Faculty of Engineering & Architecture
Master in Industrial Sciences: Electronics-ICT

Academic year 2013–2014

Master thesis

Implementation of Recursive InterNetworking Architecture on Android

Mathieu Devos

Promotor:	Prof. Dr. Ir. D. Colle
Copromotor:	Dr. D. Staessens
Internal Promotor:	Dr. Ir. K. Casier
Supervisor:	Ing. S. Vrijders

Nederlandstalig abstract

Nederlandstalige abstract

Abstract

In this literature study we will attempt to clearly state the research question and provide adequate background. This should provide us with enough information to start working on the actual research problem. We will provide a roadmap trying to answer this question and the clearly map the path that will be followed and where potential issues might arise. We will also elaborate on why this research question is relevant at this current time.

The literature study is constructed as follows. First we will describe the origin of the internet and look at how the internet got to where it is right now. In the same chapter we will inspect some other alternatives proposed for internet throughout the years. We will conclude the chapter with some of the shortcomings the current internet model faces. In the following chapter we will state the main research question alongside with the specifics this question poses. Following the main question we will explain the basics of Recursive InterNetworking Architecture (RINA), take a closer look at the IRATI implementation of RINA and finish with the restrictions we might encounter for the Android platform. The next chapter will be specifically about RINA on the android platform with specific sections about the wireless Shim-DIF and WiFi Media Access Control. The final chapter intends to draw a conclusion about the literature study and show the position of the study in the whole thesis.

Table of contents

Chapter 1

Architecture and source of the Internet

Starting of this study is done with a small introduction. Since the thesis and the research question 2.1 focus on networking aspect and more specifically on Internet we must first declare what exactly this is how it has come to be. In this chapter we will handle the origin and evolution of the Internet, other alternatives to Internet and some of the issues with the current Internet.

1.1 Origin and evolution of the current Internet

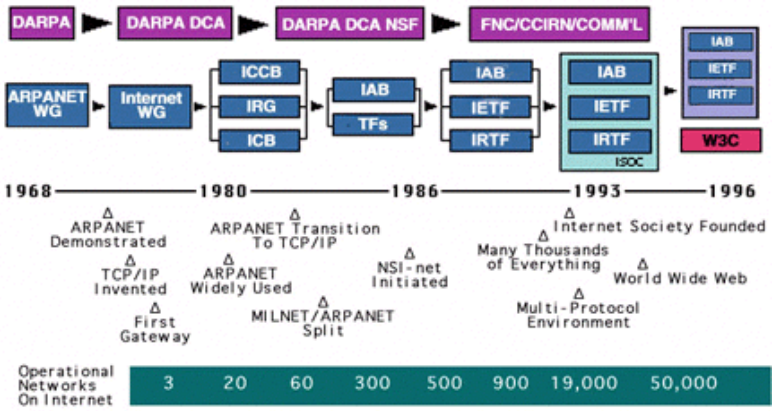


Figure 1.1: Timeline of the Internet (internetsociety.org, 2014)

Internet started out as research aimed at packet switch in the early 1960s. The name of the very first packet switching network was called ARPANET (Advanced Research Projects Agency NETwork), this was a data network established in the USA. While research began early 1960s, it was first established in 1969. The main difference between circuit switching and packet switching is that in packet switching one line of communication can be used for many different packets. These packets can have different sources and/or different destinations, this formed the base of packet switching networks. The first ever packet switched network was set up in California on 29 October 1969 (Salus, 1995). It was commonly thought that ARPANET was set up to survive a nuclear blast, but in fact it was set up as a proof-of-concept and could handle a partial network failure (Hafner, 1998). Later it became clear that this type of network was very robust and was praised for it's ability to withstand partial network failures (Abbate, 2000).

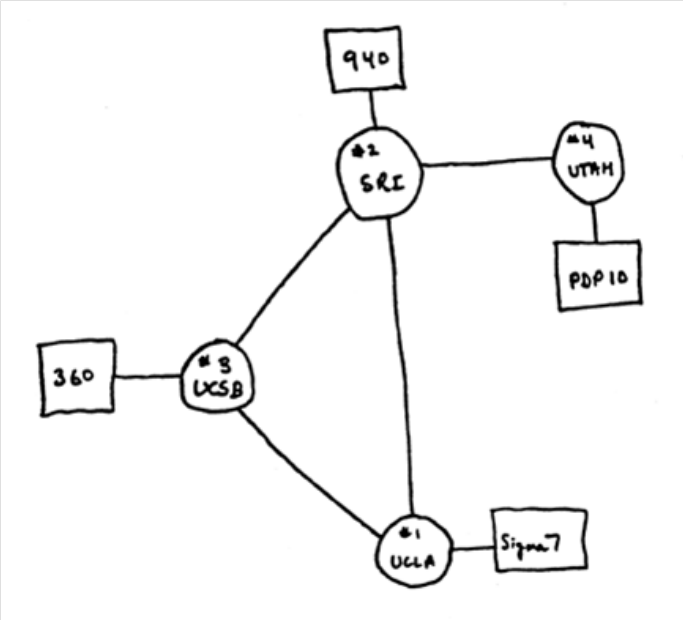


Figure 1.2: Early version of ARPANET (Staessens, 2014)

After the first proof of concept ARPANET quickly expanded during the 1970s. It both expanded on the protocol it used and on the amount of connected nodes. We must first notice that in ARPANET there was no talk of client/server, it was originally designed as a peer-to-peer network (Berners-Lee, 2000; Salus, 2008). ARPANET also went outside the boundaries of the USA when it connected to a Norwegian node in 1973. Later other nodes were included such as a node in Britain, Sweden, ... (see image 1.3. The

packet switching network left the proof-of-concept phase in 1975 when it was declared operational (Salus, 2008). While the technology that ARPANET was run on is currently unimportant, the significance of the protocol that was used in this early network deemed to be humongous.

When ARPANET was first launched in 1969 it used the 1822 protocol (Heart et al., 1970), it was named after the report number. This protocol was designed to work cross-architecture and consisted of several fields: a message type, host address (numerical), and a data field. Messages were sent across the network using early routers, called: *Interface Message Processors*. These devices were the early versions of routers. The entire system worked with either a direct, local link where messages were unicasted or were further broadcasted to other IMPs. Once the message had been successfully delivered and acknowledgment was sent back across the network to the sender. ARPANET was entirely designed for this protocol but on top of this protocol the NCP (Network Control Program) protocol was added which in essence meant that more layers added. These protocols were deemed outdated in 1983 when the transition was made to TCP/IP a huge amount of changes were required[1].

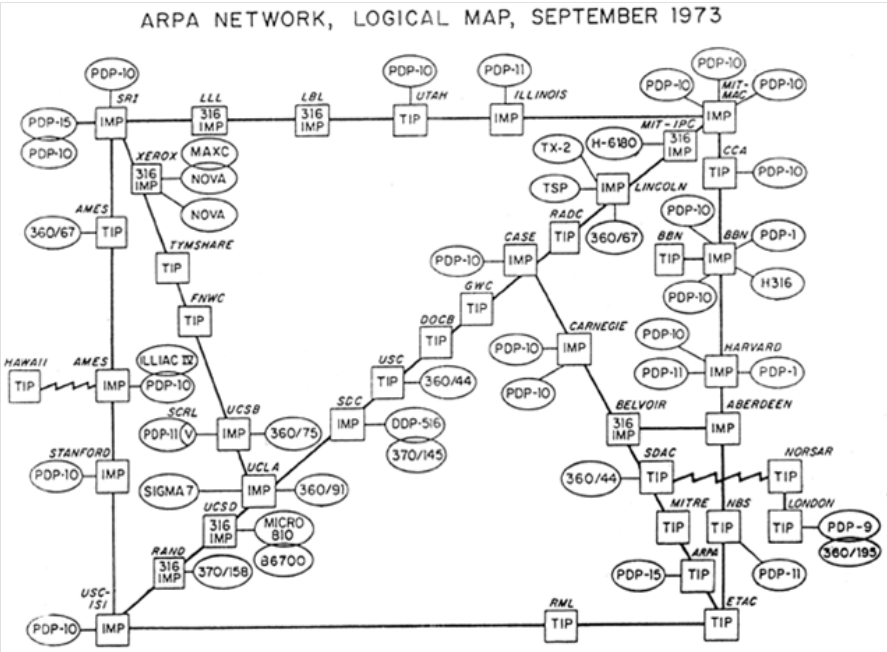


Figure 1.3: Late version of ARPANET (Staessens, 2014)

[1]Hence why this is one of the most notable *flag days* in history of software

In 1983 the transition was made towards the currently used TCP/IP protocol. This marked the start of the early *Internet*. TCP/IP stands for Transport Control Protocol / Internet Protocol. This means it is in fact two protocols, transport layer protocol (TCP) and Internet Layer protocol (IP). ARPANET also allowed for layering of higher-level protocols and this is where the OSI model was born. ARPANET decommissioned in 1990 when the transition towards the current Internet had been made. This was mainly due the rise of ISPs (Internet Service Providers) in the late 1980s and 1990s. We can see that ARPANET was essential in the birth of the current Internet.

In later years the Internet became a standardized product. Several standardization bodies regulate the current Internet. The one responsible for the TCP/IP standards is the Internet Engineering Task Force (IETF[1]). While ISO (International Organization for Standards) is responsible for the overarching 7-layer model. Since the bottom part of the 5-layer model and the 7-layer model is exactly the same and we will be working close to the bottom, we will not be discussing the differences between these models further.

1.2 Previous alternatives

One of the most notable alternatives to ARPANET was the French-developed CYCLADES. It was created shortly after the birth of ARPANET. The main reason of this research project was to explore alternatives to ARPANET. The core principle was still the same though as also this network was a packet switching network. Some concepts from CYCLADES were later applied to the current version of the Internet, such as: host-responsibility and end-to-end protocol.

At the time in the 1970s several research networks were developed. While all these networks were packet switching, the main point of argument was the role of network or host. Either the network or the host had to be responsible to deliver the packet and this divided the early networks in two groups. Other early research networks on packet switching were: DECnet, EIN nee COST II, EPSS, GEIS, IPX/SPX, Merit Network, ... These networks were very similar to ARPANET and did not feature any noticeable new changes. The most important of these alternatives was CYCLADES and most of the flaws that ARPANET showed were filled by using technology and research from CYCLADES.

[1] `http://www.ietf.org/`

1.3 Shortcomings of the current Internet

The current model of Internet shows quite a few shortcomings. The main reason here is that the original protocols were never revised or alternatives were never considered. This static approach has lead to the use of a lot of hacks, patches, band-aids, ... A very recent and obvious example is the need currently required to change from IPv4 towards IPv6. The reason for this is that we are simply running out of IPv4 addresses and thus we need another protocol to handle this. The question then rises: why do we need so many changes to a system that should be scalable? This questions can easily be answered when we look at the history of the Internet. It is based on a rigid system with very little science behind it. When we take a close look at the OSI model1.4 we see a huge number of protocols who all have to work in conjunction with each other and overlap in several occasions[1].

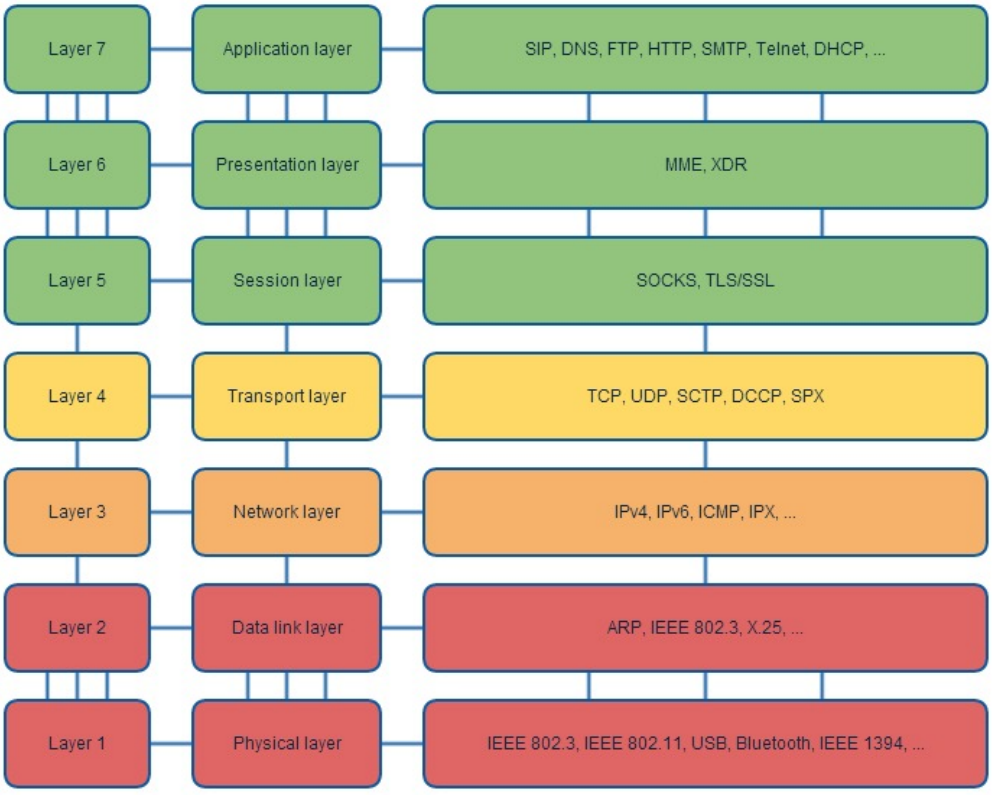


Figure 1.4: OSI model with examples

[1]Example: IPv4 and IPv6 overlap

Some examples of these issues in the current Internet are: multihoming, denial-of-service, port mapping, NAT (Network Address Translation), IP geomapping, ... While some of these issues are solvable, they require a number of band-aids on the current Internet. This stems from the origin of the Internet. While it started out as a research net it was almost instantly made to be a production network. This was done while other networks, such as CYCLADES, were still researching and solving obvious problems. Due to faulty decisions the Internet continued to be build on these protocols and thus became flawed from the start. For example it is currently impossible to send a packet to two unique destinations (IP addresses), this requires you to send two different packets. This also causes problems with mobility where targets change IP addresses quickly when running through different Internet providing cells. Handovers from routers is a big hassle for mobile users and slows down the entire process of staying online permanently. While short interruptions are not an issue when loading a website, it can prove fatal when using this Internet for live communications, monitoring, ... Proper multihoming can potentially fix this issue, but this is currently very hard to achieve in the current state of the Internet.

Another example is the use of NAT, Network Address Translation. This is needed because we are currently running out of IPv4 addresses and it enables small networks to have multiple nodes connected to the Internet at the same time. The problem is that this only solves issues in one direction. People from outside of the network can no longer see the people behind the NAT-router. This causes issues for several applications, such as peer-to-peer applications and many others. It also means that the router is actually breaching the model because it alters packets (specific port number and ip). In the normal model this is never changed between end hosts.

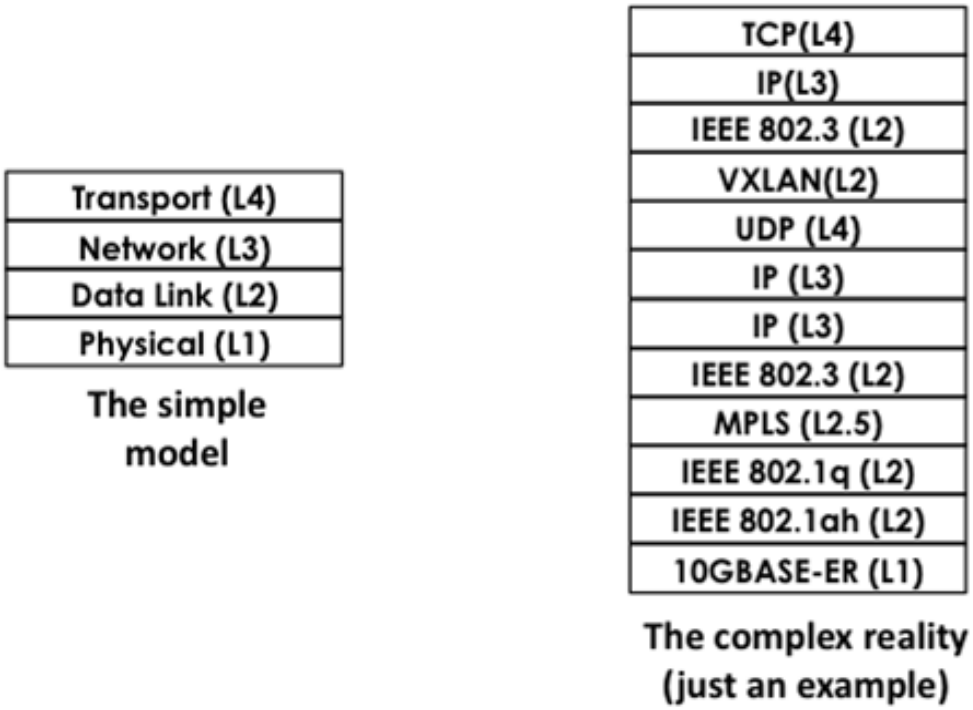


Figure 1.5: Example of the current Internet layer system (Staessens, 2014)

Further issues become apparent when looking at common items in some layers. We see that it is possible to map IP-addresses to geological information, thus violating privacy of the people using the Internet. Secondly when an attacker obtains an IP-address it becomes quite easy for this attacker to spam this IP with data causing a disruption in the service of the host. This is known as: Denial-of-Service Attack. Other mapping problems are the common application-port mapping that occurs. When anybody intercepts a packet he can read the port number and deduce for what application that packet can be used. This leads to more privacy issues and limits applications in their use as they are not free to choose a port number.

We see that the current Internet has quite a lot of issues and can use a general, uniform model or architecture as an answer. Continuing down the road of constantly applying band-aids to a already broken system is clearly not an answer. A clear slate is need and this is where the research question will try to find an answer to.

Chapter 2

RINA alternative

"Networking is Inter Process Communication (IPC) and IPC only"

John Day, *Patterns in Network Architecture: A return to fundamentals*

In this chapter we will address the alternative for the current Internet. This alternative is called *Recursive InterNetworking Architecture* (RINA). We will start this chapter off with the main research question and state how this question will be answered. Following this we will take a closer look at RINA, both the function and the history will be discussed. The followup section will delve further into RINA and look towards the current technical implementation we are researching, *Investigating RINA as Alternative to TCP/IP* (IRATI). After this section the reader is capable of comprehending how RINA functions and what the main research question is.

2.1 Research question

We will first present the research question as this will clarify what will be researched, what should be developed and what answers we are looking for. This is applicable for both the literature study as well as for the entire thesis.

The research question is stated as follows:

How to run RINA on Android over WiFi?

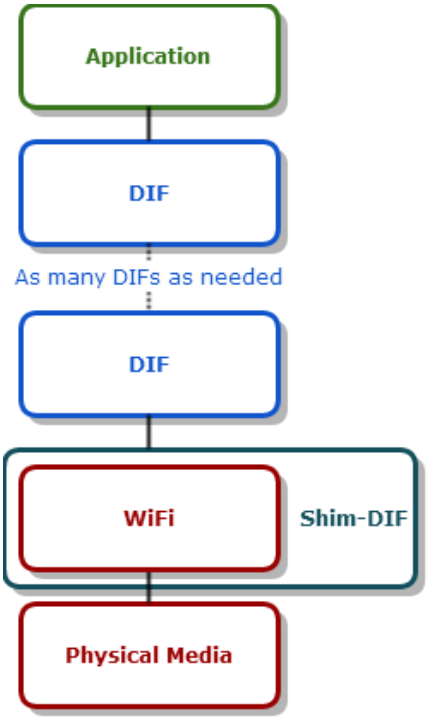


Figure 2.1: RINA over WiFi

This is ultimately the question we are trying to answer. This question alone does not provide enough background and will need some further elaboration, which will be provided in this section.

First we must note that RINA is the theory, actual working models are currently very scarce. We thus require a technical implementation of RINA. This will not be self constructed, we will be building further on a provided codebase. This project is *Investigating RINA as Alternative to TCP/IP* (IRATI) (Vrijders et al., 2014). This European project is a collaboration between:

i2CAT Foundation `http://www.i2cat.net/en`

Nextworks `http://www.nextworks.it`

iMinds `http://www.iminds.be/en`

Interoute `http://www.interoute.com/`

and is trying to bring a codebase for RINA upon which commercial implementations can be based. A working Recursive InterNetworking Architecture has already been developed

for Linux operating systems. Since the Android platform is based on a trimmed and edited version of the Linux platform we will use the previously established code as a base.

A current working Shim-DIF (more about specific working of RINA in 2.2) has been constructed for 802.1q (basic Ethernet) on Linux operating systems. A big portion of this research question is how to port the IRATI prototype on Android, more specifically to extend the prototype towards a working WiFi Shim-DIF. A key point for this Shim-DIF and the thesis as a whole is the need to be dependency free. When this Shim-DIF is dependency free it will guarantee the full and seamless working of RINA on mobile devices, specifically on the Android platform. Other physical connection mechanisms for Android such as: 3G, 4G, ... This is not part of this thesis.

Another part of the research is dedicated towards the differences between the Linux and Android platform. Since IRATI is build on mostly C in kernelspace and C++ and Java in userspace. These coding languages are well supported in Linux and in various other platforms. The issue here rises that the library that handles C++, *glibc*, is not available in the Android operating system. On Android a variant of this library is available, called *bionic*, this library however has more limited functionality, specifically some alterations on how C++ is handled.

One final piece of research that will be completed is to figure out how to optimally map RINA API to the current WiFi standard. For this we will take a closer look at the current WiFi standard (802.11n) (Ortiz, 2009; Paul and Ogunfunmi, 2008; Perahia, 2008). We will examine the interaction this standard makes with several layers and where the Shim-DIF will some remodeling to seamlessly fit on the current WiFi standard.

This research question is now clearly stated and throughout this literature study we will try find the needed background information to help us formulate an answer in the thesis. The answer will not only be limited to pure text form, but shall also include working code for the prototype of the *Shim-DIF over WiFi on Android*.

2.2 RINA basics and origin

Before we go any further we initiate with clearly stating what *Recursive InterNetworking Architecture* (RINA) is, how it functions and what the origin of RINA is. As has been shown in section 1.3, the current status of the Internet is facing a long list of issues. This is where RINA provides an adequate answer, it looks at previous network architectures

and tries learn from those. In the end RINA proposes a basic, clear, and adequate answer for the networking needs.

RINA is a proposed architecture by John Day in his 2008 book: "Patterns in Network Architecture: A return to fundamentals" (Day, 2008). Let us start by explaining the full name of RINA: *Recursive InterNetworking Architecture*, first the last part: *Architecture*, this states what the actual goal of this is. To implement an architecture that provides support for Network communication. One of the most important words in RINA is definitely: *Recursive*. As can be seen in the IRATI symbol: a recursive tree. Recursive is defined as: "pertaining to or using a rule or procedure that can be applied repeatedly." (dictionary.com, 2013), this instantly shows that the entire architecture is build on a base that can be repeated as many times as needed. It does not involve a static amount of layers, instead it can use as many or as few as needed to set up communication between two applications. The middle part: *InterNetworking* shows that this architecture involves communication between and on networks and is not limited or restricted to single networks.

The main principle in RINA is that networking is simply and only: *Inter-Process Communication* (IPC) (Day, 2008). This is the premise on which entire RINA is founded on. An IPC is provided by IPC processes, a group of these coherent IPC processes forms a *Distributed IPC Facility* (DIF) through enrollment. Every DIF is managed and has its own scope, it provides a way for processes to communicate between each other. DIFs have the same mechanism but are configured to different policies and thus have their own sets of rules. A DIF can ultimately be seen as a layer. For example the first DIF provides IPC directly to user applications. Below that you have a DIF that takes care of the communication between user A and the router user A is connected to. Every DIF has their own scope and once it becomes clear what the scope is for the DIF, the function of this DIF will be evident and be tailored to that scope. This repetition of DIFs goes on recursively until the entire chain of communication can be formed and finally the application from user A is interacting with the application from user B.

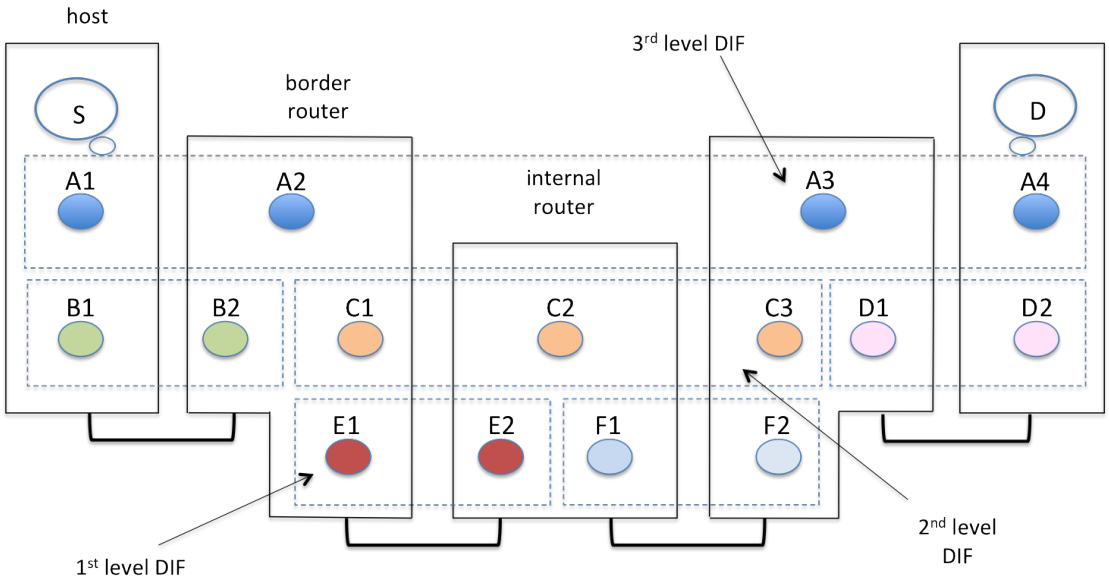


Figure 2.2: Recursive InterNetworking Architecture example (irati.eu, 2013a)

An example how communication between two processes is established with RINA can be seen in the image 2.2 above. Here we see application S on the host trying to communicate with application D on the other side of the communication chain. We see that for this specific example 3 levels of DIFs are used. These DIFs stack recursively from the bottom level (1st level) up till the DIF that actually takes care of the communication between the IPC and the application process.

In this architecture we see that every application process, which also includes the IPC processes, has a name this is unique in that DIF. This is the way routing is ultimately done, but it no longer forces users to link applications to port numbers or specific IP addresses. While IP addresses can still be used in lower level DIFs for routing, the application does not to be aware of this. Naming stays within the DIF and is not visible outside that layer. In every DIF is a directory that contains these names of the IPC process names and associates these to the application names (layer above the DIF). This is where we instantly see the possibility for multihoming. One application name can have different IPC processes in the layer below, but due to the design this application process does not need to be aware of this. The architecture will choose the most optimal routing and provide communication between the application processes in the layer above that requested it. Finally we must name the way a DIF sees it's surrounding DIFS. A DIF

on a layer above (towards host process) is named the application. The process below the DIF is called the Point of Attachment.

2.3 IRATI implementation

In this section we will address the technical implementation of the RINA model. The implementation that we will be using is developed in the IRATI project. More information on this project can be found in the research question 2.1.
Here we opt to examine how the model is formed, what functions are already developed and which parts are useable yet. Finally we will quickly skim over the functions that are required for successfully finding an answer to our research question.

The IRATI project is trying to build a working, open source, technical implementation of the Recursive InterNetworking Architecture (irati.eu, 2013b). It focuses on tree main pillars:

1. Produce a working DIF over Ethernet (figure 2.3)
2. Provide an alternative to the current Internet that delivers the same services
3. Tackle current problem-cases of Internet with the RINA alternative

These goals engulf the entire project and can be seen as the coarse lines of the IRATI project. To complete all these goals we need to have several more specific subparts, one of these is the recently developed *RINA over Ethernet on Linux Operation Systems* (Vrijders et al., 2013). This is one of the main goals of the project and almost all future work will use this as a foundation to build upon. Here a fully functional RIN (Recursive InterNetwork) architecture is being set up for Linux operating systems. After this is completed and other groundwork has been completed in the IRATI project it is finally the goal to open source this part of the project. This should provide a working Ethernet between UNIX-like OS's and open the road for applications to use RINA if they wish to do so after the release of the source code.

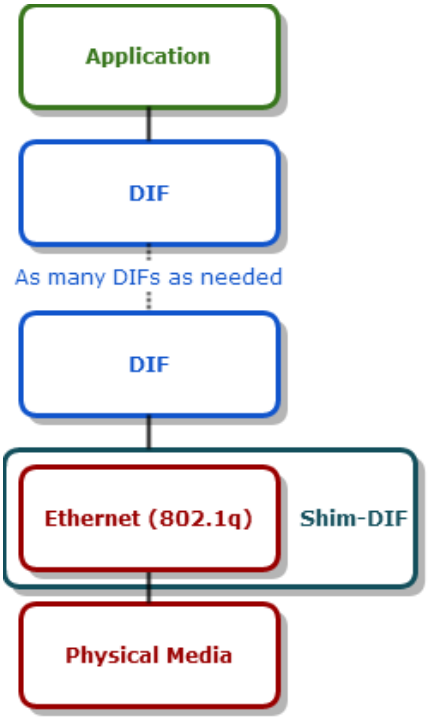


Figure 2.3: Recursive InterNetworking Architecture example over Ethernet

A second component for the IRATI project is the Shim-DIF to handle TCP/IP traffic (figure 2.4). Assuming that RINA will not be used as an island structure, meaning that RINA will still interact with the current implementation of the Internet. More specifically, RINA will still interact with TCP/IP traffic. Here we see a fully function RINA architecture with on top of that the TCP/IP model. One advantage here is that this eases the transition for applications who still work with protocols made to interact with TCP/IP. An example here is game clients that have servers running on specific IP addresses with specific ports. These don't have to instantly change drastically for RINA (no flag day), they are still able to run on TCP/IP over RINA.

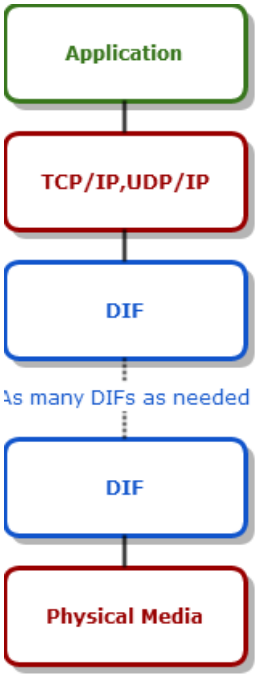


Figure 2.4: RINA example with TCP/IP,UDP/IP on top of it

Once these goals have been acquired the expansion of the project can commence. This development can come from different angles. For example other operating systems can be tackled so the project can be run across multiple operating systems (POSIX threads). Hereby we must state that the current goal is only to incorporate open source systems. The IRATI project is thinking about expanding towards the RIN architecture on JunOS, a FreeBSD-based operating system used in *Juniper Networks (juniper.net, 2013)* routers. Here we can imagine a future where an entire network comprising of JunOS-based routers and workstations with UNIX-like OS's. Both fully equipped with RINA thanks to the IRATI project. This can lead to proper, full scale network testing and no longer pingpong tests between two nodes.

Finally we will briefly touch on where the thesis will fit in this project. The thesis is focused on the WiFi implementation on Android operating systems (figure 2.1). More specifically the thesis will aim to implement the IRATI stack on the Android platform on the 802.11n standard.

Chapter 3

RINA on WiFi

Following the introduction to the RINA alternative in chapter 2 we will now take a closer look to RINA over WiFi on Android. What the exact needs are for this implementation and where special focus will be needed. We kick this chapter off with a section about IEEE 802.11 MAC protocol. Followed by the WiFi Shim-DIF. Finally we will take a closer look at the Android restrictions (compared to Linux).

3.1 IEEE 802.11 Media Access Control

In this section we will address the WiFi MAC[1] (Gast, 2005). Since the protocol covers both layer 1 (physical layer) and a part of layer 2 (Media Access Control) we will limit ourselves here to the layer 2 interaction. The IRATI project code will have to fit seamless on this protocol to ensure maximal optimization for wireless communication. In this light we must instantly make an important remark. In normal operation mode the device drivers for user devices will be set to *STA* mode (Station infrastructure mode). This enables basic device functions, however two devices in STA mode will not be able to communicate with each other unless an Access Point (AP) is presented to communicate with. As can been seen in figure 3.1.

[1] 802.11 Media Access Control

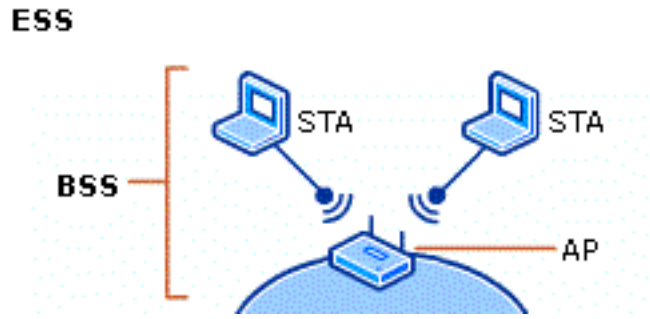


Figure 3.1: 802.11 Infrastructure mode (STA and AP) (technet.microsoft.com, 2014)

Drivers can also function in other modes, some of which will be handled later in this thesis, some who will not be handled at all. Other modes that drivers are capable of are (wireless.kernel.org, 2014):

AccessPoint (AP) infrastructure mode Access Point for a master device in a network, normal mode for WiFi router. *Not handled further*

Monitor (MON) mode A passive mode that allows monitoring of all packets the device receives, can be double used in some devices. *Used in testcases*

Ad-Hoc (IBSS) mode Enables communication between other ad-hoc devices without AP (figure 3.2). *Used in testcases*

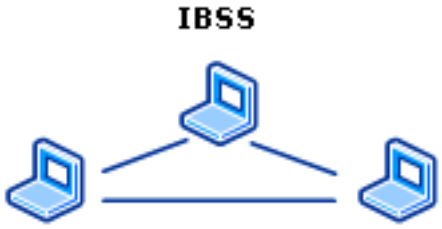


Figure 3.2: 802.11 Ad-hoc mode) (technet.microsoft.com, 2014)

Wireless Distribution System (WDS) mode Enables communication between devices (mostly APs in a single ESS), uses the 4-addresses from the layer-2 header[1]. *Not handled further*

Mesh Enables communication in Wireless Mesh Networks, used to set up intelligent, dynamic routes. *Not handled further*

[1]Other modes only use these partially and mostly leave one or more fields empty

For the initial part of this thesis we will leave the device in either STA or IBSS mode. This means that the drivers will *translate* the 802.11 MAC-headers to 802.1q MAC-headers when in STA mode (see Appendix 6.1.8. If this happens in IBSS mode we will have to test or ask authorized sources[1]. The difference can be seen in the figure below (figure 3.3). We see that the entire header (Ethernet) becomes quite a bit easier to understand in STA mode. Since the IRATI project already has a working Ethernet Shim-DIF this will, under normal conditions, fit on this MAC header in STA mode. When expanding further down the road we will have to implement a Shim-DIF capable of handling the 802.11 MAC header.

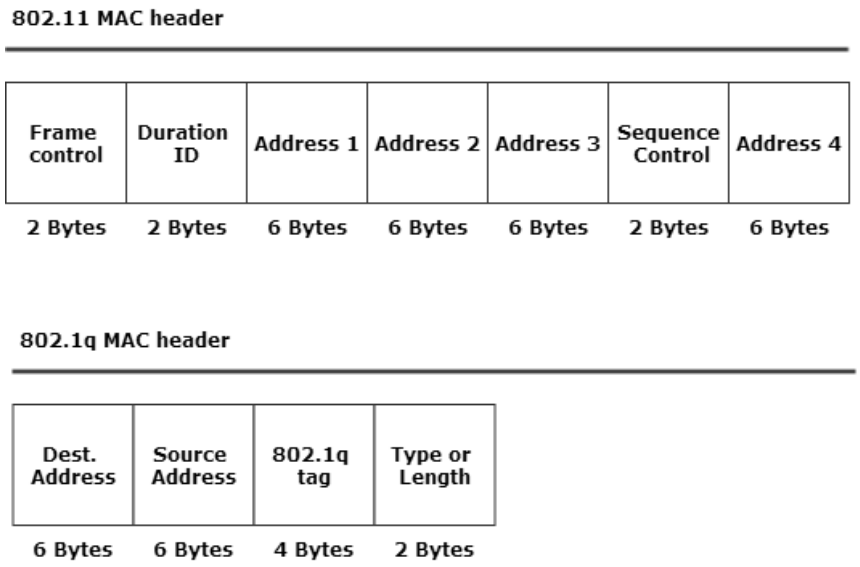


Figure 3.3: 802.11 and 802.1q MAC headers

Since WiFi (802.11) MAC headers have quite a lot more functions (30 bytes compared to 802.1q's 18 bytes) we will have to adjust the code for this. The functions that are added can potentially prove useful for the architecture and thus further study should be implemented. This research will focus on the overlap of functions between the MAC header and the functions provided by the IPC API. It can ultimately prove advantageous to use the full 802.11 header alongside with RINA instead of having drivers reform this header to a more easy to comprehend Ethernet MAC header.

[1]Linux Wireless Kernel group for example

3.2 Shim-DIF for wireless

While under normal operation of the research question we delve further into the Shim-DIF over WiFi (see figure 2.1. However, due to reasons stated in the previous section we clearly see that this has now been reformed to:

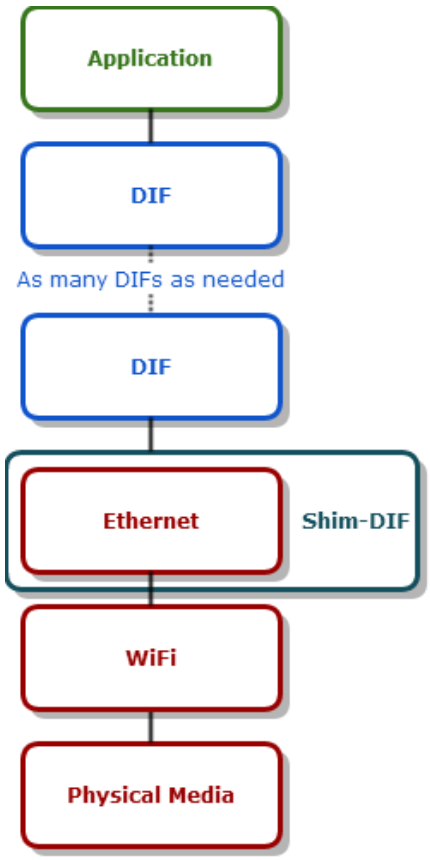


Figure 3.4: RINA over WiFi (updated)

In normal conditions this means we can fully copy the code from IRATI we must remind ourselves that we are still working on another Operating System and this comes with consequences. These are handled in the next section. However if time allows this, we will take a closer look at the 802.11 packets and provide a Shim-DIF for the 802.11 standard (802.11n specifically). This provides a base for other developers who wish to update drivers further down to road to make said drivers compatible with RINA.

3.3 Android restrictions

While Android is a UNIX-like operating system and based on Linux, it does come with some restrictions. In this section we will be comparing Android OS to the Linux OS. The reason for this is that the IRATI project already has a working RINA implementation on Linux and this will be used for the further work of this thesis. Secondly because Android is based on Linux operating system and its kernel. At one point in history the two operating systems shared a common kernel and while this common component is planned, it is currently not the same kernel.

Before we can further inspect the Android operating system, compared to Linux OS, we first need to identify the differences between the *Android kernel* and *Linux kernel*. For most of this information we will be using a talk by John Stultz, comparing Android to a stock Linux kernel (Stultz, 2011; Wiki, 2013a). The amount of changes is not that large as it comprises of around 25 000 lines of changes, compared to the entire 15 millions lines of code for the full kernel. This places the changes around ~0.2% of the total code. Of course some of these changes can have large impacts on the actual kernel itself. These changes will be discussed here. Some of the items in the Android patches include:

- Ashmem
- Binder
- Pmem
- Logger
- Early suspend
- Wakelocks
- Various small *hacks* to facilitate a mobile OS
- ...

Because a Linux kernel is meant for desktop and server hardware (both very similar), Android aims to improve in the hardware department. This allows the use of more and varied hardware through its kernel. A second large point of interest for Android is the power management. While traditional systems are just plugged in with the cable and thus don't need to worry as much about power. The opposite is true for mobile, battery powered devices where power management is of utmost importance. Another change

between the two kernels is the way error reporting is done and the attempt to increase security on the Android kernel. Finally the Android kernel aims to improve performance, especially for its intended users, mobile platforms.
Some of these changes to the original kernel have been talked about quite a bit, such as the wakelocks. However we find that these changes will not impact the current implementation of the IRATI stack into Android. Further down the road additional optimization can be acquired between the Android kernel and the IRATI stack. Also RINA will have to be able to work with Paranoid Networking so that networking does not become blocked by this feature in Android kernels. For this part further research will be conducted.

As shown above, the current version (and future versions) of the Android kernel will not pose any problems for the IRATI stack. However, an operating system does not only comprise of its kernel. This is an important part and the aspect where we see the biggest change between Linux and Android. When looking at a structured overview of an operating system (UNIX-like), we see that libraries play a big part in the undertaking of an operating system. Here we find the biggest change between Android and Linux. While Linux uses glibc (gnu.org, 2013) and Android uses the Bionic library (Contributers, 2013; Wiki, 2013b).

The restrictions from the Bionic library that affect the inclusion of the IRATI stack on Android are mostly limited to the C++ language. The way Bionic handles C++ is quite unique as it aims to alter the use of C++ as a whole. This issue here is the following: IRATI stack is build on C++. For the most part C++ is still supported by the library, but it does come with some restrictions. Before we inspect those restrictions we must first look at the reason why the glibc was not used in the first place. This can be declared very fast and accurate. Glibc is a *slow* and *huge* library. While this is not an issue for systems that run on Linux (desktops, servers, ...) it does form a problem for small-scale mobile platforms. Another issue with glibc is that it falls under GPL (GNU General Public License) and the smaller version, uClibC falls under LGPL (Lesser GNU General Public License). This implies that everything that uses these libraries also falls under these licenses, something Google was looking to avoid. Hence the option to use a new library, Bionic, which uses as BSD license and thus can shield its applications from the GPL and LGPL licenses. Finally we must note that glibc is quite large and meant for high frequency processors, where bionic is a lot smaller and works very fast, even without high speed processors[1].

[1]High frequency CPUs are becoming available for mobile platforms at this very moment

Because the C++ restrictions of the Bionic library are of utmost importance to this IRATI project inclusion we will now take a closer look at these restrictions. The most profound and important restriction is the lack of support for C++ exceptions. Google engineers deemed these exceptions bloated and largely impractical for use thus the support for this was entirely cut. When people still want to use exceptions they are advised to try another library, add a library that does support these or switch to Java programming language in the userspace. This is an important change as the current IRATI project does use C++ exceptions and thus the code will need to be retailored to fit seamless on the Android platform. Another option is to try and implement changes on Android so that the current IRATI project can be ported over. Secondly, Bionic does not contain a C++ Standard Template Library. This is for obvious reasons to try and keep Bionic as small as possible. When applications wish to use a C++ STL they will have to include one with the application or acquire it beforehand. Other changes bionic made can be viewed at Pthreads. These are threads that are standardized and should be compatible cross-platform. Some changes bionic made to these threads include: cancellation, pthread_once(), pthread_atfork(). Here it is recommended that when these functions are used in the original code, to revise the code and work around these changes. The changes are in place again to reduce the bloated code in glibc and endorse optimized coding. The final changes that bionic implements are the lack of support for wide and locale characters and some user-account-related functions.

Finally we see that the biggest restrictions on Android come from the use of the Bionic library. This library limits some uses of C++ and requires a bypass from the original application. We note that the IRATI project is written partially in C++ and uses exceptions. This will require a workaround on either the operating system side or on the IRATI project. How this will be handled is one of the major questions in this thesis.

Chapter 4

Conclusion and placement of the study in the thesis

In the final chapter of this literature study we will draw a conclusion and show where the literature study will fit in the master thesis. This chapter will be heavily edited and may potentially be partially (or totally) removed when the literature study is incorporated into the thesis. The main reason for this is that a lot of extra chapter need to be added to this thesis. Several tests need to be conducted and code needs to be programmed, this means that the conclusion will be quite different. This also means that the literature study will be purely that and no conclusions will be drawn from only this aspect, the conclusions will be multi-faceted.

4.1 Conclusion literature study

As we can see from the initial chapter the Internet is not perfect. It was built on a proof-of-concept with a very basic set of rigid protocols. This has lead to several disadvantages that currently require heaps of work for just temporarily repairs. For other issues we just require a plain new protocol to solve this issue. A clear answer is needed and this is what we present: Recursive InterNetworking Architecture, RINA. This new architecture requires quite a bit of initial work to set up a technical project to take over all the functions of the current Internet. After this initial work it will require minimal maintenance and prove to be very scalable in the long run. The main reason for this is the recursive function in the entire architecture.

RINA is an architecture that stands on one basic principle: networking is an InterProcess Communication (IPC) and IPC alone (Day, 2008). This IPC is provides an IPC process.

A group of these coherent processes (same layer) forms a Distributed IPC Facility (DIF) when these processes are enrolled. These DIFs stack recursively until the application on host A can communicate with the application on host B. Every one of these DIFs has a unique set of functions and operates in its own scope.

This shows the way RINA functions, but since RINA is only a model we need a technical implementation. This implementation we will be using and supporting is the IRATI (Investigating RINA as Alternative to TCP/IP) project. This European project focuses on a technical implementation of RINA for UNIX-like operating systems. It already has a working Shim-DIF for Ethernet (802.1q). In this thesis we will try to port the IRATI stack to Android. Finally we will try to make a working Shim-DIF for WiFi on Android.

Finally we take a closer look to the DIF that will be researched in this thesis. The Shim-DIF for wireless. This Shim-DIF will use the Ethernet MAC header as its main pillar and build further upon this. In research from this literature study we have concluded that 802.11 (WiFi) MAC headers are not available because drivers reform these to 802.3 headers. This means that the initial research question is altered in a manner that reflects these findings. To illustrate this with images: we initially started with image 2.1 and after this study we came to the conclusion that we will have to work on a model represented in image 3.4. When all the above questions have been answered and submitted, we can assume the research question (2.1) has been solved.

4.2 Placement of the literature study in the master thesis

This literature study functions as a background study for the master thesis. In this study we have looked at the cause of the problem, clearly stated the research question, and finally we have provided adequate research to start the thesis. The background study and information has thus been provided to the researchers. This information shall be used to continue the thesis from this point on. Several research topics and points of importance have been pointed out in this study.

As a final word we should note that this literature study will most likely be further expanded quite a bit. However this final chapter will most likely be removed from the thesis as it does not actually provide an answer to the research question. The conclusion in the thesis will provide a full and thorough answer to the question.

Chapter 5

SHIM DIF for 802.11

5.1 Introduction

In this chapter we will stipulate the specifics for a functional Shim DIF over WiFi. More specifically over 802.11. Firstly we must note that this is not a fully functional DIF and is only meant to provide legacy support towards the 802.11 protocol. The purpose of this Shim DIF is to represent 802.11 as a DIF towards the DIF on top of this. Due to this we will not try to improve or change specifics of the 802.11 protocol but we will try to map them as seamlessly as possible towards the RIN Architecture. The Shim DIF will be used as an adaptor, this implies that towards layers under 802.11 they will only see the 802.11 layer, while layers looking from above will see a working DIF in function of RINA.

LLC
MAC
PHY

Table 5.1: Overview of 802.11 protocol parts

The 802.11 protocol consists of 3 main parts and the adaptor will try to span over all these. On top we have the 802.2 LLC layer, below that is the 802.11 MAC layer and finally we have the 802.11 physical layer. We must note that the LLC layer is an old protocol that has been reused for this 802.11, it will not be changed and fully used in it's current version. The MAC layer has been changed as recently as 2007 with 802.11e. It has several fields reserved for future use and could be changed down the road. This means

that when this MAC layer is changed that these changes will have to be addressed in the Shim DIF. Finally is the physical layer presented at the bottom of the 802.11 scope. We instantly note that this changed quite often but provides no use towards the Shim DIF. This physical layer will thus not be used for the Shim DIF and changes to this should not reflect in the wrap above the WiFi protocol.

WiFi is not, and will never be, a fully functional RINA DIF. This means that some limitations apply to this protocol. These are:

- Limited QoS cubes
-

5.2 Mapping of 802.11 LLC header

We must initiate this section with stating that the 802.11 LLC header, which situates itself on top of the MAC header, is actually the same as the 802.2 LLC header. For further information and documentation on this standard we refer to the IEEE document (Society, 1998).

5.2.1 DSAP and SSAP address fields

These address fields, each 8 bits large, are used to identify the application process on top of the current DIF. While the fields are 8 bit large, the first 2 bits (LSB and following bit) are reserved for use by the standard. This means that we have 6 bits for the actual address. This still leaves us with $2^6 = 64$ addresses at both source and destination.

First bit by IEEE, second bit by ISO. Mention this!

Chapter 6

Background study 802.11

6.1 Mapping of 802.11 header

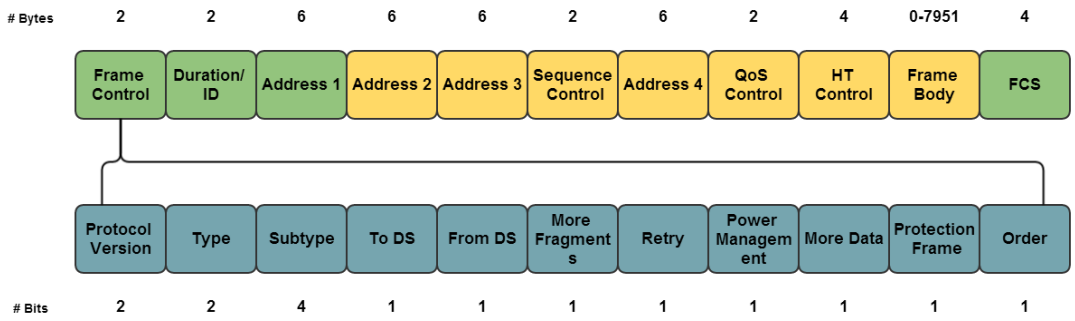


Figure 6.1: 802.11 MAC frame

6.1.1 Frame Control

In this section a closer look will be taken at the Frame Control. This part is a requirement in every 802.11 MAC frame and will be carefully analyzed.

Protocol Version

This first field consists of 2 bits and is currently always 00. The field is fully reserved for future revisions of the 802.11 header. This field will always stay 00 in the current version of the SHIM DIF.

Type & Subtype

Two fields that consist of 2 and 4 bits. The combination of these two fields determines the function of the current frame. Currently 3 different types are used as type field, these are:

- Management
- Control
- Data

Since these fields are used to set up communication between two devices these fields won't be touched on in the SHIM DIF. Further information on these fields can be found in the IEEE Std. 2012 (Society, 2012)

To DS & From DS

The next two fields are fairly self-explanatory. They decide whether a packet is traveling from or towards a Distribution System (DS). When both of these fields are set to 0 it implies that a frame is traveling straight from one STA to another STA without going to a DS first. Both fields on value 1 means that we are using this in mesh mode and the 4 addresses will be used. Since this SHIM DIF should only providing base function for 802.11 this will not be handled further.

More Fragments

This one bit field is set to 1 when either data or management frames have more fragments. In all other cases it should be 0.

Retry

A one bit field limited to data or management frames. It is set to 1 when this frame is a retransmission, in all other cases it is set to 0.

Power Management

This one bit field's value is heavily reliant on the entire frame. For this limited use SHIM DIF the use of this frame should be copied from the 802.11 standard (Society, 2012).

More Data

Another one bit field with a specific purpose that will be copied from the standard. This field provides information about following frames that are buffered at the AP for this specific STA.

Protection Frame

The second to last field in the Frame Control field is like many other fields a 1 bit field. This bit contains information about the Frame Body. When this Frame Body contains information that has been encrypted this field is set to 1. This is however only the case in specific cases and should be carefully analyzed from the standard. An example here is that this field can never be set to 1 when the Frame Body does not contain any data at all.

Order

The final bit in the Frame Control field provides two purposes. It can be set to one in either non-QoS frame or in QoS frames. In both the 1 value provides a different use for the system. Since this also has no further use for the SHIM DIF it will not be handled in detail.

6.1.2 Duration / ID

After the Frame Control field we have the second required field to make a valid 802.11 frame. This is the Duration or ID frame. The frame is 2 bytes (16 bits) large and can be used in various ways. This field is dependent on the type and subtype field earlier mentioned in the Frame Control field. It provides information for the total duration of the frame or provides an ID that can be used to identify and order the frame in it's rightful order.

6.1.3 Address 1, 2, 3 & 4

These 4 address fields all use the same type of address fields used in other Ethernet MAC frames. They consist of 6 bytes (48 bits) each and only address 1 most be filled in to complete a valid frame. The addresses can be used for following purposes:

BSSID Basis Service Set Identifier

SA Source Address

DA Destination Address

TA Transmitting STA Address

RA Receiving STA Address

These fields are to be used as the points of attachment MAC addresses of the underlying interfaces. The shim DIF is bound to these interfaces through this address.

6.1.4 Sequence Control

A 2 byte field that is split up in 2 subfields. Note that this field is not present in control frames, only in data and management ones. The first subfield is the fragment number and is 4 bits long. This subfield tells the fragment number of the frame, starting at 0 for the very first piece and incrementing by 1 step for every consequent fragment. The second subfield is 12 bits long and is the sequence number subfield. This number is the order of the frames and provides information towards the system about the order of frames.

6.1.5 QoS Control

This field provides information about QoS (Quality of Service) settings the frame currently provides. The 16 bit value is dependent on the type, subtype and the transmitting STA of the frame. QoS Control field is needed when the QoS bit in the subtype of a frame is set to 1. The field often contains information about Traffic Identifiers (TID), ACK policy, EOSP, ... A closer look in at these different fields can be taken in the 802.11 standard document (Society, 2012) (p389).

6.1.6 HT Control

The final field before the actual frame body is a 4 byte field. The field contains information about the High Throughput of the frame. It is present in Control Wrapper frames and QoS data frames. Since this has little use in RINA it will left as it is and copied exactly from the current standard (Society, 2012).

6.1.7 Frame Body

This field is between 0 and 7951 bytes long and contains the actual body of the frame. This will contain further DIFs and ultimately the actual data is should be transferred.

Notice that for some frames this can actually be 0, this implies it can be an optional field.

6.1.8 FCS

Final field of the 802.11 frame is a 4byte long Frame Control Sequence. It contains a 32 bit CRC calculated over the entire MAC frame, including the body. The use of this field is to detect errors in the entire frame. It should be copied exactly for the SHIM DIF usage.

Bibliography

Abbate, J. (2000). *Inventing the Internet (Inside Technology)*. The MIT Press.

Berners-Lee, T. (2000). *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. HarperBusiness.

Contributers (2013). The gnu c library. `https://github.com/android/platform_bionic`.

Day, J. (2008). *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall.

dictionary.com (2013). Recursive | define recursive at dictionary.com. `http://dictionary.reference.com/browse/recursive?s=t`.

Gast, M. (2005). *802.11 Wireless Networks: The Definitive Guide*. O'Reilly Media, second edition.

gnu.org (2013). The gnu c library. `http://www.gnu.org/software/libc/`.

Hafner, K. (1998). *Where Wizards Stay Up Late: The Origins Of The Internet*. Simon & Schuster.

Heart, F., Kahn, R., Ornstein, S., Crowther, W., and Walden, D. (1970). The interface message processor for the arpa computer network. *Proc. 1970 Spring Joint Computer Conference*, 36:551–567.

internetsociety.org (2014). Brief history of the internet - internet timeline | internet society. `http://www.internetsociety.org/internet/what-internet/history-internet/brief-history-internet`.

irati.eu (2013a). The recursive internetwork architecture. `http://irati.eu/the-recursive-internetwork-architecture/`.

irati.eu (2013b). The recursive internetwork architecture objectives. `http://irati.eu/goals/`.

juniper.net (2013). Junos network operating system - juniper networks. `http://www.juniper.net/us/en/products-services/nos/junos/`.

Ortiz, S. (2009). IEEE 802.11n: The Road Ahead. *Computer*, 42(7):13–15.

Paul, T. and Ogunfunmi, T. (2008). Wireless LAN Comes of Age: Understanding the IEEE 802.11n Amendment. *Circuits and Systems Magazine, IEEE*, 8(1):28–54.

Perahia, E. (2008). IEEE 802.11n Development: History, Process, and Technology. *IEEE Communications Magazine*, 46(7):48–55.

Salus, P. H. (1995). *Casting the Net: From ARPANET to INTERNET and Beyond*. Addison-Wesley Professional.

Salus, P. H. (2008). *The ARPANET Sourcebook: The Unpublished Foundations of the Internet (Computer Classics Revisited)*. Peer-to-Peer Communications Inc.

Society, I. C. (1998). Part 2: Logical Link Control. *IEEE Network*.

Society, I. C. (2012). Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Network*.

Staessens, D. (2014). Clean slate network architectures - are there new ways for networking? iMinds JRA1 workshop.

Stultz, J. (2011). Android os for servers? `http://elinux.org/images/8/89/Elc2011_stultz.pdf`.

technet.microsoft.com (2014). How 802.11 wireless works: Wireless. `http://technet.microsoft.com/en-us/library/cc757419(v=ws.10).aspx`.

Vrijders, S., Salvestrini, F., Grasa, E., Tarzan, M., Bergesio, L., Staessens, D., and Colle, D. (2014). Prototyping the Recursive InterNet Architecture: The IRATI project approach. *IEEE Network*.

Vrijders, S., Trouva, E., Day, J., Grasa, E., Staessens, D., Colle, D., Pickavet, M., and Chitkushev, L. (2013). Unreliable inter process communication in Ethernet: migrating to RINA with the shim DIF. In *5th International Workshop on Reliable Networks Design and Modeling (RNDM-2013)*, pages 97–102.

Wiki, E. L. (2013a). Android kernel features - elinux.org. `http://elinux.org/Android_Kernel_Features`.

Wiki, E. L. (2013b). Android notes - elinux.org. `http://elinux.org/Android_Notes#C_Library_.28bionic.29_info`.

wireless.kernel.org (2014). modes - linux wireless. `http://wireless.kernel.org/en/users/Documentation/modes`.

Appendices

Email conversation Linux Wireless Mailinglist

Mathieu Devos <mathieu.dvs@gmail.com>

Skb and ieee80211 headers

12 messages

Mathieu Devos <mathieu.devos@ugent.be> Wed, Jul 31, 2013 at 11:39 AM
To: linux-wireless@vger.kernel.org

Hi,

I hope this is the right place to ask for a little bit of help as I'm currently beyond stuck on a challenge I'm trying to accomplish. I'm trying to write a "simple" LKM that properly uses a ieee80211 header to print information about the mac addresses (addr1->addr4) and later down the road try to send my own data.

I only need to get L2 working, no need for TCP/IP, just a proper ieee80211 based on input from skb would be huge for me.

So my issue: when placing the ieee80211 on my mac_header after I hook my skb from my wireless device (wlan0 on android - I9100) I get a huge amount of zero's and random(?) numbers when trying to print the addresses. This leads me to the first conclusion that mac_header is placed wrong when using 80211. After that I saw a lot of people just using the skb->data pointer. Now this gives even weirder issues for me and actually totally crashes my kernel.
So I went back to starting with printing as much info as possible.
This is a sample output after I hook my packet type:

Skb->dev->name: wlan0

Skb->head: 0xe1d37040

Skb->mac_header: 0xe1d372a9

Skb->data: 0x510 (!!!)

Skb->tail: 0xe1d37460

Skb->len: 617

Skb->hdr_len: 0

When trying to just capture this and only print a certain message when one of the addresses maches my dev->dev_addr I never get any data while the phone is connected and actively browsing the internet.

I'm aware that before I throw my hook some data is being changed around already in net/core/dev.c and in net/mac80211/rx.c The weird part is that these seem to be putting on ethernet headers (skb->protocol = eth_type_trans(skb, dev); AND kb_pull_inline(skb, ETH_HLEN);
eth = eth_hdr(skb);) on items that should be ieee80211 headers.

Any insights as to why my data header is in such a weird spot (nowhere between my head and my tail) or where I should call the

ieee80211_header on? I have tried working my way back from tail with len and adding another ETH_HLEN but while I get data, it never really matches my own mac addr so I'm assuming the data is still pretty wrong.

Added links:

https://github.com/mathieudevos/kernelmodules/blob/master/ethernet_test.c (my own program)
https://github.com/mathieudevos/linux_kernel_3.2.48 (used to get all the .c files from to acquire information)

If possible I'd like to write a small guide after these issues have been fixes for people who like me would like to get started with a basic LKM in the ieee80211 part of linux.

If this is not the place to ask these questions, please disregard me (hopefully it is) but all help would be welcome.

Kind regards,
Mathieu Devos

Arend van Spriel <arend@broadcom.com> Wed, Jul 31, 2013 at 12:08 PM
To: Mathieu Devos <mathieu.devos@ugent.be>
Cc: linux-wireless@vger.kernel.org

On 07/31/2013 11:39 AM, Mathieu Devos wrote:

> Hi,
>
> I hope this is the right place to ask for a little bit of help as I'm currently beyond stuck on a challenge I'm trying to accomplish. I'm trying to write a "simple" LKM that properly uses a ieee80211 header to print information about the mac addresses (addr1->addr4) and later down the road try to send my own data.
>
> I only need to get L2 working, no need for TCP/IP, just a proper ieee80211 based on input from skb would be huge for me.
>
> So my issue: when placing the ieee80211 on my mac_header after I hook my skb from my wireless device (wlan0 on android - I9100)

Not sure what you goal is, but what wireless device is that? You may just get 802.3 packets from the device.

Gr. AvS

[Quoted text hidden]

Mathieu Devos <mathieu.devos@ugent.be> Wed, Jul 31, 2013 at 12:28 PM
To: Arend van Spriel <arend@broadcom.com>
Cc: linux-wireless@vger.kernel.org

Hi,

It's an android smartphone (I 9100 - Samsung galaxy S2) so it does not have a normal ethernet 802.3 input even. I check before selecting the device that it's wireless (through ieee80211_ptr) and this properly returns the wlan0 device which should be on the 80211 standard.

My goal is to get the ieee80211_header properly on the skb with this

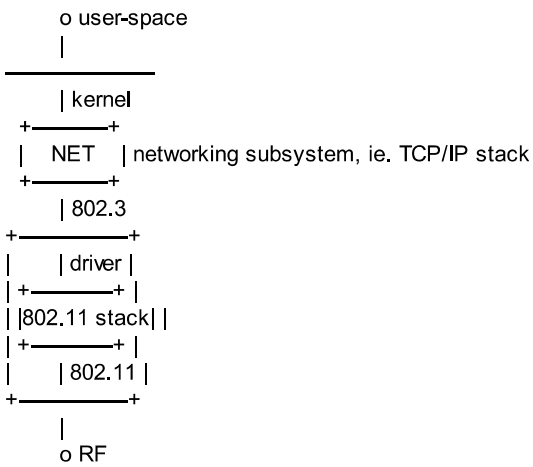
device, but some of the pointers and original data in the skb seem totally off. This leaves me clueless as to where to put this ieee80211_header.
I've tried putting it right on skb->head (wrong I know, but I was getting desperate), on skb->mac_header (also wrong, no idea why though), I went back from skb->tail with len and even added ETH_HLEN to that as well because you can see that before my hook gets activated: skb_pull_inline(skb, ETH_HLEN);
In the end I'm left with a header that is forced onto data but with a wrong origin pointer thus basically leaving me with all wrong data in the header.

Kind regards,
Mathieu Devos
[Quoted text hidden]

Arend van Spriel <arend@broadcom.com> Wed, Jul 31, 2013 at 1:05 PM
To: Mathieu Devos <mathieu.devos@ugent.be>
Cc: linux-wireless@vger.kernel.org

> On 07/31/2013 12:28 PM, Mathieu Devos wrote:
> > Hi,
> >
> > It's an android smartphone (I 9100 - Samsung galaxy S2) so it does not have a normal ethernet 802.3 input even. I check before selecting the device that it's wireless (through ieee80211_ptr) and this properly returns the wlan0 device which should be on the 80211 standard.

sigh Welcome in the world of protocol stacks, wireless, networking (choose your poison). Let me draw the picture.



The device hooks up to the networking subsystem as an ethernet device and as such it receives 802.3 packets. These are converted to 802.11 packets by the 802.11 stack. Now depending on your device that happens in the device driver or on the device itself. Another option is that this is done by mac80211 (kernel provided 802.11 stack), but that is probably not the case, but to be sure I ask again: what wireless device do you have in your galaxy S2?

Gr. AvS
[Quoted text hidden]

Mathieu Devos <mathieu.devos@ugent.be> Wed, Jul 31, 2013 at 2:39 PM
To: Arend van Spriel <arend@broadcom.com>

Cc: linux-wireless@vger.kernel.org

Hi,

The wireless chip is a Broadcast BCM4330 chip. After looking around a bit I found that this is a fullMAC and links to the driver on wireless.kernel: http://wireless.kernel.org/en/users/Drivers/brcm80211 and also links directly to android itself: https://android.googlesource.com/platform/hardware/broadcom/wlan

Still trying to learn a lot in this tightly packed world of protocol stacks, wireless and all the other poisons. Seems like I still have a long way to go. Thank you already for helping me out with these issues and taking the time to explain these things to me.

Kind regards,
Mathieu Devos
[Quoted text hidden]

Arend van Spriel <arend@broadcom.com> Wed, Jul 31, 2013 at 2:55 PM
To: Mathieu Devos <mathieu.devos@ugent.be>
Cc: linux-wireless@vger.kernel.org

> On 07/31/2013 02:39 PM, Mathieu Devos wrote:
> > Hi,
> >
> > The wireless chip is a Broadcast BCM4330 chip. After looking around a bit I found that this is a fullMAC and links to the driver on wireless.kernel: http://wireless.kernel.org/en/users/Drivers/brcm80211 and also links directly to android itself: https://android.googlesource.com/platform/hardware/broadcom/wlan

I suspected. The bcm4330 is indeed a fullmac device, which means the 802.11 stack is running on the device. The driver on Android is located under:

https://android.googlesource.com/kernel/samsung/+/android-samsung-3.0-ics-mr1/drivers/net/wireless/bcmdhd/

Not sure which android version you have.

The brcm80211 on wireless.kernel.org is the upstream linux driver.

Gr. AvS
[Quoted text hidden]

Mathieu Devos <mathieu.devos@ugent.be> Wed, Jul 31, 2013 at 3:45 PM
To: Arend van Spriel <arend@broadcom.com>
Cc: linux-wireless@vger.kernel.org

Alright,

Seems like I have fixed some issues while added some others. Since I assume that when my hook gets activated the data pointer should be at the start of the 802.3 header I casted an ethhdr (8023) on top of that and it seems that on my notebook this is handled correctly (I can actually check on my own mac addr and see that these frames are for/from me).
The issue with android: the data pointer (see previous mails) is totally off and causes a full blown kernel crash. Trying to set it manually leaves me with nothing really. Could more information about

skb->data manipulation be in those drivers and is there a reason why not everything is ended with the same data structure (thus standardizing the part where the hook comes in). My next issue is with mac_header that seems off just as much in both my notebook LKM and in my android device (when casting 8023 or 80211 headers on them they deliver totally wrong headers).
Somebody in the office told me that the 8023 header is added after the 80211 and thus to my logic when casting skb->data - sizeof(*wirelessheader) I should end at the start of the 80211 header. This however is wrong, are there more padding bytes present and if so, how much would I have to move the pointer or is do I have to use a whole other function for this.

A quick remark as to why I'm doing all this stuff: this is preparation for my master thesis where I'll be researching the possibility of RINA (instead of TCP/IP) in the wireless stack (android specifically). Since this is preparation work I'd love to end this with just having an sk_buff with correctly placed 802.3 header (works on my notebook, not on android device) and with a properly set up 802.11 header (so I can later on map functions from/to rina to 802.3/802.11).

Thanks again for the quick answers and the already provided help.

Kind regards,
Mathieu Devos
[Quoted text hidden]

Arend van Spriel <arend@broadcom.com> Wed, Jul 31, 2013 at 5:04 PM
To: Mathieu Devos <mathieu.devos@ugent.be>
Cc: linux-wireless@vger.kernel.org

> On 07/31/2013 03:45 PM, Mathieu Devos wrote:
> > Alright,
> >
> > Seems like I have fixed some issues while added some others. Since I assume that when my hook gets activated the data pointer should be at the start of the 802.3 header I casted an ethhdr (8023) on top of that and it seems that on my notebook this is handled correctly (I can actually check on my own mac addr and see that these frames are for/from me).

How does your hook work? Are you intercepting packets?

Regards,
Arend
[Quoted text hidden]

Mathieu Devos <mathieu.devos@ugent.be> Wed, Jul 31, 2013 at 5:11 PM
To: Arend van Spriel <arend@broadcom.com>
Cc: linux-wireless@vger.kernel.org

This is my hook: it gets called after I set dev to wlan0. I used a guide for this and it seems to work for my notebook where I'm able to find the 802.3 header but still no 802.11.

```
static int ptype_function(struct sk_buff *skb, struct net_device *dev,
struct packet_type *ptype, struct net_device *dev2);

static void throw_hook(struct net_device *dev){
  ptype.type = htons(ETH_P_ALL);
```

```
    ptype.func = &ptype_function;
    ptype.dev = dev;
    dev_add_pack(&ptype);
    printk(KERN_CRIT "Done setting up packet type");
}
```

All code can be found here:
android: https://github.com/mathieudevos/kernelmodules/blob/master/ethernet_test.c
notebook: https://github.com/mathieudevos/wifi_kernelmodules/blob/master/ethernet_test.c

Kind regards,
Mathieu Devos
[Quoted text hidden]

Mathieu Devos <mathieu.devos@ugent.be> Fri, Aug 2, 2013 at 9:49 AM
To: Arend van Spriel <arend@broadcom.com>
Cc: linux-wireless@vger.kernel.org

Hi,

An update is in order here I believe (and an apology because I resorted to asking my questions on IRC and not sticking to the mailing list, for that: sorry).
Let's start things off quickly with my main goal with this: the code that I'm trying to write is supposed to be an adapter for a new type of internet (RINA), this is in function of my master thesis (I'm new to Linux Kernel itself). So what exactly does this mean? I'd like to be able to hook packets (skb) in the kernel with an LKM and just use one layer (datalink layer) with the hardware layer being handled by the drivers and the hardware itself. Everything on top of that should be RINA specific code.
The specifics about the master thesis is that this all should be done on android, thus using the wireless stack. I believed that in an skbuff you had the formed 802.3 header through drivers but you still also had the original 802.11 header, this however, is now proven wrong, the 802.11 header is thrown out after the drivers are done with it thus it's impossible for me to map RINA to the 802.11 header. After resuming testing with this news I'm still left with a couple of questions however:

- Can I still use the added functionality of 802.11 as compared to 802.3 by talking directly to the device and setting it's fields? I'm thinking of using "struct wireless_device" and/or "struct wiphy" and the functions that come with it.

- Is it possible to find a general form of skb frame with correctly set headers? I have currently tested my code (https://github.com/mathieudevos/kernelmodules/blob/master/ethernet_test.c) and on one android device (galaxy S2 I9100, BCM4330 wireless chip) I had my 802.3 header by at my skb->head. On my own htc one X (htc endeavoru, wireless chip unknown still) I had my 802.3 header at my skb->mac_header and on my notebook (wireless 5100 AGN) this header was present on skb->data. Is it coincidence that these 3 are totally different and is there a function (that perhaps calls the drivers) to set these skb's in the correct prepared form? And if so, will this break when I void the layers on top of the mac layer.

While I know that a part of this still needs to be researched, for instance say it's possible to set certain fields of the device to use added 802.11 functionality I won't be setting the channel manually,

that's still left to the driver, but other added functions might be more useful for RINA.
What I'm hoping to achieve before really heading off into this new linux adventure is to be able to properly receive an SKB that has the same format on my devices and later down the road try to send set up an SKB and send that one myself.

I'd like to finish with another apology for not sticking to the mailing list, for that I'm truly sorry.

Kind regards,
Mathieu Devos
[Quoted text hidden]

Arend van Spriel <arend@broadcom.com> Wed, Aug 7, 2013 at 12:27 PM
To: Mathieu Devos <mathieu.devos@ugent.be>
Cc: linux-wireless@vger.kernel.org, "David S. Miller" <davem@davemloft.net>, Eric Dumazet <edumazet@google.com>

+ networking experts

On 08/02/2013 09:49 AM, Mathieu Devos wrote:

> Hi,
>
> An update is in order here I believe (and an apology because I resorted to asking my questions on IRC and not sticking to the mailing list, for that: sorry).
> Let's start things off quickly with my main goal with this: the code that I'm trying to write is supposed to be an adapter for a new type of internet (RINA), this is in function of my master thesis (I'm new to Linux Kernel itself). So what exactly does this mean? I'd like to be able to hook packets (skb) in the kernel with an LKM and just use one layer (datalink layer) with the hardware layer being handled by the drivers and the hardware itself. Everything on top of that should be RINA specific code.

So basically you want to be able to use RINA stack as a drop-in replacement for the TCP/IP stack keeping the netdev api unchanged so device drivers do not have to change.

> The specifics about the master thesis is that this all should be done on android, thus using the wireless stack.

I can not follow your line of thinking here. The android requirement (a stupid one if you ask me) does not restrict you to the wireless stack. There are android platforms with ethernet connectivity.

> I believed that in an skbuff you had the formed 802.3 header through drivers but you still also had the original 802.11 header, this however, is now proven wrong, the 802.11 header is thrown out after the drivers are done with it thus it's impossible for me to map RINA to the 802.11 header. After resuming testing with this news I'm still left with a couple of questions however:

An sk_buff is just a structure that makes it convenient to add or remove layer specific header (or tail) data moving from one layer to the other. An ethernet device with get an 802.3 packet, which is straightforward. A wireless device also gets an 802.3 packet so from the networking stacks' perspective there is no difference between a ethernet or a wireless device.

Looking at architecture shown in [1], I think RINA does not need to care about 802.11 either as the device

drivers have their own 802.11 stack, which takes care of converting 802.3 to 802.11 packets and vice versa (or use the kernel provided 802.11 stack, ie. mac80211 to take care of that).

> - Can I still use the added functionality of 802.11 as compared to 802.3 by talking directly to the device and setting it's fields? I'm thinking of using "struct wireless_device" and/or "struct wiphy" and the functions that come with it.

As said I do not think you should care about 802.11. The structures mentioned are data structures for configuration purposes on respectively wireless interface and wireless device level.

> - Is it possible to find a general form of skb frame with correctly set headers? I have currently tested my code (https://github.com/mathieudevos/kernelmodules/blob/master/ethernet_test.c) and on one android device (galaxy S2 I9100, BCM4330 wireless chip) I had my 802.3 header by at my skb->head. On my own htc one X (htc endeavoru, wireless chip unknown still) I had my 802.3 header at my skb->mac_header and on my notebook (wireless 5100 AGN) this header was present on skb->data. Is it coincidence that these 3 are totally different and is there a function (that perhaps calls the drivers) to set these skb's in the correct prepared form? And if so, will this break when I void the layers on top of the mac layer.

How generic do you want it to be? As said the sk_buff is a pretty dumb struct which is mostly protocol/stack independent. General advice here is to avoid dealing with the pointers directly and use skb function api.

The hook mechanism dev_add_pack() you are using in [2] is new to me, but it seems you are doing it while the packet is being processed in the network stack so the state of the sk_buff can be pretty unpredictable. Just remember you are probably not the only one handling this packet. The stack internals are something I tend to stay clear off. Maybe one of the networking experts can elaborate.

> While I know that a part of this still needs to be researched, for instance say it's possible to set certain fields of the device to use added 802.11 functionality I won't be setting the channel manually, that's still left to the driver, but other added functions might be more useful for RINA.

Examples of useful wireless specific functions? As I see it RINA is an IPC based network stack on top of some physical layer, ie. ethernet. The architectural picture in [1] even shows it can run on top of a TCP/IP stack using sockets.

> What I'm hoping to achieve before really heading off into this new linux adventure is to be able to properly receive an SKB that has the same format on my devices and later down the road try to send set up an SKB and send that one myself.

It seems to me that you should familiarize yourself with linux networking artifacts. Maybe [3] is a good read although it may be pretty outdated.

Regards,
Arend

[1] http://irati.eu/irati-first-phase-report-on-use-cases-requirements-analysis-updated-rina-specifications-and-high-level-software-architecture-available/
[2] https://github.com/mathieudevos/wifi_kernelmodules/blob/master/ethernet_test.c
[3] http://shop.oreilly.com/product/9780596002558.do
[Quoted text hidden]

Mathieu Devos <mathieu.devos@ugent.be> Wed, Aug 7, 2013 at 2:16 PM
Draft To: Arend van Spriel <arend@broadcom.com>

Cc: linux-wireless@vger.kernel.org, "David S. Miller" <davem@davemloft.net>, Eric Dumazet <edumazet@google.com>

Hi,

> So basically you want to be able to use RINA stack as a drop-in replacement
> for the TCP/IP stack keeping the netdev api unchanged so device drivers do
> not have to change.

Correct.

> I can not follow your line of thinking here. The android requirement (a
> stupid one if you ask me) does not restrict you to the wireless stack. There
> are android platforms with ethernet connectivity.

This is wrongfully stated by me, the implementation of the RINA library should be on the wireless stack and for this master thesis the testing was chosen to be done on the android kernel. Thus most likely going towards smaller devices so to see if rina is applicable on small devices with fewer resources than the average notebook. It's thus implementation first on wireless and then specifically on android and not the other way around.

[Quoted text hidden]