Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Departamentul de Calculatoare



# LUCRARE DE DISERTAȚIE

# Economisire de Energie pentru Conectivitate Wireless Multipla

**Conducător Științific:**
Prof. Dr. Costin Raiciu

**Autor:**
Doru Cristian Gucea

București, 2017

University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department



# MASTER THESIS

# Energy Saving for Multiple Wireless Connectivity

**Scientific Adviser:**
Prof. Dr. Costin Raiciu

**Author:**
Doru Cristian Gucea

Bucharest, 2017

This work wouldn't have been possible without the help of my Intel Team. I especially like to thank my colleague, Andra Paraschiv, for her invaluable feedback and relentless hours of code debugging.
My team-lead, George Milescu, helped me to overcome the no-go situations with his deep knowledge and his brilliant ideas regarding the internal working of various software and hardware technologies.
Also, my manager, Bogdan Diaconescu inspired me with his passion for mobile technologies and discussions about the latest trends in this domain.

# Abstract

Multiple Wireless Connectivity is a technology that allows a mobile device to start and transmit data on multiple Wi-Fi connections by using multiple virtual interfaces mapped on the same physical Wi-Fi card. Nowadays, it is more and more common for a mobile device to use in parallel both a Wi-Fi Direct connection (aka P2P) and a regular Wi-Fi 802.11 connection. Also, research technologies like Multi-WiFi shows that using multiple 802.11 Wi-Fi connections in parallel improves the user-experience in terms of throughput and delay.

This master project is split in two directions. The first one is to identify and analyze the existing solutions for Multiple Wireless Connectivity on mobile devices with a focus on the power consumption perspective. The second direction is to design and implement an algorithm that reduces the energy consumption on mobile devices when multiple Wi-Fi connections are used in parallel. In this thesis I show that the existing solutions for Multiple Wireless Connectivity lack any optimizations for energy consumption. Starting from this observation, I show that my algorithm can drop the energy consumption by up to 50 percent while keeping the user-experience at an acceptable level.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# State of the Art for IEEE 802.11 Power Save

The power save algorithm described in this thesis is implemented starting from the classical power save algorithm implemented in the ath9k_htc driver with direct applicability to the AR9271 chip.

In this chapter I present the AR9271 chipset and the interaction between mac80211, ath9k_htc driver and AR9271 firmware with emphasis on the power save mechanism.

## 1.1 AR9271 Chip

As stated in the datasheet, the Atheros AR9271 is a highly integrated single-chip solution for 2.4 GHz 802.11n-ready wireless local area networks (WLANs) that enables a high-performance 1x1 configuration for wireless station applications demanding robust link quality and maximum throughput and range. The AR9271 integrates a multi-protocol MAC, baseband processor, analog-to-digital and digital-to-analog (ADC/DAC) converters, 1x1 radio transceiver, RF switch, and USB interface in an all-CMOS device for low power and small form factor applications.

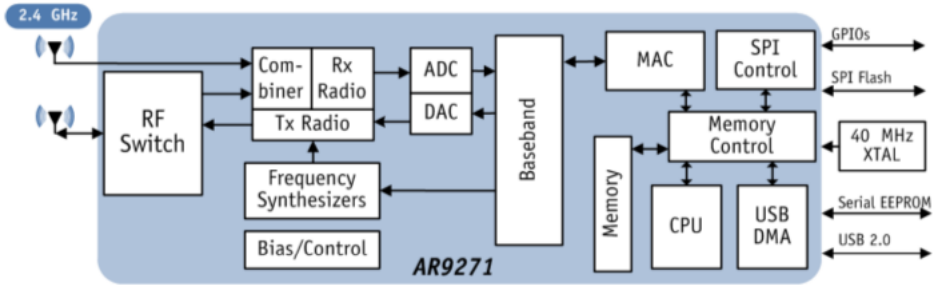


Figure 1.1: AR9271 system block diagram

The AR9271 implements half-duplex OFDM, CCK, and DSSS baseband processing, supporting 72.2 Mbps for 20 MHz and 150 Mbps for 40 MHz channel and IEEE 802.11b/g data rates. Other features include signal detection, automatic gain control, frequency offset estimation, symbol

timing, and channel estimation. The AR9271 MAC supports the 802.11 wireless MAC protocol, 802.11i security, receive and transmit filtering, error recovery, and quality of service (QoS).

The AR9271 supports one transmit traffic stream and one receive traffic stream using one integrated Tx chain and one receive chain for high throughput and range performance. The Tx chain combines baseband in-phase (I) and quadrature (Q) signals, converts them to the desired frequency, and drives the RF signal to the antenna. The frequency synthesizer supports frequencies defined by IEEE 802.11b/g/n specifications.

The AR9271 supports frame data transfer to and from the host using a USB interface that provides interrupt generation/reporting, power save, and status reporting. Other external interfaces include serial EEPROM and GPIOs. The AR9271 is interoperable with standard legacy 802.11b/g devices.

# Chapter 2

# Nexus 5 Analysis

One of the first Android phones used for studying the Multiple Wireless Connectivity was the Google Nexus 5 [18]. The main reason for this choice is the availability of both 2.4 Ghz and 5Ghz Wi-Fi frequencies which opens the door for the analysis of channel switching use-cases.

The analysis is focused on the parallel usage of both the regular Wi-Fi 802.11 connectivity and Wi-Fi Direct connectivity. The main discovery is that the Power Save algorithm is automatically disabled once a second virtual interface is created, in this particular case the interface for the Wi-Fi Direct connection.

What I tried to do was to replace this second interface with an interface for a secondary Wi-Fi 802.11 connection then implement my own Power-Save algorithm but this proved to be a show-stopper because the firmware for Nexus 5 is closed-source and it was impossible to make the stack work in parallel with two virtual interfaces in managed mode.

## 2.1 Testbed

Some modifications had to be done to the Wi-Fi Direct functionality so CyanogenMod 13 Hammerhead was chosen as operating system as the community was very active when the work for this project was started (2014). In the meantime, the original project died and was rebranded in Lineage OS [14].

As hardware I used an AC750 Wireless Dual Band Gigabit Router [17], a Monsoon Power Monitor device [11], a WiFi Spectrum analyzer and two Nexus 5 smartphones. One of the phones acts as P2P-GO and the other one as P2P client. In some tests, the P2P client is also connected to the Access Point. Also, in some tests, the Power Monitor Device measures the power consumption of the P2P client.

### 2.1.1 Capturing Air Packets

The first attempt to capture the packets for the Wi-Fi Direct connection was to use an Intel Dual Band Wireless-AC 7260 card [2] in monitor mode then to use Wireshark [19] for analysis. The problem was that only control frames like RTS/CTS and Block ACK were captured but no data packets.

However, the Wi-Fi Spectrum analyzer (see Figure 3.4) showed us that channel 1 was almost fully occupied so data frames were transmitted but our card in monitor mode couldn't decode those frames.

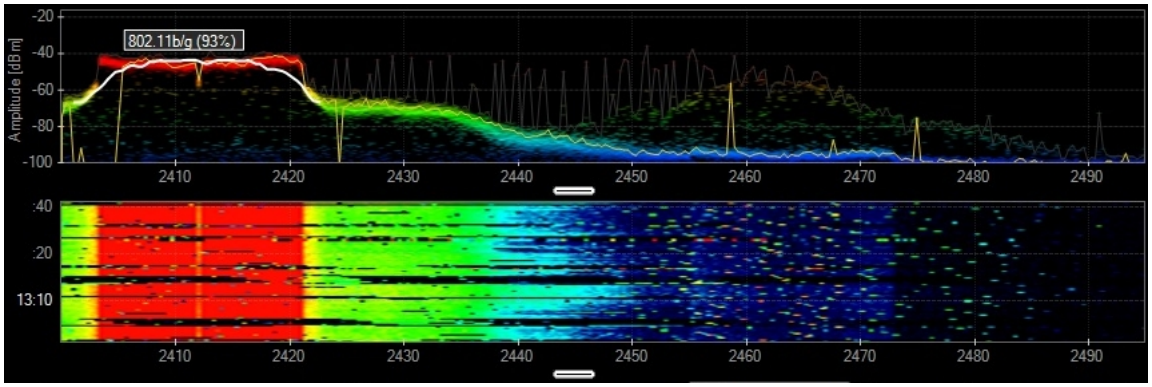


Figure 2.1: Spectrum analyzer capture

After starting a discussion [8] with the Intel Wi-Fi driver maintainer (iwlwifi), it proved that the problem was caused by the lack of low-density parity-check (LDPC) coding capability in hardware. Our card, Intel 7260 supports only Viterbi decoding. The next step was trying to install an Intel 7265 card which does support LDPC. The problem was that our laptop uses the older mini-PCIe connection while we needed an M.2 connection for the 7265 card.

Unable to install a newer card we tested an external card, a TP-Link TL-WN722N card [6] which proved to have LDPC capabilities. The problem with this card is the limitation to capture only 802.11n packets on the 2.4 frequency.

Wireshark was setup with the SSID and the password for the Wi-Fi Direct connection in order to decrypt the air packets. This parameters were taken from the P2P-GO and setup in Wireshark: Edit->Preferences->Protocols->IEEE 802.11->Decryption Keys->wpa pwd.

```
network={
        ssid="DIRECT-gQ-Android_cf8c"
        bssid=be:f5:ac:ff:e5:df
        psk="Lkb7rPop"
        proto=RSN
        key_mgmt=WPA-PSK
        pairwise=CCMP
        auth_alg=OPEN
        mode=3
        disabled=2
        p2p_client_list=66:89:9a:81:0d:95
}
```

Listing 2.1: Listing from /data/misc/wifi/p2p_supplicant.conf on the P2P GO

### 2.1.2 Setting up the Operating System

Default instructions [5] for compiling Cyanogen Mod 13 were used. However, there was a problem related to the extraction of proprietary blobs from the smartphone [7]. The script used for extracting the proprietary blobs is located inside *$CM_ROOT/device/lge/hammerhead/extract-files.sh* and its role is to execute a series of *adb pull* commands towards the phone. The script failed to download some files and the reported error was unexisting files. However, the real problem was that the *adb pull* command had insufficient permissions for accesing those file. The solution was to start *adb pull* with root permissions by running *adb root* before running the *extract-files.sh* script.

Team Win Recovery Project (TWRP) was used for installing CM13 on the Nexus5 smartphone. This is an open-source software that provides a touchscreen-enabled interface for allowing users to install/update third-party firmware [15]. This software allowed us to install SuperSU for gaining root access.

### 2.1.3 Wi-Fi Direct Connection Parameters

Our tests measure the power consumption for the P2P-Client device while sniffing packets from the air using a Wi-Fi card in monitor mode. So the Monsoon device has to stay connected to the P2P-Client while the Wi-Fi card is sniffing packets from the Wi-Fi Direct channel. The main problem was that the election of the P2P Group Owner/ P2P client and the selection of the best Wi-Fi channel was driven by the P2P Group Owner Negotiation process and there were cases when we measured the power consumption for the P2P GO and no data was captured by the monitor interface.

In order to solve this problem, a smarthone that has this patch [10] applied will become the P2P-GO and the communication channel frequency will be 2412 Mhz. For becoming the P2P-GO, the smartphone that has this patch applied will advertise a higher value for the intent and for the tie-breaker. Also, the advertised channel list will include a single supported channel.

### 2.1.4 Testbed for Power Measurement

The first step was to get access to the Nexus5 battery pins. This was pretty difficult because the battery is not replaceable and the access to internal components is difficult. Using a special smartphone dissasemble kit from IFixit [12] access to battery pins was obtained.

There are three pins for battery connection: two of them are for the regular + and - pins and the third one is connected to an internal thermistor, enabling the charger to avoid over-temperature problems. Figure 2.2 shows a black wire glued to the - pin, a yellow one to the + pin and the green one is for the thermistor.

Once we connected the Nexus 5 pins to the Power Monitor and tried to supply current to the smartphone we encountered an Over Current Error from the Power Tool software (the software that comes with the device). After trying different values for current and voltage with no success, the following work-around was found: enable the voltage out for the Monsoon device (with default values for voltage and current, 3.7V with 4.7A) but with the smartphone disconnected in the first phase. After Power Tool reports that the voltage was enabled with no error, the smartphone can be connected.

The problems seems to be caused by the Monsoon power up sequence [9]:

- Power up with no current limit for 20 miliseconds
- Run for 1 second with the current limit set to 500 mA
- Run continously with the current limit set to 4.6 A

## 2.2 Virtual Interfaces

The Wi-Fi chip used on Nexus 5 is Broadcom 4339 [3] [12]. This chip supports 20, 40 and 80Mhz wide channels up to 256QAM. The 802.11 supported technologies supported are a/b/g/n/ac and and it can reach single-stream spatial multiplexing up to 433.3 Mbps data rate.

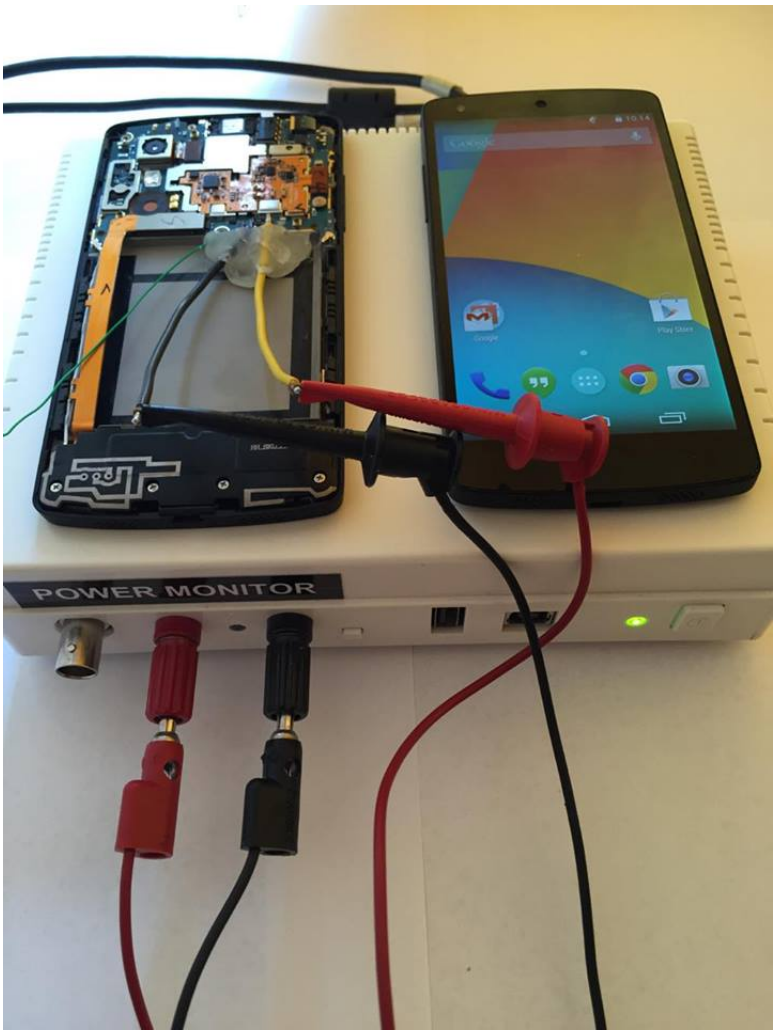


Figure 2.2: Monsoon Setup

The driver used for this chip is bcmdhd which is the Android version for the mainline Linux driver - brcmfmac [4]. Unfortunately, bcmdhd is a Full Mac Driver, which means that the mac80211 processing is done in firmware, on the Wi-Fi chip. The firmware is proprietary and closed-source so no modification can be done at that level.

The only modification that we need to do inside the driver was to create two virtual interfaces in the managed mode.

```
1 static const struct ieee80211_iface_combination
2 sta_p2p_iface_combinations[] = {
3         {
4         .num_different_channels = 2,
5         .max_interfaces = 3,
6         .limits = sta_p2p_limits,
7         .n_limits = ARRAY_SIZE(sta_p2p_limits),
8         },
9 };
```

Listing 2.2: Interface combinations for bcmdh driver

Looking in the existing code, we noticed that the sta_p2p_iface_combinations structure supports the creation of 3 interfaces which can operate on two different channels. But if we look deeper in the sta_p2p_limits field we can see that only two interfaces can be truly used in parallel: one in station mode and the other one in P2P-GO/P2P Client Mode. The third interface can be used only as a buffer interface for the cases when the P2P-GO is removed and its corresponding interface has to migrate temporary to a station interface. However, we tried to use this third interface to connect to an 802.11 Access Point but the connection fails due a Prefferred Network Offload error. In PNO, the firmware is configured with a number of SSIDs and it notifies the host when it finds one of those. For solving the problem it seems that we need to investigate the PNO firmware code.

## 2.3 Channel Switching Tests

For the first set of experiments we set the AP on the 5 Ghz frequency then we connected the first Nexus 5 acting as P2P GO to the AP using regular Wi-Fi. A Wi-Fi Direct connection is established between the Nexus 5 phones.

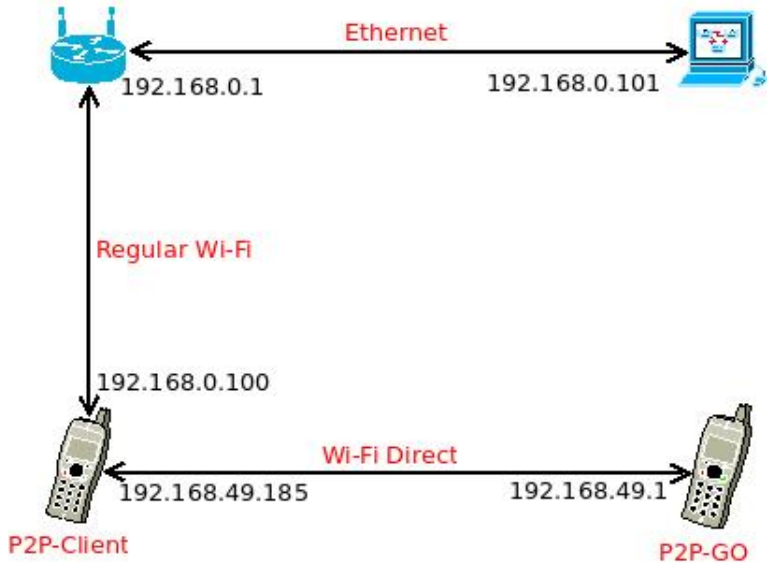


Figure 2.3: Topology

### 2.3.1 Channel-Switching Quantum

In order to determine the switching time quantum, ping packets were generated simultaneously on both paths, from 192.168.49.1 towards 192.168.49.184, and from 192.168.0.101 towards 192.168.49.

After analysing the RTT for both paths, a channel switching quantum of 60ms can be deduced. This quantum seems to be hard-coded as we tested with different ping intervals but the RTT distribution is the same (see Figure 2.4, Figure 2.5). Worse, even if we don't generate any traffic on one path (Regular Wi-Fi/Wi-Fi Direct), the switching time quantum is the same. This demonstrates that the channel-switching alghoritm runs with the same parameters in all situations and no algorithm for adjusting the channel switching quantum according to the traffic distribution is taken into consideration.

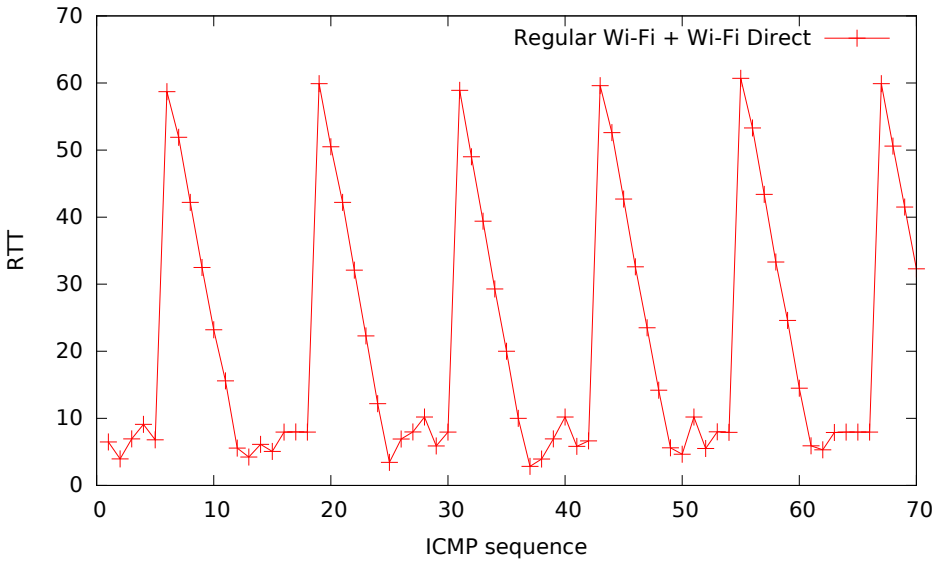


Figure 2.4: RTT for the Regular Wi-Fi Path

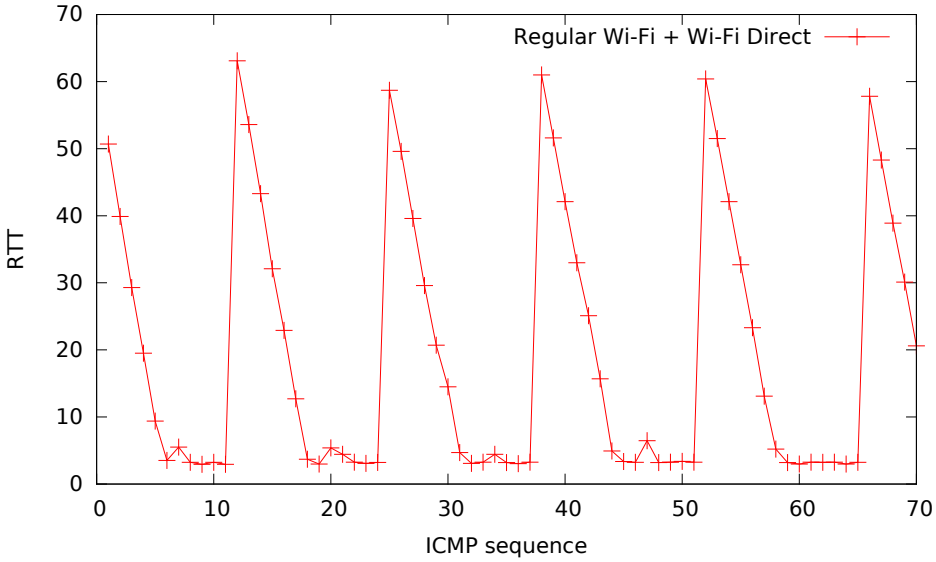


Figure 2.5: RTT for the Wi-Fi Direct Path

When the channel switching algorithm does not run, the average RTT value is very low, with an average RTT value at around 5 ms. For example, Figure 2.6 shows the RTT value for the case when we have just a regular Wi-Fi connection. The situation is the same for the Wi-Fi Direct only connection.

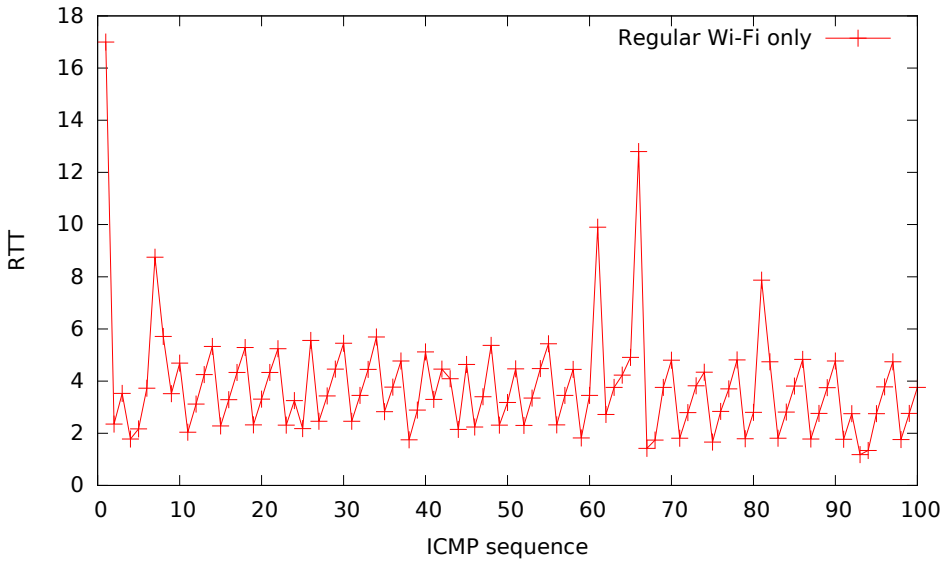


Figure 2.6: RTT for the Regular Wi-Fi Path

### 2.3.2 Channel-Switching Overhead

The Channel Switch Overhead is the percent of time needed by the hardware to switch between two frequencies and no data can be transmitted or received during this switch. The idea of this test is to determine the maximum capacity of both paths by generating UDP packets with iperf in a single wifi scenario and getting the throughput value reported by iperf. Then, send UDP packets on both paths in a multiple wireless connectivity scenario with iperf and again get the throughput value. In an ideal case, if there is no channel switch overhead, the ratio between the throughput in a single wifi scenario and the throughput in a multi wifi scenario should be 2 for each channel.

For the Regular Wi-Fi path, the channel capacity is about 230 Mbits/s. This value was obtained by generating UDP traffic from 192.168.0.101 towards 192.168.0.100. For the Wi-Fi Direct Path the channel capacity is about 55Mbits/s. This value was obtained by generated UDP traffic from the P2P-GO towards the P2P-Client.

The next step was to generate traffic on both paths in parallel. The iperf client used the maximum capacity of a path as parameter for bandwidth. The iperf server run on the P2P-client.

As can be seen in Figure 2.7, the P2P throughput value is pretty stable and is about 30 Mbits/s. What is strange is that the throughput value for Regular Wi-Fi drops from about 85 Mbits/s to about 5 Mbits/s and this indicates an implementation bug in the channel switch algorithm. In order to determine the channel switch overhead, we can assume that the throughput for Regular Wi-Fi can maintain its value at 85Mbits/s. So the overhead is (1 - ((85 + 30) / (230 + 55) / 2)) * 100, which means that about 19% of time is lost with the channel switching.

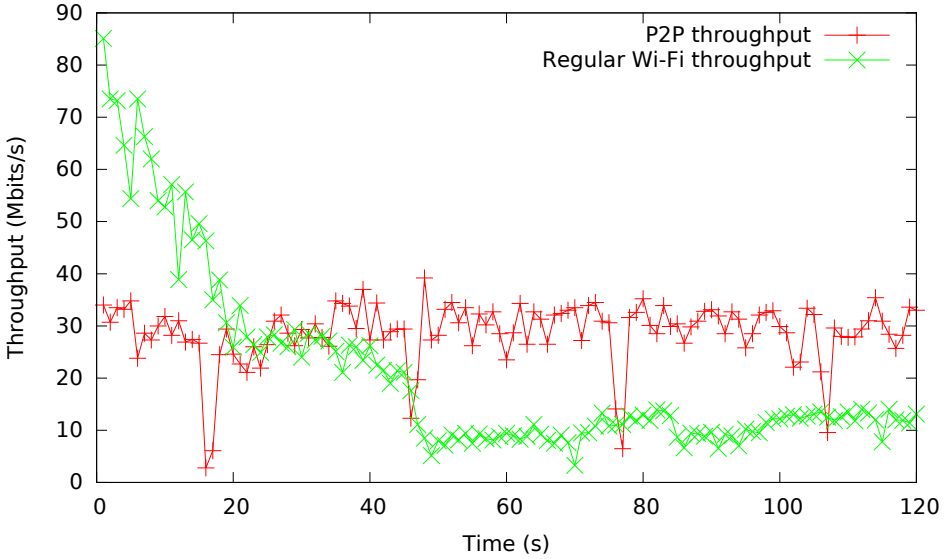


Figure 2.7: UDP traffic for both paths, in parallel

## 2.4 Average Power Tests - Channel Switching

First of all we stopped all the running applications that came by default with the Nexus 5 devices. Then we measured the average power for 120s and made sure that there are no power spikes generated by an unwanted application. The average power in this case was 9.37mW.

For the next experiments we wanted to measure the energy consumption for different type of downloads. The steps are the following:

- connect to the phone using the USB cable (adb shell).
- start an iperf instance: iperf -s -u -i 1 » results_file && nohup&. Using this command we'll open a socket listening for UDP connections and the throughput results will be written in results file. The nohup command allows the iperf instance to run after we disconnect the USB cable.
- disconnect the USB cable.
- on the laptop start a long lasting UDP connection: iperf -c PHONE_IP_ADDRESS -u -t 6000 -i 1
- after the client iperf connection started, push the "RUN" button in the Monsoon device that will start gather consumption statistics.
- after Monsoon gathered results for 120 seconds, press "STOP" button in the Monsoon Software and save the results: the consumption file resulted from Monsoon and the results file from the phone where we have throughput.

### 2.4.1 Phone connected only to the AP

The average power for 120s when the phone is connected just to the AP but no data is transferred is 60.57mW. This value should be lower but it seems that some IPV6 Router Advertisements messages introduce some spikes in the power consumption graph from Figure 2.8. The blue line represents the average power, the red line represents the minimum power and the green one represents the maximum power.

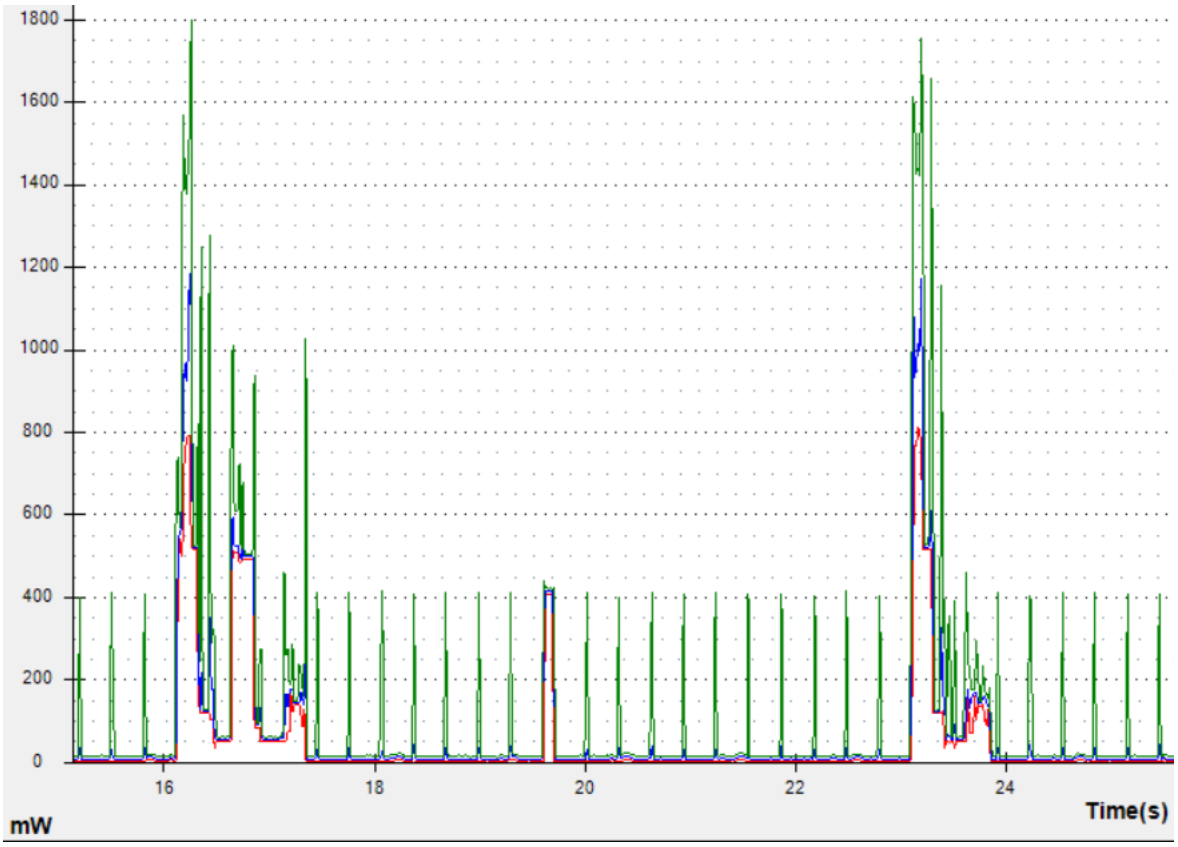


Figure 2.8: Power when the phone is connected just to the AP

### 2.4.2 Phone connected in parallel to the AP and to the P2P GO

In this case the average power consumption for 120s is 441.36mW. This value is correct as I repeated the experiments and the results are consistent.

### 2.4.3 Data transfer using the Regular Wi-Fi connection, P2P connection only active

In this experiment, the average power consumption for combined regular Wi-Fi + Wi-Fi Direct is compared with the average power consumption for regular Wi-Fi when we use the same throughput values. For this experiments, the Wi-Fi Direct connection is not used for data transfer, just an active connection between the P2P client and the P2P GO is kept alive.

Detailed steps for the experiment are illustrated next. For x in (2.5, 5, 7.5, 10 Mbits/s) do:

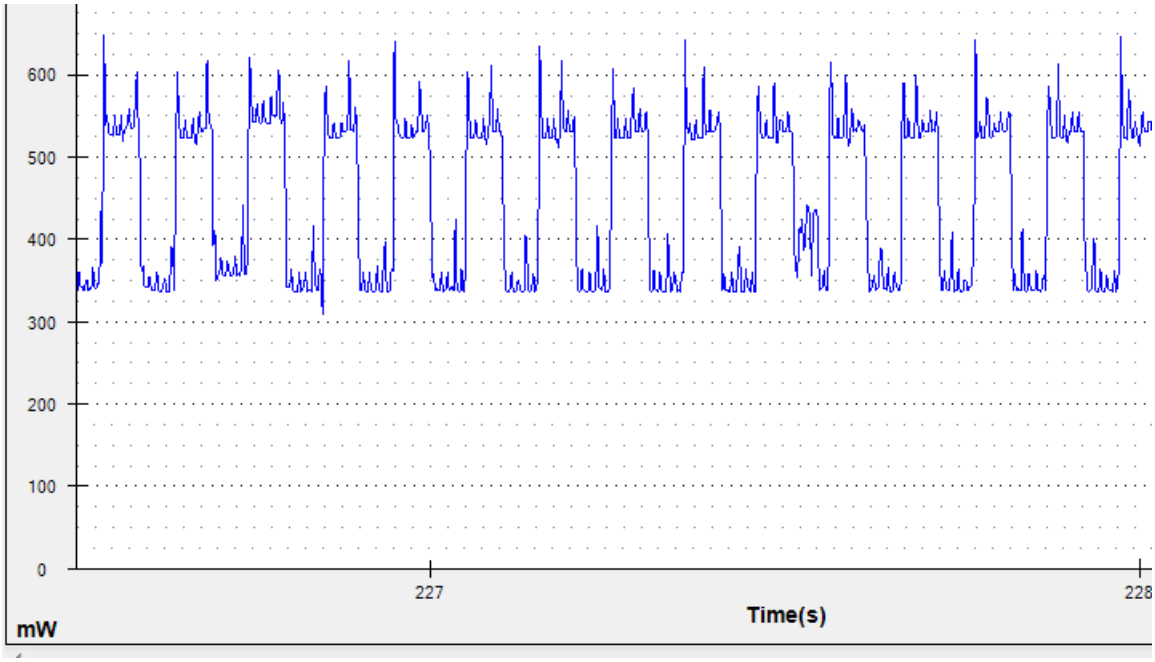


Figure 2.9: Power when the phone is connected in parallel to AP and to the P2P-GO

- connect the phone both to the AP and to the P2P GO
- on the phone start the iperf server to listen for new connections on the interface corresponding to the AP
- on the laptop generate traffic towards the client (iperf -c 192.168.0.100 -b x)
- calculate the average throughput value using the statistics file from the phone and save this average value (call it y)
- save the y value and the associated power average value

The results are:

Table 2.1: Power Monitor measurements (P2P + Regular Wi-Fi)

| Throughput (Mbits/s) | Average Power (mW) |
|---|---|
| 2.49 | 472.8 |
| 4.99 | 512.67 |
| 7.48 | 547.66 |
| 9.89 | 583.29 |

- connect the phone phone just to the AP
- on the phone start the iperf server
- on the laptop generate traffic towards the client (iperf -c IP_ADDR_PHONE -b y)
- On the client calculate the average throughput and save this average value

The results are:

It seems that by simply enabling the Wi-Fi Direct connection we get a power consumption improvement. We repeated the experiments several times and the results are consistent. This

Table 2.2: Power Monitor measurements (Regular Wi-Fi only)

| Throughput (Mbits/s) | Average Power (mW) |
|---|---|
| 2.49 | 573.29 |
| 4.99 | 634.14 |
| 7.48 | 671.45 |
| 9.89 | 752.45 |

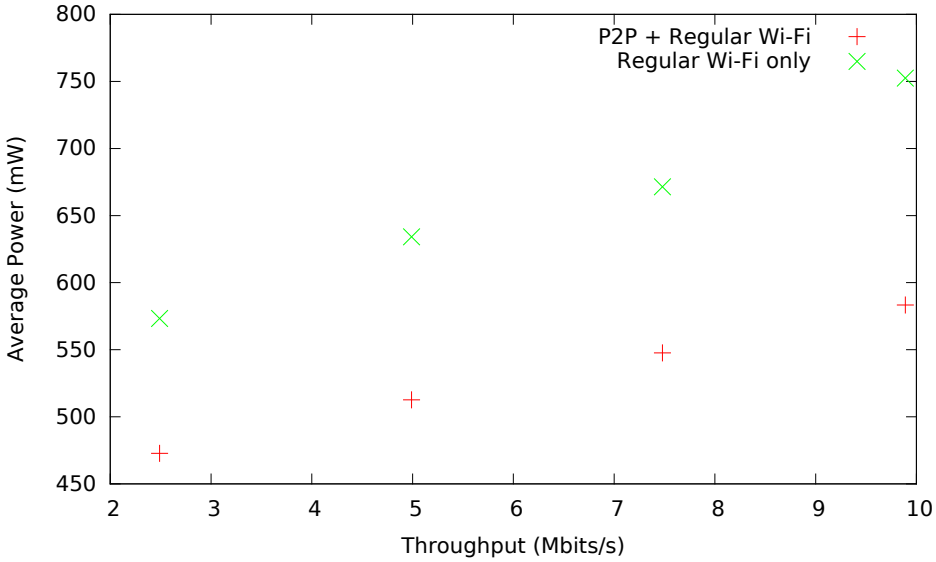


Figure 2.10: Average Power for 120 seconds

improvement could be explained by the batching mode of 802.11 where several packets are aggregated in a single, larger one. While the phone is on the Wi-Fi Direct channel, the packets are aggregated by the AP on the Regular Wi-Fi connection. When the client switches again on the Regular Wi-Fi connection, the larger packets are transmitted faster and more energy is saved.

### 2.4.4 Data transfer using both the Regular Wi-Fi connection and the P2P connection

In this experiment, the average power consumption for combined regular Wi-Fi + Wi-Fi Direct was compared with the average power consumption for regular Wi-Fi when the same throughput values are used. Both the regular Wi-Fi and the Wi-Fi Direct connections were used for data transfer.

- connect the phone both to the AP and to the P2P GO
- on the phone start 2 iperf server instances: one that listen for new connections on the interface corresponding to the AP and one that listen for new connection on the interface corresponding to the Wi-Fi Direct interface
- on the laptop generate traffic towards the client (iperf -c 192.168.0.100 -b x, iperf -c 192.168.49.1 -b x)
- calculate the combined average throughput value using the statistics file from the phone

and save this average value (call it y)

- save the y value and the associated power average value

The results are:

Table 2.3: Power Monitor measurements (Regular Wi-Fi + P2P parallel transfer)

| Throughput (Mbits/s) | Average Power (mW) |
|---|---|
| 4.99 | 472.8 |
| 9.96 | 575.67 |
| 14.5 | 685.19 |
| 19.05 | 742.89 |

- connect the phone phone just to the AP
- on the phone start the iperf server
- on the laptop generate traffic towards the client (iperf -c 192.168.0.100 -b y)
- on the client calculate the average throughput and save this average value

The results are:

Table 2.4: Power Monitor measurements (Regular Wi-Fi transfer only)

| Throughput (Mbits/s) | Average Power (mW) |
|---|---|
| 4.99 | 634.14 |
| 9.96 | 757.79 |
| 14.5 | 787.36 |
| 19.05 | 985.13 |

Also In this case the power average values are better in case of regular Wi-Fi combined with Wi-Fi Direct.

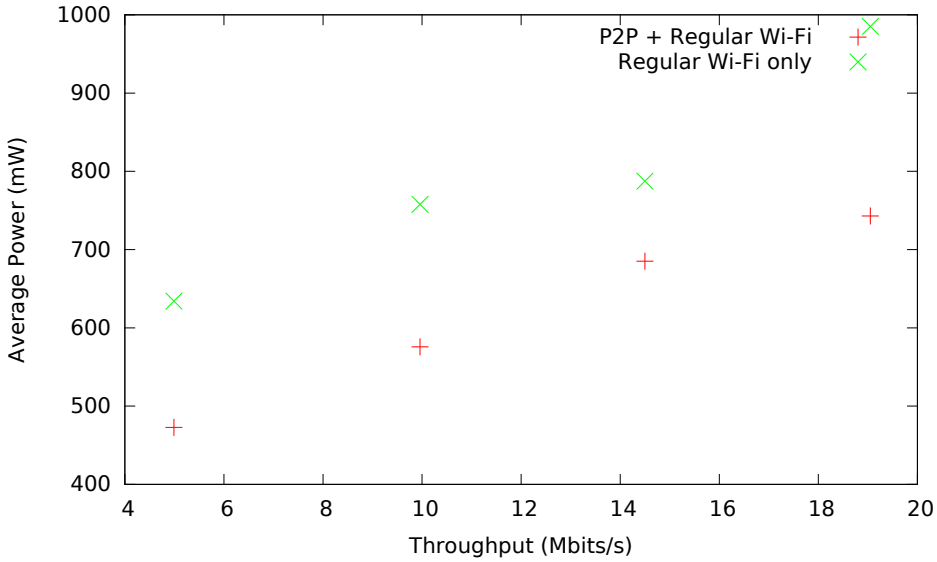


Figure 2.11: Average Power for 120 seconds

## 2.5 Average Power Tests - Single Channel

For this test, both the Regular Wi-Fi and the Wi-Fi Direct connections are using the same 2.4 channel and no data is transferred, only the management frames. The average power for 120s is around 210mW. In this case no energy spikes are observed which demonstrates that there is no power saving scheme implemented.

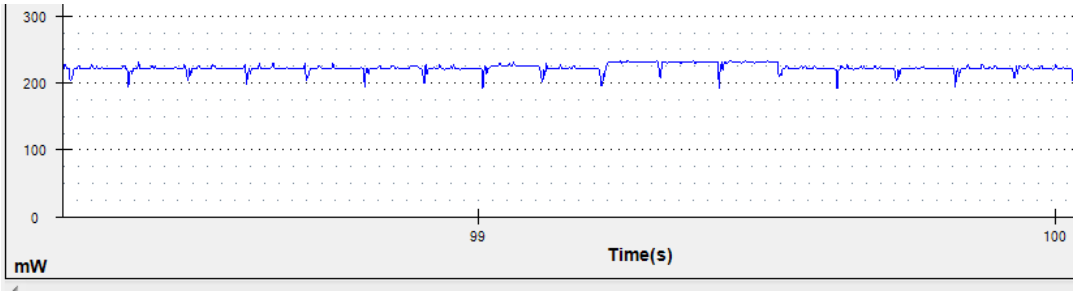


Figure 2.12: Power when the phone is connected in parallel to AP and to the P2P-GO

## 2.6 Power Management

For understanding the multiple Wi-Fi connectivity on Nexus 5, traffic was generated using the ping command. An interesting behaviour occured when ICMP packets were generated at intervals greater than 500ms: once the screen was turned off no packets were put into the air. After analysing the Broadcom driver code, we found the root of this behaviour inside *kernel/lge/hammerhead/drivers/net/wireless/bcmdhd/dhd.h*. The define *DHD_PACKET_TIMEOUT_MS* is set to 500 which mean that if the ping interval is lower than this timeout, the ICMP packets are sent even when the screen is turned off, otherwise the ping activity is suspended and resumed only after the screen is on again.

It proved that *DHD_PACKET_TIMEOUT_MS* configures the timeout for a wakeup event which is the basis for the Android Power Management model. For a better understanding of this model, the original discussion from the mailing list [13] is analysed.

The main task of the Android PM is first to verify that there are no pending wakeup events, then to freeze all user/kernel space processes and allow the device to go in suspend mode. Stated problems are related to the loss of wakeup events. First, if a wakeup event occurs exactly at the same time when /sys/power/state is being written to, the event may be delievered to user space right before the freezing of it in which case the space consumer of the veent may not be able to process it before the system is suspended. Second, if a wakeup event occurs after user space has been frozen and that event is not a wakeup interrupt, the kernel will not react to it and the system will be suspended.

The sys interface for PM management is:

- /sys/power/state - write "mem" to enter suspend mode
- /sys/power/wakeup_count - counter of wakeup events. This value will be read in the kernel code base and verified if equal or not to an interval counter. This check is done to know if the device can enter or not in suspend ode and write in /sys/power/state. May be read from or written to by user space. Reads will always succeed and return the current value of the wakeup events counter. Writes, however, will only succeed if the written number is equal to the current value of the wakeup events counter. If a write is successful, it will cause the kernel to save the current value of the wakeup events

> counter and to compare the saved number with the current value of the counter at certain points of the subsequent (or hibernate) sequence. If the two values don't match, the suspend will be aborted just as though a wakeup interrupt happened. Reading from /sys/power/wakeup_count again will turn that mechanism off;

The Android PM is a user-space process that that will first read from /sys/power/wakeup_count. Then it will check all user space consumers of wakeup events known to it for unprocessed events. If there are any, it will wait for then to be processed and repeat. In turn, if there are not any, it will try to write to /sys/power/wakeup_count and if the write is successful, it will write to /sys/power/state to start suspend, so if any wakeup events occur past that point, they will be noticed by the kernel and will eventually cause the suspend to be aborted.

Drivers and kernel subsystems can signal wakeup events. If the event is not explicitly handed over to user space and "instantaneous", they can simply call *pm_wakeup_event()* and be done with it. Second, if the event is going to be delivered to user space, the subsystem that processes the event can call pm_wakeup_begin() right when the event is detected and pm_wakeup_end() when it's been handed over to user space. *pm_get_wakeup_count()* and *pm_save_wakeup_count()* fail if they are called when events_in_progress is nonzero. For pm_save_wakeup_count() that's pretty obvious (I think) and it also kind of makes sense for pm_get_wakeup_count(), because that will tell the reader of /sys/power/wakeup_count that the value is going to change immediately so it should really try again.";

One possible problem that can still appear in PM is that processes can be frozen before an event is handled by the kernel. For example and interrupt handler might receive the event and start processing it by calling *pm_request_resume* - but if the PM workqueue thread is already frozen then the processing won't finish until something else wakes the system up.

The codebase for this analysis is represented by:

- kernel/lge/hammerhead/drivers/net/wireless/bcmdhd
- kernel/lge/hammerhead/drivers/base/power
- kernel/lge/hammerhead/kernel/power
- system/core/libsuspend
- sysfs

## 2.7 Conclusions

The implementation for multiple Wireless Connectivity on Android has multiple problems. Both for Single and Multiple Channel Connectivity, there is no power save algorithm implemented and the Wi-Fi card enters a high energy-mode once the second connection is established. Also, the channel swirching algorithm is pretty rudimentary because the channel switching quantum is the same with no load-balancing depending on the traffic pattern.

Unfortunately, no improvements could be done to this algorithm as the code for it is in the Broadcom closed-source firmware. The next step was to find a smartphone whose firmware code for Wi-Fi is open-source but it seems that all implementations are proprietary solutions.

# Chapter 3

# Power-Save Algorithm for Multiple Wireless Connectivity

As shown in the previous chapter, the first step for implementing a power save algorithm for multiple wireless connectivity is an open-source firmware. In most of the cases this also implies an open-source hardware described by a datasheet. This is due to the fact that the firmware code works directly with the Wi-Fi card registers.

Currently, there is no smartphone whose firmware is open-source for the Wi-Fi card so the work-around was to use an Wi-Fi dongle, a TP-Link TL-WN722N card [6] who is totally open-source. This chapter starts with the challenges in using this dongle with Nexus 5, it continues with an analysis of the open-source firwmare and it presents our algorithm for Power Save implemented on top of this platform.

## 3.1 Using the Wi-Fi dongle with Nexus 5

The micro-USB port of the smartphone was used for connecting the Wi-Fi dongle, as can be observed in Figure 3.1. The driver used by the dongle is ath9k_htc so we had to enable loadable module support plus ath9k_htc in the kernel menuconfig. However there was an 'Exec format' error while trying to insert the ath9k_htc.ko module using the insmod shell command. The insmod command calls in the bacground the init_module system call:

```
1 int init_module(void *module_image, unsigned long len, const char *
    param_values);
```

Listing 3.1: init_module system call

The role of the init_module is to load an ELF image into kernel space, perform the necessary relocations and initializations then run the module's init function. This functions receives third parameters: the module_image argument points to a buffer containing the binary image to be loaded; len specifies the size of that buffer and the param_values is a sting containing space-delimited specifications of the values for module parameters. In our case, there was a problem in the busybox implementation for insmod: the second argument was 8 bytes long instead of 4. When the init_modules was looking for the parameters passed on the stack and was trying to parse the third parameter it would actually look in the last 4 bytes of the second parameter, which were all zeros in most of the cases. So the third parameter was NULL in most of the cases. Later, in kernel space when SYSCALL_DEFINE3 is caled with a NULL parameter, the -EFAULT error is set. The solution was to use a 4 byte long parameter for the len argument.


Figure 3.1: Wi-Fi dongle connected to Nexus 5

### 3.1.1 Measuring the energy consumption for the Wi-Fi Dongle

The chipset mounted inside the TP-Link dongle is AR9271. Both the firmware [16] and the datasheet are open-source. In order to avoid the noise introduced by the phone components and by the USB layer, our first approach was to measure the energy consumption of the AR9271 chipset only. If we could find an entry point for the current in the AR9271 chip then we could easily use the AUX port of the power monitor device. This port allows for simultaneous measurement of the current going through an external power supply by using a sense resistor with a very low resistance (0.1 ohm) and a bayonet Neill-Concelman (BNC) connector.

AR9271 has 68 pins and the datasheet states that pins 17, 26, 33,47 represents digital 3.3 V power supply - VDDP33. Having multiple 3.3V power supply pins is common practice because it allows the PCB designer to create better PCBs in terms of RF immunity, RF emissions and signal crosstalk.

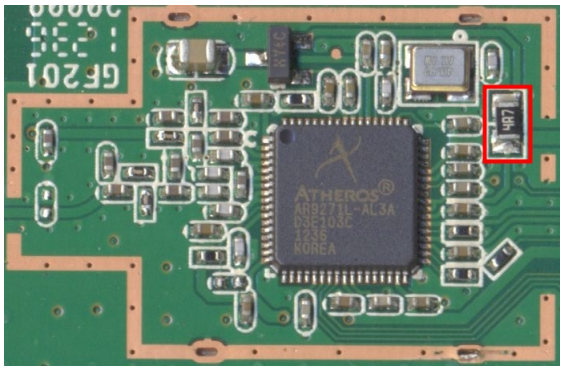


Figure 3.2: A9271 subsystem from TP-Link WN7222N

However, the power monitor device has a single grabber for IN current so a common entry point for these multiple 3.3V pins was was needed. Although the datasheet for the AR9271 chip is

open-source, the datasheet for the entire TP-Link Dongle is closed-source. Figure 3.2 shows the circuit around the AR9271 chip and it seems that the solution to the above problem would be to intercept the 3.3V rail after the step down converter, the trace right after 4R7, highlighted with red. Probably, the power consumption parameters from section 6.9 o f the datasheet were gathered using this technique. More details about this discussion can be found at [1]

Because the above assumptions were not backed-up by a datasheet we decided that is safer to measure the power consumption of the entire USB dongle. This was achieved by using the USB channel of the power monitor that offers the possibility to intercept the connection between the Wi-Fi Dongle and a USB port. More exactly, on the Power Monitor device are two ports: an USB type A port and a USB type B port. The Wi-Fi dongle was plugged-in the USB type A port, while the the laptop is connected to the USB type B port. The Monsoon Software offers the possibility to monitor this USB channel.

To download code or data when testing a device, USB can be used to connect the device to the Mobile Device Power Monitor. However, when connecting to USB, USB charges the device, which disturbs the current measurements. To remedy this, the Mobile Device Power Monitor has an Auto USB passthrough mode. Auto USB passthrough mode is useful for testing, because in Auto USB passthrough mode, the USB pass-through is disconnected whenever sampling starts. After sampling has completed, USB is reconnected automatically so that test data can be loaded to the device. The Auto USB passthrough mode setting is shown in below in Figure 3.3.

This technique proved to be useful because the graphic for the avergage power when no data is transmitted/received is constant and we were able to see the traffic spikes generated when by the AR9271 enters/exit sleep mode.

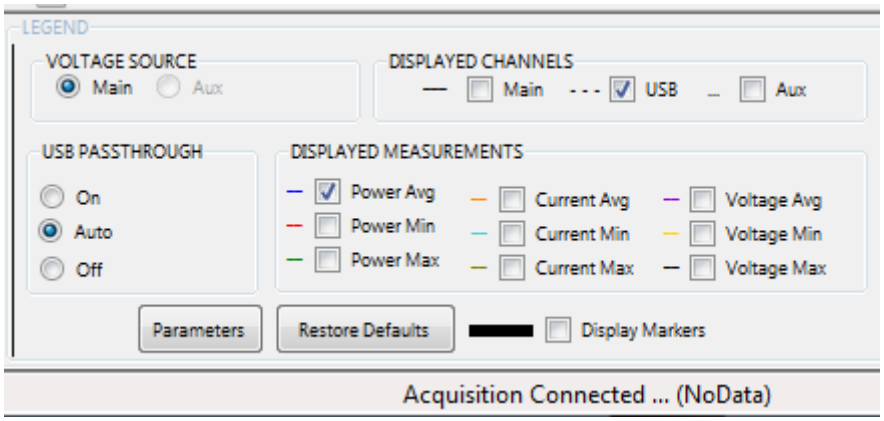


Figure 3.3: Options for USB channel monitoring

### 3.1.2 Firmware Debugging for the Wi-Fi Dongle

The AR9271 is a USB/Wifi SoC with onboad RAM, ROM, flash and the actual wireless chip. The wireless core is an off-shoot of the AR9285, a single-chip solution. For firmware debugging, the AR9271 chip has 16 GPIO pins and some of them can be configured to work in UART mode. Gathering data from these pins is done using a TTL UART adapter. For example, the operating system running on the SoC offers the possibility to print the value of variables.

Our first approach was to solder the TX/RX wires of the TTL UART directly to the GPIO pins of the AR9271 chip from the TP-Link Dongle. Unfortuntely, the distance between two nearby pins is very small and we didn't have specialized soldering equipment. The solution was to use another dongle, a ALFA Network AWUS036NHA which uses the same AR9271 chip but has accesible UART lines on the PCB.

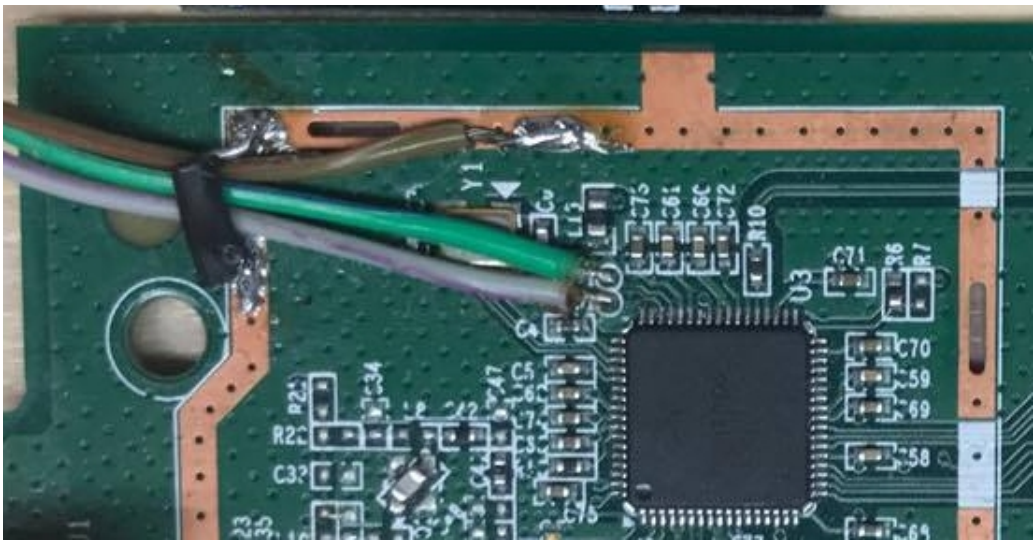


Figure 3.4: TODO

# Appendix A

# Project Build System Makefiles

## A.1 Makefile.test

```
1  # Makefile containing targets specific to testing
2
3  TEST_CASE_SPEC_FILE=full_test_spec.odt
4  API_COVERAGE_FILE=api_coverage.csv
5  REQUIREMENTS_COVERAGE_FILE=requirements_coverage.csv
6  TEST_REPORT_FILE=test_report.odt
7
8
9  # Test Case Specification targets
10
11 .PHONY: full_spec
12 full_spec: $(TEST_CASE_SPEC_FILE)
13         @echo
14         @echo "Generated␣full␣Test␣Case␣Specification␣into␣\"$^\""
15         @echo "Please␣remove␣manually␣the␣generated␣file."
16
17 .PHONY: $(TEST_CASE_SPEC_FILE)
18 $(TEST_CASE_SPEC_FILE):
19         $(TEST_ROOT)/common/tools/generate_all_spec.py --format=odt
               -o $@ $(TEST_ROOT)/functional-tests $(TEST_ROOT)/
               performance-tests $(TEST_ROOT)/robustness-tests
20 #
21
22 # ...
```

Listing A.1: Testing Targets Makefile (Makefile.test)

# Bibliography


[1] Discussion about AR9271 power pins. https://github.com/qca/open-ath9k-htc-firmware/issues/108, Accesed August 2017.

[2] Intel Dual Band Wireless-AC 7260 card. https://www.intel.com/content/www/us/en/products/wireless/wireless-products/dual-band-wireless-ac-7260.html, Accesed August 2017.

[3] BCM 4339 Wi-Fi chip. http://www.mouser.com/ds/2/100/Radio%20with%20Integrated%20Bluetooth%204.1%20and%20FM%20Receive-961626.pdf, Accesed July 2017.

[4] Source code for brcmfmac. https://github.com/torvalds/linux/tree/master/drivers/net/wireless/broadcom/brcm80211/brcmfmac, Accesed July 2017.

[5] Cyanogen Mod 13 compilation for Nexus 5 Smartphone. https://zifnab.net/~zifnab/wiki_dump/Build_for_hammerhead.html, Accesed August 2017.

[6] TP-LinK TL-WN722N WiFi dongle. http://www.tp-link.com/lk/download/TL-WN722N.html, Accesed August 2017.

[7] Proprietary Blobs Explanation. https://wiki.lineageos.org/proprietary_blobs.html, Accesed August 2017.

[8] Monitor Mode for Intel 7260 card. http://www.spinics.net/lists/linux-wireless/msg149435.html, Accesed August 2017.

[9] Reference Manual for Monsoon Power Monitor. https://www.msoon.com/LabEquipment/PowerMonitor/downloads/PowerMonitor_ManualVer1.3.pdf, Accesed July 2017.

[10] Wi-Fi Direct Patch for P2P-GO Election. https://drive.google.com/open?id=0B5SBH08PU_ChOFU0SWRQOHVGYnc, Accesed August 2017.

[11] Monsoon Power Monitor. https://www.msoon.com/LabEquipment/PowerMonitor/, Accesed July 2017.

[12] Teardown of Google Nexus 5. https://www.ifixit.com/Teardown/Nexus+5+Teardown/19016#s53729, Accesed July 2017.

[13] Power Management on Android Mailing List Discussion. http://lkml.iu.edu/hypermail/linux/kernel/1006.2/01499.html, Accesed August 2017.

[14] Lineage OS. https://lineageos.org/, Accesed July 2017.

[15] Team Win Recovery Project. https://en.wikipedia.org/wiki/TWRP, Accesed August 2017.

[16] Github repository for AR9271 firmware. https://github.com/qca/open-ath9k-htc-firmware, Accesed August 2017.

[17] TP-Link Archer C2 Router. http://www.tp-link.com.au/products/details/cat-9_Archer-C2.html, Accesed July 2017.

[18] Google Nexus 5 Smartphone. http://www.gsmarena.com/lg_nexus_5-5705.php, Accesed July 2017.

[19] Wireshark software for packet analysis. https://www.wireshark.org/, Accesed August 2017.