ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

OTTIMIZZAZIONE, GUIDATA DALL'APPLICAZIONE, DELLA SCANSIONE DI CANALI WIFI IN KERNEL LINUX

Relatore:
Chiar.mo Prof.
Vittorio Ghini

Presentata da:
Federico Zappi

Sessione III
Anno Accademico 2012/2013

Indice

1 INTRODUZIONE

La tecnologia Wi-Fi permette la comunicazione tra dispositivi elettronici per mezzo di onde radio. La comunicazione si svolge su un certo numero di frequenze, anche dette canali. I wireless access point usano Wi-Fi permettendo a dispositivi elettronici di collegarsi a loro senza fili e connettendoli a una rete cablata. Questi access point operano su un determinato canale, e access point nella stessa area dovrebbero usare canali diversi per non generare troppo traffico nei canali.
Per individuare gli access point nell'area, il dispositivo invia un messaggio su ogni canale; gli access point ricevono questo messaggio e rispondono notificando così la propria presenza. Il procedimento di scoperta degli access point è detto scansione Wi-Fi.

Nel kernel Linux, il codice che effettua la scansione Wi-Fi è presente nel modulo net/mac80211. In questa tesi si è cercato di modificare tale codice, con lo scopo di rendere il metodo di scansione variabile a seconda di esigenze di applicazioni utente. Il lavoro svolto è consistito nella creazione di un'applicazione che potesse comunicare con il kernel in modo da indicargli preferenze riguardo a come effettuare la scansione.

Il motivo dietro a questo lavoro è il fatto che alcune operazioni di scansione possono non essere adatte in particolari situazioni, le quali sono conosciute solo a livello applicazione. L'applicazione creata si propone quindi come strumento tramite il quale è possibile ottimizzare la scansione dei canali Wi-Fi nelle circostanze particolari note alle applicazioni.

Nel capitolo Scenario viene descritta la tecnologia Wi-Fi e la gestione della scansione del kernel Linux.
Nel capitolo Obbiettivo si esprime con precisione ciò che l'applicazione creata rende disponibile e lo scopo del progetto.
Nel capitolo Strumenti vengono descritti i socket netlink, che sono stati utilizzati per far comunicare l'applicazione e il kernel.
Nel capitolo Progettazione viene descritto come il lavoro di sviluppo del codice (creazione

dell'applicazione e modifiche nel kernel) è stato svolto, senza entrare nei particolari del codice.

Nel capitolo Note implementative si descrive in dettaglio come il lavoro finale funziona, mostrando anche alcune parti di codice.

Nel capitolo Valutazioni si esprimono considerazioni sul risultato del progetto.

Nel capitolo Conclusioni e sviluppi futuri viene data una valutazione finale del lavoro e si indica i punti che dovranno essere osservati per uno sviluppo futuro.

2 SCENARIO

2.1 Wi-Fi

Wi-Fi è una tecnologia che permette a un dispositivo elettronico di scambiare dati o connettersi ad Internet mediante l'uso di onde radio.
Molti dispositivi possono sfruttare la tecnologia Wi-Fi, per esempio computer, console di videogiochi, smartphone, alcune videocamere digitali, tablet e lettori di musica digitali. Questi oggetti possono connettersi a una rete, come per esempio Internet, attraverso il collegamento con un access point.

L'utilizzo di Wi-Fi può essere meno sicuro rispetto all'uso di una connessione cablata (come ad esempio Ethernet) perché un intruso non necessita di una connessione fisica per spiare la comunicazione. Le pagine web che usano SSL sono sicure, ma accessi ad internet non protetti sono facilmente individuabili dagli intrusi. Per questo motivo Wi-Fi ha adottato varie tecnologie di crittografia. WEP, il primo metodo di protezione, è risultato debole e altri meccanismi più sicuri sono stati aggiunti in seguito (WPA, WPA2).

Per connettersi a una rete locale mediante Wi-Fi un computer deve essere provvisto di un'interfaccia di rete wireless. La combinazione di un computer e un'interfaccia è chiamata stazione. Tutte le stazioni condividono una frequenza radio usata come canale di comunicazione. Le trasmissioni su questo canale sono ricevute da tutte le stazioni in portata. L'hardware non segnala quando una trasmissione è andata a buon fine e pertanto è chiamato un meccanismo di consegna best-effort. Ogni stazione è costantemente in ascolto sul canale di comunicazione per ricevere le trasmissioni disponibili.

Un dispositivo che fa uso di Wi-Fi può connettersi ad Internet quando si trova in portata di una rete wireless che è configurata in modo tale da rendere disponibile questo servizio. Router che comprendono un digital suscriber line modem o un cable modem e un access point Wi-Fi, spesso presente in case e altri edifici, forniscono accesso ad Internet e

internetworking a tutti i dispositivi connessi a loro, sia con cavo che senza.
Wi-Fi permette anche comunicazioni dirette tra un computer e un altro senza bisogno dell'utilizzo di un access point. Questa è chiamata connessione ad hoc. Alcuni dispositivi possono anche condividere la loro connessione internet usando la connessione ad hoc, diventando così hotspot o "router virtuali".

Le connessioni Wi-Fi possono subire disturbi o la velocità della rete può essere rallentata in presenza di altri dispositivi Wi-Fi nella stessa area. Molti access point usano un canale di default al momento del primo utilizzo, contribuendo alla congestione in certi canali. Wi-Fi pollution, ovvero un eccessivo numero di access point presenti nella stessa area, può prevenire l'accesso e interferire con altri dispositivi che utilizzano altri access point. Questo può diventare un problema in aree ad alta densità, come un grosso complesso di appartamenti o edifici di lavoro con molti Wi-Fi access point.
Inoltre altri dispositivi fanno uso della banda di frequenza utilizzata da Wi-Fi: forni a microonde, dispositivi ZigBee, dispositivi Bluetooth, telefoni portatili, baby monitor, e altri.

Passando invece ai vantaggi di usare un collegamento Wi-Fi, questa tecnologia permette una installazione economica di reti locali. Inoltre posti dove non è possibile avere una connessione cablata, come spazi esterni e alcuni monumenti storici, possono invece ospitare una rete locale wireless.

I produttori di computer stanno inserendo adattatori di rete wireless in molti portatili. Il prezzo del chipset per Wi-Fi continua a diminuire, rendendola una opzione di collegamento di rete presente in sempre più dispositivi.

Access point di differenti marche possono interagire a un livello base di servizio. Diversamente dai telefonini, un qualsiasi dispositivo Wi-Fi standard è in grado di funzionare in qualsiasi punto del mondo.

Il meccanismo di protezione WPA2 è considerato sicuro, ammesso che venga usata una password adeguata. I nuovi protocolli di qualità del servizio (WMM) rendono Wi-Fi più adatta a essere utilizzata per applicazioni sensibili alla latenza (per esempio che lavorano con audio e video). Meccanismi di conservazione dell'energia (WMM Power Save)

estendono la durata della batteria.

Il numero di canali utilizzati nella comunicazione Wi-Fi non è uguale per tutti i paesi: l'Australia e l'Europa permettono due ulteriori canali rispetto a quelli permessi negli Stati Uniti (1-13 vs. 1-11), mentre il Giappone ne ha un altro in più (1-14).

Un segnale Wi-Fi occupa cinque canali nella banda di 2.4 Ghz. I numeri che differiscono di 5 o più (come 2 e 7) non si sovrappongono. In Europa e in Giappone l'uso dei canali 1, 5 9 e 13 è raccomandato.

2.2 Wireless Access Point

Un wireless access point (AP) è un dispositivo che permette dispositivi wireless di connettersi a una rete cablata usando Wi-Fi. L'AP solitamente si connette a un router (attraverso una rete cablata), ma può anche essere un componente integrale del router.

Molti AP supportano la connessione di multipli dispositivi wireless a una rete con cavo. Un tipico uso aziendale comprende l'installazione di diversi AP a una rete con cavo per fornire un accesso wireless alla rete locale dell'ufficio.

Una hotspot è un uso comune degli AP, con il quale dispositivi wireless possono connettersi a Internet senza considerare la particolare rete alla quale sono connessi al momento. Il concetto è diventato comune in grosse città, dove una combinazione di coffehouses, librerie, così come access point pubblici di proprietà di privati, permettono agli utenti di stare più o meno continuamente connessi a Internet, mentre sono in movimento.

Un AP può agire come arbitro della rete, decidendo quando un device a lui connesso è abilitato a trasmettere. Tuttavia la maggior parte delle reti wireless IEEE 802.11 correnti non implementano questa funzionalità, usando invece un algoritmo pseudo-random chiamato CSMA/CA.

Un 802.11 AP può tipicamente comunicare con 30 dispositivi presenti entro un raggio di circa 100 m da lui. Tuttavia, la portata reale dell'access point può variare significativamente, a seconda di variabili come luogo esterno o interno, altezza rispetto al terreno, ostruzioni

vicine, altri dispositivi che agiscono sulla stessa frequenza (come detto precedentemente), tipo di antenna, tempo meteorologico, frequenza sulla quale si opera, potenza dei dispositivi. La portata degli AP può essere estesa tramite l'uso di ripetitori, i quali possono far rimbalzare o amplificare i segnali radio che normalmente non sarebbero ricevuti.

2.3 Scansione dei canali Wi-Fi

Per effettuare una connessione Wi-Fi con un wireless access point, un dispositivo deve prima sapere della sua esistenza. Per venire a conoscenza degli AP con i quali può dialogare si effettua la scansione, cioè l'ascolto dei vari canali Wi-Fi.

Esistono due metodi per effettuare una scansione: scansione attiva e scansione passiva. La scansione attiva consiste nell'inviare un messaggio in broadcast sul canale scelto e attendere le risposte dei vari AP. La scansione passiva consiste invece nell'attendere questi messaggi sul canale scelto. Questi messaggi di risposta che gli AP inviano vengono infatti inviati costantemente ogni 100ms, ma un dispositivo può sollecitare l'AP a inviarlo anticipatamente (aspettando un tempo di circa 10ms).

2.4 Scansione Wi-Fi nel kernel Linux 3.6.8

2.4.1 Gestione della scansione

Quando l'interfaccia di rete diventa abilitata, il kernel di linux comincia a effettuare le scansioni dei canali per individuare gli access point nell'area. Queste scansioni sono inizialmente effettuate in tempi piuttosto vicini le une alle altre, ma per ogni scansione successiva è atteso sempre più tempo prima di effettuarla nuovamente fino ad avere una scansione ogni due minuti.

Non viene sempre effettuato lo stesso metodo di scansione, ma ci sono tre possibilità. Un primo controllo viene fatto riguardo alla possibilità di effettuare una scansione hardware. Per capire se la scansione hardware è disponibile si controlla se la funzione *hw_scan* è stata definita.

Se l'hardware/driver non supporta tale scansione, successivamente si controlla se bisogna analizzare il solo canale corrente, cioè il canale con il quale si è attualmente connessi a un access point. In questo caso la scansione effettuata è quella passiva e non c'è bisogno di bloccare il traffico di rete.

I canali che devono essere analizzati sono elencati in una struttura (*struct cfg80211_scan_request*) che contiene i dati della richiesta di scansione, e anche alcuni campi di informazione da impostare in seguito. Questa struttura viene passata come argomento alla funzione che viene invocata quando si richiede di voler fare una scansione (*ieee80211_scan*) ed è successivamente passata come argomento anche alla funzione che effettua la decisione su come effettuare la scansione (*__ieee80211_start_scan*), che è la funzione che effettua i controlli qui descritti.

Se i due controlli precedenti non sono andati a buon fine viene effettuata una normale scansione (scansione software). Questo metodo effettua una scansione attiva e la trasmissione e ricezione di pacchetti durante la procedura è (parzialmente) disabilitata.

2.4.2 Metodo di scansione software

Quando c'è bisogno di effettuare questo tipo di scansione il dispositivo potrebbe essere o meno connesso a un access point. Nel caso in cui il dispositivo sia già connesso, una scansione di tutti i canali potrebbe interferire con eventuali trasmissioni da e verso l'acces point. I pacchetti in coda di trasmissione subirebbero un ritardo di consegna e i pacchetti in arrivo non potrebbero essere ricevuti, con eventuale loro perdita.

Per evitare di bloccare le comunicazioni, per tutta la durata della scansione si alterna la scansione dei canali con la ricezione e l'invio dei pacchetti. Durante il ciclo di scansione, ogni volta che si sta per analizzare un nuovo canale, si verifica se è il caso di continuare o interrompere la scansione in base a certi parametri che dovrebbero garantire una certa qualità del servizio.

Per evitare di perdere i pacchetti in arrivo, il kernel sfrutta il sistema di Power Save Mode (PSM). Quando è il momento di effettuare la scansione, il dispositivo invia un pacchetto con un bit PSM impostato a 1, segnalando all'access point che il dispositivo sta entrando in

Sleep Mode (anche se non ci andrà realmente). In questo modo l'access point conserverà in un buffer interno i pacchetti destinati al dispositivo per un noto periodo di tempo, dopo il quale non è più garantito che i pacchetti siano ancora memorizzati. Quando si decide di effettuare questa operazione vengono anche bloccate le code di trasmissione in uscita.

Quando la scansione viene interrotta (o terminata), il dispositivo comunica all'access point che ha terminato la Sleep Mode (tramite il bit PSM) e sblocca le code di trasmissione, permettendo nuovamente le comunicazioni in entrambi i sensi.

3 OBBIETTIVO

L'obbiettivo di questo lavoro di tesi è stato quello di creare un'applicazione a livello utente in grado di fornire controllo da parte di quest'ultimo sul comportamento della scansione dei canali Wi-Fi del kernel Linux 3.6.8. Perchè questa scelta?

In principio si era pensato che la scansione effettuata dal kernel (quella software) venisse eseguita senza interruzioni, cioè senza essere momentaneamente bloccata dal bisogno di inviare o di ricevere pacchetti per altri motivi, e l'obbiettivo principale era quello di fare in modo che ciò fosse possibile. In seguito, tramite la ricerca di un altro studente, è emerso che la scansione effettuata dal kernel sembrava comportarsi proprio in quel modo, anche se alcuni dubbi sono rimasti. Tuttavia si è pensato che potesse essere utile aggiungere altre funzionalità.

Le funzionalità a cui successivamente si è puntato sono:

- dare priorità alla scansione rispetto alla ricezione e all'invio di altri pacchetti (ovvero evitare che la scansione venga interrotta);
- disattivazione/riattivazione della scansione;
- ricezione di notifiche dal kernel a seguito della terminazione di una scansione completa.

Queste funzioni permettono di adattare la funzionalità di scansione del kernel in base alle esigenze di chi utilizza il sistema, in particolare i primi due punti. La parola ottimizzazione nel titolo della tesi si riferisce a questa proprietà di adattamento.

La funzionalità di effettuare una scansione senza interruzioni permette una scansione più rapida, da utilizzare nel caso si consideri più importante la riuscita di una scansione in poco tempo che la buona qualità delle trasmissioni. Questo potrebbe servire nel caso in cui un dispositivo voglia effettuare scansioni molto frequenti e non abbia problemi a subire ritardi nella comunicazione. Questa funzionalità è presente solo per la scansione software.

La funzionalità di disattivazione della scansione permette di evitare il periodico controllo del sistema e eventuali altre richieste di scansioni. In questo caso non viene bloccata solamente la scansione software, ma ogni scansione consentita. Da usare nel caso si pensi di non aver bisogno di effettuare scansioni Wi-Fi.

L'ultima funzionalità permette di venire a conoscenza della terminazione di una qualsiasi scansione software, non appena si raggiunge la terminazione della scansione. Gli altri due casi di scansione non vengono notificati.

Senza un controllo su come effettuare la scansione Wi-Fi da parte delle applicazioni, cioè senza avere la possibilità di adattare il comportamento della scansione a seconda delle esigenze richieste da esse, si possono rischiare di avere due effetti deleteri:

- da un lato effettuare scansioni non necessarie che sprecano energia (ad esempio nel caso il nodo mobile sia fermo la scansione potrebbe essere dilazionata);
- dall'altro lato effettuare scansioni troppo separate nel tempo e quindi avere informazioni non più attendibili sulla presenza di access point.

In sostanza, in questo lavoro si vuole mettere a disposizione del livello applicazione un modo per determinare con precisione il funzionamento della scansione dei canali Wi-Fi in funzione di condizioni (posizione, velocità, applicazioni in esecuzione) che possono essere note solo a livello applicazione.
Infine, bisogna mettere in evidenza che l'applicazione a cui si vuole fornire il controllo sulle scansioni non sarà una qualunque applicazione bensì una sorta di manager delle interfacce di rete, con permessi di amministratore di sistema. Ciò impedisce che più applicazioni possano dare ordini contrastanti al kernel riguardo alle scansioni da effettuare.

4 STRUMENTI

4.1 Netlink socket

I socket netlink sono speciali IPC (Inter-Process Communication) usati per trasferire informazioni tra kernel e processi a livello utente. Questi socket forniscono un canale di comunicazione full-duplex, cioè un canale che permette lo scambio di messaggi in contemporanea da entrambe le parti. Per l'utilizzo dei socket netlink si fa uso delle API standard dei socket, per quanto riguarda il codice dei processi a livello utente. Una speciale API è dedicata invece ai moduli del kernel. I socket netlink utilizzano l'address family AF_NETLINK, così come AF_INET è usato per i socket TCP/IP.

I socket netlink offrono diverse funzionalità: nel file header include/linux/netlink.h sono definiti una ventina di canali che permettono di comunicare con differenti moduli kernel o gruppi netlink. Alcuni esempi:

- NETLINK_ROUTE: canale di comunicazione tra user-space routing daemons, come BGP, OSPF, RIP e il modulo del kernel di forward dei pacchetti. I daemons aggiornano la tabella di routing del kernel attraverso questo protocollo netlink.
- NETLINK_FIREWALL: riceve pacchetti inviati attraverso IPv4 firewall code.
- NETLINK_NFLOG: canale di comunicazione per user-space iptable management tool e modulo Netfilter del kernel.
- NETLINK_ARPD_ per gestire la tabella arp da livello utente.

E' possibile aggiungere nuovi canali.

I socket netlink operano in maniera asincrona perché, come con qualsiasi altra API SOCKET, forniscono una coda per gestire eventuali grandi quantità di messaggi inviati in un piccolo intervallo di tempo (burst of messages). La system call per inviare un messaggio netlink inserisce il messaggio nella coda netlink del ricevente e quindi invoca il gestore dei messaggi in arrivo del ricevente. Questo può decidere se gestire subito il messaggio o se lasciarlo nella coda e rimandare l'elaborazione a un momento più opportuno.

Diversamente da netlink, le system call richiedono una elaborazione sincrona; se quindi si usa una system call per inviare un messaggio da spazio utente a kernel, il processo di scheduling del kernel potrebbe subire ripercussioni nel caso in cui il tempo di gestione del messaggio fosse lungo.

Il codice di una system call è linkato staticamente al kernel a tempo di compilazione: per questo motivo non è appropriato includere il codice di una system call in un modulo caricabile, che è il caso per molti driver di dispositivi. Con i socket netlink non esiste nessuna dipendenza dal tempo di compilazione tra il nucleo netlink del kernel e gli usi di netlink nei moduli caricabili.

I socket netlink supportano l'invio di un messaggio in multicast, che e' un altro beneficio rispetto a system call, ioctl e proc. Un processo può inviare un messaggio in multicast a un indirizzo netlink di gruppo e un qualsiasi numero di altri processi può mettersi in ascolto su quell'indirizzo di gruppo. Questo fornisce un buon meccanismo per la distribuzione degli eventi da kernel a spazio utente.

System call e ioctl sono semplici IPC, nel senso che una sessione per questi IPC può essere iniziata solo da applicazioni in spazio utente. Tuttavia, nel caso il kernel avesse un messaggio urgente da distribuire a una applicazione, non c'è modo di farlo direttamente utilizzando questi metodi. Solitamente le applicazioni hanno bisogno di fare poll sul kernel per individuare i cambiamenti di stato, sebbene il polling sia un'operazione costosa. Netlink permette al kernel di iniziare una sessione.

5 PROGETTAZIONE

Come prima cosa si è installato il kernel 3.6.8 sulla macchina, quindi si è verificato il comportamento della scansione inserendo alcune printk nel codice del kernel dove viene effettuata la scansione (contenuto in net/mac80211). Guardando i risultati delle printk si è visto che la scansione effettuata era la scansione software, e che le scansioni venivano effettuate periodicamente, con tempi di attesa più corti in seguito all'abilitazione del dispositivo wireless che poi si allungavano per diventare tempi di circa due minuti di attesa.

In seguito quello che si è tentato di ottenere è stato inviare correttamente un messaggio dall'applicazione al kernel, che era la cosa fondamentale per la realizzazione del progetto. L'utilizzo del socket netlink non è stato molto problematico, ma la documentazione riguardo al suo utilizzo non è stata facilmente ottenibile. Si è inizialmente fatto riferimento a un articolo su internet, il quale descriveva molto bene come utilizzare i socket, ma la parte di codice kernel-side di netlink non funzionava nel kernel 3.6.8, riportando degli errori di compilazione. Il fatto era che il codice di netlink era cambiato rispetto al codice dell'articolo, il quale non è riportato a che versione del kernel appartenga.

Dopo un po di altre ricerche all'interno del codice netlink del kernel 3.6.8 e su internet si è riuscito a capire come utilizzare netlink. Si è quindi riuscito a creare la comunicazione tra kernel e applicazione, inserendo nel kernel una funzione da chiamare ogniqualvolta il kernel avesse ricevuto un messaggio sul canale netlink dedicato. E' stato infatti aggiunto un nuovo canale netlink modificando il file di inclusione linux/netlink.h.
L'applicazione iniziale consisteva semplicemente nel prendere il primo argomento passato al programma al suo avvio da terminale, inviarlo al kernel e terminare l'esecuzione. Il kernel riceveva il messaggio e lo stampava con una printk.

Successivamente si è cercato di fare in modo che il messaggio “scan_off” disabilitasse la scansione e che il messaggio “scan_on” la riattivasse. La ricezione dei messaggi causava la modifica dello stato di una variabile intera, il cui valore veniva controllato all'interno della

funzione *ieee80211_scan* (contenuta in cfg.c), che è la funzione invocata quando viene richiesta una scansione.
Inizialmente si è tentato di interrompere la scansione con un "return 0;". Il valore 0 era stato deciso cercando di capire che valore di ritorno dovesse restituire la funzione in caso di terminazione normale della scansione. Tuttavia si è verificato un problema....

Il problema consisteva nel fatto che, una volta disattivata la scansione, non si riusciva più a riattivare la scansione. In particolare non si ricevevano più chiamate alla funzione *ieee80211_scan* (non compariva il testo della printk che informava dell'inizio di esecuzione della scansione, posta all'inizio di essa).
La causa del problema non è stata approfondita, ma era chiaro che restituire un valore di ritorno non era sufficiente, ma che bisognava modificare qualche altra struttura o variabile, o chiamare una qualche funzione che lo facesse.

In seguito si è trovata una funzione che notificava la terminazione di una scansione, con l'opzione per indicare se la scansione era stata annullata per un qualsiasi motivo; si è inserita questa funzione subito prima del "return 0;" e in questo modo si è riuscito ad ottenere la riattivazione della scansione.

Il blocco di una richiesta di scansione effettuato nella funzione *ieee80211_scan* non blocca soltanto il metodo di scansione software, ma anche gli altri due casi (scansione hardware e scansione passiva sul canale attuale). Infatti la funzione che decide che tipo di scansione effettuare è chiamata attraverso l'esecuzione di questa funzione.

La variabile che indicava se la scansione fosse abilitata o meno era inizialmente contenuta in cfg.c ed era una variabile privata che veniva modificata attraverso un metodo pubblico. Questo metodo pubblico veniva invocato da main.c.
In seguito, con l'aggiunta delle altre funzionalità e quindi delle altre variabili, si è deciso di spostare le variabili in main.c. Infatti le variabili si trovavano in due file diversi (scan.c e cfg.c) e main.c doveva includere i file di inclusione dei due che contenevano la dichiarazione dei metodi per modificare le variabili.

Raggruppando le variabili in main.c, i cambiamenti sono stati che i metodi per modificare le variabili sono stati eliminati e le variabili sono state rese pubbliche, in quanto gli altri file dovevano referenziarle. E' stato quindi creato un file di inclusione che contiene le dichiarazioni delle variabili, da includere nei file scan.c e cfg.c.

Per ogni variabile esistono due messaggi che, se ricevuti dal kernel (dalla funzione di gestione dei messaggi netlink inserita in main.c), ne alterano il valore. L'unica cosa da fare per cambiare il comportamento della scansione è impostare il valore di questi “indicatori”. In base al loro valore vengono effettuate e/o saltate certe istruzioni nel codice.

La funzionalità per evitare che la scansione software venisse interrotta è stata semplice da inserire. Quello che è stato fatto è stato controllare il valore della variabile che indica se la scansione deve proseguire senza interruzioni e nel caso la scansione non dovesse essere interrotta evitare il controllo per garantire la qualità del servizio (con possibile perdita di pacchetti all'access point) e continuare la scansione come se non ci fossero comunicazioni in corso.

L'ultima funzionalità, l'invio di notifiche all'applicazione a seguito della terminazione di una scansione (software), richiedeva l'invio di un messaggio netlink dal kernel verso l'applicazione, ma quale applicazione?

E' infatti possibile che più applicazioni vogliano essere informate di questo evento. Per inviare il messaggio a tutte le applicazioni in ascolto si è tentato di effettuare l'invio del messaggio in multicast, cioè l'invio a un indirizzo di gruppo sul quale le applicazioni si sarebbero dovute mettere in ascolto.
Purtroppo i tentativi per l'invio di un messaggio in multicast sono stati negativi, e dopo qualche giorno si è deciso di effettuare un invio a una singola applicazione. Nel codice di ricezione dei messaggi netlink, nel caso in cui il messaggio fosse la richiesta di invio di notifiche, si è memorizzato in una variabile l'identificativo del processo mittente; il valore di questa variabile determina a quale processo deve essere inviata la notifica.

La decisione di inviare il messaggio in questo modo, tuttavia, comporta un potenziale problema. L'invio di notifiche funziona correttamente se le applicazioni si alternano tra di loro sulla ricezione di questi messaggi, ma se due applicazioni si mettono in ascolto in contemporanea la notifica arriverà solamente all'applicazione il cui messaggio sarà stato gestito per ultimo, in quanto la variabile che conterrà l'identificativo del processo a cui inviare la notifica, precedentemente impostato con l'identificativo dell'altro processo, conterrà il suo identificativo e perderà l'identificativo dell'altro.

Quello che si è detto sopra vale per l'applicazione progettata per questa tesi. L'applicazione infatti, quando viene lanciata da terminale senza il passaggio di argomenti, invia uno specifico messaggio al kernel (in cui chiede l'invio di notifiche) e quindi si mette in ascolto sul canale dedicato, attendendo continuamente messaggi dal kernel.
L'invio di richiesta di notifiche, quindi, è effettuato solamente una volta. Se l'applicazione attende un qualsiasi numero di minuti senza ricevere notifiche non se ne preoccupa e continua ad attendere invano.

Tornando a parlare del kernel, durante la progettazione dell'invio di un messaggio dal kernel all'applicazione c'erano stati altri errori, oltre a quelli dovuti ai tentativi di inviare un messaggio in multicast, i quali tuttavia si è riuscito a risolvere. Il problema era stato principalmente un errore di comprensione dell'utilizzo del socket netlink.
Quello che avevo tentato di fare era stato creare un nuovo socket netlink, inviare un messaggio e chiuderlo. Tuttavia la creazione del nuovo socket netlink falliva e ho dovuto inserire qualche printk nel codice di netlink per scoprire l'errore. Questo problema viene discusso più avanti nelle note implementative.

Trovato il problema, ho deciso di usare il socket netlink già creato (quello usato per la ricezione dei messaggi, in main.c) anche per l'invio di notifiche. Come si è già detto nel capitolo “Strumenti”, infatti, i socket netlink forniscono un canale di comunicazione full duplex, quindi l'invio di messaggi in contemporanea da entrambi gli estremi non comporta problemi.

Il motivo per cui avevo tentato di creare un nuovo socket è stato perché volevo inserire il codice per l'invio di notifiche nel file scan.c, che è il file dove si rileva la terminazione della scansione.
Alla fine, dopo alcune modifiche, si è inserito il codice per l'invio di notifiche nel file main.c. La dichiarazione della funzione è stata inserita nel file di inclusione dove già erano state inserite le variabili. Terminata la scansione, in scan.c, viene quindi chiamato il metodo di invio di modifiche (che abbiamo detto si trova in main.c).

L'esecuzione della funzione di invio di notifiche, a seguito di problemi precedenti, inizialmente veniva effettuata in un nuovo thread. Si pensava infatti che l'esecuzione di questa funzione avrebbe potuto provocare dei problemi nel kernel in seguito al ritardo di terminazione della funzione di scansione.
Quando infine si è terminata la scrittura del codice di invio si è provato a rimuovere la creazione del thread e si è visto che (apparentemente) ciò non comportava problemi.

Riassumiamo il funzionamento dell'applicazione e il funzionamento della parte di codice definita nel kernel.

L'applicazione si comporta in due modi, che dipende dal numero di argomento che le vengono passati all'avvio da terminale:

- 1 o più argomenti: l'applicazione tenta di inviare il primo argomento passatogli come messaggio al kernel. Il messaggio viene inviato anche se la sua ricezione dal kernel potrebbe non avere conseguenze sul funzionamento della scansione. Al termine dell'invio del messaggio l'applicazione termina.
- nessun argomento: l'applicazione invia un messaggio predefinito che comunica al kernel di inviargli le notifiche di fine scansione. L'applicazione rimane quindi in ascolto sul canale netlink in attesa di ricevere tali notifiche. Ogni volta che riceve una notifica si rimette in ascolto sul canale.

Il comportamento del kernel:

- In fase di inizializzazione del modulo mac80211 viene creato un socket netlink sul canale dedicato, registrando la funzione di gestione dei messaggi in arrivo in modo che ogni volta che venga ricevuto un messaggio sul canale netlink dedicato venga richiamata tale funzione.
- Quando la funzione di gestione viene eseguita controlla il messaggio ricevuto e cambia il valore di alcune variabili, eventualmente facendo operazioni aggiuntive (memorizzare il pid del processo mittente, nel caso del messaggio delle notifiche).
- La variazione del valore di queste variabili influisce sul comportamento della scansione, in quanto sono state aggiunti controlli sul valore di queste variabili che evitano le istruzione che verrebbero normalmente eseguite e/o ne eseguono delle nuove.

6 NOTE IMPLEMENTATIVE

6.1 Applicazione

6.1.1 main

Nella funzione principale del programma la prima cosa che si verifica è il valore di *argc*, cioè il numero di argomenti passati al programma. Un argomento c'è sempre ed è il nome del programma, quindi si controlla se il valore di *argc* è maggiore di 1, per sapere se è stato passato il messaggio da inviare.
In caso positivo quello che si fa è chiamare la funzione per inviare il messaggio al kernel (*send_message*) che prende come parametro una stringa, cioè *argv[1]*.
In caso negativo viene inviato al kernel il messaggio per farsi inviare le notifiche ("scan_notification_on"), sempre utilizzando *send_message,* per poi entrare in un ciclo infinito (loop) dove si effettua una *recvmsg* sul socket netlink, il quale è definito globalmente e viene creato dalla funzione di invio del messaggio.

```
puts("Waiting for end of scan messages");
do {
        memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));
        recvmsg(sock_fd, &msg, 0);
        printf("Received message payload: %s\n", (char *)NLMSG_DATA(nlh));
}while(1);
```

Le variabili *nlh* e *msg* sono anch'esse definite globalmente; *nlh* è un puntatore a una struct nlmsghdr mentre *msg* è una struct msghdr.
La struct nlmsghdr costituisce l'header del messaggio netlink, che deve essere utilizzato quando si effettua una comunicazione con i socket netlink. La struttura viene allocata all'interno della funzione *send_message* e *nlh*, che punterà alla struttura appena creata, viene inserito all'interno di *msg*.
L'impostazione del socket e di *msg* viene fatta anch'essa all'interno di *send_message*.

NLMSG_SPACE e NLMSG_DATA sono delle macro definite in linux/netlink.h: la prima ritorna lo spazio che un messaggio netlink con payload della lunghezza passata occupa in byte, mentre la seconda ritorna un puntatore a void che punta all'indirizzo di inizio payload del messaggio netlink ad esso passato.

MAX_PAYLOAD è la grandezza massima che può avere il messaggio da inviare. Anche nel kernel è presente una define che definisce la grandezza massima di un messaggio, che è uguale a MAX_PAYLOAD, ma non è necessario che i due valori siano uguali.

Al termine del programma viene chiuso il socket netlink.

6.1.2 send_message

Oltre alle variabili globali sopra descritte (*sock_fd*, *nlh*, *msg*) sono definite globalmente altre variabili che vengono utilizzate in questa funzione ma che comunque devono rimanere in memoria anche dopo la terminazione di essa. Queste variabili sono *iov* (struct iovec) e *src_addr* e *dest_addr* (struct sockaddr_nl).
La struct iovec viene usata per indicare a *msg* l'header del messaggio netlink da usare. Possiede due campi a cui viene assegnato il puntatore all'header netlink (*nlh*) e la lunghezza del messaggio.
Le altre due variabili sono gli indirizzi netlink per i processi mittente e destinatario, cioè le strutture che descrivono i due capi della comunicazione.

```
/* create a new netlink socket using channel NETLINK_TEST */
sock_fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_TEST);
```

All'inizio della funzione viene creato il socket netlink tramite la funzione *socket*. Netlink è un servizio datagram oriented, e come secondo argomento si può scegliere sia SOCK_RAW che SOCK_DGRAM. NETLINK_TEST è il canale netlink che è stato aggiunto a quelli già esistenti per la comunicazione con il kernel in ambito della scansione. E' definito in linux/netlink.h.

```
/* set the source address, the address of this process */
memset(&src_addr, 0, sizeof(src_addr));
src_addr.nl_family = AF_NETLINK;
src_addr.nl_pid = getpid();

bind(sock_fd, (struct sockaddr*)&src_addr, sizeof(src_addr));

/* set the destination address */
memset(&dest_addr, 0, sizeof(dest_addr));
dest_addr.nl_family = AF_NETLINK;
dest_addr.nl_pid = 0;        /* kernel */
dest_addr.nl_groups = 0;   /* unicast */
```

Successivamente vengono impostati i due indirizzi netlink di mittente e destinatario e viene effettuata una *bind* sul socket. Il campo *nl_pid* degli indirizzi contiene il pid del processo che esso rappresenta anche se in realtà non è sempre così: il campo *nl_pid* è un identificativo che può assumere un qualunque valore. Infatti, nel file dove la struttura dell'indirizzo netlink è definita, il campo *nl_pid* è commentato con "port ID". Per inviare il messaggio al kernel, comunque, il campo dell'indirizzo del destinatario deve essere impostato a 0.
Il campo *nl_groups* serve per definire se l'indirizzo è un indirizzo multicast o no. Con il valore zero si indica che l'indirizzo non è di gruppo.

A questo punto viene creato l'header del messaggio netlink tramite una *malloc*: la macro NLMSG_SPACE fornisce la dimensione che la struttura occuperà in memoria a seconda della grandezza massima consentita per il messaggio. Nell'header vengono impostati i campi di lunghezza del messaggio netlink (compreso header), *pid* e *flag*, il quale viene impostato a 0.
In questo caso il campo *pid* rappresenta davvero il pid del processo mittente, e serve alle applicazioni per capire da che processo proviene il messaggio. Il suo valore non ha conseguenze dirette nella trasmissione del messaggio (non viene esaminato da netlink). Altri

campi “opachi” a netlink sono un campo dedicato al numero di sequenza del messaggio e un campo dedicato al tipo.

In seguito viene posizionato il payload (la stringa che si vuole inviare) all'interno del messaggio netlink con una *strcpy*, usando la macro NLMSG_DATA per ottenere l'indirizzo dove cominciare a scrivere la stringa.
Infine viene inviato il messaggio con una *sendmsg*.

6.2 Kernel

6.2.1 Inizializzazione

Il modulo net/mac80211, all'avvio del sistema, viene inizializzato tramite la funzione di inizializzazione *ieee80211_init*. All'interno di questa funzione è stata inserita la chiamata alla funzione che prepara il kernel alla ricezione e gestione dei messaggi netlink, in arrivo su un socket netlink creato sul canale dedicato.

Sia la funzione *ieee80211_init* che la funzione creata risiedono nel file main.c. In questo file sono state definite alcune variabili globali: le variabili per indicare se una certa funzionalità è abilitata o meno, una variabile intera *pid*, un puntatore al socket netlink (di tipo struct sock) e una variabile di tipo struct netlink_kernel_cfg.
Le variabili per le funzionalità sono gli unici dati pubblici tra quelli elencati, e sono impostati in modo che le funzionalità iniziali siano quelle offerte normalmente dal codice originale del kernel.
La variabile *pid* conterrà l'identificativo del processo mittente, cioè il process ID (che l'applicazione dovrà inserire nell'header). E' inizializzata a 0.
Il puntatore a struct sock (*nl_sk*) conterrà l'indirizzo di memoria del socket netlink mentre l'altra struttura (*cfg*) conterrà alcune opzioni da associare al socket al momento della creazione.

(*le variabili pubbliche, insieme alla funzione per l'invio delle notifiche, sono dichiarate in un file di inclusione che viene incluso nei file che ne devono fare uso, cioè cfg.c e scan.c*)

/ set the callback function */*
cfg.input = netlink_callback;
/ create a new netlink socket with NETLINK_TEST channel and cfg options */*
nl_sk = netlink_kernel_create(&init_net, NETLINK_TEST, THIS_MODULE, &cfg);

La funzione chiamata effettua le precedenti istruzioni. Il campo input della struttura *cfg* indica la funzione (è un puntatore a funzione) che deve essere chiamata a seguito della ricezione di un messaggio. La funzione di ricezione deve accettare un puntatore a una struttura struct sk_buff e non deve restituire risultati al momento della terminazione.

Dopo aver impostato la funzione di callback viene chiamata la funzione *netlink_kernel_create* con gli argomenti descritti. *init_net* è un puntatore a una struct net mentre THIS_MODULE è un puntatore a struct module.
A questo punto, dopo aver controllato che il socket sia stato creato correttamente, la funzione termina.

6.2.2 Gestione dei messaggi

Come è stato scritto sopra, la funzione per l'elaborazione dei messaggi in arrivo accetta come argomento un puntatore a una struttura struct sk_buff. La struttura sk_buff (socket buffer) che viene fornita contiene il puntatore al messaggio netlink ricevuto (compreso di header), il quale è un puntatore a char e deve quindi essere convertito in un puntatore a struct nlmsghdr. Il payload è quindi ricavato usando la macro NLMSG_DATA.

Estratto il payload si esamina che messaggio è stato ricevuto e a seconda di questo si imposta il valore di una delle variabili pubbliche per attivare/disattivare la funzionalità corrispondente.
Nel caso in cui il messaggio sia quello della richiesta di notifiche si memorizza il pid del processo mittente, estraendolo dall'header netlink.

In seguito è descritto come il valore delle variabili influenza il comportamento del kernel.

avoid_scan: questa variabile è inizializzata a 0 e indica se la scansione è disattivata o meno (1: la scansione deve essere evitata, 0: la scansione si comporta normalmente). Il controllo sul valore di questa variabile viene fatto all'inizio della funzione *ieee80211_scan*, presente nel file cfg.c. Come già si era detto precedentemente, questa funzione viene chiamata quando si richiede di effettuare una qualsiasi scansione.

```
if(avoid_scan) {
        cfg80211_scan_done(req, true);
        return 0;
}
```

Il metodo *cfg80211_scan_done* notifica che la scansione è stata terminata. L'argomento *req* è un puntatore alla struttura che descrive la richiesta di scansione (struct cfg80211_scan_request) che viene passata alla funzione *ieee80211_scan*. L'altro argomento notifica che la scansione è stata cancellata per un qualche motivo (aborted).
La chiamata a *cfg80211_scan_done* è necessaria perché terminando la funzione con solo il *return* si verifica un problema: se la scansione viene disattivata non è più possibile riattivarla, in quanto non si ricevono più chiamate a *ieee80211_scan*.

scan_interruptible: inizializzata a 1, indica se la scansione software può essere interrotta temporaneamente per impedire un blocco delle comunicazioni potenzialmente fastidioso (1: la scansione può essere interrotta, 0: la scansione non può essere interrotta). Questo controllo viene effettuato nella funzione *ieee80211_scan_state_decision*, contenuta in scan.c. Questa funzione viene chiamata durante l'esecuzione della scansione software per decidere se continuare la scansione o interromperla per un certo periodo. A seconda della scelta effettuata viene impostata una variabile ad un certo valore.
Il valore di questa variabile definisce in che stato si trova la scansione. Questo stato è analizzato dalla funzione che chiama la *ieee80211_scan_state_decision,* all'interno di uno switch.

I possibili stati sono:
SCAN_DECISION, SCAN_SET_CHANNEL, SCAN_SEND_PROBE, SCAN_SUSPEND e SCAN_RESUME. La funzione *ieee80211_scan_state_decision* viene chiamata quando lo stato vale SCAN_DECISION.

All'interno della funzione che deve decidere se sospendere o no la scansione, tale decisione viene effettuata dalla seguente condizione:

if (associated && (!tx_empty || bad_latency || listen_int_exceeded))

Questa condizione, se verificata, imposta come stato SCAN_SUSPEND (indicando che la scansione deve essere interrotta), altrimenti lo stato viene impostato a SCAN_SET_CHANNEL (che indica di analizzare il prossimo canale).

La variabile *associated* indica se si è attualmente connessi a un access point, *tx_empty* indica se le code di trasmissione sono vuote (non ci sono messaggi da inviare), *bad_latency* indica se il tempo richiesto per effettuare un'altra scansione è troppo alto rispetto a quello che può essere impiegato in modo da mantenere una buona qualità del servizio (non c'è il tempo per effettuare la scansione di un nuovo canale) e *listen_int_exceeded* indica se si è superato il tempo per cui l'access point garantisce di memorizzare i messaggi in arrivo (listen interval).
Per evitare di interrompere la scansione, questo controllo, insieme al calcolo dei valori di *bad_latency* e *listen_int_exceeded*, viene effettuato solo se scan_interruptible vale 1. Se la variabile vale 0 viene sempre impostato come stato SCAN_SET_CHANNEL.

scan_notification: inizializzata a 0, questa variabile indica che il kernel deve inviare una notifica all'applicazione in seguito alla terminazione di una scansione. Al momento la notifica viene fatta solo per la scansione software, non appena la scansione termina.

La scansione termina quando, nella funzione che verifica lo stato del processo di scansione, tale stato vale SCAN_DECISION e non sono rimasti canali da analizzare. Come si era già

detto nella descrizione della gestione del metodo di scansione, i canali da analizzare sono elencati nella richiesta di scansione. Inoltre, è anche presente un campo intero che indica il numero dei canali elencati.

Se, quindi, i canali analizzati fino a quel momento sono pari al numero di canali elencati nella richiesta di scansione, la scansione è finita perché non ci sono più canali da analizzare. A questo punto viene impostata una variabile che indica che la scansione è terminata con successo e in seguito si chiama la funzione *__ieee80211_scan_completed*.

Poco dopo il controllo sul numero di canali rimasti da analizzare viene invocata la funzione per inviare la notifica all'applicazione. La notifica viene quindi inviata solo nel caso in cui la scansione sia terminata correttamente e non sia stata cancellata.

6.2.3 Invio delle notifiche

Il codice per inviare la notifica è contenuto in main.c. La prima istruzione della funzione è un controllo nel quale si verifica se il socket netlink è stato creato, se la funzionalità di invio di notifiche è attiva e se la variabile *pid* è diversa da 0 (in realtà, al momento, questo controllo non dovrebbe essere necessario).
Vengono quindi creati un nuovo socket buffer (struct sk_buff) e un nuovo header netlink (struct nlmsghdr).

```
/* Create a new socket buffer */
rskb = alloc_skb( MAX_PAYLOAD, GFP_KERNEL );
/* Create a new netlink header and put it into the previous created buffer */
nlh = nlmsg_put(rskb, 0, 1, 0, MAX_PAYLOAD, 0);
```

MAX_PAYLOAD è la massima lunghezza consentita per il messaggio. Ha lo stesso valore del MAX_PAYLOAD definito nell'applicazione, ma non è necessario che sia così. La funzione *nlmsg_put*, come descritto, non si limita a creare il socket netlink ma lo inserisce nel buffer specificato nel primo argomento e ne imposta i campi (in ordine: *pid*, *seq*, *type*, *len*, *flags*).

Per inserire l'header nel buffer *nlmsg_put* chiama la funzione *skb_put*, che prende un socket buffer e la lunghezza che deve essere occupata all'interno del buffer; la funzione restituisce un puntatore a char che viene convertito da *nlmsg_put* in puntatore a struct nlmsghdr e passato come valore di ritorno.

In seguito si copia il messaggio di notifica all'interno dell'header e si impostano alcuni campi. Tramite la macro NETLINK_CB si accede al campo cb del socket buffer, che sembra essere uno spazio dedicato (nei commenti: “Free for use for any layer. Put private vars here”, consiste in un vettore di char di grandezza 48).
La macro converte il puntatore a char restituito dal socket buffer in un puntatore a struct netlink_skp_params e vi accede.

```
#define NETLINK_CB(skb)        (*(struct netlink_skb_parms*)&((skb)->cb))

struct netlink_skb_parms {
        struct ucred        creds;        /* Skb credentials    */
        __u32               pid;
        __u32               dst_group;
};
```

Non si è molto certi di come utilizzare la struttura, in quanto non veniva utilizzata nel codice netlink della versione precedente trovato come esempio. Il campo *pid* è stato impostato con la variabile *pid* di main.c (quello dell'applicazione mittente, che aveva inviato la richiesta di ricevere notifiche) e *dst_group* è stato impostato a 0 (dovrebbe indicare se l'indirizzo è di gruppo o no).
Il campo rimanente possiede tre variabili intere: *pid*, *uid* e *gid*. Il campo *pid* è stato impostato a 0.

Successivamente, tramite la funzione *nlmsg_unicast*, il messaggio viene inviato all'applicazione. La funzione prende tre parametri: il socket netlink, il socket buffet e il port ID del socket netlink di destinazione (quello impostato nell'indirizzo netlink creato dall'applicazione che è stato usato per fare la bind sul suo socket netlink). Il port ID è impostato a *pid*.

Alla fine del progetto, con la funzionalità di invio di notifiche funzionante, si sono effettuate delle prove modificando il valore di port ID dei campi di applicazione e kernel. La comunicazione funzionava anche con valori di port ID del socket dell'applicazione pari a 1 e 8888, invece che *getpid*.

6.2.4 Implementazioni precedenti

Quando si era cominciato a scrivere il codice nel kernel per inviare le notifiche di fine scansione all'applicazione, inizialmente si era pensato di inserire il codice all'interno del file scan.c, che è il codice dove si capisce quando una scansione è terminata (come è stato descritto poco prima).
Nel punto in cui si verifica la fine della scansione si era inserita una chiamata alla funzione che doveva: creare un nuovo netlink socket, inviare un nuovo messaggio all'applicazione e chiudere il socket. Si è scritto il codice, si è effettuata la compilazione ma nel momento in cui la funzione veniva eseguita il sistema si bloccava. Inizialmente si era pensato che il blocco fosse dovuto a un ritardo della terminazione della scansione, per cui si è spostata l'esecuzione del codice in un thread. Si è poi scoperto che la creazione del socket falliva e ritornava NULL, quindi si è cercato di capire il perché fallisse.

Si è cercato di risalire all'errore inserendo alcune printk nel codice che doveva essere eseguito (il codice di *netlink_kernel_create* e le funzioni da lei chiamate). Si è quindi scoperto che l'errore veniva generato dalla funzione *netlink_insert* e riportava EADDRINUSE. Il problema era che la struct net utilizzata per creare il nuovo socket (*init_net*) era già stata usata per creare un altro socket in main.c sul canale netlink dedicato. Usare un'altra struct net o un altro canale ritornava un nuovo socket.

A questo punto si è però deciso di utilizzare il socket precedentemente creato (quello dell'inizializzazione), perchè non si voleva usare un nuovo canale e perché non si avevano informazioni su come definire una nuova struct net, e cosa essa rappresentasse.

Un'altra implementazione precedente è stata il tentativo di inviare il messaggio di notifica a un indirizzo di gruppo. Anche questo tentativo di implementazione è stato relativamente

lungo e infruttuoso, e quando infine si è tentato e (dopo un po) riuscito a effettuare l'invio a un indirizzo unicast non si è più tornati a riprovare, a seguito dei nuovi risultati, l'implementazione precedente. I vari tentativi, infatti, avevano richiesto molto tempo e non si voleva rischiare di perderne dell'altro avendo già del codice funzionante. Tuttavia l'invio di una notifica a più destinatari potrebbe essere una buon servizio da offrire.

7 VALUTAZIONI

Il codice creato permette a un'applicazione di:

- bloccare le esecuzioni di scansioni (sia software, che hardware che passiva);
- effettuare una scansione senza interruzioni, senza che essa possa essere sospesa con lo scopo di non bloccare per troppo tempo altre trasmissioni in Wi-Fi;
- ricevere notifiche quando il kernel termina una scansione software.

L'applicazione può bloccare l'esecuzione di scansioni quando non vuole permettere scansioni periodiche (o comandate) e riabilitarle quando pensa che sia il caso.

Una scansione che non viene sospesa è più veloce rispetto a una che non viene sospesa. Nella macchina in cui si è testato il lavoro, la scansione veniva sospesa ogni volta che si finiva di scandire un canale. La scansione così eseguita richiedeva un tempo di circa 4,5 secondi, mentre la scansione senza interruzioni impiegava un tempo di circa 1,75 secondi. La differenza è di poco meno di 3 secondi che, nel caso ci sia fretta di completare la scansione, sembrano piuttosto significativi.

La scansione senza interruzioni è stata resa possibile solo per la scansione software. La funzionalità si può applicare solo a una scansione attiva, quindi si potrebbe tentare di applicarla alla scansione hardware, che però non si è capito come funzioni.

L'invio di notifiche a fine scansione è stata resa possibile solo per la scansione software. Si potrebbe rendere disponibile anche agli altri due casi di scansione.
L'invio di notifiche, così come è stato implementato, è disponibile solo a una applicazione alla volta. Questa è una grossa limitazione che si crede sia molto importante rimuovere.

Inoltre, l'invio di notifiche presenta un difetto di cui ci si è accorti soltanto alla fine. Nel caso un'applicazione voglia ricevere notifiche di terminazione della scansione, questa invia il messaggio di richiesta di notifiche al kernel e il kernel cambia valore alla variabile della

funzionalità corrispondente e comincia a inviare messaggi. Quando l'applicazione termina non viene dato nessun avviso del fatto al kernel, il quale continua a inviare messaggi a vuoto!
Si potrebbe fare che l'applicazione, prima di uscire, notificasse al kernel la sua terminazione, e il kernel disattivasse la funzionalità. Una questione che ci si è posti, comunque, è: che conseguenza ha l'invio dei messaggi inviati e non ricevuti?

In seguito sono elencati i dati dei tempi di esecuzione di alcune scansioni, ottenuti inserendo alcune printk. “RICHIESTA SCANSIONE” indica l'entrata nella funzione che viene eseguita quando si richiede una scansione (*ieee80211_scan*). Le successive tre scritte provengono dal metodo di gestione della scansione (dove si sceglie che scansione effettuare).

Scansione iniziale (le prime due scansioni non vengono interrotte): circa 1.75 secondi
[50.917751] !!! RICHIESTA SCANSIONE !!!
[50.917756] *** INIZIO SCANSIONE ****** SCANSIONE Software ***
[50.933589] *** SCANSIONE start_sw_scan ***<7>
[50.933602] SCANSIONE Channel: *2412*
[51.067025] SCANSIONE Channel: *2417*
[51.200762] SCANSIONE Channel: *2422*
[51.334510] SCANSIONE Channel: *2427*
[51.468243] SCANSIONE Channel: *2432*
[51.601958] SCANSIONE Channel: *2437*
[51.735730] SCANSIONE Channel: *2442*
[51.869426] SCANSIONE Channel: *2447*
[52.003164] SCANSIONE Channel: *2452*
[52.136948] SCANSIONE Channel: *2457*
[52.270634] SCANSIONE Channel: *2462*
[52.404362] SCANSIONE Channel: *2467*
[52.538125] SCANSIONE Channel: *2472*
[52.671828] SCANSIONE TERMINATA

Scansione con interruzioni: circa 4.25 secondi

[70.972891] !!! RICHIESTA SCANSIONE !!!

[70.972896] *** INIZIO SCANSIONE ****** SCANSIONE Software ***

[70.972899] *** SCANSIONE start_sw_scan ***<7>

[70.972982] SCANSIONE Channel: *2412*

[71.315758] SCANSIONE Channel: *2417*

[71.107156] !!! SCANSIONE SOSPESA !!!

[71.449483] !!! SCANSIONE SOSPESA !!!<7>

[71.658010] SCANSIONE Channel: *2422*

[72.000399] SCANSIONE Channel: *2427*

[71.791806] !!! SCANSIONE SOSPESA !!!

[72.134061] !!! SCANSIONE SOSPESA !!!<7>

[72.342715] SCANSIONE Channel: *2432*

[72.685076] SCANSIONE Channel: *2437*

[72.476379] !!! SCANSIONE SOSPESA !!!

[72.818763] !!! SCANSIONE SOSPESA !!!<7>

[73.027367] SCANSIONE Channel: *2442*

[73.369709] SCANSIONE Channel: *2447*

[73.161018] !!! SCANSIONE SOSPESA !!!

[73.503396] !!! SCANSIONE SOSPESA !!!<7>

[73.712030] SCANSIONE Channel: *2452*

[74.054289] SCANSIONE Channel: *2457*

[73.845653] !!! SCANSIONE SOSPESA !!!

[74.188028] !!! SCANSIONE SOSPESA !!!<7>

[74.396635] SCANSIONE Channel: *2462*

[74.738986] SCANSIONE Channel: *2467*

[74.530326] !!! SCANSIONE SOSPESA !!!

[74.872665] !!! SCANSIONE SOSPESA !!!<7>

[75.081280] SCANSIONE Channel: *2472*

[75.214928] SCANSIONE TERMINATA

Scansione senza interruzioni: circa 1.75 secondi

[470.183072] !!! RICHIESTA SCANSIONE !!!

[470.183077] *** INIZIO SCANSIONE ****** SCANSIONE Software ***

[470.183080] *** SCANSIONE start_sw_scan ***<7>

[470.183165] SCANSIONE Channel: *2412*

[470.317105] SCANSIONE Channel: *2417*

[470.450872] SCANSIONE Channel: *2422*

[470.584616] SCANSIONE Channel: *2427*

[470.718302] SCANSIONE Channel: *2432*

[470.852036] SCANSIONE Channel: *2437*

[470.985768] SCANSIONE Channel: *2442*

[471.119493] SCANSIONE Channel: *2447*

[471.253166] SCANSIONE Channel: *2452*

[471.386927] SCANSIONE Channel: *2457*

[471.520641] SCANSIONE Channel: *2462*

[471.654403] SCANSIONE Channel: *2467*

[471.788110] SCANSIONE Channel: *2472*

[471.921839] SCANSIONE TERMINATA

8 CONCLUSIONI E SVILUPPI FUTURI

Tramite questo progetto è stato creato un controllo sui meccanismi di scansione Wi-Fi del kernel Linux 3.6.8. Il risultato è uno strumento in grado di dialogare con il kernel e impostare le preferenze indicate dall'utente o da altre applicazioni. Infatti, applicazioni a conoscenza di certe condizioni potrebbero sentire l'esigenza di impostare alcuni particolari della scansione (per esempio un nodo mobile fermo potrebbe non necessitare di scansioni frequenti). Certamente l'applicazione, la comunicazione e le funzionalità disponibili possono essere migliorate con ulteriori modifiche e aggiunte. Queste modifiche dovrebbero essere guidate da delle esigenze concrete. Bisognerebbe individuare bene come questo strumento possa essere utilizzato e adattarlo al compito richiesto, cercando comunque di renderlo uno strumento il più aperto possibile a tutti i possibili utilizzi che se ne potrebbe fare.

Certamente bisognerebbe gestire il conflitto tra più applicazioni che ne richiedono l'utilizzo. Potrebbe succedere, infatti, che siano presenti all'insaputa dell'utente applicazioni che intendono imporre delle politiche di scansione incompatibili le une con le altre, e il kernel si troverebbe nella situazione di vedersi arrivare ordini contrastanti di continuo. Questo risultato sarebbe un ulteriore problema aggiunto.
Una direzione di sviluppo indicata è quindi quella di rendere l'applicazione che ha il controllo sulla scansione del kernel uno strumento più solido, modificando il modo in cui essa può venire chiamata da altre applicazioni per fare in modo che le politiche di scansione non vengano modificate dall'ultima applicazione arrivata ma che siano decise in modo che la scansione sia il più stabile possibile.

Un'altra direzione di sviluppo consiste nel modificare l'aspetto di come vengono effettuate le notifiche. Ci sono varie cose che possono essere modificate.
L'invio di notifiche solamente a una applicazione alla volta è certamente una politica che si desidera modificare. Ricevere notifiche, cioè sapere quando accade un certo evento, è una funzionalità che non modifica il metodo di scansione. Il richiedere di essere avvertito

quando una scansione termina provoca unicamente il compito del kernel di inviare la notifica, non comporta possibili conflitti con comandi di attivazione/disattivazione di funzionalità (a parte, nel caso venga creata, la chiusura della funzionalità di notifica da dell'applicazione). Ciò che quindi si può cercare di fare è gestire l'invio in multicast delle notifiche, dal kernel o dall'applicazione, a seconda di come si concepisce la funzione dell'applicazione (solo l'applicazione può dialogare con il kernel?).
Inoltre l'invio di notifiche non dovrebbe continuare anche quando nessuna applicazione è in attesa di tali messaggi.
Per finire, l'invio delle notifiche è attualmente disponibile solo per le scansioni software. L'applicazione creata non può attualmente capire quando una scansione hardware o una scansione sul solo canale attuale viene terminata. Sicuramente l'aggiunta di queste notifiche sarebbe cosa utile. Nel caso si facesse si potrebbe chiedere di ricevere notifiche indipendentemente dal tipo di scansione effettuata o richiedere notifiche solo al termine di una particolare scansione.

Passando alle nuove funzionalità da aggiungere si potrebbe dare la scelta di effettuare una scansione attiva sul canale attuale. La scansione sul solo canale attuale, nel caso non sia supportata la scansione hardware, è infatti permessa solo tramite scansione passiva, in quanto il controllo se fare la scansione software viene dopo il controllo di se fare la scansione passiva. Certamente renderebbe la scansione più personalizzabile, ma bisogna vedere se realmente ci può essere la necessità di terminare la scansione con un guadagno di al massimo 90 ms. Si è abbastanza scettici sull'utilità di questa funzione.
Un'altra ricerca potrebbe essere capire come effettuare un comando di scansione, in modo da offrire la funzionalità di ordinare al kernel di effettuare una scansione. Anche qui bisognerebbe capire se questa funzionalità può essere utile.

In conclusione, il progetto svolto permette di scegliere il metodo di scansione utilizzato dal kernel e ricevere informazioni di notifica. Questo può aiutare alcune applicazioni a modificare il comportamento del kernel e a gestire situazioni comprensibili al loro livello. La cosa importante da vedere ora è come queste applicazioni possono interagire con questo strumento, cercare di capire eventuali problemi di gestione che possono nascere e capire se

le applicazioni possono beneficiare nel loro compito di ulteriori funzionalità.
La cosa più importante che si consiglia di fare è quindi analizzare l'utilità e la validità dello strumento di gestione della scansione Wi-Fi creato in questo progetto.

Bibliografia

- Introduzione Wi-Fi,

  wikipedia

- Comportamento della scansione del kernel Linux 3.6.8,

  tesi di Guberti Mattia: “Ottimizzazione della scansione Wifi: responsiveness”

  progetto di De Santis Roberto: “Scansione WiFi parcellizata nel kernel Linux”

- Socket netlink,

  http://www.linuxjournal.com/node/7356/print: “Why and How to Use Netlink Socket”