Alma Mater Studiorum · Università di Bologna

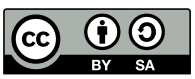
SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Design and development of a software architecture for seamless vertical handover in mobile communications

Relatore:
Chiar.mo Prof.
Vittorio Ghini

Presentata da:
Matteo Martelli

Sessione II
Anno Accademico 2015-2016



Abstract

In this work I firstly present an overview on current wireless technology and network mobility focusing on challenges and issues which arise when mobile nodes migrate among different access networks, while employing real-time communications and services. In literature many solutions propose different methods and architectures to enhance vertical handover, the process of transferring a network communication between two technologically different points of attachment. After an extensive review of such solutions this document describes my personal implementation of a fast vertical handover mechanism for Android smartphones. I also performed a reliability and performance comparison between the current Android system and my enhanced architecture which have both been tested in a scenario where vertical handover was taking place between WiFi and cellular network while the mobile node was using video streaming services. Results show the approach of my implementation to be promising, encouraging future works, some of which are suggested at the end of this dissertation together with concluding remarks.

Introduction

According to current statistics[33][27], the number of smartphone shipments have almost tripled in the last five years and figures are still increasing. It is expected that by 2017, more than one in three people globally will have a smartphone. Such devices are nowadays almost always connected to the Internet giving users the opportunity to constantly interact with the rest of the world. Also wireless technologies and broadband systems are strongly improving over time providing more reliability and higher data rates. This drives users to involve more intruiguing and more performance demanding services such as high definition voice and video calls, live video broadcasts and online gamining. While employing these kind of services and their related real-time applications, users may move geographically among different point of attachments to the Internet. Ensuring that no service interruption is perceived by users in this scenario is a complex task.

The process of moving between different wireless network is referred to *handover* (or *handoff*) and in literature many works focus on finding the best way to manage situations in which handover takes place while users employ real-time communications. After having inspected these related works I decided to develop a set of software components and tools which aim to help the vertical handover process on mobile devices.

In the first three chapters of this document we will see what mobile communications consist of and which problematics mobility architectures have to deal with. Chapters 4 and 5 will then focus on the design choices and the development process of my personal project. In chapter 6 I will show how

my project suits well in a practical scenario through experimental results. Finally, in the last chapter some possibile future works will be addressed togheter with concluding considerations.

Before facing three lengthy chapters about the state of the art on current wireless techonology and mobility related works, the following section sums up the main characteristics and goals of my personal project in order to introduce readers to the software I implemented.

Summary on my personal project

I worked on improving the handover process which takes place when Android smartphones move from a WiFi technology access network to a cellular technology access network and back while using a real-time video streaming service. Currently such smartphones use a network technology with considerable periods of service disruption during the network switch. The basic idea behind my project is to allow smartphones to simultaneously use both network technologies with the aim of improving handover reliability and performances without involving any user application modification. This idea comes from the ABPS project[53] which will be introduced in next chapters. My implementation efforts took care of enhancing an existing software module, called TED, designed to work with the linux kernel. Also I developed a set of proxy applications which interact with each others, with TED and with user applications. At the end I evaluated the overall enhanced system behaviour by conducting a series of experiments.

For more details, see chapters 4, 5 and 6.

Contents

List of Figures

Chapter 1

Overview of mobile communications

In the last years users gained easy access to both mobile terminals and high bandwidth connections. In fact, both smartphones and wireless technologies rapidly evolved offering daily use of the Internet to their users. Voice over IP (VoIP) and more generally real-time communications are increasingly used in daily communications.

Typical scenarios, current technologies and common goals and challenges in the context of network mobility will be introduced in this chapter.

1.1 Current scenarios

Modern cities are frequently covered with several public WiFi Access Points (APs) which allow citizens and tourists to freely connect to the Internet. Also, users may have personal access to private WiFi APs all over the city or other daily visited locations. Moreover, users of smartphone devices may likely have access to the Internet through cellular networks which provide packet switching subsystems. For instance, let us consider a mobile terminal user whom daily walks from its home to its workplace while making VoIP calls. Let us assume that it mainly uses cellular networks for data ex-

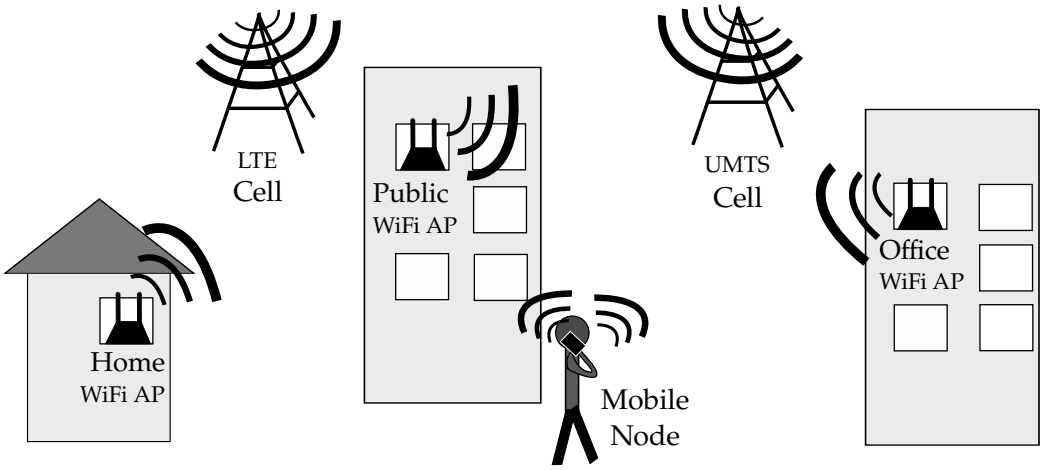


Figure 1.1: Mobility scenario

change and that it encounters several WiFi APs along the path, for example its home WiFi AP, some public WiFi APs and its office WiFi AP (figure 1.1). Since data connection plans of cellular networks are usually provided with metered data traffic, the user may prefer WiFi APs when available as they often provide access to unmetered data traffic connections. Moreover, the user may prefer WiFi over cellular network since the network coverage of the latter may be poor in some indoor rooms, thus causing considerable battery drain. On the other hand, communications over the cellular network may be preferable when WiFi connectivity starts to deteriorate, for example moving away from the AP or when the user is served by public APs in crowded places (congestion).

Switching from one type of connection technology to another may cause noticeable interruptions in the ongoing real-time conversations due to the considerable amount of time needed for the handover to take place. Handover is the process of moving between different wireless networks and it will be described in the next chapter.

1.2 Mobility

The term mobility *mobility* refers to the ability to move freely and easily. In computer networks, support for mobility refers to the ability to keep communications active during movement across different networks. More challenges arise when communications are real-time, thus including VoIP, video-conferences, online gaming, screen sharing and so on. Mobility support for real-time communications is a hot research topic since it is considered a complex task that involves many heterogeneous technologies and agents whose characteristics are constantly evolving. In fact, a wide range of solutions that aim to support mobility have been proposed in literature. They focus on different aspects and operate at different layers of the protocol stack. However, they often share the same goals, deal with similar issues and refer to a common terminology. In particular, end-node terminals which can move across networks are often called *Mobile Nodes* (*MNs*), and end-node terminals which are fixed and do not experience frequent network re-configurations are called *Correspondent Nodes* (*CNs*). Generally both entities are taken into account in order to cover two different typical situations: in the first scenario both end nodes involved in the communication are MNs while in the second more relaxed scenario one end-node is an MN and the other one is a CN. We will see in the chapter 3 how some of the existing solutions suit well for the second scenario but do not satisfy the necessary requirements when both end-points can move to a different network at the same time.

1.2.1 Current technologies

Mobile devices are able to access the Internet network through wireless points of attachment. Wireless technologies can work with short or long range radio systems. Currently, the most used short range wireless technologies are the following.

WiFi: WiFi (Wireless Fidelity) specifications are defined by the *IEEE 802.11*

standards and it allows electronic devices to connect to a wireless LAN (WLAN) network. WiFi mainly operates at 2.4GHz and 5GHz frequencies. Coverage range depends on many factors such as the specific 802.11 protocol the AP runs (a/ac/b/g/n/etc.), the transmitter power, which antennas are used, the position of the AP (indoor or outdoor) and so on. Anyway, the coverage range of typical WiFi installations can vary from around 20 meters to 150 meters[70]. Like the coverage range, also the experienced data rates can vary depending on many factors. For instance, when the perceived signal is weak, WiFi stations tend to operate at more reliable modulation schemes which ensure stable communications but decrease the data rate. The majority of WiFi devices are currently 802.11g and 802.11n compatible, offering maximum data rates of 600Mbps with the 5GHz band. Anyway, it is worth noting that at the time of writing, only few broadband internet access plans can offer such data rates.

Bluetooth: Bluetooth is a wireless technology for exchanging data using short range radio transmissions. Specifications and services are mantained by the Bluetooth Special Interest Group (SIG). Bluetooth was intended for building Wireless Personal Area Networks (WPANs) used for interconnecting personal devices in a short-range area. It is designed for low power consumption indoor work. Coverage range can vary from around 50cm to around 100 meters. Like the WiFi technology, Bluetooth operates in the ISM radio band at 2.4GHz. *Bluetooth Low Energy*[5] and *Bluetooth High Speed*[4] are two extensions which expand on the Bluetooth application use case. The first one is intended to work with devices that run for very long period of time thanks to its power efficiency, while the second lets users quickly exchange large amount of data by momentary enabling a second radio.

Generally, WiFi prevails over Bluetooth when devices must access the Internet as it enables faster connections (higher bit-rates), better range from the base station and better security. In any case, when MNs are not covered

by any WiFi AP, they rely on cellular network technologies which operate with long range radio system. Cellular networks technology can be classified by their generation number[71][47].

1G: In the first generation of cellular networks, the concept of *cell* system were introduced (Japan 1979, US 1984). A cellular network is composed by many cell sites, each covering a small area. The basic idea was to assign a different operating frequency to each cell, allowing partial overlapping.

2G: In the second generation of cellular networks (late 1980s), analog cell systems were replaced by digital, circuit-switched cell systems defined by the *GSM* standard. Voice calls started to be digitally encoded and compressed allowing more calls to be transmitted in the same amount of radio bandwidth. Other benefits introduced by digital signal were encrypted conversations and data services such as SMS text messages and emails.

2.5G: *General Packet Radio Service* (*GPRS*) is a 2G-3G transitional technology, thus often called 2.5G. GPRS was developed, and opened in 2000, to build packet-switching systems on the existing GSM networks which were circuit-switching based. The main advantages introduced by GPRS are the support for IP networking and higher data-rates, typical 56Kbps downlink and 14.4Kbps uplink, compared to GSM data rate, max 9.6Kbps both downlink and uplink. The popularity of GPRS soared thanks to its ease of deployment. In fact, to provide GPRS services on the top of GSM, it was sufficient to add few GPRS Support Nodes (GNS), which acted as gateways to the Internet network, and to do some other small changes to the existing 2G networks.

2.75G: *Enhanced Data Rates for GSM Evolution* (*EDGE*) was invented and introduced by AT& T (former Cingular) as an upgrade to the existing GPRS and 2G networks. EDGE substantial enhancements are at the physical layer which includes a new form of modulation: 8 Phase Shift

Keying (8PSK) also used in 3G networks. The new modulation method allowed data communications to be exchanged at a typical data rate of 384Kbps and the already existing GPRS infrastructure could be used.

3G: 3G is a family of standards used for mobile devices and mobile telecommunication services that comply with the *International Mobile Telecommunications-2000* (*IMT-2000*) specifications by the *International Telecommunication Union*. Telecommunications companies employing networks advertised as 3G had to fulfill the IMT-2000 requirements. Many technologies were accepted as 3G standard like CDMA2000 developed in US, WCDMA (Wideband-CDMA) developed in Europe and TD-SCDMA developed in China. WCDMA and TD-SCDMA are both access methods used in the *Universal Mobile Telecommunication System* (*UMTS*) technology officially launched in 2002. WCDMA uses a code division multiple access method (as the UMTS competitor CMDA2000) while TD-SCDMA combines both code division and time division multiple access methods.

Data rates in 3G networks can reach around 2Mbps downlink and 384Kbps uplink with UMTS. Later enhancements of UMTS are *High Speed Downlink Packet Access* (*HSDPA*) and *High Speed Uplink Packet Access* (*HSUPA*) which belong to the *High Speed Packet Access* (*HSPA*) family, sometimes also called 3.5G. A later version of HSPA called HSPA+ enables data rates up to 42Mbps introducing MIMO (Multiple-Input, Multiple Output) technology and higher order modulation (64QAM) techniques.

4G: The fourth generation of cellular networks is intended to be an all-IP based solution. The purpose is to integrate different radio access networks together relying on the Internet network as the backbone. The IMT-Advanced is a set of requirements for 4G standards and defines 100Mbps and 1Gbps as the peak speeds for high mobility communications (e.g. trains, cars) and low mobility communications (e.g. pedes-

trians) respectively. Even if LTE and WiMax technologies do not fulfill the IMT-Advanced requirements, the *ITU Radiocommunication Sector* (*ITU-R*) agreed that the term "4G" can be applied to these technologies since they provide a substantial improvement with respect of the 3G technologies. ITU indicated "LTE-Advanced" and "WirelessMAN-Advanced" as the official designation of IMT-Advanced[22]. In fact, 4G with LTE-Advanded can reach data peak speeds of 1Gbps downlink and 500Mbps uplink, which is considerably higher than the respective 100Mbps downlink and 50Mpbs uplink of LTE and WiMax. 4G requirements also mandate smooth handoff across heterogeneous networks, global roaming across multiple networks and spectral efficient system. *Orthogonal Frequency-Division Multiple Access* (*OFDMA*) was chosen as the multiple access method to the wireless medium.

5G: At the time of writing, there is no standard for 5G deployment. The Next Generation Mobile Networks Alliance defines some requirements the fifth generation of cellular networks should fulfill[75]. They focus on increasing the capacity support of the mobile networks, allowing high data rate communications for more users per unit, reducing latency, enhancing the spectral efficiency and supporting the Internet Of Things and massive wireless sensor networks requirements. The Next Generation Mobile Networks Alliance's work tries to cover the expected scenarios and challenges of 2020, covering measures that may support the expected dramatic increase of the data volume by that date.

Since in some areas, cells and cellular network subsystems do not still provide support for new technologies, smartphone users can experience different data rates and throughputs in their communications. A recent report[32], shows the comparison between 3G and 4G worldwide coverage. While many countries are improving their 4G coverage, many areas around the world lack of 4G coverage. A quote from the report well summarizes the current situation of 3G and 4G network coverage in Europe:

> *Europe in particular is still leaning heavily on its extensive 3G infrastructure. In Germany, Italy, France and the U.K., the chances a 4G subscriber will connect to an LTE network are little better than a coin flip.*

Furthermore, many carriers around the world decided to turn off their GSM networks in 2017 in order to free up more bandwidth for faster 3G and 4G networks. However since many older devices and legacy services are still using 2G networks, some network operators in Europe will postpone the GSM turnoff date[24].

1.2.2 Goals and issues

Research in network mobility focuses on the main task of guaranteeing continuative network connectivity to mobile devices as they move geographically. This is not an easy task since we live in a world of heterogeneous wireless networks which all merge into the global Internet infrastructure, built and designed in the era of wired networks. Things become more difficult if voice and video real-time applications are involved, since they often require high data rates and low latencies. It is true that wireless technologies and broadband systems are strongly improving but at the same time users are expecting more reliability, better service quality from mobile communications and support for new use cases such as high definition video broadcasts and calls, augmented/virtual reality online games[30], real-time data streaming from sensor networks and so on.

In the next chapters we will see many solutions that cover network mobility. After that, an implementation of an *early-packet-loss-detection* method will be described in this document as my personal contribution to the *Always Best Packet Switching* (*ABSP*) project, which aims to improve the handover process and the continuative network connectivity in VoIP communications.

Chapter 2

Seamless vertical handover: state of the art

A mobile node can connect to the Internet through connection points of the wireless access networks (e.g. WiFi, LTE, etc.), often called *points of attachment*. *Handover* (or handoff) is the process of transferring a network communication from one point of attachment to another. A handover process is called *seamless* if the MN does not perceive any interruption while moving between two points of attachment. There are two different types of handover: *horinzontal*, when the MN moves between access points of the same wireless technology (e.g. between two UTMS cells), and *vertical*, when the MN moves between access points of different wireless technology (e.g. switching between WiFi to cellular network and viceversa). The figure 2.2 outlines the two different types of the handover process.

MNs can be equipped with multiple *Network Interface Cards* (*NICs*) since they can connect to different wireless technologies access points. In fact, today's common MNs are often equipped with a WiFi NIC and a cellular network NIC. Thus, this document focuses on the vertical handover process which takes place when MNs move between WiFi networks and cellular networks and back. In this chapter we will cover various approaches and metrics that may be considered for the handover decision.

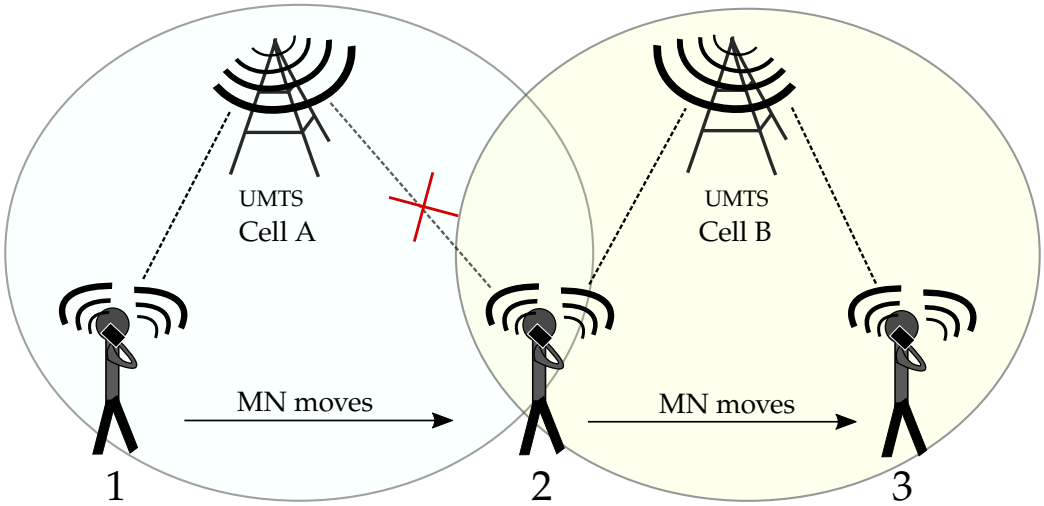


Figure 2.1: Horizontal handover: a MN changes access network using the same NIC, while moving.

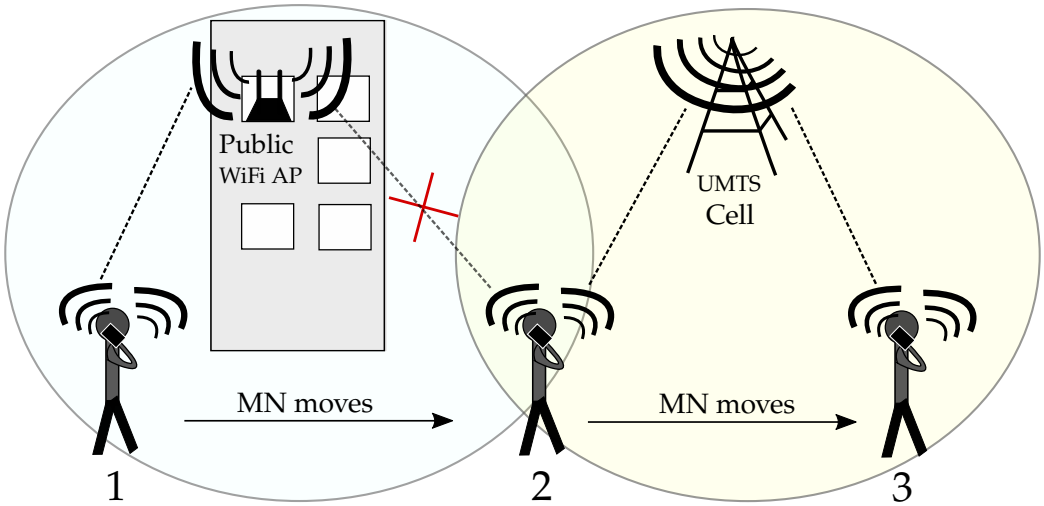


Figure 2.2: Vertical handover: a MN changes access network and the NIC used, while moving.

2.1 Handover criteria

The decision of whether to perform a handover is taken according to status information constantly gathered by the MNs about current and the neighbor access networks. In literature, some works[51][88][63] outline the metrics that can be used for the handover decisions. The most frequently used metrics or most considered in research are:

RSSI: The Received Signal Strength Indicator (RSSI) is a measurement of the power level being received by a radio interface. Thus the higher the RSSI, the stronger the signal. Currently, Signal Strength is one of the most used metrics for handover decision as it directly relates to network coverage. It is calculated at the physical layer of the NIC and continuously updated by the MN while moving in order to determine if network coverage is still available.

Bandwidth: Bandwidth is directly related with QoS. Sometimes the signal strength metric is not sufficient to determine which network can offer a better service quality, for instance when two networks of different access radio technologies overlap. In this case it may be convenient to choose the access network which can provide the higher bandwidth. The maximum bandwidth can be easily estimated from the type of radio technology if the modulation scheme is known. However, the actual bandwidth may be far lower than the estimated one. Thus, the actual bandwidth calculation may require some practical tests which are often time consuming and can slow down the handover process. It is worth mentioning that bandwidth per user may also vary depending on the network load.

Frame Retransmission: Another metric that can be used for the handover decision is the number of frame retransmissions. Like the RSSI, the number of frame retransmissions can be used to estimate the reduction of signal strength: the higher the retransmission number, the weaker

the signal. Moreover, a high number of frame retransmissions may indicate the presence of radio interference. In this case the RSSI may not be affected and thus may fail to indicate low link quality[62].

Battery Power: One of the handover metrics that can be considered is the battery power. Sometimes it may be preferable to handover to the more power efficient network.

Traffic limit: At the time of writing, Internet access through cellular networks is often limited in the amount of traffic, for instance a user can navigate at full speed only up to N GigaBytes of traffic per month. Hence this may be considered in the handover decision but obviously it is directly dependent on the user's mobile plans and user preferences.

2.2 MN-controlled Vertical Handover

The handover process between cellular network cells or between WiFi APs is controlled or partially assisted by the network subsystems. Differently, the handover process between a WiFi network and a cellular network must be entirely controlled by the MN since there is no external link-layer entity that can interconnect the two networks.

In this section we will analyze some mechanisms and software tools developed for handling the vertical handover process between WiFi and cellular networks.

2.2.1 Media Independent Handover

The IEEE standard 802.21[84] defines a *media-independent handover* (*MIH*) framework that includes a set of tools to exchange information, events and commands between heterogeneous link-layer entities in order to facilitate the vertical handover process. It requires the implementation of an additional layer of the protocol stack, between the data-link-layer and the network layer,

on both mobile terminals and network entities (such as APs) in order to provide a standard interface of interaction. ODTONE is an implementation of the 802.21 standard, developed as an open source OS independent MIH framework. The ODTONE interface that communicates with the link-layer devices is called Link SAP whose current implementation offers only two types of events that can be used as handover metrics: link down and link up events[14]. Anyway, the Link SAP interface can be enanched with future extensions[17].

2.2.2 Transmission Error Detector

Transmission Error Detector (*TED*) is a component of the *Always Best Packet Switching* (*ABPS*) [53] architecture and it is essentially a software tool able to provide the MN with 802.11 data-link-layer information about frame retransmissions and successful (unsuccessful) frame receptions at the AP. TED can then deliver this information to the software modules at the application layer providing metrics for the handover decision. Currently, TED supports WiFi only but it could be extended to other technologies in the future.

The ABPS architecture, that will be described in the next chapter, and the TED module are the bases for my personal project which will be explained in details in the chapter 4.

2.2.3 Enabling/Disabling NICs

In order to save battery power, smartphone users often tend to disable the NIC that is known to be useless in certain situations or for a certain period of time: for instance, users may turn off their WiFi NIC while walking away from their home WiFi AP and they are sure there will be no accessible WiFi APs along the path they're going to traverse.

The automatic vertical handover operation requires all NICs to be active in order to perform scans looking for suitable points of attachment. In [50],

the authors of ABPS proposed a solution called *"Oracle"* that automatically understands when to activate or deactivate NICs according to geo-located information about WiFi APs. Essentially, the geographic positions of user accessible WiFi APs are stored in a local database on the MN during an initial mapping phase. Later in the daily use of the mobile device the database is frequently queried in order to deactivate the WiFi NIC when not needed, for example when, according to the database, there is no suitable WiFi AP within a suitable distance. Moreover, the Oracle can decide to deactivate the cellular network NIC when the MN is associated to a WiFi AP and to re-activate it when the WiFi communication starts to deteriorate (using RSSI or other QoS metrics). A recent implementation of the Oracle for Android devices has been developed by Luca Milioli in his master's thesis work[55]. He has also introduced a functionality to geo-localize the WiFi APs according to the identifiers of cellular network cells. Even if less accurate than GPS geo-localization, this method is less battery consuming.

It is clear that the Oracle does not perform vertical handover by itself, but it is an interesting tool able to optimize power consumption in multi-homed mobile devices that employ vertical handover features.

Chapter 3

Seamless host mobility: state of the art

In the previous chapter, we have seen some of the fundamental aspects of several vertical handover mechanisms. Those mechanism mostly give the end user devices the capability to decide whenever it might be reasonable to move from one layer 2 access technology to another. However, other elements must be taken into account while dealing with host mobility, such as *communication continuity* and *reachability*. In fact, after a Mobile Node (MN) performs a handover, it should remain reachable from its Correspondent Node (CN) and the previously initiated communication should not be interrupted.

A recent survey[51] exhaustively analyses the main architectural solutions for mobility support in wireless networks. In the following sections some of them will be covered, leaving out those which are conceptually similar and share the same approach with the most noted ones.

3.1 Solutions at the network layer

3.1.1 Mobile IP

Mobile IPv4 (or *MIPv4*)[77] and *Mobile IPv6* (or *MIPv6*)[78] are two IETF standard network protocols that were introduced in order to address

the need of MNs to be reached while moving between different IP sub-networks. In fact, the IP protocol assumes that a node's point of attachment to the Internet is uniquely identified by its IP address. Thus, when a node accesses the Internet through a different sub-network, it changes its IP address as well, and packets destined to its old IP address would be dropped.

The key idea behind Mobile IP is that each MN is always identified by two IP addresses, a *home address* and a *care-of address*. When a MN is at home, it can be directly reached through its home address. Otherwise when it is situated away from its home, IP packets addressed to its home address are routed to its care-of address, which correspond to the node's current location.

The IPv4 version of the Mobile IP protocol introduces new entities called agents:

Home Agent: a router that tunnels and forwards IP packets to the MN when it is away from home. It is located on a MN's home network and maintains an up-to-date node's care-of address.

Foreign Agent: a router which is located on the MN's current network, when this is different from its home network. The Foreign Agent is responsible of de-tunneling and delivering to the MN packets tunneled by the Home Agent.

Figure 3.1 shows the communication paths between the CN and the MN. As introduced before, we notice that all the packets transmitted by the CN are directed to the Home Agent which tunnels and forwards all traffic to the current Foreign Agent. It is worth noting how the traffic originated by the MN is directly routed to the CN without involving the Home Agent. This routing scheme is often called *triangular routing*, since the backward routing path is different from the original routing path.

In [60] and [57], the authors cover some of the issues that MIPv4 leads to. One of the most discussed is the ingress filtering: a protection filter mechanism against IP address spoofing attacks widely used in routers[39][49].

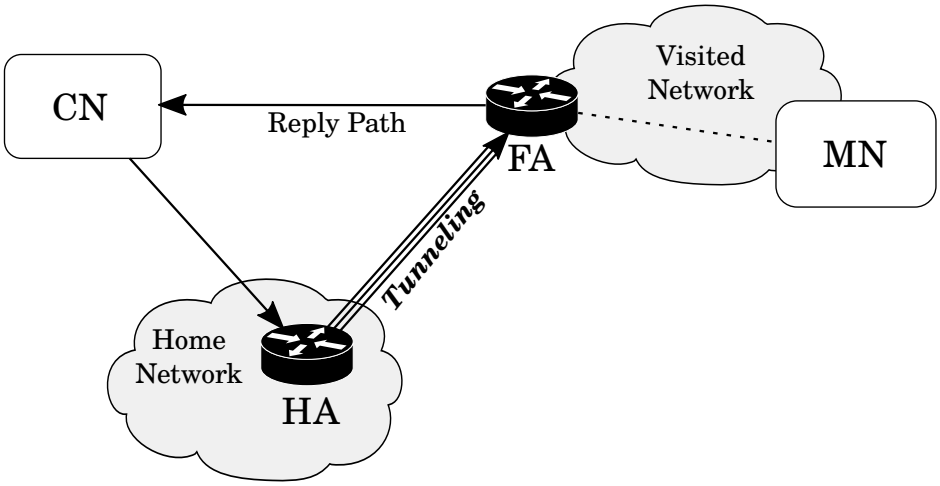


Figure 3.1: Mobile IPv4 Triangular Routing.

To overcome this problem, *reverse tunnelling*[72] was introduced in MIPv4. Its main downside resides is its inefficiency. In fact, as the authors of [69] state, *"reverse tunneling causes lower mobile connection throughput and higher roundtrip times"*.

MIPv6 has several benefits over MIPv4. This is mostly given by the inner features of IPv6, such as *neighbour discovery* and *address auto-reconfiguration*[46]. Furthermore, the Foreign Agent is no longer required by the fact that an IPv6 MN obtains a new unique IPv6 address when it moves to any different access network. Obviously, MIPv6 only works on infrastructures with IPv6 capabilities.

Monami6[85], a later extension of MIPv6, allows MNs to register multiple care-of addresses to their Home Agents. With this approach, a MN can configure an IPv6 global address for each of its NIC. In fact, Monami6 introduces the multihoming capability in MIPv6. Besides that, all the Mobile IP approaches do not overcome the potential presence of symmetric firewall systems.

In the next sections we will cover more solutions and protocols concerning host mobility.

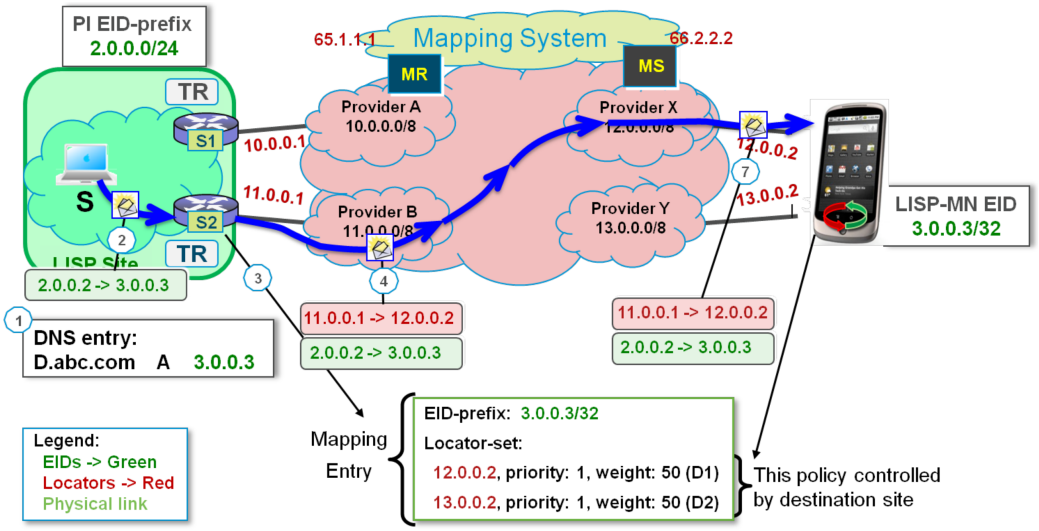


Figure 3.2: LISP data-packets encapsulation and decapsulation[42].

3.1.2 LISP

Location/ID Separation Protocol (*LISP*)[48] is another network layer tunneling solution. LISP considers two type of IP addresses: *Endpoint Identifiers* (*EIDs*), which identify hosts, and *Routing Locators* (*RLOCs*), which identify network attachment points.

Essentially, when a host A wants to communicate with a second host B, it transmits its data to the EID IP address of the second host. When data-packets originated from A reach the first LISP-enabled border router, the Mapping System finds the RLOC identifier which corresponds to the EID of B. Packets are then encapsulated and sent to the exit LISP-enabled border router, identified by the found RLOC, which the host B is attached to. This latter router eventually decapsulates and forwards the packets originated by host A to host B. Figure 3.2 shows these LISP tunnelling operations.

When a host changes its point of attachment, its EID will remain unchanged but it will obtain a new RLOC identifier. Also, the Mapping System will take care of updating the EID-to-RLOC binding. Leaving aside the details of the Mapping System entities and operations, it is worth noting that LISP introduces some form of NAT traversal[64]. However it is clear that

the main drawback of this architecture is the requirement of LISP-enabled border routers.

3.2 Solutions between the network and the transport layer

Location Independent Addressing for IPv6 (*LIN6*)[66] and *Shim6*[74] are two of the existing solutions that insert an intermediate layer between the network and transport layers of the protocol stack.

The main downside of this type of architectures is the requirement to modify the protocol stacks of both the MNs and the CNs. While it may be reasonable for the MNs to support specific mobility implementations of the protocol stack, the CNs may not be interested in such mobility support.

3.2.1 LIN6

The LIN6 architecture is the IPv6 compatible implementation of *LINA*, which stands for Location Independent Network Architecture. Similarly to LISP, LINA's basic idea is to introduce two concepts that are *node identifier* and *interface locator*. Instead of splitting the IP addresses and tunnelling data, the authors of LINA and LIN6 propose to split the network layer of the current protocol stacks in two sublayers: an *identification sublayer* and a *delivery sublayer*. Moreover, LINA uses *Mapping Agents* at the identification sublayer to deal with the resolution of the interface locator which correspond to an actual node indentifier. Mapping Agents can be located externally to the nodes' networks and their addresses must be obtained through DNS lookups before being cached.

3.2.2 Shim6

Shim6's approach is based upon adding an additional layer between the network and the transport layers. Shim6 proposes the use of network layer

IPv6 addresses as locators and *Upper Layer Identifiers* (*ULID*) for nodes identification.

Shim6 defines a mechanism of failure detection used to detect outages. In case of outage, Shim6 uses the Reachability Protocol (REAP)[34] to determine and update valid locator pairs. Moreover, unlike the previously described solutions, the approach of Shim6 for locator updates is related to timer expiration and not to movement detection. For this reason it has been considered not suitable for highly dynamic environments[51].

3.3 Solutions at the transport layer

Many solutions that work at the transport layer consider the end nodes as proactive location registry. In fact, in many protocols such as *Datagram Congestion Control Protocol* (*DCCP*)[65], *Mobile Stream Control Transport Protocol* (*m-SCTP*) [86] and *TCP enhancement TCP-migrate*[82], each end-system directly informs the CN when its IP address changes. This approach fails when both the end nodes involved in a communication are mobile and try to simultaneously perform a handover. It is obvious that both the MNs may become mutually unreachable since there is no third agent involved in the communication.

To overcome the mutual unreachability issue, *MSOCKS*[68] proposes the use of an external proxy that splits the TCP connection between two endpoints. With this scheme, the external proxy can migrate a connection when the MN changes its IP address. In addition, the proxy relies on a technique called *TCP Splice* that, as stated by the MSOCKS authors, *"gives split connection proxy systems the same end-to-end semantics as normal TCP"*. Essentially the goal of a TCP Splice is to let the end nodes believe they are directly connected by a single TCP connection. To achieve this goal, the TCP Splice technique is implemented by altering the headers of TCP packets, including TCP ACKs, received from one connection to make them appear to belong to the second connection. It is worth noting that this ap-

proach is not compatible with IPSec since the content, or even the headers, of IP packets, may be encrypted and alterations of IP and TCP headers may not be feasible. The authors of [43] point at the existence of this downside for a similar approach used by Performance Enhancing Proxies (PEPs)[41].

3.4 Solutions at the session layer

A key-role in establishing sessions between end nodes might be played by signalling protocols. These protocols deal with the initialization and control of communication sessions and multimedia streams. Since a signalling method is fundamental for establishing multimedia communications between end nodes, we review in this section the basic properties and functionalities of two common and widely used signalling protocols: the *Session Initiation Protocol* (*SIP*)[79] and *Jingle*[67], a signaling method extension for the *Extensible Messaging and Presence Protocol* (*XMPP*)[80]. Let us notice that after a communication session is initiated, the multimedia streams exchanged between the end nodes often rely on the *Real-Time Transport Protocol* (*RTP*) and the *RTP Control Protocol* (*RTCP*). They run over UDP and are used to transfer and control real-time traffic. Sometimes multimedia streams can be directly exchanged with UDP or TCP.

3.4.1 SIP

SIP works at the session layer of the protocol stack and relies on SIP public servers for end nodes discovery. Each end-point user has a unique SIP identifier. SIP users, before communication initiation, must register to a SIP registrar server. The SIP registrar server sends back their contact list to end-users where each contact corresponds to a hostname (or IP address). When a user A wants to initiate a SIP-communication with B, it sends an *INVITE* message directly to B. If some communication parameter changes (such as IP address), a *re-INVITE* message can be used.

Another interesting aspect is that SIP allows the presence of SIP proxies

that play the role of routing requests between end nodes. This gives the opportunity to user A to send an INVITE message to user B through a proxy server which will take care of finding the next SIP hop (the client B or another proxy). With this approach end nodes do not need to deal with configuration changes of their counterparts.

However the SIP architecture introduces an additional delay due to the message/response behaviour. In particular when an MN changes its IP address, it interrupts the communication, sends a re-INVITE message to the CN and resumes the transmission only after receiving a response from the CN. Clearly this behaviour does neither satisfy mobility efficiency goals nor seamless handover requirements. To optimize handover management of SIP-based mobile communications, the authors of [36] propose a technique for session continuity that exploits a SIP-based mechanism able to establish new SIP-connections without interrupting the multimedia flows of the old connections.

3.4.2 Jingle

Jingle is an XMPP protocol extension for initiating and managing multimedia communication between XMPP entities[67]. It was originally developed by Google and implemented in the *Google Talk* service[58]. In 2013 Google replaced Google Talk with *Hangouts* which does not support XMPP[25] at all. Jingle however is still used in many VoIP and videoconferencing applications[59][87][10].

XMPP allows the exchange of XML structured data over a network between any two (or more) entities[80]. It is implemented using a client-server based architecture. A client needs to contact a server in order to exchange XML data with other clients. Similarly to SIP proxies, two or more XMPP servers can connect to each other to enable inter-domain or inter-server communications. XMPP natively uses TCP transport for its communications. Essentially, clients open long-lived TCP connections with the servers. In this way long sessions of XML streams are allowed.

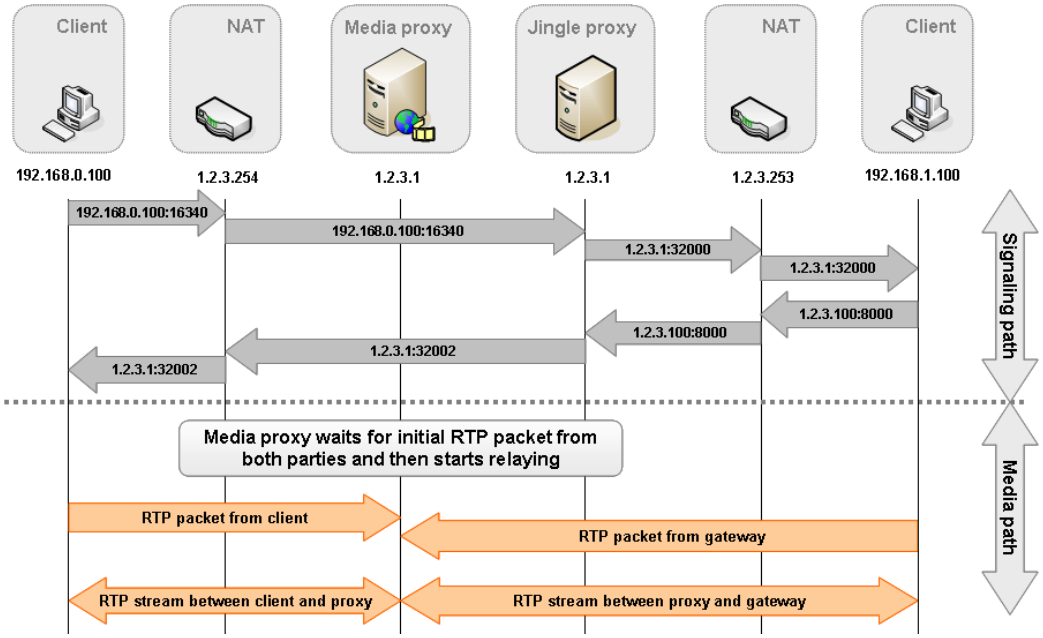


Figure 3.3: Jingle signaling and media relaying[20].

Moreover, since most restrictive firewalls may block outgoing connections on XMPP ports, the XMPP community developed an HTTP transport mode for XMPP communication. This because HTTP and HTTPS ports are often non blocked by firewalls. Protocols such as *BOSH*[76] or *Websockets*[83] are used for keeping alive long TCP connections and exchanging bidirectional XML data streams over HTTP requests/responses.

Jingle uses XMPP messages to set up, manage, and tear down multimedia sessions. Sessions can use TCP, UDP, RTP, or even in-band XMPP itself as transport methods. Like SIP, once the session is established, the media is exchanged directly peer-to-peer or through a media relay (*Jingle Relay Nodes*[44]).

Figure 3.3 shows a multimedia communication between two peers initiated with Jingle and then relayed through a Jingle Relay Node. It is important to note that the two proxies could be running on the same server machine.

The stream management XMPP extension[61] introduces the *Resumption*

operation which, similarly to SIP re-INVITE events, allows clients to quickly resume former streams rather than re-establishing them after a network outage or a vertical handover.

3.4.3 Non standard signaling

Nowadays, many VoIP and videoconferencing services, such as *Skype*[35], *Google Hangouts*[28] and the recent *Google Duo*[29] for instance, do not rely on standard signaling protocols. Even WebRTC[38], which is a collection of communications protocols and APIs for developing applications with Real-Time Communications (RTC) capabilities, does not mandate a signaling protocol leaving the choice to the applications. This consideration wants to point out that while implementations of standard signaling protocols exist, many services and applications still use non-standard ad-hoc signaling solutions.

3.5 NAT and Firewall issues

As introduced before when both MN and CN are behind symmetric NAT/firewall systems, many additional complications should be considered. In these cases, end nodes usually cannot accept incoming traffic if it is not related to any previously outgoing traffic. For instance, if an end-node behind a restrictive firewall tries to listen to incoming traffic on a certain port, external packets directed to that port would be likely dropped. Otherwise, when an end-node first transmits some data to a certain server, its Operating System will create a socket listening on a randomly chosen source port which the server could respond to. As a further complication, hosts' IP addresses are often masqueraded by NATs installed on border routers that act as gateways between home networks and the Internet. In fact, end-users behind NATs cannot expose their hosts' IP addresses to be directly reached by external hosts. In this case, an external relay must be employed, which can receive from and forward to both end-users. Also, both end-users must contact the relay before initiating the communication (figure 3.4).

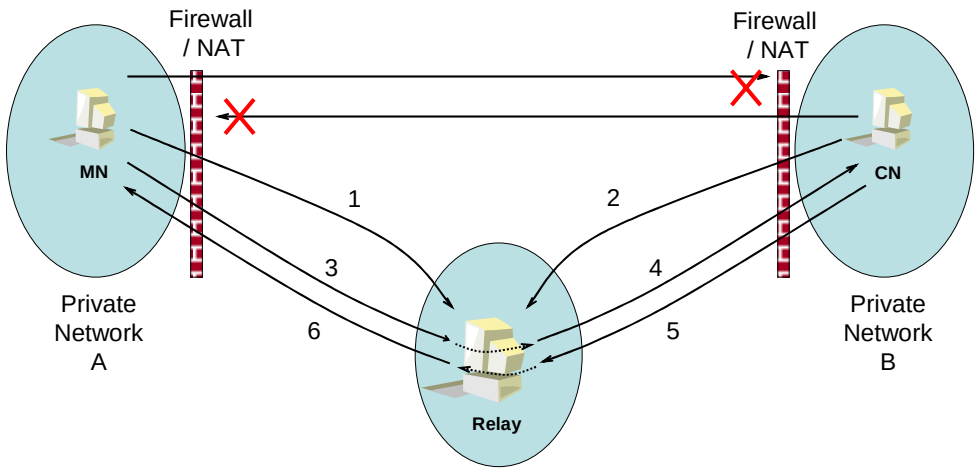


Figure 3.4: Employment of a data relay to cope NATs and firewalls[53].

As already pointed out before, the architectures and protocols we have covered in the previous section do not take into account the presence of NAT and firewall systems.

3.6 External relay solutions

Solutions that employ external relays overcome NAT and firewall systems and do not require modifications to the current network infrastructure. Some modifications are required to MNs and end-to-end communications are split in two paths: one from the first end-node to the relay and one from the relay to the second end-node. External relays are called *visible* when end-system applications are aware of them, or *invisible* when they are hidden to the applications.

3.6.1 ABPS

Always Best Packet Switching (ABPS)[53] is a distributed architecture which provides a better host mobility approach to cope with the issues described in the previous sections. It is essentially a session layer visible relay/proxy based solution. Also, it deals with the MN vertical handover capa-

bilities monitoring all the concurrent NICs available on the MN and reconfiguring the system according to the current status of the NICs. Moreover the SIP-RTP communications originated from the high level applications, are transparently transmitted through the most appropriate NIC or simultaneously through multiple NICs.

Specifically at the MN two components are introduced by the ABPS architecture: a network interfaces manager called Oracle (sec. 2.2.3), which enables or disables the NICs according to various parameters, and a client proxy that can decide which NIC should be used to transmit the application packets. This decision is taken according to the information provided by the TED component (section 2.2.2). A simple "WiFi first" policy is adopted by the client proxy: if the WiFi NIC is available and the WiFi transmission is is in a good status (high rate of frames successfully delivered to the AP), the client proxy forwards the application packets through the WiFi NIC only. Otherwise, when the WiFi transmission starts to deteriorate (high rate of frames not successfully delivered to the AP or high rate of frame retransmissions), the client proxy also starts forwarding the application packets through the cellular network NIC.

An important characteristic is that the client proxy receives TED notifications per each frame sent through the WiFi NIC. This mechanism is called *early-packet-loss-detection* and means that the client proxy is aware of the delivery status and number of retransmissions for each sent WiFi frame. Since the client proxy can decide which NICs a single datagram should be sent through, the architecture is called "Always Best Packet Switching". The name also wants to differentiate the architecture from the "Always Best Connected" (ABC) type of services, in which usually only one NIC at time can be used. Thus, the ABPS model tries to reduce the handover timing overhead anticipating the use of a second (or more) NIC(s) before the communication breaks or degrades too much. Also, a second or more NICs can be used in parallel with the first NIC when the handover is not going to take place, because for instance the communication deteriorates only for a short period

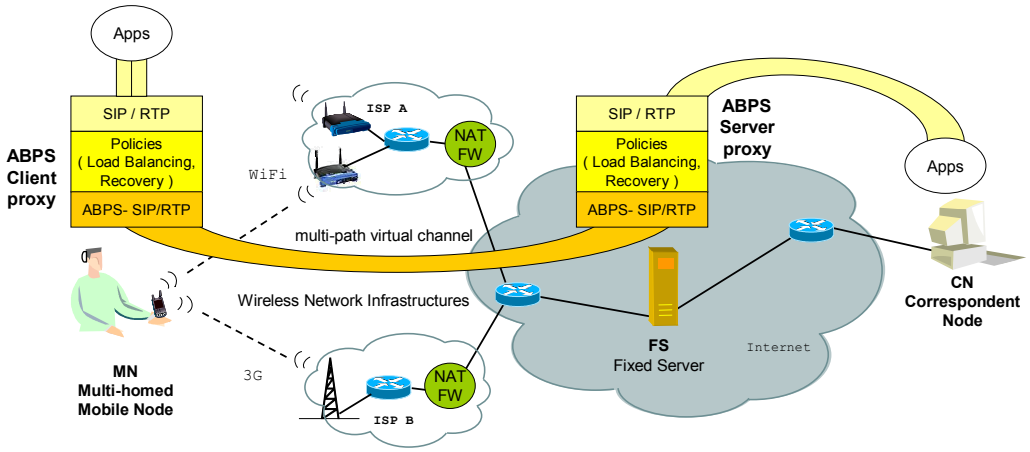


Figure 3.5: The ABPS architecture[53].

of time, but in any case redundant transmissions would lead to a better communication quality, reinforcing reliability.

Let us remember that, whenever it is "convenient", the Oracle may choose to turn off one or more interfaces in order to save battery power. For instance, it may be convenient to turn off the WiFi NIC when it is not associated to any AP. However, if it is known that in the proximity of the MN there is an accessible AP, it may not be convenient to turn off the WiFi NIC.

The ABPS architecture also relies on SIP-compliant visible proxy servers which store the source IP addresses of the MN network interfaces in order to allow communication continuity between the MNs and their CNs. Moreover an ABPS proxy server can act as a relay that lets an MN and its CN communicate even if they are both behind symmetric NATs and firewalls.

Figure 3.5 shows the ABPS architecture in a typical use case scenario: the MN is equipped with multiple NICs and wants to communicate with the CN using a SIP-RTP VoIP application. The application sends data packets to the Client Proxy that elects a NIC to forward traffic to the Fixed Server. SIP-RTP data packets are eventually relayed to the CN by the Fixed Server. Also, a backward path is used to let the CN transmit its own SIP-RTP stream to the MN.

An obvious limit of ABPS is that, at the time of writing, it is strictly

dedicated to SIP-RTP based applications and cannot be exploited for other applications. Extending ABPS to other protocols is a current goal of its authors.

3.6.2 UPMT

Many applications are not designed to work with proxies. To overcome this limitation, invisible relay solutions transparently intercept network traffic generated by the application running on a MN and redirect it to a local proxy. The local proxy then forwards the application traffic to a server proxy/relay. In fact, like ABPS, invisible relay services also rely on two proxy entities but applications are unaware of their existence. One interesting invisible relay solution is *Universal Per-application Mobility management using Tunnels* (*UPMT*)[40] which is based on a modification of the MNs' linux kernel. In particular, the UPMT modification alters the *netfilter*[73] subsystem of the Linux kernel. In brief, it allows the exposure of a virtual NIC to the user space and tunnels all the virtual NIC outgoing traffic adding an UDP+IP encapsulation layer. The tunneled traffic is then forwarded to the external relay through one of the currently available physical NICs. This allows support to applications that are not suitable for the explicit use of external proxies. Despite its interesting approach and its open source implementation, UPMT respects the ABC model and does not implement a per packet loss detection.

3.6.3 FRHP

The term *Fast Reactive Hidden Proxy* (or *FRHP*) has been introduced in [51] in order to refer to a class of solutions which combine both invisible proxy and the feature of early-packet-loss-detection. At the time of writing no implementation of such class of solutions exists. In any case this approach has been utilized in *vehicular networks*[56] and might form the basis for an interesting extension of existing external relay solutions such as ABPS and

UPMT. The first one would benefit from the inclusion of non-proxy-suitable applications while the second one would take advantage of the early-packet-loss-detection technique, with a likely improvement in terms of handover latency.

Chapter 4

Project goals and design

The previous chapters summed up the state of the art of vertical handover and host mobility. From now on, this document focuses on the design choices and the development process of my personal project.

4.1 Project goals

The project presented in this document aims to help the vertical handover process on mobile devices. In particular, I wanted to consider a today's practical scenario focusing on a moving smartphone user's vertical handover. According to current statistics[31], Android is the most popular Operating System for smartphones. It is based on the Linux kernel and some of its system characteristics are similar to the common GNU/Linux OS distributions. Furthermore my project is based on some components of the ABPS architecture whose previous implementations were developed for Linux systems. Hence the main goal of my project is to provide a fast vertical handover functionality to today's Android multi-homed smartphones equipped with one WiFi NIC and one cellular network NIC in order to improve mobile real-time and VoIP communications. An enhanced TED component (sec. 2.2.2) has been used and ported to the Android kernel to let it rely on the "early-packet-loss-detection" mechanism. Moreover early implementations of

the ABPS client proxy (called client proxy or `tedproxy`), the ABPS relay (called relay) and some dummy CN software tools have been developed in order to present a working demo application and inspire future development. Such demo application shows fast vertical handover functionality during the use of a camera streamer application which runs on an Android smartphone and streams its captured video to a remote CN while the user is moving.

We will see in the next sections the design choices while the implementation details will be explained in the next chapter. At the end, in chapter 6, an analysis of some experimental results and a discussion about possible future works will be covered.

4.2 Mobile node

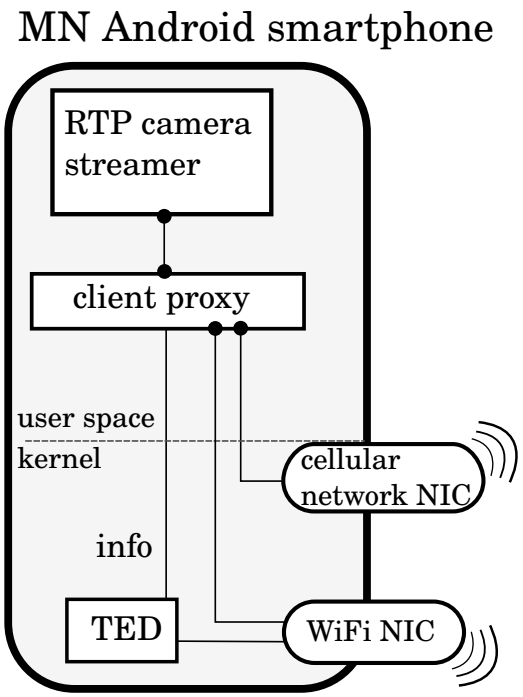


Figure 4.1: MN design structure

As mentioned before, the MN is in practice an Android smartphone. I chose to use an LG Nexus 5 as a reference device since it ships out with the stock Android OS. On the top of the Operating System I chose to use a camera streamer application that periodically sends captured video frames to a remote relay. Also, such camera streamer application allows to specify the destination IP address and the destination port so that the local client proxy can be employed. In fact all the traffic originated by the application passes through the client proxy which in turn forwards all the packets to the relay. Both the application camera streamer and the client proxy reside in Android OS user space while the TED component should be built in the Android kernel. The MN design structure is shown in figure 4.1 and it is clearly a stack: the user interacts with the camera application which exchanges packets with the local client proxy. The latter interacts with the TED module and takes decisions about which NIC it should transmit from.

Figure 4.2 outlines the Android software stack. System applications and user applications such as email clients, games, messaging apps, etc., rely on the Java API Framework. This also applies to many real-time and VoIP user applications. For this reason an RTP camera streamer built in Java has been chosen to be the real-time user application running on top of the Java API Framework stack layer. The client proxy instead directly resides on top of the Native C libraries stack layer. This choice comes from the fact that the original ABPS client proxy for GNU Linux distributions was implemented in C directly employing the GNU libc library. Thus, adapting the ABPS client proxy to the Android architecture does not require the Java API Framework. Moreover such proxy application is not intended to provide any user interaction facilities and the Java API Framework would only introduce needless complexity.

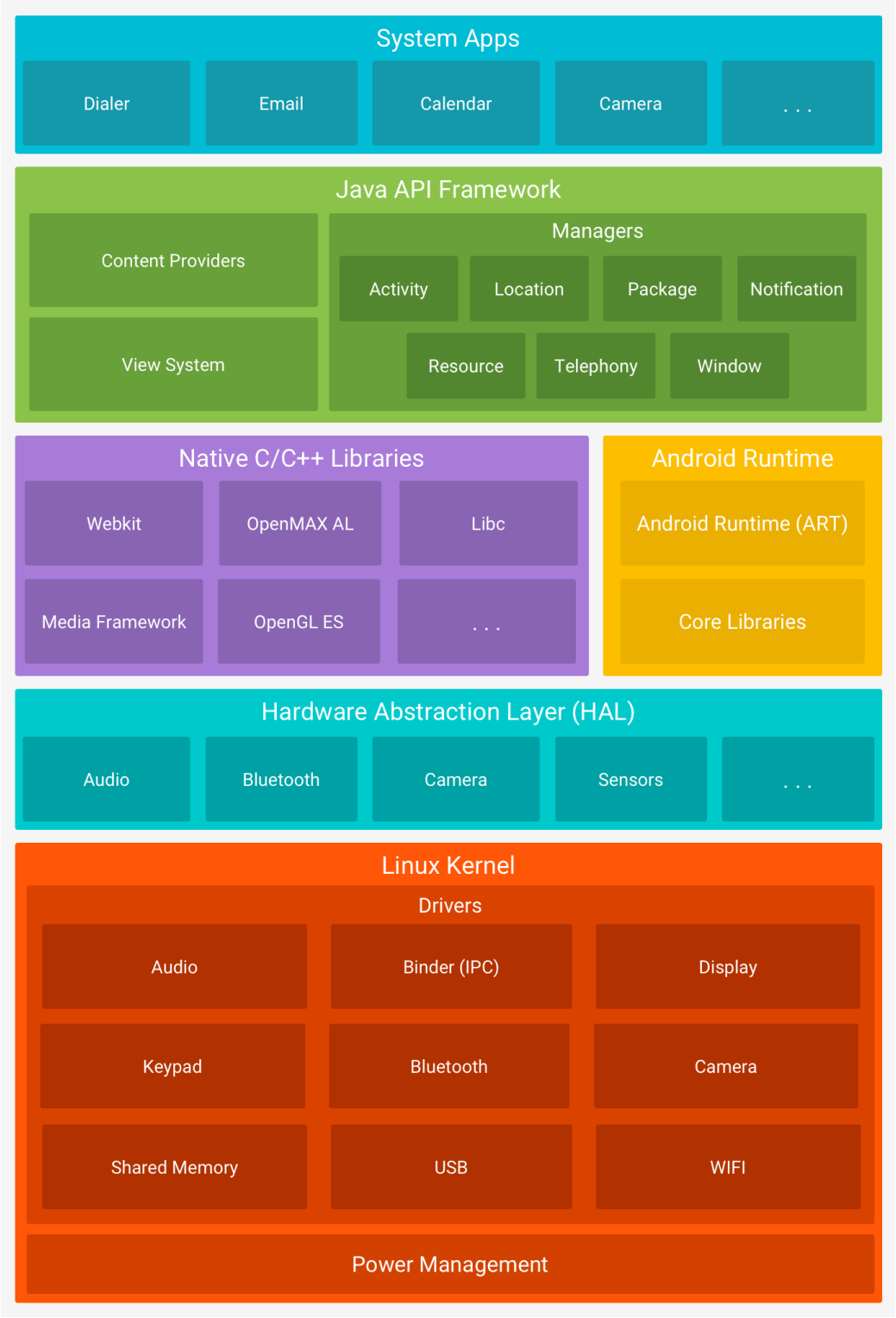


Figure 4.2: Android Platform Architecture[26]

The third ABPS component of interest that runs on the MN is the TED module which resides in the Linux Kernel layer. In particular TED is a software patch that modifies some of the linux kernel components such as the socket structure, the UDP message handler, the IP packet handler and the *mac80211* kernel subsystem. The latter consists of driver APIs for 802.11 WiFi *SoftMAC* devices which are those network interfaces whose frame management is expected to be done in software by the kernel[1]. In fact TED interacts with the mac80211 subsystem software part that handles frames transmission and frames delivery status providing data-link-layer information to the client proxy.

A similar feature makes use of the 802.11 frame ACK status to the application layer already exists in the Linux Kernel and can be employed through the `SO_WIFI_STATUS` socket option. This option was introduced because 802.1X EAPOL handshake implemented in `hostapd` requires knowing the delivery status of 802.11 frames[23]. However this option does not provide information about retransmissions and does not support packets fragmentation: if the application sends a transport layer packet that is big enough to be fragmented then the socket with the `SO_WIFI_STATUS` option enabled reports to the application the ack status of the first fragment only. On the contrary TED provides support for both retransmission information and packets fragmentation. In any case developing TED as an integration of the `SO_WIFI_STATUS` option would be an interesting future work.

The main problem for these approaches is that most of the WiFi chips integrated in smartphones are of the *FullMAC* type thus implementing all the data-link-layer management in their firmwares. FullMAC wireless NICs do not support mac80211 and both TED and the `SO_WIFI_STATUS` option cannot be employed in this kind of WiFi chips. The choice of FullMAC NICs comes form the fact they allow smartphones' processors to save power by offloading certain operations such as *association*, *authentication*, *scanning*, etc. Also, most of the FullMAC NICs' firmwares are closed source and directly maintained by chip vendors which do not allow any external modification.

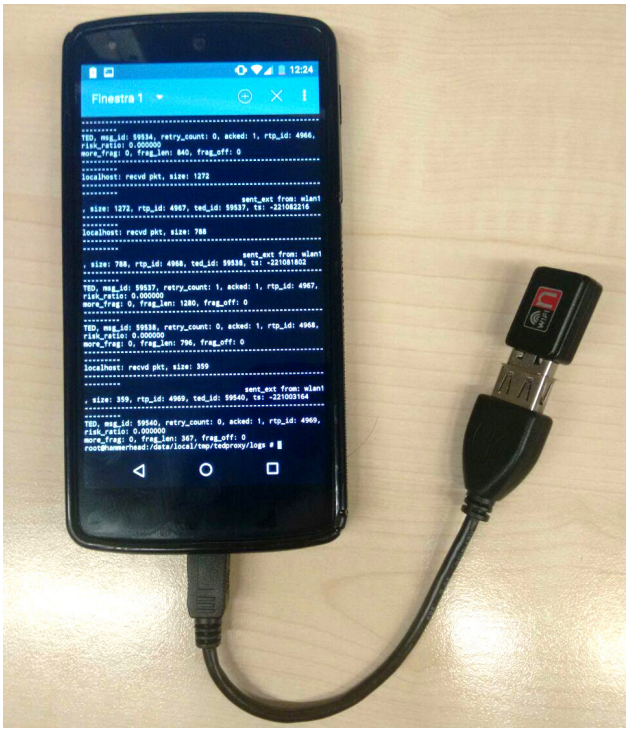


Figure 4.3: Picture of the MN device. An LG Nexus 5 smartphone with an USB WiFi dongle which integrates a Ralink RT2870 chipset.

While the authors of NexMon[81] disassembled and patched the closed source firmware of the Nexus 5 WiFi chip successfully enabling monitor mode, performing reverse engineering for many different chips guaranteeing a reliable QoS is an obviously too difficult task and would introduce enormous complications. On the other hand open source firmwares are not expected to be released by WiFi chips vendors in the early future and a reverse engineering approach for enabling TED functionality in firmwares may be worth a try in future works. Hopefully, the spreading of this kind of works may convince chip vendors to release source code of their firmwares or at least provide a software interface able to access some information internally hold by these FullMAC chips. In any case I opted for an external USB WiFi dongle (figure 4.3) with a Ralink RT2879 chip which is of the SoftMAC type and supports the mac80211 subsystem.

This let me focus on developing the client proxy, enhancing the TED module and testing their behaviour. Moreover, fast vertical handover capabilities may also be advantageous in other contexts with less strict power consumption requirements and different hardware environments such as connected cars[9].

4.3 Relay and Correspondent Node

In order to deploy the MN described in the previous section in a practical scenario, other two components are required: the external relay and the CN.

The external relay is required since I consider both the MN and the CN being behind NATs and firewalls as most of current end nodes are today. The idea was to develop a simple UDP relay application which listens on two UDP ports, one per end-node. If end nodes send an initialization datagram to the relay, the latter would be able to forward incoming datagrams in both directions and the MN and CN can bypass NATs and firewalls. It is important to remember that the MN is equipped with multiple NICs and may encounter network reconfigurations thus the relay may receive initialization messages from different source IP addresses and ports of the same MN. In this case the relay should store and continuously update all the current MN's source IP addresses and ports in order to forward incoming traffic from the CN to all the MN's active NICs. Moreover an authentication mechanism must be employed to avoid undesired forwarding to third-parties. Such authentication mechanism is out of the scope of this work but must be covered in future enhancements.

In this project the CN simply refers to the receiving counterpart of the MN. It should be able to send an initialization message to the external relay in order to be reachable since, like the MN, it resides behind a symmetric NAT and firewall. After the initialization phase the CN should simply listen to incoming RTP (over UDP) messages from the relay, decode the RTP stream, re-encode it into video frames and eventually storing them into an output

file. The idea is to achieve a real-time video streaming service which would be easily extensible to a VoIP or videoconference service in future works.

Chapter 5

Project development

5.1 TED

In this section we will see in detail how TED works and how I contributed to its development.

5.1.1 Previous versions and working principles

Previous implementations of TED already existed for the Linux Kernel. After being developed by Vittorio Ghini and presented in 2011 as part of the ABPS architecture[53], it has been reviewed and modified several times as the Linux Kernel was evolving and being updated. The last working version of TED before my contribution was developed for version 4.0.1 of the Linux Kernel by Gabriele Di Bernardo and Alesssandro Mengoli in their bachelor's thesis works[54][52]. They introduced support for IPv6 and based the interlayer information passing mechanism on the `sk_buff` kernel structure. Simplifying, an allocation of the `sk_buff` structure contains the headers and the payload of a packet together with some additional information that the kernel may use to correctly manage it. We can then think of a direct relation between packets and `sk_buff` structures. When a user space process sends a packet through a TCP or UDP socket, the kernel will create a new `sk_buff` structure and link it to the transport layer header and the payload

of the packet. Then the sk_buff structure is passed to the IP management module of the kernel. At this stage if the payload of the transport layer packet is bigger than the MTU then the kernel splits the sk_buff structure in as many sk_buff fragments as it needs. That is, each new IP packet fragment correspond to a separate sk_buff structure. These new sk_buff structures are stored in a linked list for future retrieval. Each new sk_buff structure, or the old one if fragmentation is not required, are passed to the layer 2 module of the selected transmitting NIC. In case the selected NIC is a SoftMAC WiFi device, sk_buff structures land on the mac80211 module which manage 802.11 frames and where the TED core logic resides. For each sk_buff structure mac80211 creates a data-link-layer frame which is then passed to the NIC device driver. Just before the actual transmission, TED stores in internal ted_info structures some information about every new created frame. ted_info structures are collected in a linked list and indexed through frame sequence numbers. When the corresponding AP sends an acknowledgement frame back to the station or when the station gives up after a certain amount of retransmissions, TED retrieves the corresponding ted_info structure from its list, enriching it with the delivery status information and the retransmission count.

The following snippet shows the ted_info structure.

```
struct ted_info
{
        __le16 mac_frameid;       /* 802.11 frame id */
        uint32_t transport_pktid; /* UDP datagram id */

        /* 80211 layer info */
        u8 acked;
        u8 retry_count;
        /* network layer fragment info */
        u16 fragment_data_len;
        u16 fragment_offset;
        u8 more_fragment;

        struct timespec tx_time;
        struct timespec rx_time;
        struct ted_info *next;
};
```

Some of the information contained in the ted_info structure will be delivered to the user space process which started the send operation.

Currently TED only works with UDP transport layer packets.

To let the user space process receive TED notifications it has to send datagrams using the sendmsg() function. This function allows sending additional control information to the kernel along with the payload. Such piece of information consists of a pointer address to an internal integer variable (the UDP datagram id).

This is done through the msg_header structure which must be passed as argument to sendmsg():

```
msg_header.msg_iov = iov; /* Contains the datagram */
msg_header.msg_iovlen = 1;
msg_header.msg_control = cbuf;
msg_header.msg_controllen = sizeof(cbuf);
cmsg = CMSG_FIRSTHDR(&msg_header); /* Contains ctl info */
cmsg->cmsg_level = SOL_UDP;
cmsg->cmsg_type = TED_CMSG_TYPE;
cmsg->cmsg_len = CMSG_LEN(sizeof(id_pointer));

/* Copy the address of our user space pointer (id_pointer)
 * into the cmsg data. Later the kernel will put a new id
 * directly in our user space pointer accessing cmsg data. */
memcpy((uint32_t *)CMSG_DATA(cmsg), &id_pointer, sizeof(id_pointer));
msg_header.msg_controllen = cmsg->cmsg_len;
sendmsg(sd, &msg_header, MSG_NOSIGNAL | MSG_DONTWAIT);
```

In this way when the kernel takes control of the datagram to be sent, it generates a datagram identifier and copies it directly to the user space variable. Such identifier is also stored in the sk_buff structure in order to let it pass through the transport and network layers, finally allowing TED to store it in the transport_pktid variable of the corresponding ted_info structure(s). Let us notice that since the transport layer datagram may be fragmented, more ted_info structures can refer to the same datagram. After TED is aware of the delivery status and the retransmission number of a transmitted frame it can send a notification the user space process. It achieves that by putting a message in the socket error queue containing delivery status, retransmission count and fragment information. In practice TED

creates a new sk_buff, which corresponds to the notification message and passes it as argument to the sock_queue_err_skb() function. Such function also requires the pointer to the sock structure that identifies the socket which owns the original sent datagram as argument. This latter pointer to the socket structure is stored in every sk_buff. Clearly, the user space process must have enabled error notifications through the IP_RECVERR socket option during the socket creation and should be listening for error messages from the socket after sending datagrams. From the asynchronous nature of this approach comes the importance of a shared identifier between kernel and user space. For instance the user space process may receive TED notifications referring to old sent datagrams and without identifiers it may be hard to know which datagram the notification refers to. This also applies to fragmentation information: even if the user space process identifies the correct datagram, the latter may be composed of many fragments. How the user space process uses such information is application dependent but take for instance the case of the client proxy: it may retransmit a non delivered datagram through the second NIC. We will see in the next section that at the moment the client proxy does not perform datagram retransmission but relies on TED notifications to decide when to start transmitting the following datagrams to the second NIC too. After the efforts of Mengoli and Di Bernardo, TED provided a good implementation of the ABPS "early-packet-loss-detection" mechanism. However it did lack IPv6 fragmentation support thus my first contribution in TED was implementing such functionality.

5.1.2 IPv6 Fragmentation Support

Before showing kernel modifications which enable support for IPv6 fragmentation, let us remember how IPv6 packet headers are structured. Essentially IPv6 headers consists of two parts: a fixed header and a variable set of optional headers called extension headers. The *Next Header* field in the fixed header indicates the type of the first extension header. It is present in all the extension headers indicating the type of the following header (if any).

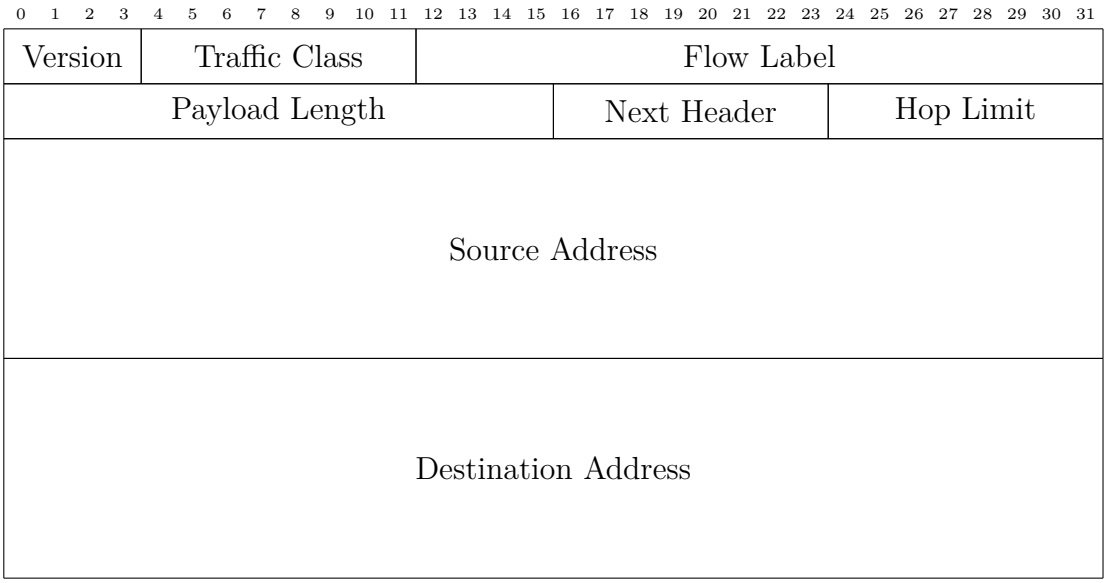


Figure 5.1: IPv6 Fixed Header format

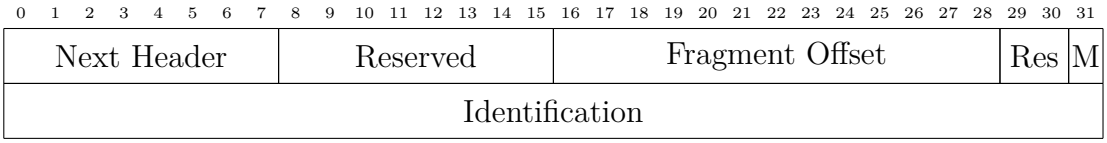


Figure 5.2: IPv6 Fragment Extension Header format. The *M* field refers to the *More Fragment* flag and indicates if the current fragment is the last one (0) or if there are successive fragments (1).

The *Next Header* field of the last extension header, or of the fixed header if no extension header is present, specifies the type of the upper layer protocol header (e.g. UDP or TCP). In fact, extension headers form a chain connecting the fixed header and the upper layer protocol headers through the *Next Header* field.

Figure 5.1 shows the IPv6 fixed header format. Let us notice that the IPv6 fixed header does not include *Flags* and *Fragment Offset* fields which are instead used in the IPv4 header to specify fragment information. In IPv6 headers the *Fragment Extension Header* (fig. 5.2) includes such information.

Let us see now how I introduced support for IPv6 fragmentation in TED. The `ipv6_get_udp_info()` function retrieves IPv6 fragment information of a frame to be transmitted before being passed to the NIC driver. The modified `ipv6_get_udp_info()` function scans the whole extension header chain until it finds the fragment header:

```
*fragment_offset = *more_fragment = hdrs_len = error = 0;
nexthdr = ipv6_hdr(skb)->nexthdr;
target = NEXTHDR_FRAGMENT;
do {
        struct ipv6_opt_hdr _hdr, *hp;
        unsigned int hdrlen;
        found = (nexthdr == target);
        if ((!ipv6_ext_hdr(nexthdr)) || nexthdr == NEXTHDR_NONE)
                break;
        hp = skb_header_pointer(skb, offset, sizeof(_hdr), &_hdr);
        if (hp == NULL) {
                error = -EBADMSG;
                break;
        }
        if (nexthdr == NEXTHDR_FRAGMENT)
                hdrlen = 8;
        else if (nexthdr == NEXTHDR_AUTH)
                hdrlen = (hp->hdrlen + 2) << 2;
        else
                hdrlen = ipv6_optlen(hp);
        if (!found) {
                nexthdr = hp->nexthdr;
                offset += hdrlen;
        }
        hdrs_len += hdrlen;
} while (!found);
```

Actually this is just a sanity check because the fragment header should be the first successor of the fixed header according to RFC 2460 specifications[45]. In any case, once the fragment header is found the `frag_off` bitmask can be accessed and the *Fragment Offset* and the *More Fragment* fields can be retrieved:

```
/* fh is the pointer to the fragment header struct accessed through
 * its offset from the beginning of the fixed header */
fh = skb_header_pointer(skb, offset, sizeof(_frag), &_frag);
if (fh) {
        *fragment_offset = ntohs(fh->frag_off) & ~0x7;
        *more_fragment = ((fh->frag_off & htons(IP6_MF)) > 0);
}
```

The last information we need to obtain is the *Fragment Length*. When a transport layer packet is fragment in more IPv6 packets, only the fixed header and the fragment extension header must be replicated in each fragment. Thus the correct length of a fragment must not include the fixed header length and the extension header length. The *Payload* field in the fixed header already excludes the fixed header length thus the correct fragment length is calculated subtracting the fragment extension header length from the payload field value. Actually, in consistency with the previous sanity check, everything between the end of the fixed header and the end of the fragment header is subtracted from the payload:

```
*fragment_data_length = ntohs(payload_iphdr->payload_len) - hdrs_len;
```

5.1.3 TED porting on android custom linux kernel 3.4

Once TED also supported IPv6 fragmentation I started with adapting TED for the linux kernel version installed on the Nexus 5. Most of the Android smartphones run with old versions of the linux kernel. In detail, the Google team periodically forks the official kernel, often called vanilla[21], and starts adding support for new device drivers and for the Android IPC system called Binder[2]. It is important to notice that these modifications the Google team implements are maintained in a separate repository and are not pushed on the vanilla kernel. At the time I started this project the Nexus 5 smartphone was running with Android modified linux kernel version 3.4. Thus I had to backport the TED module to such linux kernel version and test it on the Nexus 5 device. After this operation an unexpected issue was occurring which did not allow TED to work properly: essentially I found that in version 3.4 the `sk_buff` structure was being orphaned before being passed down to the mac80211 module except for the case `SO_WIFI_STATUS` or `SO_TIMESTAMP` socket options were being set in socket creation. Orphaning a `sk_buff` structure means cutting its reference to the socket (`sock` structure) which owns it. This prevented TED from correctly sending error message to the user space process since the socket error queue was not accessible.

This behaviour changed at some point of the vanilla kernel around version 3.11 with a kernel patch that removed the `sk_buff` orphaning. Interestingly, this patch was also introduced in a later subversion of the vanilla kernel 3.4, precisely in the 3.4.35, but not in the Android custom kernel 3.4. This shows how the Android custom linux kernel and the official vanilla linux kernel follow two completely different development branches. However, after having spotted the patch I applied it to the Android custom kernel and TED started working correctly.

5.1.4 Refactoring

Different developers put their hands on TED often focusing on different features without any implementation process guidelines. Obviously such approach is preferable when a quick proof of concept must be released in a short time but it may increase the risk of messy code which would be hard to maintain in future. Things become worse if the software to be maintained depends on other highly dynamic piece of software which is continuously changing over time such as the linux kernel. When I started looking at the TED source code it required me some time before understanding all of its parts and how it was interacting with the kernel. Moreover the TED core included duplicated functions, code written following different coding styles; also it lacked clear comments.

These characteristics reflected the development life of this software thus I thought a refactoring phase was necessary for simplifying future development of TED. First of all I adopted the linux kernel coding style[16] as best I could for re-writing TED in order to enhance readability and coherence with the rest of the kernel source code. Then I removed unreachable code and tried merging duplicate functions and structures where possible. As a last step I put all the modification TED introduced in a set of patches, one for each supported version of the kernel. Patches allow developers to quickly spot where modifications are. Moreover, patches are easily adaptable to different software versions.

Hoping future developers would benefit from my TED refactoring efforts, I encourage them to maintain good code quality in future works.

5.1.5 Open issues

Currently TED presents two implementation bugs which must be resolved in future developments:

1. Few datagrams which pass through the TED module seem to not be transmitted at all. At some point after the `sendmsg()` invocation the TED patched kernel fails the send operation. This issue has been spotted during tests and such lost datagrams have not been counted as "losses" in the experimental results since they occurred in an area with good WiFi signal and far from the handover area.

2. TED sometimes notifies the user space process with false positives. For instance just after a TED notification indicating a non acknowledged frame a successive TED notification arrives, indicating its frame has been acknowledged and no data-link-layer retransmission has occurred. These frames are suspicious since they are not being delivered to the relay. This issue seems to occur more frequently in consecutive tests without rebooting the device.

Both issues are serious and their cause is hard to spot by analysing the source code. A deep debug would be required to fix them. Also sniffing radio traffic may be a good strategy in order to determine the actual status of frames. Future developers could however consider to re-implement TED features as an extension of `SO_WIFI_STATUS` option. Since the latter is already well integrated in the kernel, such choice may resolve the aforementioned issues.

5.2 Proxy Client

In the previous chapter we have seen how the ABPS proxy client logic works. Let us now focus on its implementation details. I started developing such proxy enhancing an old application which aimed to test TED capabilities.

5.2.1 Network

Let us remember that the ABPS proxy client (called `tedproxy`) must be able to forwards datagrams originated by the real-time user application (RTP camera streamer) to the external relay through one chosen NIC or through both NICs simultaneously. Also, `tedproxy` must be able to forward datagrams received from the external relay back to the application. To achieve these goals `tedproxy` creates one UDP socket for receiving/sending datagrams from/to the application and two UDP sockets, one for each NIC, for sending/receiving datagrams to/from the external relay. The latter two sockets are directly bound on the NICs by enabling the `SO_BINDTODEVICE` socket option during their creation phase. Such option allows `tedproxy` to avoid reconfiguring internal routing tables every time a NIC obtains a new IP address after a network reconfiguration.

I chose to handle forwarding of datagrams according to an event based policy. Essentially when a datagram is placed by the kernel in the input queue of a socket, an "input" event arises in `tedproxy` which reads and forwards such datagram. More precisely the forward operation is "lazy", meaning that the read datagram is first placed in an output queue and then eventually forwarded after `tedproxy` finishes reading pending messages from the input queue or it reaches a maximum number of reads. For instance, when the RTP camera streamer sends a datagram (or more) to `tedproxy`, this one awakes from its event-waiting state, reads the datagram content and pushes it to one of its internal outgoing queues. That is, `tedproxy` keeps an internal output queue for each socket. Thus in case the read datagram was sent from

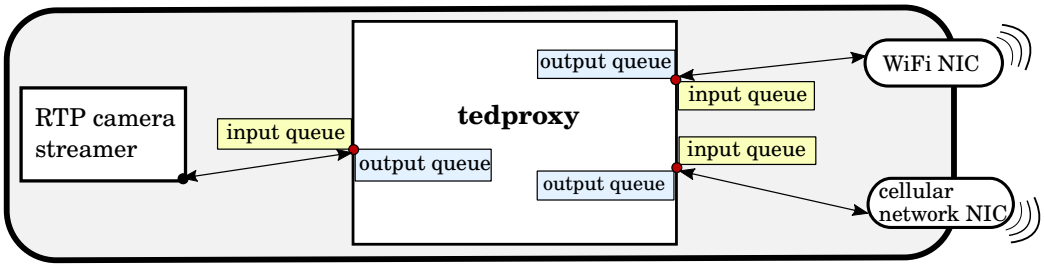


Figure 5.3: `tedproxy` internal output queues and socket input queues.

the local application the outgoing queue is the one whose socket is bound to the currently selected transmitting NIC. Clearly since `tedproxy` may decide to transmit through both NICs simultaneously, datagrams may be pushed to both output queues. Figure 5.3 outlines `tedproxy` internal output queues and socket input queues. To clarify, socket input queues are handled by the kernel while output queues are internally defined (in `tedproxy`) arrays of messages used to implement lazy forwarding. Indeed sockets also have their output queues which are handled by the kernel but for simplicity they are not shown in figure 5.3.

For convenience let us call "TED-enabled socket" the socket whose related datagrams are sent with TED notifications enabled and which is bound to a mac80211 compliant WiFi NIC. As we have seen in the previous section TED notifies the user space process, `tedproxy` in this case, by pushing error messages in the error queue of the TED-enabled socket. Thus `tedproxy` also awakes for error events related to the socket which is bound to the WiFi NIC (if supporting mac80211). Thanks to error messages `tedproxy` becomes aware of the delivery status and retransmission count of previously sent (over WiFi) datagram fragments. According to such information it decides whether or not to also start sending datagrams from the cellular NIC. It is important to note that if a NIC loses connectivity (e.g. disassociates form the AP or is turned off) `tedproxy` simply suppresses related error messages and no socket recreation is required thanks to the `SO_BINDTODEVICE` option and since only UDP sockets are taken into account.

5.2.2 Handover parameters

The following code snippet is executed every time a TED notification is received:

```
if (!ted_info->status || (ted_info->retry_count > conf.retry_th)) {
        esock->pkt_risk++;
} else {
        esock->pkt_ok++;
}
risk_ratio = ((float)esock->pkt_risk)/conf.cwin;
if (risk_ratio > (float)conf.ratio_th) {
        /* Enabling support device */
        *ted_dev_only = 0;
        esock->pkt_risk = esock->pkt_ok = 0;
}
if (esock->pkt_risk + esock->pkt_ok > conf.cwin) {
        if (risk_ratio <= conf.tolerance && !(*ted_dev_only)) {
                /* WiFi link is stable, deactivating support device */
                *ted_dev_only = 1;
                /* ... omitted code ... */
        }
        esock->pkt_risk = esock->pkt_ok = 0;
}
```

Essentially `tedproxy` maintains two counters for the TED-enabled socket: `pkt_ok` which counts the number of acknowledged frames and `pkt_risk` which counts the number of non delivered frames plus the number of frames which have been retransmitted more than `retry_th` times. This latter is in fact a threshold over which a frame is considered at risk. Actually such frame may be delivered but the high number of retransmissions indicates a bad link and subsequent frames may be lost. For every received TED notification `tedproxy` calculates the ratio of risky frames over all the frame sent inside a certain window, called critical window or `cwin`. If such ratio, called `risk_ratio`, is higher than a predefined threshold, called `ratio_th`, than subsequent datagrams will be sent from the support (cellular network) NIC too. Moreover when the `cwin` limit is reached `tedproxy` resets counters and checks if `risk_ratio` has decreased below a certain `tolerance`. If so `tedproxy` deactivates the support NIC considering the WiFi link as stable. This approach reflects the "WiFi first" policy of ABPS.

5.2.3 Basis for datagram retransmission

In addition to delivery status and retransmissions information, the TED-originated datagram id and information about fragmentation are also present in TED notifications. These values are required to understand which datagram and which fragment the notification refers to in order to perform datagram retransmission on the support NIC. At the moment datagram retransmission is not yet implemented in `tedproxy` but I already worked developing some of the core features it requires: when a datagram is sent through the TED-enabled socket, `tedproxy` stores it into a hashtable[1] indexing it by its TED originated datagram id. Later, when `tedproxy` receives a TED notification, it retrieves the original datagram from its internal hashtable. Moreover it can identify which fragment of the retrieved datagram the TED notification refers to. This might be useful to determine a criterion for datagram retransmission. For instance if a sent datagram is composed of many fragments and just few of them encountered retransmissions, datagram retransmission through the support NIC may be unnecessary. On the other hand if many fragments of a big datagram are considered at risk (due to high retry count or not being delivered to the AP), datagram retransmission through the support NIC may be convenient. More investigations and analysis are encouraged for future works.

5.3 Relay and CN tools

In order to experiment with MN behaviour in a realistic environment I developed a simple relay application written in python and software tools to let the CN work with the relay and the *ffmpeg* application.

In the first place, the relay application essentially advertises two UDP ports which both the MN and the CN can send datagrams to. Once both end

[1] I used a hashtable implementation written by Davide Berardi[37]. Thanks to its macro-based and compact design I could easily include it in `tedproxy` without any external dependency.

nodes sent their first datagram to the relay, the latter can forward subsequent datagrams to both directions.

The following simplified code snippet outlines the simple logic of the relay:

```
while True:
        ready_socks,_,_ = select.select([sl, sr], [], [])
        for sock in ready_socks:
                data, addr = sock.recvfrom(MAX_BUFF_SIZE)
                if sock.fileno() == sl.fileno():
                        #Received packet from left

                        if leftSources is None:
                                leftSources = []

                        if addr not in leftSources:
                                leftSources.append(addr);

                        if rightSource is not None:
                                if sn > lastSN:
                                        sr.sendto(data, rightSource)
                elif sock.fileno() == sr.fileno():
                        rightSource = addr;
                        if leftSources is not None:
                                for addr in leftSources:
                                        sl.sendto(data, addr)
```

Essentially the relay waits for input events from two sockets, a "left socket" `sl` and a "right socket" `sr`. The first one is related to the MN's incoming datagrams while the second one is related to CN's incoming datagrams. When the first datagram is received by the right socket (CN side) the relay marks the datagram's source address as `rightSource`. On the other hand since the MN is likely equipped with multiple NICs and can deliver datagrams with different source IP addresses, when a datagram arrives from the left socket the relay checks if its source IP address has already been registered and if it's not it adds it to the `leftSources` array. When both `rightSource` and `leftSources` have been initialized, meaning both end nodes have sent their first message, the relay can start forwarding from left to right and from right to left. In the latter case the relay forwards datagrams to all the registered MN's source IP addresses. At the moment the relay does not implement any aging policy for such IP addresses but a

simple solution may consists in maintaining only one MN source IP address per MN NIC.

Another thing to notice is that the relay also implements a simple policy to discard duplicate datagrams coming from the MN. To achieve these features the relay reads the RTP serial number `sn` from the payload of all incoming datagrams from the MN. Thus it forwards a datagram from left to right only if its `sn` is more recent than the most recent `sn` obtained until such reception. Such policy is obviously naive and fails when datagrams are not received in a sorted fashion. However it resulted sufficiently reliable for initial experimentation and analysis. A simple solution for future works would be to maintain a limited buffer of forwarded datagrams' serial numbers and forward an incoming datagram only if its `sn` is not still contained in the buffer.

The CN tools I implemented are quite simple. One of the them, called `cnproxy`, is essentially a local proxy which begins with sending an initialization datagram to the external relay and then starts listening for relay response datagrams. It should be clear now that those response datagrams the relay sends to the CN are originated by the MN. `cnproxy` then forwards all the relay response datagrams to a local running instance of `ffmpeg` which, as already mentioned before, is able to decode RTP messages, re-encode them into video frames and create an output video file. The second tool I implemented simply acts as a local HTTP server which provides video format specifications to `ffmpeg` through an SDP formatted file. This is required since no SIP handshake is yet performed between end nodes.

Trough the external relay and the CN tools both nodes are able to bypass their NATs and firewalls and initiate an RTP communication. The implemented system supports a bidirectional RTP communication at the moment but the CN only provides a receiving entity. We will see in chapter 6.2 how extensions of the current implemented architecture would be easy to develop in future works.

Chapter 6

Experimental tests

In this chapter I will show results obtained from tests conducted in a typical scenario in which an MN first leaves the coverage area of the current selected WiFi AP, then uses the cellular network for a certain amount of time and eventually connects back to the previously accessed WiFi AP. In particular the MN was a LG Nexus 5 device running Android 6 Marshmallow with a TED-enabled kernel and the ABPS client proxy. An RTP camera streamer was used as a simple real-time application. The captured video of the smartphone camera was being sent to an intermediate relay server and lastly forwarded to a remote CN.

The purpose of these tests is to show how the ABPS components at the MN can improve vertical handover process. In the following sections we will see some technical aspects of the experimental setup and an analysis of obtained experimental results. At the end, considerations about future works will be addressed.

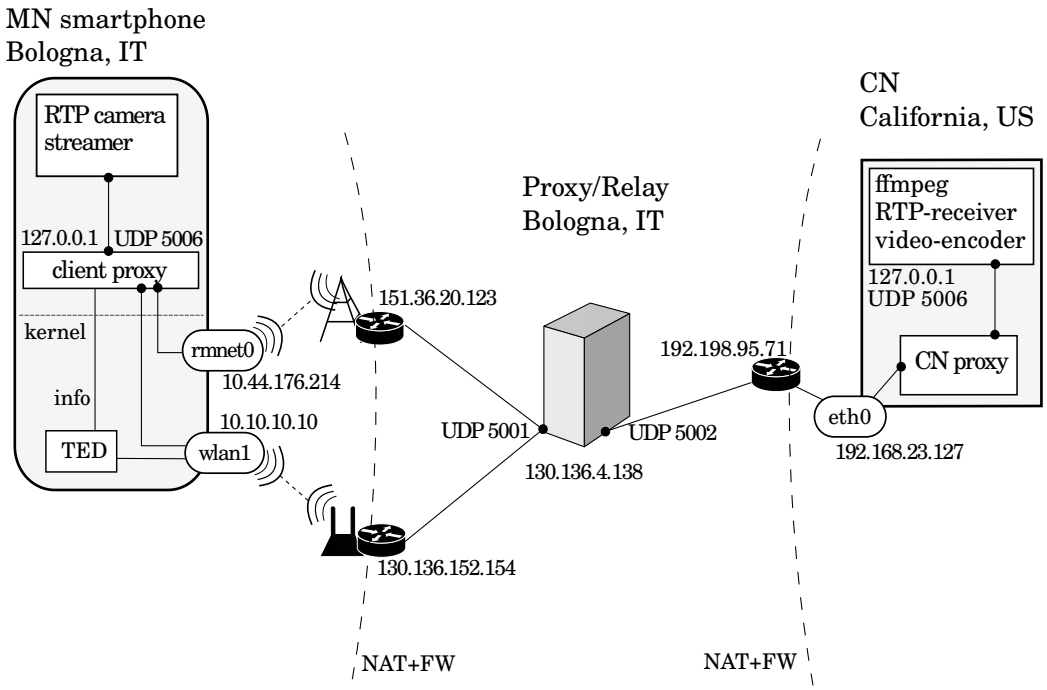


Figure 6.1: Experimental setup.

6.1 Experimental Setup

Figure 6.1 outlines the setup in which I conducted the experiments. I used a LG Nexus 5 smartphone as a multi-homed MN which could connect to both WiFi and cellular networks (UMTS, HSPA, LTE, etc.). As mentioned before, I modified the MN Android system substituting the factory kernel with a TED-patched kernel in order to enable support for the "early-packet-loss-detection" mechanism. Let us remember that the inner WiFi module is of the type FullMAC and could not be used with TED since it does not support the mac80211 subsystem (sec. 5.1.5), thus an external USB WiFi dongle was used instead (4.3). The MN also employed the ABPS client proxy, called `tedproxy`, which dealt with traffic redirection to the NICs. The real-time application I used was simply streaming video frames captured from the camera to `tedproxy` which bound an UDP socket to a custom port on localhost. In fact, the camera application was streaming UDP datagrams

containing data frames encoded with the RTP format and it essentially acted as the sending part of a real-time VoIP or videoconferencing application.

On the opposite side, the CN was a remote Linux Virtual Private Server (VPS) running two software components: a local proxy which was simply forwarding incoming RTP traffic to a localhost UDP port, and *ffmpeg* which is a video converter capable of decoding an RTP data stream, re-encoding the stream in video frames and storing them in a video output file. In practice `ffmpeg` acted as the receiving part of a real-time VoIP or videoconferencing application reading UDP datagrams on a local UDP port which the CN local proxy was forwarding traffic to.

Both proxies of the two end nodes had to contact the intermediate remote relay first in order to bypass firewalls. As explained before, the intermediate relay first exposes two UDP ports, one per end node, then the MN and the CN send an initialization packet to the relay to let their OSes open the corresponding ephemeral ports to which the relay can then respond to. Also both end nodes are behind NATs and could not reach each other through their NIC IP addresses without relying on a public intermediate proxy or forwarding ports at their borders routers. Thus the remote relay server lets end nodes bypass firewalls and NATs and it relays RTP traffic between them. Let us also remember that each time the MN transmits from a different NIC and each time a network reconfiguration occurs on the MN, the relay registers the new source IP address and port in order to forward traffic from the CN back to all the MN's NICs.

Tests were performed physically at the Department of Computer Science in Bologna (Italy) where both the MN and the intermediate relay were located while the CN was deployed in California (US). Network RTTs were: ~60ms between MN cellular network NIC (with LTE technology) and the relay, ~20ms between MN WiFi NIC (with optimal coverage range) and the relay and ~125ms between the relay and the CN. The camera application was transmitting at a variable data-rate of ~400Kbps.

With these tests I wanted to investigate the behaviour of the MN's soft-

ware parts which aim to enhance the vertical handover process. In particular I focused on client proxy responsiveness and TED reliability in a realistic mobility scenario. It is clear that the setup does not take into account transmissions originated by the CN directed to the MN. It actually considers unidirectional communications only. Providing bidirectional communications to the environment however would be a simple enhancement for future works.

Appendix A contains detailed documentation explaining the steps required to retrieve, build, install and configure all the software parts necessary for deploying the experimental setup and reproducing the experiments. In the next section we will examine some experimental results.

6.2 Experimental Results

I performed 20 tests, 10 while enabling `tedproxy` and the TED module on the MN and 10 without involving TED and `tedproxy` at all. Each test lasted around 2 minutes during which I was walking from inside the CS department laboratory, where the WiFi AP was located, to the outdoor environment out of WiFi AP coverage and then moving back to the starting point close to the WiFi AP. During these experiments the MN was in the coverage range of LTE technology cells with poor signal strength inside the laboratory but better signal strength once outdoors. In all the tests both NICs were activated and the OS had just to reconfigure routing tables when associating/disassociating to/from the WiFi AP. In half of the experiments, 5 over 10, that I performed without involving TED and `tedproxy` the camera streamer application was drastically interrupting the RTP stream during the handover process. Figure 6.2 shows the resulting camera streamer error message. Also the camera streamer application was restarting the RTP stream only after a 30 seconds timeout or after the user pressed the “Try again” button. With the new stream, the RTP packet counter was also being reset to 0. Such error message is likely due to the fact that the application tried to send a datagram before the underlying system recognized the WiFi link disruption

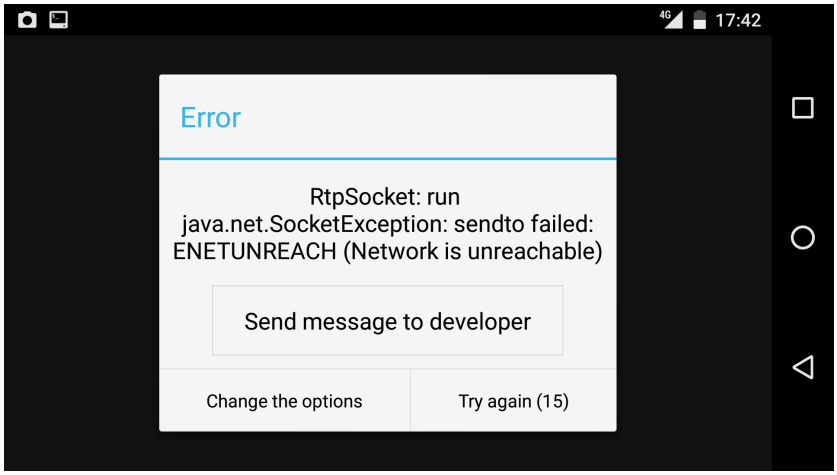


Figure 6.2: Camera streamer application error.

and reconfigured the routing tables in order to let packets be transmitted through the cellular NIC. This did not happen in any of the tests in which TED and `tedproxy` were not used since the camera application was always transmitting to `tedproxy` thus always having a stable route to it. Moreover, since `tedproxy` relied on sockets created with the `SO_BINDTODEVICE` it could always choose to transmit from both NICs, however the routing tables were configured. Indeed a more aggressive application could suppress such error message and directly retry sending without waiting any timeout. In any case, waiting for a timeout before restarting the session may be reasonable for user applications since they are not aware of the MN's connection status. Also remember that ABPS aims to work with any applications without requiring any modifications.

In the 5 remaining tests conducted without involving TED and `tedproxy` that did not fail with any error message the camera application can be considered as an application with a more aggressive approach that does not wait any timeout before re-starting the RTP-stream. We are now going to see that in any case `tedproxy` and TED caused the vertical handover process to be more seamless.

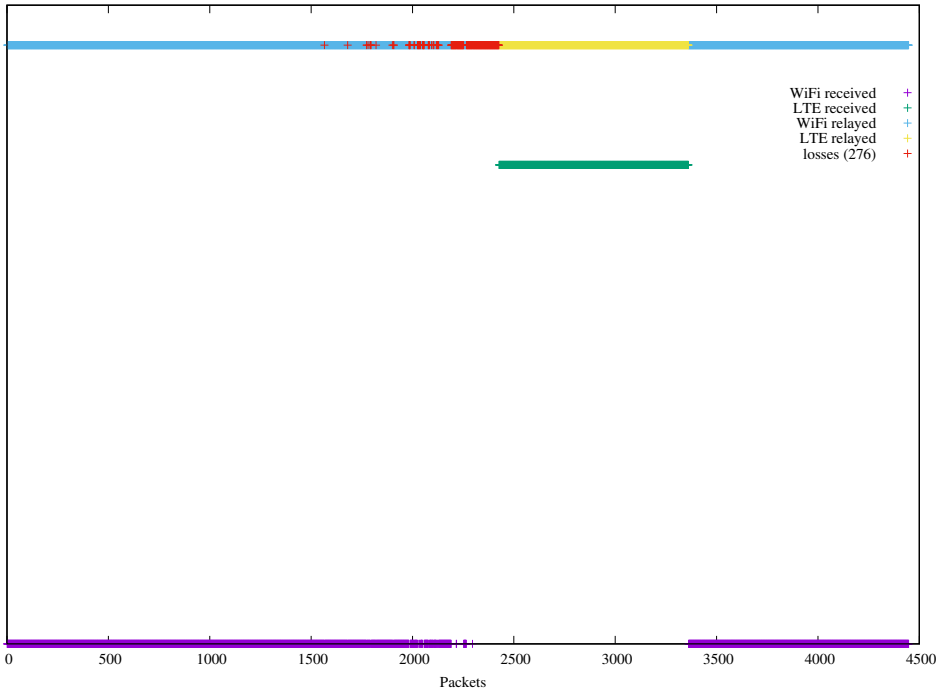


Figure 6.3: Results of one experiment in which TED and `tedproxy` were disabled.

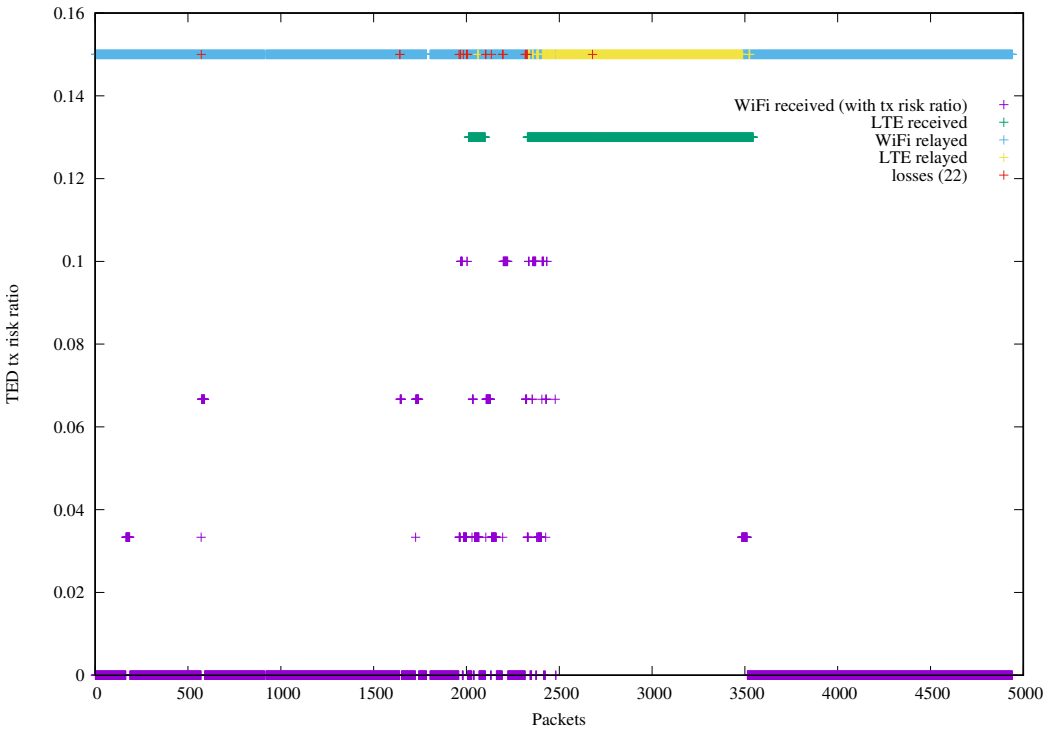


Figure 6.4: Results of one experiment in which TED and `tedproxy` were enabled.

Figure 6.3 and 6.4 show plotted results of two experiments: TED and `tedproxy` were disabled in the first and the camera application did not fail during vertical handover, in the second TED and `tedproxy` were enabled and used. Plotted data represent RTP packets received by the relay. Violet and green points in particular represent packets received by the relay that were sent from the MN's WiFi NIC and from the MN's cellular network NIC respectively, light blue and yellow points represent packets received by the relay and forwarded to the CN that were sent from the MN's WiFi NIC and from the MN's cellular network NIC respectively. Let us remember that when the relay receives two duplicate packets it only forwards the one which arrives first, hence light blue and yellow points also indicate which NIC prevailed. The x-axis indicates the RTP packet identifiers while the y-axis indicates the `risk_ratio` value calculated by `tedproxy`. In fact the y-axis is relevant only for violet points of figure 6.4. Lastly red points indicate packets not received at all by the relay (losses). Actually in TED enabled experiments a small amount of losses caused by an unresolved bug, mentioned in section 5.1.5, have not been considered since they were not sent at all by the WiFi NIC and because they occurred when both signal strength and TED notifications indicated good link quality. Indeed, these results do not claim to be accurate and do not provide reliable metrics but want to offer a first order analysis of the ABPS components' behaviour in a realistic scenario. In figure 6.4 we can observe how the MN was starting to transmit from both NICs when `risk_ratio` was reaching the 0.1 threshold (`risk_th`). Also, `cwin` was set to 30, `retry_th` to 4 and `tolerance` to 0.07. I refer to section 5.2.2 for an explanation of such parameters. Briefly this means that if the number of risky packets was reaching 10 percent of all the sent packets then `tedproxy` was starting to use the cellular network NIC too. Every 30 packets sent, `tedproxy` was resetting packet counters and if `risk_ratio` was decreasing back below the `tolerance` value then `tedproxy` was considering the WiFi link as stable and thus disabling the cellular network NIC. In contrast figure 6.3 shows how, without relying on `tedproxy` and TED, the MN starts transmitting from the

cellular network NIC only after the WiFi signal completely deteriorates and the station disassociates from its AP. This leads to higher data loss and longer unavailability periods.

For completeness figure 6.5 shows results of one of the experiments in which TED and `tedproxy` were disabled and the application was restarting the RTP stream after displaying an error message and a "retry" button. In these tests in particular I pressed the "retry" button as soon as possible. In such figure the x-axis indicates elapsed seconds from the beginning of the experiment instead of RTP packet ids. This modification was necessary since the camera application was resetting the RTP packet counter generating a fresh session. While previous figures showed a comparison of packet losses, figure 6.5 wants to give an idea of the long unavailability period, ~13 seconds between the WiFi disassociation and the communication recovery. I refer to unavailability period as the longest time passed between the reception of two consecutive datagrams at the relay in one experiment.

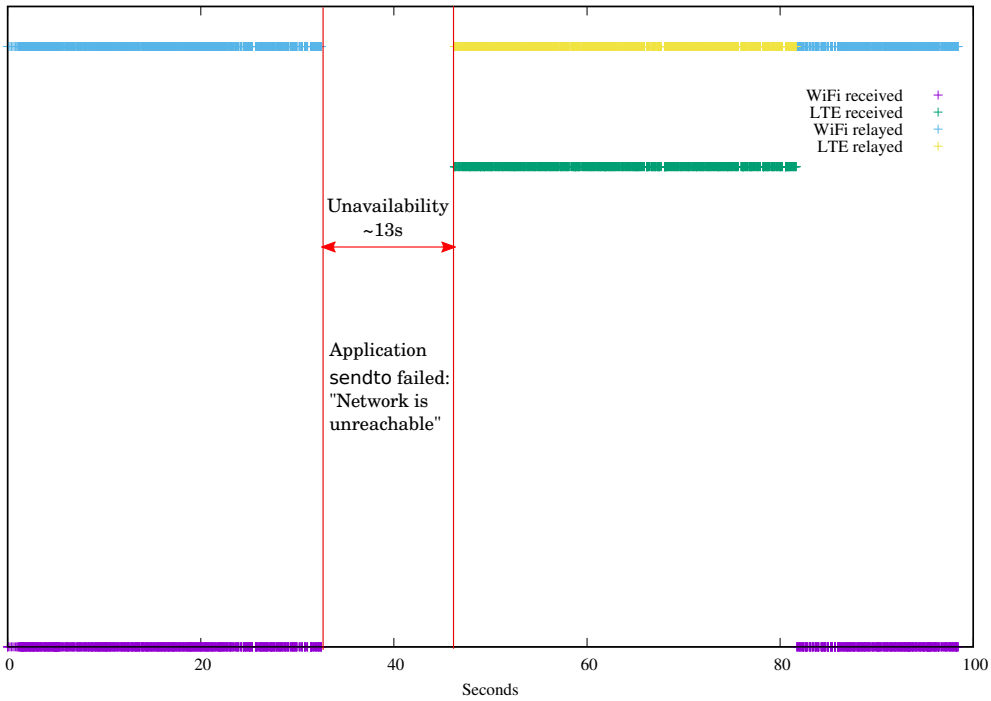


Figure 6.5: Results of one experiment in which TED and `tedproxy` were disabled and the application camera raised an error.

	tedproxy & TED	no tedproxy no TED	no tedproxy no TED with error app
average packet loss	36	364	450^{estimate}
average unavailability time	458 ms	7219 ms	10393 ms

Table 6.1: Results of conducted experiments in average.

Let us now consider some results in average which have been calculated considering the handover period only, excluding possibly corrupted and non handover related data caused by TED issues documented in section 5.1.5. Experiments involving TED and `tedproxy` resulted in an average packet loss of 36 packets and an average unavailability period of $458ms$. Moreover, the experiments that did not involve TED and `tedproxy` and in which the application did not raise any error resulted in an average packet loss of 364 packets and an average unavailability period of $7210ms$. Lastly the experiments that did not involve TED and `tedproxy` and in which the application failed with the previously mentioned error resulted in an average unavailability period of $10393ms$. In these last experiments the correct number of packet losses was not calculated because the application reset the counter of RTP packets after failure. However a simple estimate considering the average bitrate and the average unavailability period resulted in around $400 - 450$ packet losses. Important to note is that these estimated losses refer only to the unavailability period while in the other experiments those losses concerned the whole handover period. Table 6.1 sums up these values.

Even if still inaccurate these results highlight the potentiality of the ABPS fast vertical handover method encouraging further developments and enhancements. Also it is worth noticing that when the MN moved again closer to the WiFi AP, its station did not re-associate to the AP until signal strength reached a certain threshold. This late association policy of Android

OS helps the handover process in this particular scenario where a good LTE coverage was provided. Further investigations may point out that with poor cellular network signal strength it may be convenient to associate earlier to the WiFi AP with lower signal threshold and start transmitting from both NICs sooner.

Future works and conclusions

Future works

As already pointed out in this document, ABPS software is in early development stage. Despite the promising idea behind the project more effort must be put in its implementation. Some of the most important features future developers should focus on are:

- Integrate TED features with the `SO_WIFI_STATUS` socket option. TED would benefit from a clearer implementation and sockets used for sending TED notifications would have a specific option set and would be treated differently by the kernel. Moreover, maintaining TED between different kernel versions would be easier and the TED patch size would be considerably reduced as the `SO_WIFI_STATUS` option already exists in the mainline kernel. Also features enabled by the `SO_WIFI_STATUS` socket option would benefit from fragmentation support and retransmission information advertisement which might attract the interest of other developers, giving TED features more chances to be accepted in the mainline linux kernel.

- In section 5.1.5 we have seen some currently unresolved issues in TED. Indeed resolving such issues would be a high priority task for future works. However, adapting TED to the `SO_WIFI_STATUS` socket option would require significant modification to TED's current implementation and that may resolve those issues by itself.

- More handover criteria: an interesting enhancement could consists in adding more information in TED notifications, such as the current bitrate. For instance a MN that is moving far from a WiFi AP may decide to start transmitting earlier from the cellular network NIC too after detecting the advertised bitrate over WiFi has decreased, even if frames are still being acknowledged. Also `tedproxy` might introduce more handover criteria such as RSSI, bandwidth, RTT, jitter, etc. on both NICs' communication channels. Obviously such information can not be retrieved with a "per-packet" granularity because it would require some time to be calculated. It may be however worth considering to improve the handover decision.

- Since TED notifications contain information about fragmentation it would be quite easy to add support for datagram retransmission in `tedproxy`. As already mentioned in section 5.2.3, `tedproxy` was implemented with this feature in mind. In fact a set of functions and the environment to achieve datagram retransmission already exist in `tedproxy` core.

- An implementation of the ABPS Oracle described in section 2.2.3 exists for Android 4.4 (KitKat). Integrating it with other ABPS software components would be essential to offer more credibility to the whole architecture. At the time of writing in fact the device which `tedproxy` runs on must have both NICs always active to let parallel NICs transmission work properly. This clearly lead to a faster battery drain. The ABPS Oracle would take care of activating and deactivating NICs only when necessary according to an internal database of WiFi APs and cellular network cells mapping. Integration should be quite simple and would require porting the existing Oracle implementation to the current and next versions of Android.

- Future developments should provide signalling capabilities to ABPS intermediate proxies in order to support SIP or Jingle compliant ap-

plications. Also in order to support those closed source applications which rely on proprietary signalling protocols and do not allow explicit definition of proxies further investigations may focus on implementing ABPS services with a hidden proxy based architecture. In this way applications would not be aware of ABPS intermediate proxies working to enhance vertical handover process.

- For future experiments extending the test scenario would be a strong requirement. At the moment the experimental environment only supports tests employing unidirectional RTP streams. Testing bidirectional RTP streams would be easily achievable by adding an RTP receiver on the MN and a RTP streamer on the CN. Then the successive step would be experimenting real-time communication applications. *Jitsi*[11] may be a good candidate application since it supports explicit proxy definition and both SIP and Jingle signalling.

- At the moment the most hard to solve limitation for ABPS approach in smartphone devices is that their integrated WiFi chips are of the FullMAC type. As we have seen in section 4.2 this prevents TED to access data-link-layer information held inside the WiFi chip firmware. Future investigations should focus on discovering the presence of mobile devices with SoftMAC chips or FullMAC chips with open source firmwares. Since this direction seems quite discouraging it may be worth to try contributing to the NexMon project[81] or similar works in order to inject custom TED code into closed binary firmware. Also the spreading of this kind of works may convince chip vendors to adopt a more open policy about their firmwares.

Conclusions

In this document we have seen the main aspects concerning real-time communications in the context of mobility. Despite facing it in our daily life, there is still a long way to go before we can say mobile communication fully meets our expectations of reliability and high service quality. In fact, mobile users still have to deal with many issues while on the move and using VoIP calls, videoconference applications, video-broadcasting services, etc. at the same time. In particular I focused on some issues which arise during the vertical handover process such as session discontinuity, host unreachability and communication unavailability.

In the first half of this document we have studied different solutions several researchers have put forward. However, only a few of them are deployable in today's network infrastructures which includes NATs and firewalls. Thus I strove to keep my efforts feasible on currently deployed infrastructure. I opted to contribute to the ABPS project, an architecture for enhancing host mobility and vertical handover which is also capable of dealing with NATs and firewalls. We have seen how the software I developed fulfills initial expectations, enhancing vertical handover process in a realistic scenario. In fact results shown in the previous chapter are promising: during the handover process, no communication interruption is perceived by applications while employing ABPS components with evident indications of performance improvements such as considerable packet loss reduction and better service availability.

It is clear the current implementation of this project is quite immature and results are still inaccurate. However my work confirms how the ABPS architecture is still innovative and promising and it encourages future development and investigation.

Appendix A

Testers and developers documentation

The following sections describe the steps and actions required for building all the ABPS software parts I have implemented. They form essentially an early "documentation for testers and developers" whose intent is to provide a guideline for deploying the same test scenarios I used in my analysis phase and for setting up an implementation starting point for future developers.

At the time of writing this documentation is maintained on my personal ABPS github repository: *https://github.com/matteomartelli/ABPS*. Often in the next sections it will be used the terms "this repository" or the "abps repo" in reference of such repository. Thus the first required step is to clone the repository since it contains all the software tools which this documentation refers to:

```
git clone https://github.com/matteomartelli/ABPS.git
```

A.1 TED kernel and proxy application

This section explains how to to build a custom kernel with the support for TED (Transmission Error Detector) and how to use it with the TED proxy application. Different build methods are needed whether you want to build

the TED kernel for a GNU Linux distribution or Android distribution, since some additional steps are required for the latter.

A.1.1 Build Linux kernel

Get kernel sources

Clone the linux kernel repository in your local machine.

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux
    .git linuxrepo
```

Then move to the version branch you want to build. TED patches are currently available for versions 4.1.0 and 3.4.0 but you can always move to a different version branch and manually adjust the TED patch. Let's assume you chose the 4.1.0 version.

```
cd linuxrepo
git checkout v4.1
```

In order to be sure your git head is at the desired version, just check the first 3 lines of the Makefile located in the repository root directory.

Patch the kernel

If you chose the version 3.4.0 of the kernel you should first apply an official patch needed by TED to work properly. This patch was introduced in later versions and allows TED to read some information about its socket at lower levels of the network stack.

From the root directory of the linux kernel repository:

```
patch -p1 < abps_repo/ted_proxy/kernel_patches/net-remove-
    skb_orphan_try.3.4.5.patch
```

Then apply the TED patch. For the 4.1.0 version only this step is needed to patch the kernel sources.

```
patch -p1 < abps_repo/ted_proxy/kernel_patches/ted_linux_(
    your_chosen_version).patch
```

Build kernel

Several well documented guides explaining how to build a custom kernel exists in the web[13][12].

As guide [13] explains, `make localmodconf` is recommended for faster compilation but be sure that all the modules you will need later are currently loaded. At the end, just run to start the build:

```
make -jN
```

(where N is the number of parallel compilation processes you want to spawn).

After your custom kernel is built just run with root privileges:

```
make modules_install
make install
```

or refer to your linux distribution kernel install method.

A.1.2 Android

The following steps are known to be working, at the time of writing, for the LG Nexus 5 with Android 6 Marshmallow.

Prerequisites

You will need some Android tools such as `adb` and `fastboot`. In debian like distributions and Arch linux distributions they should be available through the `android-tools` package. Also you need the "Android NDK toolset". This toolset will be used to compile the `tedproxy` application and can be downloaded from the Android developers reference website[3].

Enable root privileges

Since TED requires a device which supports mac80211, the inner WiFi module (Broadcom BCM4329) of the LG Nexus 5 can't be used. In fact the BCM4329 module works as a Full MAC device with a proprietary and closed

source firmware, without any support for the mac80211 subsystem. Anyway the TED kernel and its proxy application can be tested using an external USB WiFi dongle which supports the mac80211 driver interface. To do so, some Android system configuration files must be edited with root privileges. Also, the `tedproxy` application binds the sockets directly to the network interfaces through the socket option `SO_BINDTODEVICE` which requires root privileges. Root tools for Android, essentially allow users to execute the `su` binary file which is missing by default for security purposes.

The tool I used is Superuser[18]. It is open source and offers both the boot image that includes the su binary file and the permissions control application. The installation procedure for the Nexus 5 is quite simple since the superuser community already provides pre-built boot images at https://superuser.phh.me.

Once you have obtained the boot image for your device, first you have to unlock the bootloader, then simply flash the boot image with the fastboot tool (it may damage your device):

```
fastboot oem unlock
fastboot boot nameofrecovery.img
```

You can also use a custom recovery such as TWRP[19] which allows you to backup the original boot image first.

If a pre-built image for your device is not available you should execute the content of the superuser.zip installation file in a custom recovery such as TWRP since it can run scripts with root privileges. In brief, if you "flash" the zip file in the TWRP recovery, it extract the zip file and executes the extracted scripts. These scripts essentially copy the current boot partition in a boot image file, extract the boot image, extract the inner ramdisk image and copy the `su` binary file into the extracted ramdisk. Lastly a modified boot image will be re-created and actually flashed to the device.

Get tools and kernel sources

First take a look at the official android documentation[8] in order to understand which kernel version you need depending on your device and get the right tools. These are the steps needed to get the right tools and kernel version for a LG Nexus 5 device.

Get the pre-built toolchain (compiler, liker, etc..), move it wherever you like and export its path:

```
git clone https://android.googlesource.com/platform/prebuilts/gcc/
    linux-x86/arm/arm-eabi-4.6
sudo mv arm-eabi-4.6 /opt/
export PATH=/opt/arm-eabi-4.6/bin:\$PATH
```

You can also copy the last line inside your `.bashrc`.

Get the kernel sources:

```
git clone https://android.googlesource.com/kernel/msm
git checkout android-msm-hammerhead-3.4-marshmallow-mr2
```

A.1.3 Patch the kernel

From the root directory of the android kernel repository:

```
patch -p1 < abps_repo/ted_proxy/kernel_patches/net-remove-
    skb_orphan_try.3.4.5.patch
patch -p1 < abps_repo/ted_proxy/kernel_patches/ted_linux_3.4.patch
```

Configure and build

In order to test TED with the Nexus 5 you should add in your custom kernel the support for a USB Wi-Fi dongle that is mac80211 capable. This repo provide a custom configuration which add the support for the Atheros `ath9k_htc` and Ralink `rt2800`. You can of course make your own configuration to add the support for different devices.

First of all let's prepare the environment. From the root directory of the android kernel repository:

```
export ARCH=arm
export SUBARCH=arm
export CROSS_COMPILE=arm-eabi-
```

Then let's make the configuration. If you are fine with my custom configuration:

```
cp abps_repo/tedproxy/android_build/hammerhead_defconfig_ted .config
```

Otherwise if you want to make your own configuration:

```
cp arch/arm/configs/hammerhead_deconfig .config
make menuconfig
```

Then start the build:

```
make -jN
```

(where N is the number of parallel compilation processes you want to spawn).

Common issues

Compiling old kernels from a new linux distribution may require some workaround.

If you encounter an error on `scripts/gcc-wrapper.py`, it may be caused by the fact that your default python binary links to `python3` while that script wants `python2`.

A simple workaround consists in replacing the first line of `scripts/gcc-wrapper.py` with:

```
#! /usr/bin/env python2
```

Another error that you may encounter is

```
Can't use 'defined(@array) (Maybe you should just omit the defined()?)
     at kernel/timeconst.pl
```

To avoid this just remove all the `defined()` invocation, without removing the inner array variables, from `kernel/timeconst.pl` as the comment in the error suggests.

Once the build is finished, the kernel image is located at `arch/arm/boot/zImage-dtb`.

Enable the USB Wi-Fi Dongle

As many Android devices, the Nexus 5 boot process is handled by an init script contained in the ramdisk filesystem image, which is itself contained in the boot image together with the kernel image.

The init script is the one who starts the `wpa_supplicant` daemon with the default interface `wlan0`. I had to modify the init script inside the ramdisk image in order to start `wpa_supplicant` with the external usb dongle, which is `wlan1`. To do so, you'd need to extract the original boot image from your device, extract the ramdisk image, modify the `init.hammerhead.rc` file substituting all the `wlan0` with `wlan1`.

This repository already provides a modified ramdisk image, anyway these are the steps you'd need to follow in order to retrieve your original boot image from the Nexus 5:

Get the original boot.img

```
#use the following commands to find the boot partition
ls -l /dev/block/platform/
#now we know the device platform is msm_sdcc.1
ls -l /dev/block/platform/msm_sdcc.1/by-name
#now we know the boot partition is mmcblk0p19
#by [boot -> /dev/block/mmcblk0p19]
#use the following command to retrieve the boot.img
su
cat /dev/block/mmcblk0p19 > \
    /sdcard/boot-from-android-device.img
chmod 0666 /sdcard/boot-from-android-device.img
```

You can then copy the image to your machine with `adb pull` or the MTP protocol.

Extract the original boot.img Once you have your original boot image in your hand, you can proceed with the extraction.

I recommend to read this guide[7] and use this tool[6] for the boot image extraction. The latter is also mirrored in this repository also under `abps_repo/tedproxy/android_build/boot/boot-extract`.

```
./boot-extract boot.img
```

Store the output of the extraction, since it will be necessary later for the boot image re-creation. In my case this is the output:

```
Boot header
  flash page size       2048
  kernel size           0x86e968
  kernel load addr      0x8000
  ramdisk size          0x12dab8
  ramdisk load addr     0x2900000
  second size           0x0
  second load addr      0xf00000
  tags addr             0x2700000
  product name          ''
  kernel cmdline        'console=ttyHSL0,115200,n8 androidboot.
      hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_v2.
      enable=1'

zImage extracted
ramdisk offset 8845312 (0x86f800)
ramdisk.cpio.gz extracted
```

Extract and edit the original ramdisk Once you have your ramdisk image `ramdisk.cpio.gz` you can extract it with:

```
mkdir ramdisk_dir
cd ramdisk_dir
zcat ../ramdisk.cpio.gz | cpio -i
```

Finally you can edit the `init.hammerhead.rc` file substituting all the `wlan0` with `wlan1`.

Create the custom ramdisk After that, re-create the ramdisk filesystem image with the `mkbootfs` tool:

```
cd ..
abps_repo/tedproxy/android_build/boot/mkbootfs/mkbootfs ramdisk_dir >
    ramdisk.ted.cpio
gzip ramdisk.ted.cpio
```

The result is a modified ramdisk image: `ramdisk.ted.cpio.gz`. If this fails to boot you can try with the pre-made custom ramdisk available at `path_to_abps_repo/tedproxy/android_build/boot/ramdisk.ted.cpio.gz`.

Create the custom boot.img Now re-create the boot image from both the custom kernel and the custom ramdisk image:

```
abps_repo/tedproxy/android_build/boot/mkbootimg/mkbootimg --base 0 --
    pagesize 2048 --kernel_offset 0x00008000 \
--ramdisk_offset 0x02900000 --second_offset 0x00f00000 --tags_offset 0
    x02700000 \
--cmdline 'console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead
    user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1' \
--kernel path_to_kernel/zImage-dtb --ramdisk path_to_ramdisk/ramdisk.
    ted.cpio.gz -o boot.img
```

Note how the offsets correspond to the addresses printed out by the boot image extract script.

Enable wlan1 in system files Editing the init.rc file is not enough, as some other Android services still refer to the wlan0 device.

You need also to substitute `wlan0` with `wlan1` in `/system/build.prop` and `/system/etc/dhcpcd/dhcpcd.conf`. These files are persistent in the Android system partition thus you just need to edit them once.

Also you need to copy the firmware of your WiFi dongle in /system/etc/-firmware. You can copy directly from your working machine or from the official repository[15].

At the end you can boot the custom boot.img with fastboot:

```
adb reboot bootloader
sudo fastboot boot boot.img
```

Or if you are sure of what you are doing you can flash it:

```
sudo fastboot flash boot boot.img
```

Once the Nexus rebooted it can be useful to activate the remote adb access:

```
adb tcpip 5555
```

Then you can plug the external Wi-Fi dongle with an OTG cable and control the device with remote adb:

```
adb connect nexus_ip_address:5555
```

Open issue The sleep mode of the external Wi-Fi dongle is not handled properly. In fact turning off the LCD screen cause the Wi-Fi device to de-associate from the network. As a simple workaround you can use an Android App to force your screen active but consider that this approach will lead your device to rapidly exhaust its battery power.

A.2 Build and run `tedproxy`

A.2.1 Build

Before you can build the `tedproxy` application you must ensure you are running a TED custom kernel. Then some header files called "user api (or uapi)" must be modified. These headers simply contain declarations of kernel constants and macros that also the user applications may need to recall. If you want to build the `tedproxy` application for linux distributions:

```
su
cp /usr/include/linux/errqueue.h \
    /usr/include/linux/errqueue.h.bkp
cp /usr/include/linux/socket.h \
    /usr/include/linux/socket.h.bkp
cp abps_repo/tedproxy/uapi/errqueue.h \
    /usr/include/linux/errqueue.h
cp abps_repo/tedproxy/uapi/socket.h \
    /usr/include/linux/socket.h
```

Otherwise if you want to build the `tedproxy` application for Android:

```
cd path_to_ndk/platforms/android-21/arch-arm/usr/include/linux
cp errqueue.h errqueue.h.bkp ; cp socket.h socket.h.bkp
cp abps_repo/tedproxy/uapi/errqueue.h errqueue.h
cp abps_repo/tedproxy/uapi/socket.h socket.h
```

At the end you just need to run the `build.sh` script with `linux` or `android` as argument for building the respective versions of teproxy. The build.sh is at `path_to_abps_repo/tedproxy/tedproxy`.

The Android version binary file will be placed at `libs/jni/tedproxy` and you can copy it on your device with adb push.

A.2.2 Run

`tedproxy` is intended to work as a local proxy which listens UDP traffic from a local bound socket and forwards all the input packets to the remote host through one or multiple network interfaces binding a socket for each one. With the '-i' option you can specify the name of the chosen network interfaces. If one of them is a WiFi mac80211 capable device, you can put the 't:' prefix before the device name and tproxy will enable TED notification for that device. In this case, packets will be forwarded to the "TED interface" only as long as the number of packets received at the AP is sufficiently high. Whenever this condition is no longer satisfied, `tedproxy` also enables the other network interfaces for forwarding traffic.

For instance `tedproxy` can listen to UDP local port 5006 and forward everything to host 130.136.4.138 at UDP port 5001 through the wlan1 mac80211 device and the rmnet0 device which is usually the cellular network interface on Android smartphones:

```
tedproxy -b -i t:wlan1 -i rmnet0 5006 130.136.4.138 5001
```

A.3 Relay and CN tools

A.3.1 Relay

The relay activation is quite simple. From the proxy server that you elected as the RTP relay do the following:

```
cd abps_repo/udprelay
python3 udprelay.py 5001:5002
```

Start the `udprelay` process which listens on two UDP ports. Indeed you can specify any ports you like but keep in mind the first is where the MN attaches to and the second is where the fixed CN attaches to.

A.3.2 CN tools

CN tools is a set of software tools collected in `abps_repo/cntools` that are required to enable an RTP receiving end on the CN. First of all you need `ffmpeg` installed on the CN machine. At the time of writing `ffmpeg` is not present on debian distributions since it has been replaced with `avconv`. I did not investigate further but `avconv` did not work well in my test environment. Thus since my CN was running a debian "jessie" distribution I compiled `ffmpeg` from sources. Once you obtained `ffmpeg` working on your CN machine you can run the `launcher.sh` script:

```
cd abps_repo/cntools
./launcher.sh 130.136.4.138 5002 5006 outputvideo.ogg
```

where 130.136.4.138 is the remote host IP address, 5002 the remote UDP port and 5006 the local UDP port.

The script first launches the `dummystdserver` HTTP server which serves an SDP file at the address `http://127.0.0.1:8080/camera.sdp`. Then it launches `cnproxy` passing to it the host IP address and both ports as arguments and runs `ffmpeg` as last step. When launched `cnproxy` sends an initialization datagram to the remote host then starts forwarding remote host's responses to the local UDP port, 5006 in this case, which `ffmpeg` listens to.

Thus `ffmpeg` first gets the `camera.sdp` file from the `dummystdserver` then receives on a local UDP port, specified in the sdp file, datagrams forwarded by `cnproxy` and eventually writes the output video file. Be sure the remote UDP port corresponds to the one the destination is listening to and that the local UDP port corresponds to the one specified in the sdp file.

A.3.3 Put everything together

Now that you have everything installed and configured you can stream some UDP traffic from the multi-homed mobile device to the CN passing through the relay. For instance let us consider an RTP camera streamer installed on the mobile device that sends RTP packets to the CN. On the mobile device you must first run `tedproxy`:

```
tedproxy -b -i t:wlan1 -i rmnet0 5006 130.136.4.138 5001
```

then activate the relay application on the remote proxy:

```
python3 udprelay.py 5001:5002
```

Now start the CN tools with the `launcher.sh` script:

```
./launcher.sh 130.136.4.138 5002 5006 outputvideo.ogg
```

It will send an initialization packet to the relay and then it will wait for user confirmation before starting `ffmpeg`. Before confirming, start the stream from the mobile device. Be sure the RTP streamer application sends packets to 127.0.0.1 and UDP port 5006 which is the port `tedproxy` is listening to. I used a proprietary camera streamer application (https://play.google.com/store/apps/details?id=com.miv.rtpcamera). However any similar application should work well.

Bibliography

[1] About mac80211. http://web.archive.org/web/20160915033548/https://wireless.wiki.kernel.org/en/developers/documentation/mac80211.

[2] Android binder. http://web.archive.org/web/20160625113918/http://elinux.org/Android_Binder.

[3] Android ndk downloads page. https://developer.android.com/ndk/downloads/index.html.

[4] Bluetooth high speed. http://web.archive.org/web/20161026135102/https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy.

[5] Bluetooth low energy. http://web.archive.org/web/20161026135102/https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy.

[6] boot-extract. https://github.com/csimmonds/boot-extract.

[7] Booting android. http://www.slideshare.net/chrissimmonds/android-bootslides20.

[8] Building kernels. http://web.archive.org/web/20161021065948/http://source.android.com/source/building-kernels.html.

[9] Definition of connected car – what is the connected car? defined. http://web.archive.org/web/20160610023540/http://www.autoconnectedcar.com/definition-of-connected-car-what-is-the-connected-car-defined/.

[10] Empathy. https://wiki.gnome.org/Apps/Empathy.

[11] Jitsi - open source and video calls and chats. https://jitsi.org.

[12] *KernelBuild*. http://web.archive.org/web/20160825174928/https://kernelnewbies.org/KernelBuild.

[13] *Kernels/Traditional compilation*. http://web.archive.org/web/20161107160059/https://wiki.archlinux.org/index.php/Kernels/Traditional_compilation.

[14] Link sap. http://web.archive.org/web/20161101105514/http://atnog.github.io/ODTONE/documentation/odtone/app/link_sap_index.html.

[15] linux firmware. http://git.kernel.org/cgit/linux/kernel/git/firmware/linux-firmware.git.

[16] Linux kernel coding style. http://web.archive.org/web/20160818090632/https://www.kernel.org/doc/Documentation/CodingStyle.

[17] Odtone faq. http://web.archive.org/web/20161101110546/http://atnog.github.io/ODTONE//faq.html.

[18] Superuser. https://github.com/koush/Superuser.

[19] Twrp. https://twrp.me/about.

[20] Jingle media relaying, 2006. http://antecipate.blogspot.it/2006/10/jingle-media-relaying.html.

[21] Vanilla kernel, 2009. http://web.archive.org/web/20160103124026/https://wiki.debian.org/vanilla.

[22] Itu world radiocommunication seminar highlights future communication technologies, 2010. http://web.archive.org/web/20161028154219/http://www.itu.int/net/pressoffice/press_releases/2010/48.aspx.

[23] [rfc] net: add wireless tx status socket option, 2011. http://web.archive.org/web/20161116155805/http://www.spinics.net/lists/netdev/msg176403.html.

[24] Carriers to switch off gsm networks in 2017, 2015. http://web.archive.org/web/20150622055109/http://www.techwalls.com/carriers-switch-off-gsm-networks-2017.

[25] Google talk discontinued; users told to switch to hangouts app, 2015. http://web.archive.org/web/20151222130559/http://en.yibada.com/articles/13790/20150216/google-talk-being-discontinued-users-instructed-switch-chrome-app-hangouts.htm.

[26] Adroid platform architecture, 2016. https://web.archive.org/web/20161106210627/https://developer.android.com/guide/platform/index.html.

[27] Global smartphone shipments forecast 2010-2020, 2016. https://www.statista.com/statistics/263441/global-smartphone-shipments-forecast.

[28] Google+ hangouts app hands-on, 2016. https://www.engadget.com/2013/05/15/google-hangouts-app-hands-on.

[29] Meet google duo, a simple 1-to-1 video calling app for everyone, 2016. https://googleblog.blogspot.it/2016/08/meet-google-duo-simple-1-to-1-video.html.

[30] The pokémon go effect on the network, 2016. https://web.archive.org/web/20160818234741/http://www.networkworld.com/article/3095796/lan-wan/the-pokmon-go-effect-on-the-network.html.

[31] Smartphone os market share, 2016 q2, 2016. http://web.archive.org/web/20161026052346/http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[32] The state of lte, 2016. http://web.archive.org/web/20161027104614/http://opensignal.com/reports/2016/02/state-of-lte-q4-2015/.

[33] Worldwide smartphone forecast update, 2016–2020: September 2016, 2016. https://www.idc.com/getdoc.jsp?containerId=US41725515.

[34] J Arkko and I van Beijnum. Rfc 5534: Failure detection and locator pair exploration protocol for ipv6 multihoming, 2011.

[35] Salman A Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *arXiv preprint cs/0412017*, 2004.

[36] Paolo Bellavista, Antonio Corradi, and Luca Foschini. Ims-compliant management of vertical handoffs for mobile multimedia session continuity. *IEEE Communications Magazine*, 48(4):114–121, 2010.

[37] Davide Berardi. Hashtable implementation, 2016. http://web.archive.org/save/_embed/https://raw.githubusercontent.com/berdav/macro-hashtab/master/hashtable.h.

[38] Adam Bergkvist, D Burnett, and Cullen Jennings. A. narayanan," webrtc 1.0: Real-time communication between browsers. *World Wide Web Consortium WD WD-webrtc-20120821*, 2012.

[39] Matt Bishop and LT Heberlein. Attack class: Address spoofing. In *Proceedings of the Nineteenth National Information Systems Security Conference*, pages 371–377, 1996.

[40] Marco Bonola, Stefano Salsano, and Andrea Polidoro. Upmt: universal per-application mobility management using tunnels. In *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*, pages 1–8. IEEE, 2009.

[41] John Border, Markku Kojo, Jim Griner, Gabriel Montenegro, and Zach Shelby. Performance enhancing proxies intended to mitigate link-related degradations. Technical report, 2001.

[42] Albert Cabellos, Alberto Rodríguez Natal, Loránd Jakab, Vina Ermagan, Preethi Natarajan, and Fabio Maino. Lispmob: Mobile networking through lisp. *LISPmob white paper*.

[43] Carlo Caini, Haitham Cruickshank, Stephen Farrell, and Mario Marchese. Delay- and disruption-tolerant networking (dtn): an alternative solution for future satellite networking applications. *Proceedings of the IEEE*, 99(11):1980–1997, 2011.

[44] Thiago Camargo. Xep-0278: Jingle relay nodes. *XEP XEP-0278, June*, 2011.

[45] S Deering and R Hinden. Rfc 2460: internet protocol, version 6 (ipv6). *Proposed Standard) http://tools.ietf.org/pdf/rfc2460.pdf*, 1998.

[46] Stephen E Deering. Internet protocol, version 6 (ipv6) specification. 1998.

[47] Noor Mohammad's Faltoos. A survey report on "generations of networks: 1g, 2g, 3g, 4g, 5g, 2013. http://web.archive.org/web/20150415050515/http://www.slideshare.net/noorec786/generations-of-network-1-g-2g-3g-4g-5g.

[48] Dino Farinacci, Darrel Lewis, David Meyer, and Vince Fuller. The locator/id separation protocol (lisp). 2013.

[49] Paul Ferguson and Daniel Senie. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. Technical report, 1997.

[50] Stefano Ferretti, Vittorio Ghini, Moreno Marzolla, and Fabio Panzieri. Walking with the oracle: Efficient use of mobile networks through location-awareness. In *Wireless Days (WD), 2012 IFIP*, pages 1–6. IEEE, 2012.

[51] Stefano Ferretti, Vittorio Ghini, and Fabio Panzieri. A survey on handover management in mobility architectures. *Computer Networks*, 94:390–413, 2016.

[52] Vittorio Ghini and Gabriele Di Bernardo. Transmission error detector per wi-fi su kernel linux 4.0. 2015.

[53] Vittorio Ghini, Stefano Ferretti, and Fabio Panzieri. The "always best packet switching" architecture for sip-based mobile multimedia services. *Journal of Systems and Software*, 84(11):1827–1851, 2011.

[54] Vittorio Ghini and Alessandro Mengoli. Un approccio cross-layer all'affidabilita guidata dalle applicazioni. 2015.

[55] Vittorio Ghini and Luca Milioli. Studio ed implementazione di un algoritmo per la gestione dell'handover in ambiente android. 2014.

[56] Eugenio Giordano, Lara Codecà, Brian Geffon, Giulio Grassi, Giovanni Pau, and Mario Gerla. Movit: the mobile network virtualized testbed. In *Proceedings of the ninth ACM international workshop on Vehicular inter-networking, systems, and applications*, pages 3–12. ACM, 2012.

[57] Youn-Hen Han. Mipv4 & mipv6 - overview of ip mobility protocols. http://www.cs.unibo.it/~ghini/didattica/sistdistrib/MIPv4_MIPv6.pdf.

[58] Henrik Ingo. Session initiation protocol (sip) and other voice over ip (voip) protocols and applications. *Sesca Technologies, Finland*, 2011.

[59] Emil Ivov. Hangout-like video conferences with jitsi videobridge and xmpp.

[60] Nikita Jora. Mobile ip and comparison between mobile ipv4 and ipv6. *Journal of Network Communications and Emerging Technologies (JNCET) www. jncet. org*, 2(1), 2015.

[61] Justin Karneges, Peter Saint-Andre, Joe Hildebrand, Fabio Forno, Dave Cridland, and Matthew Wild. Stream management. 2015.

[62] Shigeru Kashihara, Kazuya Tsukamoto, and Yuji Oie. Service-oriented mobility management architecture for seamless handover in ubiquitous networks. *IEEE Wireless Communications*, 14(2):28–34, 2007.

[63] Meriem Kassar, Brigitte Kervella, and Guy Pujolle. An overview of vertical handover decision strategies in heterogeneous wireless networks. *Computer Communications*, 31(10):2607–2620, 2008.

[64] Dominik Klein, Matthias Hartmann, and Michael Menth. Nat traversal for lisp mobile node. In *Proceedings of the Re-Architecting the Internet Workshop*, page 8. ACM, 2010.

[65] Eddie Kohler, Mark Handley, and F Floyd. Rfc 4340: Datagram congestion control protocol (dccp). 2006.

[66] Mitsunobu Kunishi, Masahiro Ishiyama, Keisuke Uehara, Hiroshi Esaki, and Fumio Teraoka. Lin6: A new approach to mobility support in ipv6. In *Proceedings of the Third Inernational Sympsium on Wireless Personal Multimedia Communications*, volume 43, 2000.

[67] Scott Ludwig, Joe Beda, Peter Saint-Andre, Robert McQueen, Sean Egan, and Joe Hildebrand. Xep-0166: Jingle. *XMPP Standards Foundation*, 2009.

[68] David A Maltz and Pravin Bhagwat. Msocks: An architecture for transport layer mobility. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1037–1045. IEEE, 1998.

[69] Mirco Marchetti and Michele Colajanni. Adaptive traffic filtering for efficient and secure ip-mobility. In *Proceedings of the 4th ACM symposium on QoS and security for wireless and mobile networks*, pages 43–50. ACM, 2008.

[70] Bradley Mitchell. What is the range of a typical wi-fi network?, 2016. https://web.archive.org/web/20161026103757/https://www.lifewire.com/range-of-typical-wifi-network-816564.

[71] Sarmistha Mondal, Anindita Sinha, and Jayati Routh. A survey on evolution of wireless generations 0g to 7g. *International Journal of Advance Research in Science and Engineering-IJARSE*, 1(2):5–10.

[72] G Montenegro. Rfc 3024, reverse tunneling for mobile ip, revised, 2001.

[73] Linux Netfilter. Firewall, nat, and packet mangling for linux.

[74] Erik Nordmark and Marcelo Bagnulo. Shim6: Level 3 multihoming shim protocol for ipv6. Technical report, 2009.

[75] Afif Osseiran, Federico Boccardi, Volker Braun, Katsutoshi Kusume, Patrick Marsch, Michal Maternia, Olav Queseth, Malte Schellmann, Hans Schotten, Hidekazu Taoka, et al. Scenarios for 5g mobile and wireless communications: the vision of the metis project. *IEEE Communications Magazine*, 52(5):26–35, 2014.

[76] Ian Paterson, Dave Smith, Peter Saint-Andre, Jack Moffitt, Lance Stout, and Winfried Tilanus. Xep-0124: Bidirectional-streams over synchronous http (bosh). *Draft Standard. Accessed January*, 16, 2013.

[77] C Perkins et al. Rfc 5944 ip mobility support for ipv4, 2010.

[78] C Perkins, D Johnson, and J Arkko. Rfc 6275: mobility support in ipv6. *Internet Engineering Task Force (IETF)*, 2011.

[79] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, and Eve Schooler. Sip: session initiation protocol. Technical report, 2002.

[80] Peter Saint-Andre. Extensible messaging and presence protocol (xmpp): Core. 2011.

[81] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. Nexmon: A cookbook for firmware modifications on smartphones to enable monitor mode. *arXiv preprint arXiv:1601.07077*, 2015.

[82] Alex C Snoeren, Hari Balakrishnan, and M Frans Kaashoek. Reconsidering internet mobility. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 41–46. IEEE, 2001.

[83] L Stout, J Moffitt, and E Cestari. An extensible messaging and presence protocol (xmpp) subprotocol for websocket. Technical report, 2014.

[84] Kenichi Taniuchi, Yoshihiro Ohba, Victor Fajardo, Subir Das, Miriam Tauil, Yuu-Heng Cheng, Ashutosh Dutta, Donald Baker, Maya Yajnik, and David Famolari. Ieee 802.21: Media independent handover: Features, applicability, and realization. *IEEE Communications Magazine*, 47(1):112–120, 2009.

[85] R Wakikawa, T Ernst, K Nagami, and V Devarapalli. Draft-ietf-monami6-multiplecoa-07,". *Multiple Care-of Addresses Registration*, 2008.

[86] Wei Xing, Holger Karl, Adam Wolisz, and Harald Müller. M-sctp: Design and prototypical implementation of an end-to-end mobility concept. In *In Proc. 5th Intl. Workshop The Internet Challenge: Technology and Applications*, page 43, 2002.

[87] GUO Yangyong. An enterprise instant messaging software design and implementation. *Computer Programming Skills & Maintenance*, 24:036, 2010.

[88] Mariem Zekri, Badii Jouaber, and Djamal Zeghlache. A review on mobility management and vertical handover solutions over heterogeneous wireless networks. *Computer Communications*, 35(17):2055–2068, 2012.