad command error", "ril error" etc.

(ii) It is used to set the link properties for a particular data connection (starting with cleaning the previous data) like interface name of the link, IP address of the link, DNS servers and gateways using the method setLinkProperties(). If an error occurs, the link properties are cleared and the process is tried again. It returns the particular response that were initially enumerated as a group.

12. NetworkInfo

It is used to describe the status of a network interface. It uses its method getActiveNetworkInfo() to send an instance representing the current network connection to the ConnectivityManager.

Location: /frameworks/base/core/java/android/net

Important Features:

(i) It maintains different states viz. IDLE, CONNECTING, DISCONNECTING etc in a hashmap that can later be used to check the status of a connection in a function call. This information can be set using the method setDetailedState().

(ii) Its method getType() returns the type of network that the class currently holds. The network types are defined inside the ConnectivityManager class. It may be mobile, wifi, Bluetooth or any other defined in the class.

(iii) The method isConnected() can be called to check if a network connectivity exists and a data transfer is possible or not and isAvailable() is used to check if a network connectivity is possible or not. This is especially useful in the cases where the network is unavailable due to coverage area radio being off etc.

(iv) To check if the device on the current network is roaming or not (i.e. out of his home network). This is used to keep in check the roaming charges that may incur on the user due to this situation.

13. DataConnectionStats

It extends from BroadcastReceiver. It saves the statistics of the current data connection.

Location: /frameworks/base/services/java/com/android/server/connectivity

Important Features:

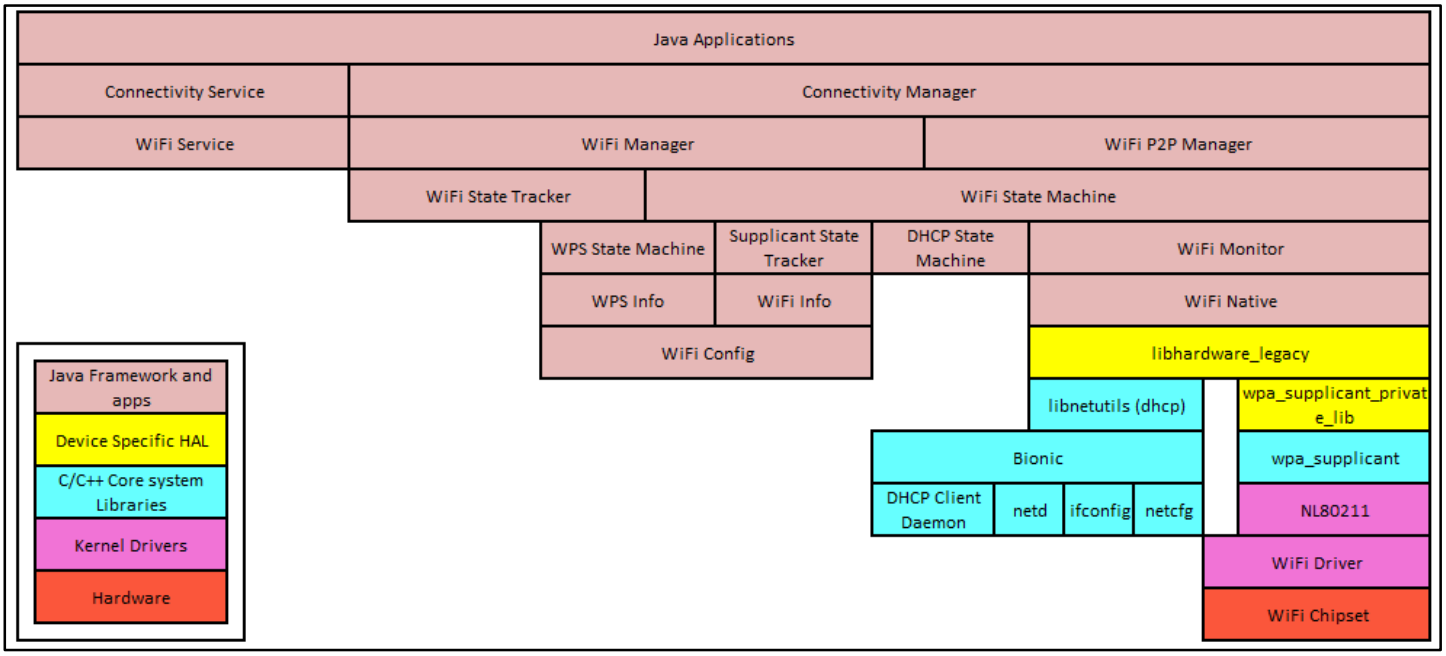
(i) It uses the method startMonitoring() to monitor the changes in the state of the various parameters of a connection viz. service state, signal strength, data connection state (whether up/down) using the constructor of the class PhoneStateListener.

(ii) Its method notePhoneDataConnectionState() is used to check the state of the sim and if it is GSM or CDMA and confirms the data network type associated with it.

14. SignalStrength

It contains the phone signal strength related information. The methods of this class can be used by the external applications also in addition to the internal codes. It also checks if the connected network is GSM or not using a Boolean variable.

Location: /frameworks/base/telephony/java/android/telephony



12 How WiFi works on Android devices

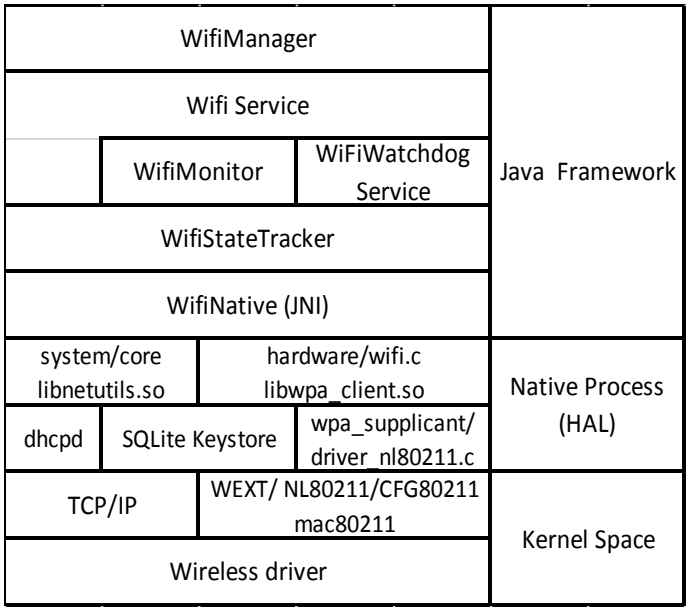
The Wifi Architecture on Android can be divided into three parts: The Java Framework (name of the codes mentioned above), the Hardware Abstraction Layer (wifi.c, wpa_supplicant), which is also known as the native system library and the kernel space modules (wireless stack, driver).

The Java framework communicates with the wpa_supplicant using the native interface wifi.c and uses the wireless extension to control the wifi driver.

File Locations:

1. **Java framework:** base/wifi/Java/Android/net/wifi
2. **JNI Code** (Android_net_wifi_Wifi.cpp): frameworks/base/core/jni/
3. **wifi.c:** hardware/libhardware_legacy/wifi

The JNI code communicates with the HAL layer. The HAL code is in wifi.c which communicates with wpa_supplicant over the control interface.



13 STATIC VS DYNAMIC LINKING

The runtime libraries provided by the Android compiler toolchain `arm-Linux-Androideabi-`, (Android libc) is the Bionic libc rather than the GNU libc (glibc) i.e. `arm-Linux-gnueabi-` and is therefore present on the NDK and all the Android devices. Even if we decide to build the binaries using glibc on NDK, though it is possible to do so, the name will clash with the system libc when trying to install it on the Android devices if we are in the process of building a dynamic library but if we are building the glibc as a static library the above issue is skirted as we never need to install a static library.

Now, to statically link against glibc (and other dependencies) while cross-compiling with the arm toolchain (or ndk-build), if we decide to build the glibc using the NDK, then the Android.mk uses the variable BUILD_STATIC_LIBRARY to build the static libraries. But in case we decide not to use the NDK, it will be problematic since it is highly discouraged for mobile platforms.

During the start-up the statically compiled executables load faster than the dynamically linked ones as the dynamic library loading is not involved. But on the other hand the disk requirement and the memory requirement increases with the statically linked executables. This is because when we link the Android app's JNI libraries against the Bionic libc we can inherit only shared read-only access to a copy already in the memory.

If we need to use any additional functionality of the bionic libc which is not there in it, we can always provide our own implementation of the function if it is missing from the system libraries.

13.1 DYNAMIC LINKING

The following process details how to go about generating the dynamically linked executables:

1. Download the toolchains from CodeSourcery for ARM GNU Linux. The tool chain are the legacy toolchains for which static compilation is also feasible.

2. We can then compile the source files as:

```
$ arm-none-Linux-gnueabi-gcc -c hello.c
$ arm-none-Linux-gnueabi-gcc -c start.c
$ arm-none-Linux-gnueabi-ld \
    --entry=_start \
    --dynamic-linker /system/bin/linker -nostdlib \
    -rpath /system/lib -rpath ~/tmp/Android/system/lib \
    -L ~/tmp/Android/system/lib -lc -o hello hello.o start.o

$ adb push hello /data/hello/
# cd /data/hello
# ./hello Hello, world!
```

   Here the files hello.c and start.c are any sample files being compiled using the toolchains above. The following options are being used:

   --dynamic-linker: Gives the place of the dynamic linker.
   -rpath: location where the dynamic linker search libraries.
   -L: Location where the ld searches the library (option used to denote: -l at compile time).

3. The next step would be to try and run the binary and see if it works.

13.2 Differences between Bionic libc and glibc

1. The bionic libc does not handle C++ exceptions. Neither do its routines throw exceptions themselves, nor do they pass exceptions from a called function back to their caller. Support for C++ exceptions causes significant overhead to the function calls. And since Java, which is Android's primary language of implementation handles the exceptions within the run time package, it is not required for C++ to do so as well.
2. The pthread implementation has been developed for Android. It is an implementation of POSIX pthreads that are necessary for supporting threads in Dalvik JVM.

14 Additional Documentations

Below is a list of all the presentations throughout Spring 2015 semester:

- Unified Heteogenerous Network Middleware - Project Plan.pdf
- Unified Heteogenerous Network Middleware - Week 3 Presentation.pdf
- Unified Heteogenerous Network Middleware - Week 6 Presentation.pdf
- Unified Heteogenerous Network Middleware - Week 7 Presentation.pdf
- Unified Heteogenerous Network Middleware - Week 10 Presentaion.pdf
- Unified Heteogenerous Network Middleware - Week 13 Presentaion.pdf
- Unified Heteogenerous Network Middleware - Week 16 Presentaion.pdf

These presentations are all uploaded to the Wiki, and may help you understand what has been accomplished in the semester.

Final presentations of the semester:

Lingyuan He - Unified Heteogenerous Network Middleware - Final Presentation.pdf

Dhruv Kuchhal - Unified Heterogeneous Networking - Java Framework - Final presentation.pdf

There are also some additional previous documents in Wiki that we think are necessary to enhance the understanding of how the project works:

- Network Manager Document
- Network Manager Presentation
- Policy Manager
- Policy Language
- Sock Proxy
- MIPv6
- Security Architecture

15 Future Work

Future tasks are available on both Android and Linux.

On Linux:

- Update and maintain the software components to newer Operating System
- Refactor middleware for higher standard and readability
- Continue to research a better way to switch interface
- Design and implement a new policy language

On Android:

- Continue to research Android Framework on modification
- Make use of Android Framework to write modifications, using the native method to switch between interfaces. Then make the modification work with HIPL, ODTONE and the control middleware.
- Continue to experiment with HIPL to make it work
- Work with cross-compiled middleware and ODTONE to debug and test them

Other side tasks that could be valuable:

- Restart using Mobile IPv6, looking into new implementations
- Looking into Android compilation, research the possibility of to modify Android framework and quickly boot it without having to totally flash the target device.