

POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria



ANALISI DEI LIMITI DELL'IMPLEMENTAZIONE
DI WI-FI DIRECT IN ANDROID
PER RETI OPPORTUNISTICHE

Relatore: Prof. Luciano BARESI

Tesi di laurea di:
Stefano CAPPA Matr. 796552

Anno Accademico 2014–2015

Ringraziamenti

Come prima cosa voglio ringraziare la mia famiglia per l'investimento fatto, la fiducia e la pazienza dimostrata.

Ovviamente, ringrazio il Professor Luciano Baresi per l'opportunità data e per l'infinita pazienza nel leggere le mie email lunghe quanto La Divina Commedia ed anche il Professor Sam Guinea per avermi fornito sempre gli Smartphone Android, senza i quali questo lavoro di tesi sarebbe stato impossibile.

Quindi, mi sembra doveroso ringraziare anche il Professor Antonio Capone e i suoi dottorandi per un rapido scambio di email che ha permesso di superare un piccolo problema, il Professor Marco Santambrogio e tutto il gruppo “morphone” per aver seguito la mia presentazione infinita e per i consigli forniti, nonché per l'ospitalità nel loro laboratorio NECSTLab.

Infine, ringrazio anche Fabio Malabocchia di Telecom Italia, per l'interessante chiacchierata al telefono per discutere di Wi-Fi Direct.

A questo punto, un breve ringraziamento a tutti gli amici, compagni di classe e non, che hanno sopportato i miei deliri per tutti questi anni e per la compagnia, soprattutto nella Biblioteca di Architettura, la mia seconda casa per ben due anni.

Semplicemente, grazie.

Stefano

Sommario

I dispositivi mobili come Smartphone e Tablet hanno rivoluzionato il concetto di mobilità, dando vita a nuovi scenari di utilizzo. Uno tra i più interessanti ed ampi è quello delle Opportunistic Networks, cioè reti che non richiedono un'infrastruttura preesistente, ma permettono la nascita di comunicazioni spontanee e caratterizzate da una forte dinamicità. Il problema principale è che i protocolli di comunicazione disponibili sono troppo giovani ed ancora da perfezionare in tale ambito.

Questo lavoro di tesi tratta nello specifico uno di essi, cioè Wi-Fi Direct, che non è stato pensato per scenari di questo tipo e soprattutto, la corrispondente implementazione in Android presenta grosse limitazioni. L'obiettivo è quello di studiare tali limiti nel dettaglio e proporre una soluzione per estendere il protocollo. L'approccio adottato consiste nell'analisi dell'architettura di Android, dall'alto livello, fino a quello più vicino all'hardware, iniziando dalla presentazione della soluzione progettata.

Nel corso di questo lavoro di tesi sarà mostrata, inoltre, la progettazione di due app Android con lo scopo di verificare i limiti del protocollo Wi-Fi Direct in Android e proporre uno scenario specifico di utilizzo in cui essi non sono significativi, creando un'app utilizzabile a tutti gli effetti.

Indice

Introduzione	1
1 Stato dell'arte	7
1.1 Wi-Fi	7
1.1.1 Wireless Mesh Network	8
1.2 Wi-Fi Direct	9
1.3 Scenari di utilizzo	10
1.3.1 Condivisione video in streaming	11
1.3.2 Riduzione del traffico sulla rete mobile	13
1.3.3 Diffusione d'informazioni impedendo la censura	14
1.3.4 Utilizzo in caso di emergenze e disastri	15
1.3.5 Evitare incidenti tra veicoli	16
2 Realizzazione di due app in SDK	19
2.1 PingPong	19
2.2 Pigeon Messenger	25
3 Esplorazione del codice sorgente	31
3.1 Visibilità dei metodi Java in Android	32
3.2 Estendere il package android.net.wifi.p2p	34
3.3 Modificare il package android.net.p2p	34
3.4 Utilizzare NDK	39
3.4.1 Esempio di app NDK per wpa_supplicant	41
4 Architettura, kernel e Build System di Android	47
4.1 L'architettura di Android	48
4.2 Il processo di boot in Android	49
4.3 Requisiti	55
4.4 Inizializzare l'ambiente di sviluppo	57
4.4.1 Installare Java, Android NDK e le dipendenze	58
4.4.2 Configurare l'accesso USB in Ubuntu	59
4.4.3 Variabili d'ambiente e CChace	59

4.5	Scaricare il codice sorgente di Android	60
4.6	L'Android Build System e la compilazione di Android	62
4.6.1	Cross-compilare Android per ARM	65
4.7	Compilare il kernel e creare zip installabili	66
4.8	Inserire il kernel in uno zip installabile	68
4.9	Configurare il kernel di Android	74
5	Wpa_supplicant	77
5.1	Interfacce di rete virtuali	77
5.2	Wpa_supplicant	80
5.3	Compilare wpa_supplicant e wpa_cli	83
5.4	Eseguire wpa_supplicant e wpa_cli	83
5.4.1	Inizializzazione	83
5.4.2	Configurazione di wpa_supplicant	86
5.4.3	Avvio di wpa_supplicant e wpa_cli	87
5.5	Utilizzare wpa_supplicant e wpa_cli	89
5.6	Modificare il kernel per creare le VIF	93
5.7	Verificare le Valid Interface Combinations con iw di Linux	99
6	Driver Wi-Fi in Android	105
6.1	I Driver Wi-Fi in Linux	105
6.1.1	Il passato: WE	105
6.1.2	Il presente: nl80211, cfg8011 e mac80211	106
6.2	Visualizzare gli errori di wpa_supplicant	109
6.3	Studio dei driver Broadcom	115
Conclusioni		121
Bibliografia		130
A File 51-android.rules per Ubuntu 14.04		131

Elenco delle figure

1.1	Tipi di reti Wi-Fi (BSS, ESS, IBSS)	8
1.2	Mesh network	9
1.3	Esempio di applicazione di una Mesh network	9
1.4	Streaming video da Smartphone a Samsung Smart TV	12
1.5	Esempio di Discovery con l'aiuto di un dispositivo vicino	12
1.6	Esempio (a) invio contenuto senza iDLS, (b) con iDLS	13
1.7	Architettura Subscribe-and-Send	15
1.8	Architettura iTrust	16
1.9	Topologia multi gruppo con e senza indirizzi IP	17
1.10	Rappresentazione di uno scenario di utilizzo di DSRC	18
2.1	Diagramma a stati di PingPong in generale	21
2.2	Diagramma a stati di PingPong nello specifico	22
2.3	Diagramma UML delle classi di PingPong	23
2.4	Interfaccia grafica dell'app PingPong	24
2.5	Interfaccia grafica dell'app Pigeon Messenger	26
2.6	Diagramma a stati di Pigeon Messenger in generale	28
2.7	Diagramma a stati di Pigeon Messenger con coda messaggi	29
2.8	Diagramma di sequenza di Pigeon Messenger	29
2.9	Diagramma UML delle classi di Pigeon Messenger	30
3.1	Order and Export di Eclipse con SDK modificato	33
4.1	Esempio di Code line in Android	47
4.2	Android framework	49
4.3	Android framework nel dettaglio	50
4.4	HAL	50
4.5	Processo di boot completo	51
4.6	Processo di boot - passo 1	52
4.7	Processo di boot - passo 2	53
4.8	Processo di boot - passo 3	54
4.9	Android root filesystem	55

4.10 Processo di boot - passo 4	55
4.11 Processo di boot completato	56
4.12 Licenza dei file binari proprietari ed estrazione completa	62
4.13 Architettura dell'Android Build System	64
4.14 Creazione configurazione di default per il kernel	69
4.15 Kernel compilato	69
4.16 Mkbootimg.h	70
4.17 Unmkbootimg eseguito	73
4.18 Configurazione kernel Linux	74
5.1 Stack di rete modificato con Microsoft MultiNet	79
5.2 Stack di rete modificato con Juggler	79
5.3 Connessione di due dispositivi con wpa_supplicant	91
5.4 Connessione di tre dispositivi con wpa_supplicant	92
5.5 Wpa_supplicant GO con due client su interfaccia p2p0	93
5.6 Wpa_supplicant primo Client connesso al GO	93
5.7 Wpa_supplicant GO con il secondo client su interfaccia di gruppo . .	94
5.8 Wpa_supplicant secondo Client connesso al GO	94
5.9 Kernel make menuconfig “Device Drivers”	96
5.10 Kernel make menuconfig abilitazione VIF	96
6.1 Architettura libnl	107
6.2 Architettura generale dei driver Wi-Fi nel kernel Linux	108
6.3 Architettura nel dettaglio dei driver Wi-Fi nel kernel Linux	108
6.4 Architettura Wi-Fi Nexus 5 e Samsung Galaxy S4	110
6.5 Risultato del comando grep -r “wl_cfg80211_add_virtual_iface” . .	116
6.6 Funzione del file wl_cfg80211.c	117
6.7 Errore “Virtual iface add failed”	117
6.8 Ricerca con grep wl_cfgp2p_ifadd	118
6.9 Definizione funzione wl_cfgp2p_ifadd in wl_cfgp2p.c	118
6.10 Chiamata a wldev_iovar_setbuff	118
6.11 Ricerca con grep di wldev_iovar_setbuff	118
6.12 Definizione wldev_iovar_setbuff e chiamata a wldev_ioctl	118
6.13 Definizione di wldev_ioctl e chiamata dhd_ioctl_entry_local	119

Elenco delle tabelle

4.1	Posizioni di default dei moduli in base al template usato	65
4.2	Tabella Build configuration per il comando lunch	67
4.3	Differenze tra le varie buildtype	67
4.4	Nomi e posizioni dei codici sorgenti del kernel	67

Elenco degli algoritmi

3.1	Blocco di codice statico per caricare con NDK le librerie native	35
3.2	Costruttore della classe WiFiNative	35
3.3	Costruttore della classe WifiP2pServiceImpl	36
3.4	Metodo che scansiona le interfacce di rete disponibili	37
3.5	Metodi per ottenere i nomi delle interfacce, specificandone l'indice .	37
3.6	Classe com_android_server_wifi_WifiNative.cpp	38
3.7	Metodo in C++ per ottenere le interfacce, dall'indice	39
3.8	Funzione C++ per ottenere i dati su RSSI	40
3.9	Inizializzazione RSSI per un dispositivo Wi-Fi Direct	40
3.10	Android.mk in Eclipse	41
3.11	Application.mk in Eclipse	41
3.12	Esempio Activity di un'app con NDK	42
3.13	External Tools Configurations di Eclipse per eseguire javah	43
3.14	Metodo <i>nativo provawpa</i> implementato in C con JNI	43
3.15	Risultato di readelf di libwpa_client.so	44
3.16	Esempio chiamate da C/C++ a Java	45
4.1	Aggiornare Ubuntu	57
4.2	Installare Java 7 e 6 e scegliere la versione da usare	58
4.3	Installare le dipendenze in Ubuntu	58
4.4	Installare Android NDK	58
4.5	Configurazione variabili d'ambiente in .bashrc	59
4.6	Configurazione variabili d'ambiente in .profile	60
4.7	Comandi per installare <i>Repo</i>	60
4.8	Comandi per inizializzare <i>Repo</i>	61
4.9	Avvio del download del sorgente di Android	61
4.10	Inizializzare l'ambiente per compilare Android	66
4.11	Scegliere obiettivo con lunch, attivare la ccache e compilare	66
4.12	Scaricare il codice sorgente del kernel msm Qualcomm	68
4.13	Configurazione di default e compilazione del kernel	68
4.14	Compilare mkbootimg dalla repository di Android	71
4.15	Compilare unmkbootimg da Github	71

4.16 Comando mkbootimg di Google per rimpacchettare boot.img	72
5.1 Download e compilazione di wpa_supplicant e wpa_cli	84
5.2 Procedura per installazione di Android SDK	85
5.3 Preparazione per eseguire wpa_supplicant e wpa_cli	86
5.4 wpa_supplicant.conf	87
5.5 p2p_supplicant.conf	87
5.6 Avviare wpa_supplicant e wpa_cli	88
5.7 wpa_cli avviato in “interactive mode”	88
5.8 Interfaccia p2p creata da wpa_supplicant e socket associato	94
5.9 MacVTap, operazione non supportata	95
5.10 Creazione interfaccia di rete di tipo MacVTap	96
5.11 Ifconfig mostra la nuova VIF	97
5.12 Ifconfig mostra la nuova VIF con IP assegnato ed UP	97
5.13 Lista dei socket Wi-Fi	97
5.14 Errore di connessione di wpa_supplicant con interfaccia MacVTap .	98
5.15 Tentativo di creazione di gruppi con p2p_group_add	98
5.16 Download e compilazione di iw per Android	100
5.17 Installazione di iw per Android	100
5.18 Ottenere lista VIFC	101
5.19 iw list su Nexus 4, Nexus 5 e Samsung Galaxy S4	102
5.20 iw dev per Samsung Galaxy S4 (GO) e Nexus 5 (Client)	103
6.1 Log dell'avvio kernel Linux con il comando dmesg	116
6.2 Log dell'errore durante la creazione di due VIF con il comando dmesg	116
6.3 Log dmesg con una printk aggiunta per mostrare il MAC address . .	120

Introduzione

Contesto

Nel 2013 sono stati contati 1,2 miliardi di Smartphone/Tablet in tutto il mondo [56]. Questi numeri impressionanti stanno crescendo senza freni e sono destinati solamente ad aumentare, infatti si prevedono 4 miliardi di Smartphone nel 2017 [56]. Inoltre, stanno nascendo nuovi mercati di dispositivi “Smart”, come per esempio quello degli *Smart Watch*, ancora giovane ed acerbo, ma pronto ad esplodere, o la diffusione di dispositivi di fascia bassa nei paesi in via di sviluppo.

Con il crescere del numero degli utenti, stanno aumentando anche il numero delle funzionalità di ogni dispositivo, sempre più vicine ad egualare quelle dei classici computer portatili. Anche le richieste e le esigenze degli utenti stessi stanno crescendo, alimentando nuove ricerche. Un esempio è il desiderio di essere sempre connessi ad internet e poter navigare a velocità sempre più elevate in ogni luogo. Infatti, in precedenza il problema della connettività su dispositivi mobili non è mai stato affrontato nel modo in cui si è costretti a fare ora. Adesso vi è una vera e propria spinta e richiesta degli utenti che dovrà essere soddisfatta. La comunicazione come siamo abituati a concepirla non è più sufficiente, ora si vuole un vero e proprio scambio di grandi quantità di dati in mobilità. Il concetto di “mobilità” non è stato considerato nel modo in cui lo vediamo oggi, perché alla fine degli anni ’90, quando è nata la tecnologia Wi-Fi, è stata pensata per un ambiente più statico. Oggi invece gli utenti richiedono sempre più dinamismo e di conseguenza dovrà, e di fatto sta subendo, una vera e propria evoluzione.

Questi desideri stanno dando vita ad un mondo, che negli ultimi anni è stato definito *Internet Of Things*, cioè un nuovo modo di usare la rete, dove ogni oggetto è connesso. In questo settore possiamo considerare il concetto di *Opportunistic Networks*, cioè reti senza fili, in cui i nodi sono dispositivi portati da utenti, senza infrastrutture di rete, con ricerca e comunicazione automatica in ambienti vari ed estremamente dinamici. In queste reti l’utente e il dispositivo sono un tutt’uno.

Queste nuove richieste si sono scontrate con un settore non pronto per soddisfarle, cioè le tecnologie e protocolli di comunicazione, che hanno dovuto reinventare la comunicazione tra dispositivi mobili con requisiti differenti. Infatti, negli ultimi anni stanno nascendo nuove soluzioni che mirano a rendere tali scenari non più idee in ambito della ricerca, ma pura realtà alla portata di tutti. Oggi ci sono diverse di queste tecnologie, ma sono ancora molto giovani, come per esempio *Wi-Fi Direct*, *Bluetooth 4* ed *LTE Direct* (ancora non disponibile pubblicamente).

In questo lavoro di tesi mi concentrerò sulla prima, cioè *Wi-Fi Direct*, che approfondirò nella Sezione 1.2. Il problema principale per il suo utilizzo in scenari per lo più assimilabili ad *Opportunistic Networks*, è che il protocollo non è stato pensato per gestire reti in situazioni molto dinamiche ed imprevedibili. Lo stato attuale di questo protocollo di rete è più vicino all'essere un passo intermedio verso il raggiungimento dell'obiettivo delle *Opportunistic Networks*.

Scopo del lavoro

Lo scopo di questo lavoro di tesi è quello di studiare e sperimentare una di queste tecnologie, cioè *Wi-Fi Direct*, su dispositivi Android. In particolare, l'obiettivo è il superamento di due ostacoli intrinseci nel protocollo per la creazione delle *Opportunistic Networks*, costruendo un livello software o middleware sopra ad esso, senza modificarne lo standard. Gli obiettivi sono:

- rendere il protocollo scalabile al crescere del numero dei dispositivi;
- rendere trasparente all'utente la ricostruzione della rete.

Nonostante il primo problema riguardi la scalabilità e il secondo la disponibilità, hanno in comune lo stesso requisito. Quest'ultimo si ricollega ai problemi già affrontati in precedenza da altri ricercatori. Infatti, a volte sono state proposte soluzioni difficilmente utilizzabili su un vasto numero di prodotti di diverse marche; altre volte appoggiandosi a più tecnologie, per cercare di superare i limiti di *Wi-Fi Direct*. Parlerò nel dettaglio di questo e di una ricerca che si avvicina molto al mio obiettivo nel Capitolo 1, ma che sfrutta anche la tecnologia Wi-Fi classica. Altre soluzioni, tra cui le più famose sono *AllSeen* (precedentemente *AllJoyn*) [1] e *Intel CCF* [58], sfruttano la stessa idea che per risolvere i problemi di un protocollo lo si può combinare con altri e scegliere quello più adatto in quel preciso momento.

Il mio obiettivo è differente, infatti voglio lavorare solamente su *Wi-Fi Direct* senza appoggiarmi ad altre tecnologie, per risolvere i limiti sopracitati. Per tale motivo non parlerò più delle altre tecnologie, alternative a *Wi-Fi Direct*.

Per raggiungere gli obiettivi, il requisito più importante, in comune con entrambi, è che un dispositivo Android possa appartenere a più gruppi *Wi-Fi Direct* contemporaneamente.

raneamente. Questo perché, idealmente, si potrebbero usare i dispositivi in comune tra i gruppi per ridirezionare il traffico dati mettendo in comunicazione tutti i dispositivi. Inoltre, questo permetterebbe di rendere la rete sempre disponibile tramite il concetto di ridondanza, cioè la replicazione di alcuni nodi della rete. Queste sono all’incirca le caratteristiche delle *reti mesh*, ma che purtroppo non sono supportate da Android.

Il problema principale è che in Android non è permesso che un dispositivo appartenga a due gruppi contemporaneamente. Questo non è un limite dello standard *Wi-Fi Direct*, il quale è molto aperto e allo stesso tempo impreciso su tale argomento, piuttosto una limitazione presente in Android, probabilmente per scelta di Google e degli altri partner che contribuiscono nello sviluppo di questo sistema operativo.

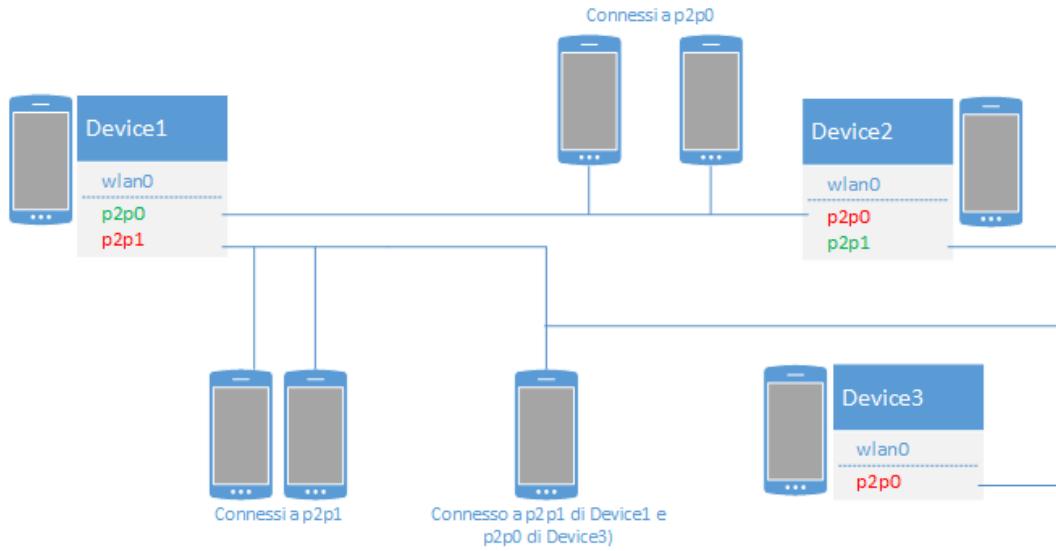
Prima di affrontare ciò in modo tecnico, propongo la mia soluzione teorica per risolvere il problema.

Scalabilità

Per rendere il protocollo scalabile in Android si può adottare la strategia intuitiva di far partecipare un dispositivo a più gruppi *Wi-Fi Direct* (avvicinandosi al concetto di *rete mesh*), cioè di utilizzare più interfacce di rete virtuali contemporaneamente. L’approccio comune adottato nelle pubblicazioni che citerò nel Capitolo 1 è quello di sfruttare le uniche interfacce preesistenti in Android. Ciò significa dover utilizzare il protocollo *Wi-Fi Direct* con un’interfaccia ed una rete Wi-Fi classica in modalità *Access Point* o *ad-hoc* con l’altra (parlerò in modo più approfondito di questi concetti nel Capitolo 1). Invece, la mia idea si basa sull’estensione di *Wi-Fi Direct* tramite la creazione di più interfacce virtuali di tipo P2P. Anche il concetto di *interfaccia di rete virtuale* lo approfondirò in seguito, ma intuitivamente può essere considerato come l’equivalente di una macchina virtuale, cioè su una sola scheda di rete fisica si può creare più interfacce di rete, dette “virtuali”.

A tal proposito, nella figura seguente, mostro la soluzione per rendere il protocollo scalabile considerando più interfacce P2P contemporaneamente. In questo modo, potendo connettere ogni nodo a più gruppi, il problema della scalabilità sarebbe risolto. Infatti, ogni dispositivo potrebbe gestire comunque un numero limitato di *Client* e comunicare con gli altri quando è necessario, tramite trasmissioni *multi-hop*. Quindi, ogni *Group Owner* (GO)¹ raccoglierebbe i dati dei propri *Client* e li inoltrerebbe agli altri. Questa soluzione è molto simile a quella proposta in [26], l’unica differenza consiste nella mia volontà di adottare solo interfacce P2P per rendere il tutto più scalabile e sfruttare i miglioramenti introdotti col protocollo *Wi-Fi Direct*, rispetto alle *reti ad-hoc*, che discuterò nel Capitolo 1.

¹In *Wi-Fi Direct* si definisce *Group Owner* l’equivalente di un *Access Point* Wi-Fi, cioè un dispositivo che gestisce un gruppo, a cui si collegano altri dispositivi, detti *Client*. Per maggiori informazioni fare riferimento alle specifiche ufficiali [3].



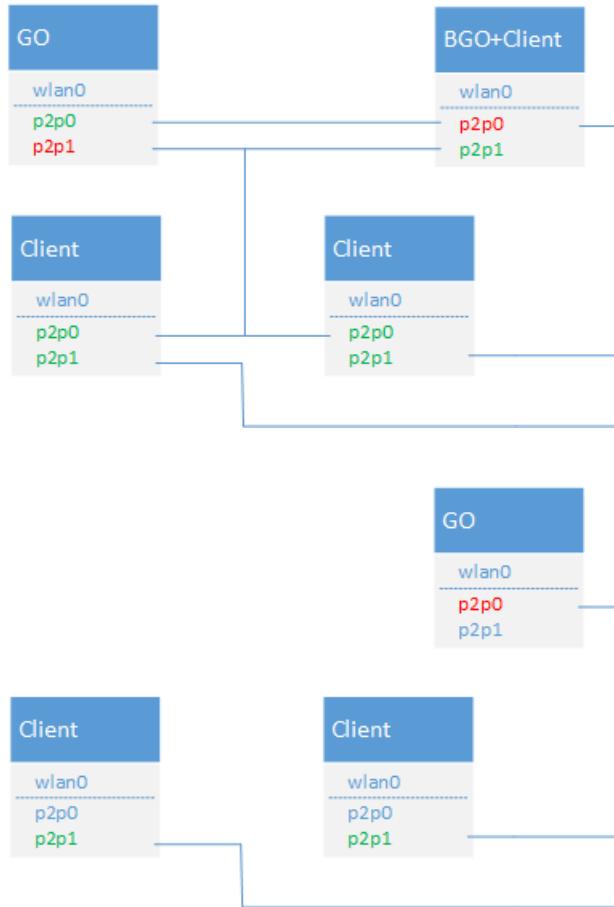
Disponibilità

Nelle specifiche di *Wi-Fi Direct* c'è scritto che il ruolo di GO non è trasferibile quando la rete è formata, ma se dovesse non essere più disponibile, l'intera rete cadrebbe di conseguenza. Per far sì che una rete *Wi-Fi Direct* sia sempre disponibile, è necessario creare un sistema che alla caduta di un GO sia in grado di ricollegare i Client ai GO esistenti, oppure eleggere uno dei Client come GO al quale essi si collegheranno automaticamente. In ogni caso, poiché *Wi-Fi Direct* non è un protocollo pensato per questo (non solo in Android, ma anche nelle specifiche), è necessario che alla formazione del gruppo e ad ogni cambiamento nel gruppo stesso, tutti i Client e il GO siano informati sullo stato di tutti i dispositivi presenti in esso. Ciò è facilmente realizzabile adottando un approccio a scambio di messaggi. Una tecnica simile verrà adottata nel Capitolo 2 per realizzare l'app chiamata *Pigeon Messenger*. Anche in questo caso, il problema principale è che per rendere totalmente trasparente il processo, cioè non creare disagi agli utenti finali, l'operazione dovrebbe essere svolta in tempi molto brevi. Purtroppo, a causa di problemi intrinseci nel protocollo dovuti alla *fase di Discovery*² non si riesce ad ottenere performance accettabili, come dimostrerò nel Capitolo 2 tramite l'app chiamata *PingPong*.

Quindi ho deciso di pensare ad una soluzione differente, cioè far sì che ogni GO abbia un *Backup Group Owner* (BGO) scelto tra i Client del gruppo stesso. A questo BGO sono collegati tutti gli altri Client del gruppo e il GO stesso. Quindi alla caduta del Group Owner, il BGO si può accorgere facilmente del cambiamento di stato del GO e continuare le comunicazioni con i dispositivi. Nelle figure seguenti, c'è un

²In *Wi-Fi Direct* la *fase di Discovery* è un intervallo di tempo in cui i dispositivi cercano un canale di comunicazione comune per poter dialogare. Per maggiore informazioni vedere le specifiche ufficiali del protocollo [3].

esempio di questa situazione, infatti nella prima si può trovare il GO e il suo Backup, mentre nella seconda il GO cade (per esempio esce dall'area di ricezione o finisce la batteria) e il BGO diventa velocemente il GO, grazie al fatto che i collegamenti sono già stati creati prima, non si verificherebbero ritardi legati al protocollo.



Come già anticipato, questa soluzione richiede comunque l'uso di interfacce di rete virtuali, esattamente come la soluzione al problema precedente. Visto che gli obiettivi posti sono due, le soluzioni di scalabilità e disponibilità devono valere contemporaneamente, questo fa sì che siano necessarie molte più interfacce di rete virtuali per ogni dispositivo. Per esempio, se impongo il limite che un dispositivo può appartenere a due soli gruppi, sarebbero necessarie due interfacce per la connessione ed altre due di “backup”. Quindi, l'obiettivo si estende a trovare un modo per supportare un numero di interfacce virtuali variabile, almeno quattro. Questo fa sì che tutte le soluzioni con l'interfaccia Wi-Fi classica configurata come *rete ad-hoc* o *Access Point* siano scartate a priori (che descriverò nel Capitolo 1).

Struttura della tesi

Il documento è strutturato secondo i seguenti capitoli.

- Il capitolo 1 descrive lo stato dell'arte e introduce al protocollo Wi-Fi, alle tipologie di rete e a *Wi-Fi Direct*. Inoltre, comprende un inquadramento sullo sviluppo attuale di progetti relativi principalmente ad *Opportunistic Networks* con reti *ad-hoc* e/o *Wi-Fi Direct*.
- Il capitolo 2 spiega alcuni problemi dell'implementazione di *Wi-Fi Direct* in Android, tramite la progettazione ed implementazione di due app, *PingPong* e *Pigeon Messenger*.
- Il capitolo 3 descrive lo studio del framework di Android, dei suoi limiti e le sue possibili estensioni.
- Il capitolo 4 mostra l'*Android Build System*, le procedure per compilare Android, il kernel e modificarne la configurazione.
- Il capitolo 5 parla di interfacce di rete virtuali, come abilitarle nel kernel di Android ed i loro limiti, per poi utilizzare *wpa_supplicant* con l'obiettivo di risolvere i problemi accennati in questa introduzione.
- Il capitolo 6 analizza nel dettaglio i driver Wi-Fi nel kernel Linux, in particolare quelli di Broadcom per il Nexus 5. Infine, mostra i limiti di tali driver per quanto riguarda la gestione delle interfacce di rete virtuali.
- Le appendici mostrano informazioni e procedure basate sulle mie esperienze che possono essere utili per svolgere alcune operazioni accennate in questo lavoro di tesi.

Capitolo 1

Stato dell'arte

Nelle successive sezioni descriverò le tecnologie Wi-Fi e *Wi-Fi Direct*. Dati i loro vasti documenti di specifica, mi concentrerò sui concetti necessari per comprenderne il funzionamento, quindi ometterò i dettagli che non hanno avuto rilevanza in questo lavoro di tesi. Successivamente, tratterò alcuni scenari di utilizzo di *Wi-Fi Direct* e farò riferimento allo stato attuale della ricerca. Per un maggiore approfondimento del protocollo *Wi-Fi Direct*, fare riferimento alla specifiche ufficiali in [3].

1.1 Wi-Fi

Nel 1999, diverse compagnie sparse in tutto il mondo si sono unite per lavorare assieme in un'associazione no-profit chiamata *Wi-Fi Alliance* [2], con l'obiettivo di diffondere le tecnologie wireless senza considerare il loro marchio. Da ciò è nata la tecnologia chiamata commercialmente come *Wi-Fi* [11]. Essa permette la connettività tra dispositivi senza fili in una rete locale basandosi sulle specifiche dello standard *IEEE 802.11* [112]. Oggi Wi-Fi è nel 25 percento delle abitazioni di tutto il mondo, con circa due milioni di dispositivi venduti solo nel 2013. L'adozione di Wi-Fi continua a crescere con la comune visione di connettere tutti ed ovunque. Ad alimentare ancora di più la diffusione di questa tecnologia sono stati i dispositivi mobili come Smartphone e Tablet, che fin dalla loro nascita hanno adottato Wi-Fi. Infatti, ricordo che il primo iPhone uscito nel 2007, considerato come il primo Smartphone, ha avuto fin da subito un chip Wi-Fi (802.11 b/g) per la connessione ad Internet [95].

Una rete Wi-Fi definita secondo IEEE 802.11 è costituita da *Stazioni* (STA), cioè dispositivi che possono collegarsi in un rete senza fili tramite *Wireless Network Interface Controller* (WNIC). Le stazioni si dividono in *Client* e *Access Point* (AP). Quest'ultimo permette la comunicazione tra i Client e di solito fa da router wireless. L'insieme di tutte le STA che possono comunicare tra loro è detto *Basic Service Set* (BSS). Ogni BSS ha un *identificativo* (ID) chiamato *BSSID* che corrisponde al MAC address dell'Access Point. Ci sono tre tipi di BSS [99] (Figura 1.1).

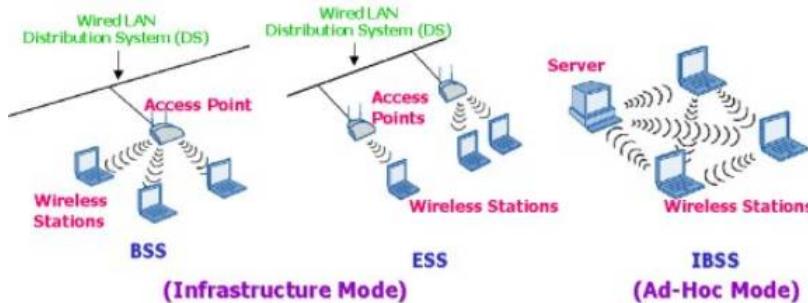


Figura 1.1: Tipi di reti Wi-Fi (BSS, ESS, IBSS)

- Infrastructure BSS: c'è un solo AP che controlla tutto il traffico.
- Extended Service Set (ESS): due o più BSS che condividono lo stesso SSID e credenziali di accesso e appaiono come un singolo BSS. I BSS possono anche condividere lo stesso canale di trasmissione.
- Indipendent BSS (IBSS) o *ad-hoc*: non prevede un Access Point tra le proprie STA, quindi permette la comunicazione tra i propri Client senza che un AP li coordini, cioè può essere considerato come un antenato di *Wi-Fi Direct*.

Poiché parlerò più volte di *reti ad-hoc* o IBSS, sia in questo capitolo, sia nei successivi, è importante evidenziare che i problemi principali sono:

- velocità di trasferimento di soli 11 Mbps;
- non poter utilizzare la connessione internet e quella *ad-hoc* contemporaneamente;
- non compatibilità con tutti i dispositivi (quelli Android spesso non la supportano, per scelte dei produttori).

1.1.1 Wireless Mesh Network

Per concludere questa breve trattazione su Wi-Fi, ci tengo a precisare che in realtà esiste una quarta modalità di funzionamento, detta Wireless Mesh Network [98], le cui specifiche sono state definite in IEEE 802.11s [94]. Una *Wireless Mesh Network* (WMN) è una rete costituita da nodi organizzati in una *topologia mesh*, come in Figura 1.2. Essa è anche una forma di *rete ad-hoc*, che è molto vicina alle *Mobile Ad-Hoc Network* (MANET), anche se quest'ultime considerano la mobilità dei nodi.

Le Mesh newtork sono spesso costituite da *Mesh Client*, *Mesh Router* e *Gateway*. Generalmente, i *Mesh Client* sono laptop o dispositivi mobili, mentre i *Mesh Router* inoltrano il traffico dai/ai *Gateway*, che potrebbero anche essere connessi ad Internet. Solo STA di tipo Mesh posso partecipare in questa rete, infatti una stazione di questo

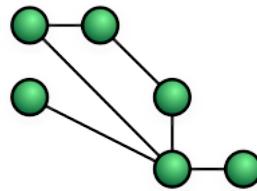


Figura 1.2: Mesh network

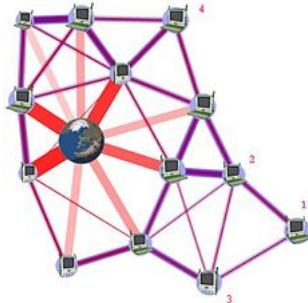


Figura 1.3: Esempio di applicazione di una Mesh network

tipo non fa parte di un BSS o di un IBSS, quindi una STA Mesh comunica solo con altre stazioni della stessa tipologia. Anche se, IEEE 802.11s fornisce meccanismi di proxy per permettere la comunicazione tra nodi Mesh e dispositivi 802.11 classici [94].

Le caratteristiche più importanti di una Mesh Network sono l'affidabilità e la ridondanza. Infatti, quando un nodo non può più funzionare, i restanti possono continuare a comunicare tra loro, direttamente o attraverso altri nodi intermedi.

In Figura 1.3 è rappresentata una possibile applicazione di queste reti, cioè quella di fornire una connessione internet anche ai nodi al di fuori dell'area di ricezione dell'AP (per esempio i nodi 1, 2, 3 e 4), sfruttando la connessione delle STA Mesh più vicine.

1.2 Wi-Fi Direct

Nel 2007, *Wi-Fi Alliance* ha introdotto una nuova tecnologia chiamata *Wi-Fi Direct*, inizialmente nota come *Wi-Fi P2P*, con l'obiettivo di supportare comunicazioni *peer-to-peer* (P2P) promettendone una facile configurazione ed utilizzo. Inoltre, ha fatto sì che tale protocollo permettesse velocità di trasferimento di 250 Mbps, superiore alle tecnologie concorrenti (per esempio *LTE Direct*, *Bluetooth 4*, *reti ad-hoc IBSS*) e compatibilità su tutti i dispositivi Wi-Fi, semplicemente tramite un aggiornamento software, aprendo le porte a nuovi scenari di utilizzo.

Un dispositivo che fa parte di una rete *Wi-Fi Direct* è detto *Peer* (o “P2P Device”). Può essere un *Group Owner* (GO), cioè un AP che fornisce funzionalità BSS ai Client, a volte chiamato *Software Access Point* (Soft-AP), oppure può essere un

Client che si distingue in *Legacy* e *P2P*. Il primo non supporta tutte le funzionalità del protocollo, quindi in questo lavoro di tesi non lo considererò, mentre il secondo è il dispositivo che è connesso al GO. L'insieme di GO e i suoi Client è detto *gruppo* (P2P Group) che può essere *Standard*, *Persistent* o *Autonomous*. Per maggiori informazioni ed approfondimenti, in particolare sulla *fase di Discovery*, i tipi e la formazione dei gruppi, rimando alle specifiche ufficiali disponibili in [3].

Purtroppo, nella definizione di tali specifiche, *Wi-Fi Alliance* non ha considerato molte applicazioni che in ambito di ricerca hanno trovato terreno fertile, in particolare quelle in cui la rete è soggetta a continui cambiamenti, o costituita da dispositivi molto differenti. In questa situazione ci sono le *Opportunistic Networks*, che rappresentano la naturale evoluzione delle *Mobile Ad-Hoc Network* (MANET), superando le limitazioni ed i vincoli di questo paradigma, a causa di topologie di rete estremamente dinamiche. Esse traggono vantaggio dalla mobilità degli individui e dal conseguente dinamismo di rete, definendo comunicazioni generate dagli incontri casuali tra le persone e i loro dispositivi. In questo scenario, quando due utenti entrano in contatto, possono sfruttare comunicazioni dirette per fornire accesso ad internet, scambiare contenuti, offrire servizi, condividere risorse ed inviare messaggi, anche tra altri dispositivi che non sono in contatto in quel preciso istante [73].

1.3 Scenari di utilizzo

In passato sono usciti molti articoli di progetti che si basano su *reti ad-hoc* come tecnologia di comunicazione. Al contrario, *Wi-Fi Direct* è un protocollo ancora abbastanza giovane e spesso le ricerche sono riadattamenti, da *reti ad-hoc* a *Wi-Fi Direct*, di lavori precedenti. Ne mostrerò alcuni esempi nella Sottosezione 1.3.3. Molte di esse si concentrano nella soluzione di problemi specifici in un determinato scenario, come per esempio lo streaming video, la gestione di emergenze o la comunicazione libera senza possibilità di censura, ma non cercano di risolvere gli attuali problemi di *Wi-Fi Direct*, piuttosto trovano dei modi per aggirarli.

In questo lavoro di tesi, mi sono posto come obiettivo quello di risolvere i problemi specifici descritti nell'introduzione, che sono alla base di molte di queste ricerche. L'unico articolo che conosco che affronta uno di questi problemi in modo specifico e dettagliato è [26], riassunto nella Sezione 1.3.4. Gli autori sfruttano la combinazione di Wi-Fi e *Wi-Fi Direct*, mentre il mio obiettivo è estendere quest'ultimo protocollo, risolvendo a livello software i due problemi specifici già citati, senza utilizzare il classico Wi-Fi. La differenza è che con la soluzione che propongo, il protocollo risulterebbe ancora più scalabile.

Per quanto riguarda il secondo problema, cioè quello di rendere resistente ai guasti una rete *Wi-Fi Direct*, non conosco lavori specifici al riguardo.

1.3.1 Condivisione video in streaming

In questi ultimi anni, nelle abitazioni si stanno diffondendo sempre più dispositivi elettronici Wi-Fi, di tipo diverso dal punto di vista delle capacità di rete, elaborazione e gestione del consumo energetico. Per esempio le Smart TV¹, i *Media Player* connessi alla TV, i dispositivi di archiviazione di rete (NAS [101]) con supporto *DLNA* [113], gli Smartphone, i Tablet oltre ai classici PC e laptop.

In un ambiente casalingo, per la condivisione di dati tra questi dispositivi non è necessario avere un *Access Point* connesso ad internet, poiché lo scambio di informazioni avviene nella rete locale (LAN). Quindi, poter utilizzare reti *Wi-Fi Direct* per la condivisione locale di dati è uno scenario in cui questo protocollo trova una perfetta collocazione. Addirittura, lo si può utilizzare per lo streaming video, sfruttando l'elevata velocità di trasmissione/ricezione dei nuovi dispositivi [52].

Con la diffusione del nuovo standard IEEE 802.11ac (velocità di trasferimento superiori ad 1 Gbps), lo streaming video ad alta definizione è sempre più interessante, anche per risoluzioni 4K. Infatti, è evidente che gli utenti vogliono poter vedere video su schermi sempre più grandi, lo dimostra l'attuale trend di mercato, cioè i nuovi Smartphone, i Phablet² e le TV sempre più grandi. In questi casi, poter inviare in streaming il video dal proprio Smartphone alla TV è sempre più interessante, tanto che Google l'ha realizzato con il *Chromecast* [40], mentre Samsung ed altri produttori hanno integrato le tecnologie necessarie direttamente nelle loro TV [12, 79].

È comunque importante sottolineare che i *Media Player* come per esempio quelli prodotti da Western Digital (WD TV Live) o Apple (Apple TV) non sono una novità sul mercato. Infatti, la WD TV Live permette già di utilizzare *Wi-Fi Direct* per lo streaming video e addirittura per la condivisione dello schermo di uno Smartphone Android, come il Nexus 5.

Streaming video su N dispositivi senza interruzioni

Uno scenario che ha attirato l'attenzione in ambito di ricerca è quello di sfruttare le potenzialità di *Wi-Fi Direct* per visualizzare un video da uno Smartphone/Tablet su N dispositivi in un ambiente domestico, sfruttando il protocollo DLNA [113]. Il problema principale è che la procedura di ricerca dei dispositivi vicini, detta *Discovery*, richiede alcuni secondi e intanto il flusso video potrebbe subire interruzioni. L'articolo di Bong-Jin Oh e Sunggeun Jin [12] mostra come modificare questa fase del protocollo per renderla più veloce ed evitare interruzioni nel flusso video, nel caso della specifica applicazione in esame. La tecnica si basa sullo sfruttamento di

¹Con Smart TV si definisce commercialmente la categoria di apparecchiature elettroniche di consumo che hanno come principale caratteristica l'integrazione di funzioni e di servizi legati a internet. La stessa locuzione è utilizzata in modo più generico per definire la tendenza alla convergenza tecnologica tra il mondo dei personal computer e quello della televisione [106].

²*Phablet* è un termine informale per dispositivi touch-screen con schermi compresi tra i 5.3 pollici e i 6.99 pollici, che combinano le caratteristiche degli Smartphone e dei Mini Tablet [102].



Figura 1.4: Streaming video da Smartphone a Samsung Smart TV

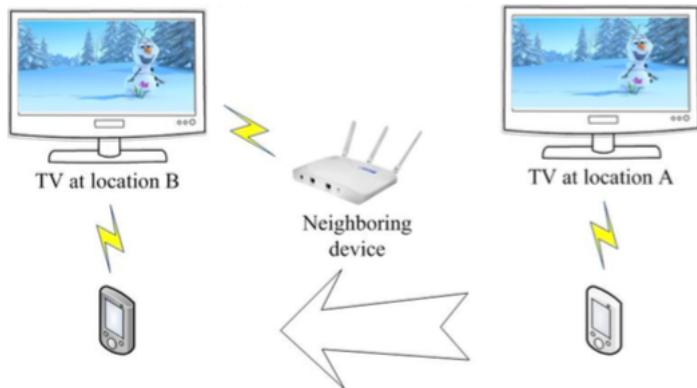


Figura 1.5: Esempio di Discovery con l'aiuto di un dispositivo vicino [12]

un meccanismo di caching. Lo scenario proposto dagli autori è quello di una persona che con lo Smartphone guarda un video sulla TV, dopodiché inizia a muoversi ed automaticamente continua il video su una seconda TV in un altro luogo. Il miglioramento della fase di Discovery consiste nel salvataggio di informazioni in una cache, per far sì che i dispositivi vicini si aiutino a trovarsi, come in Figura 1.5. Per migliorare l'*hit ratio* della cache, l'AP può stimare il movimento dello Smartphone in modo che l'*Access Point* stesso possa mandare le risposte solo nel caso in cui lo Smartphone si stia avvicinando [12].

Condivisione collaborativa di contenuti multimediali in streaming

Hayoung Yoon e JongWon Kim si sono occupati sempre di questo problema, ma essendo uno scenario costituito da dispositivi molto vari che coesistono, tra cui pro-

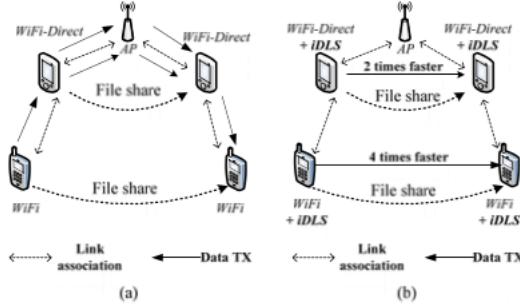


Figura 1.6: Esempio (a) invio contenuto senza iDLS, (b) con iDLS [52]

dotti Wi-Fi convenzionali e *Wi-Fi Direct*, oltre al problema della *qualità del servizio* (Quality of Service, *QoS*) hanno evidenziato che il trasferimento dati tra i dispositivi può non essere efficiente. La loro soluzione è *Decentralized cOllaborative Media contents Streaming* (DOMS) che si occupa di questi problemi sfruttando: una versione modificata di *DLS*, che hanno chiamato *inter-BSS Direct Link Setup* (iDLS)³ e la condivisione collaborativa di contenuti multimediali tra i dispositivi. Per la *QoS* è stato creato lo standard *IEEE 802.11e* che definisce i concetti di *QoS enabled STA* (QSTA), dove una QSTA può trasmettere dati direttamente ad un'altra creando una connessione diretta, cioè usando quello che è stato specificato nel protocollo *DLS*. Quest'ultimo ne permette un aumento del throughput [52].

1.3.2 Riduzione del traffico sulla rete mobile

LU Xiaofeng, HUI Pan e Pietro Lio' [68] hanno utilizzato *Wi-Fi Direct* in un ambito differente, cioè hanno pensato ad un modo per ridurre il traffico a cui sono soggette le reti mobili. Infatti, attualmente le reti dati cellulari non hanno capacità sufficienti per soddisfare le crescenti richieste degli utenti e nonostante gli operatori telefonici siano in grado di aumentare le capacità della rete, tutto ciò è costoso e richiede tempo. Quindi, si può utilizzare un protocollo per far scambiare dati tra i dispositivi, senza doverli scaricare nuovamente dalla rete mobile. Il problema è che i contatti tra i terminali sono costantemente disturbati dalla mobilità stessa degli utenti. Questo può creare problemi quando le performance sono essenziali, come lo streaming video, ma vi sono scenari in cui un piccolo ritardo può essere considerato accettabile dall'utente, per esempio l'invio di una e-mail.

Gli autori partono dal presupposto verosimile che molte persone tenderanno a richiedere gli stessi contenuti. Scaricarli su ogni dispositivo può rischiare di saturare la rete, ma quando un utente richiede un contenuto si aspetta di ottenerlo nel più breve tempo possibile. Per questo hanno pensato ad un'architettura in cui prima di

³Direct Link Setup (DLS) permette trasferimenti diretti stazione-a-stazione nello stesso BSS e fa parte di IEEE 802.11e. Esiste anche una versione pubblicata con IEEE 802.11z detta TDLS che permette il funzionamento del protocollo anche senza AP. Di fatto TDLS è considerata come un'alternativa a Wi-Fi Direct [104].

ottenere il contenuto attraverso la rete cellulare, venga richiesto ai dispositivi vicini. Gli autori si sono basati su alcune ricerche preesistenti, come quella di Haddadi che ha proposto una soluzione scalabile chiamata *MobiAd* per sfruttare le informazioni in ambiente mobile per fornire pubblicità personalizzate. Oppure quella di Dimatteo et. al., che hanno proposto un'architettura per migrare il traffico da reti cellulari a AP Wi-Fi metropolitani, considerando come riferimento il traffico di video in streaming.

Inoltre, nel passato sono stati presentati molti protocolli di *Opportunistic Routing*, come *Epidemic Routing* (ER) affetto da problemi di performance e con dimensione del buffer limitate, o *PRoPHET* che usa la storia degli incontri tra dispositivi per calcolare la transitività di un messaggio alla destinazione, cioè tiene conto della probabilità che un dispositivo riesca ad inviare un messaggio a destinazione. Un'altra soluzione è *Spray-and-Wait*, un protocollo di routing che tenta di trarre benefici dalla replicazione. Invece, *MaxProp* replica solo messaggi che non sono posseduti dal nodo incontrato. Quando i dispositivi si incontrano, si scambiano i messaggi che non hanno. Purtroppo, la complessità di *MaxProp* è elevata.

La soluzione degli autori, chiamata *HPRO*, è un protocollo che si basa sul fatto che è meglio inviare i dati solo quando i nodi incontrati raggiungono una certa soglia. Essi hanno creato un'architettura *Subscribe-and-Send* rappresentata in Figura 1.7, basata su una tabella detta *Subscription table*, dove un utente si *sottoscrive* ai contenuti attraverso l'infrastruttura di rete cellulare, ma non li scarica se non strettamente necessario. Alcuni utenti che hanno i contenuti li inviano ad altri *subscriber* attraverso Wi-Fi con *HPro*. Usando questo protocollo, chi possiede il contenuto lo invia quando il numero dei suoi vicini è più grande o uguale di una certa soglia che varia in base alla densità locale di chi lo possiede. Le simulazioni dicono che *HPro* e l'architettura *Subscribe-and-Send* permettono di ridurre il traffico di rete cellulare [68].

1.3.3 Diffusione d'informazioni impedendo la censura

Lombera et. al. hanno creato *iTrust* [59], cioè un sistema P2P decentralizzato di pubblicazione, ricerca ed acquisizione con l'obiettivo di disseminare informazioni impedendone la censura. *iTrust* è stato creato focalizzandosi su HTTP, SMS, ma anche *Wi-Fi Direct*. Gli autori hanno scelto Android, perché l'unico con *Wi-Fi Direct*. Inoltre, hanno citato altri progetti simili, come per esempio:

- *Commotion Wireless* [29] che ha l'obiettivo di impedire il controllo o la censura delle comunicazioni da parte di regimi autoritari;
- Thomas et. al. [63] che hanno creato un sistema per la gestione dei disastri e emergenze usando Smartphone Android, permettendo connessioni senza la rete cellulare. Inoltre, hanno proposto *Smart Phone Ad-Hoc Networks* (SPAN) che

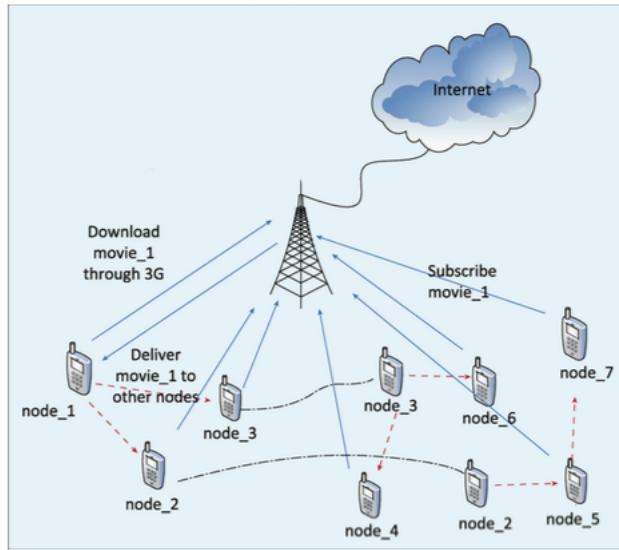


Figura 1.7: Architettura Subscribe-and-Send [68]

riconfigura i chip Wi-Fi sugli Smartphone per agire come router Wi-Fi, ma non utilizza *Wi-Fi Direct*;

- il *Serval Project* [80] che sta sviluppando una piattaforma wireless ad-hoc per Smartphone chiamata *Serval BatPhone*. Il progetto ha come obiettivo le zone rurali e popolazioni in zone isolate/remote, gestione di disastri ed emergenze ed ovviamente le comunicazioni anche in caso di censura. Questo progetto usa le reti *ad-hoc* su dispositivi Android, apportando delle modifiche agli Smartphone;
- Meroni et. al. [75] hanno creato una piattaforma per dispositivi Android, che usa reti *ad-hoc* per ridurre il consumo delle reti cellulari.

1.3.4 Utilizzo in caso di emergenze e disastri

Un’altro possibile utilizzo di *Wi-Fi Direct* è stato mostrato da Malabocchia et. al. [26] permettendo la comunicazione dei peer tra più gruppi. Lo scenario di utilizzo è quello delle emergenze, dove strutture sovraccaricate o danneggiate possono essere aiutate/sostituite da dispositivi Android. In caso di emergenza, devono essere inviati alcuni messaggi in broadcast per informare e istruire le persone su come comportarsi, anche se sono al di fuori della copertura cellulare. Comunque, organizzatori o membri di squadre di emergenza possono usarlo per predisporre punti di raccolta e ciò richiede che l’informazione raggiunga le persone in una precisa regione. Addirittura, un’individuo nella folla potrebbe usarlo per richiedere aiuto per una persona vicina che si è sentita male [26].

Per raggiungere tali obiettivi, gli autori hanno creato un *sistema di tunnelling* per rinviare un messaggio attraverso più hop in più gruppi. Per disseminare dati attra-

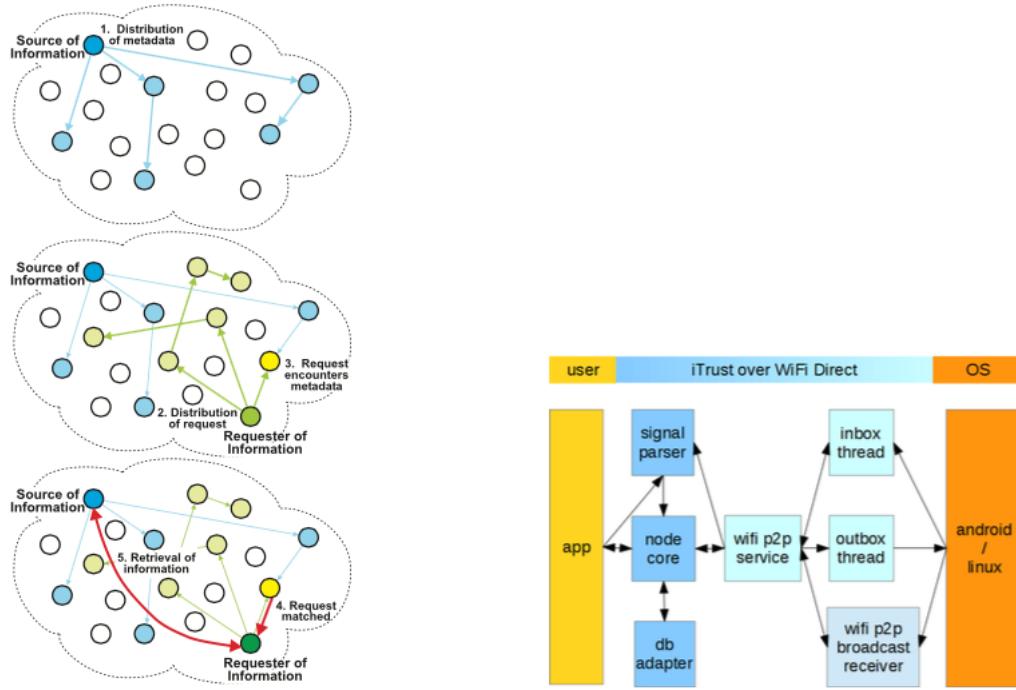


Figura 1.8: Architettura iTrust [68]

verso un grande numero di dispositivi hanno proposto un topologia che connette un numero arbitrario di gruppi, come in Figura 1.9. In teoria, l'approccio potrebbe scalare all'infinito, ma questa soluzione prevede l'uso di *Wi-Fi Direct* in contemporanea con una comune connessione Wi-Fi su Android. Nonostante ciò, come già anticipato nell'introduzione di questo documento, tale ricerca è quella che più si avvicina ad uno dei miei obiettivi.

1.3.5 Evitare incidenti tra veicoli

Il numero di veicoli in circolazione sulle strade è in continuo aumento e il rischio di incidenti cresce. In U.S.A. è stato creato *Dedicated short-range communications* (DSRC) [93], cioè canali di comunicazione Wi-Fi a medio/corto raggio progettati per ambito automobilistico e un corrispondente insieme di protocolli e standard. DSRC è un rete in cui i nodi sono rappresentati dai veicoli in movimento e da unità fisse poste lungo le strade che entrano in comunicazione (vedi Figura 1.10). L'obiettivo è quello di fornire informazioni sul traffico e avvisi di sicurezza, cioè per ridurre il rischio di congestione e di incidenti. Questo è più efficace rispetto a lasciare che il veicolo li risolva individualmente.

Inoltre, è in fase di sviluppo da General Motors e altre società come Toyota e BMW, una tecnologia chiamata *Vehicular-to-Vehicular* (V2V). Essa è anche nota come *Vehicular Ad-Hoc Network* (VANET), cioè la variazione di una *Mobile Ad-Hoc Network* (MANET), orientata ai veicoli. L'obiettivo è formare *reti ad-hoc* tra i

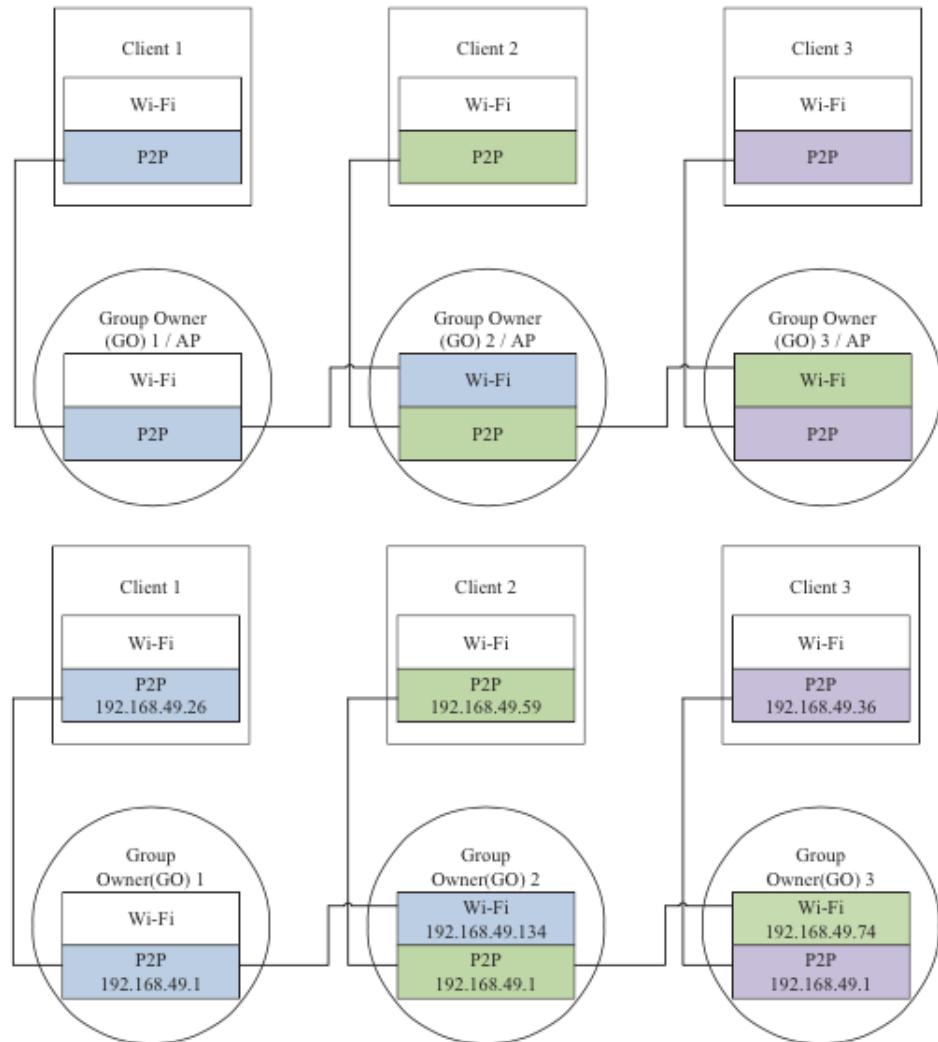


Figura 1.9: Topologia multi gruppo con e senza indirizzi IP [26]

1. Stato dell'arte



Figura 1.10: Rappresentazione di uno scenario di utilizzo di DSRC

veicoli per evitare incidenti. Nell'aprile 2014, negli U.S.A. questa tecnologia è stata approvata e nel 2017 dovrebbe diventare obbligatoria [111]. Un'altra applicazione è quella in cui le informazioni sono scambiate tra veicoli e infrastrutture, cioè *Vehicle to Infrastructure* (V2I), facilitando lo scambio di informazioni per esempio sui parcheggi liberi o sul traffico [110].

Poiché in questo lavoro di tesi mi sono concentrato su *Wi-Fi Direct*, è sensato pensarne l'applicazione in questo ambito, soprattutto perché riflette perfettamente una *Opportunistic Networks* ed evita di creare costose infrastrutture lungo le strade. Un esempio di studio in tale campo è stato realizzato da Chaitra Satish [85].

Capitolo 2

Realizzazione di due app in SDK

Nell'introduzione ho spiegato quali sono gli obiettivi e come raggiungerli dal punto di vista teorico. Prima di scendere nel dettaglio di questo problema, ho utilizzato le API di Android *Software Development Kit* (SDK) [34] in *Android Studio* [35] per esplorarne le capacità e le funzioni nel caso di *Wi-Fi Direct*. Poiché, ho già spiegato che Android non permette ad un dispositivo di essere in più gruppi contemporaneamente, ho realizzato due app con l'obiettivo di nascondere tale limite all'utente finale. Una si concentra su un caso specifico di utilizzo, cioè la messaggistica con la possibilità di accodare i messaggi e riconnettersi automaticamente alle persone con cui si sta comunicando. Invece, l'altra sull'idea di far partecipare un dispositivo a due gruppi, cioè una sequenza di connessioni e disconnessioni per dare l'impressione di una partecipazione simultanea in più gruppi.

In questo capitolo non mi concentrerò sugli aspetti implementativi, poiché i progetti completi in *Android Studio* sono disponibili online nelle *repository GitHub* [15,16], piuttosto sulla progettazione. A causa della natura totalmente asincrona di queste app, cioè basate su eventi, ho realizzato diversi diagrammi per spiegarne il funzionamento. Infatti, ho creato quelli a stati per mostrare l'evoluzione delle app, dall'apertura fino al raggiungimento della funzionalità desiderata, e quelli delle classi per mostrare alcune componenti importanti.

2.1 PingPong

La prima app permette ad un Client già connesso ad un GO di svolgere questa procedura ciclica:

1. scollegarsi dal proprio GO;
2. collegarsi ad uno specifico GO nelle vicinanze;
3. scollegarsi nuovamente;

4. ricollegarsi al GO iniziale e così via.

Da questo il nome dell'app, *PingPong*. Riferendomi al fatto il dispositivo Client si comporta come la pallina, colpita dalle due racchette (rappresentate dai GO) e costretta a rimbalzare da un lato ad un altro. Durante la realizzazione di quest'app, sia nella documentazione, sia nel codice, ho utilizzato una terminologia specifica per identificarne le componenti e le azioni svolte. Per essere coerente, userò questi termini anche in questo documento.

- “PingPong device”: è il Client che salta da un gruppo all’altro. Riprendendo l’analogia del ping-pong, inteso come gioco, è la pallina.
- “Rimbalzare” o “rimbalzo”: azione che corrisponde all’esecuzione della procedura ciclica, definita poco fa.
- “Ping device”: è il GO del primo gruppo *Wi-Fi Direct* a cui il *PingPong device* è connesso, prima di iniziare a *rimbalzare*.
- “Pong device”: è il GO del secondo gruppo *Wi-Fi Direct* a cui il *PingPong device* si connetterà, durante il *rimbalzo*.
- “PingPong mode”: modalità da attivare sui dispositivi che vogliono partecipare ad uno dei gruppi, mentre il *PingPong device* *rimbalza*. Permette di riavviare automaticamente la *fase di Discovery*, necessaria tra ogni disconnessione e riconnessione per individuare nuovamente i *Peer*, quando è in corso il *rimbalzo*. Per raggiungere tale obiettivo ho creato questa modalità che si basa solo su eventi per svolgere determinate azioni. Normalmente è disattivata, infatti non è necessaria per la connessione o lo scambio di file, ma è importante solo per sincronizzare i dispositivi durante la procedura di *rimbalzo* del *PingPong device*.
- “Test mode”: modalità che ho creato per testare la funzionalità del *rimbalzo* del *PingPong device*. Essa non fa altro che utilizzare come *Pong device*, il *Ping device* stesso. Cioè, un dispositivo si scollega dal proprio GO ed automaticamente vi si ricollega, e così via all’infinito. Questo è utile per testare la funzionalità in caso di due soli Smartphone/Tablet a disposizione. Infatti, perché questa app possa essere provata in modo rilevante sono necessari cinque dispositivi, anche se ne consiglio almeno sei.

L’obiettivo di *PingPong* è quello di far sì che un Client, cioè il *PingPong device*, possa far parte di due gruppi contemporaneamente, senza che ciò avvenga realmente. Infatti, l’idea è di nascondere tale limite all’utente, facendo sì che il dispositivo si connetta e disconnetta ai GO dei due gruppi, in modo che possa inviare un file a due GO contemporaneamente.

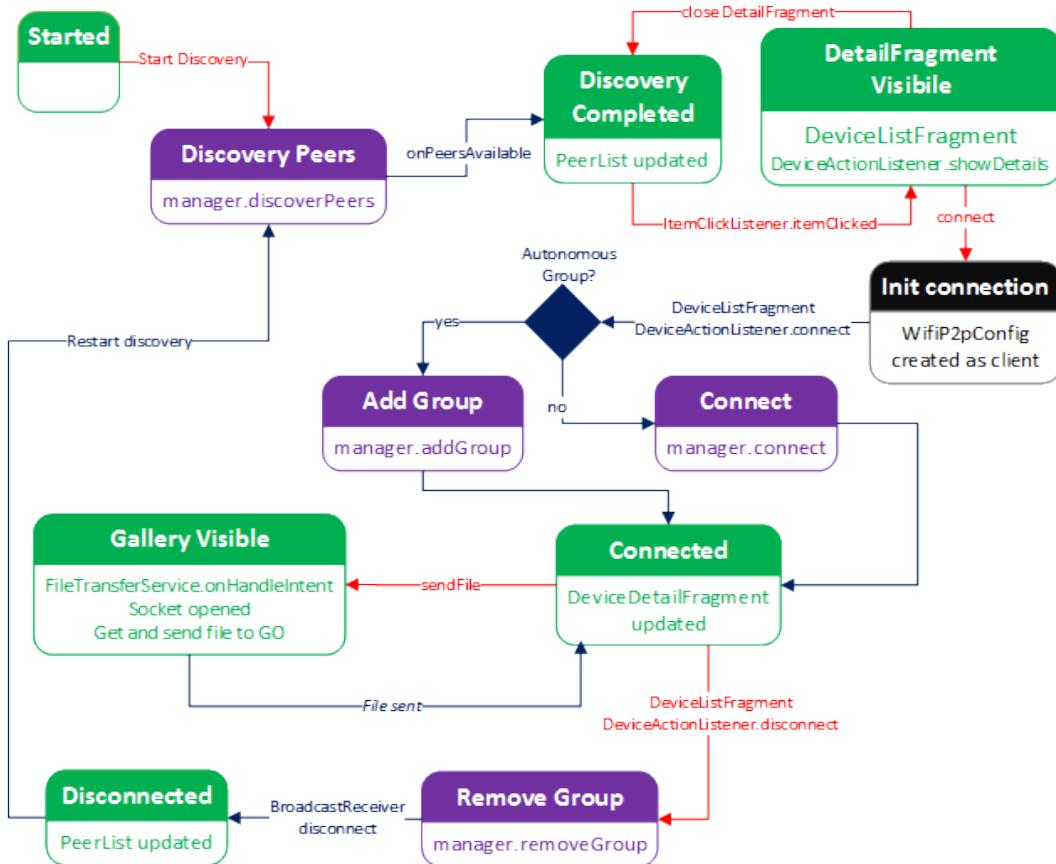


Figura 2.1: Diagramma a stati di PingPong in generale

In Figura 2.1 c'è il diagramma a stati di *PingPong*, con solamente la parte che gestisce le funzionalità base, mentre in Figura 2.2 c'è la sola parte che si occupa della logica del *rimbalzo*. La convenzione utilizzata è:

- in verde sono rappresentati gli stati in cui l'app aggiorna l'interfaccia grafica ed attende l'azione dell'utente;
- in nero le operazioni interne che appena terminate portano subito allo stato successivo;
- in viola sono gli stati legati a *Wi-Fi Direct*, cioè operazioni associabili alle fasi del protocollo e operazioni sui gruppi;
- le frecce rosse sono operazioni richieste esplicitamente dall'utente interagendo con l'interfaccia grafica.

Per rendere più facile l'estensione di quest'app ho deciso di creare un'astrazione degli oggetti *WiFiP2pDevice* e *WiFiP2pGroup* che ho chiamato rispettivamente *P2PDevice* e *P2PGroup*. L'obiettivo è far sì che possa aggiungere informazioni come la presenza in più gruppi che Android normalmente non permette. Inoltre, ho

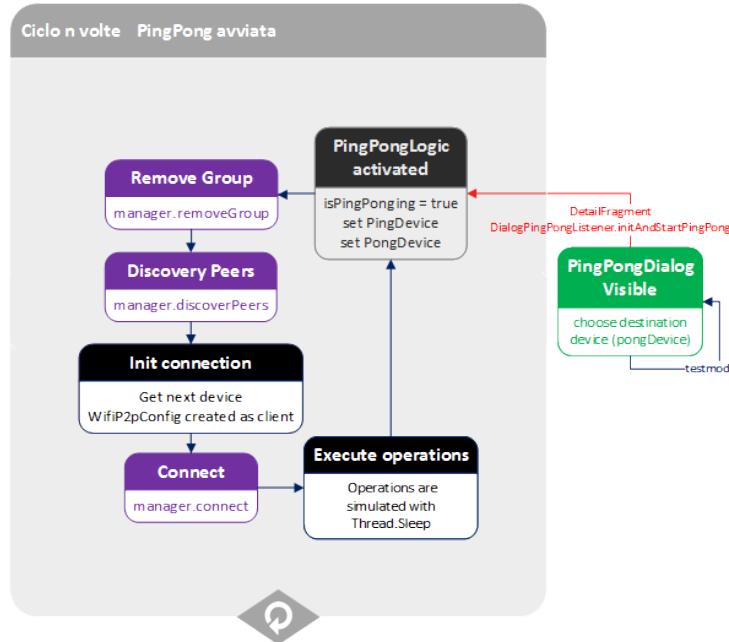


Figura 2.2: Diagramma a stati di PingPong nello specifico

realizzato le liste di questi elementi, come quella dei dispositivi nelle vicinanze (*PeerList*), i Client di un GO (*ClientList*), la *PingPongList* con i GO tra cui il *PingPong device*, ma soprattutto la lista *P2PGroups*, cioè quella dei gruppi a cui il dispositivo appartiene. Ovviamente, ogni *P2PGroup* definisce la lista dei suoi Client oltre a parametri aggiuntivi come l'indirizzo IP del GO. Poiché le API di Android non hanno questa funzione e per una questione di praticità, ho creato il concetto di *LocalP2PDevice*, cioè il riferimento al dispositivo P2P in uso. Oltre al *WiFiP2pDevice*, contiene anche due attributi utili per la gestione di un gruppo *Autonomous*, ma soprattutto quello per abilitare la *pingpong mode*. Poiché le classi: *ClientList*, *LocalP2PDevice*, *P2PGroups*, *PeerList* e *PingPongList* sono uniche e soprattutto devono essere accessibili da ogni package¹ dell'app, ho implementato il design pattern chiamato *Singleton*.

L'interfaccia grafica è composta da una sola *Activity* che implementa varie interfacce e gestisce tutti gli eventi. Questo perché *Wi-Fi Direct* è implementato con una logica basata su eventi. Essi possono essere intercettati utilizzando un *Broadcast Receiver* e implementando opportune interfacce. La grafica è estremamente modulare ed interamente basata su *Fragment* che sono caricati e mostrati a runtime dinamicamente. Tutti gli elementi hanno subito personalizzazioni grafiche per migliorarne l'aspetto e soprattutto utilizzare *Relative Layout* rendendoli adattabili al variare delle dimensioni dello schermo. Per supportare al meglio la nuova versione del

¹Un package ha lo scopo di riunire classi o entità analoghe logicamente correlate. Per maggiori dettagli vedi [http://it.wikipedia.org/wiki/Package_\(Java\)](http://it.wikipedia.org/wiki/Package_(Java)).

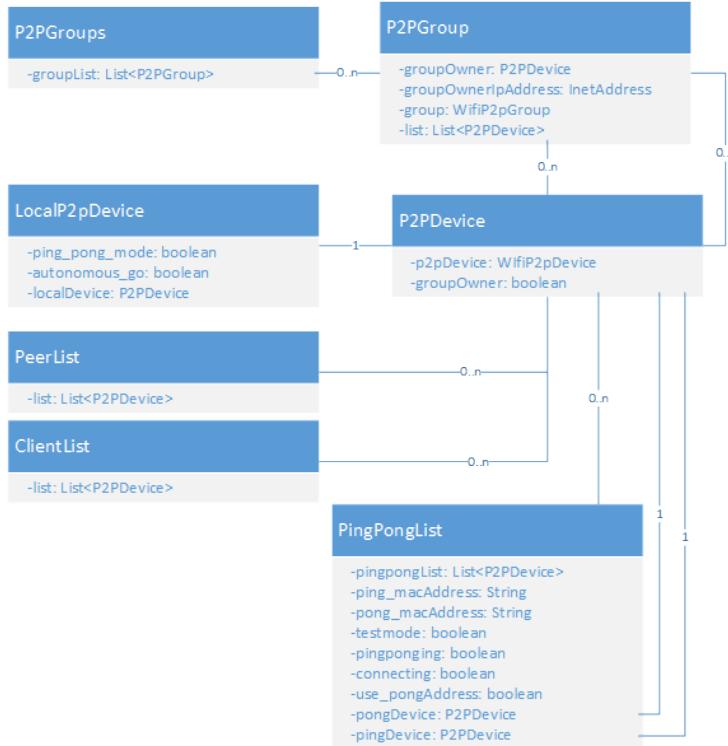


Figura 2.3: Diagramma UML delle classi di PingPong

sistema operativo, Android Lollipop, ho creato l'app in *Material Design*, sfruttando le più recenti API compatibili con KitKat, cioè incluse in *appcompat v7* e *v4*, tra cui *Toolbar*, *RecyclerView*, *CardView* (vedi Figura 2.4).

I *Fragment* che contengono liste di elementi, utilizzano *RecyclerView* e gli *Adapter* associati. Per mantenere la logica di *Wi-Fi Direct* separata dalla grafica, ho fatto in modo che i *Fragment* definiscano delle interfacce interne, che sono implementate dall'*Activity* che a sua volta definisce i metodi esposti dalle interfacce. Tale tecnica è detta *Callback Interface* e permette di non dover passare riferimenti di oggetti come per esempio l'*Activity*. In questo modo, ho fatto sì che *WiFiP2pManager*, cioè la classe principale che gestisce *Wi-Fi Direct*, sia definita solo nell'*Activity* e mai utilizzata in altre classi.

La logica che gestisce il *rimbalzo* si trova in un *AsyncTask*, con l'obiettivo di disaccoppiarla dal resto dell'app, in particolare dall'interfaccia grafica. Purtroppo, a causa di come sono state realizzate le API di *Wi-Fi Direct* in Android ed in particolare a causa della loro natura totalmente asincrona basata su eventi, è impossibile isolare in classi singole alcune funzioni. Di conseguenza, la logica che gestisce il *rimbalzo* si trova solo parzialmente nell'*AsyncTask*, che ho chiamato *PingPongLogic*. Per comprendere al meglio come ho realizzato la funzionalità in questione, consiglio di visionare il codice sorgente. In particolare, iniziando dal metodo *doInBackground* di *PingPongLogic*. Da esso si possono seguire facilmente tutti i metodi chiamati per

2. Realizzazione di due app in SDK

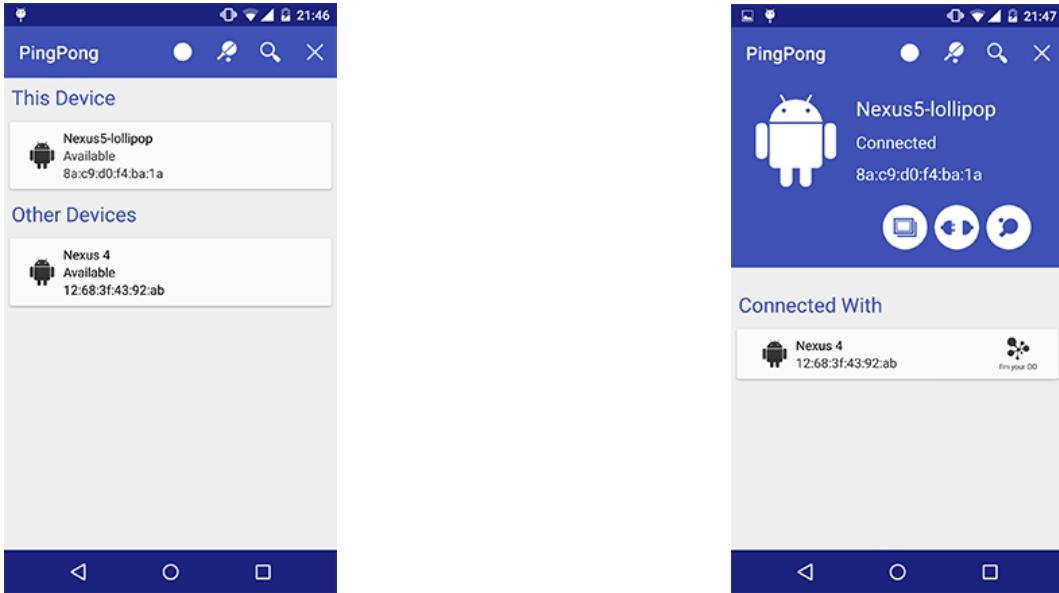


Figura 2.4: Interfaccia grafica dell'app PingPong

capire il flusso di eventi. Dato che la logica è asincrona, molti eventi non sono in sequenza, ma dipendono dall’attivazione di metodi definiti in interfacce gestite da Android. Quindi, ho aggiunto alcuni commenti per indicare dove continuerà l’esecuzione. Seguendo metodi e commenti si può “percorrere” tutto il ciclo di *rimbalzo* e ricominciarlo proseguendo all’infinito. Questa sequenza è esattamente quella che fa l’app e che ho rappresentato in molto semplificato in Figura 2.2. Faccio notare che ho dovuto inserire alcuni ritardi tra quelle operazioni per far sì che i dispositivi coinvolti nella *PingPong mode* riescano a sincronizzarsi a vicenda, cioè che le *fasi di Discovery* avvengano in instanti ravvicinati, tali per cui i dispositivi possano individuarsi e negoziare per la connessione. La *fase di Discovery* può essere considerata il “tallone di Achille” di questo protocollo, infatti il tempo per individuare i dispositivi è troppo elevato. A peggiorare la situazione entra in gioco l’implementazione di questo protocollo in Android, in cui dopo alcuni secondi dall’avvio della *fase di Discovery*, i dispositivi non sono più in grado di vedersi, nonostante la ricerca sia ancora in esecuzione. Non ho trovato nessun riferimento nella documentazione ad eventuali timeout in Android, ma secondo [73] è di 120 secondi. Dalla mia esperienza, ho notato che ci sono problemi anche con tempi molto inferiori.

Come si può vedere dalla *repository* ufficiale del progetto, il test su sei dispositivi ha comunque avuto successo, ma ciò ha richiesto alcuni tentativi. Per applicazioni reali, consiglio di non utilizzare tale tecnica. Proprio a causa delle difficoltà incontrate nella realizzazione di quest’app, ho creato il video (il cui link si trova nel file *readme* di *Github*) per dimostrarne il funzionamento, ma tale situazione, con una sincronizzazione così perfetta, non è facile da raggiungere ed ha richiesto la manipolazione di alcuni ritardi, fino al raggiungimento del risultato voluto. Un altro motivo

per cui ho utilizzato questi ritardi è che se l'esecuzione di operazioni come *connect*, *disconnect* e *discovery* sono fatte in modo consecutivo e troppo ravvicinato, tendono a fallire per motivi che non dipendono da me, ma dall'implementazione di Android.

2.2 Pigeon Messenger

In *PingPong* ho scelto un ambito in cui le performance sono importanti, ma cosa succede analizzando una situazione differente come la messaggistica tra utenti? La risposta è *Pigeon Messenger*, cioè un'app che permette ai dispositivi nelle vicinanze di comunicare tra di loro. Ogni dispositivo può gestire più chat contemporaneamente, dando l'illusione che le conversazioni avvengano nello stesso istante, sfruttando il fatto che il destinatario può impiegare alcuni secondi per rispondere. Questo, grazie al fatto che le performance non sono un requisito essenziale.

Poiché, l'obiettivo è illudere l'utente di essere connesso a più chat contemporaneamente, i messaggi che sono stati inviati nel momento in cui la connessione non è presente, devono essere accodati ed inviati appena torna disponibile. Inoltre, scrivere ed inviare un messaggio ad un utente deve corrispondere subito ad un tentativo di connessione con esso, per far sì che il messaggio, magari importante, sia recapitato il prima possibile. Ovviamente, se l'utente vuole fare ciò mentre sta chattando con un altro, l'app deve scollegarsi dal primo per collegarsi al secondo. Queste sono esattamente le funzionalità principali di *Pigeon Messenger*.

Durante la creazione di quest'app, ma in particolare durante la stesura della documentazione, ho utilizzato il termine “Eternal discovery” per indicare la funzione che ho realizzato per far sì che ogni azione che cambia lo stato di connessione con un *Peer*, riavvii il più rapidamente possibile la *fase di Discovery*. Considerando gli stessi problemi già affrontati con *PingPong*, cioè legati alla *fase di Discovery*, non ci sono state difficoltà rilevanti nella sincronizzazione. Quando ci si disconnette e ci si riconnette, i dispositivi nelle vicinanze sono trovati nuovamente.

Come per *PingPong*, ho definito *LocalP2PDevice* che rappresenta il dispositivo corrente, però ho aggiunto anche *P2PDestinationDevice*, cioè l'astrazione di un *P2PDevice* con il riferimento all'indirizzo IP del dispositivo a cui è collegato. Poiché, in quest'app sono permesse solo connessioni tra due dispositivi e non di gruppo, l'IP è sempre quello dell'altro dispositivo.

Ho realizzato l'interfaccia grafica nello stesso modo di *PingPong*, ove possibile riutilizzando gli elementi già creati, ma con una maggiore attenzione alla user-experience, dato che questa app potrebbe essere usata realmente dagli utenti. Per far sì che l'accesso alle funzioni, in particolare alle chat, fosse la più semplice e rapida possibile, ho scelto l'uso dei *tab*. Ovviamente, questa caratteristica rende la gestione dell'app più complessa per diversi motivi.

2. Realizzazione di due app in SDK

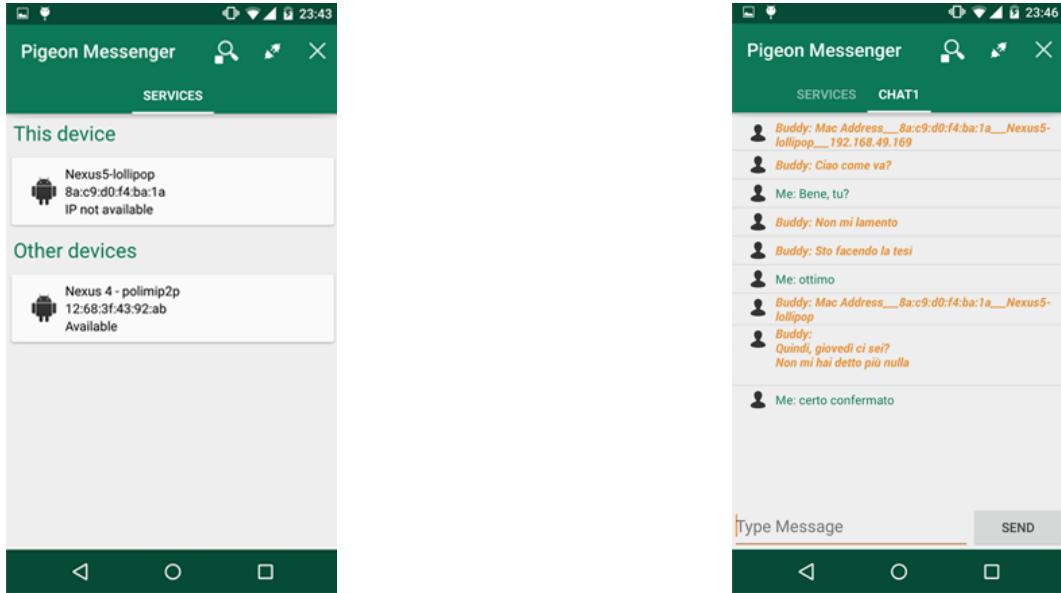


Figura 2.5: Interfaccia grafica dell'app Pigeon Messenger

- I *tab* richiedono l'uso di un *Fragment* principale con un *ViewPager* e un *Adapter* associato.
- I *tab* sono *Fragment* di diverso tipo. *WiFiP2pServicesFragment* è il primo e viene creato all'avvio dell'app, mentre gli altri sono *WiFiChatFragmentList*. Questo richiede un meccanismo per riassociare gli indici della lista dei *ChatFragment* e quella dei *tab*. Avrei potuto scambiare la posizione dei *Fragment*, ma questo non sarebbe stata una buona scelta dal punto di vista dell'usabilità dell'app.
- I *WiFiChatFragmentList* non sono creati staticamente, ma solo in base alla necessità, cioè quando inizia una nuova chat. Inoltre, non vi sono limiti sul numero di *Fragment* instanziabili.
- È necessario creare una logica che permette all'interfaccia grafica di mostrare automaticamente il *Fragment* della chat in corso, sia se appena creata, sia se già esistente.
- Per capire quale chat è associata ad un dispositivo e quindi usarlo come dispositivo di destinazione per la connessione, far sì che vi sia una lista di dispositivi e una logica per associarli ai *tab* (in modo da mantenere la separazione tra logica e grafica).
- Per far capire all'utente quale sia la chat attiva in corso, bisogna utilizzare icone o colori differenti.

In *Pigeon Messenger*, l'utilizzo delle API di *Wi-Fi Direct* è differente rispetto a *PingPong*, perché in quest'app ho utilizzato la funzione per gestire i servizi tramite il protocollo *Bonjour* (per maggiori informazioni fare riferimento alle specifiche [3]). A causa di questa caratteristica, la logica è più complessa, inoltre bisogna gestire più eventi. Soprattutto, l'interazione tra logica e grafica richiede l'uso di *Handler*, per evitare problemi ed eseguire troppe operazioni nel *Thread* dell'interfaccia grafica o doverlo invocare manualmente su grosse porzioni di codice. Questa tecnica, utilizzata anche nel codice sorgente di Android è ottima, perché permette, non solo di scambiare messaggi testuali, ma anche oggetti.

I colori e la logica con cui ho realizzato gli schemi nelle Figure 2.6 e 2.7 è la stessa di *PingPong*. Nella prima è rappresentato il diagramma degli stati. Cioè, una volta avviata l'app e creato il *TabFragment*, inizia la ricerca di altri servizi. Appena sono disponibili, l'interfaccia grafica è aggiornata. Una volta che l'utente esegue la connessione, viene inizializzata con un *WifiP2pConfig* personalizzato per far sì che chi avvia la connessione sia il Client. Di conseguenza, il *Broadcast Receiver* riceve la notifica e per mia scelta richiede le informazioni sulla connessione, in modo da ottenere gli indirizzi IP e distinguere un GO da un Client. Quindi, ho potuto creare il *SocketHandler* corretto, salvare l'indirizzo *IP* e mostrarlo nell'interfaccia grafica. Infine, raccolti tutti i dati tramite un messaggio inviato dall'*Handler* del *ChatManager*, ho sfruttato il meccanismo a scambio di messaggi in cui il GO informa il Client del suo nome, il MAC Address e l'indirizzo IP assegnato al Client, mentre il Client informa il GO del nome e del MAC address, come in Figura 2.8 (non ha bisogno dell'IP, perché un Client lo può ottenere facilmente dalle API Android, cosa che non avviene col GO). Lo scambio dell'IP è essenziale poiché Google non ha creato API per farlo.

Nel momento in cui i dispositivi leggono le informazioni ricevute dall'altro, possono capire chi è il dispositivo in questione, verificare se la chat è già stata avviata e quindi aggiornare l'interfaccia grafica in modo opportuno. A questo punto, l'utente può inserire messaggi nella chat. Nel caso di un evento di disconnessione, oltre alle azioni svolte dal *Broadcast Receiver*, c'è la procedura di *Eternal Discovery* che riavvia immediatamente la *fase di Discovery* riselectando automaticamente il *Fragment* della lista dei servizi. A questo punto, se l'utente sceglie un dispositivo a cui non si è mai collegato, ricomincia la procedura descritta poco fa, altrimenti, viene inviata la lista di tutti i messaggi, accodati prima che avvenisse la connessione e mostrando il *tab* corretto. Per brevità, in questo diagramma ho sostituito la procedura di connessione, di richiesta delle informazioni sulla connessione e di inizializzazione con il commento “is connected?”. Nella Figura 2.7 c'è una parte aggiuntiva della logica di questa app che spiega l'aggiunta di messaggi in coda quando la connessione non è ancora disponibile. Ciò può avvenire in seguito a un problema nella *fase di Discovery*, che può essere causato dall'irraggiungibilità o dalla richiesta dell'utente.

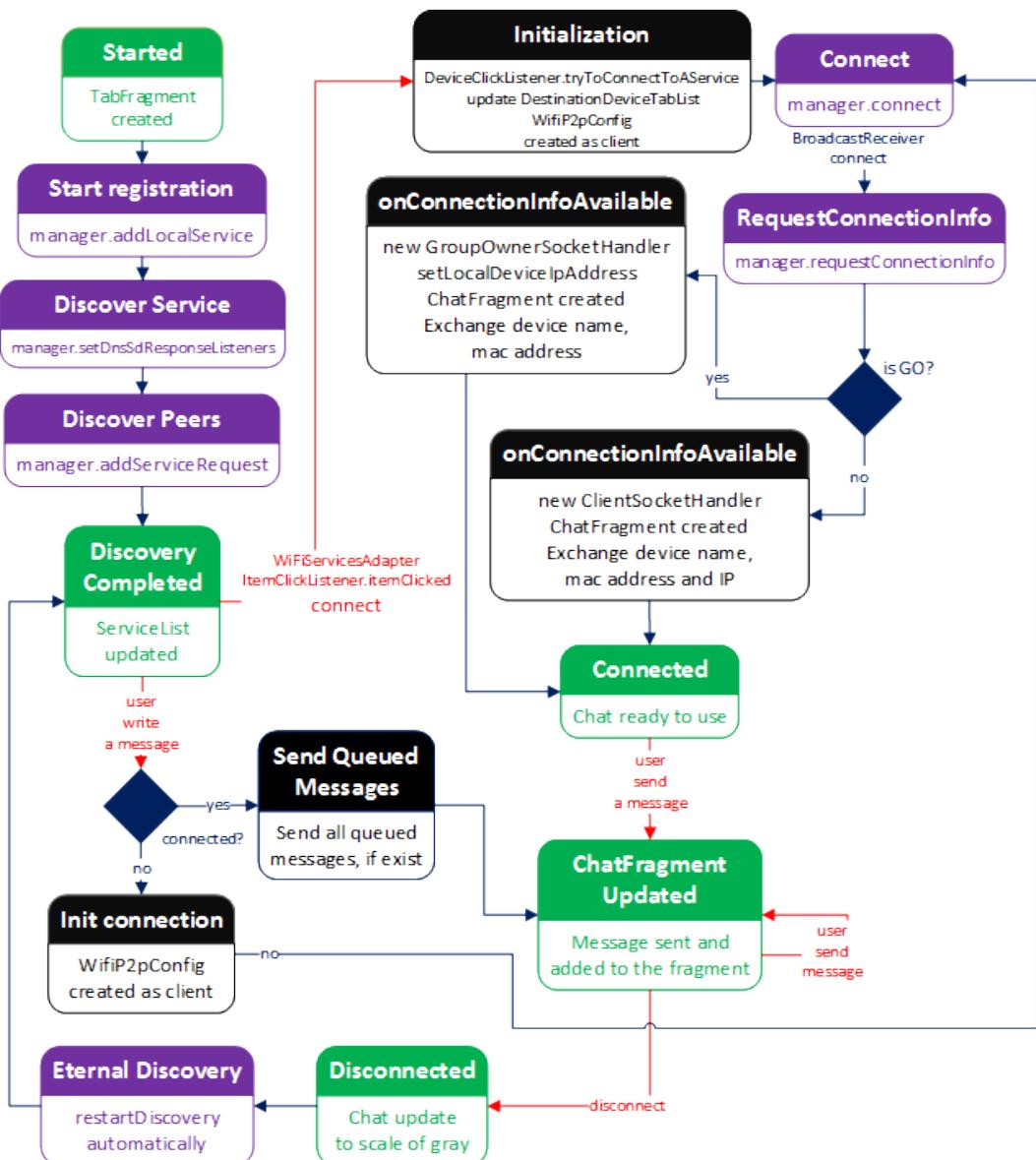


Figura 2.6: Diagramma a stati di Pigeon Messenger in generale

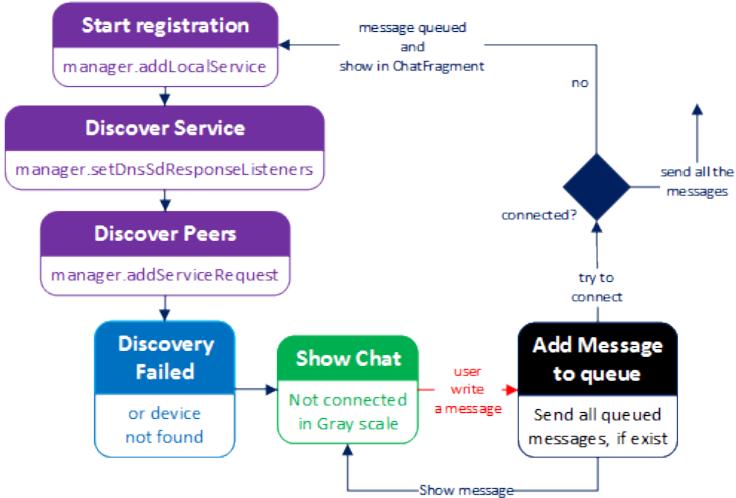


Figura 2.7: Diagramma a stati di Pigeon Messenger con coda messaggi

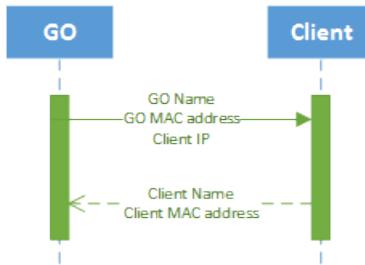


Figura 2.8: Diagramma di sequenza di Pigeon Messenger

Per completezza ho riportato in Figura 2.9 il diagramma delle classi, estremamente ridotto, concentrandomi soprattutto sull'uso di interfacce. I colori giallo e verde servono per mostrare la classe e l'interfaccia definita all'interno di tali classi, ma implementata da quella collegata tramite relazione con linea tratteggiata. Questo diagramma è utile per comprendere la struttura dei *Fragment*, ma soprattutto mettere in evidenza ciò che ho accennato prima, cioè che è l'*Activity* a gestire gli eventi *Wi-Fi Direct*. Per mantenere ancora di più i *Fragment* separati dalla logica del programma, ho fatto in modo che la gestione dei *SocketHandler* per la comunicazione non avvenisse in essi. Per non parlare di *ChatManager*, il quale è totalmente disaccoppiato dalla grafica, infatti si trova su un *Thread* separato. Nel *ChatManager* c'è la logica che gestisce l'invio dei messaggi dal *Thread* in questione all'interfaccia grafica, tramite gli *Handler* definiti nei *SocketHandler*. Anche in questo caso, lo schema è una semplificazione per mostrare gli elementi fondamentali di quest'app.

2. Realizzazione di due app in SDK

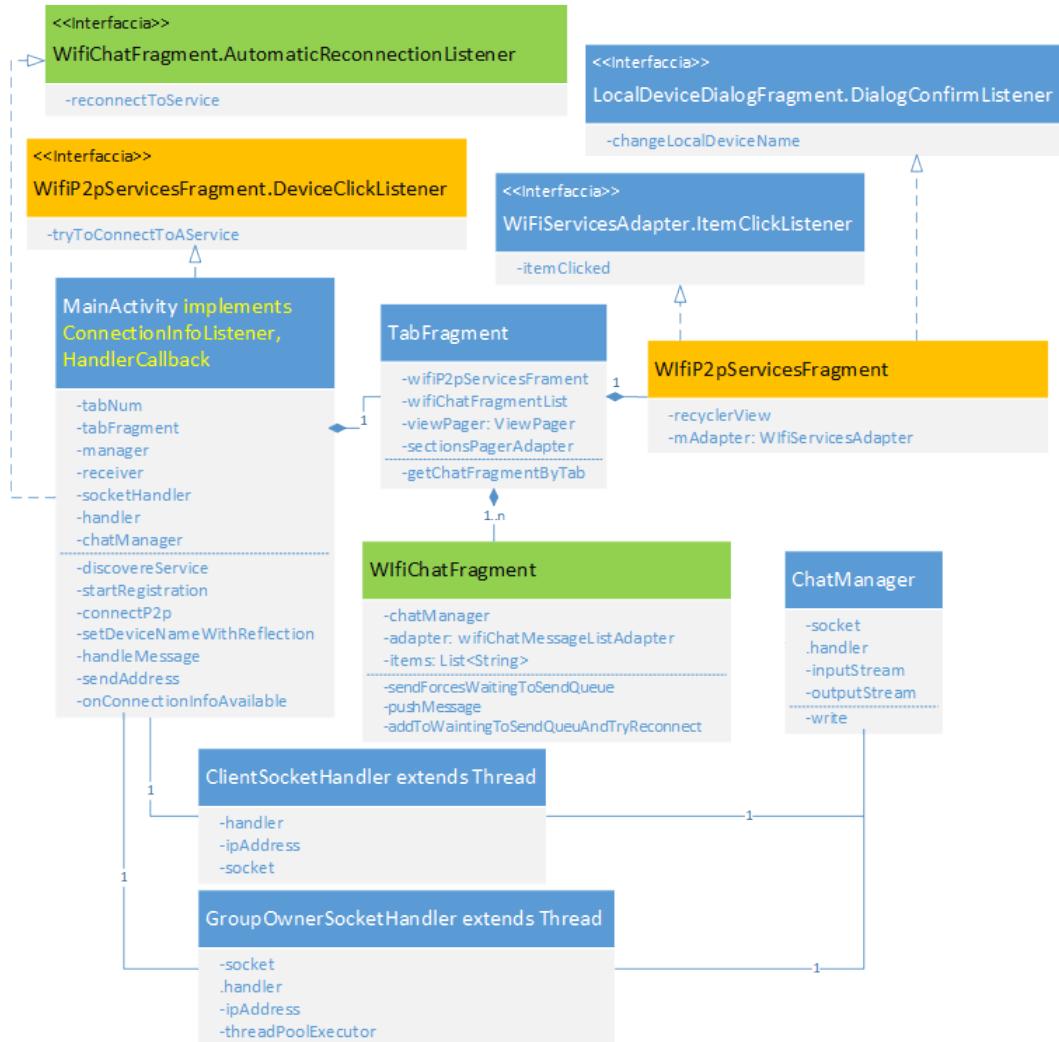


Figura 2.9: Diagramma UML delle classi di Pigeon Messenger

Capitolo 3

Esplorazione del codice sorgente

Nell'introduzione ho spiegato quali sono gli obiettivi e come raggiungerli dal punto di vista teorico, mentre nel Capitolo 2 ho realizzato due app per studiare i limiti dell'implementazione attuale di *Wi-Fi Direct* in Android. Ciò che ho appreso è che le API dell'SDK in Android sono estremamente limitate, in particolare vi sono metodi mancanti, ma soprattutto problemi nella gestione della *fase di Discovery* del protocollo.

Per raggiungere gli obiettivi prefissati, le API fornite da Google non sono sufficienti, infatti non permettono che un dispositivo possa fare parte di più gruppi *Wi-Fi Direct* contemporaneamente, come già spiegato in [26]. *Wi-Fi Alliance* non vieta questo nelle specifiche, ma allo stesso tempo non fornisce nemmeno dettagli specifici al riguardo. Purtroppo, Google non ha implementato tale funzionalità nel sistema operativo. Nell'introduzione ho già spiegato che per raggiungere tale obiettivo è necessario gestire manualmente le interfacce di rete. Dato che non esistono API per questo, ho deciso di analizzare il codice sorgente del sistema operativo, per capire dove sono i limiti in questione. Cioè, ho studiato l'implementazione del package `android.net.wifi.p2p`. Esso è costituito da un insieme di classi generiche per *Wi-Fi Direct* e altre più specifiche (nel sotto-package `nsd`) che si riferiscono a *Network Service Discovery* (NSD) sfruttando i protocolli *Universal Plug and Play (UPnP)*¹ e *Bonjour*². Ho già utilizzato sia le funzioni nel package, sia quelle nel sotto-package, nella realizzazione delle due app nel Capitolo 2, ma affrontando il problema ad alto livello. Ora le analizzerò dal punto di vista del codice sorgente, fornendo anche alcuni dettagli su Android *Native Development Kit (NDK)* [33].

¹Universal Plug and Play (UPnP) è un protocollo di rete per permettere a diversi terminali di connettersi l'uno all'altro. Il termine UPnP deriva da “Plug and play” cioè “Inserisci e utilizza” [108].

²Bonjour è un marchio registrato da Apple per indicare una tecnologia nata per individuare automaticamente la presenza di servizi nelle reti domestiche [90].

3.1 Visibilità dei metodi Java in Android

Prima di procedere, è bene precisare che Android è costituito principalmente da classi Java, file C e C++. L'SDK di Android fornisce agli sviluppatori di app solo i metodi Java creati da Google. Per utilizzare funzioni più a basso livello è necessario usare l'NDK. In questa sezione mi occuperò solamente dal primo, ma nel proseguo del capitolo avrò modo di affrontare alcuni dei concetti più importanti, con l'aiuto di alcuni esempi. Per quanto riguarda i metodi Java di Android, si possono classificare in tre modi:

1. *esposti* ed accessibili agli sviluppatori;
2. *nascosti* tramite l'*annotation*³ `@hide`, utilizzati solo nel sorgente di Android;
3. *interni*, cioè nel package `com.android.internal` [25].

I primi sono quelli accessibili e documentati ufficialmente da Google, i secondi sono utilizzati per svolgere funzioni interne ad altri metodi e non sono invocabili direttamente, mentre gli ultimi sono interni, cioè usati dal sistema operativo. L'uso dell'*annotation* `@hide` è una particolarità di Android, infatti in Java non esiste. Essa è usata da Google per compilare l'SDK esponendo solo alcune delle funzioni agli sviluppatori. Quindi può essere considerata come un sistema per gestire la visibilità agli sviluppatori, senza modificarla nel codice sorgente di Android.

Ricordo brevemente che i metodi *private* di Java non sono inclusi nel *Javadoc* [74] e allo stesso tempo sono accessibili solo dalla classe stessa. Al contrario, per esempio, i *public* sono accessibili ovunque e sono inclusi nella documentazione. Google ha deciso di utilizzare la visibilità esattamente come in Java, ma nascondere alcuni di questi metodi *public* (e non solo) nella documentazione e nell'SDK. L'alternativa sarebbe stata usare quelli *private*, ma ciò avrebbe potuto creare problemi nell'invocazione dei metodi all'interno del sorgente di Android.

Quindi, l'*annotation* `@hide`, al di sopra di un metodo *public*, lo nasconde allo sviluppatore di app, ma fa sì che Google possa invocarlo liberamente da ogni package e classe. Infatti, `@hide` ottiene la massima priorità nello stabilire se tale metodo/attributo/classe debba essere incluso nella documentazione ed essere accessibile. Detto ciò, è facilmente intuibile che uno sviluppatore non possa invocare metodi `@hide` in quanto nascosti, mentre il sistema operativo sì.

In realtà, esistono alcune tecniche che ho sperimentato per poter invocare metodi di qualunque tipo.

1. Modificare il file “.jar” dell'SDK di Android per rendere accessibili i metodi `@hide`. Per ottenere ciò, non è richiesta la modifica del codice sorgente del

³Una annotazione Java è un modo per aggiungere metadati nel codice sorgente Java, che possono essere disponibili al programmatore durante l'esecuzione [89].

sistema operativo, perché questi metodi Java sono rimossi solamente nell'SDK e non in Android. Quindi, è sufficiente ottenere il file “.jar” che contiene tutte le API usate in Android e sostituirlo con quello nell'SDK sul proprio PC/MAC. Per fare ciò, è sufficiente compilare il codice sorgente di Android (procedura che descriverò nel dettaglio nel Capitolo 4) e importare, come libreria esterna in Eclipse o in un IDE equivalente, il file `out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/classes.jar`. Questo permette di far riconoscere all'IDE i metodi `@hide` e compilare il progetto. Perché ciò funzioni, bisogna far si che `classes.jar` abbia priorità maggiore di `android.jar` (file di default), cioè per esempio in Eclipse si deve trovare più in alto nella scheda “Order and Export”, come in Figura 3.1.

2. *Java reflection* [96]: si tratta della soluzione migliore, perché permette di invocare sia i metodi *interni* sia quelli `@hide`, senza modificare l'SDK, ma soprattutto senza creare problemi di compatibilità, legati alla versione di Android da cui è stato estratto il file `classes.jar`. Ho utilizzato questa metodologia anche nel Capitolo 2.

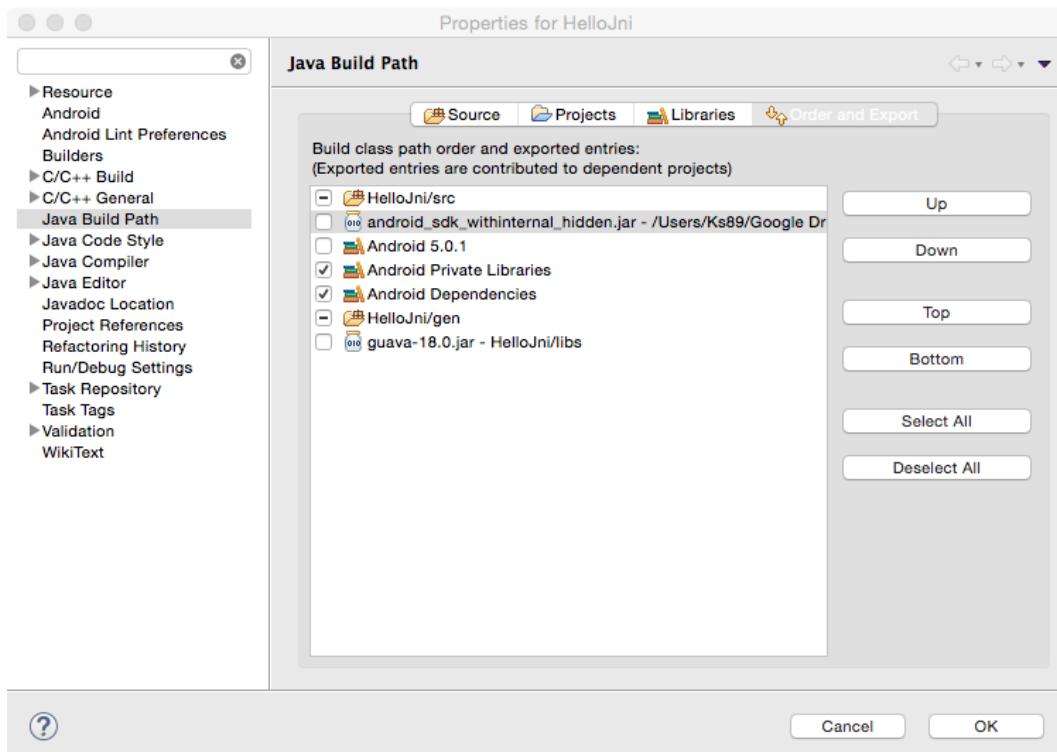


Figura 3.1: Order and Export di Eclipse con SDK modificato

3.2 Estendere il package android.net.wifi.p2p

Per raggiungere gli obiettivi ho studiato il codice sorgente di Android, in particolare quello del package *android.net.wifi.p2p*. Tale operazione ha messo in evidenza una situazione molto preoccupante, poiché nessuna delle funzioni a me necessarie è stata implementata. Il primo approccio al problema è stato quello di estendere il package per implementare le funzionalità mancanti. Ma anche ipotizzando di usare la *Java reflection* per re-implementare metodi *interni* o *@hide*, Android continuerrebbe ad usare la versione originale e non quella creata da me. Inoltre, il codice di questo package risulta difficilmente estendibile, come mostrerò nella Sezione 3.3.

Un altro fatto da considerare è che spesso questi metodi sono *nativi*, cioè sono solo le dichiarazioni di metodi Java, che tramite JNI, chiamano funzioni C++ e ciò richiede di personalizzare dei file C++ di sistema, non modificabili tramite *Java reflection*⁴.

Il secondo approccio è stato quello di estendere completamente *android.net.wifi.p2p*, importando il sorgente nel progetto, rinominando i package e le classi in modo da avere una propria implementazione, ma ciò richiede troppe modifiche, infatti bisogna addirittura personalizzare la classe *Activity* per poter estendere *android.net.wifi.p2p*. Questo perché tali classi sono tutte legate tra loro e non sono pensate per essere modificate così.

Per i motivi descritti in questa sezione, ho deciso di cambiare approccio e tentare di modificare il sorgente di Android e di conseguenza ricompilare l'intero sistema operativo, creando cioè una *Custom ROM di Android* o *Custom Firmware* [88].

3.3 Modificare il package android.net.p2p

Il codice sorgente di Android è *Open source* [48] e *android.net.wifi.p2p* si trova in [31]. Analizzando il package ho notato che la classe principale che si occupa di gestire le interfacce di rete è chiamata *WiFiNative.java*.

Per fare ricerche nel sorgente di Android si è rivelato molto utile il comando *grep -r* di Linux, ma ancora di più il sito web *Grepcode* [49]. Infatti, con quest'ultimo mi sono accorto che da Android KitKat a Lollipop molte classi interne sono state spostate, per esempio *WiFiNative.java* in KitKat era in [50], ora si trova in [51]. Analizzando attentamente questa classe di Lollipop, perché più aggiornata, ho notato che:

- non c'è un *ArrayList* di interfacce di rete, infatti il nome dell'interfaccia è nella variabile *interfaceName*;

⁴ *Java Native Interface (JNI)* è un framework che permette di richiamare da Java metodi *nativi*, cioè scritti in altri linguaggi di programmazione, come per esempio C, C++ e assembly. Principalmente serve per chiamare funzionalità intrinsecamente non portabili e che pertanto non posso essere scritte in Java [97].

- contiene molti metodi dichiarati come *nativi*, cioè usa JNI per chiamare funzioni C/C++. Infatti, è definito il caricamento della libreria *nativa* libwifi-service.so in un blocco di codice statico (vedi Algoritmo 3.1), prima dell'*onCreate* della classe Activity. Questa libreria contiene codice *nativo* C/C++ (parlerò più nel dettaglio di questi concetti nella Sezione 3.4).

Algoritmo 3.1 Blocco di codice statico per caricare con NDK le librerie native

```
/* Register native functions */
static {

    /* Native functions are defined in libwifi-service.so */
    System.loadLibrary("wifi-service");
    registerNatives();

}
```

Nell’Algoritmo 3.2 ho riportato il costruttore di WiFiNative.java, in cui il nome dell’interfaccia è passato come parametro. Tramite la funzionalità “Find Usage” di Grepcode [32], si può vedere che questa classe è istanziata in com.android.server.wifi.WifiStateMachine e com.android.server.wifi.p2p.WifiP2pServiceImpl.P2pStateMachine.

Algoritmo 3.2 Costruttore della classe WiFiNative

```
public WiFiNative(String interfaceName) {
    mInterfaceName = interfaceName;
    mTAG = "WifiNative-" + interfaceName;
    if (!interfaceName.equals("p2p0")) {

        mInterfacePrefix = "IFNAME=" + interfaceName + " ";
    } else {

        // commands for p2p0 interface don't need prefix
        mInterfacePrefix = "";
    }
}
```

A sua volta WifiStateMachine.java è istanziata in com.android.server.wifi.WifiServiceImpl.java. In essa si trova la variabile *interfaceName*, la quale è inizializzata con *mInterfaceName = SystemProperties.get("wifi.interface", "wlan0")*. Analizzare il funzionamento di questa classe non è il mio obiettivo, perché mi sto concentrando su Wi-Fi Direct e non sul classico Wi-Fi. Infatti, mi interessa l’interfaccia P2P chiamata *p2p0* e non *wlan0*. La classe rilevante è *P2pStateMachine*, il cui costruttore

Algoritmo 3.3 Costruttore della classe WifiP2pServiceImpl

```
public WifiP2pServiceImpl(Context context) {  
  
    mContext = context;  
    //STOPSHIP: get this from native side  
    mInterface = "p2p0";  
    mNetworkInfo = new NetworkInfo(  
  
        ConnectivityManager.TYPE_WIFI_P2P, 0, NETWORKTYPE, "");  
    mP2pSupported = mContext.getPackageManager().  
  
        hasSystemFeature(PackageManager.FEATURE_WIFI_DIRECT);  
    mThisDevice.primaryDeviceType = mContext.getResources().  
        getString(  
            com.android.internal.R.string.config_wifi_p2p_device_type);  
    HandlerThread wifiP2pThread = new  
        HandlerThread("WifiP2pService");  
    wifiP2pThread.start();  
    mClientHandler = new ClientHandler(wifiP2pThread.getLooper());  
    mP2pStateMachine = new P2pStateMachine(TAG,  
        wifiP2pThread.getLooper(), mP2pSupported);  
    mP2pStateMachine.start();  
}
```

è nell'Algoritmo 3.3. Come si può notare, Google inizializza la variabile *mInterface* con il valore costante “*p2p0*”. Però dal commento subito sopra, si capisce che è solo temporanea e che deve essere ottenuta tramite apposite chiamate alla parte nativa in C/C++. Ciò dimostra come l'implementazione di *Wi-Fi Direct* in Android sia ancora in corso ed incompleta.

Ciò che ho detto fino a questo punto mostra un problema, cioè che le classi per gestire le interfacce di rete sono estremamente specifiche e legate ai limiti riscontrati nel Capitolo 1. Se in più si analizza la classe in cui queste interfacce vengono utilizzate realmente, cioè si ritorna a studiare *WifiNative.java*, si può notare come in tutto il codice sorgente i nomi delle interfacce siano inseriti direttamente nelle condizioni degli *if-else*. Questa situazione mostra quanto sia difficile estendere il package e la necessità di modificare direttamente il codice di Android. È bene precisare che la situazione è migliorata rispetto a KitKat, infatti nelle versioni precedenti, c'è stato un uso ancora più massiccio di costanti “*wlan0*” e “*p2p0*” all'interno del codice. Per esempio, si può notare che in Lollipop sono stati aggiunti i metodi negli Algoritmi 3.4 e 3.5, che mostra come Google sia effettivamente interessata ad estendere queste funzionalità, anche se ciò sta richiedendo veramente molto tempo.

Algoritmo 3.4 Metodo che scansiona le interfacce di rete disponibili

```
private static native int getInterfacesNative();
synchronized public static int getInterfaces() {

    synchronized (mLock) {
        if (sWifiIfaceHandles == null) {
            int num = getInterfacesNative();
            int wifi_num = 0;
            for (int i = 0; i < num; i++) {

                String name = getInterfaceNameNative(i);
                Log.i(TAG, "interface[" + i + "] = " + name);
                if (name.equals("wlan0")) {
                    sWlan0Index = i;
                    wifi_num++;
                } else if (name.equals("p2p0")) {
                    sP2p0Index = i;
                    wifi_num++;
                }
            }
            return wifi_num;
        } else {
            return sWifiIfaceHandles.length;
        }
    }
}
```

Algoritmo 3.5 Metodi per ottenere i nomi delle interfacce, specificandone l'indice

```
private static native String
getInterfaceNameNative(int index);

synchronized public static String
getInterfaceName(int index) {
    return getInterfaceNameNative(index);
}
```

Algoritmo 3.6 Classe com_android_server_wifi_WifiNative.cpp

```

/*
 * JNI registration.
 */
static JNINativeMethod gWifiMethods[] = {
    /* name, signature, funcPtr */
    { "loadDriver", "()Z", (void *)
        android_net_wifi_loadDriver },
    (...)

    { "getInterfacesNative", "()I", (void*)
        android_net_wifi_getInterfaces},
    { "getInterfaceNameNative", "(I)Ljava/lang/String;",
        (void*) android_net_wifi_getInterfaceName},
    (...)

}

int register_android_net_wifi_WifiNative(JNIEnv* env) {
    return AndroidRuntime::registerNativeMethods(env,
        "com/android/server/wifi/WifiNative",
        gWifiMethods, NELEM(gWifiMethods));
}

/* User to register native functions */
extern "C"
jint Java_com_android_server_wifi_WifiNative_registerNatives
(JNIEnv* env, jclass clazz) {
    return AndroidRuntime::registerNativeMethods(env,
        "com/android/server/wifi/WifiNative", gWifiMethods,
        NELEM(gWifiMethods));
}

```

Per completezza riporto negli Algoritmi 3.6 e 3.7 anche il metodo *nativo* *getInterfaceNameNative(int index)*. L’implementazione in C++ per ottenere i nomi delle interfacce di rete, chiamato dalla classi Java con JNI, si trova nel codice sorgente di Android in */frameworks/opt/net/wifi/service*, nel file *com_android_server_wifi_WifiNative.cpp*. Essendo una classe C++, è necessario eseguire le registrazioni JNI dei metodi, come mostrato nell’Algoritmo 3.6. Per brevità ho rimosso alcune righe codice, sostituendole con “(...”).

Nonostante non faccia espressamente parte di questo lavoro di tesi, analizzando la classe *com_android_server_wifi_WifiNative.cpp*, mi sono accorto che Google ha introdotto le funzionalità per ottenere informazioni sul *Received Signal Strength Indication* (RSSI), questo può essere molto utile per la localizzazione di dispositivi in ambienti chiusi, ottenendo la distanza precisa tra il router e il dispositivo stesso. E’ importante perché in [53] è stato messo in evidenza il fatto che Google ha forzato il parametro dell’RSSI a un valore costante, lasciando un commento per indicare che verrà implementato in futuro, cioè una situazione molto simile a quella dell’Algorit-

Algoritmo 3.7 Metodo in C++ per ottenere i nomi delle interfacce, specificandone l’indice

```
static jstring android_net_wifi_getInterfaceName(JNIEnv*  
    *env, jclass cls, jint i) {  
  
    char buf [EVENT_BUF_SIZE];  
    jlong value = getStaticLongArrayField(env, cls,  
        WifiIfaceHandleVarName, i);  
    wifi_interface_handle handle =  
        (wifi_interface_handle) value;  
    int result = ::wifi_get_iface_name(handle, buf,  
        sizeof(buf));  
    if (result < 0) {  
        return NULL;  
    } else {  
        return env->NewStringUTF(buf);  
    }  
}
```

mo 3.3. Il metodo in questione, realizzato completamente in C++ è nell’Algoritmo 3.8. Purtroppo, ciò non è disponibile per quanto riguarda la parte di *Wi-Fi Direct*, infatti, invece di chiamare l’apposito metodo per ottenere il corretto valore di RSSI, Google ha lasciato la stessa inizializzazione ad una costante con il commento *TODO*, già descritta in [53], come si può vedere nell’Algoritmo 3.9. Quindi, non vi è stato nessun miglioramento per quanto riguarda la parte di *Wi-Fi Direct*.

3.4 Utilizzare NDK

Oltre ai metodi Java e le funzioni C/C++, Android è costituito da un programma chiamato *wpa_supplicant* a cui si appoggia totalmente per quanto riguarda ogni operazione che deve svolgere il chip Wi-Fi del dispositivo. Questo vuol dire che tutto quello che ho mostrato, ad un certo punto, sfrutterà la comunicazione tra processi di Linux per “parlare” con *wpa_supplicant*. In questa sezione non descriverò cos’è nel dettaglio o come funziona, poiché lo farò nel Capitolo 5, ma mi concentrerò su un esempio di come utilizzare NDK per chiamare le funzioni di *wpa_supplicant*. Infine, valuterò se questa alternativa è adeguata per raggiungere gli obiettivi prefissati, oppure no. Anticipo subito che c’è un problema, infatti non tutte le funzioni C di *wpa_supplicant* sono invocabili tramite Android NDK. Per capire ciò, faccio un esempio creando un’app con NDK in Eclipse.

Algoritmo 3.8 Funzione C++ per ottenere i dati su RSSI

```
static void onRttResults(wifi_request_id id, unsigned
    num_results, wifi_rtt_result results[]) {
    JNIEnv *env = NULL;
    mVM->AttachCurrentThread(&env, NULL);
    (...)

    for (unsigned i = 0; i < num_results; i++) {
        wifi_rtt_result& result = results[i];
        jobject rttResult = createObject(env,
            "android/net/wifi/RttManager$RttResult");
        (...)

        setIntField(env, rttResult, "rssи", result.rssi);
        setIntField(env, rttResult, "rssи_spread",
            result.rssi_spread);
        setIntField(env, rttResult, "tx_rate",
            result.tx_rate.bitrate);
        (...)

        setIntField(env, rttResult, "distance_cm",
            result.distance);
        setIntField(env, rttResult, "distance_sd_cm",
            result.distance_sd);
        (...)

        env->SetObjectArrayElement(rttResults, i, rttResult);
    }
}
```

Algoritmo 3.9 Inizializzazione RSSI per un dispositivo Wi-Fi Direct

```
public WifiP2pPeer(Context context, WifiP2pDevice dev) {
    super(context);
    device = dev;
    setWidgetLayoutResource(
        R.layout.preference_widget_wifi_signal);
    mRssi = 60; //TODO: fix
}
```

3.4.1 Esempio di app NDK per wpa_supplicant

Ho creato un nuovo progetto NDK in Eclipse ed ho usato i *makefile*⁵ degli Algoritmi 3.10 e 3.11 nella cartella *jni* del progetto. Nel primo algoritmo sono definiti i *moduli* da compilare, nel secondo le architetture verso cui *cross-compilare*. I dettagli specifici dei termini *cross-compilare*, *modulo* e *makefile* saranno spiegati in dettaglio nel Capitolo 4.

Algoritmo 3.10 Android.mk in Eclipse

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
#il nome local_module che dico qui è quello che poi chiamerò
#in java System.loadLibrary("hello-jni");
LOCAL_MODULE      := hello-jni
LOCAL_SRC_FILES  := hello-jni.c
LOCAL_SHARED_LIBRARIES := wpa_client
include $(BUILD_SHARED_LIBRARY)
include $(CLEAR_VARS)
LOCAL_MODULE := wpa_client
LOCAL_SRC_FILES := libwpa_client_k.so
include $(PREBUILT_SHARED_LIBRARY)
```

Algoritmo 3.11 Application.mk in Eclipse

```
APP_ABI := armeabi armeabi-v7a
```

Per ora, mi voglio concentrare solo sul fatto che l’Algoritmo 3.10 definisce due moduli, il primo, chiamato *hello-jni*, è quello principale che ne include un secondo col nome *wpa_client*. Quest’ultimo usa il “*template PREBUILT*”, cioè si tratta di un modulo già compilato ed incluso nel progetto. Più semplicemente, lo si può pensare come ad una libreria esterna, che ho importato nel progetto NDK, in cui voglio utilizzarne le funzioni. Infatti, si tratta di quella collegata a *wpa_supplicant*. Invece, il primo modulo è quello principale che usa il “*template BUILD*” ed implementa metodi *nativi*, che nel mio caso è uno solo e si chiama “*provawpa*”. Essi sono definiti nel file Java nell’Algoritmo 3.12. Parlerò più avanti del concetto di “*template di un Android.mk*”, poichè ora non è particolarmente rilevante, quindi rimando questo argomento al Capitolo 4.

⁵GNU Make è un programma per generare eseguibili dal codice sorgente. Make capisce come compilare questo codice usando file chiamati “.makefile” [28].

Algoritmo 3.12 Esempio Activity di un'app con NDK

```
public class HelloJni extends Activity {  
  
    private String instanceField = "Instance Field";  
    private static String staticField = "Static Field";  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        Log.d(TAG, "calling jni method");  
        this.provawpa()  
    }  
    public native String provawpa();  
    private String instanceMethod() {  
        return "Instance Method";  
    }  
    private static String staticMethod() {  
        return "Static Method";  
    }  
    static {  
        System.loadLibrary("wpa_client");  
        System.loadLibrary("hello-jni");  
    }  
}
```

A questo punto, ho dovuto generare i file “.h” nella cartella *jni*, ma ciò non è un problema perché possono essere creati tramite *javah* in modo automatico, addirittura configurando Eclipse per eseguire i corretti comandi come un profilo di “External Tools Configurations”. La configurazione in MAC OS X 10.10 che ho utilizzato è nell’Algoritmo 3.13. In *Location* ho specificato il percorso completo di *javah*, che si trova nell’SDK. In *Working Directory* ho specificato la cartella con i file “.class” del progetto ed infine, in *Arguments* i parametri da passare a *javah* per poter creare i file “.h” associati ai “.c” nella cartella *jni*. Ho citato questa configurazione poiché le versioni sui libri di riferimento non sono risultate funzionanti. L’unico elemento che non ho ancora mostrato è la parte in C, cioè la funzione che l’*Activity* chiama tramite il metodo *public native String provawpa()*.

Per una questione di completezza ho deciso di aggiungere anche l’Algoritmo 3.16 in coda a questo capitolo. Sebbene non sia parte dell’app per *wpa_supplicant*, mostra alcuni esempi interessanti d’invocazioni di metodi Java dal codice C. Infatti, una volta ottenuto il riferimento alla classe Java nella variabile JNI chiamata “*clazz*”, ho potuto ottenere gli attributi statici e non statici della classe. Nella parte restante dell’algoritmo, ho fatto lo stesso, ma questa volta con due metodi, uno statico e uno non statico ed infine, ho restituito una stringa alla classe Java. Per i dettagli ed

Algoritmo 3.13 External Tools Configurations di Eclipse per eseguire javah

Location: /Library/Java/JavaVirtualMachines/

jdk1.7.0_71.jdk/Contents/Home/bin/javah

Working Directory:

\${workspace_loc:/HelloJni/bin/classes}

Arguments: -verbose -jni -o \${workspace

loc:/HelloJni/jni}/hellojni.h -classpath
/Users/Ks89/android-sdk-macosx/platforms
/android-19/android.jar:\${workspace_loc:
/HelloJni/bin/classes} com.example.hellojni.HelloJni

Algoritmo 3.14 Metodo *nativo provawpa* implementato in C con JNI

```
(...)
//extern perche' e' una funzione chiamata nel file
//libwpa_client_k.so
//specificato da noi in Android.mk
extern struct wpa_ctrl * wpa_ctrl_open(
    const char *ctrl_path);

//metodo jni in c che corrisponde a provawpa in
//java, nel package
//com.example.hellojni.HelloJni.java. Perché
//funzioni deve avere
//il nome seguendo questo specifico formato.
//jstring è il tipo string di JNI, mentre
//thiz e' l'instance (sui libri di riferimento,
//non si usa this, ma in genere
//la parola "thiz" nel codice C, per indicare il
//riferimento all'oggetto che ha chiamato il
//metodo in questione)
jstring Java_com_example_hellojni_HelloJni_provawpa(
    JNIEnv* env, jobject thiz ) {

    //esempio di chiamata di una funzione in un
    //file .so importato
    //ovviamente, in questo esempio non fa nulla,
    //in quanto vengono passati dati casuali ed
    //insignificanti
    wpa_ctrl_open((env)->NewCharArray(env,3));
    return (*env)->NewStringUTF(env, "metodo");

}
```

Algoritmo 3.15 Risultato di readelf di libwpa_client.so

```
(...)
3: 00001491    440 FUNC  GLOBAL DEFAULT  8 wpa_ctrl_open
(...)
25: 00001649     34 FUNC  GLOBAL DEFAULT  8 wpa_ctrl_close
26: 0000166d    176 FUNC  GLOBAL DEFAULT  8 wpa_ctrl_cleanup
(...)
31: 0000171d    290 FUNC  GLOBAL DEFAULT  8 wpa_ctrl_request
(...)
40: 000018bd      6 FUNC  GLOBAL DEFAULT  8 wpa_ctrl_attach
41: 000018c3      6 FUNC  GLOBAL DEFAULT  8 wpa_ctrl_detach
42: 000018c9     32 FUNC  GLOBAL DEFAULT  8 wpa_ctrl_recv
43: 000018e9     68 FUNC  GLOBAL DEFAULT  8 wpa_ctrl_pending
44: 0000192d      4 FUNC  GLOBAL DEFAULT  8 wpa_ctrl_get_fd
(...)
```

approfondimenti di un così vasto argomento rimando ai seguenti libri [23, 67, 84].

Tornando all'esempio, mi sono accorto che con questo metodo posso invocare solo le funzioni esposte nel file `libwpa_client.so` legato a `wpa_supplicant`. Quindi, quante e quali sono le funzioni esposte? E' sufficiente utilizzare il comando del terminale Linux: `readelf -a libwpa_client.so | grep "func" libwpa_client.so`, usando il file `libwpa_client.so` del proprio dispositivo (oppure compilare Android seguendo la procedura del Capitolo 4 ed estrarre lo stesso dalla cartella “*out*”). La lista di funzioni in `libwpa_client.so` di Android Lollipop 5.0.1 è nell'Algoritmo 3.15.

Come si può notare, sono davvero poche quelle esposte e soprattutto non riguardano le interfacce di rete. Per tale motivo, questa soluzione non si è rivelata efficace. Così ho deciso di cambiare totalmente approccio.

Prima di spiegare ciò, è necessario acquisire alcuni concetti importanti riguardo le procedure di compilazione di Android e del suo kernel nel Capitolo 4, in modo da avere tutti gli elementi per procedere con lo studio di `wpa_supplicant`, nel Capitolo 5.

Algoritmo 3.16 Esempio chiamate da C/C++ a Java

```

(...)

//esempio chiamata metodo di una classe java
//se non vengono trovati Call<Class>Method verificare il
//build path e le import
jstring Java_com_example_hellojni_HelloJni_stringFromJNI(
    JNIEnv* env, jobject thiz ) {
    jclass clazz;
    clazz = (*env)->GetObjectClass(env, thiz);
    jfieldID instanceFieldId;
    instanceFieldId = (*env)->GetFieldID(env, clazz,
        "instanceField", "Ljava/lang/String;");
    jfieldID staticFieldId;
    staticFieldId = (*env)->GetStaticFieldID(env, clazz,
        "staticField", "Ljava/lang/String;");
    jstring instanceField;
    instanceField = (*env)->GetObjectField(env, thiz,
        instanceFieldId);
    jstring staticField;
    staticField = (*env)->GetStaticObjectField(env,
        clazz, staticFieldId);
    jmethodID instanceMethodId;
    instanceMethodId = (*env)->GetMethodID(env, clazz,
        "instanceMethod", "()Ljava/lang/String;");
    jmethodID staticMethodId;
    staticMethodId = (*env)->GetStaticMethodID(env, clazz,
        "staticMethod", "()Ljava/lang/String;");
    jstring instanceMethodResult;
    instanceMethodResult = (*env)->CallStringMethod(env,
        thiz, instanceMethodId);
    jstring staticMethodResult;
    staticMethodResult = (*env)->CallStaticStringMethod(
        env, clazz, staticMethodId);
    return (*env)->NewStringUTF(env, "Hello from JNI");
}

```

3. Esplorazione del codice sorgente

Capitolo 4

Architettura, kernel e Build System di Android

L'*Android Open Source Project* (AOSP) [37] è un'iniziativa creata per guidare lo sviluppo di Android da parte di Google e molte altre società come per esempio Qualcomm, Broadcom, Samsung e HTC. Nonostante sia un progetto *Open Source*, non significa che si possa visionare il codice in fase di sviluppo e provare le nuove versioni di Android prima del rilascio da parte di Google. AOSP gestisce il codice e le versioni di Android in *code line* che non corrispondono necessariamente ai *branch Git*, cioè una *code line* può essere costituita da più *branch*. Un esempio è in Figura 4.1.

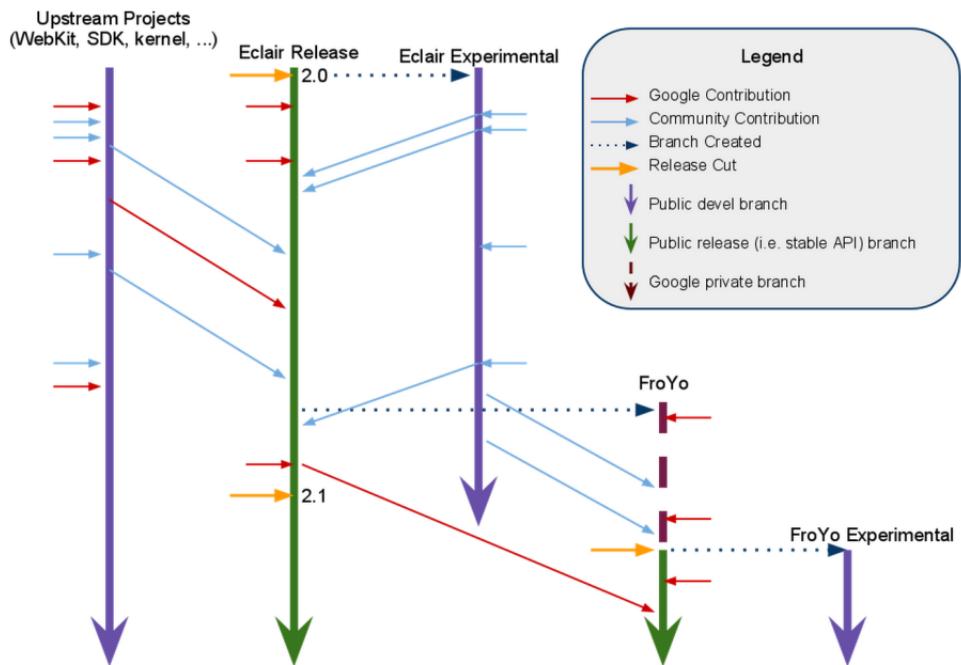


Figura 4.1: Esempio di Code line in Android

Quando esce una versione di Android di solito viene creato un nuovo *branch*. Intanto, i produttori di dispositivi e i *contributors ufficiali* possono lavorare sull’ultima versione correggendo bug e sperimentando funzionalità. Nel frattempo, Google lavora sulla prossima versione di Android internamente (eventualmente con alcune collaborazioni) su dispositivi con le caratteristiche che vuole utilizzare per la prossima versione. Una volta completata, il codice sorgente diventa disponibile nella pagina web ufficiale [36]. Oltre alle minor release, identificate con l’aumento della terza cifra della versione (per esempio tra 5.0.0 e 5.0.1), spesso per ogni versione sono usati più *branch*, specificando la release, per esempio “r1”, “r2”. Poiché Android è un progetto *Open source*, si può scaricare il codice di tutte le versioni rilasciate pubblicamente dalla *repository Git*, eventualmente modificandolo e compilandolo.

In questo capitolo mostrerò la compilazione del sistema operativo, la modifica e la compilazione del kernel ed infine, l’installazione su un dispositivo fisico reale. Tali operazioni dipendono molto dalla configurazione dell’ambiente di sviluppo scelto, quindi mi concentrerò in modo specifico solo sugli elementi essenziali e sui problemi affrontati. Inoltre, fornirò anche dei consigli pratici basati sulle mie esperienze. Alcune delle procedure che mostrerò non sono state descritte da Google in modo così dettagliato nella documentazione ufficiale [42]. Prima di analizzare i requisiti per compilare Android ed il kernel, voglio trattare brevemente alcuni aspetti più teorici dell’architettura di questo sistema operativo, analizzandone i blocchi fondamentali, ponendo particolare attenzione ai livelli più bassi.

4.1 L’architettura di Android

L’architettura di Android è fatta a livelli dove i più bassi sono vicini all’hardware, mentre i più alti all’utente, come rappresentato in Figura 4.2. Il primo è costituito dalle applicazioni. A seguire c’è il framework Android, cioè un insieme molto vasto di codice Java con cui il programmatore può interagire, già analizzato nel Capitolo 3. Dopodiché, c’è il livello delle librerie native, cioè in C/C++, che mettono in comunicazione la parte Java con la parte più a basso livello. Anche questa parte l’ho analizzata nel Capitolo 3, mostrando esempi di utilizzo di NDK, cioè lo strumento che permette di interagirci. Data la vastità dell’argomento, mi sono concentrato solo su alcuni aspetti a titolo esemplificativo, come per esempio *wpa_supplicant* che tratterò meglio nel Capitolo 5. Subito sotto c’è l’*Android Runtime*, costituito da ART e Dalvik, che permette l’esecuzione di programmi dei livelli superiori e l’HAL, che si occupa di astrarre l’hardware creando un’interfaccia software standard. Questi livelli richiederebbero una trattazione specifica, ma non fanno parte di questo lavoro di tesi. Piuttosto, mi concentrerò sull’ultimo, cioè il kernel Linux.

Per una questione di completezza ho deciso di analizzare brevemente ciò che Google chiama nella sua documentazione come *Android Low-Level System Framework*

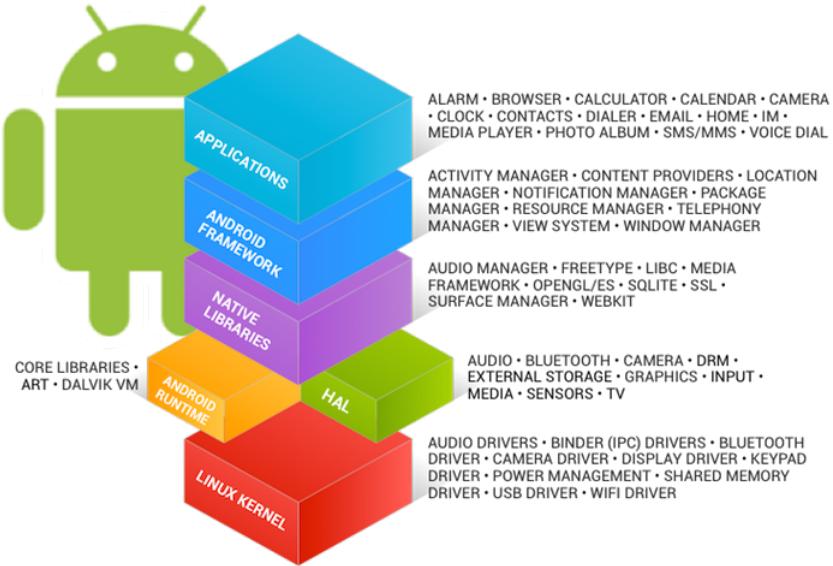


Figura 4.2: Android framework

(vedi Figura 4.3). Il primo componente è quello che ho spiegato poco fa, detto *Application Framework*. Il livello inferiore, detto *Binder IPC Proxies* contiene un meccanismo che permette all'*Application Framework* di superare i limiti dei processi e chiamare in Android i servizi di sistema, da API di alto livello. Esso è detto IPC, cioè *InterProcess Communication*. Subito sotto vi sono i servizi di sistema che permettono alle API di alto livello di comunicare con l'hardware. Per esempio, ci sono servizi come il *Windows Manager*, l'*Activity Manager* e quelli per i contenuti multimediali. Nel penultimo livello c'è l'*Hardware Abstraction Layer* (HAL), come in Figura 4.4, cioè un'interfaccia standard che permette ad Android di chiamare i driver, facendo sì che l'implementazione di basso livello e l'hardware siano separati dal software presente nei livelli superiori. Quando un produttore utilizza un suo componente, implementa la parte dell'HAL e il driver necessario. Android non standardizza l'interazione tra l'elemento nell'HAL e il driver, ma richiede che siano rispettati gli standard per far sì che il sistema operativo possa comunicare con il nuovo elemento. Di solito le implementazioni dell'HAL sono salvate in librerie condivise, cioè file “.so”. Infine vi è il kernel, basato su Linux, ma con alcune modifiche come i *wakelock*, una gestione più aggressiva della memoria, il *Binder IPC driver* ed altre caratteristiche utili in ambienti embedded. Nel corso di questo capitolo e dei successivi, analizzerò più nel dettaglio il kernel di Android.

4.2 Il processo di boot in Android

In questa sezione mostrerò il processo di boot di Android, ma per comprenderla meglio riporto una breve sintesi della struttura delle partizioni [69].



Figura 4.3: Android framework nel dettaglio

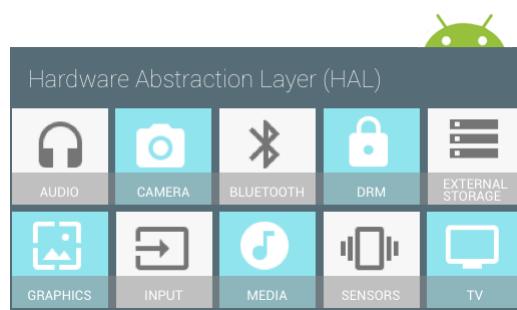


Figura 4.4: HAL

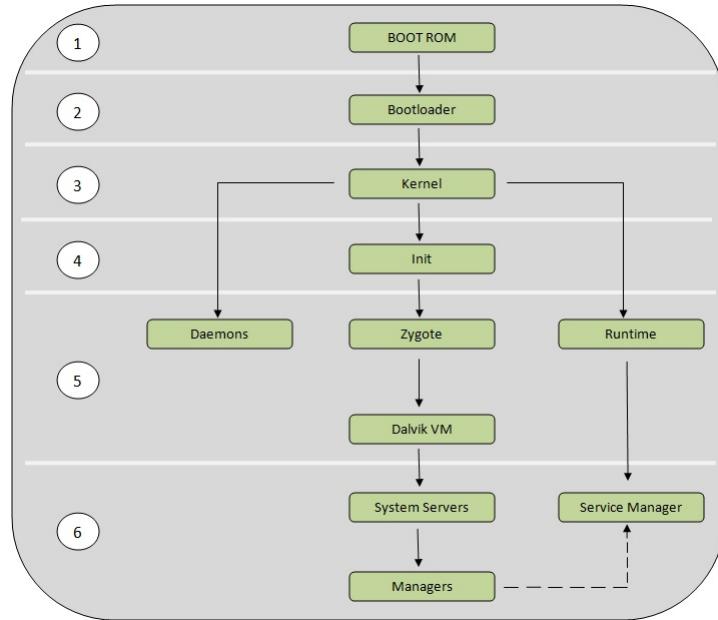


Figura 4.5: Processo di boot completo

1. /boot: include kernel e *ramdisk* e quindi è essenziale per avviare il sistema operativo.
2. /system: contiene Android, il kernel e *ramdisk*. Rimuoverla vuol dire cancellare il sistema operativo e rendere impossibile l'avvio del dispositivo.
3. /recovery: utile per i backup e può essere usata come procedura alternativa per il boot, per esempio per ripristinare un backup.
4. /data: dati dell'utente e applicazioni installate. È la partizione cancellata quando si fa il *factory reset*, cioè si ritorna alle impostazioni di fabbrica.
5. /cache: usata da Android per salvare elementi a cui accede spesso. Può essere cancellata senza rischi.
6. /misc: impostazioni di Android, tra cui le configurazioni hardware e dell'operatore.
7. /sdcard rappresenta la memoria usata dall'utente, che può essere in una partizione o in una scheda esterna.
8. altre che dipendono dal produttore.

Nel momento in cui si accende il dispositivo premendo l'apposito pulsante, iniziano una serie di eventi sequenziali che portano all'avvio di Android. Il questo lavoro di testi mostrerò queste fasi (vedi Figura 4.5) in modo sintetico e schematico [66, 86].

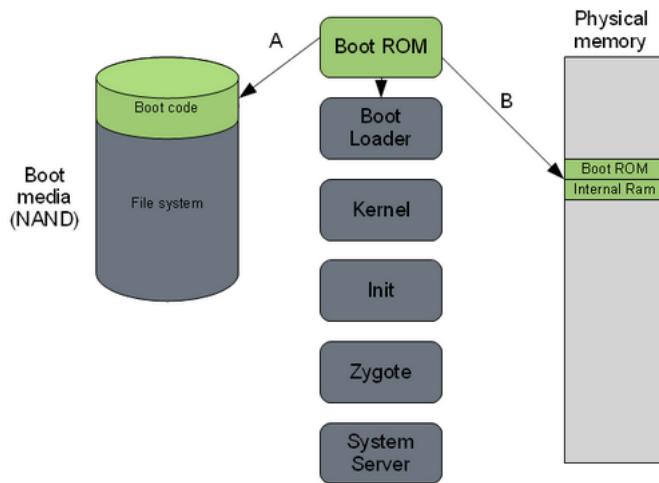


Figura 4.6: Processo di boot - passo 1

Fase 1: Avvio del sistema

Acceso il dispositivo, inizia la procedura divisa nelle due fasi, in Figura 4.6.

1. Il codice non modificabile scritto nella BootROM (che di solito è un chip dedicato) rileva il *Boot media (NAND)*. Questo serve per determinare dove si troverà il *first stage Bootloader* (cioè la prima parte del Bootloader).
2. La *BootROM* tenta di caricare il *first stage Bootloader* nella *Internal RAM*. Una volta fatto, il codice nella *BootROM* esegue un'istruzione di salto (jump) e l'esecuzione passa alla fase successiva.

Fase 2: Bootloader

Il Bootloader è in due fasi (o stage), cioè *first stage Bootloader* e *second stage Bootloader*. In Figura 4.7 è rappresentata la procedura in dettaglio.

1. Il *first stage Bootloader* rileva e imposta l'*External RAM*.
2. Una volta che l'*External RAM* è disponibile e il sistema è pronto ad eseguire operazioni, il *first stage Bootloader* esegue il caricamento della seconda fase nella *External RAM*.
3. Il *second stage Bootloader* può contenere codice per impostare il filesystem, memoria aggiuntiva ed il supporto di rete. Inoltre, può essere utile anche per attivare protezioni della memoria e di sicurezza sui moderni dispositivi¹.

¹Spesso in ambiente embedded, le società che producono i dispositivi usano Bootloader proprietari e bloccati. L'obiettivo è impedire la modifica del software, per esempio come fa Samsung.

4. Ricerca il *kernel Linux* per avviarlo, lo prende dal *Boot media* e lo mette in RAM, cioè carica la *zImage*, il file binario compresso del *kernel*. In esso vi è anche tutta la descrizione dell'hardware.
5. Esegue un'operazione di salto al *kernel Linux* e svolge alcune procedure di decompressione. Infine il controllo è passato al kernel.

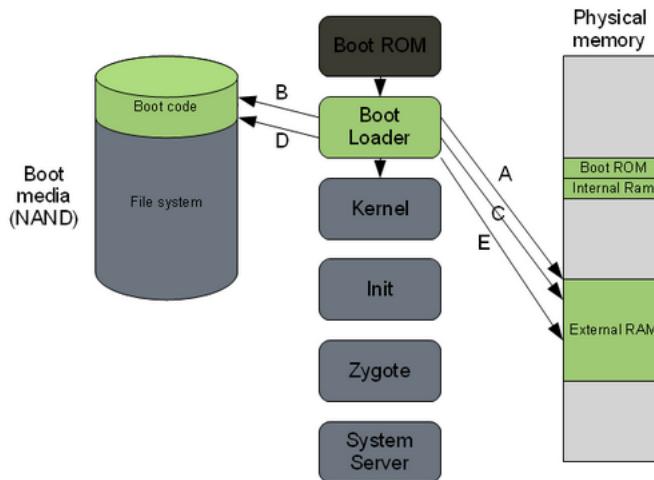


Figura 4.7: Processo di boot - passo 2

Nei dispositivi più vecchi, sono stati usati due registri: *r1* per salvare il *tipo di macchina* ed *r2* per l'*ATAGS* che contiene informazioni come la posizione e la dimensione della memoria. In quelli più moderni, il kernel non contiene più le informazioni sull'hardware, che invece sono in un file binario a parte, detto *Device Tree Blob* (DTB). Di conseguenza, il *Bootloader* carica due file binari, cioè la *zImage* e il *DTB*, passando l'indirizzo di memoria di quest'ultimo attraverso il registro *r2*. Il *tipo di macchina* non è più necessario e quindi il registro *r1* non è più utilizzato. Per architetture ARM, tutti i DTS sono in *arch/arm/boot/dts*, cioè sia i ".dts" con informazioni sulla *board* e i ".dtsi" per informazioni sul SoC. Questi file sono compilati da un programma chiamato *Device Tree Compiler*, il cui codice sorgente si trova in *scripts/dtc*. Quindi, il *DTB* generato dal compilatore è caricato dal *Bootloader* e letto dal *kernel* al momento del boot. Per maggiori informazioni e la spiegazione del linguaggio di descrizione hardware DT vedere [78].

Fase 3: Kernel Linux

Il kernel si avvia in modo molto simile sia in Android, sia in altri sistemi (vedi Figura 4.8).

1. Una volta che *Memory Management Unit* (MMU) e cache sono inizializzate, il sistema sarà in grado di usare la memoria virtuale e lanciare i processi nello spazio dell'utente (*user space*).

2. Il kernel guarda nella radice del filesystem per trovare la procedura di *init* e l'avvia come un processo utente.

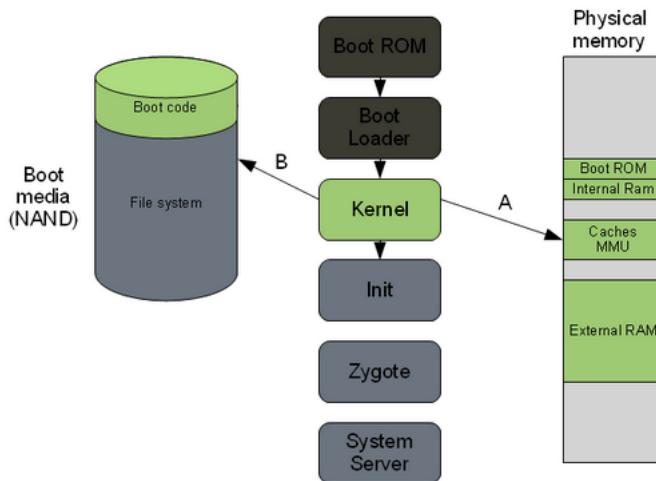


Figura 4.8: Processo di boot - passo 3

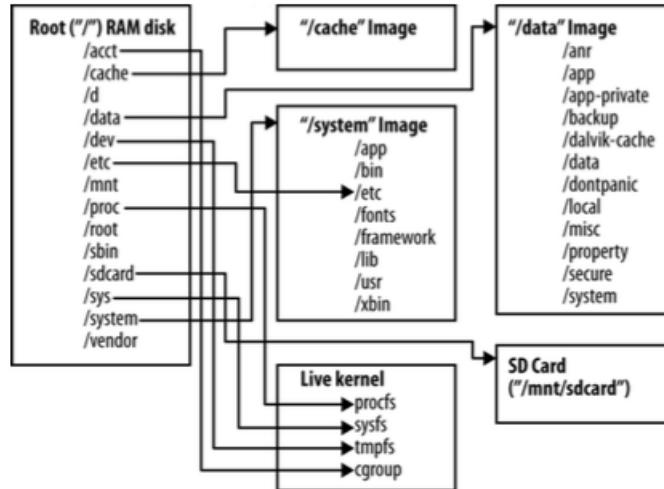
Fase 4: Il processo di init

Init è il primo vero processo eseguito ed ha due responsabilità (vedi Figura 4.10).

- Fa il *mount* di cartelle come */sys*, */dev*, */proc*, tramite *procfs*, *sysfs* e *cgroupfs* [116]. Questi ultimi tre sono tutti filesystem virtuali mantenuti dal kernel in esecuzione. Essi non sono usati per il salvataggio di dati, infatti le strutture dati sono nel kernel. Per esempio, *procfs* è il modo tradizionale con cui il kernel esporta informazioni su se stesso nello spazio dell'utente. Tipicamente, queste informazioni sono viste come file accessibili o log per ottenere informazioni.
- Esegue lo script *init.rc*, che esegue il *mount* di diverse altre “immagini” e filesystem virtuali, come in Figura 4.9. Questo perché ognuno ha scopi differenti. Per esempio, *ramdisk.img* è una “immagine” compressa prima di essere caricata in RAM dal kernel, che viene poi montata come un filesystem di root in sola lettura.

Fasi 5 e 6: Zygote, Dalvik e System server

Queste fasi non fanno parte del mio lavoro di tesi, quindi non le considererò. Comunque, in esse è avviata la parte Java di Android, c’è la gestione delle macchine virtuali Dalvik, una per ogni app ed infine sono gestiti ed avviati i servizi di sistema.



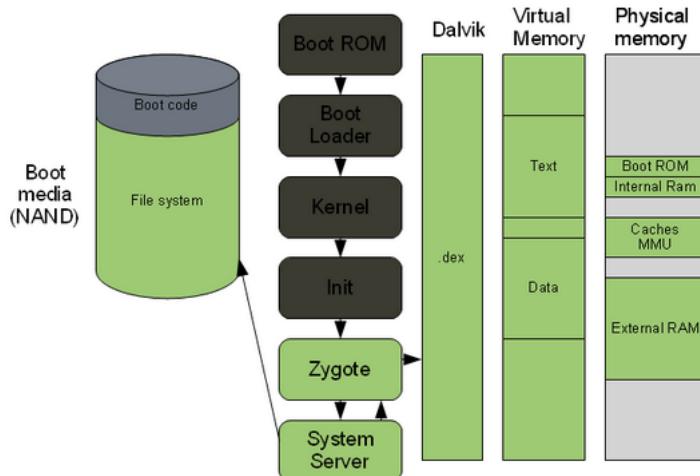


Figura 4.11: Processo di boot completato

e altri problemi che ho incontrato, come la necessità di cambiare filesystem o usare immagini “.dmg” con filesystem *HFS case sensitive*), Windows non è nemmeno supportato ufficialmente e ne sconsiglio vivamente l’utilizzo.

- Un PC/MAC con installato Linux (anche come seconda partizione) con almeno 8GB di RAM o una macchina virtuale (Google consiglia come minimo 16GB di RAM in caso di macchine virtuali). Dalla mia esperienza con VMware Fusion 2 su MAC OS X, consiglio di assegnare almeno 8GB o 10GB alla macchina virtuale Linux ed ovviamente configurarla per usare tutti i core della CPU.
- Almeno 50 GB di spazio libero, 100 GB per una singola versione e 150 GB o più per più versioni. Utilizzando la *Ccache*, che tratterò nella Sezione 4.4.3, sono necessari altri 50GB o più. Dalla mia esperienza, una macchina virtuale Linux di 200 GB di disco non permette di avere i sorgenti di KitKat, Lollipop e i kernel associati con anche i file compilati, infatti sono necessari almeno altri 50 GB. Inoltre, se possibile, è meglio non usare macchine virtuali, per questioni di performance e per problemi nel gestire immagini di dischi così grosse. Lo stesso MAC OS X mi ha creato difficoltà nel trasferimento di questi file da HD esterni usati come backup di *Apple TimeMachine*, generando errori nel processo chiamato *Finder*. Se possibile creare la macchina virtuale con un disco espandibile di almeno 1 TB e usare immagini multiple, cioè divise in tanti file.
- Python 2.6 - 2.7.
- GNU Make 3.81 - 3.82.
- Git 1.7 o superiore.

- JDK (bisogna rispettare vincoli molto stringenti sia sulla versione, sia sul fornitore, cioè Oracle o OpenJDK). Avendo compilato più versioni di Android, ho dovuto installare più versioni di Java e selezionare quella corretta prima di ogni compilazione, tramite i comandi del terminale `update-alternatives --config java` e `update-alternatives --config javac`. Le versioni sono:
 - 7 per compilare l'ultima versione su *branch master* di git (Lollipop richiede necessariamente Open JDK 7);
 - 6 per compilare versioni da Gingerbread a KitKat (KitKat richiede necessariamente Oracle JDK 6);
 - 5 per compilare versioni da Cupcake a Froyo.
- Una connessione ad internet estremamente veloce, infatti scaricare il sorgente di Android ha richiesto diverse ore, anche con una fibra ottica a 1MB/s ed un'intera notte non è stata sufficiente).
- Il processo di compilazione di Android richiede alcune ore anche su macchine di ultima generazione, quindi ho usato un MacBook Pro (mid 2012) modificato con 16GB di RAM Corsair Vengeance DDR3 1600 MHz, Intel Core i7 2,3 GHz (Turbo Boost 3,3 GHz) quad core, disco da 750 GB 7200 RPM Seagate Momentus XT ibrido con 8GB di SSD. Nonostante queste specifiche, nella macchina virtuale, la compilazione di Lollipop ha richiesto 2 ore e 40 minuti.
- Evitare qualunque tipo di aggiornamento di Ubuntu. Nel mio caso, l'installazione di alcuni aggiornamenti ha reso impossibile la compilazione ed è stato necessario ripristinare 200GB di macchina virtuale da un disco esterno di backup, con notevoli difficoltà.
- Android NDK 10d 64 bit [33] o superiore per la compilazione (nonostante non sia obbligatorio, dato che il sorgente di Android contiene già il *toolchain* [92] necessario, in tutta la procedura ho utilizzato quello incluso nell'NDK, perché mi ha dato meno problemi).

4.4 Inizializzare l'ambiente di sviluppo

Prima di procedere, ho aggiornato Ubuntu 14.04.2 via terminale con i comandi nell'Algoritmo 4.1 e ho disattivato gli aggiornamenti automatici per evitare problemi in futuro.

Algoritmo 4.1 Aggiornare Ubuntu

```
$ sudo apt-get update  
$ sudo apt-get upgrade
```

4.4.1 Installare Java, Android NDK e le dipendenze

Mi sono assicurato di rimuovere tutte le versioni di Java da Ubuntu e dopo il riavvio, ho eseguito l'installazione tramite linea di comando come nell'Algoritmo 4.2 scegliendo la versione da utilizzare con gli ultimi due comandi `update-alternatives`.

Algoritmo 4.2 Installare Java 7 e 6 e scegliere la versione da usare

```
$ sudo apt-get install openjdk-7-jdk
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java6-installer
$ sudo update-alternatives --config java
$ sudo update-alternatives --config javac
```

La guida ufficiale di Google per Ubuntu 14.04 fornisce già i comandi da utilizzare per installare le dipendenze (vedi Algoritmo 4.3). Però, non specifica che bisogna installare anche altri programmi perché le procedure di compilazione possano avere successo. Quindi, in base alla mia esperienza ho aggiunto all'Algoritmo 4.3 quattro righe con altri software utili a tale scopo. Dopodiché, ho installato Android NDK, cioè ho scaricato NDK Linux x64 da [33] ed eseguito i comandi nell'Algoritmo 4.4 dalla mia cartella *home*. In questo lavoro di tesi ho utilizzato l'ultima versione disponibile, cioè la *r10d*.

Algoritmo 4.3 Installare le dipendenze in Ubuntu

```
$ sudo apt-get install bison g++-multilib git
$ sudo apt-get install gperf libxml2-utils
$ sudo apt-get install libncurses5-dev
$ sudo apt-get install libc6-i386 lib32stdc++6
$ sudo apt-get install lib32gcc1 lib32ncurses5
$ sudo apt-get install zlib1g lib32z1
```

Algoritmo 4.4 Installare Android NDK

```
$ mkdir ~/androidNdk
$ cd ~/Scaricati
$ mv android-ndk-r10d-linux-x86_64.bin
    ./androidNdk/android-ndk-r10d-linux-x86_64.bin
$ cd ../androidNdk
$ chmod a+x android-ndk-r10d-linux-x86_64.bin
$ ./android-ndk-r10d-linux-x86_64.bin
$ rm android-ndk-r10d-linux-x86_64.bin
```

4.4.2 Configurare l'accesso USB in Ubuntu

Per configurare l'accesso USB in Ubuntu, l'approccio raccomandato da Google è di creare o modificare il file `/etc/udev/rules.d/51-android.rules` accedendo coi privilegi di root in Ubuntu. Il problema è che la guida ufficiale è per Ubuntu 8.04 e 10.04, mentre ora il sistema operativo consigliato per lo sviluppo è Ubuntu 14.04. Quindi ho utilizzato il file `51-android.rules` nell'appendice A.

4.4.3 Variabili d'ambiente e CChace

I dispositivi Android non hanno la stessa architettura di un normale PC, infatti si basano principalmente su ARM. Quindi, le istruzioni compilate devono essere per ARM e non per processori Intel o AMD.

Quando su una macchina con una certa architettura si compila un programma per un'altra, si dice “*cross-compilazione*”. Non è da confondere con la “*compilazione source-to-source*”, perché in quest'ultima è il codice sorgente ad essere tradotto, mentre nella *cross-compilazione* è il codice binario. Per fortuna, Google fornisce tutti i programmi per svolgere questa procedura, ma è necessaria un'accurata configurazione. Spesso, il principale errore nella compilazione di Android è l'uso di variabili d'ambiente sbagliate. Ciò comporta errori, che potrebbero non essere visibili all'inizio del processo, ma solo in seguito, interrompendolo e quindi perdendo tempo. In questo lavoro di tesi mi sono concentrato su Android 4.4.4 e 5.0.1 per Nexus 4 e Nexus 5, cioè architetture ARM a 32 bit molto diffuse.

Per impostare le variabili d'ambiente, ho dovuto visualizzare i file nascosti in Ubuntu ed aggiungere le righe nell'Algoritmo 4.5 al file `~/.bashrc` (se non presente crearne uno). Questa operazione permetterà la cross-compilazione del kernel. La prima riga specifica l'architettura verso cui *cross-compilare*, che nel mio caso è ARM [7], la seconda la sotto-architettura, cioè sempre ARM ed infine il parametro da passare per *cross-compilare*, cioè il nome del *toolchain* scelto, seguito da “-”. Dopodiché, ho modificato il file `~/.profile` (eventualmente crearlo) aggiungendo le righe nell'Algoritmo 4.6. Da notare che il primo *export* aggiunge alla variabile d'ambiente *PATH* il percorso del *toolchain* di NDK, mentre le altre configurano la *Ccache*. Modificati questi due file, ho riavviato il sistema operativo per applicare le modifiche.

Algoritmo 4.5 Configurazione variabili d'ambiente in `.bashrc`

```
export ARCH=arm
export SUBARCH=arm
export CROSS_COMPILE=arm-linux-androideabi-
```

Quando inizia la compilazione di Android, sono compilati molti file C e C++. Per migliorare le performance in caso di ricompilazioni, modifiche, passaggi tra diverse versioni di Android, consiglio di attivare la Ccache [18] con `USE_CCACHE=1`,

Algoritmo 4.6 Configurazione variabili d'ambiente in `.profile`

```
export PATH=$PATH:~/androidNdk/android-ndk-r10d/
    toolchains/arm-linux-androideabi-4.6/
        prebuilt/linux-x86_64/bin
#cache per compilare android
export USE_CCACHE=1
export CCACHE_DIR=~/androidsource/aosp-main/cache/.ccache
```

e la cartella della Ccache nella variabile `CCACHE_DIR`. Ccache è una cache del compilatore C/C++/Objective-C/Objective-C++ che rende più veloce la ricompilazione, poiché memorizza i file in una cartella sul disco. Ovviamente, al primo avvio sarà vuota e ogni operazione sarà un *miss*, ma successive ricompilazione potranno a degli *hit* facendo risparmiare tempo. Ccache è sicura perché fornisce sempre lo stesso output, cioè produce sempre lo stesso file oggetto. Purtroppo è limitata ai file C/C++ di Android e non a quelli Java, ciò vuol dire che il framework Android e tutte le app predefinite devono essere ricompilate ogni volta. In realtà, l'uso di questa cache è comunque vantaggioso, perché Android è costituito da una consistente parte in C/C++ e grazie a Ccache si può risparmiare quasi un'ora.

4.5 Scaricare il codice sorgente di Android

Per scaricare il sorgente di Android è necessario installare *Repo*, cioè un programma per gestire *repository* costruito sopra a *Git*. *Repo* non ha l'obiettivo di sostituire quest'ultimo, ma serve solo per rendere più facile l'uso di *Git* nel caso di Android. Il questo lavoro di tesi l'ho utilizzato tramite il comando *repo*, cioè uno script *Python* fornito da Google, e mi siamo concentrato solo sugli aspetti fondamentali per scaricare il codice sorgente. Per maggiori informazioni c'è la documentazione ufficiale [47]. *Repo* non è incluso in Linux, quindi ho eseguito i comandi nell'Algoritmo 4.7 per installarlo.

Algoritmo 4.7 Comandi per installare *Repo*

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

A questo punto ho lanciato il comando per inizializzare *Repo* in una cartella vuota `~/androidsource`, come in Algoritmo 4.8, ma sostituendo “<BRANCH NAME>” con il nome del *branch* da scaricare [38]. Per poter scaricare il codice sorgente superando i limiti assegnati da Google ad ogni indirizzo IP ad accesso anonimo, ho eseguito

l'autenticazione. Per fare ciò ho usato l'indirizzo https in Algoritmo 4.7 con la directory “/a” nel percorso, per forzare l'autenticazione. Rimuovendo “/a” si disattiva automaticamente (operazione sconsigliata). Per autenticarmi, ho eseguito lo script in Algoritmo 4.7 da [41].

Algoritmo 4.8 Comandi per inizializzare Repo

```
$ mkdir ~/androidsource
//visitare sito https://www.googlesource.com/new-password
//per ottenere il seguente script
//personalizzato
$ touch ~/.gitcookies chmod 0600 ~/.gitcookies
  git config --global http.cookiefile ~/.gitcookies
  tr , \\t <<__END__
  .googlesource.com,TRUE,/ ,TRUE,<nodec numerico>,o,
  git-<email senza la @>=<codice alfanumerico> __END__
$ cd ~/androidsource/aosp-main
$ git config --global user.email "you@example.com"
$ git config --global user.name "Your Name"
$ repo init -u
  https://android.googlesource.com/a/platform/manifest
  -b <BRANCH NAME>
```

Fatto ciò, ho iniziato il processo di download con il comando in Algoritmo 4.9. Per migliorare le performance, ho usato il parametro -jN, con N = numero core * numero thread per core * 2. Parlerò dell'opzione -jN in modo dettagliato nella Sezione 4.6.1.

Algoritmo 4.9 Avvio del download del sorgente di Android

```
$ repo sync -jN
```

Aggiungere i file binari proprietari

Per poter compilare Android ed installarlo su un dispositivo fisico e non nell'emulatore, è necessario inserire nel codice sorgente i “file binari proprietari”, solitamente driver, scaricabili a parte. Da [39] ho scaricato quelli per la versione che ho scelto, ma se volessi compilare Android dal *branch master*, dovrei usare [45]. Completato il processo di download, li ho aggiunti al sorgente.

1. Ho spostato i file scaricati su Ubuntu senza estrarli in altri sistemi operativi per evitare di modificare i permessi dei singoli file e far fallire il processo di compilazione di Android.
2. Ho creato la cartella `~/androidsource/vendor`.

3. Ho estratto i file .tgz nella cartella principale del sorgente di Android col seguente comando: `$ tar -xf nomefile.tgz`.
4. Ho modificato i permessi ai file .sh estratti permettendone l'esecuzione con: `$ chmod +x nomefile.sh`.
5. Ho eseguito i file “.sh” da terminale con `$ sudo ./nomefile.sh` accettando i termini di licenza premendo ENTER fino alla sottosezione *8.e.* Allora ho premuto ancora alcune volte il tasto invio fino alla comparsa della scritta in Figura 4.12 e mi sono fermato. Quindi ho inserito “I ACCEPT” per confermare.
6. Per sicurezza ed evitare inutili problemi in fase di compilazione ho eseguito anche il comando `$ make clobber`.

Alcuni dispositivi presenti in AOSP richiedono file binari proprietari non inclusi nella *repository*. Per esempio il Galaxy Nexus prodotto da Samsung usa driver proprietari per la fotocamera, scaricabili dal sito web di Samsung. Senza di essi, l'app per scattare le foto si bloccherà all'avvio.

```
Agreement.

e. Entire Agreement. This Agreement completely and exclusively states
   the agreement between You and Licensor regarding this subject
   matter.

Type "I ACCEPT" if you agree to the terms of the license: I ACCEPT

vendor/
vendor/lge/
vendor/lge/hammerhead/
vendor/lge/hammerhead/device-vendor.mk
vendor/lge/hammerhead/BoardConfigVendor.mk
vendor/broadcom/
vendor/broadcom/hammerhead/
vendor/broadcom/hammerhead/proprietary/
vendor/broadcom/hammerhead/proprietary/bcm2079x-b5_pre_firmware.ncd
vendor/broadcom/hammerhead/proprietary/bcm2079x-b5_firmware.ncd
vendor/broadcom/hammerhead/proprietary/bcm4335c0.hcd
vendor/broadcom/hammerhead/BoardConfigPartial.mk
vendor/broadcom/hammerhead/device-partial.mk

Files extracted successfully.
ks89@ubuntu:~/android/androidsource/aosp-main$
```

Figura 4.12: Licenza dei file binari proprietari ed estrazione completa

4.6 L'Android Build System e la compilazione di Android

L'insieme di strumenti per compilare Android è definito “*Android Build System*”, sia nella documentazione ufficiale, sia in libri come per esempio [116]. Da ora in poi utilizzerò sempre tale termine o le versioni contratte “*Build System*” e “*ABS*” per riferirmi al processo di compilazione di Android.

L'Android Build System si basa su *GNU make*², ma non usa *makefile* ricorsivi, come avviene nel kernel Linux. Infatti, in quest'ultimo si usa uno script che invoca i *makefile* delle sottocartelle, in modo ricorsivo. Invece, l'ABS ne usa uno che esplora tutte le sottocartelle fino a trovare gli *Android.mk*, cioè l'equivalente dei *makefile*. Quando vengono trovati i file “.mk”, l'ABS si ferma, a meno che essi non lo istruiscano di continuare ad esplorare le sottocartelle. Addirittura, non usa nemmeno veri *makefile*, ma si appoggia ai *Android.mk* che specificano dove il *modulo locale* deve essere compilato. Un *modulo Android* non ha nulla a che vedere con un modulo del kernel. In Android si definisce *modulo* ogni componente di AOSP che richiede di essere compilato, che può essere, per esempio, una libreria, un *package* di un'app o un file binario.

Un'altra specificità di Android è il modo con cui l'ABS è configurato. Infatti, si appoggia su un insieme di variabili che sono impostate dinamicamente come parte della shell dallo script *envsetup.sh* e dal comando *lunch* o definite staticamente in un file *buildpsec.mk*. In più, l'ABS non genera file oggetto o altri file intermedi con la stessa posizione dei sorgenti. Cioè non ci sono file “.o” vicino agli equivalenti “.c”, come per esempio avviene quando si compila un programma scritto in C. L'ABS crea un cartella chiamata *out* in cui sono salvati tutti i file generati. Quindi, il comando *make clean* cancella semplicemente la cartella “*out*”, poiché tutti i file compilati da rimuovere si trovano lì, cioè esegue la stessa operazione del comando Linux *rm -rf out*.

ABS: Architettura

In Figura 4.13 è rappresentata l'architettura dell'ABS. Dopo la fase iniziale di configurazione con *envsetup.sh* e *lunch*, si entra nel cuore di ABS, cioè *build/core*. L'entry point è il *main.mk* che si trova in *build/core* ed è invocato attraverso l'*Android.mk* di livello più alto. Ricordo che non è ricorsivo, quindi alla fine sarà creato un *makefile* unico, cioè tutti i *Android.mk* vengono incorporati in uno. Questa fase è quella che avviene subito dopo aver lanciato il comando *make*. L'ABS di Android 2.3 non la mostra nel terminale e in [116] spiega come abilitarne la visualizzazione, mentre per le versioni più recenti non è più necessario. Dopodiché, c'è la configurazione dell'ABS, cioè viene incluso il *config.mk*, ed altre operazioni. Capire esattamente come funziona tutta la procedura non è l'obiettivo di questo lavoro di tesi, quindi rimando a [116]. Le uniche informazioni che interessano sono il fatto che *envsetup.sh* definisce una serie di comandi della shell e che in tutta la procedura non vi è niente che riguarda il kernel, infatti il sorgente di Android non è fornito insieme, ma quest'ultimo deve essere scaricato e compilato a parte (vedi Sezione 4.7).

²GNU Make è un programma per generare eseguibili dal codice sorgente. Make capisce come compilare questo codice usando file chiamati “.makefile” [28].

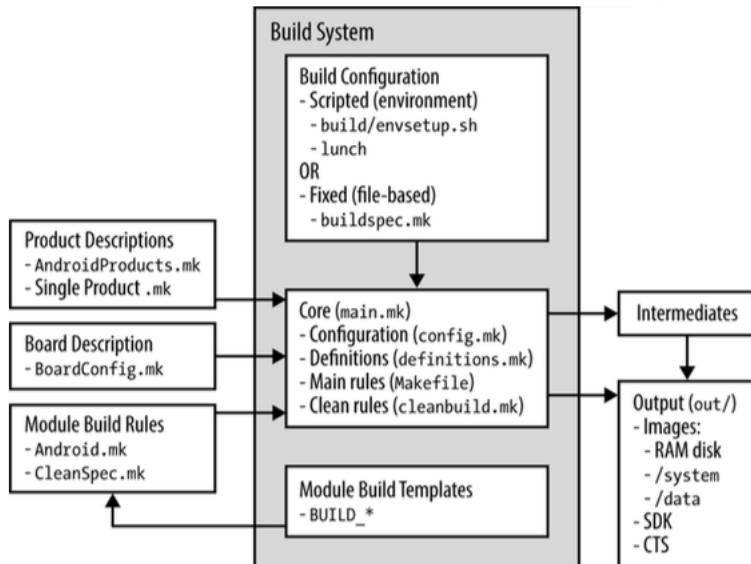


Figura 4.13: Architettura dell'Android Build System

Piuttosto, mi voglio concentrare su un argomento che nel Capitolo 3 ho rimandato, i *Module template*. Google fornisce alcuni *template* in modo che gli autori dei moduli possano far sì che i loro software siano compilati in modo corretto. Ogni *template* è fatto su misura per uno specifico tipo di modulo e gli autori possono usare un insieme di variabili ben documentato e fatte su misura. Questi *template* si trovano sempre in *build/core* e vi si accede con la direttiva *include*, la stessa che si usa in C. Un esempio di *include* è: *include \$(BUILD_PACKAGE)*. Alcuni esempi di *Module template* sono *BUILD_EXECUTABLES* (usato da comandi nativi e daemon), *BUILD_JAVA_LIBRARY* (usato dal framework Android e da Apache Harmony), *BUILD_SHARED_LIBRARY* (molto usato in *external* e in *framework/base*), *BUILD_STATIC_LIBRARY* (molto usato in tutto AOSP, in particolare in *external*), *BUILD_PREBUILT* (usato in file di configurazione e binari), *BUILD PACKAGE* (usato da tutte le app di AOSP). Questi *template* fanno sì che gli *Android.mk* siano molto compatti e forniscano con semplicità tutto ciò che serve all'autore dei moduli. Per fare un paragone con un concetto dell'ingegneria del software, sono i *design pattern* applicati a basso livello in Android. Il vantaggio di una struttura a moduli è che si può compilare anche un solo alla volta, o una lista di essi. È sufficiente sapere i nomi dei moduli, per esempio "Launcher2" lo si può compilare con *make Launcher2*. Inoltre, si può fare la pulizia di un singolo modulo alla volta. Per esempio "Launcher2" si usa *make clean-Launcher2*. I template corrispondono a delle posizioni predefinite di output nel filesystem. Riporto alcuni esempi in Tabella 4.1.

Template	Cartella d'output di default
BUILD_EXECUTABLE	/system/bin
BUILD_JAVA_LIBRARY	/system/framework
BUILD_SHARED_LIBRARY	/system/lib
BUILD_PREBUILT	No default, specificare con LOCAL_MODULE_CLASS o LOCAL_MODULE_PATH
BUILD_PACKAGE	/system/app
BUILD_KEY_CHAR_MAP	/system/usr/keychars

Tabella 4.1: Posizioni di default dei moduli in base al template usato

4.6.1 Cross-compilare Android per ARM

Innanzitutto, ho inizializzato l'ambiente con lo script `envsetup.sh`, eseguendo il comando nell'Algoritmo 4.10. Dopodiché, ho specificato la configurazione con il comando `lunch`, indicando il nome della build (*build name*) e il tipo (*build type*) nel formato *BUILD-BUILDTYPE*. Per esempio nel caso del Nexus 4, consiglio *full_mako-userdebug*, mentre il Nexus 5, *aosp_hammerhead-userdebug*. Una lista sufficientemente completa si trova nella Tabella 4.2, mentre per capire le differenze tra le varie *buildtype* riporto la Tabella 4.3.

Il vero processo di compilazione si avvia con il comando `make` nella riga 3, seguito da quello in riga 4 per creare il “.zip” installabile. Poiché il codice è veramente molto vasto e richiede molto tempo, conviene sfruttare al massimo i processori moderni multi-core e multi-thread, specificando il numero di processi paralleli per compilare con il parametro `-jN`. Nella macchina virtuale che ho utilizzato ho scelto `-j8`. Ovviamente, nelle impostazioni della macchina virtuale ho dovuto configurare il processore virtualizzato per essere multi-core. Secondo la mia esperienza un buon valore di *N* per la mia macchina virtuale è il doppio del numero dei core del processore. Ovviamente, ciò dipende molto dal tipo di CPU e dalla macchina stessa. È impossibile definire un numero standard adatto a tutti. Google consiglia di usare due volte il numero dei thread hardware, cioè *N* = numero core * numero thread per core * 2. L'unico consiglio che posso dare in questa fase è fare una compilazione con `make -j1`, la prima volta, per far sì che il log generato sia in ordine e in caso di errori sia più facile individuare la causa. Infatti, utilizzando più processi paralleli il log risulta quasi illeggibile, poiché i processi scrivono nel terminale le operazioni che stanno svolgendo in modo concorrente. Inoltre, è capitato molto spesso di incontrare problemi durante la compilazione della parte bluetooth e usb di Android. Le soluzioni che ho adottato sono state quelle di verificare di avere i driver proprietari ed eseguire una completa pulizia dei file compilati tramite `make clean` o addirittura `make clobber`. Se anche così non si ottengono risultati positivi, conviene provare a riavviare la macchina, assicurandoci che l'ambiente venga inizializzato correttamente, cioè leggendo le variabili

Algoritmo 4.10 Inizializzare l'ambiente per compilare Android

```
$ cd ~/androidsource/aosp-main  
$ source build/envsetup.sh
```

Algoritmo 4.11 Scegliere obiettivo con lunch, attivare la ccache e compilare

1. \$ lunch BUILD-BUILDTYPE
 2. \$ prebuilts/misc/linux-x86/ccache/ccache -M 50G
 3. \$ make -jN
 4. \$ make otapackage -jN
-

d'ambiente con il comando *printenv*, oppure *echo \$NOME_ VARIABILE* per vedere il contenuto di una sola di esse.

Nell'Algoritmo 4.11 ho riportato la procedura per avviare la compilazione ed impostare la Ccache a 50 GB, cioè la dimensione raccomandata da Google. Perché ciò funzioni, ho definito la variabile d'ambiente *CCACHE* con percorso *~/androidsource/aosp-main/cache/.ccache* nel file *.profile*. Per visionare in tempo reale l'utilizzo della Ccache (hit e miss) durante la compilazione, ho usato il comando *watch -n1 -d prebuilts/misc/linux-x86/ccache/ccache -s* dalla cartella principale del codice sorgente di Android.

4.7 Compilare il kernel e creare zip installabili

Ora è giunto il momento di vedere nel dettaglio la compilazione del kernel di Android. Dato che il codice sorgente è incluso in quello di Android, l'ho scaricato in base alla piattaforma in uso, cioè un SoC Qualcomm, per esempio il Nexus 4 e il Nexus 5. Accedendo alla *repository* di Google e cercando nella pagina il testo “kernel/”, si può facilmente vedere la lista di tutte quelle supportate, cioè: bcm, common, exynos, goldfish, lk, mediatek, msm, omap, samsung, tegra, x86_64. Quella associata a Qualcomm è chiamata *msm* e si trova in [46]. Per maggiori informazioni fare riferimento alla Tabella 4.4.

Il kernel l'ho scaricato in modo specifico per il mio dispositivo e per fare ciò non ho considerato i *tag* della *repository*, ma solo i *branch*, cioè per esempio Android 5.0.1 del Nexus 5 si trova nel *branch* chiamato *android-msm-hammerhead-3.4-lollipop-release*. Una volta individuato il sorgente da compilare, ho eseguito i comandi nell'Algoritmo 4.12. Prima di procedere ho verificato di avere i tre *export* corretti in *.bashrc* ed di aver eseguito i comandi nell'Algoritmo 4.13.

Quando la compilazione è terminata, nella cartella */home/<cartella utente>/androidKernel/msm/arch/arm/boot* ci sarà il file *zImage*, cioè il binario del kernel compilato e compresso. Durante i miei esperimenti, mi sono accorto che alcuni dispositivi, come il Nexus 5, usano la versione *DTB* di tale file, cioè *zImage-dtb*.

Device name	Code name	Build configuration
Nexus 6	shamu	aosp_shamu-userdebug
Nexus Player	fugu	aosp_fugu-userdebug
Nexus 9	volantis (flounder)	aosp_flounder-userdebug
Nexus 5	hammerhead	aosp_hammerhead-userdebug
Nexus 7 Wi-Fi	razor (flo)	aosp_flo-userdebug
Nexus 7 Mobile	razorg (deb)	aosp_deb-userdebug
Nexus 10	mantaray (manta)	full_manta-userdebug
Nexus 4	occum (mako)	full_mako-userdebug
Nexus 7 Wi-Fi	nakasi (grouper)	full_grouper-userdebug
Nexus 7 Mobile	nakasig (tilapia)	full_tilapia-userdebug
Galaxy Nexus GSM/HSPA+	yakju (maguro)	full_maguro-userdebug
Galaxy Nexus Verizon	mysid (toro)	aosp_toro-userdebug
Galaxy Nexus Experimental	mysidspr (toroplus)	aosp_toroplus-userdebug
PandaBoard	panda	aosp_panda-userdebug
Motorola Xoom	wingray	full_wingray-userdebug
Nexus S	soju (crespo)	full_crespo-userdebug
Nexus S 4G	sojus (crespo4g)	full_crespo4g-userdebug

Tabella 4.2: Tabella Build configuration per il comando lunch

Buildtype	Uso
<i>user</i>	accesso limitato
<i>userdebug</i>	come <i>user</i> ma con la possibilità di debug e privilegi di root
<i>eng</i>	destinata agli sviluppatori con funzionalità di debug aggiuntive

Tabella 4.3: Differenze tra le varie buildtype

Device name	Source location	Build configuration
Nexus 6	kernel/msm	shamu_defconfig
Nexus Player	kernel/x86_64	fugu_defconfig
Nexus 9	kernel/tegra	flounder_defconfig
Nexus 5	kernel/msm	hammerhead_defconfig
Nexus 7 Wi-Fi	kernel/msm	flo_defconfig
Nexus 7 Mobile	kernel/msm	flo_defconfig
Nexus 10	kernel/exynos	manta_defconfig
Nexus 4	kernel/msm	mako_defconfig
Nexus 7 Wi-Fi	kernel/tegra	tegra3_android_defconfig
Nexus 7 Mobile	kernel/tegra	tegra3_android_defconfig
Galaxy Nexus GSM/HSPA+	kernel/omap	tuna_defconfig
Galaxy Nexus Verizon	kernel/omap	tuna_defconfig
Galaxy Nexus Experimental	kernel/omap	panda_defconfig
PandaBoard	kernel/tegra	stingray_defconfig
Motorola Xoom	kernel/tegra	stingray_defconfig
Nexus S	kernel/samsung	herring_defconfig
Nexus S 4G	kernel/samsung	herring_defconfig

Tabella 4.4: Nomi e posizioni dei codici sorgenti del kernel

Algoritmo 4.12 Scaricare il codice sorgente del kernel msm Qualcomm

```
$ makedir ~/kernel_nexus5  
$ cd ~/kernel_nexus5  
$ git clone https://android.googlesource.com/kernel/msm/  
$ cd msm  
$ git checkout <branch>
```

Algoritmo 4.13 Configurazione di default e compilazione del kernel

```
$ make <device code name>_defconfig  
$ make -jN
```

Usare la `zImage` sbagliata impedisce l'avvio del sistema operativo e spesso causa un problema noto come *bootloop*, cioè il dispositivo continuerà a riavviarsi. Per scoprire se usare *DTB*, ho verificato se nel sorgente, la variabile *LOCAL_KERNEL* (nel file *device.mk*) contiene un percorso della `zImage` in versione *DTB* o no. Nel caso del Nexus 5 si può trovare in [44].

Per mantenere il dispositivo sbloccato, con i privilegi di root e poter installare solamente il kernel, ho deciso di non adottare la tecnica classica fornita da Google, tramite il comando *fastboot*, ma creare un “.zip” installabile direttamente dal dispositivo. Ovviamente, per installare Android e/o il kernel è richiesto lo sblocco del *Bootloader*, tramite una procedura detta “unlock”. Tutto ciò esula da questo documento, ma le procedure possono essere reperite facilmente su internet.

4.8 Inserire il kernel in uno zip installabile

Come anticipato nella sezione precedente, il mio obiettivo è creare un file “.zip” installabile contenente il kernel. I motivi sono i seguenti:

- la *community Android* è abituata ad utilizzare file “.zip” sia per installare *Custom ROM*, sia per *Custom kernel*;
- la maggior parte delle app che permettono di installare il kernel direttamente dal dispositivo richiedono file “.zip”, per esempio *ROM Manager* e *ROM Toolbox*.
- le più famose e diffuse *Custom Recovery* permettono di installare solo file “.zip”, per esempio *TWRP* e *ClockworkMod Recovery*;
- all'interno di un “.zip” è possibile anche inserire script personalizzati per modificare il processo di installazione.

Per semplicità sfrutterò un file “.zip” già disponibile della stessa versione del kernel che voglio installare e per lo stesso dispositivo. Questo permette di risparmiare tempo

```

androidsource examples.desktop Musica Scaricati
bin Immagini Nuova cartella Scrivania
ks89@ks89-pc:~$ cd androidK
bash: cd: androidK: File o directory non esistente
ks89@ks89-pc:~$ cd android
androidNdk/ androidsource/
ks89@ks89-pc:~$ cd androidKernel/msm/
ks89@ks89-pc:~/androidKernel/msm$ ls
android Documentation Kbuild mm sound
AndroidKernel.mk drivers Kconfig net tools
arch firmware kernel README usr
block fs lib REPORTING-BUGS virt
COPYING include MAINTAINERS samples
CREDITS init make_defconfig.sh scripts
crypto ipc Makefile security
ks89@ks89-pc:~/androidKernel/msm$ make mako_defconfig
warning: (ARCH_MSM_KRAITMP && ARCH_MSM_CORTEX_A5) selects HAVE_HW_BRKPT_RESERVED
_RW_ACCESS which has unmet direct dependencies (HAVE_HW_BREAKPOINT)
warning: (ARCH_MSM_KRAITMP && ARCH_MSM_CORTEX_A5) selects HAVE_HW_BRKPT_RESERVED
_RW_ACCESS which has unmet direct dependencies (HAVE_HW_BREAKPOINT)
#
# configuration written to .config
#
ks89@ks89-pc:~/androidKernel/msm$
```

Figura 4.14: Creazione configurazione di default per il kernel

```

arm-linux-androideabi-ld: warning: unwinding may not work because EXIDX input se
ction 27 of drivers/built-in.o is not in EXIDX output section
arm-linux-androideabi-ld: warning: unwinding may not work because EXIDX input se
ction 11 of sound/built-in.o is not in EXIDX output section
arm-linux-androideabi-ld: warning: unwinding may not work because EXIDX input se
ction 40 of net/built-in.o is not in EXIDX output section
    SYSMAP System.map
    SYSMAP .tmp_System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS arch/arm/boot/compressed/head.o
GZIP arch/arm/boot/compressed/piggy gzip
CC arch/arm/boot/compressed/misc.o
CC arch/arm/boot/compressed/decompress.o
CC arch/arm/boot/compressed/string.o
SHIPPED arch/arm/boot/compressed/lib1funcs.S
SHIPPED arch/arm/boot/compressed/ashldi3.S
AS arch/arm/boot/compressed/lib1funcs.o
AS arch/arm/boot/compressed/ashldi3.o
AS arch/arm/boot/compressed/piggy gzip.o
LD arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
ks89@ks89-pc:~/androidKernel/msm$ █
```

Figura 4.15: Kernel compilato

```

58. ** +-----+
59. ** | boot header | 1 page
60. ** +-----+
61. ** | kernel | n pages
62. ** +-----+
63. ** | ramdisk | m pages
64. ** +-----+
65. ** | second stage | o pages
66. ** +-----+
67. **
68. ** n = (kernel_size + page_size - 1) / page_size
69. ** m = (ramdisk_size + page_size - 1) / page_size
70. ** o = (second_size + page_size - 1) / page_size
71. **
72. ** 0. all entities are page_size aligned in flash
73. ** 1. kernel and ramdisk are required (size != 0)
74. ** 2. second is optional (second_size == 0 -> no second)
75. ** 3. load each element (kernel, ramdisk, second) at
76. **     the specified physical address (kernel_addr, etc)
77. ** 4. prepare tags at tag_addr. kernel_args[] is
78. **     appended to the kernel commandline in the tags.
79. ** 5. r0 = 0, r1 = MACHINE_TYPE, r2 = tags_addr
80. ** 6. if second_size != 0: jump to second_addr
81. **     else: jump to kernel_addr
82. */

```

Figura 4.16: Mkbootimg.h

e posso agire sul kernel senza perdere i privilegi di root in Android. Prima mi sono procurato uno di questi file. Per esperienze personali, i migliori per i miei dispositivi di riferimento (Nexus 4 e 5) sono quelli realizzati da “Francisco Franco” e “faux123”, disponibili rispettivamente in [30] e [27]. Sia per Nexus 4, sia per Nexus 5, ho scelto i kernel sviluppati da faux123, ma le procedure sarebbero equivalenti anche con quelli di Francisco Franco.

Per poter raggiungere il mio obiettivo ho dovuto separare il *kernel* e la *ramdisk*³ dal file *boot.img*, contenuto nel file “.zip” di “faux123”. Dopodiché, ho sostituito il *kernel* e re-impacchettato il tutto nel “.img” e poi nuovamente nel “.zip”. L’estrazione dal file “.img” non è un’operazione semplice, infatti richiede una perfetta gestione della posizione di *kernel* e *ramdisk* a livello degli indirizzi di memoria. Per capire meglio questo argomento fare riferimento al file *mkbootimg.h* nella *repository* di Android e alla Figura 4.16. Un modo per farlo è utilizzare due programmi, il primo chiamato *mkbootimg*, il cui sorgente è in *platform/system/core* [43] nella *repository* ufficiale di Android, mentre il secondo, chiamato *unmkbootimg*, si trova su Github [76]. Google fornisce *mkbootimg* in grado di svolgere anche la procedura del secondo, ma in modo estremamente complesso. Questo software non viene aggiornato da molto tempo, così uno sviluppatore ha deciso di modificarlo e crearne una versione semplificata. Prima di poterli usare, ho dovuto compilari.

³La RAM disk è una porzione di RAM utilizzata da un software come se fosse un’unità a dischi, cioè una memoria secondaria [105].

Compilare mkbootimg

Per compilare *mkbootimg* ho dovuto seguire la procedura nell’Algoritmo 4.14 ed infine ho riavviato il sistema operativo nella macchina virtuale.

Algoritmo 4.14 Compilare mkbootimg dalla repository di Android

```
$ mkdir ~/mkbootimg
$ cd ~/mkbootimg
$ git clone https://android.googlesource.com/platform/system/core
bootimg-tools
$ cd bootimg-tools/libmincrypt/
$ gcc -c *.c -I../include
$ ar rcs libmincrypt.a *.o
$ cd ..../mkbootimg
$ gcc mkbootimg.c -o mkbootimg -I../include
./libmincrypt/libmincrypt.a
$ sudo cp mkbootimg /usr/local/bin/
$ cd ..../cpio
$ gcc mkbootfs.c -o mkbootfs -I../include
$ sudo cp mkbootfs /usr/local/bin/
```

Compilare unmkbootimg

Per compilare *unmkbootimg* ho seguito la procedura nell’Algoritmo 4.15 ed infine ho riavviato nuovamente.

Algoritmo 4.15 Compilare unmkbootimg da Github

```
$ mkdir ~/unmkbootimg
$ cd ~/unmkbootimg
$ git clone https://github.com/pbatard/bootimg-tools.git
$ cd bootimg-tools/mkbootimg
$ gcc -o unmkbootimg unmkbootimg.c
$ chmod a+x unmkbootimg
$ sudo cp unmkbootimg /usr/local/bin/
```

Creare il file “.zip”

Grazie all’ultima riga degli Algoritmi 4.14 e 4.15, il terminale Linux è abilitato ai comandi *mkbootimg* e *unmkbootimg* e possono essere utilizzati ovunque, a patto di aver fatto un riavvio del sistema operativo come ho indicato. Dopo la compilazione di questi strumenti, aver fatto un *Nandroid backup* tramite la *modalità recovery* (procedura molto semplice, ma che esula da questo lavoro di tesi), ho svolto le seguenti operazioni.

1. Ho scaricato il faux123 kernel corretto da [27] ed ho estratto solo il `boot.img` dal file “.zip”.
2. Ho eseguito il comando `unmkbootimg -i boot.img` per ottenere i file `kernel` e `ramdisk.cpio.gz` (vedi Algoritmo 4.16 e Figura 4.17). Quindi, ho rimosso sia il file `kernel`, sia il `boot.img`.
3. Ho spostato il file compilato `zImage` (o `zImage-dtb` in base al dispositivo, per esempio Nexus 5 richiede la `dtb`) nella stessa cartella di `ramdisk.cpio.gz` e rinominato `zImage` in `kernel`.
4. Ho eseguito il comando mostrato automaticamente nel terminale da `unmkbootimg` (con la versione di Google, avrei dovuto inserire tutti questi parametri manualmente, dopo averli calcolati). Per esempio, per il Nexus 5 il comando da inserire su una sola riga del terminale è quello nell’Algoritmo 4.16.
5. Il risultato è un nuovo file `boot.img` contenente il `kernel` e la `ramdisk` di faux123. Quindi, ho inserito il `boot.img` nel file “.zip” per completare la procedura.
6. Come passo finale, ho rimosso tutti i file ad eccezione del “.zip” modificato e l’ho copiarlo nella `/sdcard` del dispositivo.

Algoritmo 4.16 Comando mkbootimg di Google per rimpacchettare il `boot.img` del Nexus 5

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

Installare il file “.zip” tramite TWRP Recovery Mode

Esistono diverse *modalità recovery*, io ho scelto *TWRP* su Nexus 5. La procedura è molto semplice, cioè entrare in *modalità recovery* cercando su internet la procedura corretta per lo specifico dispositivo (per i Nexus 4 e 5 bisogna premere contemporaneamente i tasti *accensione* e *volume-* quando il telefono è spento). Una volta avviata la *modalità recovery*, ho scelto la voce *install* dal menu e ho caricato il file “.zip” dalla `/sdcard`. Terminata la procedura, ho eseguito il *wipe di cache e dalvik cache* per evitare problemi all’avvio.

Durante gli esperimenti eseguiti ho avuto modo di stilare una lista di soluzioni ai problemi comuni.

```

ks89@ks89-pc:~/myzipkernel$ cd ..
ks89@ks89-pc:~$ ls
androidAltroSoftware androidsource Immagini Pubblici Video
androidKernel bin Modelli Scaricati
androidNdk Documenti Musica Scrivania
androidSdk examples.desktop myzipkernel unmkbootimg
ks89@ks89-pc:~$ cd myzipkernel/
ks89@ks89-pc:~/myzipkernel$ unmkbootimg
error: no input filename specified
usage: unmkbootimg
      [ --kernel <filename> ]
      [ --ramdisk <filename> ]
      [ --second <2ndbootloader-filename> ]
      -i|--input <filename>
ks89@ks89-pc:~/myzipkernel$ unmkbootimg -i boot.img
kernel written to 'kernel' (5046824 bytes)
ramdisk written to 'ramdisk.cpio.gz' (623181 bytes)

To rebuild this boot image, you can use the command:
  mkbootimg --base 0 --pagesize 2048 --kernel_offset 0x00008000 --ramdisk_offset
  0x02900000 --second_offset 0x00f00000 --tags_offset 0x02700000 --cmdline 'conso
le=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2 msm
_watchdog_v2.enable=1' --kernel kernel --ramdisk ramdisk.cpio.gz -o boot.img
ks89@ks89-pc:~/myzipkernel$ █

```

Figura 4.17: Unmkbootimg eseguito

- Se il dispositivo si accende ma resta tutto nero (con retroilluminazione dello schermo LCD attiva) e tende a scaldarsi molto velocemente, il problema può essere causato dall’uso della versione sbagliata del kernel, per esempio la `zImage` invece di usare la `zImage-dtb`;
- Se il dispositivo si accende, mostra il logo “Google” e si riavvia continuamente (*bootloop*), può essere dovuto ad un errore nel reimpacchettare il `boot.img` o la struttura del file “`.zip`” è stata alterata. I motivi che possono causare questo problema sono davvero molti. Mentre si opera con il kernel di Android, tale problema è frequentissimo e l’unico modo è continuare a provare ed imparare dai propri sbagli. Non c’è da preoccuparsi, in quanto è sufficiente reinstallare il kernel o nel peggior dei casi formattare tutto e re-installare Android dal file “`.zip`” del sistema operativo fornito dal produttore del telefono o da Google nel caso dei Nexus.
- Se il dispositivo si accende e resta bloccato per diversi minuti (Lollipop è più lento ad avviarsi dopo la cancellazione della cache, per sicurezza attendere anche 15 minuti prima di essere sicuri di aver commesso qualche errore) nella fase di avvio di Android, questo problema, solitamente, è causato dall’uso del kernel sbagliato. Soprattutto, avviene nel tentativo di installare il kernel per una certa versione di Android su una differente nel proprio dispositivo. Inoltre, può capitare anche nel caso non venga pulita la cache.

4.9 Configurare il kernel di Android

Prima della procedura di compilazione ho apportato alcune modifiche al Kernel. Ovviamente, si può modificare il codice sorgente, ma essendo un kernel Linux, ho potuto eseguire `make menuconfig` per personalizzarlo, aggiungendo funzionalità e moduli del kernel (che non hanno nulla a che vedere con i moduli di Android). Vi sono due metodi per farlo, il primo è modificare manualmente il file `.config`, l'altro è usare `make menuconfig` per caricare una sorta di interfaccia grafica nel terminale, come in Figura 4.18.

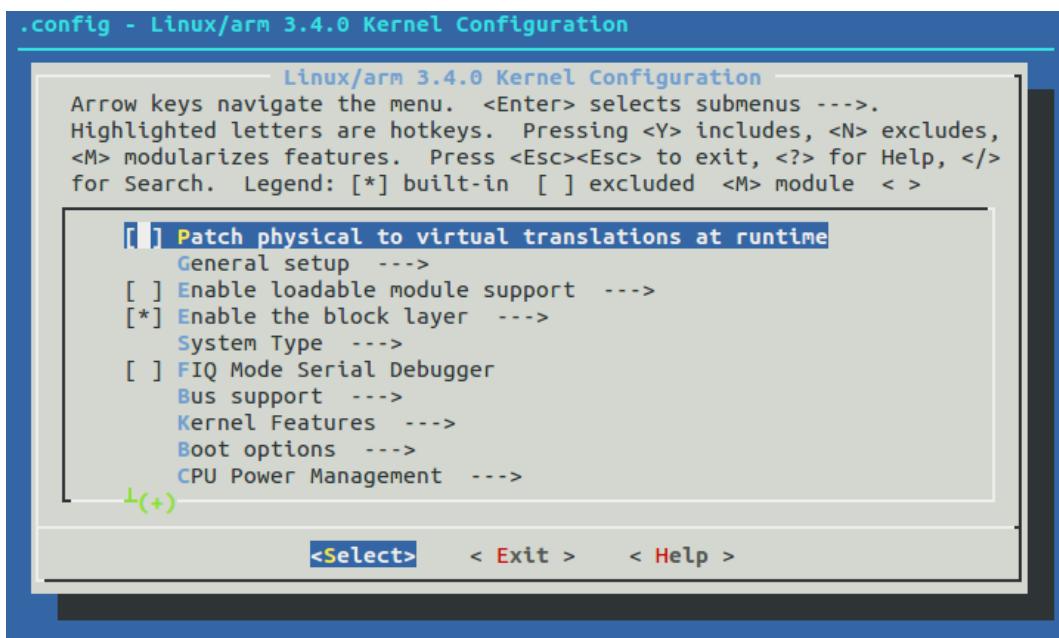


Figura 4.18: Configurazione kernel Linux

Il menu di configurazione è costituito da diverse sezioni, sottosezioni e impostazioni da abilitare/disabilitare e campi di testo da completare. Premendo il tasto “h” sulla tastiera sull’elemento del menu, si ottiene una breve descrizione del funzionamento e la lista di altre funzioni che saranno modificate automaticamente. In basso c’è anche il menu per scegliere il tipo di azione, cioè selezionare, uscire e mostrare l’aiuto. Con le frecce laterali si scorre, con barra spaziatrice si seleziona l’elemento e con invio si esplorano le voci del menu di configurazione. La voce selezione può essere usata più volte per modificare la modalità di inclusione, cioè [] è non incluso, [*] è incluso di default nel kernel e [M] è un modulo del kernel che deve essere caricato manualmente tramite il comando `insmod <Nome_Modulo.o>` sul dispositivo e ad ogni riavvio sarà necessario rieseguirlo (a meno di soluzioni particolari per far caricare automaticamente il modulo all’avvio che però esulano da questo lavoro di tesi).

Nel Capitolo 5 mostrerò come l’uso di tale menu di configurazione mi aiuterà ad

abilitare il supporto alle interfacce virtuali di rete in Android, ma per ora mi voglio concentrare solo su due impostazioni:

- *General Setup* per scegliere il nome del kernel personalizzato in (*-perf*) *Local version - append to kernel release*.
- *General Setup* per abilitare *Kernel .config support* e la sua sottovoce *Enable access to .config through /proc/config.gz*, poiché permette di vedere la configurazione del kernel direttamente sul dispositivo in cui è installato.

Terminata la configurazione, ho scelto *Exit* dal menu il basso e ho confermato con *Yes* per salvare la configurazione. Tutte le modifiche fatte in questo menu, modificano automaticamente il file *.config*. Quindi, usare il file manualmente o l'interfaccia grafica porta allo stesso risultato. A questo punto, ho compilato di nuovo il kernel per ottenere la ***zImage*** personalizzata.

Continuerò questa sezione nel Capitolo 5 per abilitare il supporto alle interfacce di rete virtuali, ma prima è necessario spiegare alcuni concetti che non ho ancora trattato.

Capitolo 5

Wpa_supplicant

In questo lavoro di tesi ho più volte accennato in modo superficiale il concetto di *Virtual network InterFace* (VIF), ma ora è necessario approfondire l'argomento, quindi ho deciso di mostrarne un'introduzione teorica basata su due progetti, il primo riguarda Windows e definisce il concetto stesso di “interfaccia virtuale di rete”, il secondo è un ottimo riferimento per Linux, a cui questo documento è più vicino. Poiché Android utilizza un kernel Linux modificato, è sempre valido il concetto di VIF, ma con importanti limitazioni che non riguardano in modo specifico *Wi-Fi Direct*, ma la gestione di qualunque tipo di VIF. Tale problema sarà spiegato nel dettaglio lungo questo capitolo, tramite *wpa_supplicant* e *iw* ed in modo ancora più approfondito nel successivo.

5.1 Interfacce di rete virtuali

Il concetto di “interfaccia di rete virtuale” o *Virtual network InterFace* (VIF) è stato presentato per la prima volta da Microsoft nel 2003 con il nome di *MultiNet* [8], con l'obiettivo di connettere un PC a più reti wireless contemporaneamente, questo perché avere più chip Wi-Fi sullo stesso dispositivo causa consumi di energia troppo elevati. L'idea è quella di virtualizzare una singola scheda di rete fisica, introducendo un livello intermedio al di sotto di quello IP (nello *stack di rete*) che alterna continuamente le connessioni tra le reti wireless. Il vantaggio principale è che i protocolli TCP/IP e le applicazioni non richiedono nessuna modifica. L'utilità delle VIF riguarda:

- la possibilità di connessione a più reti wireless;
- estensione della rete, cioè usare *Access Point* che estendono l'area di ricezione della rete;
- nodi *Gateway*, cioè un nodo connesso ad internet e contemporaneamente ad una *rete ad-hoc*;

- l'aumento delle capacità delle *reti ad-hoc*;
- l'utilizzo di macchine virtuali per connetterle a diverse reti fisiche.

Prima di questo progetto, una scheda di rete avrebbe potuto connettersi solo ad una rete fisica e soprattutto non avrebbe potuto interagire con altri dispositivi su frequenze diverse. Inoltre, un nodo di in una *rete ad-hoc* non avrebbe potuto interagire con altri, anche se sullo stesso canale. Questo perché IEEE 802.11 specifica diversi protocolli per la comunicazione tra IBSS e BSS e non si occupa della sincronizzazione.

MultiNet definisce la virtualizzazione di una scheda di rete tramite l'astrazione di più schede che sono adattatori sempre attivi su una singola scheda di rete fisica. Questo lo si ottiene facendo *multiplexing*¹ della scheda di rete attraverso più reti wireless, cioè si alternano le varie interfacce usando uno slot temporale detto *Active Period* per ogni rete operante su un particolare canale. La somma degli *Active Period* di tutte le reti connesse è chiamato *Switching Cycle*. L'architettura di base su *MultiNet Protocol Driver* che è un livello intermedio tra IP a MAC. Il driver espone adattatori di rete multipli sempre attivi (uno per ogni rete wireless) e permette che vi sia un IP diverso assegnato ad ogni rete. Idealmente, ogni adattatore di rete esposto dal driver dovrebbe avere anche un MAC address diverso.

Utilizzare *MultiNet* rispetto ad avere più schede Wi-Fi fa sì che i consumi energetici siano inferiori. Infatti, anche quando i chip sono in stato di *idle*, consumano comunque di più (circa il doppio) di *MultiNet*. Usando la *Power Save Mode* (PSM) si riesce a ridurre il consumo della soluzione a due chip, diventando paragonabile a *MultiNet*, ma ciò riduce il throughput e quindi le performance diminuiscono.

Questo progetto è nato su Windows XP, ma poi ci sono state diverse soluzioni in Linux come per esempio *Juggler* per il kernel Linux 2.6. In *Juggler* [4] si alternano le reti wireless secondo un algoritmo *Round Robin*, aggiornando continuamente SSID, BSSID e la frequenza nella scheda di rete fisica. *Juggler* usa alcuni buffer per salvare i pacchetti di rete e il dispositivo virtuale, o *pseudodevice*, impersona un'interfaccia di rete cablata ethernet, con un IP privato statico e un MAC Address Ethernet, per distinguerlo da interfacce fisiche di rete. Se *Juggler* deve passare da una rete ad un'altra ed è su una frequenza diversa, deve essere cambiata nella scheda di rete fisica, insieme a SSID, MAC address e modalità di rete. Ho spiegato questo concetto perché Android è basato su kernel Linux, quindi supporta le interfacce di rete virtuali, anche se in modo molto limitato e perché da questo capitolo parlerò molte volte di tali argomenti.

¹Multiplexing è la tecnica per cui più canali trasmissivi in ingresso condividono la stessa capacità trasmissiva disponibile in uscita [100].

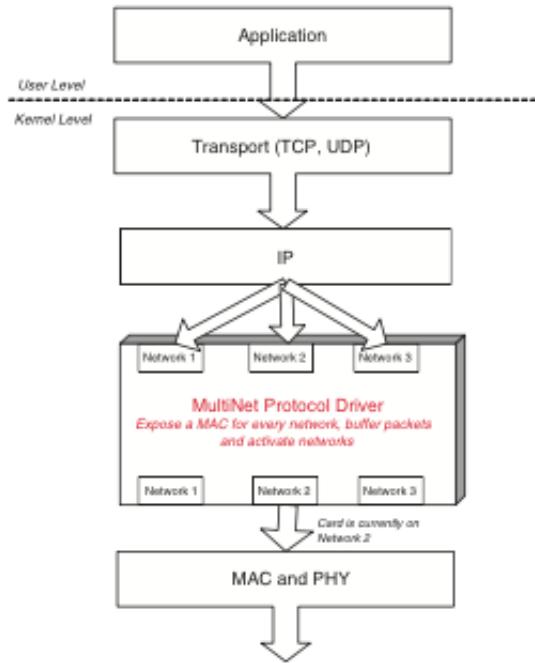


Figura 5.1: Stack di rete modificato con Microsoft MultiNet

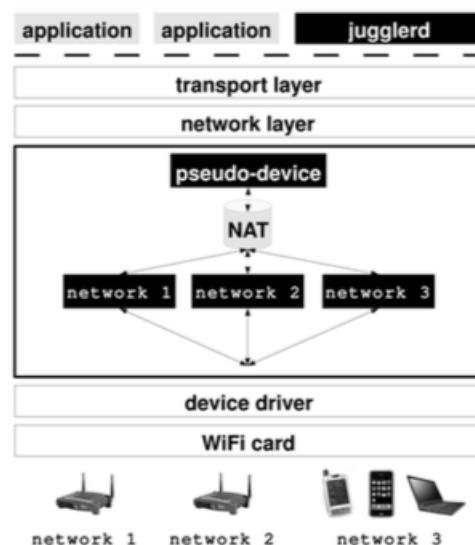


Figura 5.2: Stack di rete modificato con Juggler

5.2 Wpa_supplicant

Dopo il tentativo di estendere o modificare le API di Android nel Capitolo 3, per i motivi già spiegati, ho deciso di cambiare totalmente approccio. Esso non si basa più sull’interazione con il framework Android, ma sul livello “nativo nell’architettura di Android”, come in Figura 4.2. Cioè, nella parte in cui non si utilizza direttamente Java, poiché vi sono programmi scritti principalmente in C/C++. Uno di questi è *wpa_supplicant*. Ho già accennato questo software nel Capitolo 3 per esplorare la soluzione con NDK (Sezione 3.4) e usarlo per un esempio pratico, ma ora approfondirò l’argomento.

Jouni Malinen tra il 2002 e il 2003 ha sviluppato un programma chiamato *hostapd* e i *driver Host AP* con l’obiettivo di creare un *daemon*² nello spazio dell’utente (*User space*) per la gestione degli *AP* in Linux. Ancora oggi questo programma è supportato ed aggiornato. Nel 2004 ha realizzato anche *wpa_supplicant*, cioè un’altro *daemon*, ma dedicato più al lato *Client* rispetto a quello *AP*. Anch’esso continua ad essere aggiornato da una community vasta di sviluppatori indipendenti e di società come Google, Sony, Intel, Broadcom, ma la principale è Qualcomm per cui Malinen stesso lavora.

wpa_supplicant, come dice il nome, è un *supplicant*, cioè un’entità (in questo caso è un software) ad un’estremità di un segmento point-to-point di rete locale che cerca di essere autenticato da un *autenticatore* all’altro estremo di questo collegamento. Normalmente un *supplicant* è software installato sui computer degli utenti finali. L’utente invoca il *supplicant* ed invia le credenziali per connettere il computer ad una rete sicura. Se l’autenticazione ha successo, tipicamente l’*autenticatore* permette al computer di connettersi alla rete. Quindi, in un certo senso, un *supplicant* si riferisce ad un utente o un client di una rete che tenta di accedere a risorse protette da un meccanismo di autenticazione IEEE 802.1X [107]. *Wpa_supplicant* è un software multipiattaforma per Linux, BSD, Mac OS X e Windows con supporto per WPA e WPA2 (IEEE 802.11i / RSN) e può essere usato sia per desktop/laptop, sia per sistemi embedded [5, 71].

Questo software è stato progettato per essere eseguito come un *daemon* che resta in esecuzione in background e controlla la connessione wireless. Quindi, agisce da *back-end* e supporta software di *front-end* o a linea di comando come *wpa_cli* o ad interfaccia grafica come *wpa_gui*. *Wpa_supplicant* usa un processo di compilazione per selezionare quali funzioni includere, proprio come il kernel Linux e questo permette di generare un eseguibile molto piccolo [107].

Principalmente è utilizzato in distribuzioni Linux, infatti, quando è nato Android, cioè un sistema operativo basato su un kernel Linux modificato, è stato utilizzato co-

²Un demone (daemon in inglese) è un programma eseguito in background, cioè senza che sia sotto il controllo diretto dell’utente, tipicamente per fornire un servizio [91].

me base per tutte le operazioni Wi-Fi di questi dispositivi. Oggi, tutti gli Smartphone e Tablet Android con un chip Wi-Fi utilizzano questo *daemon* per le operazioni di rete Wi-Fi. Con la standardizzazione di *Wi-Fi Direct*, anche *wpa_supplicant* è stato aggiornato supportando questo tipo di protocollo. Ad oggi, in Android, usa lo stesso codice sorgente della versione per Linux, l'unica differenza è che la compilazione viene gestita dall'ABS, tramite `Android.mk` che ne configura le funzionalità. Poiché Google partecipa a questo progetto, fornisce supporto ufficiale alla compilazione per Android con il file `Android.mk` corretto, che utilizzerò in seguito per ottenere gli eseguibili di *wpa_supplicant* e *wpa_cli*.

Come ho già accennato, *wpa_supplicant* si trova nel livello nativo dell'architettura di Android e di conseguenza è necessario “parlare” con questo software tramite JNI, operazione che Android svolge internamente, ma con una struttura complessa e disordinata. Io ho pensato ad una soluzione più semplice e più facile da estendere in futuro, cioè utilizzare i software di *front-end*, come *wpa_cli* per “parlare” con *wpa_supplicant* tramite linea di comando in Android. Il risultato che otterrei, è un software che espone le funzioni di *wpa_supplicant* all'utente, bypassando tutti i limiti del framework Android visti nel Capitolo 3.

File di configurazione

Wpa_supplicant è configurato usando un file di testo con una lista di tutte le reti e i parametri che deve utilizzare. Il file in questione si chiama `wpa_supplicant.conf`. Su Android ce n’è un altro chiamato `p2p_supplicant.conf`. Il primo è utilizzato per avviare *wpa_supplicant* sull’interfaccia di rete `wlan0`, mentre il secondo sulla `p2p0` (per *Wi-Fi Direct*).

Il file d'esempio con questi parametri è disponibile in [72], ma per avere una visione veramente chiara dell'argomento ho dovuto esplorare il codice sorgente, fino a trovare il file che esegue il parsing di questi comandi, per capirne la sintassi e i valori assegnabili. `Wpa_supplicant.conf` [72] d'esempio nella *repository Git* ufficiale costituisce anche la documentazione, ma data la dimensione di questo file e dal numero di parametri utilizzabili, ho scelto di concentrarmi su quelli che ho trovato più utili per i miei scopi.

1. *network = { (...) }* blocco che specifica una nuova rete wireless salvata con tutti i parametri di configurazione per permettere, per esempio, la riconnessione automatica. Android non gestisce questo autonomamente, ma lascia che sia *wpa_supplicant* a farlo, tramite questo file di configurazione che aggiorna ogni volta che ci si connette ad una nuova rete.
2. *update_config=1* permette a *wpa_supplicant* di aggiornare il file di configurazione, cioè di sovrascriverlo. Questo è molto utile, altrimenti l'aggiunta di

5. Wpa_supplicant

blocchi *network* o il loro aggiornamento non sarebbe possibile. In Android è attiva di default, sia in `p2p_supplicant.conf`, sia in `wpa_supplicant.conf`.

3. *ctrl_interface=<percorso assoluto>* indica il percorso assoluto della cartella in cui sono presenti i socket utilizzati da *wpa_cli* per comunicare col processo di *wpa_supplicant*. Il percorso predefinito in Linux è `/var/run/wpa_supplicant`, mentre in Android è `/data/misc/wifi/sockets`. In quest'ultimo, tale parametro non è presente di default, semplicemente perché Google non fornisce nel sistema operativo compilato *wpa_cli*. Infatti, si appoggia a socket creati da Android stesso, che non possono essere utilizzati con *wpa_supplicant*.
4. *ap_scan=<valore intero>* permette di stabilire la modalità di scansione. Infatti *wpa_supplicant* richiede al driver di eseguire la scansione degli AP. Il comportamento cambia in base al suo valore. Ci sono tre possibili opzioni.
 - (a) =1: di default, *wpa_supplicant* inizia a scansionare e seleziona l'AP. Se non trova un AP associato alle reti salvate può essere inizializzata una nuova rete *ad-hoc IBSS* o in modalità AP, se configurate.
 - (b) = 0: il driver si occupa della scansione, selezione dell'AP e dei parametri di associazione IEEE 802.11.
 - (c) = 2: come 0, ma associa l'AP usando un sistema di sicurezza e un SSID (ma non BSSID). In questa modalità, i blocchi di rete nel file di configurazione sono provati uno ad uno, fino a che il driver non avvisa che l'associazione è avvenuta con successo. Ogni blocco di rete deve specificare esplicitamente le politiche di sicurezza.
5. *driver_param="field=value"* per passare parametri al driver. Il formato è specificato dall'interfaccia del driver. In molti casi questo non serve, ma come mostrerò nel proseguo del capitolo, nel caso di Android ho usato proprio questo parametro.
6. *config_methods=<lista parametri separati da spazio>* cioè la lista di tutte le modalità di configurazione supportate. I parametri disponibili sono *usb*, *ether*
net, *label*, *display*, *ext_nfc_token*, *int_nfc_token*, *nfc_interface*, *push_button*, *keypad*, *virtual_display*, *physical_display*, *virtual_push_button*, *physical_push_button*. In Android, questi parametri sono già impostati e non è stato necessario modificarli.

Supporto

Sebbene esista una documentazione ufficiale, non è neanche lontanamente sufficiente per usare correttamente *wpa_supplicant*. Inoltre, il codice sorgente è molto

esteso, infatti il solo file *wpa_supplicant.c* è di più di 10.000 righe e per la maggior parte senza nemmeno una riga di commento, quindi, di fatto, l'unico vero esperto è il creatore stesso, cioè Jouni Malinen. Però, per fortuna esiste una mailing list pubblica, molto simile a quella usata dal kernel Linux, in cui è possibile porre domande e spesso a rispondere è proprio Jouni in persona, oltre ad altri sviluppatori, per esempio di Qualcomm e Intel. Purtroppo, la maggior parte di loro lavora su Linux e il supporto che forniscono è destinato a questo sistema, lasciando a me il riadattamento ad Android, per nulla scontato e banale.

Nel proseguo di questo capitolo parlerò solamente di *wpa_supplicant* in Android. Le procedure pratiche riguardano la mia personale esperienza. Purtroppo, non ho trovato nessun supporto specifico per *wpa_supplicant* in Android.

5.3 Compilare *wpa_supplicant* e *wpa_cli*

Negli anni sono uscite molte versioni di questo software. In Android Lollipop 5.0.1 è presente la 2.3, mentre in KitKat la 2.1. Per quanto riguarda *Wi-Fi Direct* conviene avere l'ultima implementazione disponibile (*branch master*), per assicurarsi di avere quella con tutti i miglioramenti possibili, poiché come già ripetuto, *Wi-Fi Direct* è un protocollo giovane e ogni miglioramento è ben accetto. In Lollipop 5.0.1 ho fatto esattamente questo, cioè ho utilizzato una versione ancora in sviluppo circa a metà tra la 2.3 e la 2.4 (appena uscita). Per fare ciò ho scaricato dalla *repository* ufficiale di *wpa_supplicant* il codice sorgente (vedi Algoritmo 5.1 righe 1-3). Dopodiché, l'ho sostituito a quello presente in AOSP (righe 4-18) mantenendo la struttura con i due collegamenti simbolici e ho compilato solamente i moduli *wpa_supplicant* e *wpa_cli* (vedi Algoritmo 5.1 righe 19-22). La procedura completa è nell'Algoritmo 4.10 e il risultato della compilazione sono diversi file, comprese alcune librerie “.so”. Sono sufficienti solo i due eseguibili *wpa_supplicant* e *wpa_cli*, copiandoli dalla cartella *out/target/product/<nomedispositivo>/system/bin/* alla */sdcard* del dispositivo.

5.4 Eseguire *wpa_supplicant* e *wpa_cli*

Dopo che ho compilato *wpa_supplicant* e la sua interfaccia a linea di comando *wpa_cli*, li ho messi sul dispositivo e li ho eseguiti. Purtroppo non è sufficiente fare copia-incolla dei file per eseguirli facilmente. Innanzitutto, è obbligatorio un dispositivo con i privilegi di root, ma soprattutto la una procedura da seguire ha una variante nel caso di Android 5.0 o superiore.

5.4.1 Inizializzazione

Per poter procedere ho dovuto installare l'Android SDK perché contiene il programma *adb*. Quindi, ho scaricato l'SDK e seguito i comandi nell'Algoritmo 5.2.

Algoritmo 5.1 Download e compilazione di wpa_supplicant e wpa_cli

```
1. $ mkdir ~/wpa_supplicant_master
2. $ cd ~/wpa_supplicant_master
3. $ git clone http://w1.fi/hostap.git
4. $ cd ~/androidsource/aosp-main/external/wpa_supplicant_8
5. $ rm -r hs20
6. $ rm -r src
7. $ rm -r wpa_supplicant
8. $ rm -r hostapd
9. $ rm Android.mk
10. $ cp -r ~/wpa_supplicant_master/hostap/hostapd
     ../hostapd
11. $ cp -r ~/wpa_supplicant_master/hostap/src ../src
12. $ cp -r ~/wpa_supplicant_master/hostap/wpa_supplicant
     ../wpa_supplicant
13. $ cp -r ~/wpa_supplicant_master/hostap/hs20 ../hs20
14. $ cp ~/wpa_supplicant_master/hostap/Android.mk
     ../Android.mk
15. $ cd wpa_supplicant
16. $ ln -s ~/androidsource/aosp-main/external/
     wpa_supplicant_8/src
17. $ cd ../hostapd
18. $ ln -s ~/androidsource/aosp-main/external/
     wpa_supplicant_8/src
19. $ cd ~/androidsource/aosp-main
20. $ source build/envsetup.sh
21. $ lunch
22. $ make wpa_supplicant wpa_cli -jN
```

Algoritmo 5.2 Procedura per installazione di Android SDK

```
$ mkdir ~/androidSdk  
$ cd ~/Scaricati  
$ mv android-sdk_r24.0.2-linux.tgz  
./androidSdk/android-sdk_r24.0.2-linux.tgz  
$ cd ../androidSdk  
$ tar zxvf android-sdk_r24.0.2-linux.tgz  
$ rm android-sdk_r24.0.2-linux.tgz  
$ cd tools  
$ android
```

Dopo che ho installato gli elementi consigliati, nella cartella `~/androidSdk/platform-tools/` è apparso il file eseguibile `adb`. Quindi, ho collegato il dispositivo al PC/-MAC, ho attivato la *modalità debug* nelle impostazioni e ho copiato i file compilati nella `/sdcard`. Dopo queste operazioni, ho seguito la procedura di inizializzazione descritta nell’Algoritmo 5.3. Per far sì che il comando `adb` sia individuato dal terminale di Ubuntu, ho dovuto aggiungere nel file `.profile` la riga `export PATH=$PATH:~/androidSdk/platform-tools` e riavviare la macchina virtuale.

A questo punto, ho seguito la procedura nell’Algoritmo 5.3 (inserendo solo i comandi dopo ai simboli `$` e `#`, ovviamente). Questa sequenza di operazioni apre la *shell* di `adb` per poter interagire con il dispositivo (riga 1), si esegue coi privilegi di root (riga 2) e fa il *mount* e il *remount* di `/system` in modalità lettura e scrittura per poter modificare i file (riga 4). La riga 3 serve per visualizzare la lista delle partizioni in Android, utile per individuare quella che nel nome termina con “`/system`”. Una volta scoperta, ho copiato il nome che per esempio su Nexus 5 è `/dev/block/platform/msm_sdcc.1/by-name/system` a fianco del comando in riga 4. Dopodiché, la riga 6 crea un backup di `wpa_supplicant` pre-installato e copia in `/system/bin` i due file compilati nell’Algoritmo 5.1 (righe 7-8). Infine, imposta i permessi, il proprietario e il contesto *SELinux* di tali file ai valori di default (righe 9-13).

SELinux

NSA Security-Enhanced Linux (SELinux) fornisce un sistema di controllo degli accessi a grana fine (in inglese fine-grained). È utilizzato in due modi, *permissive mode* in cui le violazioni alle regole sono registrate, ma senza prendere azioni, ed *enforcing mode* in cui non sonomesse violazioni. SELinux è stato introdotto in Android 4.3 in *permissive mode*, poi in *enforcing mode* in Android 4.4, ma comunque la maggior parte dei dispositivi con 4.4 hanno continuato a funzionare con la precedente modalità. Da 5.0 la *enforcing mode* è diventata attiva di default. Quando funziona con la prima non ci sono problemi, ma se in *enforcing mode* la parte dell’app

Algoritmo 5.3 Preparazione per eseguire wpa_supplicant e wpa_cli

```
1. $ adb shell
   * daemon not running. starting it now on port 5037 *
   * daemon started successfully *
2. $ su
3. # mount
4. # mount -o remount,rw <nome completo di /system>
5. # cd /system/bin
6. # mv wpa_supplicant wpa_supplicant_backup
7. # cp /sdcard/wpa_supplicant wpa_supplicant
8. # cp /sdcard/wpa_cli wpa_cli
9. # chmod 755 wpa_supplicant
10. # chmod 755 wpa_cli
11. # chown 0.2000 wpa_supplicant
12. # chown 0.2000 wpa_cli
13. # chcon u:object_r:wpa_exec:s0 wpa_supplicant
```

che richiede i privilegi di root può avere comportamenti non voluti ed inaspettati.

Per verificare da un'app la presenza di SELinux si può leggere */sys/fs/selinux/enforce* tramite *libsuser* con *Shell.SU.shell()*. A definire quali sono le regole di sicurezza applicate al processo è il *Contesto SELinux* (in inglese *SELinux Context*) [21]. Alcuni contesti sono:

- u:r:init:s0 - Contesto *init* più elevato;
- u:r:shell:s0 - Shell senza privilegi, come *adb shell*;
- u:r:system_app:s0 - App di sistema;
- u:r:recovery:s0 - Recovery;
- u:object_r:wpa_exec:s0 - Wi-Fi.

Questo argomento, che riguarda la sicurezza informatica, è molto vasto e richiederebbe una trattazione più ampia, ma non fa parte di questo lavoro di tesi. Le uniche informazioni che ho fornito sono quelle essenziali per poter comprendere il concetto e non ne sono necessarie altre.

5.4.2 Configurazione di wpa_supplicant

Tentare di eseguire *wpa_supplicant*, senza un'accurata configurazione causerà errori, tipicamente con codice “255” o messaggi come “segmentation fault”. I file di configurazione *wpa_supplicant.conf* e *p2p_supplicant.conf* sono in */data/misc/wifi*. Una volta eseguita la procedura di inizializzazione, ho utilizzato un file manager con privilegi di root dal Google PlayStore per modificare questi file.

Algoritmo 5.4 wpa_supplicant.conf

```
ctrl_interface=/data/misc/wifi/sockets
disable_scan_offload=1
update_config=1
device_name=occam
manufacturer=LGE
model_name=Nexus 4
model_number=Nexus 4
serial_number=XXXXXXXXXXXXXX
device_type=10-0050F204-5
config_methods=physical_display virtual_push_button
pmf=1
tdls_external_control=1
```

Algoritmo 5.5 p2p_supplicant.conf

```
ctrl_interface=/data/misc/wifi/sockets
disable_scan_offload=1
driver_param=use_p2p_group_interface=1
update_config=1
device_name=Nexus 4
device_type=10-0050F204-5
config_methods=virtual_push_button physical_display keypad
p2p_ssid_postfix=-Nexus 4
persistent_reconnect=1
pmf=1
```

Negli algoritmi 5.4 e 5.5 ho messo i file di configurazione utilizzati in KitKat su Nexus 4. Le righe cruciali, per il corretto funzionamento di *wpa_supplicant* in Android, sono *ctrl_interface* (il percorso in cui si trovano i socket) e *driver_param* (descriverò questo parametro nel corso del capitolo).

5.4.3 Avvio di wpa_supplicant e wpa_cli

A questo punto, dopo essermi assicurato di aver il Wi-Fi abilitato nelle impostazioni di Android, ho ripetuto i passi da 1 a 5 dell’Algoritmo 5.1 e seguito la procedura nell’Algoritmo 5.6. La prima riga è utile solo su Lollipop e serve per cambiare il contesto *SELinux* di *wpa_supplicant*, mentre la seconda serve per terminare in modo forzato il processo *wpa_supplicant* in esecuzione (mi è capitato di doverla eseguire più volte, perché il processo si interrompesse). La terza riga per avviarlo eseguendolo su un solo processo, ma con due interfacce contemporaneamente, cioè *wlan0* per quanto riguarda il Wi-Fi e *p2p0* per la parte *Wi-Fi Direct*. Per una guida completa sui parametri che si possono passare a questo programma, consiglio di eseguire il comando *wpa_supplicant -help*, ma i principali e quelli rilevanti in questo lavoro di

5. Wpa_supplicant

Algoritmo 5.6 Avviare wpa_supplicant e wpa_cli

```
1. # chcon u:object_r:system_file:s0 wpa_supplicant
2. # killall -SIGKILL wpa_supplicant
3. # wpa_supplicant -B -iwlan0 -Dnl80211
   -c /data/misc/wifi/wpa_supplicant.conf -N -B
   -ip2p0 -Dnl80211 -c /data/misc/wifi/p2p_supplicant.conf
4. # wpa_cli -ip2p0 -p /data/misc/wifi/sockets
```

Algoritmo 5.7 wpa_cli avviato in “interactive mode”

```
wpa_cli v2.1-devel-4.4.4
Copyright (c) 2004-2013, Jouni Malinen <j@w1.fi> and contributors
This software may be distributed under the terms of the BSD
license.
See README for more details.
Interactive mode
>
```

tesi sono:

- -B : necessario per eseguirlo come *daemon*;
- -i<interfaccia di rete> : per specificare l’interfaccia su cui eseguirlo;
- -D<driver> : specifica il driver da utilizzare (in Android usare sempre *nl80211*, perché supportato e più recente degli altri);
- -c : specifica il file di configurazione;
- -N : esegue i comandi successivi su un altro processo, internamente a *wpa_supplicant*, facendo sì che col comando *ps w* appaia solo un processo col nome *wpa_supplicant*.

Detto ciò, ho avviato *wpa_cli*, tramite la procedura riportata nell’Algoritmo 5.6 (riga 4), specificando sempre l’interfaccia di rete con “-i”, con l’unica condizione che essa sia stata precedentemente avviata con *wpa_supplicant* in modo manuale o automatico. Il parametro “-p” specifica il percorso dei socket, necessario per far sì che il processo *wpa_cli* possa comunicare con quello di *wpa_supplicant*. Aggiungendo “-dd” dopo ogni comando di *wpa_supplicant* separato da “-N”, permette di abilitare la modalità con più messaggi di debug, ma dalla mia esperienza non sempre funziona, dipende dal dispositivo. Inoltre, *wpa_cli* ha due tipi di modalità di avvio: la *interactive mode* e la modalità con passaggio di parametri. Per tutto questo lavoro di tesi ho usato la *interactive* perché più comoda, ma per esporre le funzionalità di *wpa_cli* in Java dovrò usare l’altra modalità, oltre ad avere i privilegi di root e appoggiarmi alla libreria *libsuperuser* di *Jorrit Jongma*, conosciuto nell’ambiente del modding ed hacking di Android come “Chainfire” [20].

Attenzione, a causa delle nuove misure di sicurezza di Lollipop è necessario modificare il *contesto SELinux* di *wpa_supplicant* in *u:object_r:system_file:s0*, impedendo al framework Android di connettersi ad internet. Terminate le operazioni, è necessario ripristinare quello di default *u:object_r:wpa_exec:s0* per riabilitare il Wi-Fi in Android. Quasi sempre questa operazione richiede il riavvio completo del dispositivo.

5.5 Utilizzare wpa_supplicant e wpa_cli

Dopo aver avviato *wpa_cli*, ho potuto utilizzarlo per connettere più dispositivi. Innanzitutto, prima di procedere ho dovuto eseguirlo su tutti, che nei test effettuati erano Nexus 5 e Samsung Galaxy S4. Per una lista completa dei comandi e di tutti i parametri, che possono essere utilizzati, non ho fatto riferimento né all'help di *wpa_cli*, né alla documentazione fornita nei file *readme*, poiché aggiornati raramente. L'unico metodo che ho trovato per capire come utilizzare questo programma è leggere il sorgente del file *wpa_cli.c*, disponibile nella *repository* ufficiale, concentrandomi sul parsing delle stringhe passate come parametri.

Connessione di due dispositivi

Il primo passo è avviare la fase di Discovery su entrambi i dispositivi con *p2p_find*. Appena un dispositivo ne trova un altro apparirà un messaggio: *P2P-DEVICE-FOUND* con il MAC address, il nome e altri parametri. A volte è capitato che un dispositivo non mostrasse i messaggi *P2P-DEVICE-FOUND*, nonostante fosse comunque in grado di farlo, in particolare con i Samsung Galaxy S4. Purtroppo, è sempre accaduto in modo imprevedibile, l'unica soluzione che ho trovato è utilizzare il comando *p2p_peers* per ottenere la lista dei dispositivi rilevati. Quindi, ho usato questi risultati per ottenere il MAC address e procedere con la connessione. Il comando *p2p_find*, in realtà, permette di specificare molti più parametri, infatti la sintassi è:

```
p2p_find [timeout in seconds] [type=<social|progressive>] \
[dev_id=<addr>] [dev_type=<device type>] \
[delay=<search delay in ms>] [seek=<service name>] [freq=<MHz>].
```

È importante notare che senza specificare il *timeout*, la *fase di Discovery* continuerà all'infinito, o fino a che sarà sospesa da un altro comando. Dai test effettuati, dopo alcuni minuti, i dispositivi sono stati comunque in grado di trovarsi a vicenda, cosa che non accade con le API P2P in Android. Nel caso di *wpa_supplicant* ho notato molti meno problemi, per quanto riguarda la *fase di Discovery*, rispetto al framework Android.

Una volta che i due dispositivi si sono trovati, su uno dei due ho usato il comando *p2p_connect* per la connessione, specificando il MAC address di destinazione e il *wps_method*³. In questo lavoro di tesi, ho utilizzato il metodo *Push Button Configuration* (PBC), perché più comodo da utilizzare, ma ho sperimentato anche il PIN. Ho deciso di usare PBC, perché nell'ambito delle *Opportunistic Networks* è più adatto, poiché richiede la semplice pressione di un pulsante sullo schermo dello Smartphone di destinazione (anche se l'ideale sarebbe non averlo proprio e creare connessioni senza chiedere il consenso, cosa comunque possibile tramite Java reflection secondo [73]). Il comando utilizzabile per eseguire la connessione ha la sintassi:

```
p2p_connect <peer device address>
<pbc|pin|PIN#|p2ps> [display|keypad|p2ps]
[persistent|persistent=<network id>] [join|auth]
[go_intent=<0..15>] [freq=<in MHz>] [ht40] [vht]
[provdisc] [auto].
```

Anche in questo caso, si possono specificare molti parametri. Ho testato i principali in alcuni esperimenti, ma quelli che si sono rilevati utili sono: *wps_method*, *join*, *go_intent*⁴ e *persistent*. Poiché, in ambito di *Opportunistic Networks* è preferibile un gruppo *autonomous* o comunque standard, il *persistent* non lo considererò, questo basandomi sull'analisi delle performance in [73]. Principalmente ho sperimentato le soluzioni classiche perché più facili da gestire. Infatti la connessione è relativamente semplice, cioè ho utilizzato il comando *p2p_connect MAC_address_destinazione pbc go_intent=15* per eseguire la connessione facendo sì che sia il GO. Lo stesso comando l'ho eseguito anche sul dispositivo di destinazione, ma specificando il MAC address del precedente e un *go_intent* minore, per esempio 1, in modo da assicurarmi che fosse il Client. Ovviamente, forzare chi è il Client e chi è il GO non è necessario, ma si è rivelato molto comodo in fase di sperimentazione. Una volta avviata la connessione, come fa il dispositivo a cui è diretto il comando *p2p_connect* a sapere il MAC address del mittente? Semplicemente, lo riceve in un messaggio *P2P-PROV-DISC-PBC-REQ* che contiene i dati del dispositivo che sta richiedendo una connessione. In base ad essi, il destinatario può adattarsi e rispondere di conseguenza. Una volta stabilita la connessione, l'ho mantenuta per diversi minuti, per verificare che funzionasse senza alcun problema. La situazione è rappresentata in Figura 5.3.

Connessione di tre o più dispositivi

Dopo aver connesso due dispositivi, ho deciso di collegarne un terzo al GO. Purtroppo, fare ciò non è semplice come con due, infatti *wpa_cli* utilizza un'interfaccia

³Metodo da usare per WPS, per esempio la pressione di un pulsante o un codice PIN [3].

⁴Il valore intero più alto assegnato, specifica chi sarà il più probabile GO durante la negoziazione [3].

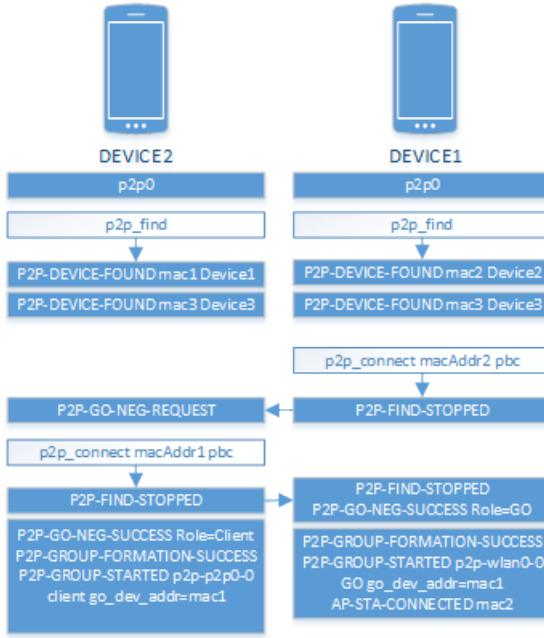


Figura 5.3: Connessione di due dispositivi con *wpa_supplicant*

principale per eseguire le operazioni e ne crea altre, cioè quelle del gruppo, con nomi simili a *p2p-p2p0-0* o *p2p-wlan0-4*. L'attuale implementazione di *wpa_cli* richiede l'esecuzione di alcuni comandi in quella principale, mentre altri in quelle di gruppo (gli sviluppatori hanno recentemente pianificato di aggiungere la funzione per lanciare i comandi su una sola interfaccia, come tra l'altro ho consigliato anche io nella mailing list dopo una lunga serie di esperimenti [17]). Quindi, una volta connessi due dispositivi, per far sì che un terzo possa collegarsi al GO del gruppo, bisogna avviare *wps_pbc* sul GO, non sull'interfaccia principale, ma su quella di gruppo [70]. Purtroppo, non si può specificare il nome dell'interfaccia di gruppo in *wps_pbc*, quindi ho dovuto aprire un'altra connessione con *wpa_cli* usando *-i<interfaccia_gruppo>* (per esempio *-ip2p-p2p0-0*) ed ho eseguito *wps_pbc*. Invece, sul dispositivo che vuole connettersi, ho utilizzato *p2p_connect MAC_address_GO pbc join*⁵.

Il risultato ottenuto sono visibili nelle Figure 5.4, 5.5, 5.6, 5.7, 5.8. Tale metodo può essere utilizzato anche senza l'*interactive mode* e questo è perfetto per la comunicazione tra più dispositivi e teoricamente permetterebbe anche di scalare con l'uso di interfacce multiple. In Figura 5.8 c'è la lista dei socket, cioè i quattro predefiniti su KitKat e quello aperto da *wpa_supplicant* dopo aver creato un gruppo *Wi-Fi Direct*. Il motivo per cui si può avviare *wpa_cli* sull'interfaccia di gruppo è proprio perché *wpa_supplicant* stesso l'ha creata apposta per fare ciò.

Invece, per quanto riguarda la disconnessione, non esiste il comando *p2p_disconnect*, ma si utilizza *p2p_group_remove* anche sui Client [77].

⁵Join è il parametro che permette di collegarsi ad un GO di un gruppo già avviato.

5. Wpa_supplicant

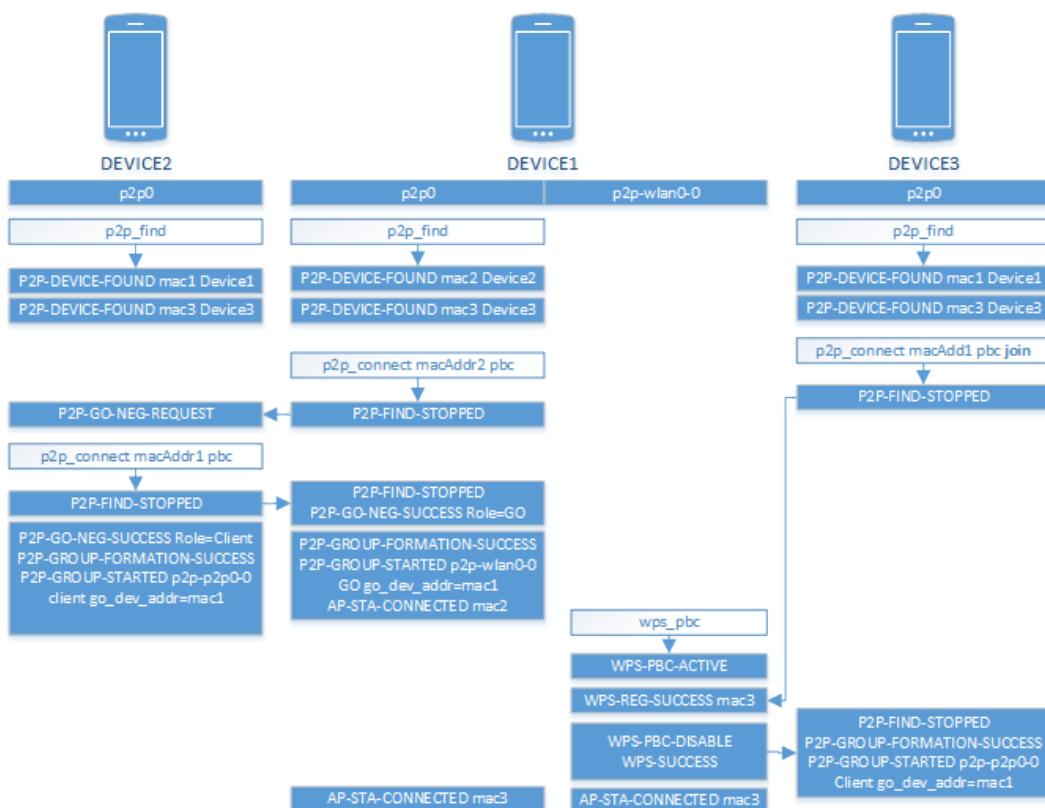


Figura 5.4: Connessione di tre dispositivi con wpa_supplicant

```

<3>P2P: Not ready for GO negotiation with fa:a9:d0:0c:e4:81
<3>P2P-GO-NEG-REQUEST fa:a9:d0:0c:e4:81 dev_passwd_id=4
<3>P2P: Building GO Negotiation Response
<3>P2P: Sending GO Negotiation Response
> p2p_connect fa:a9:d0:0c:e4:81 pbc
OK
<3>P2P-FIND-STOPPED
<3>P2P: Sending GO Negotiation Request
<3>P2P: GO Negotiation Request TX callback: success=1
<3>P2P: Received GO Negotiation Response from fa:a9:d0:0c:e4:81 (freq=2437)
<3>P2P: Sending GO Negotiation Confirm
<3>P2P: GO Negotiation Confirm TX callback: result=0
<3>P2P-GO-NEG-SUCCESS role=GO freq=2412 ht40=0 peer_dev=fa:a9:d0:0c:e4:81 peer_i
face=fa:a9:d0:0c:64:81 wps_method=PBC
<3>P2P-GROUP-FORMATION-SUCCESS
<3>P2P-GROUP-STARTED p2p-wlan0-0 GO ssid="DIRECT-ED-Galaxy_02" freq=2412 passphr
ase="mt3VSzHZ" go_dev_addr=42:0e:85:4a:3e:0f
<3>AP-STA-CONNECTED fa:a9:d0:0c:64:81 p2p_dev_addr=fa:a9:d0:0c:e4:81
<3>CTRL-EVENT-SCAN-RESULTS
<3>P2P-PROV-DISC-PBC-REQ fa:a9:d0:0c:e5:30 p2p_dev_addr=fa:a9:d0:0c:e5:30 pri_de
v_type=10-0050F204-5 name='Nexus5-pol14' config_methods=0x188 dev_capab=0x25 gro
up_capab=0x0 group=p2p-wlan0-0
<3>AP-STA-CONNECTED fa:a9:d0:0c:65:30 p2p_dev_addr=fa:a9:d0:0c:e5:30
> 

```

Figura 5.5: Wpa_supplicant GO con due client su interfaccia p2p0

```

> p2p_fnd
Unknown command 'p2p_fnd'
> p2p_find
OK
<3>P2P-DEVICE-FOUND 42:0e:85:4a:3e:0f p2p_dev_addr=42:0e:85:4a:3e:0f pri_dev_typ
e=10-0050F204-5 name='Galaxy_02' config_methods=0x188 dev_capab=0x25 group_capab
=0x0
<3>P2P-DEVICE-FOUND fa:a9:d0:0c:e5:30 p2p_dev_addr=fa:a9:d0:0c:e5:30 pri_dev_typ
e=10-0050F204-5 name='Nexus5-pol14' config_methods=0x188 dev_capab=0x25 group_ca
pab=0x0
> p2p_connect 42:0e:85:4a:3e:0f pbc
OK
<3>P2P-FIND-STOPPED
<3>P2P-GO-NEG-SUCCESS role=client freq=2412 ht40=0 peer_dev=42:0e:85:4a:3e:0f pe
er_iface=42:0e:85:4a:be:0f wps_method=PBC
<3>CTRL-EVENT-SCAN-RESULTS
<3>CTRL-EVENT-STATE-CHANGE id=-1 state=2 BSSID=00:00:00:00:00:00 SSID=
<3>P2P-GROUP-FORMATION-SUCCESS
<3>P2P-GROUP-STARTED p2p-p2p0-0 client ssid="DIRECT-ED-Galaxy_02" freq=2412 psk=
245101f2bb14a149acf4b65c698b7a4f71806e2cde7e03616df16ec92ea0b08f go_dev_addr=42:
0e:85:4a:3e:0f
<3>P2P-DEVICE-LOST p2p_dev_addr=fa:a9:d0:0c:e5:30
> 

```

Figura 5.6: Wpa_supplicant primo Client connesso al GO

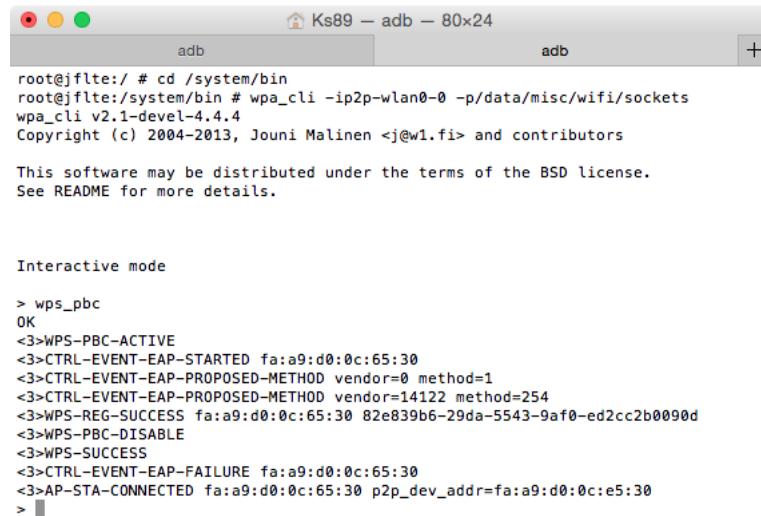
`p2p_group_remove <group interface>`

Se eseguito sul Client, si disconnette dal GO, ma tutti gli altri non subiscono variazioni, se invece è il GO ad eseguirlo, tutti i Client ricevono il comando e di conseguenza la rete cade. Lo stesso vale se il GO si spegne, o esce dall'area di ricezione.

5.6 Modificare il kernel per creare le VIF

A questo punto ho ottenuto gli elementi per descrivere l'approccio alle soluzioni dell'introduzione, infatti ho proposto di risolvere gli obiettivi usando interfacce di rete multiple. Poiché, il chip wireless sui dispositivi Android è uno solo, si tratta di interfacce di rete virtuali. Quindi, teoricamente, per raggiungere l'obiettivo, sarebbe

5. Wpa_suplicant



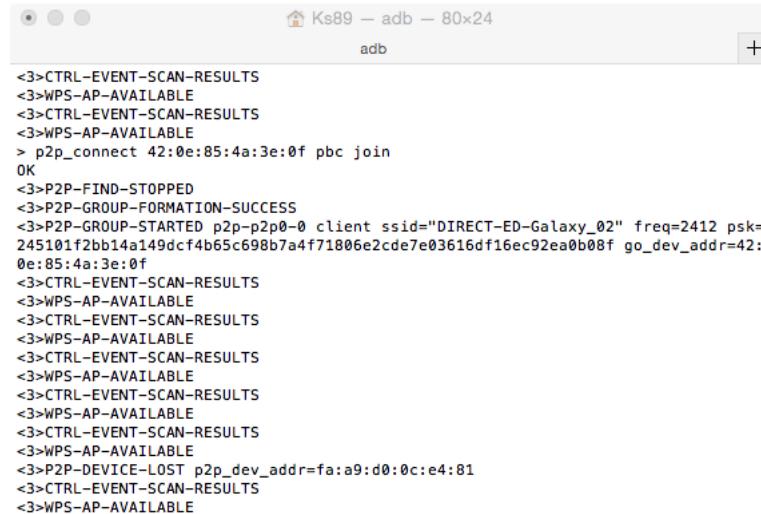
```
root@jflte:/ # cd /system/bin
root@jflte:/system/bin # wpa_cli -ip2p-wlan0-0 -p/data/misc/wifi/sockets
wpa_cli v2.1-devel-4.4.4
Copyright (c) 2004-2013, Jouni Malinen <j@w1.fi> and contributors

This software may be distributed under the terms of the BSD license.
See README for more details.

Interactive mode

> wps_pbc
OK
<3>WPS-PBC-ACTIVE
<3>CTRL-EVENT-EAP-STARTED fa:a9:d0:0c:65:30
<3>CTRL-EVENT-EAP-PROPOSED-METHOD vendor=0 method=1
<3>CTRL-EVENT-EAP-PROPOSED-METHOD vendor=14122 method=254
<3>WPS-REG-SUCCESS fa:a9:d0:0c:65:30 82e839b6-29da-5543-9af0-ed2cc2b0090d
<3>WPS-PBC-DISABLE
<3>WPS-SUCCESS
<3>CTRL-EVENT-EAP-FAILURE fa:a9:d0:0c:65:30
<3>AP-STA-CONNECTED fa:a9:d0:0c:65:30 p2p_dev_addr=fa:a9:d0:0c:e5:30
> █
```

Figura 5.7: Wpa_suplicant GO con il secondo client sull’interfaccia di gruppo p2p-p2p0-0



```
<3>CTRL-EVENT-SCAN-RESULTS
<3>WPS-AP-AVAILABLE
<3>CTRL-EVENT-SCAN-RESULTS
<3>WPS-AP-AVAILABLE
> p2p_connect 42:0e:85:4a:3e:0f pbc join
OK
<3>P2P-FIND-STOPPED
<3>P2P-GROUP-FORMATION-SUCCESS
<3>P2P-GROUP-STARTED p2p-p2p0-0 client ssid="DIRECT-ED-Galaxy_02" freq=2412 psk=
245101f2bb14a149dcf4b65c698b7a4f71806e2cde7e03616df16ec92ea0b08f go_dev_addr=42:
0e:85:4a:3e:0f
<3>CTRL-EVENT-SCAN-RESULTS
<3>WPS-AP-AVAILABLE
<3>CTRL-EVENT-SCAN-RESULTS
<3>WPS-AP-AVAILABLE
<3>CTRL-EVENT-SCAN-RESULTS
<3>WPS-AP-AVAILABLE
<3>CTRL-EVENT-SCAN-RESULTS
<3>WPS-AP-AVAILABLE
<3>CTRL-EVENT-SCAN-RESULTS
<3>WPS-AP-AVAILABLE
<3>P2P-DEVICE-LOST p2p_dev_addr=fa:a9:d0:0c:e4:81
<3>CTRL-EVENT-SCAN-RESULTS
<3>WPS-AP-AVAILABLE
```

Figura 5.8: Wpa_suplicant secondo Client connesso al GO

Algoritmo 5.8 Interfaccia p2p creata da wpa_supplicant e socket associato

```
root@mako:/ # ls -a /data/misc/wifi/sockets
p2p-p2p0-0
p2p0
wlan0
wpa_ctrl_2250-1
wpa_ctrl_2250-2
wpa_ctrl_567-1
wpa_ctrl_567-2
```

Algoritmo 5.9 MacVTap, operazione non supportata

```
root@hammerhead:/ # ip link add link wlan0 name prova type macvtap
RTNETLINK answers: Operation not supported on transport endpoint
```

sufficiente creare altre interfacce di rete, eseguire *wpa_supplicant* e poi *wpa_cli* su di esse per creare nuovi gruppi. Purtroppo non è così “semplice”.

Inizialmente, ho tentato di aggiungere una VIF con il comando *ip link add link name <nome VIF> type macvtap* per creare un’interfaccia virtuale *MacVTap*⁶, il risultato è nell’Algoritmo 5.9. Cioè, ho tentato di eseguire un’operazione non sopportata.

Questo accade perché Android non permette la creazione di interfacce virtuali di rete. In realtà ad imporre questo limite è il kernel, più precisamente il modo con cui è stato configurato. Quindi, ho deciso di modificarne la configurazione, compilarlo ed installarlo sul dispositivo. Nella Sezione 4.9, ho già accennato a come configurare il kernel, ma ora spiegherò nello specifico su quali voci di *make menuconfig* ho agito, per abilitare l’uso delle VIF.

Come primo passo sono entrato nel menu *Device Drivers* e poi nel suo sotto-menu *Network device support*, come in Figura 5.9. Tutte le impostazioni utili le ho trovate lì, cioè ho abilitato con [*] (vedi Figura 5.10):

- *MAC-VLAN support*;
- *MAC-VLAN based tap driver*;
- *Virtual ethernet pair device*.

Tutte queste impostazioni attivano a loro volta molte voci all’interno del file *.config*, ma non è necessario specificare quali, poiché *make menuconfig* fa questo lavoro tramite interfaccia grafica in modo semplificato.

Una volta compilato il kernel con *make*, rimpacchettato e installato sul dispositivo ho provveduto anche ad installare *Busybox* (dello sviluppatore con nickname “Stericson”) [87] tramite il Google PlayStore per abilitare più comandi Linux in Android, che purtroppo non sono inclusi o troppo limitati. Dopodiché, ho creato l’interfaccia di rete virtuale seguendo la procedura nell’Algoritmo 5.10, il cui risultato è visibile in 5.11. Infine, ho assegnato un IP e una netmask all’interfaccia e l’ho attivata con i comandi nell’Algoritmo 5.12.

Nonostante ciò, la lista dei socket attivi risulta quella nell’Algoritmo 5.13. Questo vuol dire che tale interfaccia, sebbene sia presente, non ha aperto un socket. Quindi perché non crearlo manualmente? La risposta è che non si può, deve essere il programma a farlo [24]. Comunque, il motivo specifico è che un’interfaccia *MacVTap*

⁶Per maggiori informazioni su MacVTap vedere [65].

5. Wpa_supplicant

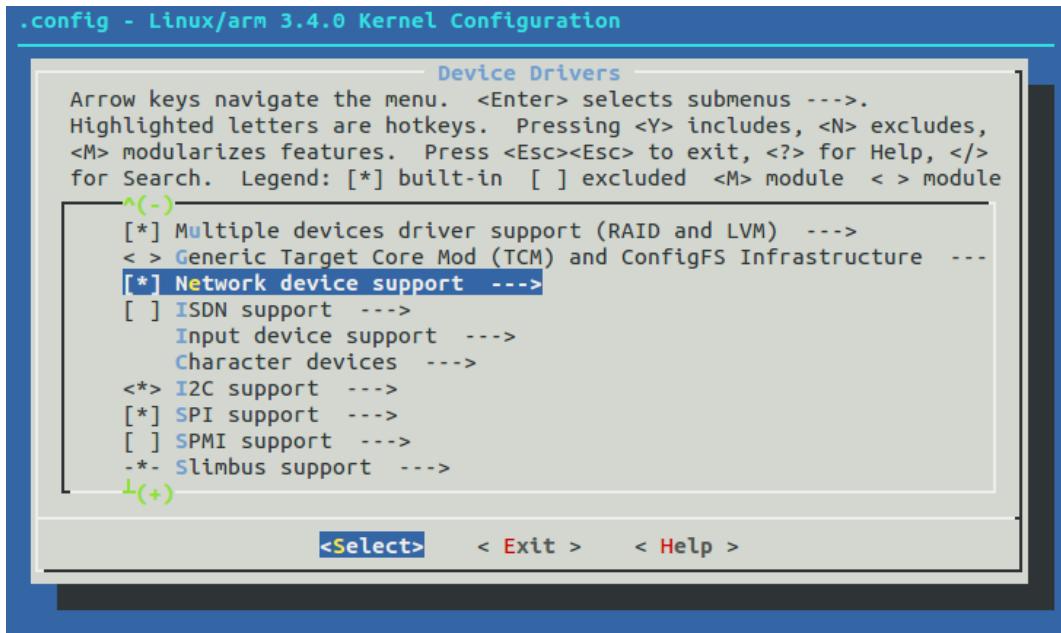


Figura 5.9: Kernel make menuconfig “Device Drivers”

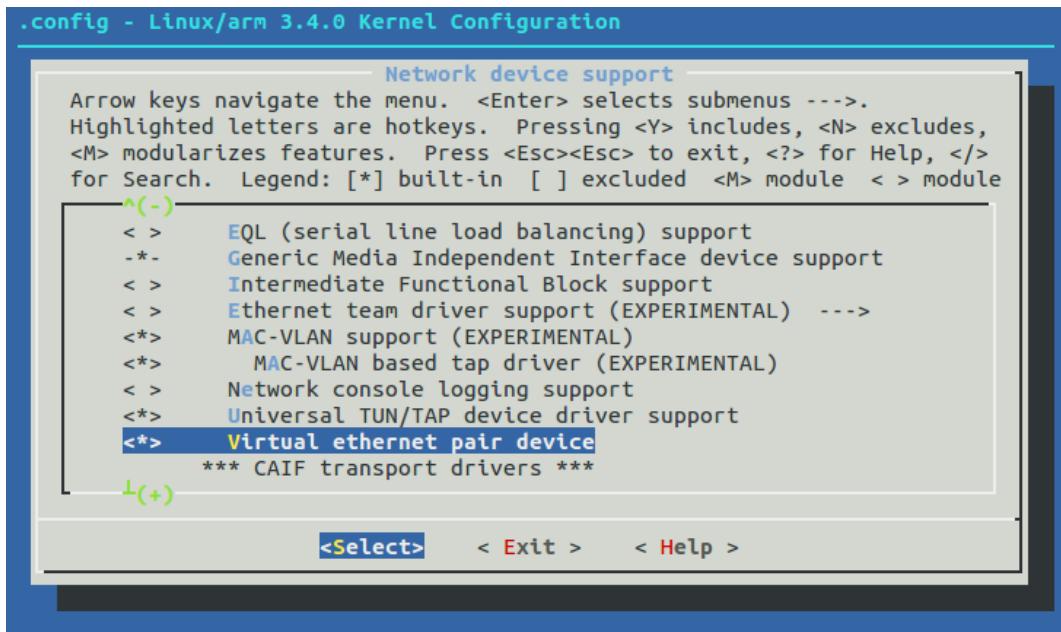


Figura 5.10: Kernel make menuconfig abilitazione VIF

Algoritmo 5.10 Creazione interfaccia di rete di tipo MacVTap

```
adb shell
shell@mako:/ $ su
root@mako:/ # ip link add link wlan0 name prova type macvtap
```

Algoritmo 5.11 Ifconfig mostra la nuova VIF

```
root@mako:/ # busybox ifconfig prova
prova      Link encap:Ethernet  HWaddr 46:9E:A6:3F:69:D9
           BROADCAST MULTICAST  MTU:1500  Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:500
           RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Algoritmo 5.12 Ifconfig mostra la nuova VIF con IP assegnato ed UP

```
root@mako:/ # busybox ifconfig prova 10.169.129.42 netmask
                  255.255.248.0 up
prova      Link encap:Ethernet  HWaddr 46:9E:A6:3F:69:D9
           inet addr:10.169.129.42  Bcast:10.169.135.255
           Mask:255.255.248.0
           UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:500
           RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Algoritmo 5.13 Lista dei socket Wi-Fi

```
root@mako:/ # ls /data/misc/wifi/sockets
p2p0
wlan0
wpa_ctrl_567-1
wpa_ctrl_567-2
```

5. Wpa_supplicant

Algoritmo 5.14 Errore di connessione di wpa_supplicant con interfaccia MacVTap

```
root@mako:/ # wpa_supplicant -B -iprova -Dnl80211  
-c/data/misc/wifi/p2p_supplicant.conf  
255| root@mako:/ #
```

Algoritmo 5.15 Tentativo di creazione di gruppi con p2p_group_add

```
> p2p_group_add  
OK  
<3>P2P-GROUP-STARTED p2p-p2p0-0 GO ssid="DIRECT-uQ"  
freq=2412 passphrase="PwrBkw3a" go_dev_addr=10:68:3f:43:92:ab  
<3>CTRL-EVENT-SCAN-RESULTS  
> p2p_group_add  
FAIL  
<5>P2P: Failed to add group interface
```

non è basata sui driver *nl80211*, necessari perché *wpa_supplicant* possa svolgere le operazioni richieste (approfondirò il concetto di *nl80211* nel Capitolo 6). Infatti, tentando di eseguire *wpa_supplicant* su questa interfaccia, il risultato è stato quello nell'Algoritmo 5.14, cioè l'errore con codice 255.

Quindi, ho verificato se è possibile creare due VIF, eseguendo due volte il comando *p2p_group_add*. La sintassi è la seguente:

```
p2p_group_add [persistent|persistent=<network id>]  
[freq=<freq in MHz>] [ht40] [vht]
```

Per svolgere questo esperimento non ho dovuto specificare nessun parametro, poiché sono tutti opzionali (essendo indicati con “[”]). Il risultato di questo comando è l'aggiunta di un gruppo *autonomous*. Il problema è che eseguendolo due volte, con l'obiettivo di creare due gruppi, cioè due interfacce differenti (*p2p-p2p0-0* e *p2p-p2p0-1*) e quindi potersi avvicinare alla soluzione del problema descritto nell'introduzione, il risultato che ho ottenuto è quello nell'Algoritmo 5.15. Cioè, non è possibile che un dispositivo Android possa connettersi a due altri contemporaneamente, in questo caso essere due volte GO. Quindi, ho analizzato il problema in molti modi, per esempio cercando di capire se è il tipo di gruppo a creare questo ostacolo, o se l'utilizzo di altri comandi per la connessione permette di aggirare il problema, ma tutto ciò è stato inutile, compreso il caso di un dispositivo GO su una interfaccia e Client su un'altra. Il problema è legato allo Smartphone/Tablet stesso, o più precisamente ai suoi driver, e non a *wpa_supplicant* [64]. Inoltre, mi sono chiesto se fosse possibile usare *wlan0*, invece che in modalità Wi-Fi, in modo P2P. Purtroppo, anche questo non è possibile, perché sono richieste interfacce di gruppo per svolgere le operazioni Wi-Fi Direct [55].

5.7 Verificare le Valid Interface Combinations con iw di Linux

Una possibile spiegazione è che siano i driver a non supportare l'aggiunta di più VIF e visionando la documentazione del kernel Linux [115], ho notato il parametro *driver_param=use_p2p_group_interface=1*, che ho aggiunto nei file di configurazione. L'obiettivo è far sì che, anche se i driver non supportano tale funzionalità, *wpa_supplicant* riesca comunque a creare le interfacce di rete. Nonostante ciò non è stato comunque possibile risolvere il problema. Così, per cercare di capire quale sia il reale supporto dei driver dei dispositivi Android, ho utilizzato il programma *iw* di Linux. Questo software non è strettamente necessario per risolvere i problemi posti nell'introduzione, ma in questo caso mi è tornato utile come strumento di analisi.

Iw è il programma a linea di comando, creato da Johannes Berg, per la configurazione di dispositivi wireless con i nuovi driver *nl80211*. È ancora in fase di sviluppo e l'unica pagina di documentazione del progetto è quella che si ottiene con *iw help* nel terminale, riassunta e migliorata nella pagina ufficiale del kernel Linux [10]. Parlerò in modo più approfondito dei driver wireless del kernel Linux nel Capitolo 6, per ora mi voglio concentrare in modo specifico su questo programma. A differenza di *wpa_supplicant* e *wpa_cli*, *iw* non è in AOSP e quindi non è possibile compilarlo facilmente. Prima di mostrare la procedura è necessario che faccia alcune precisazioni riguardo a *PIE* in Lollipop. Infatti, tentando di eseguire *iw* su Lollipop ho ottenuto l'errore: *error: only position independent executables (PIE) are supported*.

Position-independent executable (PIE) è un file eseguibile fatto interamente di *Position-Independent Code* (PIC), cioè codice che può essere messo in qualunque punto della memoria primaria ed essere eseguito senza essere vincolato ad indirizzi di memoria fissati. Come detto più volte, questi concetti legati alla sicurezza informatica non fanno parte di questo lavoro di tesi, quindi rimando a [103] per maggiori informazioni.

Il motivo per cui ho ottenuto quell'errore è che ho compilato *iw* per KitKat e ho tentato di usarlo su Lollipop, perché su quest'ultimo non si può compilare a causa di problemi di dipendenze con *libnl* non risolvibili (parlerò di *libnl* nel Capitolo 6). Dato che *iw* mi serve solo per capire i problemi di Android nel gestire più VIF, ho deciso di applicare una patch a Lollipop che disattiva il controllo di PIE e poi rimuoverla una volta terminati i test. La patch installabile tramite file “.zip” è scaricabile dalla pagina ufficiale da XDA Developers forum [19]. Fatto ciò, ho compilato *iw* (a patto di aver scaricato anche il sorgente di KitKat, infatti ho chiamato la cartella *aosp-kk*) seguendo la procedura in Algoritmo 5.16 e l'ho copiato dalla cartella *out/target/product/<nomedispositivo>/system/bin/* alla */sdcard* del dispositivo. Quindi, ho installato *iw* sul dispositivo (KitKat o dopo la *PIE Patch* anche su Lollipop) con i comandi in Algoritmo 5.17.

Algoritmo 5.16 Download e compilazione di iw per Android

```
1. $ mkdir ~/iw_master
2. $ cd ~/iw_master
3. $ git clone http://git.sipsolutions.net/iw.git
4. $ cd ~/androidsource/aosp-kk/external
5. $ cp -r ~/iw_master/iw .
6. $ cd ~/androidsource/aosp-kk
7. $ source build/envsetup.sh
8. $ lunch
9. $ make iw -jN
```

Algoritmo 5.17 Installazione di iw per Android

```
1. $ adb shell
   * daemon not running. starting it now on port 5037 *
   * daemon started successfully *
2. $ su
3. # mount -o remount,rw <nome completo di /system>
4. # cp /sdcard/iw /system/bin
5. # cd /system/bin
6. # chmod 777 iw
```

Dopodiché, per ottenere le *Valid Interface Combinations* (VIFC) tramite *iw*, ho usato i comandi nell’Algoritmo 5.18. Nel caso di Nexus 4, 5 e Samsung Galaxy S4, ordinati in base alla data di uscita sul mercato, i risultati sono consultabili nell’Algoritmo 5.19. Su questi risultati è importante fare alcune osservazioni, innanzitutto su come leggerli. Cercando su internet non ho trovato nessuna guida che aiutasse ad interpretarli in modo completo e corretto, quindi l’unica soluzione trovata per ottenere una risposta definitiva è stato analizzare il sorgente, per la precisione il file *nl80211.h* (presente sia nel kernel di Android in *msm/include/linux/nl80211.h* sia nella *repository Git* di *iw*). Ciò che interessa è la *struct enum nl80211_if_combination attrs {(...)}* e più precisamente la documentazione sopra ad essa, in cui tramite piccoli esempi è spiegata la logica di questa notazione. Per esempio:

- $\#\{tipo0\} \leq n$ significa che il numero di interfacce di tipo0 è al massimo uguale a n;
- $\#\{tipo1, tipo2\} \leq n$ significa che il numero di interfacce del tipo1 e tipo2 sommate è al massimo uguale a n;
- $\#\{tipo0\} \leq m, \#\{tipo1, tipo2\} \leq n$ significa che il numero di interfacce del tipo1 e tipo2 sommate è al massimo uguale a n e contemporaneamente si può avere anche al massimo m interfacce di tipo0;

Algoritmo 5.18 Ottenere lista VIFC

```

1. $ adb shell
   * daemon not running. starting it now on port 5037 *
   * daemon started successfully *
2. $ su
3. # iw phy phy0 info

```

- $\#\{tipo0\} \leq m, \#\{tipo1, tipo2\} \leq n, total \leq 3, \#channels \leq 2$ significa che il numero di interfacce del tipo1 e tipo2 sommate è al massimo n e contemporaneamente si può avere anche al massimo m interfacce di tipo0, ma la somma $n+m$ deve essere al massimo 3 (cioè 3 in totale), inoltre queste 3 totali possono essere su 2 canali diversi o anche sullo stesso. Quindi le condizioni *total* e *channels* sono applicate all'intera espressione;

Se in *Supported interface modes* vi è un certo tipo di interfaccia, è sempre possibile crearne almeno una di quel tipo da sola, questo non è nemmeno considerato nella VIFC, cioè questa combinazione $\#\{<type>\} \leq 1, channels = 1, total = 1$ esiste sempre. Quindi nel caso specifico dell'Algoritmo 5.19, per esempio per il Nexus 5, VIFC dice quanto segue: “si possono creare al massimo tre interfacce, su un massimo di due canali, di cui al massimo una IBSS, al massimo due di tipo managed e al massimo due tra P2P-GO e P2P-Client (cioè posso avere per esempio due GO o due Client o un GO e un Client)”. **Quindi la configurazione composta da una managed, una P2P-GO e una P2P-Client è confermata e questo permetterebbe di risolvere il problema della scalabilità descritto nell'introduzione.** Purtroppo, come ho già dimostrato in precedenza, eseguendo due volte il comando *p2p_group_add* in *wpa_supplicant*, questo non accade. La domanda è: perché? La riposta la spiegherò nel Capitolo 6, ma prima voglio fare due brevi precisazioni.

La prima è che dopo la creazione di un gruppo P2P è possibile usare *iw* per verificare la creazione dell'interfaccia di rete di gruppo, come in Algoritmo 5.20. Inoltre, è interessante vedere come le interfacce *MacVTap* non siano nemmeno rilevate da questo software, ancora a prova delle loro capacità limitate. Ciò si nota nell'Algoritmo 5.20, in cui appaiono solo interfacce a cui è associato un socket.

La seconda ed ultima precisazione è che con *iw* è possibile creare interfacce di rete virtuali, ma appoggiandosi sempre su *nl80211*, come *wpa_supplicant*, il risultato è lo stesso. Anche tentare di rimuovere quelle interfacce per crearne altre non è possibile, probabilmente per colpa di limiti nei driver Wi-Fi.

Algoritmo 5.19 iw list su Nexus 4, Nexus 5 e Samsung Galaxy S4

```
//caso Nexus 4 (novembre 2012)
Supported interface modes:
    * IBSS
    * managed
    * AP
    * P2P-client
    * P2P-GO
valid interface combinations:
    * #{managed}<=3, #{AP}<=1, #{P2P-client,P2P-GO}<=1,
      total<=3, #channels<=2

//caso Samsung Galaxy S4 GT-I9505 (aprile 2013)
Supported interface modes:
    * IBSS
    * managed
    * AP
    * P2P-client
    * P2P-GO
valid interface combinations:
    * #{managed}<=3, #{P2P-client,P2P-GO}<=2, #{IBSS}<= 1,
      total<=3, #channels<=2

//caso Nexus 5 (ottobre 2013)
Supported interface modes:
    (le stesse del GT-I9505)
valid interface combinations:
    * #{managed}<=2, #{P2P-client,P2P-GO}<=2, #{IBSS}<=1,
      total<=3, #channels<=2
```

Algoritmo 5.20 iw dev per Samsung Galaxy S4 (GO) e Nexus 5 (Client)

```
//Samsung Galaxy S4 lista interfacce di rete
root@jflte:/ # iw dev
phy#0
    Interface p2p-wlan0-0
        ifindex 11
        type P2P-GO
    Interface wlan0
        ifindex 10
        type managed
    Interface p2p0
        ifindex 9
        type managed
Unnamed/non-netdev interface

//Nexus 5 lista interfacce di rete
root@hammerhead:/system/bin # iw dev
phy#0
    Interface p2p-p2p0-0
        ifindex 22
        type P2P-client
    Interface wlan0
        ifindex 21
        type managed
    Interface p2p0
        ifindex 20
        type managed
Unnamed/non-netdev interface
```

5. Wpa_supplicant

Capitolo 6

Driver Wi-Fi in Android

Nel Capitolo 5, dopo aver parlato di *wpa_supplicant* ed *iw*, ho accennato a possibili problemi nei driver Wi-Fi nel kernel di Android. Ora li analizzerò in modo graduale fino a svelare perché *wpa_supplicant* non riesce a creare due o più interfacce di rete virtuali, nel caso specifico di Smartphone/Tablet Android. Anticipo subito che queste osservazioni e studi si basano sui dispositivi in possesso durante questo lavoro di tesi e con chip Wi-Fi prodotti tutti dalla stessa società chiamata Broadcom [14]. Non sono a conoscenza della situazione su Smartphone/Tablet con chip Qualcomm Atheros [81].

6.1 I Driver Wi-Fi in Linux

In Android, l'utente può connettersi o disconnettersi da una rete wireless e modificarne le impostazioni, quindi c'è un meccanismo che permette una comunicazione attraverso i livelli dell'architettura del sistema operativo. Cioè le funzionalità presenti nel livello più basso, detto *Kernel space*, sono esposte ad alto livello, detto *User space* o *Userland*.¹ Inoltre, lo stesso kernel deve essere in grado di comunicare con l'hardware, per esempio con il chip Wi-Fi, perché le richieste dall'utente possano essere soddisfatte. Tutte queste caratteristiche sono implementate tramite diversi livelli software che ora descriverò.

6.1.1 Il passato: WE

Wireless Extensions (WE o WEXT) è un progetto sponsorizzato da HP e dedicato alla tecnologia Wi-Fi in Linux, creato da Jen Tourrilhes nel 1997. WE è costituito

¹Il termine “Userland” o “User space” si riferisce a tutto il codice che è eseguito al di fuori del kernel. Di solito sono programmi e librerie che il sistema operativo usa per interagire col kernel. Ogni processo nello Userspace è eseguito normalmente con una sua memoria virtuale e a meno che non sia permesso in modo esplicito, non può accedere alla memoria di altri processi. Questo concetto è alla base dei sistemi di protezione per la memoria nei moderni sistemi operativi ed è la base della separazione dei privilegi [109].

da API generiche che permettono ai driver di esporre la configurazione allo *User space*. Inoltre, è affiancato da programmi chiamati *Wireless Tools* (WT) come per esempio *iwconfig*, *iwlist* e *ifrename* [60]. *WE* si basa su *Input/Output control* (ioctl), cioè una chiamata di sistema che riceve una costante numerica, la quale identifica un comando da eseguire. Si tratta di un’alternativa all’uso, che ho già accennato nel Capitolo 4, di come il kernel può comunicare con lo *User space*, scrivendo file nella partizione */proc*. Il vantaggio principale di *ioctl* è che non richiede la lettura di file in */proc* e quindi risulta più veloce, perché ottenere dati binari è più efficiente della lettura di file di testo. *Ioctl* non serve solo per il trasferimento di dati o per le funzionalità wireless, ma anche per svolgere operazioni in altri ambiti, come per esempio l’espulsione di un disco da un Lettore CD [62].

Il codice di *WE* è principalmente in *net/wireless/wext.c* nel kernel Linux, ma basandosi su *ioctl*, lo sviluppatore ha dovuto definire nuove costanti. Per esempio, se voglio cambiare l’IP su un dispositivo posso usare *ifconfig*, il quale può usare una chiamata *ioctl* specificando il nome dell’interfaccia, l’indirizzo e la costante *SIOC-SIFADDR* creata per tale scopo, definita in */usr/include/linux/sockios.h*. Purtroppo, *WE* ha alcuni difetti:

- molte funzioni devono essere re-implementate dai driver;
- specifiche sul comportamento nel dettaglio troppo vaghe;
- la semantica si basa su attributi individuali invece di specifiche azioni.

Questo progetto è stato abbandonato, perché *cfg80211*, che mostrerò tra poco, permette di emulare *WE* mantenendo la retrocompatibilità [9, 60] aggiungendo anche nuove funzionalità.

6.1.2 Il presente: *nl80211*, *cfg80211* e *mac80211*

Libnl è un insieme di librerie che forniscono API per il protocollo *Netlink* in Linux, cioè un meccanismo IPC, principalmente tra *Kernel space* e *User space*. *Netlink* è stato realizzato come successore di *ioctl* per essere più flessibile e fornire interfacce di monitoraggio, ma principalmente per la configurazione di rete per il kernel. Per maggiori informazioni su *libnl* vedere [57].

Nl80211 e *cfg80211* non utilizzano più *ioctl*, ma i *Netlink socket* [61]. Prima di procedere e vedere nel dettaglio cosa sono *nl80211* e *cfg80211*, devo spiegare il significato di concetti importanti che utilizzerò tra poco.

- **Full MAC hardware:** dispositivo il cui comportamento è determinato solo dal produttore, ha un *firmware* più grande e complesso, richiede memoria e capacità di elaborazione elevata ed è più costoso. Quindi, richiede un hardware in grado di gestire operazioni complesse ed è il *firmware* ad implementare lo *stack di rete*.

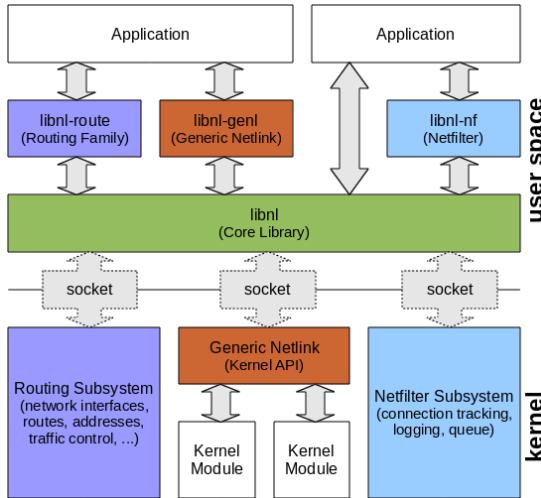


Figura 6.1: Architettura libnl

- “*Soft*” **MAC hardware**: dispositivo con un *firmware* più piccolo, più economico e il produttore ha meno controllo.

nl80211

Nl80211 è un interfaccia *Netlink* nello *User space* creata da Johannes Martin Berg nel 2006, basata su *Netlink socket* e di conseguenza dipende da *libnl*. Insieme a *cfg80211* ha sostituito *WE*, infatti *nl80211* è il lato *User space*, mentre *cfg80211* è il lato *Kernel space*, che insieme costituiscono i moderni strumenti per configurazione di dispositivi wireless in Linux.

cfg80211

Quando dallo *User space* si manda un comando a *nl80211*, prima di raggiungere l'hardware, viene gestito dal modulo del kernel *cfg80211* (il codice è in *net/wireless*, *net/wireless/nl80211.c* e *include/net/wireless.h*). Nel caso di hardware *Full MAC*, lo specifico driver si trova subito sotto a *cfg80211*, quindi quest'ultimo dovrà chiamare direttamente il primo. Invece, per hardware *Soft MAC* c'è anche *mac80211*, che è un altro modulo del kernel che implementa il livello 802.11 MAC dello *stack di rete*. Quindi, in questo caso *cfg80211* “parlerà” con *mac80211*, il quale si occuperà di comunicare con il driver inferiore. Per comprendere meglio questi concetti vedere le Figure 6.2 e soprattutto 6.3.

Cfg80211 è un livello sopra a *mac80211* che si occupa della registrazione del dispositivo (tenendo conto delle sue capacità come i canali, la banda, il bit-rate, la modalità dell’interfaccia), della gestione di stazioni AP, chiavi, *VIF* e infine della scansione. *Cfg80211* si occupa di creare interfacce di rete, cambiarne il tipo (IBSS, managed, AP e AP_VLAN, Mesh Point, monitor [114]), tenere traccia di quelle

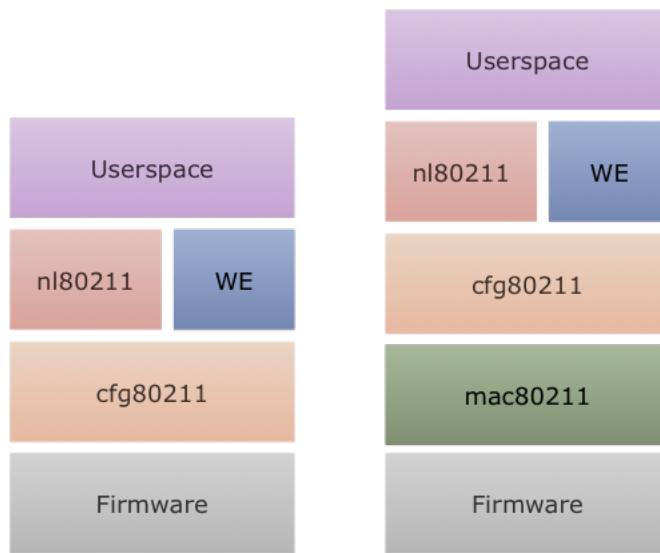


Figura 6.2: Architettura generale dei driver Wi-Fi nel kernel Linux. A sinistra per hardware Full MAC e a destra per Soft MAC.

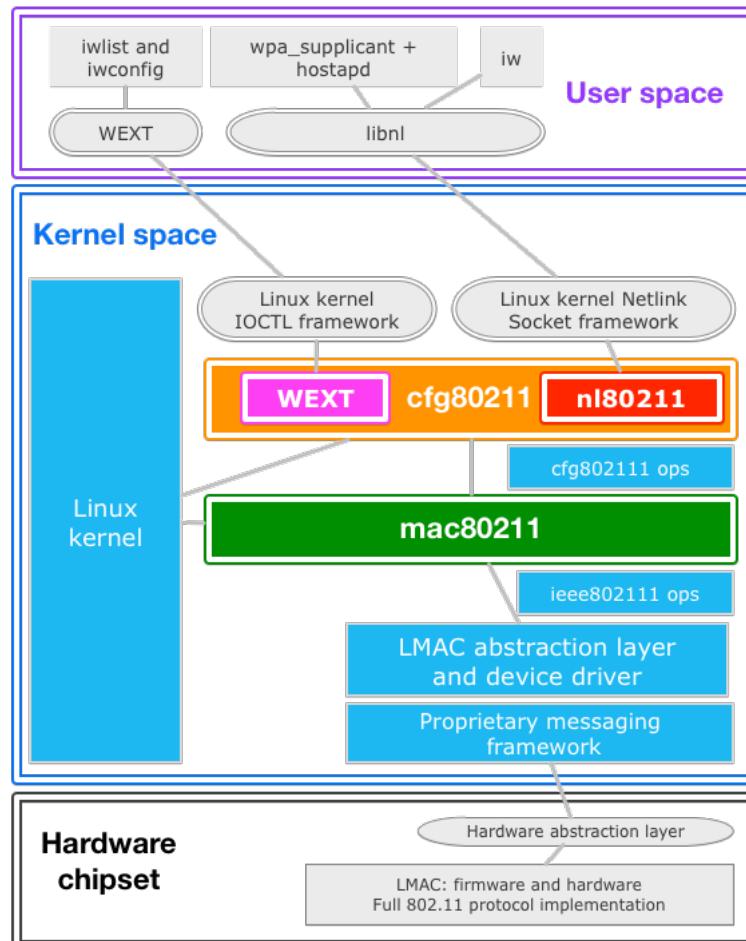


Figura 6.3: Architettura nel dettaglio dei driver Wi-Fi nel kernel Linux [22].

associate ad un dispositivo e permettere di usare più interfacce di tipo differente contemporaneamente.

Per quanto riguarda la scansione, vi sono funzioni in più rispetto a *WE*, come la gestione di SSID multipli e la possibilità di specificare il canale [9, 60]. I vantaggi principali di *cfg80211* sono che:

- utilizza i *Netlink socket* e non più *ioctl*;
- i driver devono implementare solo alcuni metodi di configurazione;
- il comportamento è definito nel dettaglio;
- la semantica si basa sulle specifiche IEEE 802.11;
- al di sopra di esso vi è un’implementazione di *WE* per fornire retrocompatibilità (come si vede in Figura 6.2).

mac80211

Mac80211 è un framework creato da Devicescape Software che gli sviluppatori di driver possono usare per dispositivi wireless *Soft MAC*, cioè quelli che permettono una gestione a grana fine (in inglese fine-grained) dell’hardware. *Mac80211* sostituisce *ieee80211softmac*, un “add-on”, per supportare *Soft MAC*, della libreria di supporto *ieee80211* usata per funzionalità di basso livello dei driver. *Mac80211* dipende da *cfg80211* per quanto riguarda la registrazione e la configurazione.

Per una visione più dettagliata dell’architettura dei driver Wi-Fi nel kernel Linux, vedere la Figura 6.3.

Architettura Wi-Fi di Nexus 5 e Samsung Galaxy S4

Ho deciso di usare come riferimento i due Smartphone più recenti a disposizione. Essi hanno lo stesso chip Wi-Fi Broadcom. L’architettura è rappresentata in Figura 6.4 con i livelli già descritti poco fa. L’unica eccezione è *brcmfmac*, cioè un driver *Open source* Broadcom per far comunicare il livello superiore standard *cfg80211* con il *firmware* proprietario, facendo una sorta di traduzione. L’intera architettura rappresentata in Figura 6.4 è costituita da codice *Open source*, ad eccezione dell’ultimo livello, cioè il *firmware*.

6.2 Visualizzare gli errori di wpa_supplicant

Dopo l’introduzione teorica è il momento di mostrare come si comporta in realtà *wpa_supplicant* e i messaggi d’errore che mostra, leggendo il *Logcat* di Android. Il metodo più rapido è usare il comando *adb logcat wpa_supplicant:V *:S* nel terminale. Questo mostrerà nella console tutti i messaggi “verbose” (V) associati a

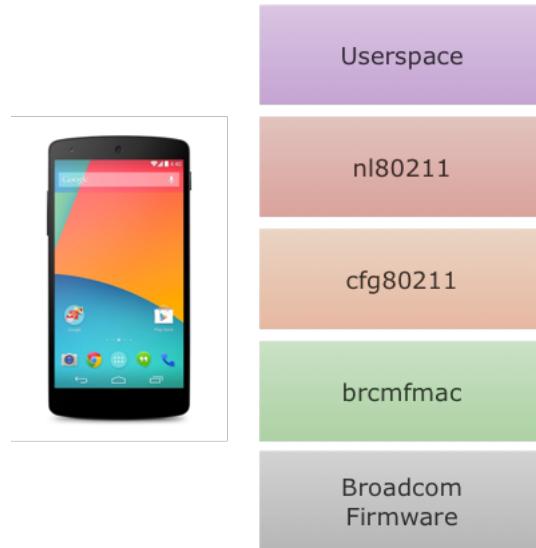


Figura 6.4: Architettura Wi-Fi Nexus 5 e Samsung Galaxy S4

wpa_supplicant, cioè di tutti i tipi: *verbose*, *debug*, *info*, *warning*, *error*. Ho deciso di fare ciò per capire:

- perché *wpa_supplicant* mostra il messaggio d'errore <5>*P2P: Failed to add group interface* e cosa lo causa;
- se il parametro passato ai driver è interpretato in modo corretto.

In questa sezione, mostrerò passo passo il log ottenuto su un Nexus 5, evidenziando le operazioni che svolge *wpa_supplicant* dall'avvio fino all'errore in questione. Ove non specificato sono messaggi di *debug* (D), altrimenti *errori* (E). Inoltre, per ottenere un log più breve e più interessante per questo lavoro di tesi, oltre a rimuovere le parti superflue, non ho avviato *wpa_supplicant* su due interfacce, ma solo su *p2p0*, cioè ho usato il comando *wpa_supplicant -B -ip2p0 -Dnl80211 -c/data/misc/wifi/p2p_supplicant.conf -dd* (“-dd” per ottenere più informazioni nel log). La versione utilizzata è la *2.1-devel-4.4.4*, cioè quella inclusa in Android KitKat 4.4.4 per Nexus 5.

1. **wpa_supplicant v2.1-devel-4.4.4**
2. **Successfully initialized wpa_supplicant**

Tra le prime operazioni svolte c'è l'inizializzazione dell'interfaccia di rete con il driver specificato, cioè i parametri *-i<nome interfaccia>* e *-D<nome driver>*. Subito dopo viene individuato il file di configurazione, indicato con *-c<percorso assoluto file di configurazione>*.

3. **Initializing interface 'p2p0' conf**

```
'/data/misc/wifi/p2p_supplicant.conf' driver 'nl80211'  
ctrl_interface 'N/A' bridge 'N/A'  
4. Configuration file /data/misc/wifi/p2p_supplicant.conf  
-> '/data/misc/wifi/p2p_supplicant.conf'
```

A questo punto, *wpa_supplicant* è pronto per leggere la configurazione facendo il parsing del file. In questa fase sono individuati problemi tra cui, per esempio, errori di battitura, parametri non esistenti ed errori nei valori assegnati. La prima osservazione che devo fare è che il parametro *driver_param='use_p2p_group_interface=1'* passato al driver è sintatticamente corretto, infatti non è apparso nessun messaggio di errore durante il parsing.

```
5. Reading configuration file  
'/data/misc/wifi/p2p_supplicant.conf'  
6. ctrl_interface='/data/misc/wifi/sockets'  
7. disable_scan_offload=1  
8. driver_param='use_p2p_group_interface=1'  
9. update_config=1  
10. device_name='Nexus5-pol4'  
11. config_methods='virtual_push_button  
physical_display keypad'  
12. p2p_ssid_postfix='-Nexus5-pol4'  
13. persistent_reconnect=1
```

A questo punto ci sono i messaggi lanciati direttamente dal driver *nl80211*, già descritto nella Sottosezione 6.1.2. La riga 14 dice che il driver *nl80211* permette l'aggiunta di interfacce virtuali di rete e la 15 che supporta anche le operazioni concorrenti su più canali di comunicazione. Dopodiché, nella riga 16, l'interfaccia *p2p0* viene creata in *phy0*, cioè sull'unico chip Wi-Fi presente nel Nexus 5 e in quella successiva è impostata come tipo STATION ed indice 20. Nelle righe 18 e 19, **nl80211 ha elaborato correttamente il parametro passato ai driver e ha confermato che saranno usate interfacce P2P di gruppo separate**. Perfetto! Ma se la configurazione e il parametro sono interpretati correttamente, dove è il problema? Per scoprirllo bisogna continuare a leggere il log.

```
14. nl80211: Use separate P2P group interface  
(driver advertised support)  
15. nl80211: Enable multi-channel concurrent  
(driver advertised support)  
16. nl80211: interface p2p0 in phy phy0  
17. nl80211: Set mode ifindex 20 iftype 2 (STATION)
```

```
18. nl80211: driver param='use_p2p_group_interface=1'  
19. nl80211: Use separate P2P group interface
```

Infatti, all’interfaccia *p2p0*, prima è assegnato il MAC address e poi è impostato l’UUID per supportare WPS. Quindi, *p2p0* viene aggiunta e *wpa_supplicant* diventa un *daemon* (come specificato dal parametro “-B”).

```
20. p2p0: Own MAC address: fa:a9:d0:0c:e5:30  
21. WPS: Set UUID for interface p2p0  
22. WPS: UUID based on MAC address - hexdump(len=16):  
      30 0c 7a b7 38 8f 5f af ae e2 49 d3 5f d4 30 01  
23. p2p0: Added interface p2p0  
24. Daemonize..
```

Ora *wpa_supplicant* è un *daemon* in background ed è in attesa di comandi, cioè quelli che ho fornito tramite *wpa_cli* eseguito sull’interfaccia *p2p0*. Infatti, il primo messaggio è *P2P_GROUP_ADD*, generato dal comando *p2p_group_add* (riga 25). Come già accennato, l’esecuzione di alcune operazioni in *wpa_cli* ferma forzatamente la *fase di Discovery* (riga 26).

```
25. p2p0: Control interface command 'P2P_GROUP_ADD'  
26. P2P: Stop any on-going P2P FIND
```

Quindi, questo comando è elaborato impostando la frequenza di funzionamento. Ricordo in modo estremamente semplificato che IEEE 802.11 a 2,4 GHz divide lo spettro in 14 sottocanali con larghezza di banda 20 MHz (o 40 nel caso della versione IEEE 802.11n). Frequenza e canale sono associati, cioè ogni canale funziona su una certa frequenza, o più precisamente in una certa banda in frequenza. Nel mio caso, non avendo specificato un canale, il programma ha scelto il primo ed ovviamente il dispositivo diventerà un GO, avendo indicato la volontà di usare un gruppo *autonomous*. Questo non è un errore, perché è corretto che il dispositivo usino canali differenti, infatti la *fase di Discovery* serve proprio per farli trovare sullo stesso canale. Dopodiché, il driver *nl80211* crea l’interfaccia di gruppo *p2p-p2p0-0* con modalità *P2P_GO* e con indice 22 (righe 30 e 31). A questo, *wpa_supplicant* assegna il MAC address all’interfaccia con il driver *nl80211* e il socket che sarà creato nella cartella predefinita (righe 32 e 33), come già mostrato nel Capitolo 4.

```
27. P2P: Set GO freq 2412 MHz (no preference known)  
28. P2P: Cannot force GO on any of the channels we are  
      already using. Use one of the free channels  
29. P2P: Create a new interface p2p-p2p0-0 for the group  
30. nl80211: Create interface iftype 9 (P2P_GO)
```

```
31. nl80211: New interface p2p-p2p0-0 created: ifindex=22
32. P2P: Created pending virtual interface p2p-p2p0-0
    addr fa:a9:d0:0c:65:30
33. Initializing interface 'p2p-p2p0-0' conf 'N/A' driver
    'nl80211' ctrl_interface '/data/misc/wifi/sockets'
    bridge 'N/A'
```

Prima di procedere, voglio specificare un fatto che tornerà utile in seguito, cioè l’assegnamento del MAC address. L’interfaccia *p2p0* usata in precedenza ha MAC address fa:a9:d0:0c:e5:30, mentre quella nuova *p2p-p2p0-0* ha fa:a9:d0:0c:65:30, cioè sono differenti. Al momento sembrerà un’osservazione inutile, ma è veramente essenziale per comprendere esattamente quale sia il motivo per cui *wpa_supplicant* non riesce a creare la seconda interfaccia di rete virtuale. Completerò la trattazione di tale argomento nella conclusione di questo capitolo.

Tornando all’analisi del log, dopo la creazione di *p2p-p2p0-0*, il driver *nl80211* ripete le stesse operazioni eseguite in precedenza (righe 14-19), ma questa volta sulla nuova interfaccia. Riporto comunque i messaggi del log nelle righe 34-39, per una questione di completezza.

```
34. nl80211: Use separate P2P group interface
    (driver advertised support)
35. nl80211: Enable multi-channel concurrent
    (driver advertised support)
36. nl80211: interface p2p-p2p0-0 in phy phy0
37. nl80211: Set mode ifindex 22 iftype 9 (P2P_G0)
38. nl80211: driver param='use_p2p_group_interface=1'
39. nl80211: Use separate P2P group interface
```

Dopodiché, una volta che l’interfaccia è aggiunta e disponibile, viene avviato il gruppo *Wi-Fi Direct*, impostando l’AP con SSID scelto in modo pseudo-casuale (rispettando le specifiche del protocollo che richiedono l’uso di “DIRECT-” come prefisso).

```
40. p2p-p2p0-0: Own MAC address: fa:a9:d0:0c:65:30
41. p2p-p2p0-0: Added interface p2p-p2p0-0
42. p2p0: P2P: Use separate group interface p2p-p2p0-0
43. p2p-p2p0-0: P2P: Starting GO
44. Setting up AP (SSID='DIRECT-D9-Nexus5-pol14')
```

A questo punto, anche il driver *nl80211* è informato dello stato di questa nuova interfaccia avviata e l’inizializzazione termina in modo corretto (righe 45-49).

```
45. nl80211: Setup AP operations for P2P group (GO)
```

```
46. nl80211: Set mode ifindex 22 iftype 9 (P2P_G0)
47. Completing interface initialization
48. Mode: IEEE 802.11g Channel: 1 Frequency: 2412 MHz
49. nl80211: Set freq 2412 (ht_enabled=1, vht_enabled=0,
   bandwidth=20 MHz, cf1=2412 MHz, cf2=0 MHz)
```

Ora, il gruppo è pronto e anche *wpa_supplicant* mostra lo stato in *wpa_cli*, tramite *P2P-GROUP-STARTED*.

```
50. P2P-GROUP-STARTED p2p-p2p0-0 GO
   ssid="DIRECT-D9-Nexus5-poli4" freq=2412
   passphrase="MCJLTwVk" go_dev_addr=fa:a9:d0:0c:e5:30
51. p2p-p2p0-0: Setup of interface done.
```

Fino a questo punto non ci sono stati problemi, anzi, solo aspetti positivi nel capire che in effetti la configurazione è corretta e che i parametri sono interpretati in modo giusto. Il problema nasce con l'aggiunta della seconda *VIF*, tramite *p2p_group_add* in *wpa_cli* (righe 52-58).

```
52. p2p0: Control interface command 'P2P_GROUP_ADD'
53. P2P: Stop any on-going P2P FIND
54. P2P: Stopping find
55. P2P: Set GO freq 2412 MHz (no preference known)
56. P2P: Force GO on a channel we are already using (2412 MHz)
57. P2P: Create a new interface p2p-p2p0-1 for the group
58. nl80211: Create interface iftype 9 (P2P_G0)
```

Infatti, inizialmente *wpa_supplicant* esegue tutte le operazioni in modo corretto, riuscendo anche a riutilizzare il canale 1 della prima interfaccia di gruppo. Però, nel momento in cui interviene *nl80211*, il quale di fatto espone semplicemente le funzionalità di *cfg80211*, compare l'errore nelle righe 59-61.

```
59. Failed to create interface p2p-p2p0-1: -12 (Out of memory)
60. Failed to create new group interface
61. Failed to add group interface
```

Questo è molto importante, perché finalmente sono a conoscenza dello specifico codice dell'errore, cioè *-12 (Out of memory)*. Nell'immediato non è d'aiuto, ma è un punto di inizio per cercare di capire perché i driver Wi-Fi non si comportano nel modo corretto.

6.3 Studio dei driver Broadcom

Dopo che ho scoperto il codice d'errore, cioè `-12 (Out of memory)`, ho cercato di capire chi lo generasse e l'unica possibilità è *wpa_supplicant* stesso. Ma chi è che causa un errore in questo programma che in cascata genera `-12 (Out of memory)`? L'unica spiegazione plausibile è che sono i driver. Così, ho preso come riferimento sempre il Nexus 5, Smartphone dotato di *Broadcom BCM4339 5G Wi-Fi combo chip* [13,54]. Questo chip usa i driver *brcm80211/brcmfmac* nel kernel di Android, ma quando ho affrontato questo problema la prima volta, l'ho fatto senza poter conoscere questo dettaglio, poiché la documentazione ufficiale è stata rilasciata da Broadcom solamente durante la scrittura di questo documento ed oltretutto in seguito ad una mia richiesta [82].

Quindi, senza conoscenze particolari su questo chip Wi-Fi, ho pensato che è molto probabile che i driver abbiano una sorta di messaggi di debug all'interno. Inoltre, essendo nel kernel di Android è probabile che scrivano tali informazioni direttamente nel log del kernel. Questa si è rivelata un'ottima idea poiché, come mostrerò tra poco, le informazioni nel log del kernel mi hanno permesso di fornire la spiegazione definitiva. È importante notare che questo log e *Logcat* sono completamente diversi e si trovano in spazi differenti, *User space* e *Kernel space* rispettivamente. Infatti, dal secondo non si può leggere i messaggi del kernel. Addirittura, *Logact* è un processo avviato con il sistema operativo, quindi tutte le informazioni dall'avvio del kernel non sono visibili.

Il metodo più comodo che ho trovato per leggere il log del kernel è usare il comando Linux *dmesg*. Il risultato, opportunamente ridotto, è riportato in questa sezione, incominciando dai primi messaggi di log all'avvio del kernel (vedi Algoritmo 6.1). Ma ciò che è più importante, sono i messaggi mostrati in concomitanza dell'esecuzione di *wpa_supplicant*, in particolare durante il lancio dei due comandi *p2p_group_add*. Come si può notare nell'Algoritmo 6.2 compaiono diversi errori, non più in *wpa_supplicant* o *nl80211*, ma nel livello inferiore, cioè *cfg80211*. Nella riga 1, l'interfaccia di rete è creata con successo, cioè la prima con nome *p2p-p2p0-0*. Dopodiché, *cfg80211* mostra comunque degli errori nonostante la *VIF* sia utilizzabile e funzionante. Non mi sono preoccupato di questo, poiché sono in grado di collegare dispositivi ed utilizzare tale interfaccia senza difficoltà.

Il problema è apparso quando ho lanciato il comando *p2p_group_add* per la seconda volta, infatti, il risultato è stato quello delle righe 8 e 9 dell'Algoritmo 6.2. Questa volta, oltre al codice d'errore, indica anche il nome “*wl_cfg80211_add_virtual_iface*”, che fa pensare ad una funzione all'interno dei driver. Per ottenere la conferma ho usato il comando *grep -r “wl_cfg80211_add_virtual_iface”* all'interno del sorgente del kernel e il risultato è in Figura 6.5. Fortunatamente questa stringa appare solo nel file */drivers/net/wireless/bcmdhd/wl_cfg80211.c*.

Algoritmo 6.1 Log dell'avvio kernel Linux con il comando dmesg

```
<6>[ 0.000000] Booting Linux on physical CPU 0
<6>[ 0.000000] Initializing cgroup subsys cpu
<5>[ 0.000000] Linux version 3.4.0-gd59db4e
    (android-build@vpbs1.mtv.corp.google.com)
    (gcc version 4.7 (GCC) ) #1 SMP PREEMPT
    Mon Mar 17 15:16:36 PDT 2014
<4>[ 0.000000] CPU: ARMv7 Processor [512f06f0]
    revision 0 (ARMv7), cr=10c5387d
<4>[ 0.000000] CPU: PIPT / VIPT nonaliasing data
    cache, PIPT instruction cache
<6>[ 0.000000] Machine: Qualcomm MSM 8974 HAMMERHEAD
    (Flattened Device Tree), model: LGE MSM 8974 HAMMERHEAD
```

Algoritmo 6.2 Log dell'errore durante la creazione di due VIF con il comando dmesg

1. <4>[8724.652887] Broadcom Dongle Host Driver: register interface [p2p-p2p0-0] MAC: fa:a9:d0:0c:65:30
 2. <6>[8724.654238] CFG80211-ERROR) wl_cfg80211_add_virtual_iface virtual interface(p2p-p2p0-0) is created net attach done
 3. <6>[8724.660003] CFG80211-ERROR) dhd_cfg80211_set_p2p_info: Set : op_mode=0x0045
 4. <6>[8724.750920] CFG80211-ERROR) wl_cfg80211_set_channel: netdev_ifidx(22), chan_type(0) target channel(1)
 5. <6>[8724.752669] CFG80211-ERROR) wl_cfg80211_set_channel: netdev_ifidx(22), chan_type(1) target channel(1)
 6. <6>[8724.753675] CFG80211-ERROR) wl_cfg80211_del_station: Disconnect STA : ff:ff:ff:ff:ff:ff scb_val.val 3
 7. <7>[8726.168810] p2p-p2p0-0: no IPv6 routers present
 8. <4>[8726.168944] 'wl p2p_ifadd' error -2
 9. <6>[8726.169368] CFG80211-ERROR)
 wl_cfg80211_add_virtual_iface : virtual iface add failed (-2)
-

```
ks89@ks89-pc:~$ cd androidKernel/msm
ks89@ks89-pc:~/androidKernel/msm$ grep -r "wl_cfg80211_add_virtual_iface"
drivers/net/wireless/bcmdhd/wl_cfg80211.c:wl_cfg80211_add_virtual_iface(struct wiphy *wiphy, char *name,
drivers/net/wireless/bcmdhd/wl_cfg80211.c:           .add_virtual_intf = wl_cfg80211_
add_virtual_iface,
```

Figura 6.5: Risultato del comando grep -r “wl_cfg80211_add_virtual_iface”

```

static struct net_device *
wl_cfg80211_add_virtual_iface(struct wiphy *wiphy, char *name,
                           enum nl80211_iftype type, u32 *flags,
                           struct vif_params *params)
{
    s32 err;
    s32 timeout = -1;
    s32 wlif_type = -1;
    s32 mode = 0;
#if defined(WL_ENABLE_P2P_IF)
    s32 dhd_mode = 0;
#endif /* (WL_ENABLE_P2P_IF) */
    chanspec_t chspec;
    struct wl_priv *wl = wiphy_priv(wiphy);
    struct net_device *_ndev;
    struct ether_addr primary_mac;
    int (*net_attach)(void *dhdp, int ifidx);
    bool rollback_lock = false;

    /* Use primary I/F for sending cmds down to firmware */
    _ndev = wl_to_primary_ndev(wl);

```

Figura 6.6: Funzione del file wl_cfg80211.c

```

chspec = wl_cfg80211_get_shared_freq(wiphy);

/* For P2P mode, use P2P-specific driver features to create the
 * bss: "wl p2p_ifadd"
 */
wl_set_p2p_status(wl, IF_ADD);
err = wl_cfgp2p_ifadd(wl, &wl->p2p->int_addr, htod32(wlif_type), chspec);

if (unlikely(err)) {
    WL_ERR((" virtual iface add failed (%d) \n", err));
    return ERR_PTR(-ENOMEM);
}

```

Figura 6.7: Errore “Virtual iface add failed”

Analizzando tale file, riportato parzialmente in Figura 6.6, c’è la definizione della *struct net_device*, che in caso di errore mostra il messaggio “Virtual iface add failed <codice errore>” in Figura 6.7, cioè quello ottenuto nell’Algoritmo 6.2, riga 9. Questo messaggio è lanciato perché la variabile *err* ha una valore non ammesso, verificato tramite la *macro* Linux *unlikely* (usata per motivi d’ottimizzazione e sfruttare la tecnica chiamata *branch prediction*). Comunque, tale argomento è al di fuori da questo lavoro di tesi, quindi mi voglio concentrare sulla funzione che genera l’errore, cioè *wl_cfgp2p_ifadd*, la cui definizione, che ho trovato grazie a grep (vedi Figura 6.8), è visibile in Figura 6.9. In questa funzione, vi è un’altra chiamata ad un’altra funzione, visibile in Figura 6.10. Tramite *grep* (Figura 6.11) ho scoperto che è definita in 6.12 *wldev_common.c*. Dopodiché, in essa ho notato che vi è la chiamata a *wldev_ioctl* in Figura 6.13 e così via addentrandsi sempre più nel meccanismo di comunicazione basato su *ioctl*, senza però giungere ad un motivo per cui *err* assuma quel valore.

Così ho pensato ad un’alternativa, cioè ritornare in *wl_cfgp2p.c* e prima di assegnare il valore ad *err*, chiamando *wldev_iovar_setbuff*, ho aggiunto la *printk*

6. Driver Wi-Fi in Android

```
ks89@ks89-pc:~/androidKernel/msm$ grep -r "wl_cfgp2p_ifadd"
drivers/net/wireless/bcmddhd/wl_cfgp2p.h:wl_cfgp2p_ifadd(struct wl_priv *wl, struc
ct ether_addr *mac, u8 if_type,
drivers/net/wireless/bcmddhd/wl_cfg80211.c:                                err = wl_cfgp2p_ifadd(wl
, &wl->p2p->int_addr, htod32(wlif_type), chspec);
drivers/net/wireless/bcmddhd/wl_cfgp2p.c:wl_cfgp2p_ifadd(struct wl_priv *wl, struc
ct ether_addr *mac, u8 if_type,
```

Figura 6.8: Ricerca con grep wl_cfgp2p_ifadd

```
s32
wl_cfgp2p_ifadd(struct wl_priv *wl, struct ether_addr *mac, u8 if_type,
                  chanspec_t chspec)
{
    wl_p2p_if_t ifreq;
    s32 err;
    struct net_device *ndev = wl_to_prmry_ndev(wl);

    ifreq.type = if_type;
    ifreq.chspec = chspec;
    memcpy(ifreq.addr.octet, mac->octet, sizeof(ifreq.addr.octet));
```

Figura 6.9: Definizione funzione wl_cfgp2p_ifadd in wl_cfgp2p.c

```
err = wldev_iovar_setbuf(ndev, "p2p_ifadd", &ifreq, sizeof(ifreq),
                         wl->iocctl_buf, WLC_IOCTL_MAXLEN, &wl->iocctl_buf_sync);
return err;
```

Figura 6.10: Chiamata a wldev_iovar_setbuff

```
drivers/net/wireless/bcmddhd/wl_cfg80211.c:                                     err = wldev_iovar
setbuf_bsscfg(dev, "ssid", &wl->p2p->ssid,
drivers/net/wireless/bcmddhd/wldev_common.c:s32 wldev_iovar_setbuf(
drivers/net/wireless/bcmddhd/wldev_common.c:      return wldev_iovar_setbuf(dev, io
var, &val, sizeof(val), iovar_buf,
drivers/net/wireless/bcmddhd/wldev_common.c:s32 wldev_iovar_setbuf_bsscfg(
drivers/net/wireless/bcmddhd/wldev_common.c:      return wldev_iovar_setbuf_bsscfg(
dev, iovar, &val, sizeof(val), iovar_buf,
```

Figura 6.11: Ricerca con grep di wldev_iovar_setbuff

```
s32 wldev_iovar_setbuf(
    struct net_device *dev, s8 *iovar_name,
    void *param, s32 paramlen, void *buf, s32 buflen, struct mutex* buf_sync)
{
    s32 ret = 0;
    s32 iovar_len;
    if (buf_sync) {
        mutex_lock(buf_sync);
    }
    iovar_len = wldev_mkiovar(iovar_name, param, paramlen, buf, buflen);
    ret = wldev_ioctl(dev, WLC_SET_VAR, buf, iovar_len, TRUE);
    if (buf_sync)
        mutex_unlock(buf_sync);
    return ret;
```

Figura 6.12: Definizione wldev_iovar_setbuff in wldev_common.c e chiamata a wldev_ioctl

```

extern int dhd_ioctl_entry_local(struct net_device *net, wl_ioctl_t *ioc, int cmd);

s32 wldev_ioctl(
    struct net_device *dev, u32 cmd, void *arg, u32 len, u32 set)
{
    s32 ret = 0;
    struct wl_ioctl ioc;

    memset(&ioc, 0, sizeof(ioc));
    ioc.cmd = cmd;
    ioc.buf = arg;
    ioc.len = len;
    ioc.set = set;

    ret = dhd_ioctl_entry_local(dev, &ioc, cmd);

    return ret;
}

```

Figura 6.13: Definizione di wldev_ioctl e chiamata dhd_ioctl_entry_local

```

printk("My message-MAC=
%02x:%02x:%02x:%02x:%02x:%02x \n", ifreq.addr.octet[0] ,
       ifreq.addr.octet[1], ifreq.addr.octet[2],
       ifreq.addr.octet[3], ifreq.addr.octet[4],
       ifreq.addr.octet[5]);

```

con il MAC address dell’interfaccia che si vuole inizializzare. Ovviamente, ho dovuto studiare le *struct* di questo file per individuare il modo per estrarre il MAC address. Per semplificare, non è altro che una printf, il cui output è visibile tramite il comando *dmesg*, cioè l’output è salvato nel log del kernel. In altre parole, ho fatto sì che nel log di *dmesg* visto all’inizio di questa sezione, appaia un messaggio aggiuntivo con il MAC address assegnato a quella *VIF*. Il risultato è visibile nell’Algoritmo 6.3 in cui si può notare che **l’interfaccia viene creata sempre con lo stesso MAC address** nel *net_device*, quindi **più gruppi avranno tutti lo stesso indirizzo fisico**. **Questa situazione non è permessa, essendo il MAC address un identificativo univoco**. Ma, la domanda è: perché avviene? Si tratta di un bug dei driver, di funzioni non implementate o altro?

Giunti a questo punto, l’unico modo per ottenere delle risposte è stato contattare uno dipendente di Broadcom, Arend van Spriel, uno degli sviluppatori che si è occupato proprio del chip Wi-Fi del Nexus 5, che è anche lo stesso del Samsung Galaxy S4. Egli ha risposto che questo hardware è supportato dai driver *brcmfmac*, cioè basati su *cfg80211* o più precisamente *Full MAC* ed ha fornito la documentazione al riguardo.

Come già accennato all’inizio del capitolo, nell’hardware *Full MAC* lo *stack 802.11* si trova nel *firmware* dei chip Wi-Fi [6]. La spiegazione è chiara, infatti **l’hardware Full MAC non utilizza mac80211, ma in cfg80211 vi è il sistema di comunicazione con il firmware proprietario** (anche se in questo caso

Algoritmo 6.3 Log dmesg con una printk aggiunta per mostrare il MAC address

```
<4>[303.999692] My message-MAC= fa:a9:d0:0c:65:30
<4>[304.016646] Broadcom Dongle Host Driver:
    register interface [p2p-p2p0-0] MAC: fa:a9:d0:0c:65:30
<6>[304.018220] CFG80211-ERROR) wl_cfg80211_
    add_virtual_iface : virtual interface(p2p-p2p0-0)
    is created net attach done
<6>[304.031038] CFG80211-ERROR)
    wl_cfg80211_determine_vsdb_mode : Same Channel
    concurrency is enabled
<4>[305.065186] My message-MAC= fa:a9:d0:0c:64:81
<4>[305.065334] 'wl p2p_ifadd' error -2
<6>[305.065412] CFG80211-ERROR)
    wl_cfg80211_add_virtual_iface: virtual iface add
```

c'è il livello aggiuntivo *brcmfmac*). Questo spiega:

1. il motivo per cui non sono riuscito a trovare **l'assegnamento del MAC address** nel sorgente dei driver. Cioè, perché è fatto nel *firmware proprietario di Broadcom*, a cui non posso accedere;
2. perché *wpa_supplicant* non è in grado di usare MAC address casuali nella creazione di interfacce di rete. Infatti, è il *firmware* a fare tutto il lavoro e scegliere il MAC address e non il software nel kernel o a livelli più elevati.

Detto ciò, ho dimostrato che è impossibile raggiungere gli obiettivi prefissati, a causa di limiti nel *firmware* proprietario (quindi NON *Open source* e NON modificabile). Infatti, come già spiegato, non si può avere interfacce di rete virtuali multiple con MAC address uguali e questi indirizzi non possono essere modificati, perché generati nel *firmware* proprietario.

Nonostante il lavoro sia di fatto concluso, voglio fare una precisazione riguardo ad *iw*. Giustamente, arrivato a questo punto mi sono chiesto per quale motivo *iw* fornisce la lista di *VIFC* sbagliata. Infatti, dice che è possibile avere due *VIF* P2P contemporaneamente, ma ciò si è dimostrato impossibile. Per generare la combinazione di interfacce utilizzabili, *iw* deve comunicare con i driver, ma il motivo per cui da informazioni sbagliate è che la funzione che fornisce quei risultati fallisce [83].

Conclusioni e sviluppi futuri

In questo lavoro di tesi ho mostrato lo studio dell'attuale implementazione di *Wi-Fi Direct* su dispositivi Android, con lo scopo di estendere a livello software le funzionalità di tale protocollo ed utilizzarlo principalmente nell'ambito delle *Opportunistic Neworks*. In particolare mi sono posto due obiettivi:

1. la scalabilità del protocollo;
2. la disponibilità della rete.

Dopo aver proposto le soluzioni teoriche ad essi, ho individuato un ostacolo legato all'implementazione di Android, in comune ad entrambi, cioè l'impossibilità per un dispositivo di far parte di due reti *Wi-Fi Direct* contemporaneamente. Ciò non è vietato dalle specifiche del protocollo, ma dal sistema operativo.

Quindi, dopo l'analisi dello stato dell'arte, mi sono accorto che tale problema non è mai stato realmente affrontato in Android. Le uniche soluzioni proposte sono tecniche per aggirarlo, sfruttando la combinazione con altri protocolli o studiando scenari limitati ed estremamente specifici. Il problema in questione è causato da limiti nella gestione di interfacce di rete virtuali da parte di Android, su cui le API ufficiali non permettono nessuno controllo allo sviluppatore.

Non avendo trovato ricerche che spieghino nel dettaglio la causa di tali limiti, ho affrontato il problema in modo strutturato e graduale, isolando le componenti del sistema operativo secondo i livelli dell'architettura e studiandone i più rilevanti per questo lavoro di tesi. L'ordine dei capitoli segue proprio la struttura a livelli dell'architettura. Infatti, ho scelto inizialmente di sperimentare le API fornite agli sviluppatori, studiandone i limiti. Per farlo ho realizzato due app.

- La prima, chiamata *PingPong*, mette in luce i problemi legati ad una particolare fase del protocollo *Wi-Fi Direct* in Android, chiamata *fase di Discovery*.
- La seconda è un esempio reale di applicazione di messaggistica per gestire reti costituite da un numero elevato e potenzialmente infinito di dispositivi. Quest'ultima applicazione risolve il problema della scalabilità in modo più che soddisfacente per il caso di studio scelto.

Poiché, l’obiettivo di questo lavoro di tesi si riferisce in modo più generale alle *Opportunistic Networks* e non a casi particolari di utilizzo, ho deciso di studiare il codice sorgente del sistema operativo. Cioè, come le API sono state realizzate da Google, per individuare il motivo per cui un dispositivo non può partecipare a due o più gruppi. Da questa analisi, sia della parte Java, sia da quella in C/C++ e JNI, ho scoperto che queste API sono ancora incomplete, oltre a risultare molto instabili.

Così, ho deciso di analizzare il livello inferiore dell’architettura del sistema operativo, fornendo prima una spiegazione teorica e poi pratica del processo di compilazione di Android e del kernel dai codici sorgente. Acquisite tali basi, ho mostrato il funzionamento del programma “nativo” che il framework Android utilizza per fornire all’utente la connettività Wi-Fi e *Wi-Fi Direct*, chiamato *wpa_supplicant*. Senza i limiti del framework Android e grazie all’uso di *wpa_supplicant*, ho individuato in modo più preciso i problemi riguardo le interfacce virtuali di rete. Quindi, ho modificato e ricompilato il kernel di Android per superare questo limite, fino a capire che si trova in una zona specifica del kernel, cioè dove ci sono i driver wireless.

Così, nell’ultimo capitolo ho analizzato tali driver, fornendo una breve introduzione storica e dettagli teorici riguardo l’architettura, per poi analizzarne l’implementazione e capire che il problema è legato ai driver proprietari della società che produce il chip hardware per la gestione della connettività wireless. Essendo proprietario (quindi NON *Open source* e NON modificabile) non è stato possibile analizzarne il codice sorgente. Nonostante ciò, ho mostrato come col passare delle generazioni di Smartphone tali limiti stiano diminuendo sempre più ed in futuro la soluzione proposta potrebbe diventare applicabile. Inoltre, dall’analisi fatta, ho dimostrato come sia possibile superare i limiti del framework Android e utilizzare direttamente software in un livello architetturale più basso. Questo potrà rilevarsi utile, qualora i produttori di hardware decidessero di implementare tali funzioni, ma Google non le esponesse agli sviluppatori.

Quindi, i risultati ottenuti sono importanti, perché dimostrano, finalmente, quale sia la reale causa dei limiti riscontrati in Android, sia nella mia sperimentazione, sia in quelle indicate nello stato dell’arte. Inoltre, l’applicazione chiama *Pigeon Messenger* dimostra un altro fatto importante, cioè che sebbene questi limiti siano presenti, esistono applicazioni in cui si possono comunque realizzare soluzioni funzionanti.

Questo lavoro di tesi, può costituire la base per future ricerche in tale ambito, per esempio studiare dispositivi più recenti e verificare se tali limiti siano ancora presenti. Allo stato attuale, è comunque possibile pensare a nuovi scenari di utilizzo in cui questi limiti risultino irrilevanti, infatti quello da me proposto riguardante la messaggistica è solo uno di essi. Inoltre, potrebbe essere interessante estendere l’app *Pigeon Messenger* a chat di gruppo ed implementare meccanismi di sicurezza per garantire la privacy.

Bibliografia

- [1] AllSeen Alliance. Allseen. <https://allseenalliance.org/>, 2015.
- [2] Wi-Fi Alliance. Wi-fi alliance. <http://www.wi-fi.org/who-we-are>, 2015.
- [3] Wi-Fi Alliance. Wi-fi direct technical specification. <http://www.wi-fi.org/discover-wi-fi/wi-fi-direct>, 2015.
- [4] Brian D. Noble Anthony J. Nicholson, Scott Wolchok. Juggler: Virtual networks for fun and profit. Gennaio 2008.
- [5] ArchLinux.org. wpa_supplicant arch linux. https://wiki.archlinux.org/index.php/WPA_supplicant_%28Italiano%29, 2015.
- [6] Gmane Arend van Spriel, Kathy Giori. Informations about linux wi-fi driver's architecture today. <http://comments.gmane.org/gmane.linux.kernel.wireless.general/136256>, 2015.
- [7] Arm. Arm. <http://www.arm.com>, 2015.
- [8] Paramvir Bahl, Pradeep Bahl, and Ranveer Chandra. Multinet: Connecting to multiple ieee 802.11 networks using a single wireless card. Technical Report MSR-TR-2003-46, Microsoft Research, August 2003.
- [9] Johannes Berg. Wi-fi control plane overview. 2009.
- [10] Johannes Berg. iw documentazione. <https://wireless.wiki.kernel.org/en/users/documentation/iw>, 2015.
- [11] Boingboing. Boingboing nome commerciale wi-fi. <http://boingboing.net/2005/11/08/wifi-isnt-short-for.html>, 2015.
- [12] Sunggeun Jin Bong-Jin Oh. A fast wifi direct device discovery with caching discovery schedulers for seamless n-screen services. *IARIA*, pages 5–8, 2014.
- [13] Broadcom. Broadcom bcm4339 5g wi-fi combo chip. <http://www.broadcom.com/press/release.php?id=s766879>, 2015.
- [14] Broadcom. Broadcom sito web. <http://www.broadcom.com>, 2015.

- [15] Stefano Cappa. Pigeon messenger github. <https://github.com/deib-polimi/PigeonMessenger>, 2015.
- [16] Stefano Cappa. Pingpong github. <https://github.com/deib-polimi/PingPong>, 2015.
- [17] Stefano Cappa. wpa_supplicant consiglio aggiunta parametro in wps_pbc. <http://permalink.gmane.org/gmane.linux.drivers.hostap/31804>, 2015.
- [18] Ccache. Ccache. <https://ccache.samba.org>, 2015.
- [19] Cernekee. Patch pie lollipop. <http://forum.xda-developers.com/google-nexus-5/development/fix-bypassing-pie-security-check-t2797731>, 2015.
- [20] Chainfire. Libreria libsuperuser. <https://github.com/Chainfire/libsuperuser>, 2015.
- [21] Chainfire. Selinux. <http://su.chainfire.eu/#selinux>, 2015.
- [22] Krishna Chaitanya. Wireless lan and linux together. <http://wire-less-comm.blogspot.in/2013/01/wireless-lan-and-linux-together.html>, 2015.
- [23] Onur Cinar. *Pro Android C++ with the NDK*. Apress, Dicembre 2012.
- [24] Jouni Malinen Dan Williams. wpa_supplicant con interfacce macvtap. <http://comments.gmane.org/gmane.linux.drivers.hostap/31818>, 2015.
- [25] Devmaze. Devmaze android metodi internal. <https://devmaze.wordpress.com/2011/01/18/using-com-android-internal-part-1-introduction/>, 2015.
- [26] Yufeng Duan Carlo Borgiattino Claudio Casetti Carla Fabiana Chiasseroni Paolo Giaccone Marco Ricca Fabio Malabocchia, Maura Turolla. Wi-fi direct multi-group data dissemination for public safety. *VDE*, pages 1–6, June 2014.
- [27] Faux123. Faux123 kernel. <http://faux.romhost.me>, 2015.
- [28] Free Software Foundation. Gnu make. <http://www.gnu.org/software/make/>, 2015.
- [29] P2P Foundation. Commotion. <http://p2pfoundation.net/Commotion>, 2015.
- [30] Francisco Franco. Franco kernel. <http://kernels.franco-lnx.net>, 2015.

- [31] Google. Android 5.1.0 - package android.net.p2p. https://android.googlesource.com/platform/frameworks/base/+log/android-5.1.0_r1/wifi/java/android/net/wifi/p2p, 2015.
- [32] Google. Android 5.1.0 - wifinative.java. [http://grepcode.com/search/usages?id=repository.grepcode.com\\$java\\$ext@com.google.android\\$android@5.0.2_r1@com\\$android\\$server\\$wifi@WifiNative\\$type=type&k=u](http://grepcode.com/search/usages?id=repository.grepcode.com$java$ext@com.google.android$android@5.0.2_r1@com$android$server$wifi@WifiNative$type=type&k=u), 2015.
- [33] Google. Android ndk. <https://developer.android.com/tools/sdk/ndk/index.html>, 2015.
- [34] Google. Android sdk api. <http://developer.android.com/guide/index.html>, 2015.
- [35] Google. Android studio. <http://developer.android.com/tools/studio/index.html>, 2015.
- [36] Google. Aosp git repository. <https://android.googlesource.com>, 2015.
- [37] Google. Aosp homepage. <https://source.android.com/>, 2015.
- [38] Google. Aosp lista dei branch. <https://android.googlesource.com/platform/manifest/+refs>, 2015.
- [39] Google. Binaries for nexus devices. <https://developers.google.com/android/nexus/drivers>, 2015.
- [40] Google. Chromecast. <https://www.google.it/chrome/devices/chromecast/>, 2015.
- [41] Google. Google source nuova password. <https://www.googlesource.com/new-password>, 2015.
- [42] Google. Guida ufficiale per compilare android e il suo kernel. <https://source.android.com/source/building.html>, 2015.
- [43] Google. mkbootimg. https://android.googlesource.com/platform/system/core/+/tools_r22.2/mkbootimg, 2015.
- [44] Google. Nexus5 file device.mk. <https://android.googlesource.com/device/lge/hammerhead/+/kitkat-cts-dev/device.mk>, 2015.
- [45] Google. Preview binaries for nexus devices. <https://developers.google.com/android/nexus/blobs-preview>, 2015.
- [46] Google. Qualcomm msm kernel sorgenti. <https://android.googlesource.com/kernel/msm>, 2015.

- [47] Google. Repo. <http://source.android.com/source/using-repo.html>, 2015.
- [48] Google. Repository git di android. <https://android.googlesource.com>, 2015.
- [49] Grepcode. Sito web grepcode. <http://grepcode.com/>, 2015.
- [50] Grepcode. Wifinative android kitkat su grepcode. http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/4.4.4_r1/android/net/wifi/WifiNative.java#WifiNative, 2015.
- [51] Grepcode. Wifinative android lollipop su grepcode. http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/5.0.2_r1/com/android/server/wifi/WifiNative.java#WifiNative, 2015.
- [52] Jong Won Kim Hayoung Yoon. Collaborative streaming-based media content sharing in wifi-enabled home networks. *IEEE*, pages 2193–2200, 2010.
- [53] Chiu C. Tan Hongxu Zhang, Yufeng Wang. Wd2: An improved wifi-direct group formation protocol. *ACM*, pages 55–60, 2014.
- [54] iFixit. Nexus 5 teardown. <https://www.ifixit.com/Teardown/Nexus+5+Teardown/19016#s53729>, 2015.
- [55] Jouni Malinen Ilan Peer. wpa_supplicant usare p2p0 o wlan0 per creare un gruppo. <http://comments.gmane.org/gmane.linux.drivers.hostap/31978>, 2015.
- [56] IlSole24Ore. Numero smartphone nel mondo nel 2013. <http://www.ilsole24ore.com/art/tecnologie/2013-05-31/smartphone-sono-miliardi-saranno-185531.shtml?uuid=Ab1oT80H>, 2015.
- [57] Infradead.org. Netlink protocol library suite (libnl). <http://www.infradead.org/~tgr/libnl/>, 2015.
- [58] Intel. Intel ccf. <https://software.intel.com/en-us/videos/introduction-to-the-intel-common-connectivity-framework-intel-ccf>, 2015.
- [59] P. M. Melliar-Smith Yung-Ting Chuang Isaì Michel Lombera, L. E. Moser. Peer management for itrust over wi-fi direct. *International Symposium on Wireless Personal Multimedia Communications, Atlantic City, NJ*, June 2013.
- [60] HP Jean Tourrilhes. Wext. http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Linux.Wireless.Extensions.html, 2015.

- [61] Stackoverflow jml, Rafal. Domanda su come funzionano nl80211 e cfg80211. <http://stackoverflow.com/questions/21456235/how-nl80211-cfg80211-libraries-work>, 2014.
- [62] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers*. O'Reilly, 2005.
- [63] Nick Modly Josh Thomas, Jeff Robble. Off grid communications with android, meshing the mobile world. *IEEE*, Novembre 2012.
- [64] Gmane Jouni Malinen. wpa_supplicant mostra un segno di limitazioni dei driver wi-fi. <http://permalink.gmane.org/gmane.linux.drivers.hostap/31999>, 2015.
- [65] Kernelnewbies.org. Macvtap. <http://virt.kernelnewbies.org/MacVTap>, 2015.
- [66] LearnLinuxConcepts. Learnlinuxconcepts procedura di boot. <http://learnlinuxconcepts.blogspot.in/2014/02/android-boot-sequence.html>, 2015.
- [67] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.
- [68] Pietro Lio' LU Xiaofeng, HUI Pan. Offloading mobile data from cellular networks through peer-to-peer wifi communication: A subscribe-and-send architecture. *IEEE*, pages 35–46, June 2013.
- [69] lyengar MIUI. Ramdisk. <http://en.miui.com/thread-15659-1-1.html>, 2015.
- [70] Jouni Malinen. wpa_cli utilizzare l'interfaccia corretta. <http://permalink.gmane.org/gmane.linux.drivers.hostap/31673>, 2015.
- [71] Jouni Malinen. wpa_supplicant. http://w1.fi/wpa_supplicant/, 2015.
- [72] Jouni Malinen. wpa_supplicant file di configurazione. http://w1.fi/cgit/hostap/plain/wpa_supplicant/wpa_supplicant.conf, 2015.
- [73] Giovanni Minutiello-Roberta Paris Marco Conti, Franca Delmastro. Experimenting opportunistic networks with wifi direct. *IEEE*, Novembre 2013.
- [74] Oracle. Oracle javadoc. <http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html>, 2015.

- [75] Gian Paolo Rossi Paolo Meroni, Elena Pagani. An opportunistic platform for android-based mobile devices. *ACM*, 2010.
- [76] pbatard. unmkbootimg. <https://github.com/pbatard/bootimg-tools>, 2015.
- [77] Ilan Peer. wpa_supplicant p2p_group_remove. <http://comments.gmane.org/gmane.linux.drivers.hostap/31588>, 2015.
- [78] Thomas Petazzoni. Device tree blob. <http://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf>, 2015.
- [79] Phandroid. Streaming video da smartphone a tv. <http://phandroid.s3.amazonaws.com/wp-content/uploads/2013/07/Chromecast-Featured-ready-to-cast.jpg>, 2015.
- [80] The Serval Project. The serval project. <http://www.servalproject.org>, 2015.
- [81] Qualcomm. Qualcomm sito web. <http://www.qca.qualcomm.com>, 2015.
- [82] Broadcom Rafal Mileki. Broadcom brcmsmac(pcie) and brcmfmac(sdio/usb) drivers. <https://wireless.wiki.kernel.org/en/users/drivers/brcm80211>, 2015.
- [83] Stefano Cappa Gmane Rafal Mileki, Krishna Chaitanya. Iw can fail? <http://comments.gmane.org/gmane.linux.kernel.wireless.general/133848>, 2015.
- [84] Sylvain Ratabouil. *Android NDK Beginner's Guide*. PACKT, Gennaio 2012.
- [85] Chaitra Satish. Inter-vehicular communication for collision avoidance using wi-fi direct. Master's thesis, Rochester Institute of Technology, Maggio 2014.
- [86] SergioSolis. Android booting process. <https://community.freescale.com/docs/DOC-102546>, 2015.
- [87] Stericson. Busybox google playstore. <https://play.google.com/store/apps/details?id=stericson.busybox&hl=it>, 2015.
- [88] Wikipedia. Android custom rom. http://en.wikipedia.org/wiki/List_of_custom_Android_firmwares, 2015.
- [89] Wikipedia. Annotazione java. http://it.wikipedia.org/wiki/Annotazione_%28Java%29, 2015.
- [90] Wikipedia. Bonjour. <http://it.wikipedia.org/wiki/Bonjour>, 2015.

- [91] Wikipedia. Daemon. http://it.wikipedia.org/wiki/Demone_%28informatica%29, 2015.
- [92] Wikipedia. Definizione di toolchain. <http://it.wikipedia.org/wiki/Toolchain>, 2015.
- [93] Wikipedia. Dsrc. http://en.wikipedia.org/wiki/Dedicated_short-range_communications, 2015.
- [94] Wikipedia. Ieee 802.11s. http://en.wikipedia.org/wiki/IEEE_802.11s, 2015.
- [95] Wikipedia. iphone edge/2g. [http://en.wikipedia.org/wiki/IPhone_\(1st_generation\)](http://en.wikipedia.org/wiki/IPhone_(1st_generation)), 2015.
- [96] Wikipedia. Java reflection. http://en.wikipedia.org/wiki/Reflection_%28computer_programming%29#Java, 2015.
- [97] Wikipedia. Jni. http://it.wikipedia.org/wiki/Java_Native_Interface, 2015.
- [98] Wikipedia. Mesh network. http://en.wikipedia.org/wiki/Wireless_mesh_network, 2015.
- [99] Wikipedia. Modi di funzionamento di una rete wi-fi. http://en.wikipedia.org/wiki/Wireless_LAN, 2015.
- [100] Wikipedia. Multiplexing. <http://en.wikipedia.org/wiki/Multiplexing>, 2015.
- [101] Wikipedia. Nas. http://it.wikipedia.org/wiki/Network_Attached_Storage, 2015.
- [102] Wikipedia. Phablet. <http://it.wikipedia.org/wiki/Phablet>, 2015.
- [103] Wikipedia. Pic e pie. http://en.wikipedia.org/wiki/Position-independent_code, 2015.
- [104] Wikipedia. Protocollo dls. http://en.wikipedia.org/wiki/IEEE_802.11e-2005#Direct_Link_Setup, 2015.
- [105] Wikipedia. Ram disk. http://en.wikipedia.org/wiki/RAM_drive, 2015.
- [106] Wikipedia. Smart tv. http://it.wikipedia.org/wiki/Smart_TV, 2015.
- [107] Wikipedia. Suplicant. [http://en.wikipedia.org/wiki/Suplicant_\(computer\)](http://en.wikipedia.org/wiki/Suplicant_(computer)), 2015.

- [108] Wikipedia. Upnp. http://it.wikipedia.org/wiki/Universal_Plug_and_Play, 2015.
- [109] Wikipedia. User space o userland. http://en.wikipedia.org/wiki/User_space, 2015.
- [110] Wikipedia. V2i. http://en.wikipedia.org/wiki/Vehicle_infrastructure_integration, 2015.
- [111] Wikipedia. V2v. <http://it.wikipedia.org/wiki/V2V>, 2015.
- [112] Wikipedia. Wi-fi. <http://it.wikipedia.org/wiki/Wi-Fi>, 2015.
- [113] Wikipedia. Wikipedia dlna. http://it.wikipedia.org/wiki/Digital_Living_Network_Alliance, 2015.
- [114] Linux Wireless. Tipi di interfacce di rete virtuali. <https://wireless.wiki.kernel.org/en/users/documentation/modes>, 2015.
- [115] Linux Wireless. Usa diverse interfacce virtuali per uso concorrente. https://wireless.wiki.kernel.org/en/developers/p2p/howto#using_multiple_virtual_interfaces_for_concurrent_usage, 2015.
- [116] Karim Yaghmour. *Embedded Android, porting, extending and customizing*. O'Reilly Media, Marzo 2013.

Appendice A

File 51-android.rules per Ubuntu 14.04

Listato A.1: File di configurazione per rilevare Smartphone/Tablet Android in Ubuntu 14.04

```
#Acer
SUBSYSTEM=="usb" , ATTR{idVendor}=="0502" , MODE="0666"
#ASUS
SUBSYSTEM=="usb" , ATTR{idVendor}=="0b05" , MODE="0666"
#Dell
SUBSYSTEM=="usb" , ATTR{idVendor}=="413c" , MODE="0666"
#Foxconn
SUBSYSTEM=="usb" , ATTR{idVendor}=="0489" , MODE="0666"
#Garmin-Asus
SUBSYSTEM=="usb" , ATTR{idVendor}=="091E" , MODE="0666"
#Google
SUBSYSTEM=="usb" , ATTR{idVendor}=="18d1" , MODE="0666"
#HTC
SUBSYSTEM=="usb" , ATTR{idVendor}=="0bb4" , MODE="0666"
#Huawei
SUBSYSTEM=="usb" , ATTR{idVendor}=="12d1" , MODE="0666"
#K-Touch
SUBSYSTEM=="usb" , ATTR{idVendor}=="24e3" , MODE="0666"
#KT Tech
SUBSYSTEM=="usb" , ATTR{idVendor}=="2116" , MODE="0666"
#Kyocera
SUBSYSTEM=="usb" , ATTR{idVendor}=="0482" , MODE="0666"
#Lenevo
SUBSYSTEM=="usb" , ATTR{idVendor}=="17EF" , MODE="0666"
```

```
#LG
SUBSYSTEM=="usb" , ATTR{idVendor}=="1004" , MODE=="0666"
#Motorola
SUBSYSTEM=="usb" , ATTR{idVendor}=="22b8" , MODE=="0666"
#NEC
SUBSYSTEM=="usb" , ATTR{idVendor}=="0409" , MODE=="0666"
#Nook
SUBSYSTEM=="usb" , ATTR{idVendor}=="2080" , MODE=="0666"
#Nvidia
SUBSYSTEM=="usb" , ATTR{idVendor}=="0955" , MODE=="0666"
#OTGV
SUBSYSTEM=="usb" , ATTR{idVendor}=="2257" , MODE=="0666"
#Pantech
SUBSYSTEM=="usb" , ATTR{idVendor}=="10A9" , MODE=="0666"
#Philips
SUBSYSTEM=="usb" , ATTR{idVendor}=="0471" , MODE=="0666"
#PMC-Sierra
SUBSYSTEM=="usb" , ATTR{idVendor}=="04da" , MODE=="0666"
#Qualcomm
SUBSYSTEM=="usb" , ATTR{idVendor}=="05c6" , MODE=="0666"
#SK Telesys
SUBSYSTEM=="usb" , ATTR{idVendor}=="1f53" , MODE=="0666"
#Samsung
SUBSYSTEM=="usb" , ATTR{idVendor}=="04e8" , MODE=="0666"
#Sharp
SUBSYSTEM=="usb" , ATTR{idVendor}=="04dd" , MODE=="0666"
#Sony Ericsson
SUBSYSTEM=="usb" , ATTR{idVendor}=="0fce" , MODE=="0666"
#Toshiba
SUBSYSTEM=="usb" , ATTR{idVendor}=="0930" , MODE=="0666"
#ZTE
SUBSYSTEM=="usb" , ATTR{idVendor}=="19D2" , MODE=="0666"
```