Unified Heterogeneous Networking Middleware Mobility Team Project Report

Spring 2017

Gaurav Mishra, Kevin Shen, Guanlin Zhou

August 5th, 2017

Contents

1 Introduction **4**
1.1 Background 4
1.1.1 Mobility Problem 4
1.1.2 TCP/IP and Mobility 5
1.2 Overview of our Work 5
2 Goals **6**
3 Mobility Protocols **7**
3.1 Mobile IP (MIP) 7
3.1.1 Overview 7
3.1.2 Architecture 7
3.1.3 Analysis 9
3.2 Host Identity Protocol (HIP) 9
3.2.1 Base Exchange 9
3.2.2 Mobility using Rendezvous Server 10
3.2.3 Analysis 11
3.3 Locator/Identifier Separation Protocol (LISP) 11
3.3.1 Overview 11
3.3.2 Architecture 12
3.3.3 Analysis 13
3.4 Comparison 13
4 Mobility Implementations **15**
4.1 MIP 15
4.2 HIP 15
4.3 LISP 16
4.3.1 Rooted Version 16
4.3.2 Unrooted Version 19
4.3.3 Conclusion 27
4.4 Open vSwitch (OVS) 27
4.4.1 Background 27
4.4.2 Our progress with OVS 28
4.4.3 Android kernel compilation 28

4.4.4 OVS kernel module compilation 30

5 Conclusion 33

6 Future Work 34

6.1 OOR 34

6.2 OVS 34

6.3 HIPL 34

6.4 Multipath TCP (MPTCP) 35

7 Acknowledgements 36

8 References 37

1 Introduction

1.1 Background

Currently, mobile phones do not have the capability of making intelligent and seamless handovers between heterogeneous networks, i.e., networks using different wireless technologies. There is usually a simple default configuration to connect to a network, which is either specified through factory settings or overridden by the user. This configuration is both static and not intelligent. In addition, mobile phones cannot make any seamless handover when moving from one administrative domain to another. When the device is assigned a new IP address, the existing TCP connection times out and breaks. Therefore, the HetNet team aims to address two problems:

- Making intelligent decisions to switch between networks (Control Middleware Team)
- Making seamless transitions between networks (Mobility Team)

1.1.1 Mobility Problem

We now introduce the mobility problem using a common phenomenon in daily life.

Let's suppose you are in a restaurant and your cell phone is connected to the Wi-Fi network provided by that restaurant, and you are downloading a large file like a video on your cell phone. When you finish you meal and leave the restaurant, your cell phone can no longer connect to the Wi-Fi network because you have moved out of its coverage, so the downloading also stops. Even though your cell phone will connect to LTE network shortly afterwards, but there is still a period of time when your cell phone is connected to no network.

The reason of this phenomenon is because of the design of TCP/IP, which does not support a dynamic transition of IP addresses within a TCP session. As a result, every time we change the network we are connected to, the IP address of our cell phone changes and the TCP session breaks [1]. However, the server has no idea of the change of our IP address, so it continues sending packets to the old IP address. And now we are assigned the new IP address, so we can not receive the packets any more.

The disadvantage of this kind of design becomes more obvious as there are more available networks around us. Hence, we need to find an alternative or improvement of TCP/IP, which enables the seamless transition between two different networks, and integrate it with the policy engine to achieve intelligent and seamless switching between networks.

1.1.2 TCP/IP and Mobility

When a device moves from a subnetwork to another subnetwork, its IP address changes. However, TCP is designed to handle a single IP address within a TCP session. TCP uses the IP address as identifier to sequence and combine packets. Thus, if the IP address changes while a TCP connection is active, the existing TCP connection times out, and the packets never reach the new IP address. Therefore, TCP/IP inherently does not support mobility.

1.2 Overview of our Work

We studied various mobility protocols, such as Mobile IP (MIP), Host Identity Protocol (HIP), and Locator/Identifier Separation Protocol (LISP). We then searched for implementations that exist for the Android operating system. This involved reading various papers, emailing professors, and looking into mailing lists. Then, we compiled and tested the implementations, installing them on both rooted and unrooted phones. Finally, we documented our results with the implementations for future reference.

2 Goals

The goals for the project are the following:

- Learn and understand how various mobility protocols work (MIP, HIP, LISP, etc.);
- Research and find existing mobility implementations for Android;
- Port the existing solutions into the HetNet project;
- Analyze the performance of each solution by comparing battery usage, CPU usage, and functionality;
- Build an API for seamless handover that can be used by the Control Middleware Team.

3 Mobility Protocols

3.1 Mobile IP (MIP)

3.1.1 Overview

Mobile IP (MIP) is a protocol designed by the Internet Engineering Task Force that allows mobile device users to switch between networks while maintaining a permanent IP address, its home address. MIP achieves mobility by associating each mobile device with two IP addresses: a home address (permanent IP address), and a care-of address (physical IP address). The mapping between these two addresses is maintained by a home agent, a router that acts as the mobility manager [2].

3.1.2 Architecture

Here are a few terms that are required to understand the architecture of MIP [2]:

- Mobile Node (MN): the device that is switching between networks.
- Correspondent Node (CN): any node communicating with the mobile node.
- Home Network: the network in which the device receives its permanent IP address.
- Home Address: the permanent IP address used to identify the device.
- Home Agent (HA): a router in the home network that tunnels packets to the device while it is not connected to the home network.
- Foreign Network: the network that the device is connected to while away from the home network.
- Care-of Address (CoA): the IP address that is assigned by the foreign agent when the device connects to a foreign network.
- Foreign Agent (FA): a router that assigns care-of addresses to visiting devices.
- Binding Update (BU): a message that associates the home address with its care-of address.

When the Mobile Node (MN) operates in its home network, packets sent by the Correspondent Node (CN) are routed directly to the MN's home address.

When the MN operates in a foreign network, it is assigned a Care-of Address (CoA) by the Foreign Agent (FA). The MN then informs the Home Agent (HA) of its CoA, and the HA maintains the mapping between home address and CoA. In order to authenticate this binding update, an IPsec Security Association must have been created between the MN and HA [3].

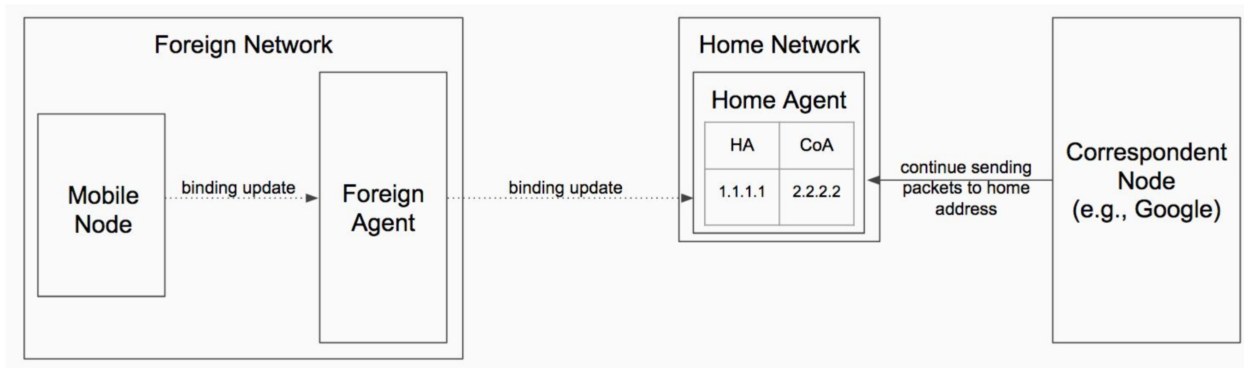

Figure 1. Binding update from MN to HA

At this point, packets sent by the CN are still addressed to the MN's home address, so the HA forwards these packets to the MN. The HA achieves this by encapsulating packets with the CoA data and tunneling these packets to the FA, which decapsulates and forwards them to the MN. Packets sent from the MN to the CN are tunneled by the FA to the HA, and then forwarded to the CN. This mode of MIP is known as bidirectional tunneling [4].

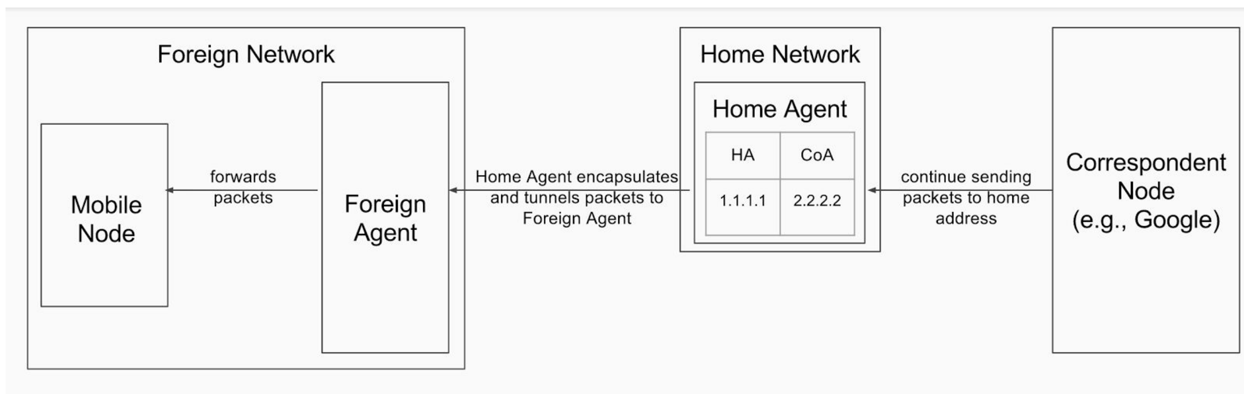

Figure 2. MIP in bidirectional tunneling mode

Bidirectional tunneling can be quite inefficient, so the MN can send a binding update (BU) to the CN, informing the CN of its new location [4]. This way, packets from the CN are routed directly to the MN, and vice versa. This mode of MIP is known as route optimization.

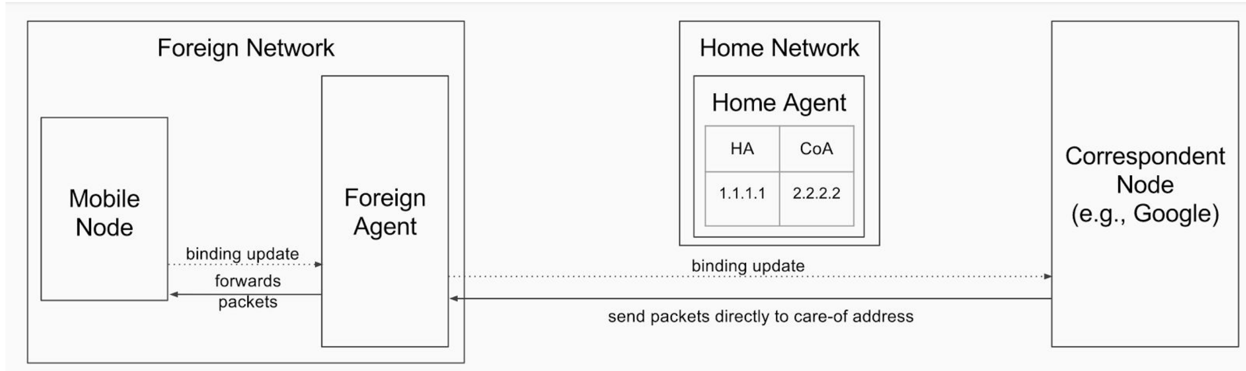

Figure 3. MIP in route optimization mode

3.1.3 Analysis

There are some advantages for MIP. MIP is a built-in feature of IPv6, and also has a specific implementation to support IPv4, which is MIPv4, so it has excellent compatibility. Another advantage of MIP is its security. In order to authenticate binding updates, an IPsec Security Association is created between the MN and HA. This prevents man in the middle attacks.

But there are also some problems with MIP. One problem lies in quality of service. Since the binding update may take longer than expected, packets may be lost when a mobile node connects to a new domain.

3.2 Host Identity Protocol (HIP)

Host Identity Protocol (HIP) has similarities with MIP. It also maps a virtual address to a physical address. However, instead of using two IP address, HIP uses a separate Host Identifier Tag (HIT) as identifier and the IP address as locator. This decouples the use of IP address as both locator and identifier [5]. HIT is a 128-bit unique host tag which remains constant even if the IP address of the host changes [6]. HIP maintains this mapping between HIT and IP address to enable mobility.

3.2.1 Base Exchange

This is the four-message step which enables the source node to setup a secure connection with the corresponding node [6].

- I1 Message: is sent from the source node to destination node, and contains Diffie Hellman list of public key identifiers from a new Host Identity namespace for mutual peer authentication.
- R1 Message: is sent from the corresponding node, and contains a puzzle, Diffie Hellman list and key.
- I2 Message: is sent from the source node to the corresponding node, and contains the solution to the puzzle, Diffie Hellman list, and key.
- R2 Message: Finalizes the base exchange. The packet is signed.

There are two main uses of the Base Exchange step [6]:

- Exchange symmetric keys using the Diffie Hellman setup, the source and the corresponding node exchange keys without sharing their private key to make a common symmetric key. This ensures that there is a symmetric key available on both ends and hence, all messages sent in the future are secure.
- Stop Denial of Service (DoS) attack by using the puzzle and solution setup. The source node will have to solve the puzzle before it can set up the connection, thereby preventing DoS attacks.

3.2.2 Mobility using Rendezvous Server

As mentioned above, HIP achieves mobility by using HIT as identifier and the IP address as locator. This HIT is stored as a new record in DNS. The rendezvous server stores the mapping between IP address and HIT. Hence whenever a host changes its network domain, it sends a binding update request to the rendezvous server to update the mapping [7]. Here is how a normal communication between two nodes using HIP with rendezvous server is performed:

- The application queries DNS to get both the HIT of destination and the DNS CNAME of its rendezvous server.
- The DNS is queried again to get the IP address of the rendezvous server.
- The transport layer uses HIT to sequence packets.
- The network layer forwards the packets to the rendezvous server of the correspondent nodes.
- Once the packet reaches the rendezvous server of the correspondent node, the rendezvous server then relays the packet to the correct IP address as it has the mapping of the correspondent node's HIT and latest IP address.

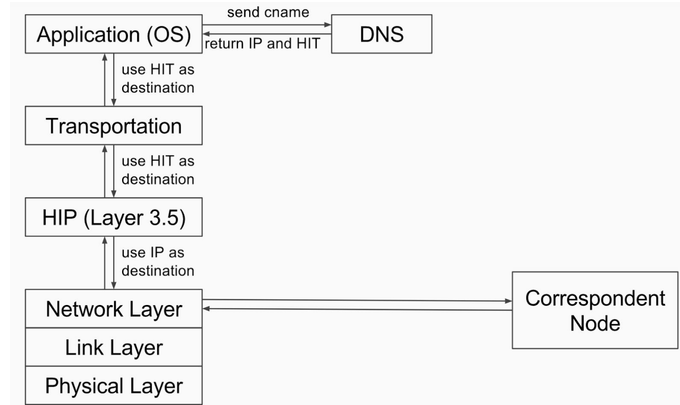

Figure 4. Work flow of HIP

3.2.3 Analysis

HIP achieves mobility by maintaining the mapping between HIT and IP address, so it supports both IPv4 and IPv6. Because as long as the mapping is maintained, HIP would work for any kind of IP address.

In order to support HIP where both devices are mobile, rendezvous servers are required as a component, which is a new network element that must be added to the infrastructure. This addition of new network is one the makes migration to HIP a difficult task.

On the other hand, a strong point of HIP is security. The Base Exchange of HIP prevents Denial-of-Service (DoS) attacks and ensures secure communication.

3.3 Locator/Identifier Separation Protocol (LISP)

3.3.1 Overview

Locator/Identifier Separation Protocol (LISP) is a protocol developed by the Internet Engineering Task Force LISP Working Group. It aims to solve the mobility problem of TCP/IP.

Similar to HIP, LISP also decouples the two purposes of IP address. It achieves this by assigning each mobile node an Endpoint ID (EID), which is used to uniquely identify a network interface

within its local network addressing context, and a Routing Locator (RLOC), which is used to identify where a network interface is located within a larger routing context [8].

3.3.2 Architecture

LISP achieves mobility by maintaining the mapping between EID and RLOC, and encapsulating and decapsulating packets. There are various components in LISP to do the three essential tasks above [8].

- Egress Tunnel Router (ETR): a device that is the tunnel endpoint, it accepts packets whose destination address is one of its own RLOCs, then decapsulates the packets and delivers them to the corresponding mobile node.
- Ingress Tunnel Router (ITR): a device that is the tunnel start point, it receives packets from a mobile node, encapsulates the packets with the RLOCs of the mobile nodes on its own side and on the destination side, and sends them to an ETR.
- Mapping Server (MS): a device that implements the mapping database distribution, which maintains the mapping between EID and RLOC.
- Mapping Resolver (MR): a device that accepts encapsulated map-request messages sent by ITR, then decapsulates them and forwards to MS.

Here is an example to illustrate the working flow of LISP. Let's suppose that there are two mobile nodes A and B in two different LISP-enabled networks where A wants to communicate with B. First of all, B provides its EID and RLOC to ETR, which registers them in MS. Now A send out packets to ITR using its own EID as source IP and the B's EID as destination IP, ITR sends map-requests (essentially A's EID and B's EID) to MR, MR then decapsulates the requests and sends to MS, MS retrieve both A's and B's RLOCs and sends back to ITR through MR. ITR now encapsulates the packets with A's RLOC as source IP, then encapsulates again with B's RLOC as destination IP, and sends them out to ETR. After ETR receives the packets, it sends queries to MS to retrieve B's EID, then delivers the packets to B [9]. The whole process is shown in Fig. 5.

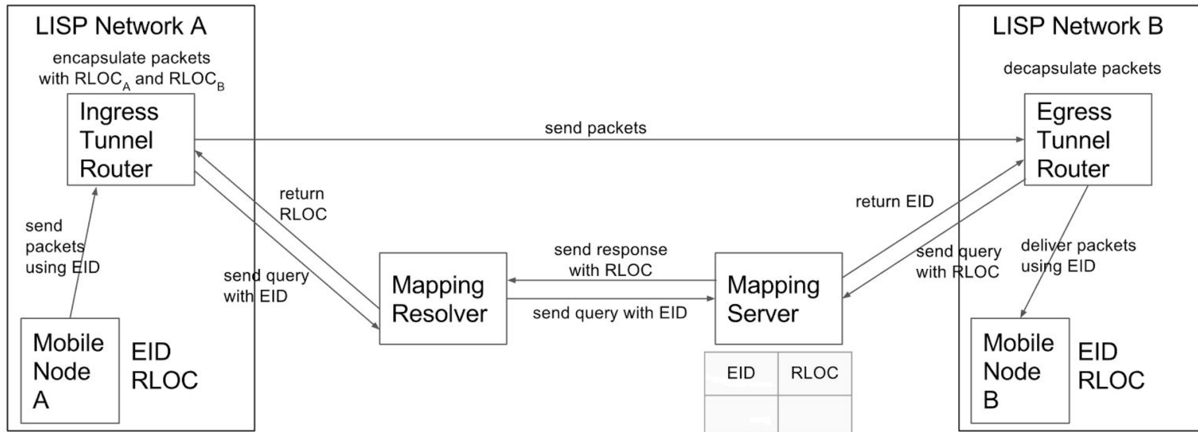

Figure 5. Work flow of LISP

3.3.3 Analysis

The biggest disadvantage for LISP is that it cannot be built directly on the current IP infrastructures. In order to support LISP, many LISP-specific components would need to be set up, including xTRs, MSs, and MRs. For a LISP mobile node to communicate with a node that is not LISP enabled, proxy xTRs must be set up as well.

Though LISP cannot work on the existing IP infrastructure directly, it is compatible with both IPv4 and IPv6. Because LISP achieves mobility by maintaining the mapping between EID and RLOC, it is similar to HIP, as long as the mapping is maintained, it would work for any kind of IP address. Another advantage of LISP is that there exists a testbed, the LISP Beta Network (https://www.lisp4.net/beta-network/), to research the real-world behavior of LISP. Participants, which include Cisco, Microsoft, and UCLA, host the necessary components worldwide.

3.4 Comparison

The biggest similarity that MIP, HIP and LISP have is that they are all doing enhancements to IP instead of abandoning it and building a new protocol. The biggest advantage of it is that the existing IP infrastructure does not need to be changed, but the original design of IP limits its mobility, so all of them maintain the mapping between a mobile node's identity and location to achieve mobility. And since the mapping, instead of IP address itself, is the key to solve the mobility problem, the three protocols are all compatible with IPv4 and IPv6. Another thing they have in common is that all of them require excessive network elements to be built into the

existing infrastructure to have them running. For MIP and HIP, they both need rendezvous servers, and for LISP, mapping servers, mapping resolvers, egress tunnel routers, and ingress tunnel routers are needed.

Though they all aim to solve the mobility problem, the approaches they take are different. To be specific, the properties on the two sides of the mapping are different for each protocol. MIP maintains the mapping between a permanent IP address and a temporary IP address, HIP maintains the mapping between HIT and IP address, and LISP maintains the mapping between EID and RLOC.

4 Mobility Implementations

4.1 MIP

The mip6d-ng project is an implementation of Mobile IPv6 for Linux and Android. Unfortunately, the custom ROMs for Android provided on their website were built for only two models. They also provide a mip6d-ng port for Android, but there is no documentation to guide us.

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

We also contacted the one of the main contributors of this project, Professor Lazlo Bokor from Budapest University, to understand the current state of the project. He informed us that this project has been deprecated, and that there are currently no resources available for helping us with this project. Therefore, we moved on to finding other implementations of mobility on Android.

4.2 HIP

Host Identity Protocol for Linux (HIPL) is an experimental open source implementation of HIP. The HIPL team was also working on porting their implementation to Android. However, they have not touched this since 2013, and it was still under development. Here are the resources that we used:

HIPL website:
http://infrahip.hiit.fi/

HIPL developers mailing list:
https://www.freelists.org/list/hipl-dev

HIPL users mailing list:
https://www.freelists.org/list/hipl-users

Since there was no workable prototype available for HIP we couldn't work on improving any solutions.

4.3 LISP

Previously known as the LISPmob.org project, OpenOverlayRouter (OOR) is an open-source implementation to deploy programmable overlay networks. In the current version, OOR uses LISP as the control plane. OOR has support for devices to operate as LISP-MN, xTR, MS/MR, or RTR. OOR runs on Linux desktops, OpenWRT home routers, and Android devices.

For the purposes of this project, we compiled and installed the OOR Android application for both rooted and unrooted devices operating as LISP-MN. In order to test OOR, we registered an EID with the LISP Beta network, which is a multi-company, multi-vendor effort to research the real world behavior of LISP.

OOR website:
http://openoverlayrouter.org/

OOR Android application compilation and installation:
https://github.com/OpenOverlayRouter/oor/blob/master/README.android.md

LISP Beta network allocation:
https://github.com/OpenOverlayRouter/oor/wiki/Beta-network

Discussion board for issues and questions of OOR:
http://webchat.freenode.net/?randomnick=1channels=#openoverlayrouter&prompt=1

4.3.1 Rooted Version

For rooted Android devices, the application has superuser access to the routing tables stored in the kernel. Therefore, in the rooted version of the OOR Android application, the routing tables are modified directly.

Clicking the start button triggers an event that is attached to a method in OOR.java class. During the build of the rooted version, the entire C library for LISP is built into one executable shared object, liboor.so. This is executed by OOR.java and it also starts OORService.java.

OORService.java tries to check if the execution of successfully started LISP service. This whole process is explained below.

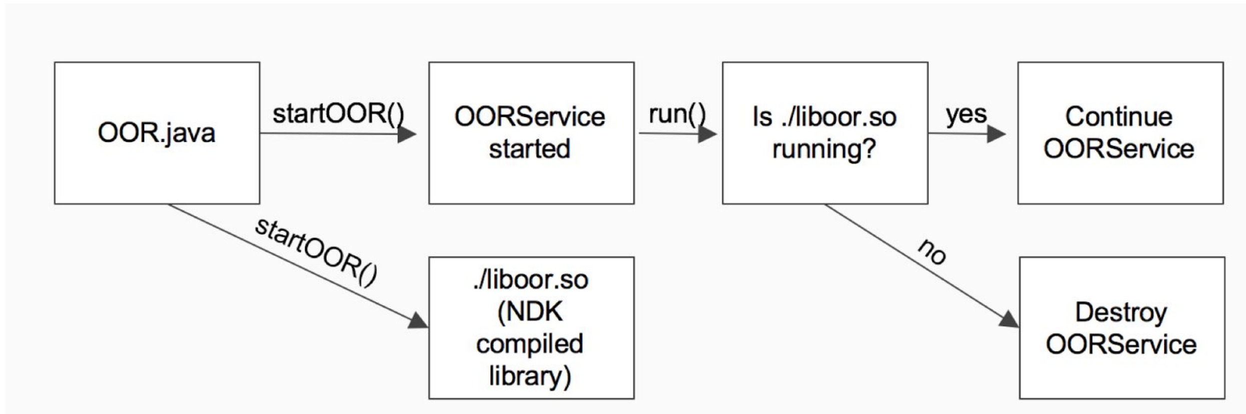

Figure 6. Code flow of the rooted version of OOR

While trying to run this version of OOR we encountered various bugs. We could fix many of the bugs by following appropriate steps listed below.

- Build Failure: this version was not building at all from source. We added the necessary dependencies and configurations needed to build the software:
    - Add the following to build.xml:

```
<target name="-pre-build">
    <exec executable="${ndk.dir}/ndk-build" dir="./jni/" failonerror="true">
        <arg value="NDK_DEBUG=1"/>
    </exec>
    <move file="./libs/armeabi/oor" tofile="./libs/armeabi/liboor.so"/>
<move file="./libs/arm64-v8a/oor" tofile="./libs/arm64-v8a/liboor.so"/>
<move file="./libs/armeabi-v7a/oor" tofile="./libs/armeabi-v7a/liboor.so"/>
<move file="./libs/mips/oor" tofile="./libs/mips/liboor.so"/>
<move file="./libs/mips64/oor" tofile="./libs/mips64/liboor.so"/>
<move file="./libs/x86/oor" tofile="./libs/x86/liboor.so"/>
<move file="./libs/x86_64/oor" tofile="./libs/x86_64/liboor.so"/>
</target>


<target name="clean" depends="android_rules.clean">
    <exec executable="${ndk.dir}/ndk-build" dir="./jni/" failonerror="true">
        <arg value="clean"/>
    </exec>
</target>
```

- Logs not getting written to the log file. Also added support for custom logs:

```
File sdcardDir = Environment.getExternalStorageDirectory();
log_file = new File(sdcardDir, "oor.log");
```

```java
public void displayMessage(String message, boolean cancelAble) {
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setTitle("Attention");
    builder.setMessage(message);
    builder.setCancelable(cancelAble);
    builder.setPositiveButton("OK", new DialogInterface.OnClickListener() {
        public void onClick( DialogInterface dialog, int id ) { dialog.dismiss(); }
    } );

    if ( cancelAble ) {
        builder.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id) { dialog.dismiss(); }
        } );
    }
    AlertDialog alert = builder.create();
    alert.show();

}
public void createLogFile(String message)
{
    /*
     * If a configuration file is not found, a default configuration file is created.
     * */
    try {

        FileWriter fstream = new FileWriter(log_file);
        BufferedWriter out = new BufferedWriter(fstream);
        out.write(message);
        out.close();

    } catch (Exception e) {
        //displayMessage("Error while writing Default Conf file to sdcard!!", false, null);
    }

}
```

- Failure to read data from the standard output:
    - The way code for standard reading from the standard output was written would always make the application hang if there is nothing to read from the standard output. We need to read from stdout because we first list all the processes running on the OS and then we try to figure out from that output, whether LISP process has started or not. We changed the “run” method in SuShell.java to take care of this properly.

```
public String run(String command)
{
    String res  = "";
    try {
        stdin.writeBytes(command+"\n");
        stdin.flush();
        if (stdout.ready()) {
            res = stdout.readLine();
        }
        if (stderr.ready()) {
            res = stdout.readLine();
        }
        createLogFile(res);
    } catch (IOException e) {
        StringWriter errors = new StringWriter();
        e.printStackTrace(new PrintWriter(errors));
        createLogFile(errors.toString());
    }
    return res;
}
```

- Debug issue using custom logs:
    - Since we were working with a massive code base and we had little knowledge of the exact implementation of LISP protocol we used the method of adding log statements to dive deep into the issue.

Sample Log Statement Added to debug:

```
iface = (iface_t *)glist_entry_data(iface_it);
if (iface->ipv4_address){
    OOR_LOG(LDBG_1, "Checkpoint 1 iface: %s", iface->ipv4_address);
}
if(!lisp_addr_is_no_addr(iface->ipv4_address)){
    OOR_LOG(LDBG_1, "Checkpoint 2 iface");
}
if(iface->status == UP){
    OOR_LOG(LDBG_1, "Checkpoint 3 iface");
}
```

After fixing these bugs, we observed that once liboor.so was executed, nothing was printed in the logs. Writing additional print statements in the C code did not make a difference.

We contacted a maintainer of OOR, Professor Albert Lopez, regarding the issue. He explained that OOR for rooted version is not to up to date with Android 5 and later, and that big changes would be required to fix the code. Since OOR also does not work for Android 4.4.1 through Android 4.4.3, we do not have devices with the appropriate systems for rooted OOR to work.

4.3.2 Unrooted Version

Unlike the rooted version, the unrooted version of LISP does not have full access to the network configurations of the cell phone, so it is run as a VPN service. Thanks to that, there are multiple

companies and institutions helping building and maintaining the LISP Beta Network, all of the infrastructures required to run OOR have been already set up and ready to use. So we only need to send a request to the OOR team for the allocation of EID, MS/MR, and other configurations.

Complying with the instructions from the website of OOR, we provided the following information in our request:

- Geographical location: New York, NY;
- Flavor of OOR: Android;
- Make/model: SM-G935U (Samsung Galaxy S7 Edge);
- Use cases: Mobility;
- How we learned about OOR: Research paper on multihoming protocols.

And here is the allocation information provided by the OOR team:

- Device name: columbia-xtr;
- Region: US-East;
- Geographic location: New York - USA;
- EID-prefix: 153.16.29.128/28 (more specifics allowed);
- EID loopback: 153.16.29.129;
- EID-prefix ipv6: 2610:D0:1153::/48 (more specifics allowed);
- EID loopback ipv6: 2610:D0:1153::153:16:29:129;
- Map Servers: {ARIN} {cisco-sjc-mr-ms-1 173.36.254.164, eqx-ash-mr-ms 206.223.132.89};
- Map Server password: wju6C2ZjV3;
- Map Resolvers: {ARIN} {cisco-sjc-mr-ms-1 173.36.254.164, eqx-ash-mr-ms 206.223.132.89};
- PETR: 69.31.31.98.

Besides the configuration data the team provided, they also maintains two websites that allows users to check the registration status and the geographical location of MS/MR.

LISP website for checking if our site is correctly registered into the mapping system:
http://www.lisp4.net/lisp-site

LISPmob website for checking the geographical location of MS/MR:
http://lispmon.net

All we need to do after we have the allocation information is to enter them in the configuration page of the OOP application, but there are some configuration options missing from the information above, we include the full configuration steps here for reference.

Step 1: Download and install the unrooted version of OOR

We used Google Play Store to install the application. We search for "OpenOverlayRouter" in Google Play Store, and installed the one with title "OpenOverlayRouter". Please note that LISPmob and LISPmob ROOT will also appear in searching result, and do not install them, because the LISPmob project has been deprecated.

The OpenOverlayRouter application in Google Play Store:
https://play.google.com/store/apps/details?id=org.openoverlayrouter.noroot&hl=en

We installed the application on Guanlin Zhou's Samsung Galaxy S7 Edge as this was the only cell phone we could use to test the application as the cell phones that Aman gave us were all rooted.

Step 2: Enter the configuration data into the application

First open the application and switch to "CONFIG" tab.
In the EID-IPv4 field, enter 153.16.29.128;
In the dropdown menu of RLOC Interface, choose wlan0 and rmnet_data0 (this is not included in the configuration data provided by the OOR team, wlan0 is for Wi-Fi networks, and rmnet_data0 is for LTE);
In the Map-Resolver field, enter 173.36.254.164;
In the Map-Server field, enter 173.36.254.164;
In the Map-Server Key field, enter wju6C2ZjV3;
In the Proxy ETR address field, enter 69.31.31.98;
Then press the button "UPDATE CONFIGURATION" to apply these changes.

Step 3: Connect to a network and run the application

We then connect to a Wi-Fi network and switch to the "OOR" tab in the application, and click on the icon in the middle of the screen, then we can see the message at the bottom saying that "OOR is running".

The code designed to do the actual logic of OOR is written in C, which is compiled by Android Native Development Kit (Android NDK) and integrated with the code for UI and application logic written in Java through Java Native Interface (JNI).

To start the application, the user needs to click on the start button, which is designed as the icon of OOR. This event will trigger the method onClick(View V) in OOR.java, which is shown in Fig. 7.

```
public void onClick(View V) {
        CheckBox oorCheckBox = (CheckBox)findViewById(R.id.startStopCheckbox);
        if (V == findViewById(R.id.startStopCheckbox)) {
                if (oorCheckBox.isChecked()) {
                        startVPN();
                        return;
                }
                showMessage(this.getString(R.string.askStopServiceString),
                                true, new Runnable() { public void run() {
                                        stopVPN();
                                        oorWasRunning = false;
                                }
                });
        }
}
```

Figure 7. The code of onClick(View V) in OOR.java

As we can see in Fig. 7, the program checks whether the checkbox is checked, this is because that the start button is designed as a checkbox internally. If the checkbox is checked, i.e., the start button is pressed, then it calls a member method startVPN(); If the checkbox is not checked, i.e., the start button is not pressed, it stops the application by calling stopVPN() method. While we are testing, the application launches successfully, so from now on we only concentrate on the logics of startVPN(), which is shown in Fig. 8.

```
public void startVPN(){
        startVPN = true;
        Intent intent = VpnService.prepare(this);
        if (intent != null) {
                startActivityForResult(intent, VPN_SER);
        } else {
                onActivityResult(VPN_SER, RESULT_OK, null);
        }
}
```

Figure 8. The code of startVPN() in OOR.java

In the method startVPN(), it uses either of the two methods startActivityForResult(intent, VPN_SER) or onActivityResult(VPN_SER, RESULT_OK, null) provided by the Android API

to create the VPN service based on whether the intent is null or not. The method it chooses then calls the method run() in OORVPNService.java, which is shown in Fig. 9.

```
public synchronized void run(){

        String storage_path = Environment.getExternalStorageDirectory().getAbsolutePath()+"/";
        jni = new OOR_JNI(this);

        try {
                // Create a DatagramChannel as the VPN tunnel.
                this.configure();
                int tunfd = mInterface.detachFd();

                vpn_running = true;

                if (jni.oor_start(tunfd, storage_path) != 1){
                        Log.e(TAG, "OOR error, check configuration file");
                        this.onDestroy();
                        return;
                }

                System.out.println("====> Starting OOR event loop ");
                jni.oor_loop();

                System.out.println(" ************************** END ************************** ");

        }catch(IllegalArgumentException e){
                Log.e(TAG, e.getMessage());
        }catch (Exception e) {
                e.printStackTrace();
        }finally{
                if (vpn_running == true){
                        vpn_running = false;
                        jni.oor_exit();
                }
                mThread = null;
                vpn_running = false;

        }

        return;
}
```

Figure 9. The code of run() in OORVPNService.java

As shown in Fig. 9, the run() method first creates an instance of the class OOR_JNI, then uses JNI to call the function oor_start(tunFD, storage_path) in oor_jni.c to start the logic for network of the application, and it also calls oor_loop() to keep the application running.

The code files where the screenshots above were taken are in our Github repository, which can be found by following the link below:
https://github.com/kevinshen/oor/tree/master/android/noroot/src/org/openoverlayrouter/noroot

The whole process of the unrooted version of OOR from user pressing start button to start the application to the application is launched successfully is shown in Fig. 10.

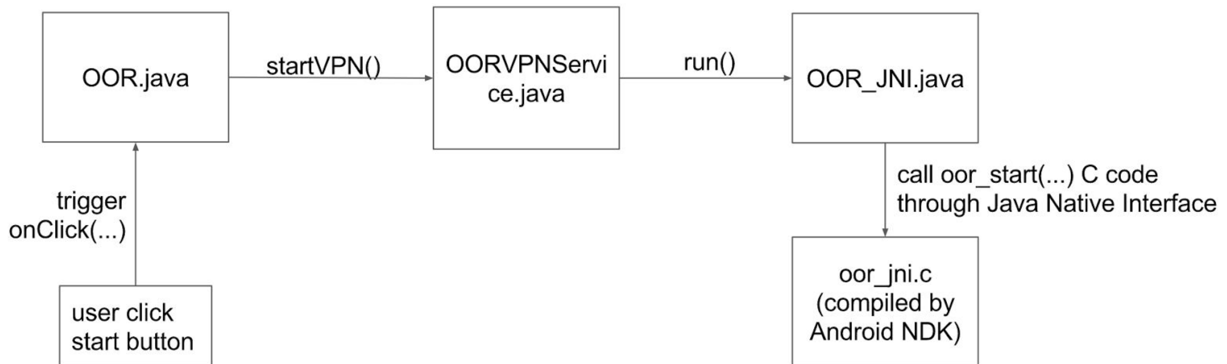

Figure 10. Code flow of the unrooted version of OOR

We tested the unrooted version of OOR under various networks including Wi-Fi networks, Mobile Hotspot, and LTE. We conducted the test by pinging the IP address 153.16.5.1, which is an active EID in the LISP Beta Network, to see if we can receive packets under those networks.

When we were connected to Wi-Fi network "Columbia University", and used Ping and DNS to ping the EID, as well as monitored the log file that OOR generated during that time. The log file shows that we successfully retrieved the RLOC of the correspondent node, and in Ping and DNS, we managed to get all packets back. The testing result is shown in Fig. 11.

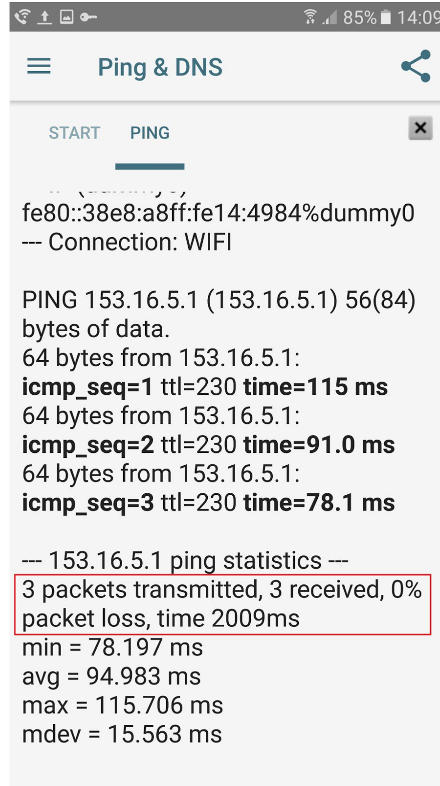

Figure 11. Testing result of the unrooted version of OOR under Wi-Fi "Columbia University"

When we connected to other networks including all other Wi-Fi networks we can reach, Mobile Hotspot, and LTE, and we ran the same test, however, we can not get any packets back. The testing result is shown in Fig. 12.

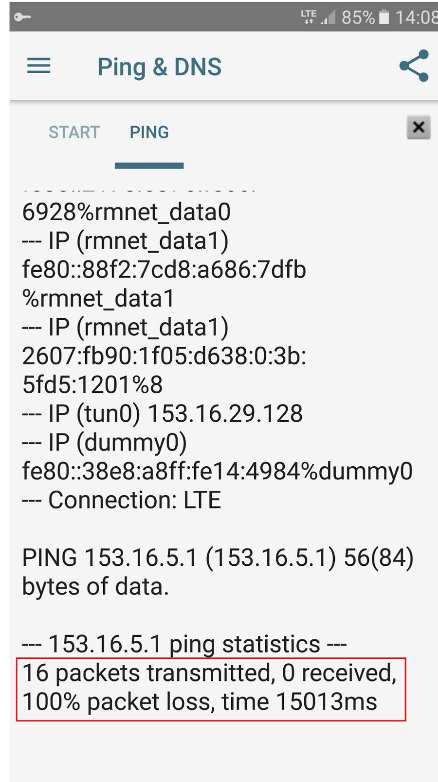

Figure 12. Testing result of the unrooted version of OOR under other networks

We further looked through the descriptions of OOR on its official website as well as the design documents in its Github repository, we found the reason is that the current version of OOR does not support IP addresses that are not publicly routable, for example, IP addresses that uses Network Address Translation (NAT) are behind firewalls.

We also analyzed why the unrooted version of OOR does not work in our specific case. Let's say our device is mobile node A, and the correspondent node is mobile node B. Very much similar to the work flow of unrooted version of OOR under general cases, B first registers its EID and RLOC into MS through ETR, however, currently B is behind a gateway router, the EID registered is a local IP address. When A sends out packets and goes through the encapsulation of ITR, the packets arrive at the ETR. Now the ETR issues a query to the MS to retrieve the EID corresponding to the RLOC. But at this point, the EID retrieved is B's local IP, and there is a gateway router sitting in between the ETR and B. Currently the ETR only knows B's local IP but does not know the IP of the gateway router, so it can not deliver the packets to B. The whole process is shown in Fig. 13.

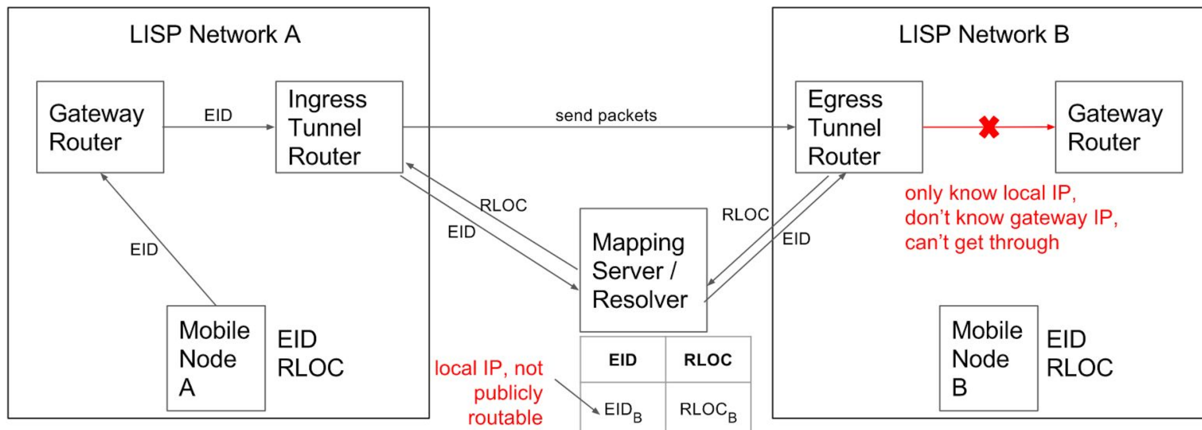

Figure 13. Testing result of the unrooted version of OOR under other networks

Though there is an option to enable NAT traversal in the application, but according to the release note of OOR, that functionality is still experimental and does not have the full functionality. Furthermore, the LISP Beta Network does not have complete support for NAT traversal either. We also reached out the main contributor of the OOR project, Albert López Brescó, he replied that there is only one MS supporting NAT traversal, but it was down, and till the time we write this report, it is still not up online, so unfortunately we can not test the NAT traversal functionality of OOR.

4.3.3 Conclusion

In its current state, OpenOverlayRouter is not an ideal solution for mobility in Android, but it should continue to be looked into. The rooted version of the OOR Android application fails to launch due to the change of implementation of IP routing tables in the Android system, and the unrooted version only works for publicly routable IP addresses.

Therefore, we do not think OOR is a very mature solution for mobility, but we still think it is a promising project, which we will explain in Section 6.

4.4 Open vSwitch (OVS)

4.4.1 Background

Open vSwitch is an open-source implementation of a virtual multilayer switch. OVS is a client that can enable us to perform software-defined networking. This has the built-in capability of mapping a physical IP address to a virtual IP address. This mapping is done in hardware. Here, we define a flow between source and destination. Even if the physical IP address changes, the virtual IP address remains the same. However if both the source and destination nodes are mobile, a mobility manager is required to enable mobility.

The mobility solution is presented by a team at Stanford in the paper "Making Use of All the Networks Around Us: A Case Study in Android."[10] The team prototyped a solution in Android that involves installing an OVS kernel module.

4.4.2 Our progress with OVS

The tasks we aimed to complete were the following:

1. Compile and flash Android kernel with no modifications;
2. Compile and flash Android kernel with small modifications;
3. Compile and flash Android kernel with OVS kernel module.

Due to lack of time, we were not able to complete these tasks. We experienced much difficulty when compiling the Android kernel with no modifications, as the kernel source had many bugs in it. In section 4.4.3, we will go over the steps we followed to compile the Android kernel, and in section 4.4.4, we will show the steps that were provided for compiling the OVS kernel module.

4.4.3 Android kernel compilation

Here we detail the steps and challenges faced when compiling the Android kernel with no modifications.

Step 1: Install Ubuntu in VirtualBox

We followed this tutorial to install an Ubuntu virtual machine:
http://www.simplehelp.net/2015/06/09/how-to-install-ubuntu-on-your-mac/

Step 2: Download Android kernel source

Depending on your device manufacturer, visit the company's open source page to find the correct kernel source to download:

Motorola: http://opensource.motorola.com/
LG: http://opensource.lge.com/osList/list?m=Mc001&s=Sc002
Huawei: http://emui.huawei.com/en/plugin/hwdownload/download
Sony: https://developer.sonymobile.com/downloads/xperia-open-source-archives/
HTC: http://www.htcdev.com/
Samsung: http://opensource.samsung.com/reception.do

Since we had an LG G3 VS985, we downloaded the "LGVS985_Android_Kitkat_V12B" source code found here:
http://opensource.lge.com/osSch/list?types=ALL&search=VS985

Step 3: Install libraries

Run the following commands:
$ sudo apt-get install -y build-essential kernel-package libncurses5-dev bzip2
$ sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0

Step 4: Download toolchain

We cloned this Git repository, which has various Android prebuilt toolchains:
$ git clone git://github.com/DooMLoRD/android_prebuilt_toolchains.git toolchains
Only the arm-eabi-4.4.3/ folder is necessary, so everything else can be deleted.

The arm-eabi toolchain can also be downloaded from the Android GoogleSource website:
$ git clone https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6

In later steps, we refer to the directory of the arm-eabi toolchain as <TOOLCHAIN_DIRECTORY>.

Step 5: Choose defconfig configuration file

In the arch/arm/configs directory of the kernel source, there are many configuration files that end in "defconfig". Select the correct one depending on the platform of your device. In our case, the correct file was "g3-vzw-perf_defconfig". In later steps, we refer to the name of this file as <DEFCONFIG_FILE>.

Step 6: Build kernel

Copy and paste the contents of this build script into a new file in your kernel source directory:
https://github.com/BlackBox-Kernel/blackbox_sprout_mm/blob/BB-MM/bb_build.sh

Change line 36 from:
export CROSS_COMPILE="/home/kunalkene1797/arm-eabi-6.0/bin/arm-eabi-"
To the following:
export CROSS_COMPILE="<TOOLCHAIN_DIRECTORY>/bin/arm-eabi-"

Change line 45 from:
make cyanogenmod_sprout_defconfig
To the following:
make <DEFCONFIG_FILE>

Start building the kernel:
$ ./bb_build.sh

At this step, we ran into error after error while the "make -j4" command was executing, and it appeared that there were many bugs within the source code. Many of these errors were thrown when gcc gave a warning that a certain variable may not have been initialized, even though there exists no case in which the variable would be accessed uninitialized. However, the warning is treated as a "Forbidden warning", so compilation fails. In many cases, initializing these variables isn't allowed, because they have been declared with the const qualifier.

We restarted the entire compilation process with other Android kernels that were available for the LG G3 VS985, including the "LGVS985_G3_Android_Lollipop_V23C" kernel source. We conclude that the kernel source for the LG G3 VS985 contains many bugs that must be fixed in other to be compiled successfully.

4.4.4 OVS kernel module compilation

The steps for compiling the OVS kernel module can be found in the following Google Doc, provided by one of the authors of the paper:
https://docs.google.com/document/u/1/pub?id=1k5jAkz_R475Ohj0OaJdWwSpAw6mmR2Mp_Ggr8_yrXsY

Here, we adapt some of the steps to make it easier for future students to follow.

Step 1: Make sure you have all the build tools

sudo apt-get install libtool autoconf build-essentials

Step 2: Download Open vSwitch

$ git clone https://github.com/openvswitch/ovs.git
$ cd ovs
$ ./boot.sh
$ ./configure ovs_cv_use_linker_sections=no -with-l26=<android_kernel_dir> KARCH=arm --with-rundir=/data/local/var

Step 3: Make change to your Makefile

$ vim datapath/linux-2.6/Makefile.main
Change this line from
$(MAKE) -C $(KSRC) M=$(builddir) modules
To the following:
$(MAKE) -C $(KSRC) M=$(builddir) ARCH=$(ARCH)
CROSS_COMPILE=$(CROSS_COMPILE) modules

Then run the following, where <TOOLCHAIN_DIRECTORY> is the directory of the arm-eabi toolchain:
$ ARCH=arm CROSS_COMPILE=<TOOLCHAIN_DIRECTORY>/bin/arm-eabi- make

Step 4: Find the kernel module under ./datapath/linux-2.6/

$ adb push openvswitch_mod.ko /data/local/tmp
$ adb shell insmod /data/local/tmp/openvswitch_mod.ko
Then you will see <4>[938.237670] Open vSwitch 1.1.0pre2, built Jan 30 2011 23:09:23 in dmesg

Step 5: Generate Needed Files for Android Apps

cd openvswitch
ARCH=arm CROSS_COMPILE=arm-eabi- make

(don't forget to change the datapath/linux-2.6/Makefile.main)
Although it's building open vswitch for the host machine, "make" will also generate all the intermediate files that we will need for building android binary files

Step 6: Download Cross-Compile Script:

http://www.stanford.edu/~huangty/cross_compile_patches.tar.gz
$ tar zxvf cross_compile_patches.tar.gz
$ cd cross_compile_patches/

Change the following variables in gen_android_ovs_app.sh:
ANDROID_NDK=/home/huangty/Research/android/ndk/cyanogenmod/prebuilt/ndk/android-ndk-r4/
ANDROID_NDK_TOOLCHAIN=/home/huangty/Research/android/ndk/cyanogenmod/prebuilt/linux-x86/toolchain/
ANDROID_SYS=/home/huangty/Research/android/ndk/cyanogenmod/
ANDROID_KERNEL=/home/huangty/Research/android/kernel/cm-kernel/
ANDROID_OVS_DIR=/home/huangty/Research/android/openvswitch

Change the following variables in Makefile:
SDKTOOL := /home/huangty/Research/android/sdk/platform-tools
LIBDIR :=
/home/huangty/Research/android/ndk/cyanogenmod/prebuilt/linux-x86/toolchain/arm-eabi-4.4.0/lib
SYSROOT :=
/home/huangty/Research/android/ndk/cyanogenmod/prebuilt/ndk/android-ndk-r4/platforms/android-8/arch-arm

Execute gen_android_ovs_app.sh
$ ./gen_android_ovs_app.sh
It will generate "android_app/" under openvswitch/

Step 7: Build and install

$ cd android_app/
$ make
$ make install

5 Conclusion

Mobility is a complex problem that can be solved by decoupling the dual purpose of IP addresses as identifier and locator. However, the current state of mobility implementations for Android is not as advanced as we expected, but there are solutions that should continue to be investigated. Although no working implementations exist for MIP or HIP, the LISP implementation, OpenOverlayRouter, remains a promising solution to mobility. Open vSwitch should also be looked into further.

We believe that a shift towards unified heterogeneous networking will be increasingly important in the coming years, and mobility is essential to achieving that shift. Taking advantage of all network interfaces and switching between them seamlessly would make a giant leap in enhancing user experience. Today users accept the fact that their connection will get interrupted while switching networks, however if we are able to solve the problem of mobility in an efficient manner(specially for hand-held devices with limited battery) there are many possible use-cases where user will have a much better and uninterrupted experience while using internet.

6 Future Work

6.1 OOR

The current version of OOR contains problems and has limited functionality. However, OOR is an ongoing project, and the OOR team is currently developing and maintaining it. The contributors are very responsive and can offer a lot of useful resources and help.

In the near future, we believe that the OOR team could build NAT traversal into OOR and set up more MSs that support NAT traversal. Thus, it is still a potential solution for mobility.

6.2 OVS

We encountered many problems while compiling the Android kernel, and unfortunately, we could not fix all of them due to lack of time.

We believe the biggest challenge is to compile the OVS kernel module. The paper from Stanford [9] also reports testing the performance of OVS by measuring system load, power consumption, and seamless connectivity. Therefore, once future students manage to compile the OVS kernel module, they should be able to immediately test its functionalities.

6.3 HIPL

The HIP for Linux team was working on porting HIPL to Android when they were last active [9]. If the team resumes work and ports HIPL to Android, then this is also a promising solution to mobility. Here are some resources for the implementation of HIP on Android:

Android port source code:
https://code.launchpad.net/~hipl-core/hipl/android-port

Android port (experimental):
http://infrahip.hiit.fi/hipl/manual/HOWTO.html#android

6.4 Multipath TCP (MPTCP)

MPTCP is an ongoing project designed and developed by the Multipath TCP working group of the Internet Engineering Task Force. MPTCP uses and augments TCP/IP by allowing a TCP connection to use multiple paths [12].

To be specific, the current design of TCP/IP only allows the device to connect to either Wi-Fi or LTE, but not to both at the same time. In MPTCP, a device can connect to both Wi-Fi and LTE at the same time and under the same TCP connection. Because the user is connected to multiple networks in MPTCP, even if one of them is down or out of range, the user can still receive packets from other networks. Thus, MPTCP maintains TCP connections and achieves seamless transitions between networks [13].

We think MPTCP is also a very promising solution for mobility, because unlike other protocols and solutions, MPTCP has already been deployed. For example, Apple's Siri has used MPTCP to switch between different wireless networks seamlessly. Furthermore, there are implementations of MPTCP for Linux, Solaris, iOS, and OS X. Many companies and institutions around the world are actively contributing to the project, including Apple, Oracle, Université catholique de Louvain, Swinburne University of Technology [12].

Unfortunately, most of the implementations for handhelds mentioned above are not open source and hence it is difficult to make any sort of comments on how well MPTCP performs on mobile devices. MPTCP seems promising but and Android implementation in the context of HetNet is yet to be tested out.

7 Acknowledgements

Firstly, we would like to thank Dr. Gaston Ormazabal, who provided us an extraordinary amount of guidance and support throughout the semester. Dr. Ormazabal met with us for several hours every week to check on our progress and help us understand concepts in mobility and networking in general. He also helped us with our presentation skills so that we were prepared for meetings with Prof. Schulzrinne. He was integral to our success with the project, and made the research course an excellent learning experience.

We also would like to thank Aman Singh for joining into our weekly meetings to give us valuable feedback on our progress. He also provided us with Android devices so that we could test the mobility implementations.

Thank you to Professor Henning Schulzrinne for his extremely valuable time and feedback on our project. We really appreciate his guidance throughout the semester.

Lastly, we would like to thank the members of the Control Middleware Team, Kai He, Qing Lan, and Shen Zhu, for helping us understand how intelligent switching between networks can be made a reality using the policy engine.

8 References

[1] Mark Sportack. TCP/IP First-Step. Cisco Press. 2004.
[2] Wikipedia. Mobile IP. https://en.wikipedia.org/wiki/Mobile_IP.
[3] Internet Engineering Task Force. Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents. https://tools.ietf.org/html/rfc3776.html
[4] Oracle. Mobile IP Administration Guide. https://docs.oracle.com/cd/E19455-01/806-7600/index.html.
[5] Wikipedia. Host Identity Protocol. https://en.wikipedia.org/wiki/Host_Identity_Protocol.
[6] Internet Engineering Task Force. Host Identity Protocol Version 2 (HIPv2). https://tools.ietf.org/html/rfc7401.
[7] Internet Engineering Task Force. Host Identity Protocol (HIP) Rendezvous Extension. https://tools.ietf.org/html/rfc5204.
[8] Wikipedia. Locator/Identifier Separation Protocol. https://en.wikipedia.org/wiki/Locator/Identifier_Separation_Protocol.
[9] Internet Engineering Task Force. The Locator/ID Separation Protocol (LISP). https://tools.ietf.org/html/rfc6830.
[10] Kok-Kiong Yap, Te-Yuan Huang, Masayoshi Kobayashi, Yiannis Yiakoumis, Nick McKeown, Sachin Katti, Guru Parulkar. Making Use of All the Networks Around Us: A Case Study in Android. http://yuba.stanford.edu/~huangty/cellnet12-yap.pdf. ACM SIGCOMM 2012. Helsinki.
[11] InfraHIP. HIP FOR LINUX. http://infrahip.hiit.fi/.
[12] Wikipedia. Multipath TCP. https://en.wikipedia.org/wiki/Multipath_TCP.
[13] Olivier Bonaventure, Mark Handley, Costin Raiciu. An Overview of Multipath TCP. http://www0.cs.ucl.ac.uk/staff/M.Handley/papers/9346-login1210_bonaventure.pdf.