Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea in Informatica

MONITORAGGIO DI DISPOSITIVI WIRELESS: PRESENTAZIONE USERSPACE

Tesi di Laurea in Reti di Calcolatori

Relatore:
Dott.
VITTORIO GHINI

Presentata da:
MATTEO BIGOI

Sessione II
Anno Accademico 2009/2010

Parole chiave:
wireless, VoIP, affidabilita', protocolli, cross-layer

Introduzione

La grande diffusione di dispositivi mobili dotati di tecnologia wireless IEEE802.11 ha portato a un forte aumento dell'utilizzo di applicazioni di fonia che sfruttano protolli VoIP e che necessitano di una notevole affidabilita' a livello di connessione di rete. A causa della intrinseca instabilita' e sensibilita' ai disturbi delle reti wireless, unita alle limitazioni del protocollo UDP utilizzato da dette applicazioni, si presenta il problema dell'affidabilita' della trasmissione dei dati nelle comunicazioni VoIP over wireless.

In questo studio viene fornito un sistema di controllo della recezione dei pacchetti UDP da parte del primo nodo interessato nella comunicazione, nel caso specifico un Wireless Access Point. E' stato necessario un excursus tra i vari livelli del protocollo di rete per estrarre informazioni sui pacchetti UDP in transito e associarli all'acknowledgement del pacchetto del livello trasporto (IEEE80211) che lo contiene. L'obbiettivo finale del lavoro di tesi e' proporre un sistema che permetta alle applicazioni VoIP di richiedere conferma dell'avvenuta ricezione dei dati trasmessi in modo da minimizzare la mancanza di controllo di connessione del protocollo UDP. Queste informazioni vengo esportate sul filesystem virtuale SysFS del kernel Linux. Viene inoltre fornita una infrastruttura di compilazione e test del codice prodotto, le librerie userspace necessarie e un semplice programma dimostrativo.

Lo studio si presenta come estensione e aggiornamento della soluzione basata su system call proposta dai Dottori Codecá Lara, Grassi Giulio.[16]. Viene inoltre trattata in dettaglio una gamma di possibili soluzioni in base alla loro semplicita' di implementazione e attinenza allo scenario proposto.

La struttura del documento si articola nei seguenti capitoli:

Primo Capitolo : Viene descritto il contesto di ricerca, i sistemi software e hardware operanti in un ambiente wireless e la pila protocollare nella quale si intende implementare il meccanismo cross-layer.

Secondo Capitolo : E' trattato in dettaglio l'obiettivo della ricerca, i diversi dispositivi wireless utilizzati e alcune scelte implementative.

Terzo Capitolo : In questo capitolo sono esposte le scelte architetturali e implementative della versione del meccanismo cross-layer implementato dai Dottori Codecá Lara e Grassi Giulio[16]. Viene inoltre trattato l'aspetto prestazionale del progetto.

Quarto Capitolo : Vengono descritti ad alto livello il filesystem SysFS ad i kernel thread, l'infrastruttura realizzata a livello kernel e le librerie disponibili in userspace. Seguono brevi esempi di utilizzo del codice di test e relativo output.

Quinto Capitolo : Vengono delineati alcuni possibili sviluppi futuri per espandere e migliorare il lavoro presentato, in particolare riguardo i nuovi standard IEEE802.11 e l'inclusione del codice nel kernel di Linux.

Sesto Capitolo : Conclusioni della ricerca e valutazioni personali dell'autore.

Indice

Introduzione **i**

1 Scenario **1**

1.1 Contesto della ricerca 2

1.2 Tecnologia Wireless 802.11 3

1.3 Dispositivi 6

1.3.1 Access Point 6

1.3.2 Schede di rete wireless 6

1.4 Layer e protocolli 9

1.4.1 Stack di rete 9

1.4.2 Livello Data Link (livello 2 del modello OSI) 10

1.4.3 Livello Network (livello 3 del modello OSI) 13

1.4.4 Livello Trasporto (livello 4 del modello OSI) 16

1.4.5 Livello Applicazione (livello 7 del modello OSI) 17

1.5 Crittografia 21

2 Obiettivi e sperimentazione **23**

2.1 Obiettivi 23

2.1.1 Cenni Storici 23

2.1.2 Ack dal prime hop e interattivita' 24

2.2 Sperimentazione 25

3 Progettazione **29**

3.1 Meccanismo Cross-Layer 29

3.2 Implementazione basata sui socket 30
3.2.1 Hardware utilizzato 30
3.2.2 Interfaccia firmware-driver 30
3.2.3 Canale di ritorno di errore dei socket 35
3.2.4 Test e valutazione 41

4 Implementazione basata su SysFS 43
4.1 Alternative per comunicazione kernelspace-userspace 45
4.2 SysFS 46
4.2.1 Organizzazione generale 47
4.2.2 Struttura del filesystem virtuale 47
4.2.3 Kernel Objects 48
4.3 Infrastruttura a livello kernel 53
4.3.1 Kernel thread 54
4.3.2 Kernel Module 56
4.4 Implementazione 57
4.4.1 Organizzazione dell' albero sorgente 58
4.4.2 Rtlkill in dettaglio 58
4.4.3 Interfaccia del modulo 71
4.4.4 Codice di test 72

5 Sviluppi Futuri 77
5.1 IPv6 77
5.2 Monitor 78
5.3 IEEE 802.11r 78
5.4 Kernel Linux 79
5.5 Sistemi Embedded 79

6 Conclusioni 81

A APPENDICE 5

Elenco delle figure

1.1 Hot Spot Fon a Bologna (maps.fon.com) 2
1.2 Meccanismo di interazione 3
1.3 Nodo nascosto 12
1.4 Frame 802.11 13
1.5 Struttura ACK Frame 14
1.6 Header del pacchetto IP 15
1.7 Header del pacchetto UDP 17

2.1 Modifiche cross-layer 26

3.1 Panoramica meccanismo Syscall 36

4.1 Panoramica meccanismo SysFS 44

Elenco delle tabelle

1.1 Codice protocolli Transport Level 16

Capitolo 1

Scenario

Una wireless LAN *(WLAN - wireless local area network)* consiste in una rete formata da due o piu' computer o dispositivi connessi senza l'utilizzo di cavi. Essa utilizza svariate tecnologie di trasmissione tramite onde radio, ad esempio espansione di spettro oppure OFDM[1]. Questa tecnologia permette comunicazioni in un' area geograficamente limitata, consentendo mobilita' all'interno dell'area di copertura del segnale radio.

Nel contesto privato, le reti senza fili sono state soggette a una forte espansione grazie alla facilita' di installazione e alla crescente diffusione di notebook, palmari e altri dispositivi simili. Alcuni tipi di esercizi come coffee-shop o centri commerciali hanno iniziato ad offrire questo tipo di servizio ai clienti, e nelle citta' sono nati progetti per piccole reti pubbliche, per esempio quelle di biblioteche e universita', oppure grandi reti civiche che forniscono connettivita' ad ampie aree urbane[2].

[1]Orthogonal Frequency-Division Multiplexing (OFDM): e' un tipo di modulazione multi-portante che utilizza un numero elevato di sottoportanti ortogonali tra loro.

[2]Un esempio e' la Rete Civica Iperbole (www.comune.bologna.it) del Comune di Bologna

1.1 Contesto della ricerca

Oggetto dello studio e' la mobilita' mediante connettivita' wireless in un ambiente geograficamente vasto, per esempio una citta', che richiede copertura del segnale radio globale, anche attraverso reti locali indipendenti, e credenziali per accedere a tutti gli hot spot presenti lungo il percorso.

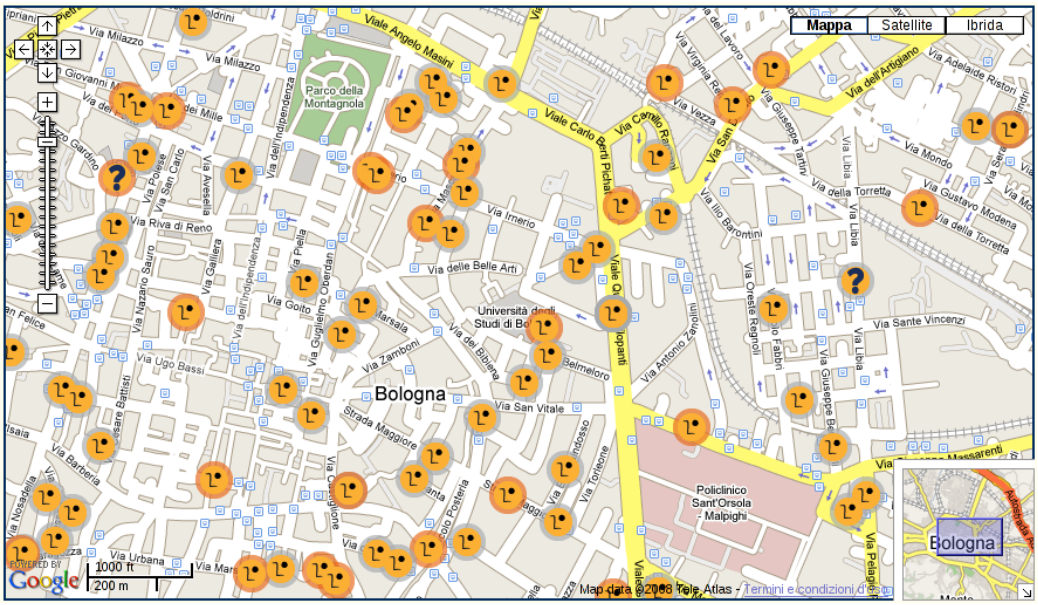

Figura 1.1: Hot Spot Fon a Bologna (maps.fon.com)

Precisamente la ricerca risponde all'esigenza di avere un'interazione effettiva con il flusso dei dati in un contesto UDP. Il protocollo UDP non si basa sull'astrazione di connessione, pertanto non espone informazioni riguardanti l'effettiva trasmissione dei pacchetti inviati. Questo protocollo, utilizzato sempre piu' spesso in ambiti di mobilita' e VoIP, ha raggiunto una diffusione tale da giustificare studi che possano mitigare i suoi limiti. Il nostro scopo e' fornire il maggior numero di informazioni possibili riguardanti la sorte dei pacchetti trasmessi. Per poterlo fare lavoreremo all'interno dello stack di rete implementando un servizio cross-layer utilizzabile in userspace.

La scelta si e' orientata immediatamente verso un sistema operativo facile da ottenere, poco costoso, di cui si avesse gia' sufficiente esperienza e che per-

mettesse di essere modificato e utilizzato direttamente durante il lavoro. Il kernel di Linux, open source e ben documentato, e' stato la prima scelta. Data la durata dei lavori lo sviluppo si e' mosso dalla versione 2.6.23.1 alla versione 2.6.25.6. Il kernel di Linux e' stato portato su numerose architetture e sistemi differenti, supportando svariate famiglie di hardware fra cui i dispositivi wireless USB.

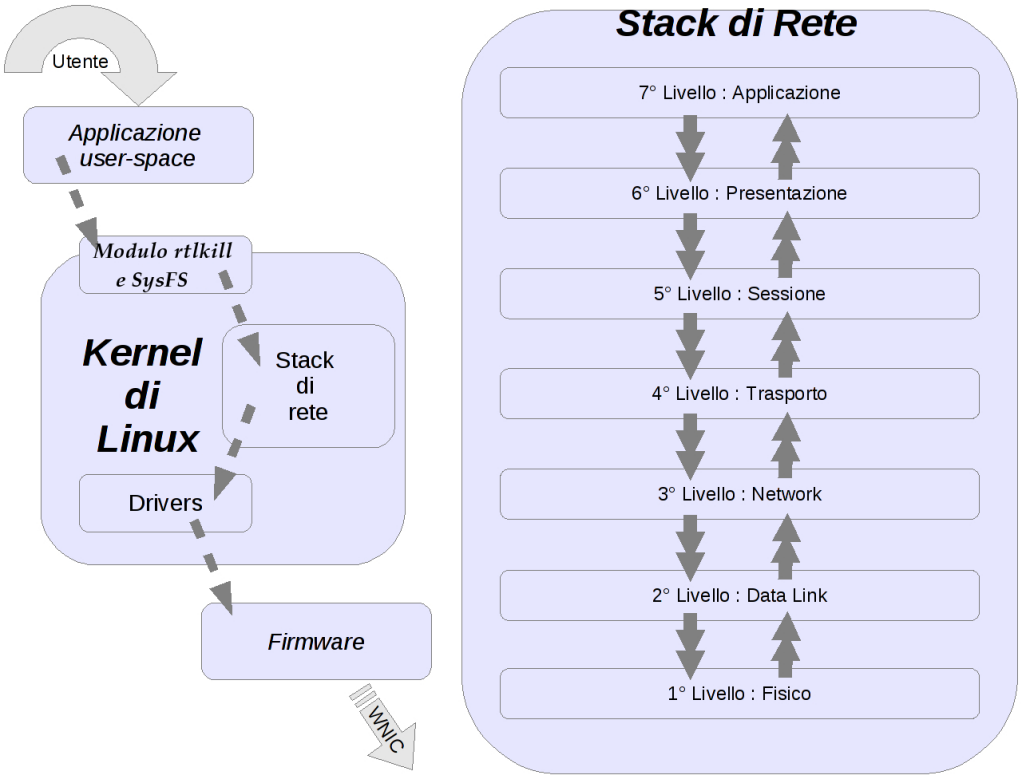

Figura 1.2: Meccanismo di interazione

Segue una panoramica sui servizi, protocolli e problematiche relative all'infrastruttura delineata precedentemente.

1.2 Tecnologia Wireless 802.11

Standard Lo *IEEE802.11*[3, 4] o *Wi-Fi* definisce uno standard per le reti WLAN sviluppato dal working group 11 della famiglia IEEE802; viene defini-

to il livello fisico e MAC del modello ISO-OSI[3] e specificata sia l'interfaccia tra client e access point che quella tra piu' client.

Nella stessa famiglia i protocolli dedicati alla trasmissione delle informazioni sono a, b e g. Gli altri standard, cioe' c, d, e, f, h, ecc. riguardano estensioni dei servizi base e miglioramenti di servizi gia' disponibili. Il primo protocollo largamente diffuso e' stato il b; in seguito si sono diffusi il protocollo a e soprattutto il protocollo g. La sicurezza e' stata inclusa in uno standard a parte, l'*802.11i*.

Lo *IEEE802.11b* e *IEEE802.11g* utilizzano lo spettro di frequenze (banda ISM[4]) in un intorno dei 2,4 GHz, mentre lo *IEEE802.11a* utilizza la banda ISM dei 5,8 GHz.

Sicurezza La cifratura, nelle versioni originali dei protocolli 802.11, era basata sulla crittografia *WEP (Wired Equivalent Privacy)*; nel 2001 un gruppo di ricercatori dell'Universita' di Berkeley presento' uno studio nel quale descriveva le vulnerabilita' del protocollo. Il gruppo si concentro' sull'algoritmo di cifratura utilizzato dal WEP (**RC4**[5]), che nell'implementazione scelta per lo standard 802.11 era molto debole e facilmente forzabile mediante attacchi statistici. Per ovviare al problema l'IEEE si mise al lavoro per progettare un'evoluzione dello standard, in contemporanea con la Wi-Fi Alliance, i cui lavori diedero vita nel 2003 al WPA[6] e nel giugno del 2004 vennero rilasciate le specifiche dell'IEEE 802.11i. Il nuovo standard rendeva le reti wireless piu' sicure e la Wi-Fi Alliance lo adotto' subito sotto il

[3]Open Systems Interconnection Basic Reference Model (OSI Reference Model): e' una descrizione a livelli di astrazione dei protocolli e dell'implementazione delle comunicazioni in una rete di computer.

[4]Banda ISM (Industrial, Scientific and Medical) e' il nome assegnato dall'Unione Internazionale delle Telecomunicazioni (ITU) ad un insieme di porzioni dello spettro elettromagnetico riservate alle applicazioni radio non commerciali, per uso industriale, scientifico e medico.

[5]RC4 o Rivest Cipher 4 e' un algoritmo di cifratura a stream progettato da Ron Rivest della RSA Security nell'anno 1987.

[6]Wi-Fi Protected Access

nome commerciale di WPA2. Il WPA2 abbandona l'RC4 come algoritmo di codifica per passare al piu' sicuro AES [7], sostituendo anche lo SHA1 al MD5 per garantire l'integrita' della comunicazione. La diffusione del nuovo algoritmo sui dispositivi gia' esistenti fu reso semplice dal costo contenuto dell'aggiornamento del firmware di sistema.

Vantaggi Le reti wireless, non risentendo dei costi di cablaggio e coprendo aree geografiche relativamente vaste, rispetto al costo, si sono diffuse rapidamente, favorite da una forte concorrenza fra i produttori di hardware e una notevole semplicita' di installazione e uso. L'esistenza di uno standard condiviso e certificato garantisce l'interoperabilita' di diverse famiglie di prodotti anche in condizioni di elevata mobilita'.

Svantaggi La latenza della comunicazione basata su schede wireless e' superiore di 1-3 ms a quella basata su cavo (da notare che le connessioni GPRS/UMTS hanno latenze nell'ordine di 200-400ms). Le connessioni Wi-Fi 802.11a/g, condividendo il mezzo fisico di trasmissione, sono molto sensibili all'affollamento dei client presso lo stesso hot spot e alle eventuali interferenze, essendo la banda operativa vicina ad altri spettri usati in molti altri ambiti. La sicurezza delle trasmissioni e' anch'essa minacciata dalla difficile contenibilita' delle onde radio, creando opportunita' per attacchi MITM[8] o DOS[9].

[7] Advanced Encryption Standard

[8] Man in The Middle: un attaccante che, interpostosi fra due client che comunicano fra di loro, intercetta in modo trasparente la comunicazione

[9] Denial Of Service: utilizzando un dispositivo che emetta disturbi sulle frequenze utilizzate dai client e' possibile paralizzare la rete wireless

1.3 Dispositivi

1.3.1 Access Point

Un Access Point e' un emettitore fisso di onde radio che, collegato generalmente a una rete cablata o a un'altro access point, permette ad un client (utente mobile) di connettersi tramite le onde radio a un servizio retrostante. Vi sono tre modalita' principali di funzionamento di un Access Point.

Master Mode E' la modalita' standard di funzionamento dell'AP, fornendo un punto di accesso ai dispositivi mobili (Laptop, Cellulari, Palmari, ecc.).

Bridged Mode Consiste nella creazione di un link wireless tra due (point-to-point) o piu' (point-to-multipoint) access point.

Repeater Mode Un Access Point configurato in questa modalita' permette di aumentare il raggio di copertura di una rete wireless, riducendo drasticamente il troughput. Questo scenario e' quello tipico del roaming, in cui un client wireless si sposta attraversando le aree di copertura di diversi Access Point mantenendo attivo il collegamento radio.

1.3.2 Schede di rete wireless

Un controller di rete wireless (WNIC - Wireless network interface controller) e' una periferica hardware che permette ad un computer di utilizzare una infrastruttura di rete non cablata e lavora ai primi due livelli del modello ISO-OSI. Il dispositivo utilizza un'antenna e la comunicazione avviene attraverso onde radio nelle bande di appartenenza.

Funzionamento Esistono due essenziali modalita' di funzionamento, conosciute come *infrastruttura* e *ad-hoc*.

Nel modello a *infrastruttura* e' necessaria la presenza di un AP. Questo gestira' tutto il trasferimento dei dati e si comportera' da hub centrale. In questa modalita' tutti i client devono essere connessi all'AP con lo stesso SSID [10] e nel caso in cui la rete sia cifrata essi devono essere a conoscenza del mezzo di autenticazione.

In una rete *ad-hoc* non e' necessario un AP poiche' ogni dispositivo puo' interfacciarsi direttamente con gli altri nodi della rete. Anche in questo caso tutti i client devono essere impostati con lo stesso SSID sullo stesso canale.

La velocita' massima di trasferimento si puo' ottenere solo se realmente vicini alla fonte della trasmissione. I dispositivi negoziano dinamicamente la velocita' di trasmissione per minimizzare la perdita di pacchetti (e quindi i tentativi di ritrasmissione).

In un desktop computer una WNIC e' solitamente connessa mediante il bus PCI, ma esistono alternative come USB, PC Card e soluzioni integrate come Mini PCI e Mini PCI Express.

Dispositivi USB

L'Universal Serial Bus e' uno standard di comunicazione seriale che consente di collegare diverse periferiche ad un computer. Progettato per consentire a piu' periferiche di essere connesse usando una sola interfaccia ed un solo tipo di connettore, permette l'hot plug[11] dei piu' svariati dispositivi..

I device USB WNIC sono di facile reperibilita' e semplici da utilizzare. Spesso le loro caratteristiche non li inseriscono al top della categoria, ma hanno vantaggi non trascurabili, fra cui il costo ridotto e la praticita' d'uso in un ambiente di test in cui sia necessario utilizzare piu' dispositivi.

[10] Service Set Identifier: nome univoco che identifica, insieme all'indirizzo mac dell'access point, una cella wireless

[11] Si dice hot plug un'interfaccia che permette il collegamento e/o lo scollegamento di un dispositivo anche a sistema avviato

Dispositivi PCMCIA

Si definisce PCMCIA[12] la porta o il dispositivo nato per l'espansione delle funzionalita' dei computer portatili. Lo standard delle PC Card nasce nel 1991 dalla fusione dei due standard JEIDA[13] 4.1 e PCMCIA 2.0.

Originariamente le PC Card furono pensate come slot di espansione per la memoria dei portatili, successivamente nello stesso formato vennero creati svariati dispositivi come schede di rete, modem e hard disk. E' da notare che la funzionalita' originaria e' andata in disuso. I dispositivi wireless PCMCIA sono degni di nota in quanto permettono di esporre esternamente l'antenna aumentando in modo sensibile la qualita' del segnale.

Dispositivi Mini PCI e PCI Express Mini

Mini PCI e' un bus standard che permette di connettere dispositivi periferici al computer. E' un'evoluzione del PCI bus (Peripheral Component Interconnect) progettato per computer portatili e altri dispositivi di piccole dimensioni.

PCI Express Mini Card e' l'evoluzione dell' interfaccia PCI Express usando il bus Mini PCI. Sviluppata da PCI-SIG[14], l'interfaccia supporta connettivita' PCI Express, USB 2.0 e Firewire. Ogni device puo' scegliere l'interfaccia che ritiene piu' appropriata al suo scopo.

I device WLAN con queste connessioni sono comunemente integrati nei laptop, rappresentando un ottimo compromesso fra potenza radio erogata e assorbimento elettrico.

[12]Personal Computer Memory Card International Association: associazione che sviluppo' il primo dispositivo.

[13]Le JEIDA sono memory card giapponesi precedentemente diffuse a quelle PCMCIA

[14]Il PCI-SIG o Peripheral Component Interconnect Special Interest Group e' un gruppo responsabile delle specifiche degli standard dei bus Peripheral Component Interconnect (PCI), PCI-X, and PCI Express (PCIe).

1.4 Layer e protocolli

1.4.1 Stack di rete

Affinche' una infrastruttura di rete possa funzionare in modo corretto ed efficace, e' necessaria una strutturazione dettagliata del software che la deve gestire. Quasi tutte le reti reali sono gestite da una struttura gerarchica di livelli che astraggono e risolvono i problemi presenti, mascherando i vari strati della comunicazione tramite interfacce condivise e standard. Numero, nome e caratteristiche dei vari livelli cambiano a seconda della tipologia di rete ma la dinamica concettuale e' comune: ogni livello utilizza i servizi forniti dal layer sottostante per offrire ulteriori servizi al layer superiore[1, 2].

Questo principio e' comunemente utilizzato nell'ingegneria e si basa sul modellare ogni layer come una black box, potendolo utilizzare semplicemente conoscendone l'interfaccia e il formato dell'informazione in uscita dopo l'elaborazione. Tra due postazioni la comunicazione avviene per livelli, cioe' un dato livello nella macchina A comunichera' con il livello corrispondente nella macchina B. Le regole e le convenzioni utilizzate in questa comunicazione vengono definite come il protocollo del layer.

Il punto di contatto fra due livelli adiacenti e' un'interfaccia sulla quale vengono definite le operazioni e i servizi forniti dal livello sottostante a quello superiore. Oltre a minimizzare la quantita' di informazioni necessarie per la comunicazione, rende possibile la modifica integrale dell'implementazione di un livello (mantenendo ovviamente intatta l'interfaccia) senza rendere necessarie modifiche ai livelli adiacenti.

La specifica di questa infrastruttura deve contenere tutte le informazioni che ne permettano la creazione dal punto di vista hardware e software.

Siccome implementazione e specifiche di due macchine differenti possono essere diverse, affinche' vi possa essere comunicazione e' necessario che vi sia un *linguaggio* comune condiviso dalle due parti; un esempio si ha nella comunicazione tramite protocollo IP tra due architetture con specifiche e implementazioni differenti.

Stack di rete e linux Si e' deciso di procedere utilizzando come riferimento lo stack di rete *mac80211* recentemente inserito nel kernel stabile di Linux dalla versione 2.6.18, in quanto il codice si presenta meglio strutturato e documentato e rappresenta un punto di arrivo per la migrazione dei vecchi driver che ancora si basano sul vecchio stack *softmac* e *generic_ieee80211*.

1.4.2 Livello Data Link (livello 2 del modello OSI)

Il livello Data Link si pone il problema della connessione diretta di due apparecchi. Si hanno quindi due sistemi connessi mediante un canale di comunicazione che si comporta come un cavo, per esempio un cavo coassiale, una linea telefonica, oppure una connessione punto-punto mediante microonde.

Mac 802.11

Il protocollo MAC (Media Access Control) e' un sottolivello del Data Link Layer. Permette l'indirizzamento univoco e l'accesso ad un canale condiviso. Questo sottolivello funge da interfaccia tra il sottolivello LLC (Logical Link Control, rientra anchesso nel Data Link Layer) e il livello fisico (primo layer del modello OSI). Inoltre emula un canale di comunicazione full-duplex in una rete ad accesso multiplo rendendo possibile comunicazioni in unicast, multicast e broadcast.

In una comunicazione point-to-point full-duplex il livello MAC non e' necessario, ma viene quasi sempre incluso in questi protocolli per questioni di compatibilita'.

Indirizzamento Gli indirizzi gestiti da questo layer vengono chiamati indirizzi fisici o indirizzi MAC. Un identificativo MAC e' un numero di serie univoco associato alla scheda di rete, rendendo possibile la consegna dei pac-

chetti ad un nodo all'interno della sottorete[15]. Un esempio di rete fisica e' una LAN Ethernet, con eventuale access point wireless che condivide con la rete cablata la stessa gerarchia di indirizzi MAC a 48 bit.

Meccanismo di accesso al canale Il meccanismo fornito da questo sottolivello e' anche conosciuto come *multiple access protocol* e rende possibile a piu' nodi di collegarsi e condividere lo stesso mezzo fisico. Questo protocollo rileva e previene le collisioni dei pacchetti in transito mediante un meccanismo di contesa del canale oppure attraverso l'allocazione di risorse riservate per la creazione di un canale logico. Questo meccanismo di controllo si fonda sul multiplexing a livello fisico.

Il protocollo piu' diffuso nelle reti wired e' chiamato CSMA/CD (Carrier sense multiple access con collision detection). Viene utilizzato all'interno dello stesso dominio di collisione, per esempio una sottorete gestita da un hub o un bus Ethernet. Una rete solitamente e' formata da vari domini di collisione interconnessi mediante bridges e switches.

Nell'ambito delle reti wireless e' predominante il CSMA/CA (Carrier sense multiple access con collision avoidance). Utilizzando questa tecnica quando un nodo mobile desidera trasmettere deve ascoltare il canale per un tempo prestabilito per verificare che non vi sia attivita'. Se il canale si trova in stato *idle* il client ha il permesso di trasmettere, in caso contrario deve posticipare l'invio dei dati. Nelle reti gestite tramite CSMA/CA, ogni client deve, prima di inziare la trasmissione, inviare un pacchetto in broadcast che istruisce le altre stazioni a sospendere l'attivita' in attesa dei dati, liberando il canale.

La prevenzione delle collisioni migliora le performance del CSMA. In caso di canale occuppato il tempo di attesa prima del test successivo viene generato casualmente, il che riduce notevolmente la probabilita' di collisioni. Contrariamente a quanto avviene nelle reti cablate, un client che trasmette

[15]Per sottorete si intende un rete fisica composta da una o piu' reti interconnesse mediante ripetitori, hub, bridge e switche, ma non IP router. Un IP router puo' connettere piu' sottoreti differenti

dati non riesce nel contempo ad ascoltare il canale per attivita', rendendo impossibile rilevare eventuali collisioni verificatesi.

A causa di questa eclusivita' del canale di trasmissione, le reti wireless soffrono anche del cosiddetto nodo nascosto.

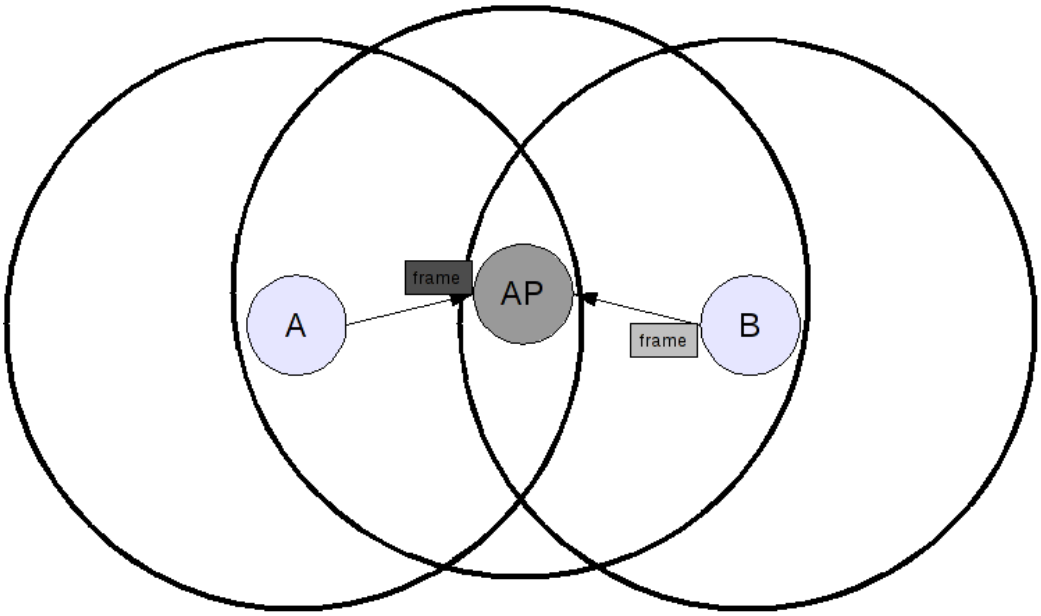

Figura 1.3: Nodo nascosto

In questo scenario A e B non sono rispettivamente a distanza di rilevamento l'uno dall'altro, ma sono connessi allo stesso access point. A trasmette verso l'ap e nel mentre anche B, siccome non ha rilevato traffico nel canale, comincia a trasmettere.Entrambe i pacchetti sono andati persi e A e B non hanno modo di rilevarlo.

Un meccanismo per incrementare le prestazioni di un sistema a canale condiviso e' stato implementato nel protocollo IEEE 802.11 RTS/CTS, il quale funge da semaforo per le trasmissioni fra i client e l'access point. L'ipotetico nodo A, prima di iniziare la trasmissione, invia all'access point un pacchetto RTS (Request to Send); quest'ultimo, se in grado di ricevere, risponde al nodo A con un pacchetto CTS (Clear to Send) e la trasmissione inizia. Questo risolve il problema del nodo nascosto in quanto un eventuale

nodo B, durante quanto descritto sopra, entra in possesso del pacchetto CTS non destinato a lui e non trasmette.

La dimensione dei pacchetti RTS/CTS varia da 0 a 2347 ottetti; se i dati da inviare hanno dimensione minore di 2347 l'invio del frame RTS non avviene e il pacchetto dati e' spedito immediatamente. Nel caso questa dimensione venga superata, viene utilizzato il meccanismo RTS/CTS.

Frame 802.11 Lo standard 802.11 definisce tre differenti tipi di frame: dati (DATA), controllo (CTRL) e manutenzione (MGMT); ognuno dei quali formato da svariati campi utilizzati dal sublayer per svolgere determinate funzioni.

Durante lo studio dei frame MAC 802.11 il primo campo preso in consi-

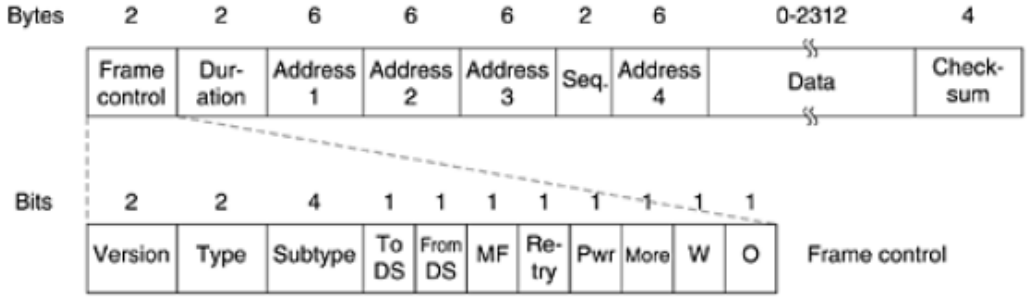

Figura 1.4: Frame 802.11

derazione e' stato il Frame Control, composto da 11 sottocampi, che contiene informazioni riguardanti il protocollo, il tipo (uno dei tre precedentemente citati), il sottotipo del frame (nel caso sia di controllo, per esempio un ACK), la presenza di frammentazione, crittografia e cosi' via.

Come si puo' notare, non e' presente nessun campo che permetta di ricondurre l'acknowledgement ricevuto al relativo pacchetto trasmesso.

1.4.3 Livello Network (livello 3 del modello OSI)

Questo livello si preoccupa di tutti i dettagli relativi al percorso nel trasferimento dei pacchetti dalla sorgente alla destinazione. La consegna

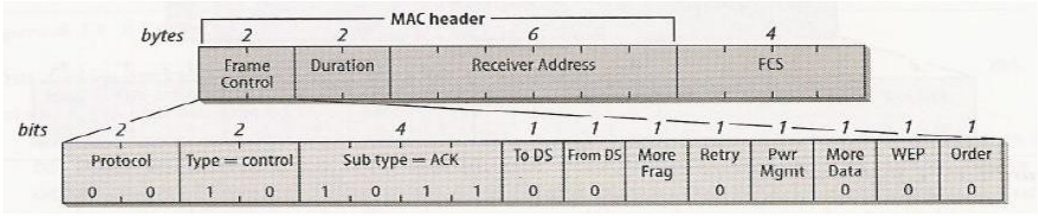

Figura 1.5: Struttura ACK Frame

deve avvenire indipendentemente dalla struttura della rete e dal numero di sottoreti presenti.

Va notato che questo e' l'ultimo livello in cui avviene logicamente una trasmissione end-to-end. E' dunque necessario che il layer conosca la topologia di reti e sottoreti per scegliere il giusto percorso attraverso esse, soprattutto nel caso in cui sorgente e destinazione si trovino in reti differenti. Inoltre, a questo livello vengono gestite le code e le congestioni.

Protocollo IPv4

Il protocollo IPv4 e' descritto nell'IETF RFC 791[9] pubblicato per la prima volta nel settembre 1981.

Questo protocollo data-oriented viene usato nelle reti a scambiamento di pacchetti (packet switching network), di tipo best-effort e non garantisce la consegna dei pacchetti a destinazione, non controlla né la correttezza dei dati né il verificarsi di ricezioni di pacchetti duplicati o in ordine di spedizione sparso.

Spazio degli indirizzi La versione 4 del protocollo IP gestisce uno spazio di indirizzi a 32 bit, rendendo possibile la creazione di solo 4.294.967.296 identificatori univoci. Alcuni di essi sono riservati, per esempio circa 18 milioni sono destinati alle reti private e circa 16 milioni sono indirizzi broadcast. Nonostante il numero apparentemente elevato, questo spazio di indirizzamento col passare del tempo si e' dimostrato insufficiente, anche se tecniche come

il NAT (Network address translation) hanno permesso di mitigare questo difetto, in attesa della migrazione al nuovo protocollo IPv6. La forma leggibile di un indirizzo IPv4 e' una quadrupla di numeri separati da un punto (192.168.10.1).

Formato del pacchetto IP Il protocollo IP prevede la seguente struttura per i datagram IP

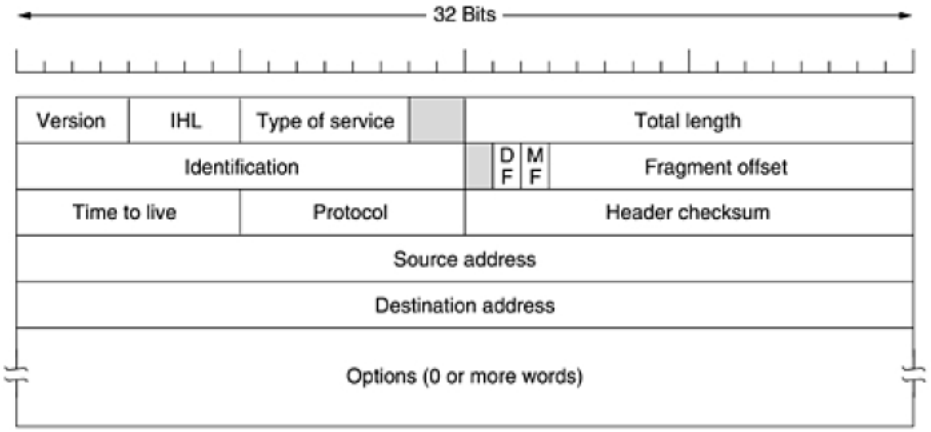

Figura 1.6: Header del pacchetto IP

Data Il campo data non fa parte dell'header IP, e non e' incluso nel checksum. Il suo contenuto e' specificato nel campo protocol e puo' essere uno qualsiasi dei protocolli del livello trasporto. I piu' comuni sono:

Frammentazione e riassemblaggio Ogni segmento logico, o fisico, di rete attraversato da un pacchetto ha un MTU (Maximum Trasmission Unit) che specifica la dimensione massima di un frame inviabile tramite quella rete.Tramite la frammentazione e' possibile dividere il payload del pacchetto in tanti pacchetti che verranno riassemblati una volta a destinazione. Per esempio la dimensione massima di un pacchetto IP e' di 65,535 bit, ma la dimensione tipica del MTU di una rete Ethernet e' di 1500 bit. Questo

Tabella 1.1: Codice protocolli Transport Level

Codice	Protocollo
1	Internet Control Message Protocol (ICMP)
2	Internet Group Management Protocol (IGMP)
6	Transmission Control Protocol (TCP)
17	User Datagram Protocol (UDP)
89	Open Shortest Path First (OSPF)
132	Stream Control Transmission Protocol (SCTP)

comporta che per inviare tutto il payload del pacchetto IP sono necessati 45 frame ethernet.

1.4.4 Livello Trasporto (livello 4 del modello OSI)

Il livello di trasporto permette il dialogo strutturato tra mittente e destinatario. I protocolli definiti permettono la comunicazione end-to-end[16].

Il principale protocollo a livello di trasporto attualmente in uso e' il TCP (Transmission Control Protocol) che implementa il concetto di connessione, permettendo il riassemblaggio dei pacchetti ed eventuale ritrasmissione di quelli persi, oltre alla gestione della congestione.

La ricerca in esame si e' concentrata sul protocollo UDP (User Datagram Protocol). Questo non e' un protocollo a trasmissione garantita e, a causa di questo, e' utilizzato nelle applicazioni che necessitano di interattivita' e velocita' a discapito della integrita' dei dati (la perdita di qualche frame di un video non e' fatale alla fruizione, mentre la fluidita' si)

[16]Per comunicazione end-to-end si intende lo scambio di informazioni tenendo in considerazione solamente sorgente e destinazione, senza prestare attenzione ai passaggi intermedi.

Protocollo UDP

L'User Datagram Protocol, definito nel 1980 da David P. Reed (RFC 768)[10], e' uno dei protocolli sui cui si basa l'architettura Internet. Le applicazioni possono creare e inviarsi dei messaggi brevi, conosciuti anche come datagram, attraverso degli appositi socket.

Il protocollo UDP non si basa sul concetto di connessione e i pacchetti possono essere ricevuti in ordine sparso, persi o duplicati.

Un esempio di applicazioni che utilizzano UDP sono Domain Name System (DNS), streaming media applications come IPTV, Voice over IP (VoIP) e Trivial File Transfer Protocol (TFTP).

UDP header I pacchetti UDP hanno un header strutturato come segue

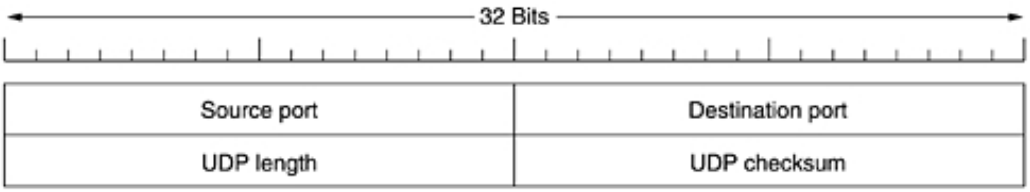

Figura 1.7: Header del pacchetto UDP

1.4.5 Livello Applicazione (livello 7 del modello OSI)

Al gradino piu' alto della pila protocollare del TCP/IP, come descritto precedentemente, si trova il livello applicazione, utilizzato dalle applicazioni utente astraendo su qualsiasi tipo di connessione o hardware sottostante. Classico esempio di un protocollo di questo livello e' l'HTTP (HyperText Transfer Protocol).

Protocollo VoIP

Con VoIP[11, 12] (Voice Over Internet Protocol) si intende il protocollo capace di gestire il passaggio da segnale audio (analogico) a flusso di pacchetti

(digitale) sulla rete e viceversa. Le risorse necessarie all'utilizzo di questo protocollo (una normale linea ADSL con almeno 32 kbps di banda) sono molto contenute rispetto alla qualita' della trasmissione, principale ragione della diffusione del mezzo agevolata anche dai costi particolarmente bassi.

Funzionamento Tipicamente il protocollo richiede due tipi di pacchetti: dati e controllo. I primi trasportano fisicamente i frame audio compressi, i secondi ne controllano ordinamento e codifica.

Esistono diversi protocolli standard:

- SIP (Session Initiation Protocol) della IETF;
- H.323 della ITU;
- Skinny Client Control Protocol proprietario della Cisco;
- Megaco (conosciuto anche come H.248);
- MGCP;
- MiNET proprietario della Mitel;
- IAX, usato dai server Asterisk open source PBX e dai relativi software client.

Vantaggi I benefici dell'utilizzo di una tecnologia quale il VoIP sono molteplici.

- Abbattimento dei costi
  - A causa della localita' della connessione ad internet, utilizzando sistemi VoIP il costo della telefonata e' solitamente quello dal piu' vicino POP[17] alla destinazione, tipicamente una chiamata urbana.

[17]Point Of Presence: il piu' vicino provider di connettivita'

  - All'interno di una azienda, anche in presenza di sedi dislocate, il VoIP consente una notevole liberta' di comunicazione, rendendo possibili sia telefonate all'interno della stessa sede che verso altre e teleconferenze completamente gratuite (l'unico costo effettivo e' quello per la connettivita').

- Servizi telematici

  - A causa della differenziazione dei possibili tipi di dato che sono trasmissibili su tecnologia IP e' possibile associare a un flusso audio anche altri servizi, ad esempio di elenco telefonico o centralino vocale.

- Number Portability

  - Come per i telefoni cellulari, e' possibile spostare virtualmente un numero di telefono fisso di rete analogica in ogni parte del mondo coperta da qualche tipo di connettivita' a pacchetti.

- Maggiore Privacy

  - La struttura a packet switching della rete Internet su cui solitamente si appoggia il VoIP rende molto piu' complesso accedere a tutti i dispositivi che si interpongono fra i due capi della comunicazione, aumentando la confidenzialita'. La possibilita' di aggiungere cifratura - a livello di protocollo VoIP o di canale di trasmissione - garantisce anche integrita' dei dati e privacy.

- Geolocalizzazione

  - Essendo la comunicazione VoIP basata su reti fisiche e' possibile localizzare geograficamente l'utente in base all'indirizzo di provenienza, rendendo piu' semplici operazioni vitali ma geograficamente localizzate come il pubblico soccorso.

Svantaggi Come tutte le tecnologie il VoIP presenta diversi svantaggi.

- Qualita' sonora
  - Nonostante la maggior parte dei protocolli VoIP commercialmente diffusi implementi tecniche volte a mitigare latenze nella ricezione dei pacchetti, pause nella conversazione, disturbi sulla linea o congestione, la comunicazione analogica e' ancora superiore sia in qualita' che in affidabilita', considerando che una perdita di pacchetti superiore al 10% rende impossibile una conversazione VoIP.
- Bandwidth
  - Le sessioni VoIP che si appoggiano alla rete locale di solito beneficiano dell'ampia larghezza di banda disponibile e della scarsita' di errori tipiche di questo mezzo, mentre le comunicazioni a distanza, che quindi utilizzano la rete Internet, solitamente soffrono molto dei problemi sopra descritti.Una soluzione commerciale solitamente offerta dai provider e' la cosidetta "banda garantita"
- Latenza
  - Una conversazione telefonica - per l'utente omologa a una tramite telefonia digitale - e' composta per oltre il 50% di pause e disturbi che spesso aggiungono significato alle parole, motivo per il quale il VoIP risente in maniera fondamentale della latenza, che deve essere tassativamente mantenuta sotto i 150 millisecondi.

Il nostro contesto nel VoIP Lo scenario applicativo da noi preso in considerazione vede utilizzare il VoIP in un contesto wireless, motivo per cui le problematiche precedentemente delineate sono rese ulteriormente importanti a causa degli ulteriori difetti del wireless.

1.5 Crittografia

Sicurezza nelle reti wireless La sicurezza in ambito wireless ha come scopo la prevenzione della connessione alla rete da parte di client non autorizzati e dell'intercettazione o la modifica dei dati in transito da parte di utenti malevoli

E' disponibile un' approfondita letteratura sulle vulnerabilita' di protocolli e algoritmi di cifratura, ma solo ultimamente si sta considerando anche l'essere umano che utilizza il dispositivo come fonte di insicurezza.

Un eventuale utente malintenzionato dispone di svariati modi per attaccare una rete wireless, la cui trattazione esula lo scopo di questo testo, ma esistono svariate tecniche volte a minimizzare i danni, come l'utilizzo di sistemi di sicurezza basati sul livello 3 dello stack, quali VPN (Virtual Private Networks) o reti autenticate (Kerberos o Radius), che tuttavia risentono di attacchi fisici del livello di trasporto (Denial Of Service).

Esistono molteplici modi per affrontare il problema delle intrusioni, ma le reti assolutamente sicure sono un'utopia. A causa della complessita' e varieta' dei sistemi connessi tramite le reti, e' necessario che ogni anello di questa catena di sicurezza sia affidabile e protetto.

Sono citate per completezza alcune delle tecniche piu' diffuse; si noti bene che queste sono efficaci solo se utilizzate combinatamente tra loro.

- Filtraggio del MAC ID
- Disabilitazione dell'assegnamento automatico degli indirizzi IP (DHCP)[18]
- Occultamento dello SSID
- Crittografia
  - WEP (Wired Equivalent Privacy)
    - Utilizza HMAC-RC4 come algoritmo di cifratura e HMAC-CRC32 per garantire l'integrita' dei dati

[18]Dynamic Host Configuration Protocol

    * Dimensione della chiave fissa (diverse lunghezze supportate a seconda del produttore)
    * Algoritmo obsoleto, ne e' stata dimostrata l'insicurezza durante gli anni ed esistono svariati tool, commerciali o meno, in grado di forzarne la sicurezza in pochi minuti.
    * Autenticazione basata su Shared Key, espone la chiave primaria a intercettazione durante l'handshake

- WPA/WPA2 (Wi-Fi Protected Access)
    * Utilizza HMAC-RC4 o CCMP come algoritmi di cifratura (rispettivamente per WPA e WPA2) e HMAC-MD5 o HMAC-SHA1 per l'integrita'.
    * Il miglioramento piu' rilevante rispetto a WEP e' stata l'introduzione del TKIP (Temporary Key Integrity Protocol) che permette di non scambiare la chiave privata durante l'autenticazione ma solo una chiave temporanea calcolata a partire dalla chiave segreta, dal MAC dei dispositivi e da alcuni numeri utilizzati una sola volta e generati casualmente dal client e dall'access point. Nella trasmissione effettiva dei dati questi sono cifrati con la chiave temporanea che, se cambiata abbastanza spesso, rende impraticabile un eventuale attacco.

Capitolo 2

Obiettivi e sperimentazione

2.1 Obiettivi

A causa delle limitazioni descritte in precedenza, la tecnologia wireless non permette di effettuare alcun controllo di flusso sulla trasmissione. I protocolli di alto livello che si appoggiano su TCP mediano questo problema delegando il controllo della connessione ai livelli sottostanti, ma in uno scenario di connettivita' VoIP, che utilizza il protocollo UDP, questo non e' possibile. Lo studio si pone come obiettivo il riportare in userspace informazioni basilari riguardanti la trasmissione dei frame 802.11 per garantire un controllo di flusso fino al prime hop[1] della comunicazione (Access Point).

2.1.1 Cenni Storici

Il lavoro e' cominciato come progetto per il corso di Laboratorio di Programmazione di Rete ed e' continuato nel Tirocinio svolto presso la Facolta' di Informatica dell'ateneo di Bologna. Il gruppo, formato da quattro persone, si e' diviso il lavoro concentrandosi da un lato sulla metodologia per reperire le informazioni riguardo i pacchetti (svolto dai Dottori Codeca' Lara e Grassi Giulio), dall'altro sul sistema di registrazione dell' applicazione e

[1]Prime Hop, si intende il primo nodo della comunicazione

ricezione dei dati raffinati (svolto da Bigoi Matteo e Lusuardi Andrea). Durante il lavoro, mentre il sistema di comunicazione verso l'userspace era in fase di implementazione, e' stato utilizzato un metodo alternativo per avere rapidamente le informazioni verso lo userspace che verra' dettagliato in seguito. Con la laurea dei due colleghi, si e' continuata la ricerca utilizzando il sistema di comunicazione basato su SysFS, anch'esso dettagliato piú avanti.

2.1.2 Ack dal prime hop e interattivita'

L'obiettivo principale del progetto e' la creazione di un protocollo cross layer che si interponga fra lo stack di rete mac80211 e lo userspace, fornendo informazioni formattate ad un monitor; questo applicativo si occupa di gestire l'eventuale ritrasmissione di pacchetti, il cambio di portante a causa di latenza o congestione e l'avvicendarsi dei programmi utente che utilizzano questo protocollo. Utilizzando UDP over IP su connettivita' wireless e' possibile ottenere informazioni solo riguardo il primo hop della comunicazione, solitamente il passaggio del pacchetto all'access point. Siccome spesso l'access point e' connesso direttamente alla rete cablata, la perdita di pacchetti da quel punto in poi e' molto improbabile. Il meccanismo cross-layer implementato, che si appoggia sul lavoro gia svolto dai Dottori Codeca' Lara e Grassi Giulio, mira a fornire un conveniente sistema userspace per la registrazione e la manipolazione dei dati di connessione.

Sono state prese in considerazione alcune problematiche, quali:

- Stabilire a quale/i pacchetto/i IP e' associato un dato pacchetto UDP.
- Stabilire a quale/i frame MAC80211 e' associato un dato pacchetto IP.
- Stabilire quale frame MAC80211 e' associato a un eventuale ACK ricevuto dall'access point.
- Individuare un metodo efficiente per trasmettere informazioni sull'acknowledgement in userspace

- Implementare un sistema di registrazione per le applicazioni ad alto livello

Il primo approccio utilizzato dai nostri colleghi e' stato sfruttare il canale di errore dei socket, associando a livello MAC il frame con le informazioni di acknowlegdment e retry count, mentre a livello network si e' associato l'id del pacchetto IP al frame in cui esso e' incapsulato. Il punto finale e' stato la modifica delle syscall di apertura e configurazione dei socket nel kernel di Linux, in particolare la funzione

```
setsockopts()
```

per poter esporre le informazioni riguardanti identificatori dei pacchetti e loro sorte tramite il canale di errore dei socket. Un' applicazione, per poter utilizzare queste proprieta', doveva essere compilata utilizzando una versione appositamente modificata delle DietLibC [2] linkata staticamente nel programma oggetto.

La continuazione di questo progetto prevede la creazione di un'interfaccia utente basata su SysFS[14]. Questa nuova implementazione non richiede l'utilizzo di syscall modificate o di versioni speciali di librerie, prevede soltanto la registrazione da parte dell'applicazione tramite alcune librerie utente appositamente create.

2.2 Sperimentazione

Dato il gran numero di dispositivi wireless presenti sul mercato, la ricerca si e' concentrata su quelli supportati dal kernel di Linux, il cui driver fosse basato sul nuovo stack mac80211. La decisione di lavorare a livello di stack di rete ha permesso di portare il codice fra versioni diverse del kernel di Linux e ha garantito la compatibilita' con qualunque dispositivo wireless supportato che utilizzi quello stack, a parte rare eccezioni. I driver e i dispositivi

[2] Progetto che mira a ottimizzare le libc per renderle piu' contenute e leggere. http://www.fefe.de/dietlibc/

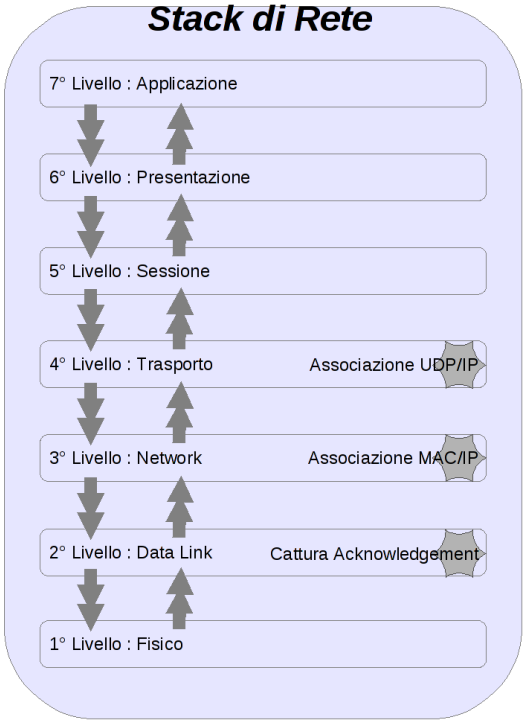

Figura 2.1: Modifiche cross-layer

wireless presi in esame sono stati un numero cospicuo. Segue un elenco dei soli dispositivi il cui driver non sia stato scartato per motivi triviali all'atto della sperimentazione.

- Ralink RT2570usb: uno dei primi device valutati; presentava notevoli problemi di associazione e conseguente trasferimento dati e il supporto del driver al nuovo stack era frammentario e incompleto.

- Realtek RTL8187usb: questo device e' basato in modo cosi' preponderante sul firmware da obbligare gli sviluppatori del driver libero a introdurre diversi workaround per garantirne il funzionamento, uno dei quali prevedeva l'acknowledgment forzato di tutti i frame, in quanto il firmware non ne dava conferma alcuna, per permettere ai protocolli di livello superiore di funzionare correttamente. Essendo tutto il progetto basato sull'avvenuto acknowledgment dei frame, il device in questione e' stato abbandonato rapidamente.

- Intel Pro Wireless 3945abg: schede miniPCI-X che montano questo chipset sono comunemente presenti in quasi tutti i laptop basati su piattaforma Intel, nonostante la scheda sia ormai stata sostituita dalle piu' moderne serie 49xx e 5xxx. La scheda dialoga con il kernel attraverso un driver che si pone come interfaccia comandi tra firmware e stack di rete. Questo e' il primo device con cui sono stati ottenuti risultati sensibili; inoltre lo studio di questo driver ha permesso di approfondire la conoscenza del funzionamento interno del kernel permettendo la migrazione del codice dal livello driver a quello superiore di stack di rete.

- Zydas ZD1211rw: una volta ottenuta la migrazione allo stack di rete, si e' scelto come hardware un dispositivo facile da reperire, poco costoso e ben supportato che ha trovato nello ZD1211 un ottimo candidato. Il driver per questi dispositivi era disponibile sia integrato con il vecchio stack di rete softmac sia in una versione nuova in fase di adattamento per supportare il nuovo mac80211. L'utilizzo di questa WNIC USB

ha permesso di ottenere il giusto compromesso fra informazioni filtrate dal firmware e necessita' progettuale di utilizzare un dispositivo USB. A tal riguardo, e' doveroso aggiungere che, dato l'ambito esclusivamente mobile in cui si colloca questo progetto, difficilmente un client mobile potra' avere connettivita' diversa da quella fornita dai device wireless USB.

Tutto il codice e' stato man mano portato alle nuove versioni del kernel di Linux rilasciate in quel periodo, assestandosi su un kernel versione 2.6.25.6 al momento della scrittura.

Capitolo 3

Progettazione

Nel seguente capitolo verranno approfonditi gli aspetti concettuali e progettuali del meccanismo cross layer, riassumendo il lavoro svolto dai nostri colleghi, mentre nel capitolo successivo verra' dettagliata l'attuale implementazione basata su SysFS.

3.1 Meccanismo Cross-Layer

Al fine di poter progettare un protocollo cross layer e' stato necessario studiare il funzionamento dei diversi livelli di astrazione del networking del kernel di Linux. Seguendo una ideale mappa del percorso dei dati dal momento in cui questi sono inviati dall'applicazione, ad esempio scrivendoli su un socket, essi vengono incapsulati in un frame UDP completo delle informazioni di gestione, fra cui il datagram id e la porta sorgente, passato al livello sottostante, quello IP. Qui avviene un secondo incapsulamento, proprio dell'UDP over IP, che inserisce il pacchetto UDP in un frame IP formattato correttamente identificato da un ip_id [1] e lo invia al livello successivo, quello dello stack di rete mac80211. Quest'ultimo livello e' direttamente connesso con il driver (e quindi indirettamente con il firmware di cui si discutera' in seguito) e si occupa di incapsulare il pacchetto IP (che ricordiamo avere al

[1] Numero sequenziale temporalmente univoco del pacchetto IP

suo interno quello UDP) all'interno di un frame IEEE802.11. Questo frame e' l'ultimo punto di accesso ai dati consentito dal kernel prima del passaggio a un livello troppo basso, come il firmware. Il meccanismo cross layer proposto permette di effettuare in modo efficiente e trasparente all'utente il passaggio di informazioni sulla sorte del pacchetto UDP senza che il protocollo utilizzato necessiti controlli di connessione. Associando i dai ricavati dai frame nei tre livelli si e' realizzato l'effettivo passaggio di informazioni tra layer diversi. Tecnicamente questo e' possibile in quanto al livello piu' basso raggiungibile, quello dello stack IEEE802.11, il frame contiene gia' tutti gli header dei pacchetti incapsulati di livello superiore, estraibili grazie alle librerie di sistema.

3.2 Implementazione basata sui socket

3.2.1 Hardware utilizzato

Al fine di comprendere al meglio il funzionamento della pila protocollare nella trasmissione delle informazioni, e' stato necessario studiare e sperimentare con un driver noto prima di potersi muovere ad alto livello nello stack di rete. E' stata scelta la scheda Intel PRO/Wireless 3945ABG, comune dispositivo presente sulla maggior parte dei portatili di piattaforma Intel con una banda di trasferimento massima di 54 Mbit.

3.2.2 Interfaccia firmware-driver

La scheda presa in considerazione e' gestita a basso livello da un firmware fornito dalla casa madre, non modificabile, che espone alcune funzioni verso il driver del kernel, inserendosi come livello di astrazione ulteriore fra hardware e driver di periferica.

A causa della complessa interazione fra firmware e driver, per poter ottenere informazioni utili e' stato necessario intercettare il frame al punto piu' basso della catena, e quindi piu' brenso di informazioni, prima di essere pas-

sato all'hardware (dal nostro punto di vista il firmware) e spedito fisicamente. Valori codificati in alcune variabili (principalmente in `int_fh`) consentono di capire il tipo di pacchetto e la direzione del flusso.

Il driver espone sei code per i pacchetti in trasmissione e una per quelli in ricezione, rendendo difficile associare il pacchetto alla relativa coda di trasmissione e impedendo, di conseguenza, di ottenere informazioni utili.

Ricezione

Durante lo studio, sono state individuate svariate funzioni fondamentali che si occupano di configurare il canale di ricezione e processare il pacchetto; elenchiamo di seguito le principali:

- **iwl_pci_probe**: Funzione che inizializza molti parametri collegati con il livello fisico con i quali popola due strutture, *iwl_priv* e *ieee80211_hw*[2].
- **ieee80211_alloc_hw** [3]: In questa funzione vengono inizializzati alcuni campi necessari alla gestione del livello fisico della comunicazione e viene allocata la memoria per la struttura ieee80211_hw. A questo livello, esiste inoltre una struttura *ieee80211_local* contenente i campi di interesse riguardo la ricezione. Agendo nella logica operativa di questa funzione si attua la separazione fra driver di dispositivo e stack di rete, unificando le interfacce comuni che definiscono determinate operazioni.
- **iwl_rx_handle**[4]: Questa funzione, invocata quando si riceve un frame, seleziona il gestore corretto per l'evento che ne ha scatenato l'invocazione. Per poter capire meglio come si comporta occorre conoscere due elementi molto importanti del driver, la gestione tramite comandi e le code in trasmissione e ricezione.
- **iwl_rx_reply_tx**: Funzione che gestisce la ricezione degli acknowledgement; opera su aree di memoria dette Socket Buffer (skb). Queste

[2]Per la definizione completa vedere linux-src/drivers/net/wireless/iwlwifi/iwl-priv.h
[3]Funzione definita in linux-src/net/mac80211/ieee80211.c
[4]Definita in linux-src/drivers/net/wireless/iwlwifi/iwl3945-base.c

strutture contengono numerosi dati, i piu' significativi sono *txq_id* e *index*, rispettivamente identificatore della coda di trasmissione sulla quale e' stato processato il pacchetto che ha appena ricevuto un acknowledgement e indice interno della stessa. Le funzioni interne del driver processano poi lo skb di cui sopra e estraggono informazioni importanti sul frame appena inviato, quali il retry count, la presenza o meno di acknowledgement e altre flags[5] per la gestione del pacchetto.

- iwl_tx_queue_reclaim [6]: Questa funzione di housekeeping controlla l'acknowledgement (o il timeout) dei pacchetti presenti sulle code di trasmissione e, prima di deallocarli, ne passa le informazioni allo stack, tramite la funzione *iwl_txstatus_to_ieee*. Una volta liberato abbastanza spazio sulle code in trasmissione vengono sbloccati eventuali frame in attesa.

- ieee80211_tasklet_handler [7]: Quando questa funzione e' invocata, a livello logico dentro lo stack di rete, discrimina i casi specifici di confermata ricezione o timeout di ogni pacchetto, invocando le dovute callback sia verso il driver che verso altre funzioni dello stack mac80211.

- ieee80211_tx_status : [8]Questa funzione interpreta le informazioni provvedute dal driver e risponde correttamente alle richieste dello stack di rete.

- __ieee80211_rx[9] Funzione invocata alla ricezione di un frame dati, invoca gestori specifici, ma, essendo esterna al percorso degli acknoledgement

[5]Con Flag Bandiera si indica una variabile binaria che codifica la presenza di una informazione.

[6]Implementata in *linux-src/drivers/net/wireless/iwlwifi/iwl3945-base.c*

[7]Implementata in *linux-src/net/mac80211/ieee80211.c*

[8]Implementata in *linux-src/net/mac80211/ieee80211.c*

[9]Implementata in *linux-src/net/mac80211/rx.c*. La particolare intestazione della funzione __ indica che essa e' da considerarsi "interna" ai meccanismi del kernel, ovvero il suo funzionamento o la stabilita' dell'interfaccia possono cambiare senza preavviso. Questo e' un comportamento mediato dalla programmazione ad oggetti, affidando pero' il rispetto delle convenzioni nelle mani del programmatore e non a vincoli interni al linguaggio.

non viene approfondita.

Trasmissione

La trasmissione dei pacchetti e' divisa in due fasi, una di competenza dello stack e l'altra del driver. Riportiamo di seguito le funzioni piu' importanti incontrate durante lo studio, a livello di driver e di stack, rispettivamente.

- iwl_mac_tx [10]: Questa e' la funzione, richiamata dallo stack di rete, dove avviene effettivamente il passaggio fra le due fasi del meccanismo di trasmissione e, effettuati i dovuti controlli, invocata la *iwl_tx_skb*.

- iwl_tx_skb [11]: Una volta invocata, questa funzione associa al frame da inviare l'area di memoria necessaria a segnalarne l'acknowledgement, oltre alla costruzione e impostazione dello skb relativo al frame e al suo header.

A livello di stack di rete, invece, le funzioni principali sono le seguenti:

- ieee80211_tx [12]: Viene invocata ogni qualvolta ci sia un frame da inviare, richiamando i relativi genitori.

- ieee80211_tx_h_sequence[13]: Questa funzione si limita a incrementare il sequence number del pacchetto poco prima della sua trasmissione.

Comandi Ad ogni evento il driver associa uno specifico comando, al cui interno sono contenute le informazione necessarie per la corretta gestione dell'evento, nei campi `cmd`[14] e `hdr`. Le schede come quella in esame sono molto complesse e siccome un elenco dettagliato dei comandi sarebbe piuttosto dispersivo e poco utile, si riportano solo le categorie in cui esse si dividono.

[10] Implementata in *linux-src/drivers/net/wireless/iwlwifi/iwl3945.c*
[11] Implementata in *linux-src/drivers/net/wireless/iwlwifi/iwl3945.c*
[12] Implementata in *linux-src/net/mac80211/tx.c*
[13] Implementata in *linux-src/net/mac80211/tx.c*
[14] I possibili comandi della scheda sono definiti in *iwlwifi iwlcommands.h*

- Comandi RXON e QOS
- Supporto Multi-Station
- Comandi per attivazione elettrica di RX, TX, LEDs
- Comandi relativi 802.11h
- Gestione del Power Management
- Notifiche e comandi relativi alla scansione della rete
- Comandi IBSS/AP
- Configurazione BT
- Notifiche e comandi RF-KILL
- Notifiche Beacon persi

Code in trasmissione e ricezione

Lo studio della funzione *iwl_rx_handle* ha messo in luce i meccanismi di gestione dell'acknowledgement di un pacchetto da parte del kernel. Questa estrae il Socket Buffer scorrendo le varie code di trasmissione e utilizza i comandi precedentemente salvati per reperire il pacchetto di cui e' stato ricevuto l'acknowledgement. E' da tenere presente che nel precedente stack wireless (ieee80211_generic) la struttura Socket Buffer conteneva direttamente i dati relativi all'header, mentre con il nuovo stack contiene semplicemente un puntatore all'header corrispondente al livello di astrazione a cui si sta operando. In particolare nello skb si trova un campo che permette di risalire al tipo di comando e quindi invocare il gestore adatto; nel nostro caso siamo interessati al comando *REPLY_TX* e al suo gestore *iwl_rx_reply_tx*, invocati solamente in presenza di un acknowledgement.

3.2.3 Canale di ritorno di errore dei socket

La prima parte del progetto che utilizza questo sistema di notifica userspace e' riassunto nelle parti fondamentali nelle pagine che seguono, tuttavia per un maggior approfondimento e' necessario fare riferimento al documento di tesi dei Dottori Codecà Lara e Grassi Giulio.

Funzionamento dell'applicazione

La figura 3.1 illustra il funzionamento del framework utilizzando il canale di ritorno di errore del socket.

Descrizione del funzionamento

Ogni volta che un'applicazione invia un datagram IP, questa viene notificata del suo indentificativo e vengono estratte alcune informazioni utili:

- Identificativo del datagram IP.
- Porta sorgente.
- Il campo more_fragment dell'header IEEE802.11 indica se il frame in questione e' seguito da altri che portano al loro interno frammenti dello stesso pacchetto ip.
- Identificatore del frame IEEE802.11, usato anche per distinguere i dati appena estratti.

Le informazioni estratte possono aiutare a capire se un dato pacchetto ha ricevuto o meno il relativo acknowledgement. La funzione ieee80211_tx_status si occupa appunto del riconoscimento dell'eventuale conferma di ricezione per ogni pacchetto inviato, aggiornando una struttura in memoria che contiene anche altre informazioni, quali il retry_count e il filtered_count, rispettivamente il numero di volte in cui un pacchetto e' stato rispedito e il numero di volte in cui esso ha atteso la fine di un'altra operazione. Grazie a queste informazioni si collegano i dati estratti al momento della trasmissione con quelli

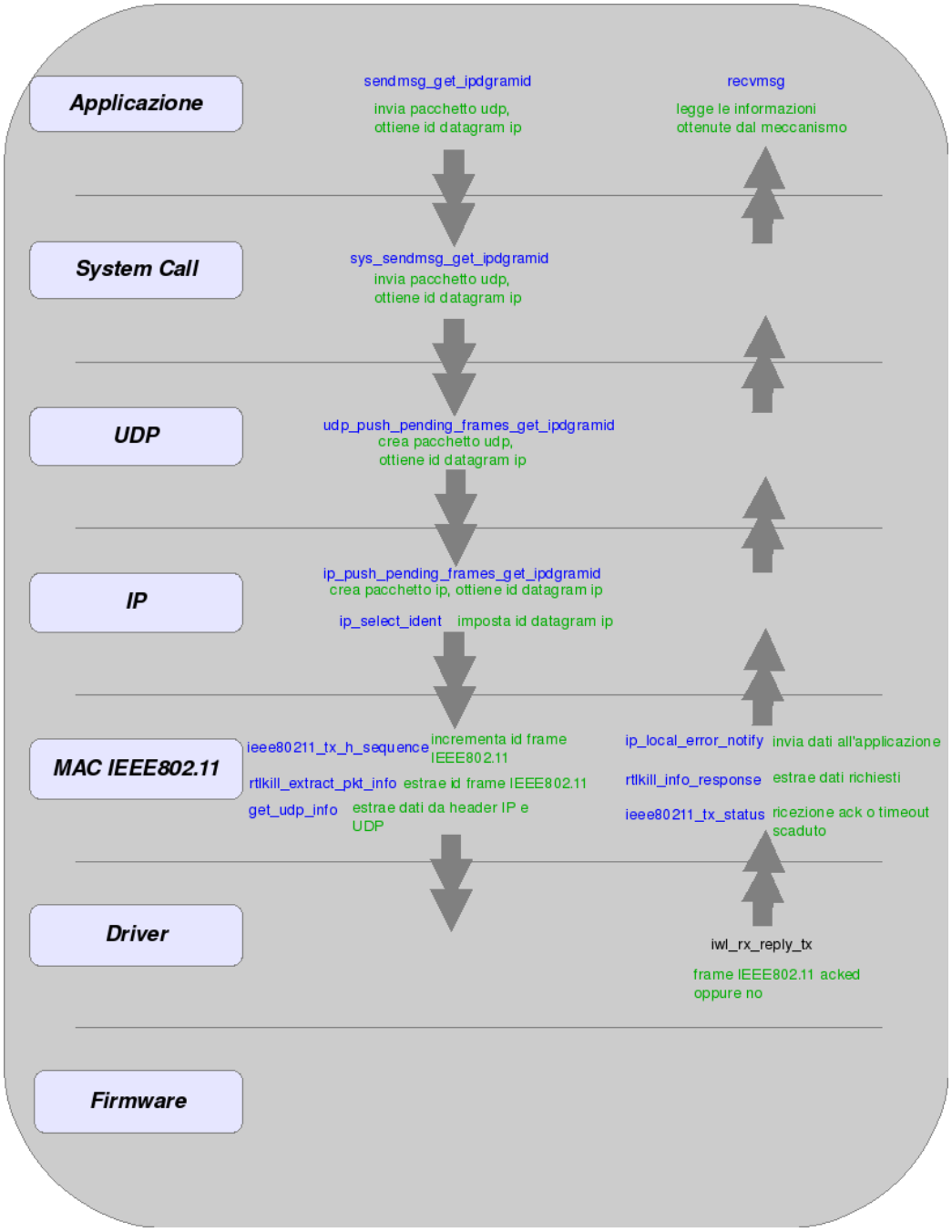

Figura 3.1: Panoramica meccanismo Syscall

sul datagramma IP, permettendo di capire se un determinato pacchetto UDP ha ricevuto ack o meno dall'access point.

Livello udp-ip

Ogni pacchetto UDP in transito viene identificato tramite il datagram IP che lo trasporta, associando i due identificativi. L'importante funzione *ip_select_ident* si occupa di selezionare l'id corretto al momento dell'incapsulamento di un pacchetto. A causa della frammentazione, e' possibile che un pacchetto IP contenga parti di pacchetti di livelli superiori, di conseguenza e' stato necessario modificare la funzione sopracitata - che in caso di frammentazione imposta l'identificatore a zero - in modo da assegnare un identificatore univoco in tutti i casi, permettendo quindi di mantenere l'associazione fra pacchetto IP e frame IEEE80211. Per quanto riguarda invece il livello UDP e' stata modificata la funzione *udp_push_pending_frames* in modo da passare come argomento un campo *__be16 ipdgramid* [15] nel quale viene salvato l'identificativo IP che comincia qui la sua "salita" verso lo userspace.

IEEE802.11

Verranno ora analizzate le modifiche principali svolte a livello stack di rete, introducendo quindi strutture dati e funzioni, ricordando al lettore che si tratta di un riassunto di un documento di tesi gia' citato in precedenza.

La struttura rtlkill_info Al fine di poter memorizzare le informazioni sui frame trasmessi in attesa di avere una risposta dall'access point e' stata creata la struttura di appoggio *rtlkill_info* [16] che contiene campi come l'id del frame IEEE80211 (usato anche per distinguere le varie istanze della struttura fra di loro), il tipo del pacchetto, alcuni timestamp di invio e ricezione

[15] Con __be16 si intende una definizione interna al kernel di un'area di memoria Big Endian di 16 bit

[16] Il nome rtlkill - abbastanza ricorrente nel progetto - deriva dal nome del primo device analizzato, il Realtek 8187 USB (per il quale i candidati contavano di scrivere un driver) e precisamente dal numero di device che si ipotizzava di distruggere durante il lavoro.

dell'ack usati per scopi statistici e un puntatore alla seconda struttura di appoggio. All'interno di *rtlkill_info* possiamo trovare l'importante struttura *packet*, contenente informazioni essenziali per la nuova implementazione del meccanismo cross layer basato su SysFS che verra' in seguito dettagliato.

Fra tutte le funzioni aggiuntive implementate le due piu' importanti sono la *rtlkill_extract_pkt_info* e la *get_udp_info*. La prima riceve in ingresso lo header del frame IEEE80211 e ne estrae le informazioni a cui si e' accennato in precedenza, quali identificativi IP e timestamps. La seconda, invece, analizza gli header dei pacchetti IP e UDP estraendone le seguenti informazioni che vengono salvate nella struttura *rtlkill_info*.

- Indirizzo sorgente (header ip)
- Indirizzo destinazione (header ip)
- Porta sorgente (header udp)
- Porta destinazione (header udp)
- Identificativo datagram ip (header ip)
- Dimensione dati trasportati (header ip)
- Byte indicante la presenza di altri frammenti ip dopo quello attuale
- Offset del campo data dal primo datagram ip della sequenza

Ricezione in dettaglio

Alcune funzioni, fra cui *ieee80211_tx_status* e *ip_local_error_notify*, sono state modificate al fine di invocare al momento giusto la funzione aggiunta *rtlkill_info_response* che estrae le informazioni di interesse dall'header del pacchetto che riceve come parametro. Leggermente piu' in dettaglio, nella funzione *ieee80211_tx_status* e' stato aggiunto un controllo che si basa sull'invocazione di *required_ip_local_error_notify* e che, in caso di esito positivo, indica che sul socket utilizzato e' stato richiesto l'utilizzo del canale di ritorno degli

errori. La funzione *rtlkill_info_response* estrae dall'header IEEE802.11 e dal parametro ieee80211_tx_status le informazioni sul retry_count, sull'avvenuto acknowledgement del pacchetto e sul sequence number (utilizzato per identificare il frame corretto nella lista di strutture rtlkill_info). Questi dati vengono uniti a quelli recuperati dalla struttura rtlkill_info e spediti al mittente del frame utilizzando la *ip_local_error_notify*, mentre se e' presente la flag DEBUG a tempo di compilazione viene invocata anche la funzione *rtlkill_info_take_response* che stampa sul log di sistema. La funzione *ip_local_error_notify* salva i dati nella memoria del socket dedicata alla gestione della coda di ritorno di errore, fra cui

- success: Intero che puo' assumere i seguenti valori:
  - 0 se non si e' ricevuto l'ack del frame inviato
  - 1 se l'ack e' stato ricevuto
  - 2 se non e' stato superato il limite di tentativi di spedizione indicati in execessive_retry
- id: del datagram ip
- more_segment: flag che indica se il pacchetto e' stato diviso in frammenti o meno e se quello attuale e' l'ultimo.

Interfaccia UserSpace

Un'applicazione che voglia utilizzare il servizio di tracciamento della conferma di ricezione dei propri pacchetti UDP inviati da parte del primo Hop di rete (in questo caso un'access point) deve seguire un certo protocollo. All'apertura del socket UDP deve settare l'opzione che consente di sfruttare il canale di ritorno degli errori dello stesso:

```
int OptVal = 1;
ris = setsockopt( socketfd, IPPROTO_IP, IP_RECVERR,
                  (char *)&OptVal, sizeof(OptVal) );
```

e leggere dalla coda di ritorno di errore i messaggi inviati dal kernel tramite una apposita read che tiene conto dello spazio utente dove memorizzare questi dati.

Syscall

L'aggiunta della syscall al kernel deve essere notificata alle librerie userspace con un'opportuna modifica delle stesse, che, ricordiamo, sono le DietLibC di cui si e' parlato nel capitolo due. Questa nuova syscall e' definita come segue:

```
#define SYS_SENDMSG_GET_IPDGRAMID          18;
        ...
__socketcall(sendmsg_get_ipdgramid,SENDMSG_GET_IPDGRAMID)
```

La funzione *__socketcall* e' un punto di ingresso al kernel piuttosto comune, viene solitamente invocata in corrispondenza di una system call e riceve infatti come parametro il nome della funzione che effettivamente verra' eseguita, in questo caso la *sendmsg_get_ipdgramid*:

```
asmlinkage long sys_sendmsg_get_ipdgramid(int fd,
                struct msghdr __user *msg,
                unsigned flags, __be16 *p_ipdgramid);
```

Una volta che queste inizializzazioni sono completate, ogni volta che un'applicazione spedisce un pacchetto UDP ne puo' leggere le informazioni riguardanti la ricezione tramite il canale di ritorno di errore del socket su cui sta spedendo, in quanto le system call sottostanti riportano le riportano al livello piu' alto tramite le callback della gestione dei socket..

La lettura del canale di errore del socket viene effettuata tramite la chiamata a

```
recvmsg(socketfd, msg, MSG_ERRQUEUE);
```

passando come parametri il descrittore del socket in uso, un'area di memoria di tipo msghdr che conterra' le informazioni e la coda di messaggi che si vuole leggere, in questo caso MSG_ERRQUEUE. Sono state inoltre fornite altre funzioni `CMSG_FIRSTHDR` e `CMSG_NXTHDR` che estraggono dal msghdr sopracitato le informazioni di supporto[17] dalle quali si accede, tramite la fun-

[17] Ancillary data

zione `CMSG_DATA`, alla struttura dove risiedono le informazioni sulla sorte del pacchetto UDP.

3.2.4 Test e valutazione

Il meccanismo appena descritto e' stato collaudato nella sua interezza utilizzando svariate metriche le cui valutazioni sono disponibili nella tesi di laurea dei Dottori Codeca' Lara e Grassi Giulio, corredate da grafici e tabelle riassuntive.

Capitolo 4

Implementazione basata su SysFS

Il lavoro svolto con il precedente meccanismo basato sul canale di ritorno di errore dei socket ha messo in luce alcuni problemi teorici e pratici, quali l'implementazione tramite modifica della tabella delle system call. L'aggiunta di una system call al kernel Linux prevede l'inserimento dell'indentificatore della stessa sia in una tabella centrale, che dalla versione 2.6.24 del kernel di Linux non e' piu' accessibile per convenzione, che nelle librerie di sistema, creando un certo numero di problemi. Lato kernel, e' necessario ottenere l'identificativo univoco per la nuova system call dalla comunita' che sviluppa il kernel, lato utente, rende necessarie modifiche al codice degli applicativi e alle librerie di sistema.

Il secondo approccio vede modifiche notevoli alla struttura riportata sopra, principalmente a causa di notevoli problemi teorici e pratici che sono venuti alla luce durante la scrittura del codice. E' stato implementato il seguente meccanismo:

Lo stack wireless raccoglie le informazioni sui pacchetti udp in transito sulle varie interfacce wireless e tramite una funzione esposta dal modulo ne aggiorna le tabelle ogni volta che viene ricevuto un pacchetto di interesse. Il kernel module prodotto crea un canale verso lo userspace attraverso il quale

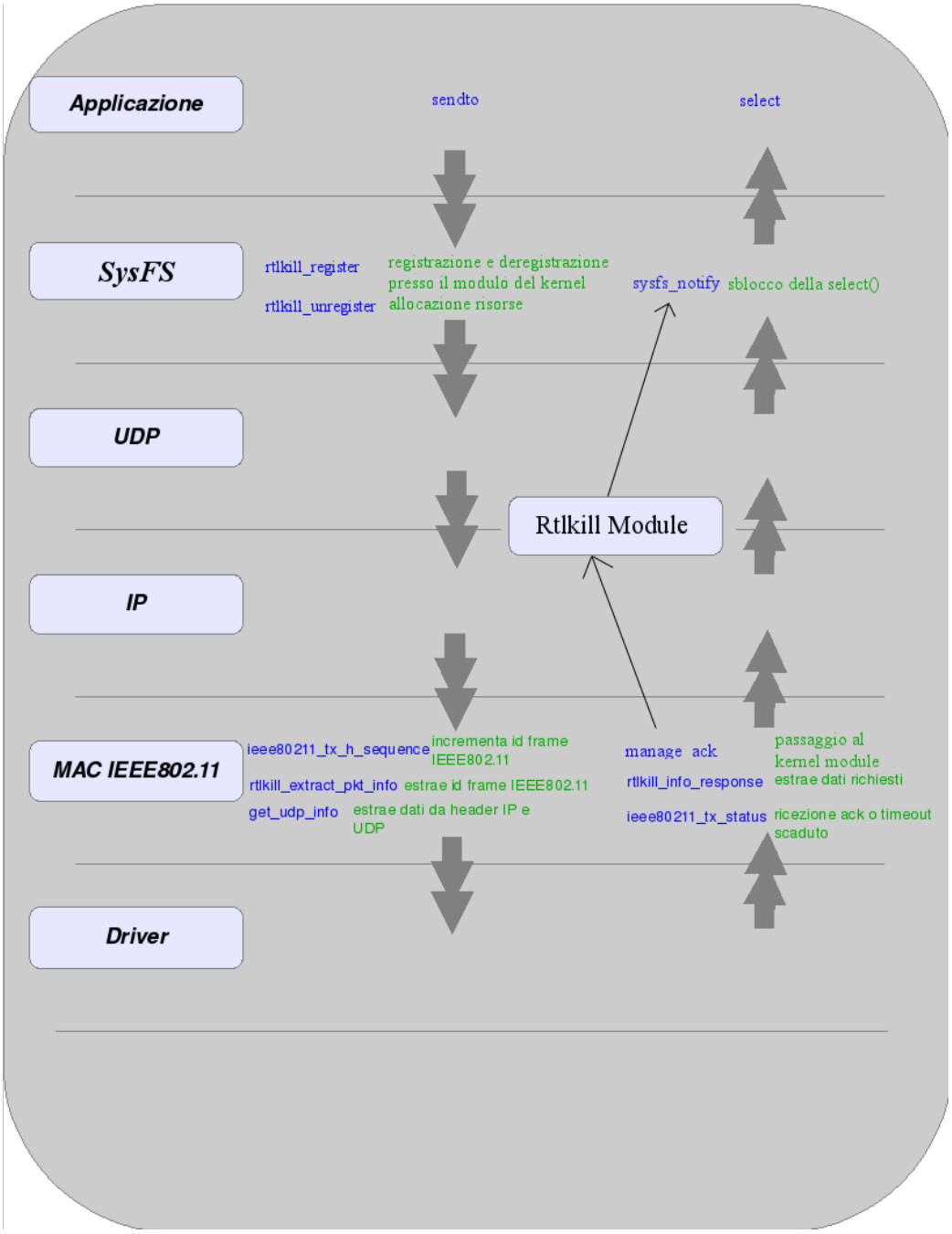

Figura 4.1: Panoramica meccanismo SysFS

verranno esposte le varie informazioni alle applicazioni che lo richiedono. Si e' dunque cercata una soluzione che permettesse di inviare, copiare o comunque esporre informazioni interne all'ambiente del kernel verso lo userspace nel modo piu' semplice ed efficiente possibile. Di seguito riportiamo alcune delle alternative considerate.

4.1 Alternative per comunicazione kernelspace-userspace

Durante lo studio sono state vagliate diverse possibilita' per comunicare i dati verso lo userspace: alcuni metodi sono stati scartati a priori, quali gli IOctl, ProcFS e lo Stack di Netlink.

- IOctl

  Eccessivamente rivolti alla gestione di device specifici per una particolare architettura.

  Necessitano di polling o di gestione basata su interrupt (Interrupt Driven)

- ProcFS[1]

  Filesystem per l'accesso a informazioni riguardanti i processi in esecuzione, e' ben documentato

  E' complesso e disomogeneo (nel tempo si sono stratificati svariati tipi di dati rappresentati) e' inoltre in fase di obsolescenza (verra' rimpiazzato da SysFS).

[1]Process FileSystem: http://en.wikipedia.org/wiki/Procfs

- Netlink [2]

  Presenta un problema simile a quello dell'implementazione tramite le system call in quanto e' necessario definire un identificatore univoco per la nuova famiglia di protocolli che si vuole implementare.

  Il sottosistema netlink e' pensato per trasferimenti di dati bufferizzati e asincroni, non per rapidi aggiornamenti di valori provenienti da aree di memoria del kernel.

- Device a Caratteri[3]

  Complesso, non permette di accedere direttamente alle aree di memoria ma prevede un buffer di copia.

  Richiede la registrazione di un identificatore univoco all'interno del kernel (Major Number), operazione che necessita l'accettazione del progetto da parte della comunita' di sviluppo del kernel Linux e tempi lunghi.

  Come nel caso delle IOctl, richiede la gestione degli interrupt, siano essi provenienti da device o da componenti software (Interrupt Software - trap).

Si e' dunque preferito, per ovviare in parte al problema, spostare tutta la comunicazione allo userspace verso SysFS.

4.2 SysFS

SysFS e' un filesystem virtuale, inserito nel kernel di linux dalla serie 2.6, che espone alcune aree di memoria del nucleo di sistema, definite come variabili di strutture nel kernel stesso, in lettura e scrittura. Queste aree di memo-

[2]Suite per la creazione di socket verso lo userspace e definizione di famiglie di protocolli: http://en.wikipedia.org/wiki/Netlink

[3]Tipo speciale di file che rappresentano una periferica o un dispositivo virtuale su cui possibile effettuare operazioni di un numero arbitrario di byte.Cf: dispositivi a blocchi

ria sono accessibili tramite alcuni file presenti solitamente nella directory sys di un sistema Linux standard[4].

4.2.1 Organizzazione generale

L'architettura di SysFS permette di mappare gerarchicamente le strutture dati di alto livello del kernel con files e directory di un filesystem virtuale, solitamente montato in sys. Questa astrazione permette, dal punto di vista del programmatore del kernel, di accedere a strutture dati che descrivono gli oggetti di interesse in modo immediato e standardizzato, e, dal punto di vista dell'applicazione userspace, di poter lavorare su queste informazioni in modo non diverso da quello usato per lavorare sui comuni file e stream.

4.2.2 Struttura del filesystem virtuale

Una istanza di SysFS e' tipicamente organizzata come segue:

```
user@linbox:~$ ls /sys/
block  bus  class  dev  devices  firmware  fs  kernel  module  power
user@linbox:~$
```

- block: contiene informazioni riguardanti i device a blocchi presenti sul sistema
- bus: wrapper di transizione, contiene link ai device effettivi presenti in dev
- class: contiene informazioni riguardo le classi di device omogenei
- dev: contiene i link simbolici, ordinati tramite MajorNumber MinorNumber, alle relative classi di device in class
- devices: contiene i device di sistema quali bus e chipset integrati che non sono acceduti in modo standard, come blocchi o caratteri

[4]Qualsiasi sistema che aderisca alla Linux Standard Base: http://www.linuxfoundation.org/en/LSB

- firmware: espone le interfaccie di controllo dei firmware in esecuzione nell'hardware di sistema
- fs: informazioni sui filesystem montati, o supportati e relativi parametri run time.
- kernel: permette di accedere a parametri di funzionamento interni del kernel, quali configurazioni del memory manager, controllo degli errori, gestione delle eccezioni, gestione dello scheduler e cosi' via.
- module: contiene i parametri dei moduli attualmente caricati nel kernel.
- power: in questa parte possiamo trovare i file relativi al power management del sistema.

SysFS e' un filesystem che contiene cicli, rispecchiando la struttura delle liste circolari di Kernel Object su cui si basa tutto il meccanismo e che dettaglieremo in seguito.

4.2.3 Kernel Objects

La parte di SysFS che risiede in kernelspace si appoggia su alcuni paradigmi di programmazione che sono tipicamente esterni ai kernel che operano a basso livello. Il kernel di Linux e' diventato, nel tempo, talmente complesso da presentare problematiche che non potevano essere risolte con l'approccio imperativo utilizzato dall'inizio dello sviluppo. Si e' dovuto dunque adottare il paradigma della programmazione in C ad oggetti [15], metodo del quale i kernel object sono ottimo esempio. Il linguaggio C non supporta da specifica il paradigma ad oggetti; e' stato quindi necessario adattare le strutture e i meccanismi tipici della programmazione a oggetti a quelle rese disponibili dal linguaggio C, mantenendone il piu' possibile simile l'approccio logico. Gli oggetti dell'omonimo paradigma sono visti, in C, come normali strutture composte da svariati tipi di dato, gli attributi, e puntatori a funzioni, utilizzate in modo omologo ai metodi di un'istanza di classe. Il kernel object

rappresenta un elemento atomico della rappresentazione in memoria della logica di sysfs, che comprende:

- Un Kernel Object, definito in *linux/lib/kobject.c*, e' una struttura di tipo kobject, avente come campi principali il nome e il contatore di riferimenti. Inoltre, al suo interno, e' presente un puntatore a un oggetto padre (permettendo la gerarchizzazione delle strutture), un campo che ne identifica il tipo specifico e, solitamente, una rappresentazione fisica nel filesystem virtuale SysFS. I kobjects non sono particolarmente significativi se non associati a dati strutturati che descrivono informazioni utili. Come regola comune, nessuna struttura puo' contenere al suo interno piu' di un kernel object, in quanto cosi' facendo il conteggio dei riferimenti sarebbe sicuramente errato, portando a codice facilmente soggetto a errori a runtime.

- Un oggetto ktype e' invece un oggetto che al suo interno contiene un kernel object, e rispettivamente ogni struttura che contiene un kernel object deve avere un ktype corrispondente che controlla le operazioni eseguite sul kobject alla sua creazione e distruzione.

- Un kset e' un gruppo di kernel objects, che possono essere dello stesso ktype o meno. A sua volta un kset contiene un kernel object associato, ma il codice che lo gestisce e' trasparente all'implementazione. Solitamente quando in sys e' presente una directory che ne contiene altre, queste corrispondono a differenti kobject all'interno dello stesso kset.

Kernel object e dati utili

I kernel objects vengono solitamente utilizzati per controllare l'accesso a oggetti piu' grandi, tipici del dominio di SysFS, motivo per il quale spesso essi sono inseriti all'interno di strutture piu' complesse. Per rendere un esempio dell'utilizzo di un kernel object segue ora un'ipotesi di utilizzo per un device teorico che ne utilizza uno in modo semplificato. Se si pensa a un kernel object in termini di programmazione ad oggetti, lo si puo' immaginare come

una classe astratta di alto livello da cui sono derivate altre classi. Un kernel object permette di effettuare operazioni poco utili se prese singolarmente, ma che possono essere utili in altri tipi di oggetti. Il linguaggio C non consente di esprimere direttamente l' ereditarieta' di classe, per cui e' necessario utilizzare altre tecniche, quali l'incapsulamento di strutture.

Tornando all'esempio ipotetico accennato sopra, supponiamo di avere il codice del modulo UIO, che definisce una regione di memoria associata a un dispositivo uio (User Input Output)

```
struct uio_mem {
    struct kobject kobj;
    unsigned long addr;
    unsigned long size;
    int memtype;
    void __iomem *internal_addr;
};
```

Seguendo l'esempio notiamo la struttura interna a uio_mem chiamata kobj, di tipo kobject, e gli altri campi di minore interesse. Data una struttura di questo tipo, per recuperare il kobject incapsulato al suo interno e' sufficiente accedere al membro "kobj". Il codice che utilizza i kobject solitamente avra' il problema opposto: dato un puntatore al kobject, per ottenere il puntatore alla struttura che lo contiene, e' necessario utilizzare la macro *container_of()*, definita in *linuxkernel.h*:

```
    container_of(pointer, type, member)
```

Nell'esempio, un puntatore a una struttura kobject "kp" puo' essere convertito a puntatore alla struttura che lo incapsula con la macro appena descritta:

```
    struct uio_mem *u_mem = container_of(kp, struct uio_mem, kobj);
```

Per completezza esiste anche una macro inversa per accedere direttamente al kernel object data la struttura che lo contiene.

Interfaccia dei kernel objects

Inizializzazione Un kernel object viene inizializzato attraverso l'invocazione di alcune funzioni di libreria, interne al kernel, che configurano alcuni parametri necessari, quali il tipo di kernel object, tramite la funzione:

```
void kobject_init(struct kobject *kobj,
                  struct kobj_type *ktype);
```

Una volta che l'oggetto e' stato inizializzato correttamente, e' necessario notificarne l'esistenza a SysFS tramite la funzione

```
int kobject_add(struct kobject *kobj,
                struct kobject *parent,
                const char *fmt, ...);
```

che attribuisce al kernel object i parametri parent, il quale identifica l'oggetto padre di quello che si sta inserendo, e fmt, che indica il nome del kernel object. Nel caso in cui la struttura che si intende aggiungere sia parte di un kernel set (kset), il parametro parent puo' essere impostato a NULL in modo da assegnare come padre del kernel object il kset stesso.

E' presente nell'interfaccia di SysFS una funzione wrapper che permette di inizializzare e registrare il kernel object in unica soluzione:

```
int kobject_init_and_add(struct kobject *kobj,
                         struct kobj_type *ktype,
                         struct kobject *parent,
                         const char *fmt, ...);
```

con argomenti identici a quelli esposti precedentemente.

Conteggio dei riferimenti Il conteggio dei riferimenti di un kernel object permette di segnalare gli oggetti e il codice associati ai kernel object attivi. Esistono due funzioni fondamentali per manipolare il conteggio dei riferimenti di un kernel object:

```
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);
```

Una chiamata alla funzione *kobject_get()* incrementa di un'unita' il conteggio di riferimenti per quel kernel object e ne ritorna il puntatore. Nel caso invece si stia rilasciando il contatore di riferimenti, e' necessario invocare la funzione *kobject_put()* che lo diminuira' di un'unita' e, nel caso in cui esso sia arrivato a zero, eliminera' il kernel object. E' da notare che il codice della funzione *kobject_init()* incrementa automaticamente il contatore a uno, e' dunque necessario aggiungere una invocazione di *kobject_put()* per rilasciarlo.

Funzioni e attributi Ogni kernel object deve esporre le funzioni che verranno invocate durante le corrispondenti operazioni di inputoutput effettuate sui relativi file di SysFS. Le operazioni di apertura, lettura, scrittura e cosi' via invocate su questi file in realta' eseguono il codice lato kernel associato all'attributo relativo al file. Viene di seguito illustrato un esempio nel caso in cui si voglia esporre allo userspace una variabile intera a 32 bit interna al kernel:

```
static int foo;
```

E' necessario implementare le funzioni per le operazioni di lettura e scrittura e impostare gli attributi del file, quali utenti e permessi. Ad esempio, per leggere e scrivere la varibile foo sono registrate le due seguenti funzioni, riportate in forma minima:

```
static ssize_t foo_show(struct kobject *kobj,
                        struct kobj_attribute *attr,
                        char *buf)
{
        return sprintf(buf, "%d\n", foo);
}

static ssize_t foo_store(struct kobject *kobj,
                         struct kobj_attribute *attr,
                         const char *buf,
                         size_t count)
{
        sscanf(buf, "%du", &foo);
        return count;
}

static struct kobj_attribute foo_attribute =
    __ATTR(foo, 0666, foo_show, foo_store);
```

Le due funzioni presentate possono essere ampliate a piacere dal programmatore per eseguire codice specifico durante le operazioni sui file di SysFS. La definizione di una struttura *kobj_attribute* alla fine del precedente segmento di codice rappresenta gli attributi che un processo userspace incontra nell'accedere al file "foo" (in questo caso, 0666[5]).

Esiste un elenco NULL terminato degli attributi assegnati a ogni file che si intende mappare, definito come segue:

[5]Permessi unix standard, vedere http //en.wikipedia.org/wiki/File_system_permissions#Symbolic_notation

```
static struct attribute *attrs[] = {
        &foo_attribute.attr,
        NULL,
};
```

Il raggruppamento degli attributi in una struttura ne permette la creazione o distruzione in un unica azione.

Ultima struttura da considerare e' la

```
static struct attribute_group attr_group = {
        .attrs = attrs,
};
```

che, se utilizzata, crea sotto directory nella struttura di SysFS aventi come nome quello dell'attributo.

4.3 Infrastruttura a livello kernel

A causa della natura asincrona della ricezione degli acknowledgement nella comunicazione wireless, e' stato necessario progettare un sistema capace di attendere le chiamate provenienti dallo stack MAC80211 e agire di conseguenza, conoscendo quali stream UDP monitorare e quali applicazioni associare a ciascuno di essi; dovendo interfacciarsi con un protocollo veloce, quale il wireless, si e' dovuto prestare particolare attenzione alle performance e al carico computazionale aggiuntivo. La dipendenza del meccanismo di tracciamento dai dati ricevuti dallo stack di rete e' stata eliminata utilizzando un kernel thread[6] che viene attivato solamente in corrispondenza della registrazione di una applicazione tramite alcuni file di SysFS. E' dunque sufficiente che lo stack di rete richiami un' apposita funzione chiamata *manage_ack* la quale, controllando le strutture dati interne al modulo del kernel proposto e aggiornate dal kernel thread tramite le registrazioni, esegue il proprio corpo solamente nel caso di stream UDP che sia necessario tracciare, altrimenti,

[6]Processo in kernel space avente tutte le caratteristiche di un normale processo, tra cui la possibilita' di essere sospeso e riavviato in base ad alcuni eventi. Verra' comunque definito in seguito.

terminando subito, permette allo stack di rete di continuare ad operare senza rallentamenti sensibili.

Riepilogando la struttura ad alto livello, un' applicazione che voglia ricevere informazioni circa la sorte dei propri pacchetti UDP deve innanzitutto registrarsi, utilizzando le funzioni di libreria fornite, presso un file di SysFS. L'operazione di scrittura su questo file risveglia il kernel thread (solitamente dormiente in attesa di eventi di questo tipo) che aggiorna le tabelle aggiungendovi l'applicazione e le informazioni riguardanti lo stream UDP da monitorare e ritornando nel file appena acceduto il nome di un'altro file di SysFS da cui leggere lo stato dei propri pacchetti. Ad ogni elaborazione di un pacchetto UDP lo stack di rete invoca la funzione *manage_ack* che aggiorna le tabelle del SysFS solo nel caso in cui lo stesso sia di interesse. L'applicazione, a sua volta, legge il file di SysFS assegnatole (che corrisponde alle tabelle interne al modulo, dal punto di vista del kernel) e, una volta terminata, si deregistra tramite la rispettiva funzione di libreria. All'atto della deregistrazione, il kernel thread viene di nuovo attivato e cancella dalle tabelle i dati dell'applicazione, scaricando lo stack di rete del lavoro necessario a tracciarle.

4.3.1 Kernel thread

Un thread rappresenta l'implementazione software del concetto di concorrenza fra processi, permettendo ad essi di rimanere sospesi fino al verificarsi di un determinato evento. Questo continuo ricambio di processi attivi e sospesi realizza l'interattivita' tipica dei moderni sistemi operativi, dove si ha l'impressione che l'elaboratore svolga piu' operazioni simultaneamente, ma in realta' e' solo un rapido avvicendamento di diversi thread. La distinzione fra diversi privilegi di esecuzione di un dato codice e' realizzata tramite un flag proprio della struttura[7] che lo rappresenta durante l'esecuzione e che permette di discriminare fra il codice utente, che non deve avere accesso ad aree di memoria riservate e a funzioni specifiche, e il codice del kernel, che per

[7]Solitamente chiamato mode, distinto in usermode e kernelmode. Vedere anche http://en.wikipedia.org/wiki/Kernel_mode#Supervisor_mode

sua natura deve avere il controllo totale dell'elaboratore. Un kernel thread e' dunque analogo al thread userspace, ma esegue in modalita' kernel.

Kernel thread di Linux Il kernel di Linux utilizza un'unica struttura per rappresentare in memoria un thread, chiamato task, sia esso in modalitá utente o kernel. Questa e' definita nella struttura *task_struct* in *includelinuxsched.h*

```
struct task_struct {
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        [...]
        int prio, static_prio, normal_prio;
        [...]
        cpumask_t cpus_allowed;
        [...]
/* task state */
        [...]
        int exit_state;
        int exit_code, exit_signal;
        int pdeath_signal;  /*  The signal sent when the parent dies  */
        pid_t pid;
        pid_t tgid;
        [...]
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->real_parent->pid)
         */
        struct task_struct *real_parent; /* real parent process */
        struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
        /*
         * children/sibling forms the list of my natural children
         */
        struct list_head children;      /* list of my children */
        struct list_head sibling;       /* linkage in my parent's children list */
        struct task_struct *group_leader;       /* threadgroup leader */
        [...]
        cputime_t utime, stime, utimescaled, stimescaled;
        cputime_t gtime;
        cputime_t prev_utime, prev_stime;
        [...]
/* CPU-specific state of this task */
        struct thread_struct thread;
        [...]
/* open file information */
        struct files_struct *files;
        [...]
/* signal handlers */
        struct signal_struct *signal;
        struct sighand_struct *sighand;
        sigset_t blocked, real_blocked;
        sigset_t saved_sigmask; /* restored if set_restore_sigmask() was used */
        struct sigpending pending;
        [...]
};
```

La creazione e la gestione di un kernel thread sono definite in *include/linux/kthread.h*, le principali usate nel modulo proposto sono le seguenti:

```
struct task_struct *kthread_create(     int (*threadfn)(void *data),
                                        void *data,
                                        const char namefmt[],
                                        ...)
```

Questa funzione crea il kernel thread a partire da un file descriptor, un puntatore a un'area di dati iniziali e il nome del thread[8].

```
#define kthread_run(threadfn, data, namefmt, ...)
```

Questa define e' un wrapper per la funzione *kthread_create* la quale, una volta creato il thread, lo avvia tramite l'invocazione di *wake_up_process*.

```
int kthread_stop(struct task_struct *k);
```

La funzione kthread_stop termina il kernel thread passatole e ne rimuove codice, stack e altri dati ancillari dalla lista dei thread attivi.

```
int kthread_should_stop(void);
```

A causa dell'importanza dei dati su cui opera un kernel thread, la terminazione di uno di questi viene segnalata dal kernel e spettera' al codice del thread stesso, nel caso in cui l'invocazione della funzione sopra menzionata ritorni positivamente, eseguire le routine di pulizia e di terminazione delle operazioni in corso prima della effettiva uscita.

4.3.2 Kernel Module

Il kernel di Linux supporta l'aggiunta di parti di codice, dette moduli, che estendono le sue funzionalita' a runtime. Un esempio calzante di modulo del kernel e' il device driver di una penna usb wireless, come quelle utilizzate durante il progetto, che viene caricato solamente durante l'utilizzo del dispositivo. Essendo il modulo del kernel la prima forma di sviluppo di una nuova

[8]E' possibile ottenere una lista dei kernel thread in esecuzione su un sistema linux tramite il comando "ps aux". I kernel thread saranno quelli racchiusi fra parentesi quadre.

funzionalita', prima della eventuale inclusione in mainline - la release stabile, l'interfaccia e' ben documentata, semplice ed espone funzioni complete per la gestione delle operazioni. Nella fattispecie, un modulo del kernel permette di leggere alcuni parametri al caricamento, ottenibili tramite il comando *modinfo*. Questa tecnica permette l'inserimento di codice in un kernel gia' in esecuzione e la relativa condivisione di aree di memoria. E' dunque possibile, utilizzando questo strumento, aggiungere porzioni di codice che ispezionano il comportamento di altre parti gia in esecuzione, dinamicamente, come e' stato fatto in questo progetto. Le funzioni fondamentali per la configurazione di un modulo del kernel sono *module_init* e *module_exit*, che si occupano rispettivamente dell'inzializzazione e della deallocazione del modulo del kernel, a runtime. Queste due funzioni prendono in ingresso come unico parametro rispettivamente un puntatore alle funzioni di inzializzazione e di distruzione definite dal programmatore, invocate dopo le operazioni di base interne al kernel trasparenti all'utente. Lasciando che sia il codice del modulo stesso a gestire i dettagli della sua inzializzazione e distruzione, e' possibile caricare diverse versioni dello stesso modulo durante i test, semplicemente compilando un nuovo modulo ogni volta che se ne modifica il codice.

4.4 Implementazione

Il progetto e' stato organizzato come un modulo esterno al kernel e come patch da applicare a un ramo vanilla[9] della versione 2.6.25.6. E' presente l'infrastruttura di compilazione, basata su make[10], come compilatore viene utilizzato GCC[11], comunemente disponibile nei moderni sistemi Linux.

[9] Versione stabile del kernel, rilasciata su http://www.kernel.org

[10] Tool di compilazione automatica, utilizza una discesa ricorsiva sull'albero dei sorgenti. http://www.gnu.org/software/make/

[11] Gnu Compiler Collection. http://www.gnu.org/software/gcc/

4.4.1 Organizzazione dell' albero sorgente

Struttura

L'albero dei sorgenti e' organizzato come segue:

```
.
|-- Makefile
|-- librtlkill.c
|-- librtlkill.h
|-- rtlkill.c
|-- rtlkill.h
‘-- sendUDP.c
```

Il Makefile contiene le direttive necessarie al tool *make* per creare i file oggetto e il modulo compilato vero e proprio. Il modulo del kernel e' implementato nel file *rtlkill.c* e prototipato in *rtlkill.h*, cosi' come, rispettivamente, le librerie userspace nei file *librtlkill.c* e *librtlkill.h*. Compilando il file *sendUDP.c* si ottiene invece l'eseguibile di test che permette di inviare alcuni dati tramite UDP ad un host sulla rete.

4.4.2 Rtlkill in dettaglio

Di seguito viene dettagliata l'implementazione del sistema cross layer lato modulo del kernel e interazioni con i programmi userspace, per comprendere il lavoro svolto sullo stack di rete e' necessario fare riferimento al Documento di Tesi dei colleghi Codecá e Grassi[16].

Strutture dati

Sono definiti i puntatori a caratteri che conterranno le informazioni da passare verso lo userspace tramite SysFS, cosi' come il canale di monitoraggio, debug e registrazione.

```
static char * rtlkill_reader0;
[...]
static char * rtlkill_reader9;

static char * rtlkill_register;

static char * rtlkill_monitor;

#ifdef DEBUG
static char * rtlkill_debug;
#endif
```

Successivamente vengono configurati i parametri del modulo, gestiti come interi, e le funzioni che permettono di leggerli a runtime definendone descrizione, nome, tipo e permessi.

```
#ifdef DEBUG
static int debug = 0;
module_param(debug, int, 0644);
MODULE_PARM_DESC (debug, " turn on (1) or off (0) the debug, defaults to 0");
#endif
```

Seguono dichiarazioni interne di primitive di mutua esclusione per la gestione della concorrenza, sia fra kernel thread che nella comunicazione fra modulo e stack di rete. La successiva struttura e' fondamentale al progetto,

```
 struct stream_ctl {
        int pid;
        int name;
        char * src_addr;
        char * dst_addr;
        char * src_port;
        char * dst_port;
};

struct stream_table {
        struct stream_ctl stream_array[MAXSTREAM];
        int lastpid;
};
```

descrive la tabella di riferimento dove il kernel thread e le altre funzioni impostano e recuperano dati relativi all'associazione fra stream udp e applicazione, tramite la quadrupla di indirizzo e porta di sorgente e destinazione, utilizzando come discriminante il legame fra porta sorgente e identificativo del processo, poiche' non e' possibile aprire due socket UDP sulla stessa porta. Oltre vengono definiti i due puntatori alle strutture che identificano i kernel thread, quello che fa riferimento all'istanza della tabella centrale e una struttura di tipo packet *stack_result* che consente l'effettivo passaggio delle informazioni tra stack di rete e modulo.

```
static struct task_struct * threadfd;
#ifdef DEBUG
static struct task_struct * debugfd;
#endif

static struct stream_table * rtlkill_stream;

struct packet stack_result;
```

La struttura packet e' definita nel file inclusione *rtlkill.h* come segue:

```
struct packet {
        u8 acked;
        u8 retry_count;
        unsigned long filtered_count;
        u8 fragment;
        __be16 ip_id;
        __be16 udp_port;
};
```

e i suoi campi specificano:

- acked: Campo intero da 8 bit, segnala i possibili valori di acknowledgement: ricevuto, non ricevuto, non richiesto o filtrato.

- retry_count: Contatore intero da 8 bit, indica il numero di volte in cui il pacchetto e' stato rispedito in seguito a un problema.

- filtered_count: Indica il numero di sequenza dell'ack che e' stato filtrato

- fragment: Flag a 8 bit che segnala se in un pacchetto e' presente frammentazione

- ip_id: Identificatore del pacchetto ip

- udp_port: Porta univoca di provenienza del pacchetto UDP

Funzioni per la comunicazione userspace

Come precedentemente esposto, la comunicazione verso userspace avviene tramite file di SysFS. Questi file, quando il modulo viene caricato, si trovano nella directory */sys/kernel/rtlkill*. La struttura della stessa e' riassunta nel seguente schema:

```
.
|-- rtlkill_register     //  scrittura e lettura, utilizzato dalle
|                            librerie userspace per la
|                            registrazione/deregistrazione
|-- rtlkill_debug        //  scrittura, controlla le funzionalita'
|                            di debug
|-- rtlkill_monitor      //  lettura, viene aggiornato con le
|                            informazioni di ogni
|                            stream UDP monitorato dal modulo
|-- rtlkill_reader0      //  lettura, ritorna alla sola applicazione
|                            a cui e' assegnato
|-- [...]                //  le informazioni sul relativo stream UDP
`-- rtlkill_reader9
```

Su ognuno di questi file possono essere invocati alcuni tipi di azioni (lettura, scrittura e cosi' via) per le quali siano state definite le relative funzioni interne al SysFS, i cui permessi sono controllati dagli attributi definiti in seguito. Si riportano di seguito le funzioni operative:

```
static ssize_t rtlkill0_show(struct kobject * kobj,
                             struct kobj_attribute * attr,
                             char * buf) {
        return sprintf(buf, "%s\n", rtlkill_reader0);
}
[...]
static ssize_t rtlkill9_show(struct kobject * kobj,
                             struct kobj_attribute * attr,
                             char * buf) {
        return sprintf(buf, "%s\n", rtlkill_reader9);
}
```

La funzione di show di cui sopra si limita a ritornare al processo userspace che la invoca il valore contenuto nel file di SysFS che corrisponde alla struttura acceduta.

La funzione di show e' molto semplice, mentre quella che permette la memorizzazione di valori e' piu' complessa, necessita infatti di svegliare il kernel thread che gestisce le registrazioni.

```
static ssize_t rtlkill_store(struct kobject * kobj,
                             struct kobj_attribute * attr,
                             const char * buf,
                             size_t count) {
        unsigned int ret;
        ret = sscanf(buf,"%99c",rtlkill_register);
        if (ret != 1) {
                return -EINVAL;
        }
        wake_up_interruptible(&threadfd_waitqueue);
        return count;
}
```

Come si puo' notare, la funzione copia i dati in ingresso nel valore di SysFS analogamente a quella precedente, ma invocando la *wake_up_interruptible* sulla coda di attesa del kernel thread, permette a quest'ultimo di riattivarsi e gestire la nuova registrazione. Il procedimento esatto per riattivare il thread verra' dettagliato piu' avanti, per ora basti sapere che a ogni registrazione di un'applicazione userspace corrisponde un'attivazione del kernel thread.

Seguono gli attibuti di lettura e scrittura dei file di SysFS e le relative strutture ausiliarie in cui essi sono contenuti.

```
 static struct kobj_attribute rtlkill_show_attribute0 =
        __ATTR(rtlkill_reader0, 0444, rtlkill0_show, NULL);
[...]
static struct kobj_attribute rtlkill_show_attribute9 =
        __ATTR(rtlkill_reader9, 0444, rtlkill9_show, NULL);

static struct kobj_attribute rtlkill_store_attribute =
        __ATTR(rtlkill_register, 0666, rtlkill_register_show, rtlkill_store);
```

Dal codice sopra, il file *rtlkill_reader0* ha codificato la lettura per tutti gli utenti, gruppi e altri, mentre il file *rtlkill_register* permette anche la scrittura, per consentire alle applicazioni di registrarsi. Le strutture che contengono gli attributi sono separate, e vengono utilizzate dall'interfaccia del SysFS in modo trasparente al modulo.

```
static struct attribute * attrs[] = {
        &rtlkill_show_attribute0.attr,
    [...]
        &rtlkill_show_attribute9.attr,
        &rtlkill_store_attribute.attr,
        NULL,
};
```

Successivamente si e' implementata la funzione *ack_parser* che, ricevuto dallo stack di rete le informazioni riguardanti il pacchetto correntemente sotto analisi, costruisce la stringa che sara' poi resa disponibile al file di SysFS associato all'applicazione proprietaria di quel pacchetto, scrivendola nella tabella in corrispondenza della porta sorgente UDP a cui quel pacchetto fa riferimento.

```
int ack_parser(char * buffer, const int pid, struct packet stack_result);
```

Ultima, ma non per importanza, si trova il puntatore a struttura kobject *rtlkill_kobj* che viene utilizzato per rappresentare la directory di output nell'albero di SysFS.

```
static struct kobject * rtlkill_kobj;
```

Funzioni per la comunicazione fra modulo e stack di rete

La comunicazione fra modulo e stack di rete avviene tramite la funzione *manage_ack* che riceve una struttura di tipo packet, come precedentemente esposto, e provvede, dopo aver controllato essere l'unica istanza in esecuzione per evitare corse critiche, ad invocare la funzione *ack_parser* per creare la stringa che poi viene scritta nel file di sysfs. La funzione *sysfs_notify* e' una introduzione relativamente recente nel kernel di linux, compare dalla versione 2.6.17, ma permette alle applicazioni di utilizzare la funzione *select* sui file di SysFS, semplificando notevolmente la lettura dagli stessi.

```
int manage_ack(struct packet stack_result) {
        register int i;
        int port;
        bool flag;
        if (mutex_is_locked(&ack_mutex)) {
                printk(KERN_ALERT "sorry, but there is another instance \
                                   of the function manage_ack running, \
                                   i cannot let you continue\n");
                return -1;
        } else {
                mutex_lock(&ack_mutex);
                for (i = 0; i < MAXSTREAM; i++) {
                        sscanf(rtlkill_stream->stream_array[i].src_port, "%d", &port);
                        if (port == stack_result.udp_port) {
                                ack_parser(rtlkill_monitor,
                                           rtlkill_stream->stream_array[i].pid,
                                           stack_result);
                                sysfs_notify(rtlkill_kobj, NULL, "rtlkill_monitor");
                                switch (rtlkill_stream->stream_array[i].name) {
```

```
                                        case 0: {
                                                ack_parser(rtlkill_reader0,
                                                    rtlkill_stream->stream_array[i].pid,
                                                            stack_result);
                                                sysfs_notify(rtlkill_kobj,
                                                             NULL,
                                                             "rtlkill_reader0");
                                                break;
                                        }
                                [...]
                                        case 9: {
                                                ack_parser(rtlkill_reader9,
                                                    rtlkill_stream->stream_array[i].pid,
%                                                           stack_result);
                                                sysfs_notify(rtlkill_kobj,
                                                             NULL,
\                                                            "rtlkill_reader9");
                                                break;
                                        }
                                }
                                port = -1;
                                flag = TRUE;
                        } else {
                                flag = FALSE;
                        }
                }
        }
        mutex_unlock(&ack_mutex);
        return (flag == TRUE ? 0 : 1);
}
EXPORT_SYMBOL(manage_ack);
```

Questa funzione rappresenta il nucleo del sistema cross layer. Il primo comando *if* viene utilizzato per evitare di eseguire il corpo della funzione nel caso ve ne sia un'altra istanza in esecuzione, questo per evitare corse critiche all'interno del modulo e per non caricare lo stack di rete del lavoro necessario a mandare dati verso il modulo quando questo non potrebbe processarli. Nel caso in cui la mutex sia libera, questa viene acquisita e viene scandita la tabella di applicazioni registrate. In caso di corrispondenza positiva viene invocato uno statement *switch* che controlla a che file di SysFS e' associata quella porta UDP e aggiorna le informazioni ivi contenute. Come ultima operazione prima di terminare, notifica verso lo userspace che il file di SysFS e' stato aggiornato permettendo ad applicazioni in attesa di risvegliarsi (tipicamente dalla funzione *select* di cui sopra) e ricevere i dati. E' da notare l'ultima riga del frammento di codice appena presentato, la funzione *EXPORT_SYMBOL* e' una primitiva dell'interfaccia interna del kernel di linux e permette di esportare una nuova funzione in modo che possa es-

sere richiamata da parti di codice anche logicamente distanti all'interno del kernel.

Registrazione

La logica di registrazione e deregistrazione viene gestita in modo congiunto dalla funzione *rederegistration* e dal kernel thread. Il meccanismo e' strutturato come segue: quando un'applicazione si registra, l'operazione di scrittura sul file di SysFS riattiva il kernel thread, che invoca la funzione di registrazione passando come argomento quanto letto dall'area di memoria di *rtlkill_register*, ovvero la stringa di registrazione che viene ben formata da una funzione della libreria userspace, di cui parleremo in seguito. Il kernel thread a questo punto invoca la funzione *rederegistration* che effettua il parsing della stringa e popola le strutture dati relative nella tabella centrale. Similmente, nella deregistrazione le stesse informazioni sono utilizzate per individuare la riga della tabella centrale da eliminare.

```
int rederegistration(char * string, bool bitmap[]) {
        bool flag = TRUE;
        register int i;
        int pos = 0 , actual ;
        char strtmp[100];
        TOKENIZER;
        if (strcmp(strtmp, "register") == 0) {
                for (i = 0; i<MAXSTREAM; i++) {
                        if ((!bitmap[i]) && (flag)) {
                                TOKENIZER;
                                strcpy(rtlkill_stream->stream_array[i].src_addr, strtmp);
                                TOKENIZER;
                                strcpy(rtlkill_stream->stream_array[i].src_port, strtmp);
                                TOKENIZER;
                                strcpy(rtlkill_stream->stream_array[i].dst_addr, strtmp);
                                TOKENIZER;
                                strcpy(rtlkill_stream->stream_array[i].dst_port, strtmp);
                                rtlkill_stream->stream_array[i].name = i;
                                rtlkill_stream->stream_array[i].pid = tmppid;
                        [...]
                                bitmap[i] = TRUE;
                                flag = FALSE;
                                sprintf(strtmp, "%s%d", "rtlkill_reader", i);
                                strcpy(rtlkill_register ,strtmp);
                        }
                }
        } else if (strcmp(strtmp, "unregister") == 0) {
                for (i = 0; i< MAXSTREAM; i++) {
                        if (rtlkill_stream->stream_array[i].pid == tmppid) {
                                switch (rtlkill_stream->stream_array[i].name) {
                                        case 0: {
                                                strcpy(rtlkill_reader0 , "\0");
                                                break;
```

```
                                        }
                                [...]
                                        case 9: {
                                                strcpy(rtlkill_reader9 , "\0");
                                                break;
                                        }
                                }
                                rtlkill_stream->stream_array[i].name = -1;
                                strcpy(rtlkill_stream->stream_array[i].src_addr , "\0");
                                strcpy(rtlkill_stream->stream_array[i].src_port , "\0");
                                strcpy(rtlkill_stream->stream_array[i].dst_addr , "\0");
                                strcpy(rtlkill_stream->stream_array[i].dst_port , "\0");
                                rtlkill_stream->stream_array[i].pid = -1;
                                bitmap[i] = FALSE;
                                sprintf(strtmp, "%d unregistered\n" , tmppid);
                                strcpy(rtlkill_register ,strtmp);
                        [...]
                        }
                }
        } else {
                printk(KERN_ALERT "WARNING: the application %u tried to register with \
                        an unknown or wrong protocol, please correct\n", tmppid);
                return 1;
        }
        return 0;
}
```

Analizzando brevemente la funzione nel dettaglio, si distinguono i due casi principali, dettagliati oltre, e il caso di default in cui un'applicazione scriva dati non ben formati nel file adibito alla registrazione o deregistrazione. Nel caso di registrazione, la stringa appena ricevuta dal kernel thread viene divisa in tokens tramite una macro

```
#define TOKENIZER
                actual = 0 ;
                while  ( string[pos] != ':' ) {
                      strtmp[actual] = string[pos] ;
                      pos++;
                      actual++;
                }
                pos++;
                strtmp[actual] = '\0'
```

definita da noi in quanto la funzione *strtok* offerta dalle librerie di sistema non e' sempre affidabile e reentrant[12]. Questa stringa divisa in token viene utilizzata per popolare i vari campi della tabella centrale associati a uno stream registratosi. L'ultima operazione che questa funzione effettua prima di uscire nel caso della registrazione consiste nel ritornare all'applicazione il file di SysFS che gli e' stato assegnato, scrivendolo su quello addetto alla

[12]Una funzione viene definita reentrant quando piu' istanze della stessa possono essere eseguite in modo concorrente

registrazione. E' successivamente compito delle librerie userspace leggere questo dato e ritornarlo all'applicazione. Nel caso in cui l'applicazione si stia deregistrando controlla, tramite la corrispondenza del process id, che la deregistrazione avvenga correttamente e ripulisce le strutture dati, ritornando come ultima operazione la stringa composta da process id e "unregistered" sul file di SysFS.

Kernel Thread

Come descritto in precedenza, si e' utilizzato un kernel thread per gestire la registrazione e la deregistrazione delle applicazioni userspace. Di seguito sono riportate parti di codice significative con brevi descrizioni del funzionamento.

Funzione rtlkill_thread Il listato seguente contiene il codice della funzione che fisicamente compone il codice eseguito dal kernel thread.

```
static int rtlkill_thread(void * data) {

        DECLARE_WAITQUEUE (wait, current);

        bool bitmap[MAXSTREAM];
        /*int times = 0;*/
        register int i;
        printk (KERN_ALERT "Rtlkill thread started\n");
        for (i = 0; i < MAXSTREAM; i++) {
                bitmap[i] = FALSE;
        }
        daemonize("rtlkill_thread");
        allow_signal(SIGKILL);
        add_wait_queue (&threadfd_waitqueue, &wait);
        while ( 1 ) {
                /* we set the thread to be interruptible, we do NOT want a deadlock! */
                set_current_state(TASK_INTERRUPTIBLE);
                schedule();
                if (signal_pending (current)) break;
                if (it_shall_exit) {
                        remove_wait_queue (&threadfd_waitqueue, &wait);
                        return 0;
                }
                /* the main registration logic */
                rederegistration(rtlkill_register, bitmap);
        }
        set_current_state(TASK_RUNNING);
        remove_wait_queue (&threadfd_waitqueue, &wait);
        return 0;

}
```

E' da notare che la funzione viene dichiarata di tipo void e riceve in input un puntatore a void obbligatorio che puo' essere utilizzato per passare un'area di memoria che funga da stack dei parametri. Come prima operazione dichiara le primitive di concorrenza e le strutture dati che utilizza, come la bitmap, che rappresenta la mappa binaria dei 10 stream che il modulo puo' seguire, un bit impostato a uno indica che lo stream corrispondente, e quindi il relativo file di SysFS, e' associato a un'applicazione. Successivamente, il kernel thread viene registrato sulle code di attesa dello scheduler, prima di entrare in un loop infinito, ove si sospende. Rimane in attesa di un segnale che viene lanciato dalla funzione *rtlkill_store* che invoca la funzione

```
wake_up_interruptible(&threadfd_waitqueue)
```

sulla coda di attesa su cui si e' sospeso il kernel thread. Questa azione risveglia il kernel thread che continua l'esecuzione dall'istruzione successiva. Esegue quindi la registrazione (o la deregistrazione) dell'applicazione che ha scritto sul file *rtlkill_register*. Nel caso il kernel thread debba terminare, viene svegliato un'ultima volta con un segnale in attesa (pending signal), termina il loop infinito, libera le strutture di mutua esclusione e ritorna zero.

Code di attesa Il kernel thread puo' essere gestito tramite apposite primitive per la concorrenza fornite dal kernel di linux, che permettono il completo controllo della vita di un processo. Queste primitive operano su particolari strutture dati, chiamate wait queue, definite nel seguente modo:

```
static DECLARE_WAIT_QUEUE_HEAD (threadfd_waitqueue);
```

Questa dichiarazione crea un nuovo puntatore a una coda di attesa, la quale e' composta da uno spinlock[13] e da una lista di puntatori a processi in attesa di quella mutua esclusione.

```
DECLARE_WAITQUEUE (wait, current);
```

[13]Metodo inefficiente ma semplice per acquisire una mutua esclusione, prevede un thread che attende in busy waiting la disponibilita' della stessa.

Una volta dichiarate le code di attesa, e' necessario inizializzarle e svolgere alcune procedure che permettono al thread di sospendersi e successivamente riavviarsi.

```
daemonize("rtlkill_thread");
allow_signal(SIGKILL);
add_wait_queue (&threadfd_waitqueue, &wait);
```

La funzione *daemonize()* permette di liberare il kernel thread da tutte le risorse utente solitamente associate a un processo. La funzione, inoltre, dalla versione 2.6 del kernel di Linux, invoca autonomamente *reparent_to_init()* che modifica il thread padre di quello appena creato, impostandolo a init[14]. In questo modo si evita che, se un thread viene terminato, i suoi figli in kernel space rimangano bloccati (processi zombie[15]. Successivamente, e' necessario invocare la funzione *allow_signal()* per permettere nuovamente al thread di gestire i segnali che sono stati disabilitati dall'invocazione della *daemonize* quali, in questo caso, SIGKILL, indicante la terminazione. Si noti, infatti, che nel codice del kernel thread viene invocata la funzione *signal_pending()* per controllare se e' in arrivo il segnale di terminazione. Infine, la funzione *add_wait_queue* aggiunge alla coda generica di attesa wait del kernel la coda propria del kernel thread.

```
        set_current_state(TASK_INTERRUPTIBLE);
        schedule();
        if (signal_pending (current)) break;
```

In questa porzione di codice, eseguita all'inzio del loop infinito, il processo imposta il proprio stato come interrompibile e, invocando la funzione *schedule* chiede allo scheduler di poter essere eseguito per sospendersi.

[14]Il processo init é il primo processo che il Kernel manda in esecuzione dopo aver terminato il suo bootstrap; esso ha il compito di portare il sistema in uno stato operativo, avviando i programmi e servizi necessari

[15]processo che nonstante abbia terminato la propria esecuzione, possiede ancora un PID ed un process control block, necessario per permettere al proprio processo padre di leggerne il valore di uscita.

Nel caso in cui sia necessario terminare il kernel thread, si effettua la pulizia delle code, tramite le funzioni

```
set_current_state(TASK_RUNNING);
remove_wait_queue (&threadfd_waitqueue, &wait);
```

e si termina il thread.

Thread di Debug Durante gli stadi iniziali di progettazione e implementazione e' stato necessario costruire un sistema che permettesse di testare il kernel module, generando un finto stream UDP. La gestione del kernel thread di debug e' analoga a quella del thread adibito alla registrazione, presenta strutture dati e funzioni di gestione equivalenti. La disponibilita' del thread di test e' subordinata alla definizione, durante la compilazione, del simbolo *RTLKILL_DEBUG* e, durante il caricamento, da un parametro *debug* che deve essere passato al modulo tramite il comando *modprobe*. Nel caso in cui il debug sia abilitato, al caricamento del modulo il thread di test non viene creato. E' necessario attivare l'infrastruttura scrivendo alcune parole chiave, dettagliate in seguito, nel file *rtlkill_debug*. Quando il thread e' in esecuzione, non fa altro che eseguire un ciclo infinito in cui si sostituisce allo stack mac80211 nell'invocare la funzione *manage_ack* passando ad essa finte strutture *packet* contenenti informazioni ben formattate ma fittizie che consentono di testare il meccanismo cross layer. La complessita' maggiore e' individuabile nella funzione *rtlkill_debug_store*, funzione dell'ambiente del kernel che viene invocata in corrispondenza di una scrittura sul file di SysFS, che implementa una macchina a stati finiti per la gestione della vita del thread di test tramite il riconoscimento dei quattro parametri *start—stop—suspend—resume* che possono essere scritti in SysFS.

Macchina a stati finiti La gestione del kernel thread di debug tramite una macchina a stati finiti si basa sulla scrittura di uno di quattro parametri nel file di SysFS, che sono associati alle seguenti azioni:

- start: Creazione e avvio del thread, scrittura di un messaggio di errore se esiste gia' un thread di debug attivo.

- stop: Terminazione ed eliminazione del thread e pulizia delle strutture dati associate, stampa un errore se non sono presenti kernel thread di test.

- suspend: Obbliga il thread a sospendersi asserendo una variabile il cui valore viene controllato nel corpo del thread stesso, ritorna errore se esistono thread gia' sospesi o se non vi sono thread in esecuzione.

- resume: Invia un segnale di tipo *SIGCONT* al thread, che riprende la sua esecuzione, e stampa un errore nel caso in cui il thread sia gia attivo o non esistente.

4.4.3 Interfaccia del modulo

Librerie userspace

E' stata sviluppata una libreria userspace che consente la corretta registrazione di una applicazione presso il meccanismo cross-layer con relativa infrastruttura di compilazione.

Protocollo di registrazione Le funzioni delle librerie userspace che gestiscono la registrazione sono *rtlkill_register* e *rtlkill_unregister*. Entrambe ricevono in input una struttura di tipo *rtlkill_registrar*, che contiene i dati riguardanti lo stream UDP che l'applicazione invocante intende tracciare, e un buffer di caratteri, utilizzato per ritornare il nome del file di SysFS associato all'applicazione nel caso della registrazione e un messaggio di conferma dal kernel nel caso della deregistrazione.

```
struct rtlkill {
  char src_addr[16];
  char src_port[6];
  char dst_addr[16];
  char dst_port[6];
};
```

Come si puo' facilmente intuire dai nomi delle variabili, i campi di questa struttura contengono gli indirizzi e le porte di sorgente e destinazione dello stream UDP di interesse. La funzione *rtlkill_register*, il cui corpo e' omesso

per brevita' ma si puo' trovare nell'appendice, costruisce una stringa ben formata utilizzando i valori della struttura registrar passatagli:

```
sprintf(stringa, "register:%s:%s:%s:%s:",
        registrar.src_addr,
        registrar.src_port,
        registrar.dst_addr,
        registrar.dst_port);
```

e, dopo aver aperto il file di registrazione, scrive su di esso la stringa. L'esecuzione viene sospesa per un tempo brevissimo, necessario a permettere la registrazione dell'applicazione nell'ambiente del kernel, dopodiche' viene letto dallo stesso file il nome di quello di SysFS che e' stato assegnato all'applicazione. La funzione di deregistrazione funziona in modo analogo, scrivendo pero' una stringa iniziante con *unregister* e leggendo successivamente un messaggio di conferma dal kernel contenente l'identificativo (pid) dell'applicazione invocante.

4.4.4 Codice di test

Durante lo sviluppo e l'implementazione del meccanismo di passaggio dei dati verso lo userspace e' stato necessario controllarne il funzionamento tramite un'applicazione di test. Questa e' stata fornita nel file *sendUDP.c* e utilizza le librerie di cui sopra per registrare una fittizia applicazione che invia pacchetti UDP in dimensione e numero variabili. Lanciando l'applicazione senza parametri e' possibile visualizzarne una lista:

```
usage: sendUDP
  packet_size
  dst_addr
  dst_port
  delayusec
  src_addr
  packet_number
```

Questi parametri permettono di configurare il numero e la dimensione dei pacchetti UDP da inviare verso una destinazione e porta specificate.

Il programma di test, all'avvio, instanzia un socket UDP e ne configura i parametri. Successivamente, utilizzando le librerie fornite registra se stesso e riceve il nome del file di SysFS sul quale ascoltare in attesa delle informazioni sui pacchetti inviati. La funzione di libreria per la registrazione e' definita come

```
rtlkill_register(register_struct, buffer);
```

dove la struttura *register_struct* opportunamente configurata contiene le informazioni sul socket UDP aperto, mentre nell'area di memoria puntata da *buffer* verra' ritornato il file di SysFS che e' stato assegnato dal kernel. Il programma attende quindi in un ciclo *for* che il file di SysFS venga aggiornato tramite una chiamata a *select* che, quando ritorna, permette di leggere l'informazione sul pacchetto inviato. Al termine del ciclo, quando cioe' sono stati inviati tutti i pacchetti, viene invocata la funzione

```
rtlkill_unregister(register_struct, buffer);
```

con parametri identici a quelli della registrazione, che libera le risorse del kernel e fa terminare il programma.

Esempio di esecuzione del test Di seguito viene riportata l'esecuzione del programma di test collegato al file di SysFS *rtlkill_monitor* che riceve informazioni sui pacchetti di tre differenti stream UDP la cui trasmissione viene disturbata ad hoc:

```
    user@rtlkill: ./sendUDP 1024 192.168.1.69 5678 192.168.1.37 10 30 monitor

    sizemsg 1024 B, delay 10 usec
    configured the register struct to be: src_addr 192.168.1.37,
                                          dst_addr 192.168.1.69,
                                          dst_port 5678
    rtlkill_monitor registered
    /sys/kernel/rtlkill/rtlkill_monitor is path to sysfs, starting select()

    #pid        #upd_port       #ip_id  #retry  #frag   #filter #ack
1   25075       12240           5464    0       1       0       0
    25075       33684           5465    0       1       0       0
    25075       33684           5466    0       1       0       0
    25075       33684           5467    1       1       0       0
5   25075       501             5468    1       1       0       0
    25075       12240           5469    3       1       0       0
    25075       12240           5470    3       1       0       0
    25075       33684           5471    0       1       0       1
    25075       33684           5472    0       1       0       1
10  25075       33684           5473    0       1       0       1
    25075       501             5474    0       1       0       1
    25075       12240           5475    0       1       0       0
    25075       33684           5476    0       1       0       0
    25075       33684           5477    0       1       0       0
15  25075       33684           5478    0       1       0       0
    25075       33684           5479    0       1       0       0
    25075       501             5480    1       1       0       0
    25075       12240           5481    1       1       0       0
    25075       33684           5482    2       1       0       0
20  25075       33684           5483    4       1       0       1
    25075       33684           5484    3       1       0       1
    25075       501             5485    0       1       0       1
    25075       12240           5486    0       1       0       0
    25075       12240           5487    0       1       0       0
25  25075       33684           5488    0       1       0       1
    25075       33684           5489    1       1       0       0
    25075       33684           5490    0       1       0       0
    25075       501             5491    0       1       0       0
    25075       12240           5492    0       1       0       0
30  25075       12240           5493    0       1       0       0
```

In colonna troviamo, rispettivamente, il PID dell'applicazione di test che sta trasmettendo i dati tramite socket UDP, l'identificatore della porta ad esso associata e il contatore incrementale dei pacchetti ip in cui lo stream e' incapsulato. Seguono *retry_count*, contatore del numero delle ritrasmissioni del singolo pacchetto, un flag che indica la presenza o meno di frammentazione, un flag *filtered* che indica l'eventuale filtraggio del pacchetto da parte del dispositivo wireless e il campo di interesse, il flag *ack*, che assume valore

1 in caso di frame non ricevuto e 0 altrimenti. Nel listato si puo' notare come, in corrispondenza delle linee 47, il collegamento fatichi a consegnare i pacchetti UDP (si noti il campo *retry_count* che aumenta, indicando piu' tentativi necessari a spedire il singolo pacchetto) fino a fermarsi per qualche pacchetto a causa dei disturbi inseriti appositamente. Appena questi cessano, la comunicazione riprende fino alle righe 2023 dove si verifica un'altra simile breve perdita di pacchetti.

Capitolo 5

Sviluppi Futuri

5.1 IPv6

L'ICANN[1] ha reso di sponibile il protocollo IPv6 sui root server DNS gia' dal dal 20 luglio 2004 ma i primi indirizzi IPv6 sono stati resi risolvibili solo dal febbraio 2008. Dati gli attuali trend di crescita della rete Internet e dell'utilizzo del computer, si stima che il protocollo IPv4 rimarra' in uso fino al 2025, per permettere a ISP, aziende e utenti di passare alla nuova versione e correggere eventuali errori.

Oltre a rispondere all'esigenza dell'aumento dello spazio di indirizzamento, l'IPv6 incorpora alcuni protocolli che prima erano separati, il piu' importante e' l'ARP, ed e' in grado di impostare automaticamente alcuni parametri di configurazione della rete, quali ad esempio il default gateway. Inoltre supporta nativamente il QOS e introduce l'indirizzamento anycast, che permette ad un computer in rete di raggiungere automaticamente il piu' vicino server disponibile di un dato tipo (un DNS, per esempio) anche senza conoscerne a priori l'indirizzo.

[1]ICANN (Internet Corporation for Assigned Names and Numbers): ente internazionale non profit, istituito il 18 settembre 1998 per proseguire in numerosi incarichi di gestione relativi alla rete Internet che in precedenza erano demandati ad altri organismi.

Al protocollo IP sono state aggiunte numerose migliorie, le piu' immediate delle quali sono

- header di lunghezza fissa (40 byte)
- impostazione di pacchetti non frammentabili dai router
- eliminazione del campo checksum

Queste tre novita' semplificano il lavoro dei router, migliorando l'instradamento e il throughput. IPv6 e' la seconda versione dell'Internet Protocol ad essere ampiamente sviluppata, e costituira' la base per la futura espansione di Internet. A causa della inevitabile adozione del nuovo standard potrebbe essere molto indicato portare l'attuale codice a questo protocollo.

5.2 Monitor

Allo stato attuale tutti i dati sulla sorte dei pacchetti inviati da una applicazione sono passati all'applicazione stessa, che deve decidere cosa fare di volta in volta e in base alle condizioni di carico. Questo relega le informazioni alla gestione dell'applicazione e non permette di ottenere una visione generale dello stato del canale. Sarebbe dunque interessante sviluppare una applicazione monitor, in stile demone Unix, che controlli la percentuale di ricezione dei pacchetti UDP di tutti i programmi che ne stanno inviando per poter decidere se cambiare canale, interfaccia o rete. Il monitor proposto potrebbe permettere a un sistema con piu' schede wireless di passare attraverso piu' reti mantenendo la comunicazione stabile.

5.3 IEEE 802.11r

Il 15 Luglio 2008 l'IEEE ha pubblicato lo standard 802.11r-2008[13] anche conosciuto come Fast BSS Transition. Queste modifiche permettono un cambio di AP dinamico e quindi rendono reale la continuita' della connessione transitando nella rete in maniera rapida e sicura.

Alcuni dettagli per permettere un servizio del genere sono gia stati introdotti in altre implementazioni del protocollo, tuttavia il ritardo di consegna di circa 100 ms e' sempre stato troppo lungo per supportare applicazioni come voce e video ed e' particolarmente problematico per la sicurezza delle connessioni che utilizzando WPA/WPA2 che possono richiedere da 2 a 8 secondi per la negoziazione.

Essendo questo standard IEEE decisamente nuovo, al momento non ne esistono implementazioni commerciali funzionanti. Sarebbe possibile adattare e integrare il lavoro proposto a questo nuovo standard, considerata la forte somiglianza dei due intenti. Rimane tuttavia una notevole soddisfazione per gli autori l'aver "previsto" la futura direzione degli sforzi della IEEE.

5.4 Kernel Linux

Durante il processo di sviluppo si e' cercato di mantenersi il piu' possibile aderenti alle linee guida[17] della programmazione nel kernel di Linux in vista di un futuro possibile lavoro di integrazione del codice nella mainline del kernel. E' tuttavia necessario segnalare una notevole inerzia dal punto di vista degli sviluppatori di Linux nell'accettare nuove proposte dovendo esse soddisfare i rigidi requisiti del codice di sistema. Integrando il lavoro svolto con lo standard sopracitato dovrebbe essere possibile ottenere maggior rilevanza, ma e' comunque un processo arduo.

5.5 Sistemi Embedded

I sistemi embedded utilizzanti tecnologie wireless, quali la maggior parte di telefoni cellulari, smartphone, access point, netbook e router, potrebbe sicuramente trarre vantaggio dall'utilizzo di sistemi che permettano la mobilita' come quello proposto. Il porting di tutta la base di codice a sistemi embedded, tuttavia, presenta alcuni problemi pratici e teorici, in primis la limitatezza dell'hardware, la differenza architetturale dei device e la moltitu-

dine di sistemi di sviluppo, implementazione e gestione dei dispositivi embedded in generale. Una facilitazione notevole si ha nel lavorare sulle piattaforme che nativamente supportano qualche versione embedded di linux, in particolare OpenWRT[2] e DDwrt[3], sulle quali il porting del codice dovrebbe essere facilitato, anche grazie agli SDK[4] open source forniti.

[2]OpenWRT e' un firmware basato su Linux per dispositibi embedded come i router di fascia consumer.

[3]Progetto simile a OpenWRT, condivide la base di codice, e' piu' semplice da configurare e installare ma e' meno potente e sviluppabile

[4]Software Developement Kit: toolchain di compilazione e documentazione di software per determinati sistemi o architetture

Capitolo 6

Conclusioni

L'attuale diffusione delle tecnologie wireless e VoIP e' in continua crescita, cosi' come le applicazioni utente sia professionali che personali.

L'utilizzo combinato delle due tecnologie e' ostacolato principalmente dalla instabilita' e suscettibilita' alle interferenze delle reti wireless da un lato, e dall'utilizzo, da parte delle applicazioni VoIP, del protocollo UDP, che non garantisce la consegna dei dati e la loro affidabilita'.

Lo scenario ipotetico delineato vede un dispositivo mobile connesso ad un access point che si muove nello spazio senza perdere connettivita' finche' rimane nel raggio di azione di uno o piu' ulteriori access point per i quali abbia le credenziali di autenticazione. Questo e' possibile grazie allo scambio di pacchetti di acknowledgement fra il client e l'access point. Le modifiche proposte allo stack di rete consentono di migliorare l'interattivita' e la comunicazione della sorte dei pacchetti all'applicazione a cui essi appartengono. Siccome i driver delle schede wireless su cui si appoggia tutto questo lavoro non sono sempre precisi nell'esporre le informazioni, esistono casi in cui le informazioni fornite allo stack sono volutamente errate (come nel caso della rtl8187) o semplicemente non affidabili a causa dell'astrazione imposta dal firmware. In questi casi il software non puo' rendersi conto della discrepanza e potrebbe indurre applicazioni userspace a prendere le scelte sbagliate.

Il codice integrato nello stack e il modulo esterno sono stabili, indipen-

dentemente dal driver utilizzato, e non appesantiscono o rallentano significativamente il kernel. Particolare cura e' stata posta nel mantenimento delle funzionalita' dello stack di rete, che non viene minimamente modificata dall'aggiunta del modulo di controllo proposto. Inoltre, il carico computazionale sulla cpu e' trascurabile e, nei test effettuati, si e' sempre riscontrata una impronta di memoria raramente superiore ai 10Kb.

Lo studio ha evidenziato come la affidabilita' e precisione dei dati riportati dalla scheda sia influenzato notevolmente dal numero di astrazioni attraverso le quali questi dati passano: le schede PCI (o comunque direttamente connesse a un bus di sistema) sono tipicamente piu' affidabili di quelle USB o PCMCIA.

Il lavoro svolto, l'interazione con i colleghi e con il relatore e la necessita' di lavorare su software attivo e attivamente sviluppato ha permesso di crescere come informatico e di acquisire competenze che esulano la mera scrittura di codice, quali l'organizzazione di un team di sviluppo, l'interazione con altri sviluppatori esperti, l'utilizzo di infrastrutture di gestione dei bug e cosi' via che hanno arricchito ulteriormente questa esperienza.

Bibliografia

[1] Andrew S. Tanenbaum, Computer Networks (4th Edition), Prentice Hall, 2003

[2] L. Peterson, B. Davie, Computer Networks: A Systems Approach (3rd Edition), Morgan Kaufmann, 2003

[3] Institute of Electrical and Electronics Engineers, IEEE 802.11^{TM}WIRELESS LOCAL AREA NETWORKS, Website, 2008, http://www.ieee802.org/11/

[4] Institute of Electrical and Electronics Engineers, IEEE 802.11^{TM}Standard, Website, 2008, http://standards.ieee.org/getieee802/802.11.html

[5] BBC, Wi-Fi: A warning signal, Website, 2007, http://news.bbc.co.uk/2/hi/programmes/panorama/6674675.stm

[6] Fabio M. Zambelli, Report semina terrore sul Wi-Fi, Website, 2008, http://www.setteb.it/content/view/3934

[7] Filippo Vendrame, I tedeschi temono il Wi-Fi, Website, 2007, http://www.oneadsl.it/28/09/2007/i-tedeschi-temono-il-wi-fi/

[8] Francesca Guido, Pericolo wi-fi, Website, 2007, http://www.datamanager.it/articoli.php?visibile=1&idricercato=21059

[9] Information Sciences Institute University of Southern California, RFC791 - Internet Protocol, 1981, http://www.faqs.org/rfcs/rfc791.html

[10] J. Postel, RFC768 - User Datagram Protocol, 1980, http://www.faqs.org/rfcs/rfc768.html

[11] Autori vari: IETF, ISI, Voice Over IP, Website, http://www.protocols.com/pbook/VoIPFamily.htm

[12] Roberto Arcomano, VoIP Howto, Website, 2002, http://www.faqs.org/docs/Linux-HOWTO/VoIP-HOWTO.html

[13] IEEE P802.11 Task Group R, Project IEEE 802.11r, Website, 2008, http://grouper.ieee.org/groups/802/11/Reports/tgr_update.htm

[14] The sysfs Filesystem, Patrick Mochel http://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf

[15] Object oriented ANSI C, Axel Schreiner
http://www.cs.rit.edu/ ats/books/ooc.pdf

[16] Un meccanismo cross-layer per il supporto all'interattivitá
in sistemi wireless, 2008 Lara Codecá, Giulio Grassi

[17] Linux Cross Reference
http://lxr.linux.no/source/Documentation/CodingStyle

Appendice A

APPENDICE

```
/*****************************************************************************

  Copyright(c) 2008 - 2009 Bigoi Matteo, Codeca' Lara, Ghini Vittorio, Grassi
  Giulio, Lusuardi Andrea

  This program is free software; you can redistribute it and/or modify it
  under the terms of version 2 of the GNU General Public License as
  published by the Free Software Foundation.

  This program is distributed in the hope that it will be useful, but WITHOUT
  ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
  FITNESS FOR A PARTICULAR PURPOSE.  See thcrisi.homelinux.nete GNU General Public License for
  more details.

  You should have received a copy of the GNU General Public License along with
  this program; if not, write to the Free Software Foundation, Inc.,
  51 Franklin Street, Fifth Floor, Boston, MA 02110, USA

  The full GNU General Public License is included in this distribution in the
  file called LICENSE.

  Contact Information:
        Prof. Ghini Vittorio - ghini@cs.unibo.it
        Bigoi Matteo - bigoi@cs.unibo.it
        Codeca' Lara - codeca@cs.unibo.it
        Grassi Giulio - grassig@cs.unibo.it
        Lusuardi Andrea - lusuarda@cs.unibo.it

*****************************************************************************/

/*
 * Libraries inclusion
 */
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/module.h>
#include <linux/err.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/signal.h>
#include <linux/sysfs.h>
#include <linux/kobject.h>
```

```
43 #include <linux/version.h>
   #include <asm/current.h>
45 #include <linux/kernel.h>
   #include <linux/mutex.h>
47 #include <linux/wait.h>
   #include <net/mac80211.h>
49 #include "rtlkill.h"

51
   /*
53  * Module parameteres initialization
    */
55 MODULE_LICENSE("GPL");
   MODULE_AUTHOR("Bigoi, Codeca, Grassi, Lusuardi \
57                 rtlkill@lists.crisidev.org");
   MODULE_DESCRIPTION("Main rtlkill 80211 frames \
59                     acknowlegdement module");

61 /*
    * Prototypes
63  */
   static int test_thread (void *);
65
   /*
67  * rtlkill sysfs objects
    */
69 static char * rtlkill_reader0;
   static char * rtlkill_reader1;
71 static char * rtlkill_reader2;
   static char * rtlkill_reader3;
73 static char * rtlkill_reader4;
   static char * rtlkill_reader5;
75 static char * rtlkill_reader6;
   static char * rtlkill_reader7;
77 static char * rtlkill_reader8;
   static char * rtlkill_reader9;
79
   static char * rtlkill_register;
81
   static char * rtlkill_monitor;
83
   static char * rtlkill_debug;
85
   /* Debug statement parameters */
87 static int debug = 0;
   module_param(debug, int, 0644);
89 MODULE_PARM_DESC (debug, " turn on (1) or off (0) the debug, \
                           defaults to 0");
91
   /* temporary pid used to copy it to the local function */
93 static int tmppid;

95 /* global counter RIMUOVERE */
   static bool was_frag = FALSE;
97
   /* flag used to check wether the thread is forced to exit */
99 static bool it_shall_exit;

101 /* kernel wait queue used to manage threads schedulation */
   static DECLARE_WAIT_QUEUE_HEAD (threadfd_waitqueue);
103 static DECLARE_WAIT_QUEUE_HEAD (debugfd_waitqueue);

105
   /* mutex used to prevent the manage_ack function to get called more than once at a time*/
```

```
107 struct mutex ack_mutex;

109 /* structures and binary flags used to manage the debug thread*/
    static bool shallistop = FALSE;
111 static bool thread_lock = TRUE;
    static bool start_mutex = TRUE;
113 static bool stop_mutex = FALSE;

115 /*
     * Stream table used to associate the 4-tuple to the process calling it
117  * X process can control many streams, but no stream can be controlled by 2 or more processes
     */
119 struct stream_ctl {
            int pid;
121         int name;
            char * src_addr;
123         char * dst_addr;
            char * src_port;
125         char * dst_port;
    };
127
    /*
129  *The stream table, used to keep record of the allocated and free controllable streams.
     */
131 struct stream_table {
            struct stream_ctl stream_array[MAXSTREAM];
133         int lastpid;
    };
135
    /*
137  * kernel threads
     */
139 static struct task_struct * threadfd;
    static struct task_struct * debugfd;
141
    /*
143  * Stream table object
     */
145 static struct stream_table * rtlkill_stream;

147 /*
     * Data coming from the stack
149  */
    struct packet stack_result;
151
    /*
153  * Show functions for the reader objects
     */
155 static ssize_t rtlkill0_show(struct kobject * kobj, struct kobj_attribute * attr, char * buf) {
            return sprintf(buf, "%s", rtlkill_reader0);
157 }

159 static ssize_t rtlkill1_show(struct kobject * kobj, struct kobj_attribute * attr, char * buf) {
            return sprintf(buf, "%s", rtlkill_reader1);
161 }

163 static ssize_t rtlkill2_show(struct kobject * kobj, struct kobj_attribute * attr, char * buf) {
            return sprintf(buf, "%s", rtlkill_reader2);
165 }

167 static ssize_t rtlkill3_show(struct kobject * kobj, struct kobj_attribute * attr, char * buf) {
            return sprintf(buf, "%s", rtlkill_reader3);
169 }
```

```
171 static ssize_t rtlkill4_show(struct kobject * kobj, struct kobj_attribute * attr, char * buf) {
        return sprintf(buf, "%s", rtlkill_reader4);
173 }

175 static ssize_t rtlkill5_show(struct kobject * kobj, struct kobj_attribute * attr, char * buf) {
        return sprintf(buf, "%s", rtlkill_reader5);
177 }

179 static ssize_t rtlkill6_show(struct kobject * kobj, struct kobj_attribute * attr, char * buf) {
        return sprintf(buf, "%s", rtlkill_reader6);
181 }

183 static ssize_t rtlkill7_show(struct kobject * kobj, struct kobj_attribute * attr, char * buf) {
        return sprintf(buf, "%s", rtlkill_reader7);
185 }

187 static ssize_t rtlkill8_show(struct kobject * kobj, struct kobj_attribute * attr, char * buf) {
        return sprintf(buf, "%s", rtlkill_reader8);
189 }

191 static ssize_t rtlkill9_show(struct kobject * kobj, struct kobj_attribute * attr, char * buf) {
        return sprintf(buf, "%s", rtlkill_reader9);
193 }

195 static ssize_t rtlkill_register_show(struct kobject * kobj,
                                     struct kobj_attribute * attr,
197                                      char * buf) {
        return sprintf(buf, "%s", rtlkill_register);
199 }

201 static ssize_t rtlkill_monitor_show(struct kobject * kobj,
                                    struct kobj_attribute * attr,
203                                     char * buf) {
        return sprintf(buf, "%s", rtlkill_monitor);
205 }

207 /*
    *Store function for the register object
209  */

211 static ssize_t rtlkill_store(struct kobject * kobj,
                             struct kobj_attribute * attr,
213                              const char * buf,
                             size_t count) {
215         unsigned int ret;
        ret = sscanf(buf,"%99c",rtlkill_register);
217         if (ret != 1) {
                return -EINVAL;
219         }
        tmppid = current->pid;
221         wake_up_interruptible(&threadfd_waitqueue);
        return count;
223 }

225 static ssize_t rtlkill_debug_store(struct kobject * kobj,
                                   struct kobj_attribute * attr,
227                                    const char * buf,
                                   size_t count) {
229         unsigned int ret;
        int pos = 0 , actual ;
231         char strtmp[20];
        if (debug == 1) {
233                 strcpy(strtmp, "\0");
                ret = sscanf(buf, "%19c", rtlkill_debug);
```

```
235                 if (ret != 1) {
                            return -EINVAL;
237                 }
                    actual = 0;
239                 while ( rtlkill_debug[pos] != '\n' ) {
                            strtmp[actual] = rtlkill_debug[pos];
241                         pos++;
                            actual++;
243                 }
                    pos++;
245                 strtmp[actual] = '\0';
                    if (strcmp(strtmp, "start") == 0) {
247                         if (start_mutex) {
                                    printk(KERN_ALERT "Starting the rtlkill test thread\n");
249                                 shallistop = FALSE;
                                    debugfd = kthread_run(test_thread, NULL, "krtlkilldebug");
251                                 thread_lock = FALSE;
                                    start_mutex = FALSE;
253                                 stop_mutex = TRUE;
                            } else {
255                                 printk(KERN_ALERT "There is an other instance of \
                                                    the thread running\n");
257                         }
                    } else if (strcmp(strtmp, "stop") == 0) {
259                         if (stop_mutex) {
                                    printk(KERN_ALERT "Terminating the rtlkill test thread\n");
261                                 ret = kthread_stop(debugfd);
                                    thread_lock = TRUE;
263                                 stop_mutex = FALSE;
                                    start_mutex = TRUE;
265                         } else {
                                    printk(KERN_ALERT "There are no more instances \
267                                                 of the test thread\n");
                            }
269                 } else if (strcmp(strtmp, "suspend") == 0) {
                            if (!start_mutex) {
271                                 printk(KERN_ALERT "Suspending the rtlkill test thread\n");
                                    shallistop = TRUE;
273                         } else {
                                    printk(KERN_ALERT "No thread to suspend\n");
275                                 thread_lock = TRUE;
                            }
277                 } else if (strcmp(strtmp, "resume") == 0) {
                            if (!start_mutex) {
279                                 printk(KERN_ALERT "Resuming the rtlkill test  thread\n");
                                    shallistop = FALSE;
281                                 kill_proc(debugfd->pid, SIGCONT, 1);
                            } else {
283                                 printk(KERN_ALERT "No thread to resume\n");
                                    thread_lock = TRUE;
285                         }
                    } else {
287                         printk(KERN_ALERT "Rtlkill Protocol Error!\n");
                    }
289                 return count;
            } else {
291                 return -EINVAL;
            }
293 }

295

297 /*
     * utility function that prints the whole stream_table structure in a human readable format
```

```
299  */
     void print_stream_table(void) {
301          int i;
             for (i = 0; i < MAXSTREAM; i++) {
303                  printk(KERN_ALERT "object number %u,\
                                        name = %d,\
305                                     pid = %d,\
                                        src_addr = %s,\
307                                     src_port = %s, \
                                        dst_addr = %s, \
309                                     dst_port = %s\n",\
                                        i, \
311                                     rtlkill_stream->stream_array[i].name, \
                                        rtlkill_stream->stream_array[i].pid, \
313                                     rtlkill_stream->stream_array[i].src_addr,\
                                        rtlkill_stream->stream_array[i].src_port, \
315                                     rtlkill_stream->stream_array[i].dst_addr, \
                                        rtlkill_stream->stream_array[i].dst_port);
317          }
     }
319
     /*
321   * Show Attributes declaration
      */
323
     static struct kobj_attribute rtlkill_show_attribute0 =
325          __ATTR(rtlkill_reader0, 0444, rtlkill0_show, NULL);

327  static struct kobj_attribute rtlkill_show_attribute1 =
             __ATTR(rtlkill_reader1, 0444, rtlkill1_show, NULL);
329
     static struct kobj_attribute rtlkill_show_attribute2 =
331          __ATTR(rtlkill_reader2, 0444, rtlkill2_show, NULL);

333  static struct kobj_attribute rtlkill_show_attribute3 =
             __ATTR(rtlkill_reader3, 0444, rtlkill3_show, NULL);
335
     static struct kobj_attribute rtlkill_show_attribute4 =
337          __ATTR(rtlkill_reader4, 0444, rtlkill4_show, NULL);

339  static struct kobj_attribute rtlkill_show_attribute5 =
             __ATTR(rtlkill_reader5, 0444, rtlkill5_show, NULL);
341
     static struct kobj_attribute rtlkill_show_attribute6 =
343          __ATTR(rtlkill_reader6, 0444, rtlkill6_show, NULL);

345  static struct kobj_attribute rtlkill_show_attribute7 =
             __ATTR(rtlkill_reader7, 0444, rtlkill7_show, NULL);
347
     static struct kobj_attribute rtlkill_show_attribute8 =
349          __ATTR(rtlkill_reader8, 0444, rtlkill8_show, NULL);

351  static struct kobj_attribute rtlkill_show_attribute9 =
             __ATTR(rtlkill_reader9, 0444, rtlkill9_show, NULL);
353
     static struct kobj_attribute rtlkill_monitor_attribute =
355          __ATTR(rtlkill_monitor, 0444, rtlkill_monitor_show, NULL);

357  /*
      * Store attribute declaration
359   */
     static struct kobj_attribute rtlkill_store_attribute =
361          __ATTR(rtlkill_register, 0666, rtlkill_register_show, rtlkill_store);
```

```
363 static struct kobj_attribute rtlkill_debug_attribute =
        __ATTR(rtlkill_debug, 0666, NULL, rtlkill_debug_store);
365
    /*
367  * Attribute groups
     */
369 static struct attribute * attrs[] = {
        &rtlkill_show_attribute0.attr,
371     &rtlkill_show_attribute1.attr,
        &rtlkill_show_attribute2.attr,
373     &rtlkill_show_attribute3.attr,
        &rtlkill_show_attribute4.attr,
375     &rtlkill_show_attribute5.attr,
        &rtlkill_show_attribute6.attr,
377     &rtlkill_show_attribute7.attr,
        &rtlkill_show_attribute8.attr,
379     &rtlkill_show_attribute9.attr,
        &rtlkill_monitor_attribute.attr,
381     &rtlkill_store_attribute.attr,
        &rtlkill_debug_attribute.attr,
383     NULL,
    };
385
    /*
387  * Group attribute somethingation
     */
389 static struct attribute_group attr_group = {
        .attrs = attrs,
391 };

393 /*
     * Kobject definition
395  */

397 static struct kobject * rtlkill_kobj;

399 /*
     * Parse ack information
401  */
    int ack_parser(char * buffer, const int pid, struct packet stack_result) {
403     int acktype;
        int fragtype;
405     bool flag = NULL;
        switch (stack_result.acked) {
407         case ACK_ERROR: {
                acktype = ACK_ERROR;
409             flag = FALSE;
                break;
411         }
            case ACK: {
413             acktype = ACK;
                flag = TRUE;
415             break;
            }
417         case ACK_NOT: {
                acktype = ACK_NOT;
419             flag = TRUE;
                break;
421         }
            case ACK_NOT_REQ: {
423             acktype = ACK_NOT_REQ;
                flag = FALSE;
425             break;
            }
```

```
427                 case ACK_FILTERED: {
                            acktype = ACK_FILTERED;
429                         flag = TRUE;
                            break;
431                 }
                    default: {
433                         acktype = stack_result.acked;
                            flag = FALSE;
435                 }
            }
437         switch (stack_result.fragment) {
                    case FRAG_LAST: {
439                         fragtype = FRAG_LAST;
                            break;
441                 }
                    case FRAG_NOT_LAST: {
443                         fragtype = FRAG_NOT_LAST;
                            break;
445                 }
                    default: {
447                         fragtype = stack_result.fragment;
                    }
449         }
            sprintf(buffer, "%d:%d:%d:%d:%d:%lu:%d", pid,
451                                                 stack_result.udp_port,
                                                    stack_result.ip_id,
453                                                 stack_result.retry_count,
                                                    fragtype,
455                                                 stack_result.filtered_count,
                                                    acktype);
457         return flag;
    }
459
    /*
461  * Kernel function invoked upon "something interesting happens" [lara]"
     */
463 int manage_ack(struct packet stack_result) {
            register int i;
465         int port;
            bool flag;
467         if (stack_result.acked == ACK_NOT_VALID) {
                    return -1;
469         }
            if (mutex_is_locked(&ack_mutex)) {
471                 printk(KERN_ALERT "sorry, but there is another instance \
                                       of the function manage_ack running, \
473                                    i cannot let you continue\n");
                    return -1;
475         } else {
                    /* locked = TRUE; */
477                 mutex_lock(&ack_mutex);
                    for (i = 0; i < MAXSTREAM; i++) {
479                         sscanf(rtlkill_stream->stream_array[i].src_port, "%d", &port);
                            if (stack_result.fragment == FRAG_NOT_LAST) {
481                                 was_frag = TRUE;
                            } else {
483                                 was_frag = FALSE;
                            }
485                         if (port == stack_result.udp_port || was_frag) {
                                    ack_parser(rtlkill_monitor,
487                                                rtlkill_stream->stream_array[i].pid,
                                                stack_result);
489                                 sysfs_notify(rtlkill_kobj, NULL, "rtlkill_monitor");
                                    /* do the rest, for each regarded file */
```

```
491                             switch (rtlkill_stream->stream_array[i].name) {
                                        case 0: {
493                                             ack_parser(rtlkill_reader0,
                                                        rtlkill_stream->stream_array[i].pid,
495                                                     stack_result);
                                                sysfs_notify(rtlkill_kobj, NULL, "rtlkill_reader0");
497                                             break;
                                        }
499                                     case 1: {
                                                ack_parser(rtlkill_reader1,
501                                                     rtlkill_stream->stream_array[i].pid,
                                                        stack_result);
503                                             sysfs_notify(rtlkill_kobj, NULL, "rtlkill_reader1");
                                                break;
505                                     }
                                        case 2: {
507                                             ack_parser(rtlkill_reader2,
                                                        rtlkill_stream->stream_array[i].pid,
509                                                     stack_result);
                                                sysfs_notify(rtlkill_kobj, NULL, "rtlkill_reader2");
511                                             break;
                                        }
513                                     case 3: {
                                                ack_parser(rtlkill_reader3,
515                                                     rtlkill_stream->stream_array[i].pid,
                                                        stack_result);
517                                             sysfs_notify(rtlkill_kobj, NULL, "rtlkill_reader3");
                                                break;
519                                     }
                                        case 4: {
521                                             ack_parser(rtlkill_reader4,
                                                        rtlkill_stream->stream_array[i].pid,
523                                                     stack_result);
                                                sysfs_notify(rtlkill_kobj, NULL, "rtlkill_reader4");
525                                             break;
                                        }
527                                     case 5: {
                                                ack_parser(rtlkill_reader5,
529                                                     rtlkill_stream->stream_array[i].pid,
                                                        stack_result);
531                                             sysfs_notify(rtlkill_kobj, NULL, "rtlkill_reader5");
                                                break;
533                                     }
                                        case 6: {
535                                             ack_parser(rtlkill_reader6,
                                                        rtlkill_stream->stream_array[i].pid,
537                                                     stack_result);
                                                sysfs_notify(rtlkill_kobj, NULL, "rtlkill_reader6");
539                                             break;
                                        }
541                                     case 7: {
                                                ack_parser(rtlkill_reader7,
543                                                     rtlkill_stream->stream_array[i].pid,
                                                        stack_result);
545                                             sysfs_notify(rtlkill_kobj, NULL, "rtlkill_reader7");
                                                break;
547                                     }
                                        case 8: {
549                                             ack_parser(rtlkill_reader8,
                                                        rtlkill_stream->stream_array[i].pid,
551                                                     stack_result);
                                                sysfs_notify(rtlkill_kobj, NULL, "rtlkill_reader8");
553                                             break;
                                        }
```

```
555                                     case 9: {
                                                ack_parser(rtlkill_reader9,
557                                                     rtlkill_stream->stream_array[i].pid,
                                                        stack_result);
559                                             sysfs_notify(rtlkill_kobj, NULL, "rtlkill_reader9");
                                                break;
561                                     }
                                }
563                             /* report success */
                                port = -1;
565                             flag = TRUE;
                        } else {
567                             flag = FALSE;
                        }
569             }
        }
571     /* return 0 in case of success, 1 in case of failure - not having found the stream.
         * it is not an error, just a signal for the registration logic */
573     /* locked = FALSE; */
        mutex_unlock(&ack_mutex);
575     return (flag == TRUE ? 0 : 1);
}
577 EXPORT_SYMBOL(manage_ack);

579
/*
581  * Main registration/deregistration function;
 */
583 int rederegistration(char * string, bool bitmap[]) {
        bool flag = TRUE;
585     register int i;
        int pos = 0 , actual ;
587     char strtmp[100];
        TOKENIZER;
589     if (strcmp(strtmp, "register") == 0) {
                for (i = 0; i<MAXSTREAM; i++) {
591                     if ((!bitmap[i]) && (flag)) {
                                TOKENIZER;
593                             strcpy(rtlkill_stream->stream_array[i].src_addr, strtmp);
                                TOKENIZER;
595                             strcpy(rtlkill_stream->stream_array[i].src_port, strtmp);
                                TOKENIZER;
597                             strcpy(rtlkill_stream->stream_array[i].dst_addr, strtmp);
                                TOKENIZER;
599                             strcpy(rtlkill_stream->stream_array[i].dst_port, strtmp);
                                rtlkill_stream->stream_array[i].name = i;
601                             rtlkill_stream->stream_array[i].pid = tmppid;
                                if (debug == 1) {
603                                     print_stream_table();
                                }
605                             bitmap[i] = TRUE;
                                flag = FALSE;
607                             sprintf(strtmp, "%s%d", "rtlkill_reader", i);
                                strcpy(rtlkill_register,strtmp);
609                     }
                }
611     } else if (strcmp(strtmp, "unregister") == 0) {
                for (i = 0; i< MAXSTREAM; i++) {
613                     if (rtlkill_stream->stream_array[i].pid == tmppid) {
                                switch (rtlkill_stream->stream_array[i].name) {
615                                     case 0: {
                                                strcpy(rtlkill_reader0, "\0");
617                                             break;
                                        }
```

```
619                                     case 1: {
                                                strcpy(rtlkill_reader1 , "\0");
621                                             break;
                                        }
623                                     case 2: {
                                                strcpy(rtlkill_reader2 , "\0");
625                                             break;
                                        }
627                                     case 3: {
                                                strcpy(rtlkill_reader3 , "\0");
629                                             break;
                                        }
631                                     case 4: {
                                                strcpy(rtlkill_reader4 , "\0");
633                                             break;
                                        }
635                                     case 5: {
                                                strcpy(rtlkill_reader5 , "\0");
637                                             break;
                                        }
639                                     case 6: {
                                                strcpy(rtlkill_reader6 , "\0");
641                                             break;
                                        }
643                                     case 7: {
                                                strcpy(rtlkill_reader7 , "\0");
645                                             break;
                                        }
647                                     case 8: {
                                                strcpy(rtlkill_reader8 , "\0");
649                                             break;
                                        }
651                                     case 9: {
                                                strcpy(rtlkill_reader9 , "\0");
653                                             break;
                                        }
655                                 }
                                    rtlkill_stream->stream_array[i].name = -1;
657                                 strcpy(rtlkill_stream->stream_array[i].src_addr , "\0");
                                    strcpy(rtlkill_stream->stream_array[i].src_port , "\0");
659                                 strcpy(rtlkill_stream->stream_array[i].dst_addr , "\0");
                                    strcpy(rtlkill_stream->stream_array[i].dst_port , "\0");
661                                 rtlkill_stream->stream_array[i].pid = -1;
                                    bitmap[i] = FALSE;
663                                 sprintf(strtmp, "%d unregistered\n" , tmppid);
                                    strcpy(rtlkill_register ,strtmp);
665                                 if (debug == 1) {
                                            print_stream_table();
667                                 }
                            }
669                     }
                    } else {
671                     printk(KERN_ALERT "WARNING: the application %u \
                                    tried to register with an unknown \
673                                 or wrong protocol, please correct\n",
                                    tmppid);
675                     return 1;
                    }
677                 /* kfree(strtmp); */
                    return 0;
679 }

681 /*
     * Kernel test thread
```

```
683  */
     static int test_thread(void * data) {
685          DECLARE_WAITQUEUE (wait, current);
             struct packet test_packets[7];
687          int i, random;
             for (i = 0; i < 7; i++) {
689                  if (!(i % 2)) {
                             test_packets[i].acked = ACK;
691                  } else {
                             test_packets[i].acked = ACK_NOT;
693                  }
                     test_packets[i].retry_count = i;
695                  if (i == 3) {
                             test_packets[i].filtered_count = 1;
697                  } else {
                             test_packets[i].filtered_count = 0;
699                  }
                     if (i % 2) {
701                          test_packets[i].fragment = 1;
                     } else {
703                          test_packets[i].fragment = 0;
                     }
705                  test_packets[i].ip_id = (1000 * i);
                     test_packets[i].udp_port = (500 + i);
707          }
             daemonize("test_thread");
709          allow_signal(SIGKILL);
             add_wait_queue (&debugfd_waitqueue, &wait);
711          set_current_state(TASK_INTERRUPTIBLE);
             while (!kthread_should_stop()) {
713                  if (signal_pending (current)) break;
                     if (it_shall_exit) {
715                          remove_wait_queue (&debugfd_waitqueue, &wait);
                             return 0;
717                  }
                     if (shallistop) {
719                          set_current_state(TASK_STOPPED);
                             schedule();
721                  }
                     for (i = 0; i < 7; i++) {
723                          manage_ack(test_packets[i]);
                             random = (40 % (i + 1)) + (4 * (i + 1));
725                          msleep(random);
                             test_packets[i].ip_id++;
727                  }
             }
729          set_current_state(TASK_RUNNING);
             remove_wait_queue (&debugfd_waitqueue, &wait);
731          return 0;
     }
733

735  /*
      * The kernel function to spawn as the kernel thread, let there be little big horn
737   */
     static int rtlkill_thread(void * data) {
739
             DECLARE_WAITQUEUE (wait, current);
741
             bool bitmap[MAXSTREAM];
743          register int i;
             printk (KERN_ALERT "Rtlkill thread started\n");
745          for (i = 0; i < MAXSTREAM; i++) {
                     bitmap[i] = FALSE;
```

```
747         }
            daemonize("krtlkill_reg");
749         allow_signal(SIGKILL);
            add_wait_queue (&threadfd_waitqueue, &wait);
751         while ( 1 ) {
                    /* we set the thread to be interruptible, we do NOT want a deadlock! */
753                 set_current_state(TASK_INTERRUPTIBLE);
                    schedule();
755                 if (signal_pending (current)) break;
                    if (it_shall_exit) {
757                         remove_wait_queue (&threadfd_waitqueue, &wait);
                            return 0;
759                 }
                    /* the main registration logic */
761                 rederegistration(rtlkill_register, bitmap);
            }
763         set_current_state(TASK_RUNNING);
            remove_wait_queue (&threadfd_waitqueue, &wait);
765         return 0;

767 }

769 /*
     * Kobject init function
771  */

773 static int rtlkill_init(void) {
            int retval;
775         bool memerr = FALSE;
            register int i;
777         mutex_init(&ack_mutex);
            printk(KERN_ALERT "RTLkill module loading\n");
779         rtlkill_kobj = kobject_create_and_add("rtlkill", kernel_kobj);
            if (!rtlkill_kobj) {
781                 return -ENOMEM;
            }
783         retval = sysfs_create_group(rtlkill_kobj, &attr_group);
            if (retval) {
785                 kobject_put(rtlkill_kobj);
            }
787         rtlkill_reader0 = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_reader0) {
789                 memerr = TRUE;
            }
791         rtlkill_reader1 = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_reader1) {
793                 memerr = TRUE;
            }
795         rtlkill_reader2 = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_reader2) {
797                 memerr = TRUE;
            }
799         rtlkill_reader3 = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_reader3) {
801                 memerr = TRUE;
            }
803         rtlkill_reader4 = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_reader4) {
805                 memerr = TRUE;
            }
807         rtlkill_reader5 = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_reader5) {
809                 memerr = TRUE;
            }
```

```
811         rtlkill_reader6 = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_reader6) {
813                 memerr = TRUE;
            }
815         rtlkill_reader7 = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_reader7) {
817                 memerr = TRUE;
            }
819         rtlkill_reader8 = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_reader8) {
821                 memerr = TRUE;
            }
823         rtlkill_reader9 = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_reader9) {
825                 memerr = TRUE;
            }
827         rtlkill_monitor = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_monitor) {
829                 memerr = TRUE;
            }
831         rtlkill_register = kmalloc(100 * sizeof(char), GFP_KERNEL);
            if (!rtlkill_register) {
833                     memerr = TRUE;
            }
835         if (debug == 1) {
                rtlkill_debug = kmalloc(20 * sizeof(char), GFP_KERNEL);
837             if (!rtlkill_debug) {
                        memerr = TRUE;
839             }
            }
841         /* stream table initialization */
            rtlkill_stream = kmalloc(sizeof(struct stream_table), GFP_KERNEL);
843         for (i = 0; i < MAXSTREAM; i++) {
                rtlkill_stream->stream_array[i].src_addr = kmalloc(16 * sizeof(char), GFP_KERNEL);
845             if (!rtlkill_stream->stream_array[i].src_addr) {
                        memerr = TRUE;
847             }
                rtlkill_stream->stream_array[i].src_port = kmalloc(6 * sizeof(char), GFP_KERNEL);
849             if (!rtlkill_stream->stream_array[i].src_port) {
                        memerr = TRUE;
851             }
                rtlkill_stream->stream_array[i].dst_addr = kmalloc(16 * sizeof(char), GFP_KERNEL);
853             if (!rtlkill_stream->stream_array[i].dst_addr) {
                        memerr = TRUE;
855             }
                rtlkill_stream->stream_array[i].dst_port = kmalloc(6 * sizeof(char), GFP_KERNEL);
857             if (!rtlkill_stream->stream_array[i].dst_port) {
                        memerr = TRUE;
859             }

861             rtlkill_stream->stream_array[i].name = -1;
                /* "and you don't want to end up in the middle of invalid memory" (cit)*/
863             strcpy(rtlkill_stream->stream_array[i].src_addr, "\0");
                strcpy(rtlkill_stream->stream_array[i].src_port, "\0");
865             strcpy(rtlkill_stream->stream_array[i].dst_addr, "\0");
                strcpy(rtlkill_stream->stream_array[i].dst_port, "\0");
867             rtlkill_stream->stream_array[i].pid = -1;
            }
869         /* we start up the kernel thread */
            threadfd = kthread_run(rtlkill_thread, NULL, "krtlkill_reg");
871         it_shall_exit = FALSE;
            if (memerr) {
873                 return -1;
            }
```

```
875
        return 0;
877 }

879 static void rtlkill_exit(void) {
        int retval;
881     register int i;
        for (i = 0; i < MAXSTREAM; i++) {
883             kfree(rtlkill_stream->stream_array[i].src_addr);
                kfree(rtlkill_stream->stream_array[i].src_port);
885             kfree(rtlkill_stream->stream_array[i].dst_addr);
                kfree(rtlkill_stream->stream_array[i].dst_port);
887     }
        kfree(rtlkill_stream);
889     kfree(rtlkill_reader0);
        kfree(rtlkill_reader1);
891     kfree(rtlkill_reader2);
        kfree(rtlkill_reader3);
893     kfree(rtlkill_reader4);
        kfree(rtlkill_reader5);
895     kfree(rtlkill_reader6);
        kfree(rtlkill_reader7);
897     kfree(rtlkill_reader8);
        kfree(rtlkill_reader9);
899     kfree(rtlkill_monitor);
        kfree(rtlkill_register);
901     if (debug == 1) {
                kfree(rtlkill_debug);
903     }
        it_shall_exit = TRUE;
905     retval = kthread_stop(threadfd);
        if (debug == 1) {
907             if (!thread_lock) {
                        retval = kthread_stop(debugfd);
909             }
        }
911     printk(KERN_ALERT "Rtlkill thread stopped\n");
        kobject_put(rtlkill_kobj);
913     mutex_destroy(&ack_mutex);
        printk(KERN_ALERT "RTLkill module unloaded\n");
915 }

917 module_init(rtlkill_init);
    module_exit(rtlkill_exit);
```