Abstract

Questa è l'introduzione.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Lo stack di rete

In questo capitolo si vuole presentare la suite di protocolli utilizzata nel progetto di tesi.
Verrà descritto brevemente il modello ISO/OSI per poi soffermarsi con particolare attenzione sulla tecnologia Wi-Fi e sui protocolli toccati dal progetto di tesi.
ISO/OSI è uno standard che definisce un modello composto su più livelli: ogni *layer* si occupa di uno specifico aspetto delle comunicazioni di rete fornendo delle funzionalità a livello superiore e sfruttando le astrazioni fornite dal livello immediatamente inferiore.
In questo modo è possibile ridurre la complessità non banale delle comunicazioni di rete.

1.1 Il modello ISO/OSI

Il modello OSI (Open System Interconnection)[1] è uno standard per le reti di calcolatori stabilito da ISO (International Standard Organization) che definisce l'architettura logica di rete come una struttura a strati composta da una pila di protocolli di comunicazione di rete suddivisa in 7 livelli, i quali insieme espletano in maniera logico-gerarchica tutte le funzionalità della rete. Ciascun layer racchiude in sé, a livello logico, uno o più aspetti fra loro cor-

relati della comunicazione fra due nodi di una rete.
I layers vanno dal livello fisico (quello del mezzo trasmissivo, ossia del cavo twisted pair o delle onde radio) fino al livello delle applicazioni, attraverso cui si realizza la comunicazione di alto livello. Come già accennato ciascun livello fornisce servizi e funzionalità al livello superiore utilizzando le astrazioni fornite dal livello inferiore. Ma vediamo brevemente le funzioni di ciascun livello all'interno dello stack ISO/OSI.

Physical Layer Si occupa di trasmettere dati non strutturati attraverso un mezzo fisico e di controllare la rete, gli hardware che la compongono e i dispositivi che permettono la connessione. In questo livello vengono decisi diversi aspetti legati al mezzo fisico come ad esempio le tensioni scelte per rappresentare i valori logici dei bit trasmessi, la durata in microsecondi del segnale che identifica un bit, la modulazione e la codifica utilizzata e l'eventuale trasmissione simultanea in due direzioni (full-duplex).

Datalink Layer Questo livello si occupa in primis di formare i dati da inviare attraverso il livello fisico, incapsulando il pacchetto proveniente dallo strato superiore in un nuovo pacchetto provvisto di un nuovo header (intestazione) e tail (coda). Questa frammentazione dei dati in specifici pacchetti è detta *framing* e i singoli pacchetti sono chiamati *frame*.
Il livello Datalink effettua, inoltre, un controllo degli errori e delle perdite di segnale in modo tale da far apparire, al livello superiore, il mezzo fisico come una linea di trasmissione esente da errori di trasmissione.

Network Layer Si occupa di rendere i livelli superiori indipendenti dai meccanismi e dalle tecnologie di trasmissione usate per la connessione e prendersi carico della consegna a destinazione dei pacchetti. È responsabile del *routing* ovvero della scelta ottimale del percorso di rete da utilizzare per garantire la consegna delle informazioni dal mittente al destinatario, scelta svolta dal router attraverso dei particolari algoritmi di routing e tabelle di

routing.
È responsabile inoltre della conversione dei dati nel passaggio fra una rete ed un'altra con diverse caratteristiche, come ad esempio reti che adottano diversi protocolli di rete: si deve occupare quindi di tradurre gli indirizzi di rete, valutare la necessità di frammentare i pacchetti dati se la nuova rete ha una diversa Maximum Transmission Unit (MTU) e di valutare la necessità di gestire diversi protocolli attraverso l'impiego di gateway. L'unità dati fondamentale è il pacchetto o *datagram*.

Transport Layer Permettere un trasferimento di dati trasparente e affidabile (implementando anche un controllo degli errori e delle perdite) tra due host.
È il primo livello realmente end-to-end, cioè da host sorgente a destinatario. Si occupa di stabilire, mantenere e terminare una connessione, garantendo il corretto e ottimale funzionamento della sottorete di comunicazione nonché del controllo della congestione: si occupa, cioè, di evitare che troppi pacchetti dati arrivino allo stesso router contemporaneamente causando così una perdita dei pacchetti stessi.
A differenza dei livelli precedenti, che si occupano di connessioni tra nodi contigui di una rete, il Transport Layer si occupa solo del punto di partenza e di quello finale. Si occupa anche di effettuare la frammentazione dei dati provenienti dal livello superiore in pacchetti, detti generalmente *segmenti*, e trasmetterli in modo efficiente ed affidabile usando il livello rete ed isolando da questo i livelli superiori. Inoltre, si preoccupa di ottimizzare l'uso delle risorse di rete e di prevenire la congestione.

Session Layer Consente di aggiungere, ai servizi forniti dal livello di trasporto, servizi più avanzati, quali la gestione del dialogo (mono o bidirezionale), la gestione del token (per effettuare mutua esclusione) o la sincronizzazione (inserendo dei checkpoint in modo da ridurre la quantità di dati da ritrasmettere in caso di gravi malfunzionamenti). Si occupa anche di inserire

dei punti di controllo nel flusso dati: in caso di errori nell'invio dei pacchetti, la comunicazione riprende dall'ultimo punto di controllo andato a buon fine.

Presentation Layer Si occupa di trasformare i dati forniti dalle applicazioni in un formato standard e di offrire servizi di comunicazione comuni, come la crittografia, la compressione del testo e la riformattazione.

Application Layer Il livello Applicazione è quello più vicino al livello utente fornisce e quindi un interfaccia tra le applicazioni e lo stack di rete sottostante che si occupa dell'invio di messaggi. I protocolli di livello applicazione si occupano quindi dello scambio di informazioni tra apps in esecuzione sull'host sorgente e quello destinatario della comunicazione.

1.2 ISO/OSI vs TCP/IP

TCP/IP sviluppato inizialmente dal dipartimento della difesa americano e utilizzato nei primi computer UNIX-based attualmente è lo *standard de facto* per tutte le comunicazioni internet.
TCP/IP, come l'OSI model, è strutturato su più livelli alcuni dei quali molto simili per caratteristiche e funzionalità a quelli di ISO/OSI: TCP/IP accorpa in un unico layer funzionalità contenute su più livelli del modello OSI.

TCP/IP è l'approccio utilizzato in ogni tipo di comunicazione internet. L'obbiettivo di ISO/OSI invece è quello fornire uno standard da usare come guideline per la definizione di protocolli e applicazioni internet.

I layer nello stack TCP/IP sono quattro e sono così organizzati rispetto al modello OSI come illustrato nella figura 1.1.

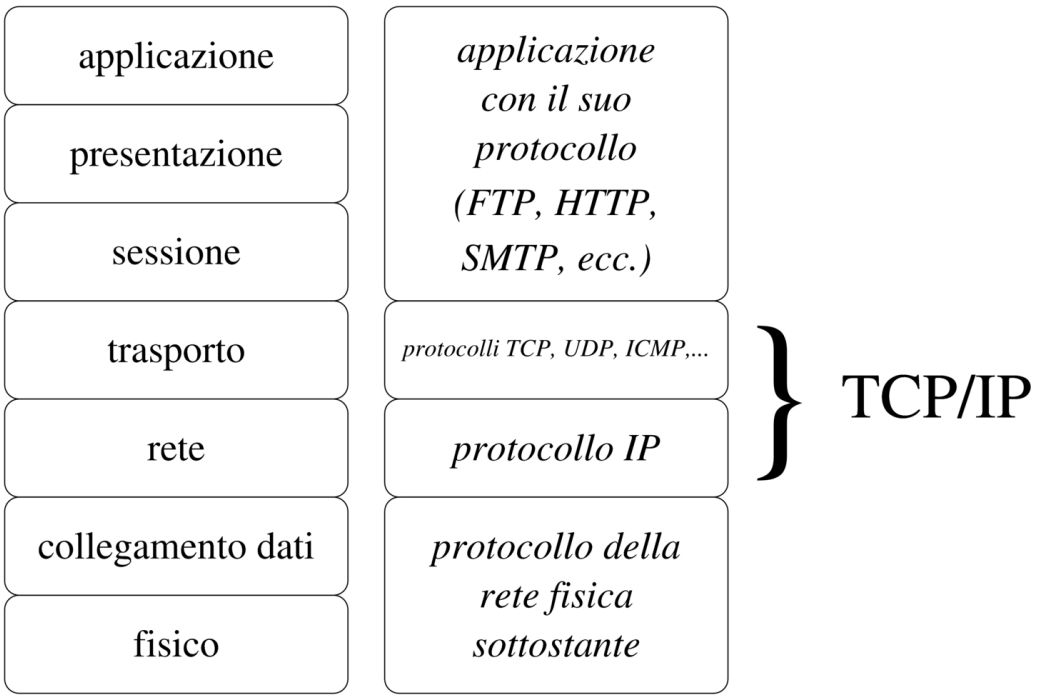

Figura 1.1: Stack di rete ISO/OSI e TCP/IP

1.3 Wi-Fi

Wi-Fi indica una tecnologia che consente a calcolatori collocati su di una stessa WLAN (Wireless Local Area Network) di comunicare senza fili attraverso specifiche frequenze di onde radio secondo le specifiche dello standard **IEEE 802.11**. Sempre più dispositivi dispongono di interfacce di rete Wi-Fi: dai laptop, agli smartphone fino agli elettrodomestici di ultima generazione che possono essere così interconnessi entro un certo raggio di copertura.
a tecnologia Wi-Fi può essere usata per fornire connettività internet ai dispositivi presenti nel raggio di copertura della WLAN se la Local Area Network è connessa a internet.

1.3.1 WLAN architecture and types

L'architettura di una WLAN è caratterizzata da diverse componenti.

Stazioni In una WLAN ciascun dispositivo munito di Wireless Network Interface Controllers (WNICs) e che quindi può comunicare senza fili è detto stazione. Vi sono due categorie di stazioni:

- **Access Points (AP)** ovvero dispositivi elettronici che ricevono/trasmettono segnali radio da/verso nodi mobili equipaggiati con schede di rete Wi-Fi.
- **Clients**, tutti i nodi mobili che possono essere equipaggiati con una wireless network interface come ad esempio laptops, smartphones o workstations.

Basic service set Il Basic Service Set (BSS) è un insieme di tutte le stazioni che possono comunicare tra loro. Ciascun BSS ha un proprio identificativo detto BSSID che corrisponde al indirizzo MAC dell'access point che serve i diversi clients per quella BSS.
Vi sono due tipi di BSS:

- Indipendent BSS (IBSS) ovvero una *rete ad-hoc* caratterizzata dall'assenza di un access point. Questo tipo di BSS non può quindi essere interconnesso con altri Basic Service Set.
- Infrastructure BSS caratterizzati dalla presenza di un access point, un BSS di questo tipo può essere connesso con altri Basic Service Set.

Extended Service Set Un Extended Service Set (ESS) è un insieme di BSS interconnessi tra loro. Gli access points in un ESS sono connessi tra loro da un *Distribution System*. Ciascun EES è caratterizzato da una stringa identificativa lunga al massimo 32 byte detta SSID.

Distribution System Il Distribution System (DS) interconnette tra loro gli access point di diversi EES. Un access point può essere principale, di inoltro o remoto. Un access point principale è collegato tipicamente alla rete cablata. Un access point di inoltro trasmette i dati fra le stazioni remote e

principali. Un access point remoto accetta i collegamenti dai client senza fili e li passa a quelli di inoltro o quelli principali.

Esistono due tipologie di rete WLAN che differiscono dalle modalità di comunicazione:

- Infrastructure, i nodi comunicano tra loro attraverso a una base station che funge da wireless access point.
- Reti ad-hoc, ovvero una rete dove le stazioni possono comunicare peer-to-peer (P2P) senza alcun access point. Questo viene realizzato tramite IBSS.

1.4 IEEE 802.11

IEEE 802.11 è uno standard di trasmissione per reti WLAN operanti su frequenze 2.4 e 5 GHz che definisce un interfaccia di comunicazione base per comunicazione Wi-Fi. Le specifiche definite nello standard 802.11 si focalizzano sul livello fisico e MAC del modello ISO/OSI. Il sistema di numerazione 802.11 è dovuto a IEEE che utilizza 802.x per indicare una famiglia di standard per le comunicazione di rete tra cui lo standard *Ethernet* (IEEE 802.3). Per tanto IEEE 802.11 si adegua perfettamente agli altri standard 802.x per reti locali wired e le applicazioni che lo utilizzano non dovrebbero notare nessuna differenza logica, una degradazione delle performance invece è tuttavia possibile. IEEE 802.11b è stato il primo protocollo largamente utilizzato seguito da 802.11a, 802.11g, 802.11n, e 802.11ac. Vi sono altri standard nella famiglia 802.11 (c-f, h, j) che sono per lo più piccole modifiche, estensioni o correzioni alle precedenti specifiche.
Per quanto concerne le performance lo stream data rate può arrivare fino a 780 Mbit/s in 802.11ac grazie anche alla tecnologia MIMO (Multiple-Input and Multiple-Output) che consente di aumentare la capacità del canale trasmissivo usando più trasmittenti e ricevitori sfruttando così il fenomeno del

multipath-propagation ovvero un segnale radio può raggiungere un ricevitore attraverso diversi percorsi, *path*.

1.4.1 Physical Layer in 802.11

Come già detto IEEE 802.11[2] utilizza le bande di frequenza 2.4 e 5 GHz e a *livello fisico* vengono usate delle tecniche di modulazione *half-duples*: in particolare viene utilizza la Orthogonal Frequency-Division Multiplexing (OFDM) che utilizza un numero elevato di sotto-portanti ortogonali tra di loro, oppure quella chiamata Direct Sequence Spread Spectrum (DSSS), che è una tecnologia di trasmissione a banda larga nella quale ogni bit viene trasmesso come una sequenza ridondante di valori, detti chip, rendendola così più resistente ad eventuali interferenze.

1.4.2 Media Access Control

Media Access Control (MAC) è un *sublayer* del livello *Data Link* del modello OSI. MAC fornisce meccanismi di indirizzamento e di controllo di accesso al canale che consentono a nodi mobili di comunicare attraverso una rete con medium condiviso. L'hardware che implementa MAC è detto *media access controller*. Le funzioni principali del MAC sono quindi quelle di regolamentare l'accesso al mezzo fisico, frammentazione dati in frame e riconoscimento frame stessi e controllo degli errori.

1.4.3 CSMA/CA

Carrier Sense Multiple Access with Collision Avoidance è un protocollo di accesso multiplo in cui i nodi cercano di evitare a priori il verificarsi di collisioni in trasmissione. Questo approccio è l'ideale per tipologie di reti nella quale non risulta possibile (oppure poco affidabile e dispendioso) rilevare un'avvenuta collisione.
Quando un nodo vuole effettuare una trasmissione ascolta il canale *(listen-before-talk)* (LBT): se il canale risulta *idle* il nodo aspetta un certo *DISF*

time (Distribuited Inter Frame Space) trascorso il quale, se il canale risulta ancora libero, comincerà a trasmettere. A termine della trasmissione il nodo sorgente aspetterà per un certo intervallo *SISF (Short Inter Frame Space)*, più piccolo di DISF, la ricezione di un *ACK*. Se il nodo sorgente non riceve alcun ACK ritrasmetterà il messaggio per un certo numero di volte.
Per tutta la durata della trasmissione e per la durata dello SISF time le altre stazioni, trovando il canale occupato, non avvieranno altre comunicazioni evitando così collisioni. La durata dello SISF inferiore a quello dell'intervallo di DISF assicura che nessuna stazione comincerà una trasmissione prima della ricezione dell'eventuale messaggio di acknowledgment da parte del nodo che ha appena concluso la trasmissione.
Nel caso in cui una stazione volesse trasmettere e rileva il canale occupato attenderà per un certo intervallo di tempo casuale, detto intervallo di *back-off*, prima di riprovare a trasmettere. L'intervallo di back-off è realizzato per mezzo di un timer che decrementa il valore di un contatore, inizializzato con il valore dell'intervallo, solamente nei periodi di inattività del canale, ovvero quando non vi sono trasmissioni, il valore del contatore resterà invece invariato durante i periodi di trasmissione da parte di altre stazioni (*frozen back-off*). Quando il valore del contatore raggiungerà lo zero la stazione effettuerà un nuovo tentativo di trasmissione. Questo meccanismo di accesso al mezzo è detto *Basic Access Mechanism*

1.4.4 Il problema dei nodi nascosti

Il problema dei nodi nascosti in una rete wireless si ha quando un nodo all'interno della rete è visibile da un Access Point ma non da tutte le altre stazioni collegate al medesimo AP. Questo può comportare una serie di problemi per quanto riguarda il controllo dell' accesso al mezzo.

Come si può vedere dalla figura 1.2 la stazione A e la C sono nel raggio di copertura della stazione B. Per qualche motivo (come può essere la distanza o un ostacolo) A e C non possono comunicare direttamente e quindi non possono nemmeno rilevare (*sensing*) la portante trasmessa dall'altra stazione

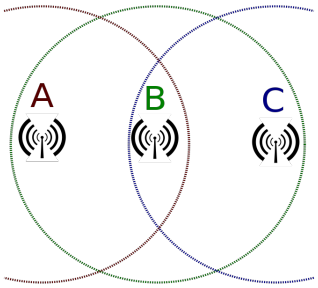

Figura 1.2: Il problema dei nodi nascosti

verso la stazione centrale B. In particolare si possono quindi verificare delle collisioni quando sia A che C, rilevando il canale libero, effettuano in contemporanea una trasmissione verso B.
Per ovviare a questo problema IEEE 802.11 definisce un meccanismo opzionale che introduce due tipi di pacchetti di controllo, in particolare:

- *RTS* (Request To Send), quando un nodo vuole trasmettere, prima di inviare il frame vero e proprio, invia al destinatario un pacchetto di tipo RTS contenente destinatario del messaggio, mittente e durata della trasmissione che seguirà.

- *CTS* (Clear To Send), quando un nodo riceve un pacchetto di tipo RTS risponde con un pacchetto di tipo CTS che contiene essenzialmente le stesse informazioni contenute nel frame di tipo RTS; quando il nodo mittente avrà ricevuto il frame CTS potrà cominciare l'inoltro del frame effettivo precedentemente annunciato tramite il rispettivo RTS.

I pacchetti RTS e CTS vengono inoltrati a tutte le stazioni comprese quindi anche quelle nascoste al mittente che si metteranno in attesa per tutta la durata della trasmissione come specificato dai frame di controllo.
Questo meccanismo non è del tutto esente da collisioni. Infatti le collisioni possono ancora avvenire durante lo scambio dei pacchetti di controllo: ad esempio due stazioni mandano contemporaneamente una Request To Send. Nonostante ciò la probabilità di collisione risulta essere più bassa e meno significativa rispetto all'approccio che non fa uso dei pacchetti RTS/CTS in

quanto i frame di controllo hanno una dimensione molto ridotta (fino a 2347 bytes).
Se una stazione vuole trasmettere un pacchetto di dimensione inferiore al frame di controllo il messaggio verrà inoltrato immediatamente senza prima generare il corrispettivo RTS.

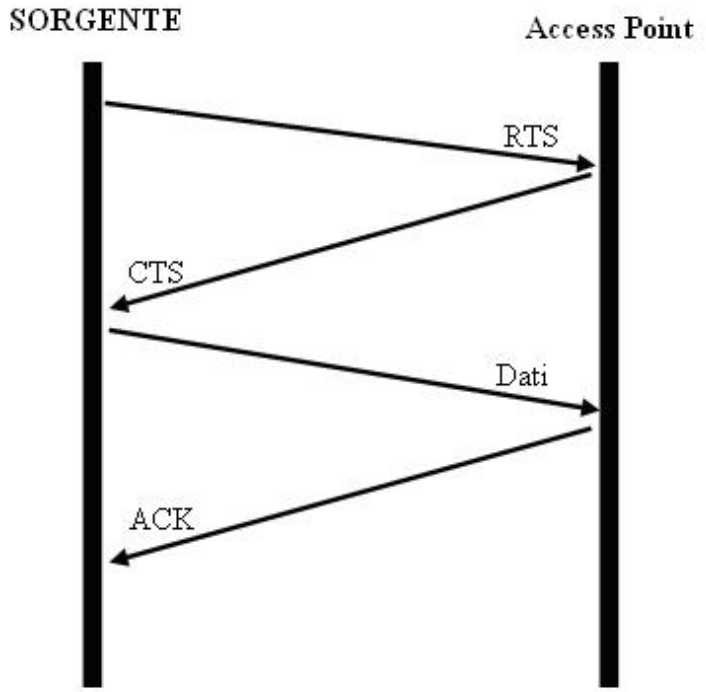

Figura 1.3: *Four-way handshake* via RTS/CTS

1.4.5 IEEE 802.11 frame

IEEE 802.11 definisce tre tipologie di frame:

- DATA, contengono meramente dati.
- CTRL, servono per facilitare l'interscambio di data frame tra le stazione; appartengono a questa categoria i frame RTS, CTS e ACK.

- MGMT, frame utili al mantenimento della comunicazione; i *beacon frame* (frame inviato periodicamente da un AP per annunciare al sua presenza e il suo SSID) appartengono a questa categoria.

Ciascun frame è composto da un *MAC header*, un *payload* e un *frame check sequence (FCS)*.

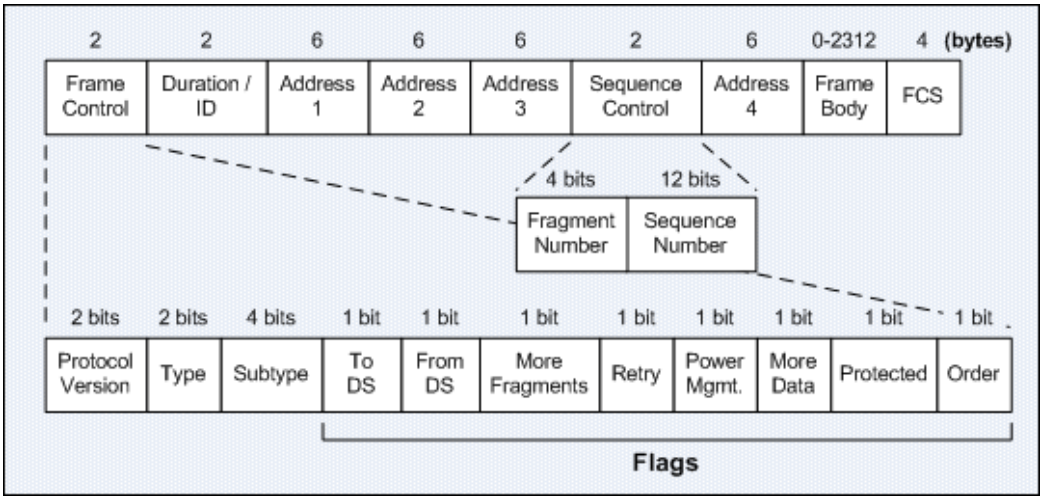

Figura 1.4: IEEE 802.11 frame

MAC header I primi due byte del MAC header contengono un campo molto interessante, il *frame control*. Il frame control contiene diversi sottocampi:

- Protocol Version, ovvero due bit rappresentanti la versione del protocollo, posto sempre a zero; altri valori sono riservati per un uso futuro.

- Type, due bit identificanti appunto il tipo di frame.

- Subtype, quattro bit che identificano il sottotipo del frame; ad esempio come beacon è un sottotipo di MGMT.

- ToDS & FromDS, ciascun campo occupa un bit e indica se un data frame è diretto o proviene da un distribution system. I frame di tipo CTRL e MGMT hanno entrambi i flag settati a zero.

- More Fragments, bit settato quando un pacchetto è viene frammentato in più frame per la trasmissione; tutti i pacchetti con eccezione dell'ultimo inviato avranno questo flag settato.

- Retry, indica se un frame è stato oggetto di ritrasmissione; utile per eliminare eventuali frame duplicati.

- Power Management: indica il *power management state* (ovvero se la stazione è in power-save state o meno) del mittente settato dopo la trasmissione. Gli AP non setteranno mai questo bit in quanto sempre attivi per la gestione delle connessioni.

- More Data, questo bit indica che c'è almeno un pacchetto disponibile; settato dagli AP per facilitare le stazioni in power-save mode.

- Protected, indica se il payload del frame è stato cifrato o meno.

- Order, questo bit è settato solamente quando i frame sono inviati in ordine uno dietro l'altro; spesso ciò non avviene per motivi di performance.

Gli altri campi contenuti nell'header MAC sono la durata di trasmissione, gli indirizzi MAC (*source*, *destination*, *transmitter* e *receiver*) e il *Sequence Control*.
Il campo Sequence Control è composto da due byte usato per identificare l'ordine dei frame spediti. I primi 4 bit corrispondono al *fregmentation number* e gli ultimi 12 bit sono il *sequence number*. Il fragmentation number indica il numero di ogni pacchetto precedentemente frammentato, il sequence number, invece, è un valore modulo 4096 assegnato a un frame e rimane costante per ogni ritrasmissione o per ciascun fragment di quel pacchetto.

Payload Il Payload ha dimensione variabile da 0 a 2304 byte; contiene informazioni provenienti dai livelli di rete superiori. Le informazioni provenienti dai livelli superiori prima di essere incapsulati in un frame MAC sono incapsulate in un frame di tipo IEEE 802.2 che definisce LLC (Logical Link Protocol) che rappresenta il sublayer superiore del livello Data-Link del modello OSI. LLC offre un'interfaccia di comunicazione omogenea verso il livello Data-Link al layer superiore (a differenza di MAC che è dipendente dal mezzo trasmissivo utilizzato). Il pacchetto 802.2 è detto PDU (Protocol Data Unit) e il suo header contiene informazioni di controllo aggiuntive e può essere esteso con SNAP (Subnetwork Access Protocol) che consente di specificare il tipo di protocollo (*Protocol ID* field) a cui appartengono i dati contenuti nel payload della PDU. Questa estensione può essere molto utile ad esempio in fase di ricezione di un frame: è possibile vedere a quale protocollo appartengono i dati contenuti nel payload e consegnare, quindi, i dati al giusto protocollo dello stack di rete (si basti pensare anche solamente ad IPv4 vs IPv6).

Frame check sequence (FCS) Spesso detto anche *Cyclic Redundancy Check* (CRC) permette di verificare l'integrità di un frame appena ricevuto: quando un frame sta per essere spedito la stazione sorgente calcola questo valore e lo appende al frame IEEE 802.11. Quando un nodo riceve il frame ricalcola l'FCS sulla base dei dati ricevuti e lo confronta con il valore contenuto nel *trailer* del pacchetto; se i due valori coincidono il frame non ha subito delle modifiche durante la trasmissione.
Il campo FCS occupa gli ultimi quattro byte del frame IEEE 802.11

1.4.6 Security in IEEE 802.11

Data la sempre più larga diffusione delle reti Wi-Fi e dalla natura del loro segnale (è molto difficile controllare quale dispositivo riceve il segnale radio) la sicurezza è un aspetto molto importante e assolutamente da non sottovalutare in 802.11. Nel corso degli anni sono stati sviluppati e proposti diversi

approcci per rendere le reti WLAN sempre meno sensibili a intercettazioni e attacchi da parte di terzi.

Access Control List Un primo banale approccio è quello dell'*access control list*. L' Access Point mantiene una lista degli indirizzi MAC autorizzati alla comunicazione: l' AP riceve messaggi provenienti solo dai clients presenti nell'access control list. Qualsiasi messaggio proveniente da una stazione non presente nella lista sarà ignorato.
Questo approccio presenta due grandi difetti. Innanzitutto fornisce solamente una politica di controllo degli accessi senza fornire nessun meccanismo di protezione sui dati trasmessi. Inoltre questo approccio può essere facilmente raggirato tramite una tecnica di *MAC spoofing*. In particolare tramite un software di *wireless network analysis* è possibile monitorare il traffico delle rete WLAN vicine e quindi captare informazioni sensibili da eventuali messaggi trasmessi in chiaro: data la mancanza di confidenzialità nei messaggi trasmessi su reti che adottano esclusivamente la politica dell'Access Control List come protezione è possibile quindi risalire ad indirizzi MAC autorizzati alla comunicazione. A questo punto è possibile modificare l'indirizzo MAC dell'interfaccia di rete (possibile sia in ambiente UNIX che Windows) per impersonare un altro client della rete WLAN.

WEP (Wired Equivalent Privacy) WEP (Wired Equivalent Privacy)[3] è stato il primo protocollo di sicurezza definito nello standard IEEE 802.11. L'obbiettivo di WEP è quello di garantire confidenzialità e integrità del traffico trasmesso in maniera wireless. Il nome è dovuto al fatto che WEP è stato pensato per fornire confidenzialità sui dati trasmessi paragonabile a quella delle reti cablate.
WEP sfrutta il cifrario a chiave simmetrica RC4 con chiave a 64 o 128 bit. La chiave WEP utilizzata è la concatenazione di due valori: il primo dinamico detto Initialization Vector (IV) e la seconda parte statica corrispondente alla chiave segreta condivisa. Il vettore di inizializzazione è una sequenza di 24 bit generata casualmente al momento dell'invio del frame da parte

dell'interfaccia di rete (per ogni trasmissione verrà generato un IV in quanto RC4 è un *cifrario a flusso*). A seconda della lunghezza della WEP key la chiave segreta condivisa sarà quindi lunga 40 bit nel caso di una WEP key di 64 bit oppure 104 bit nel caso di una chiave a 128 bit.
Al momento dell'invio di un frame la stazione sorgente genera il vettore di inizializzazione e lo concatena alla shared key. Una volta che la WEP key è stata formata viene data in pasto all'algoritmo di cifratura RC4 per produrre una stringa pseudo-casuale della lunghezza pari ai dati da trasmettere. Una volta generata la stringa pseudo-random quest'ultima viene posta in XOR dalla scheda di rete con i dati da trasmettere: il risultato assieme al vettore di inizializzazione in chiaro sarà appeso a un header IEEE 802.11 e trasmesso verso il destinatario del messaggio.
Quando il nodo destinatario riceve il messaggio cifrato come prima cosa legge l'IV lo concatena alla shared key e calcola la psudo-random string via RC4 (data la stessa WEP key la stringa pseudo-casuale generata sarà sempre uguale). Il risultato ottenuto viene posto in XOR con i dati cifrati contenuti nel frame ottenendo così il testo in chiaro.
A partire dal 2003 questo approccio non è più considerato sicuro a causa dalle numerose falle presenti in WEP e dalla facilità con cui RC4 può essere violato.

WPA (Wi-Fi Protected Access) Una volta scoperte le falle che affliggevano WEP è iniziato lo sviluppo del protocollo IEEE 802.11i, un nuovo standard considerato pienamente sicuro. Nel frattempo viene rilasciato dalla Wi-Fi Alliance WPA (Wi-Fi Protected Access)[3] che soddisfa molte delle linee guide di IEEE 802.11i Il WPA è caratterizzato da tre componenti principali:

- TKIP (Temporal Key Integrity Protocol), è la componente che più va a sostituire la logica di WEP risolvendo la maggior parte delle sue vulnerabilità. Una delle innovazioni più importanti è quella che ogni messaggio trasmesso viene cifrato con una chiave diversa in modo tale

da non esporre la chiave principale.
Molte funzioni di crittografia sono built-in nell'hardware di rete per tanto, non essendo possibile un aggiornamento software, per rendere compatibile a pieno WPA con il precedente hardware IEEE 802.11 il nuovo standard sfrutta alcune delle feature usate anche da WEP: in particolare anche WPA fa utilizzo di RC4. WPA inoltre utilizza un meccanismo di *key hierarchy* ovvero la chiave principale (Pairwise Master Key) viene utilizzata per generare chiavi temporanee come le sessione key, group keys etc etc. In particolare WPA sfrutta RC4 in modo diverso rispetto a WEP ovvero RC4 viene utilizzato per generare una chiave temporanea a partire dalla shared key anzichè per cifrare direttamente il messaggio. La prima chiave a essere generata è la *session key* che sarà poi utilizzata come seme per la generazione delle future *per-packet key.*
Ciascuna per-packet key, lunga 104 bit, è generata da una funzione hash che calcola un digest a partire dall'indirizzo MAC sorgente, il vettore di inizializzazione (che in WPA è stato esteso da 24 a 48 bit ed è implementato come un contatore, *sequence counter*, per evitare *replay attack*) e la session key. Una volta ottenuta la per-packet key le operazioni di cifratura e decifratura sono identiche a quelle di WEP con la sola differenza che il vettore di inizializzazione è sostituito con i 16 bit meno significativi del IV di WPA e con un dummy byte inserito in mezzo.
TKIP risulta quindi essere un sistema di cifratura a 128 bit a chiave dinamica molto più sicuro rispetto al sistema adottato da WEP che prevedeva 24 bit dinamici con una chiave di 40 o 104 bit statica.

- MIC (Message Integrity Code) detto anche *Michael* è una funzione hash con chiave pensata per proteggere l'integrità di un pacchetto. Il valore calcolato a partire dall'intero pacchetto non criptato è un digest di 8 byte. La funzione hash utilizzata da MIC è una funzione progettata per poter essere eseguita su dispositivi con capacità di calcolo scarse

come possono essere appunto le schede di rete. A causa del bassa capacità di calcolo la funzione hash utilizzata fornisce protezione pari a un algoritmo di cifratura con con una chiave a 20 bit considerato lo standard per una bassa protezione. Per compensare quindi una bassa protezione WPA introduce una serie di contromisure per proteggere la rete da eventuali modifiche dei pacchetti da parti di terzi (packets modification attack) ovvero quando la rete intercetta un pacchetto compromesso disabilità il collegamento wireless con gli host coinvolti per 60 secondi ed ogni dispositivo compromesso deve richiedere forzatamente una nuova session key.
Il pericolo causato da questo tipo di contromisura che un malintenzionato può intenzionalmente forgiare pacchetti invalidi in modo tale che l'access point adotti questo approccio di continuo generando così un denial of service.
Il MIC viene posto tra il payload dati e il campo CRC del frame IEEE 802.11.

- 802.1x Port based Network Access Control è un protocollo per il controllo degli accessi in una rete wireless basato sul controllo delle porte di accesso alla rete (switch e/o access point). Questo protocollo divide la rete in tre entità principali ovvero *supplicant*, cioè il client che vuole connettersi alla rete, l'*authenticator* ovvero il punto di accesso alla rete dove il supplicant vuole fisicamente connettersi (tendenzialmente uno switch o un access point che collega il nodo alla rete) e l'*authentication server* il cui compito è quello di validare l'accesso alla rete del client.
  In una rete che adotta 802.1x un nodo deve prima autenticarsi prima di poter accedere e comunicare nella rete wireless. Uno switch o un access point accetta come traffico proveniente da un client non autentificato soltanto messaggi di autenticazione di tipo EAP (Extensible Authentication Protocol) bloccando qualsiasi altra forma di traffico fino a che il client non effettua l'accesso con successo. 802.1x è inoltre responsabile della generazione e della consegna della session key una volta che il

nodo si è autentificato con successo.

IEEE 802.11i (a.k.a. WPA2) La più sostanziale differenza tra WPA e IEEE 802.11i (WPA2)[3] è che 802.11i adotta AES (Advanced-Encription Standard) per cifrare i frame. AES è un algoritmo di cifratura a blocchi che rappresenta lo stato dell'arte per quel che riguarda algoritmi di cifratura. L'unico inconveniente è che le schede di rete che supportano esclusivamente WEP non possono essere aggiornate via software per supportare AES e quindi IEEE 802.11i. Se una rete volesse adottare IEEE 802.11i tutte le stazioni dovrebbero montare hardware di rete compatibile con WPA2.

1.5 Livello Network

Qui in seguito verranno presentati i principali protocolli utilizzati a livello rete ovvero il terzo livello dello stack ISO/OSI.

1.5.1 IPv4

IPv4 (Internet Protocol version 4)[4], come descritto nel RFC 791 dell'IETF del 1981, è un protocollo connectionless per l'uso su reti a commutazione di pacchetto, come ad esempio Ethernet. È un protocollo di tipo *best-effort* ovvero non viene garantita l'effettiva consegna o se i pacchetti saranno recapitati nel giusto ordine oppure duplicati. I pacchetti scambiati a livello network sono detti *datagram*.

Indirizzi IPv4 IPv4 utilizza indirizzi a 32 bit (suddivisi in quattro gruppi da un byte tramite la *decimal dotted notation*) per identificare univocamente un singolo host . Molti degli indirizzi IP sono però riservati per scopi particolari (reti domestiche o indirizzi di *multicast*) facendo si che l'effettivo *pool* di indirizzi IP disponibili sia piuttosto ridotto in confronto alla diffusione degli ultimi anni dei dispositivi che possono connettersi alla rete.
Un indirizzo IP, ad esempio, può essere 192.168.1.102 (indirizzo IP classe C

); in un indirizzo IP, a seconda della classe e dello scopo, i bit più significativi vengono utilizzati per identificare la rete quelli meno significativi, invece, sono utilizzati per indirizzare direttamente i singoli host. Il numero di bit utilizzati per individuare la rete o gli host dipende dalla classe dell'indirizzo IP e da come è stata progettata la rete stessa.

IPv4 header Un datagram IP è formato da un header e da una porzione dati. La dimensione massima del datagram può essere di 65535 byte.

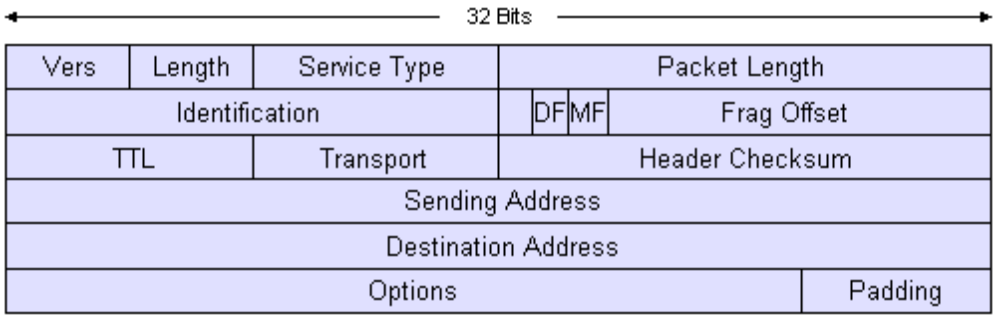

Figura 1.5: Formato dell'header IPv4

I campi dell'header IP sono:

- **Version**, 4 bit indicanti la versione del protocollo IP del pacchetto; in IPv4 questo campo assume il valore 4.
- **Internet Header Length** (IHL), 4 bit indicanti la lunghezza dell'header IP (in word da 32 bit); l'header può avere dimensione variabile a seconda se il campo *Options* è settato o meno.
- **Type of Service** (TOS), ottetto che nelle specifiche iniziali del protocollo (RFC 791) davano la possibilità all'host mittente di specificare il modo e la priorità con cui il nodo destinatario doveva trattare il datagram. In realtà questo campo non è mai stato largamente utilizzato e negli ultimi e recentemente questi 8 bit sono stati ridefiniti e hanno la funzione di *Differentiated services* (DiffServ) nell'IETF e *Explicit*

Congestion Notification (ECN) codepoints (RFC 3168) necessari per le tecnologie basate su streaming in tempo reale come ad esempio per il *Voice over IP* (VoIP).

- **Total Length**, 16 bit, indica la dimensione massima in byte dell'intero dagram IP (header e dati); tale lunghezza può variare da un minimo di 20 byte (header minimo e senza alcun dato) a un massimo di 65535 byte. In ogni momento un host deve poter gestire datagrammi di dimensione minima 576 byte mente, se necessario, possono frammentare datagram di dimensione maggiore.

- **Identification**, 16 bit, definito per identificare univocamente i vari frammenti in cui può essere suddiviso un datagram IP. Può essere anche utilizzato per valutare la presenza di datagram ridondanti (indirizzo IP sorgente più Identification identificano univocamente un pacchetto).

- **Flags** ovvero 3 bit utilizzati per il controllo del protocollo e la frammentazione dei datagrammi. Il primo bit detto *Reserved* è sempre settato a zero; vi sono poi DF (Don't Fragment) che se settato a 1 indica che il pacchetto non deve essere frammentato e MF (More Fragment) che se settato a 0 indica che è l'ultimo frammento (o il solo frammento originario) e tutti gli altri frammenti avranno quindi questo flag posto a 1.

- **Fragment Offset**, 13 bit, indica l'offset, misurato in blocchi da 8 byte, di un particolare frammento relativamente all'inizio del datagram IP originario. L'offset massimo risulta pertanto essere 65526 byte che, sommato la dimensione dell'header IP, potrebbe eccedere la dimensione massima di 65535 byte prevista per un datagram IP.

- **Time To Live** (TTL), 8 bit, indica il *tempo di vita* di un pacchetto necessario per evitarne la persistenza indefinita sulla rete nel caso in cui

non si riesca a recapitarlo al destinatario. Storicamente il TTL misurava i "secondi di vita" del pacchetto, mentre ora esso misura il numero di "salti" da nodo a nodo della rete: ogni router che riceve il pacchetto prima di inoltrarlo ne decrementa il TTL (modificando quindi anche il campo Header Checksum), quando questo arriva a zero il pacchetto non viene più inoltrato ma scartato. Tipicamente, quando un pacchetto viene scartato per esaurimento del TTL, viene automaticamente inviato un messaggio ICMP al mittente del pacchetto, specificando il codice di richiesta scaduta; la ricezione di questo messaggio ICMP è alla base del meccanismo di traceroute.

- **Protocol**, 8 bit, indica il codice associato al protocollo utilizzato nel campo dati del pacchetto IP.

- **Header Checksum**, 16 bit, è un campo usato per il controllo degli errori dell'header. Ad ogni hop, il checksum viene ricalcolato (secondo la definizione data in RFC 791) e confrontato con il valore di questo campo: se non c'è corrispondenza il pacchetto viene scartato. È da notare che non viene effettuato alcun controllo sulla presenza di errori nel campo Data deputandolo ai livelli superiori.

- **Source Address**, 32 bit, indirizzo IP associato all'host del mittente del datagram.

- **Destination Address**, 32 bit, indirizzo IP associato all'host del destinatario del datagram.

- **Options** contiene opzione facoltative e poco usate per usi più specifici del protocollo. La dimensione del campo Options deve essere multipla di 32 bit altrimenti, per raggiungere tale scopo, vengono aggiunti dei bit privi di significato, **padding**.

1.5.2 NAT

NAT Network Address Translation è una tecnica che consente di mappare uno o più indirizzi IP di un *address space* in un indirizzo IP appartenente a uno spazio di indirizzi diverso: questo avviene molto semplicemente modificando le informazioni sugli indirizzi di rete presenti in un datagram IP in transito su di un dispositivo di routing. Questa tecnica è stata originariamente pensata per semplificare le operazioni di rerouting del traffico IP in una rete senza rinumerare ciascun host di quella rete: ad esempio hosts appartenenti a una rete privata possono essere mappati su di un unico indirizzo IP appartenente a uno spazio di indirizzamento differente (tendenzialmente un indirizzo IP pubblico). Questo meccanismo viene realizzato memorizzando in un dispositivo di routing delle *translation table* che mappano gli indirizzi IP non visibili dall'esterno in uno (o più) indirizzi IP pubblici modificando i datagram in uscita facendo sembrare dall'esterno che il mittente del pacchetto è il routing device. Quando viene ricevuto un pacchetto dall'esterno verso la rete interna privata il routing device, che mantiene le translation table, inoltrerà il pacchetto all'effettivo host destinatario.
Questa tecnica è molto utilizzata anche per preservare il consumo di indirizzi IPv4. Gli indirizzi IPv4 essendo formati da 32 bit possono generare un pool di 4.294.967.296 indirizzi: se si considerano i vari diversi indirizzi riservati e la sempre più diffusione di dispositivi che possono connettersi alla rete come laptop, smartphone e IoT questo può rappresentare un limite molto serio. Per tanto a partire dal 2004 è disponibile una nuova versione dell'internet protocol IPv6 che adotta indirizzi di dimensione pari a 128 bit.

1.5.3 ARP

Il protocollo ARP (Address Resolution Protocol) è usato per determinare, dato un indirizzo IP, il corrispettivo indirizzo hardware di livello data-link. Quando un host vuole spedire un datagramma ad un altro nodo conoscendo il suo indirizzo IP ma non quello fisico, fa una richiesta ARP in broadcast

sulla rete di appartenenza, chi ha l?indirizzo richiesto risponde con il proprio indirizzo MAC.
Il corrispettivo protocollo ovvero che consente di risalire all'indirizzo IP di un host dato il suo MAC address è il protocollo RARP (Reverse Address Resolution Protocol).

1.5.4 Frammentazione

Per rendere il protocollo tollerante alle eventuali differenze di specifiche sottoreti è possibile effettuare la frammentazione dei datagram IP. In particolare ogni qual volta un datagram IP deve essere trasmesso attraverso un link con MTU (Maximum Transfer Unit, che rappresenta un limite superiore alla dimensione di un frame di livello data-link dipendente dall'hardware di rete) inferiore alla dimensione del pacchetto stesso questo verrà frammentato in più fragment che soddisfano il valore della MTU. Il router quindi preleverà il payload dal datagram IP e lo frammenterà in modo tale che il nuovo segmento dati più l'header IP possa essere contenuto come body di un frame di livello data-linke e facendo si che ogni payload del datagram abbia dimensione multipla di 8 byte (campo Offset in header IP). In tutti i frammenti tranne l'ultimo inviato, che in genere avrà anche una dimensione minore rispetto ai precedenti, avranno il flag MF (More Fragment) settato a 1.

1.5.5 Routing

Uno dei compiti caratteristici del livello rete è il *routing* ovvero decidere su quale interfaccia di rete o porta inoltrare un pacchetto ricevuto verso la sua destinazione finale. In particolare i router mantengono delle tabelle di instradamento che possono essere popolate manualmente (*routing statico*, poco scalabile adatto esclusivamente per reti molto piccole) o popolate da appositi protocolli di routing che mantengono le tabelle di routing aggiornate a seconda del variare della topologia della rete. Vi sono due tipologie prin-

cipali di protocolli di routing: *Distance Vector* e *Link State*. Nel protocollo Distance Vector ogni router misura la distanza (utilizzando una metrica che può tenere conto di diversi fattori) dai nodi adiacenti tramite le informazioni ricevute dai nodi vicini. In un protocollo Link State, invece, ciascun nodo acquisisce informazione sullo stato dei collegamenti adiacenti e inoltra queste informazioni a tutti gli altri nodi sulla rete attraverso un pacchetto di tipo Link State trasmesso tramite un algoritmo di *link state broadcast*.

1.5.6 IPv6

Internet Protocol version 6[5] è la versione più recente dell'Internet Protocol formalizzato nel 1998 da IETF a seguito della recente crescita esponenziale dei dispositivi che accedono ad internet. IPv6 adotta indirizzi a 128 bit fornendo così 2^{128} indirizzi ovvero circa 3,4 x 10^{38} indirizzi. I protocolli IPv4 e IPv6 non sono stati progettati per essere interoperabili complicando così la transizione verso l'ultima versione dell'Internet Protocol seppure siano stati ideati dei meccanismi di transizione da un protocollo verso l'altro. Tuttavia la migrazione completa verso IPv6 dovrebbe venire nel 2025 con la deprecazione definitiva di IPv4.
IPv6 oltre a estendere lo spazio di indirizzamento porta con se alcune nuove feature come la dimensione fissa dell'header a 40 byte, datagram non frammentabili dai router e la rimozione del checksum dall'header (ridondate in quanto presente in altri layer dello stack di rete). Inoltre è stati aggiunto il supporto nativo alla sicurezza (IPsec) e inseriti meccanismi di autoconfigurazione come ad esempio ARP.
Gli indirizzi IPv6, composti da 128 bit, sono rappresentati come 8 gruppi di 4 cifre esadecimali (8 word di 16 bit ciascuna) in cui le lettere vengono scritte in forma minuscola. Ad esempio fe80:0000:0000:0000:a4f5:f1ff:fe49:8d93 rappresenta un indirizzo IPv6 valido. In un indirizzo IPv6 una sequenza di zeri contigui composta da 2 o più gruppi può essere contratta con la semplice sequenza *::* riducendo così l'indirizzo appena illustrato a fe80::a4f5:f1ff:fe49:8d93.

Datagram IPv6 IPv6 specifica un nuovo header rispetto a IPv4 semplificandolo di molto eliminando alcuni campi poco usati e spostandoli in estensioni separate. Come già detto l'header IPv6 ha una dimensione fissa di 40 byte.

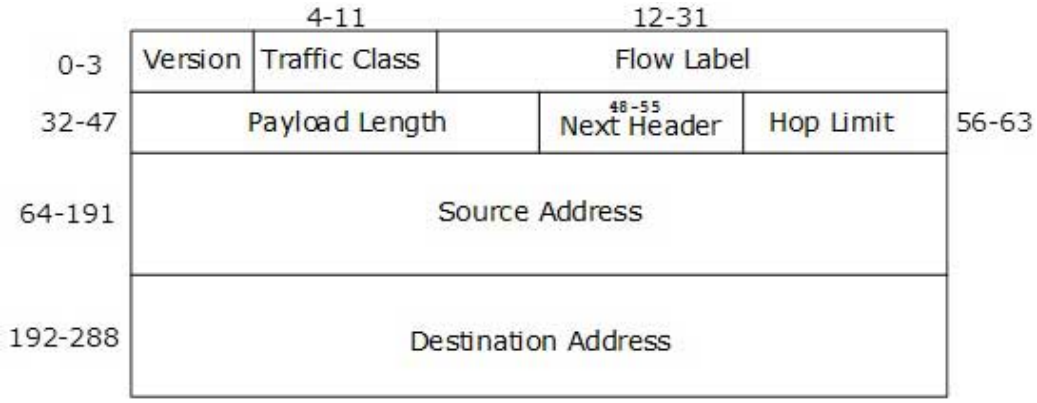

Figura 1.6: Header datagram IPv6

In particolare eventuali estensioni vengono inviate successivamente: il campo *next header* infatti definisce il tipo dell'header successivo nel caso siano inviate delle estensione altrimenti indica il protocollo del pacchetto incapsulato nel payload del datagram. È quindi possibile avere più di un estensione per ciascun datagram. Alcune tra le estensioni più comuni sono:

- **Hop-by-hop options** definisce un insieme arbitrario di opzioni intese per ogni hop attraversato.
- **Routing packet** definisce un metodo per permettere al mittente di specificare la rotta da segure per un datagramma.
- **Fragment packet** usato quando un datagramma viene frammentato.
- **No next header** indica che i dati nel payload del datagramma non sono incapsulati in nessun altro protocollo.
- **Destination options** definisce un insieme arbitrario di opzioni di interesse del solo destinatario.

- **Mobility options** usato per Mobile IPv6.

- **Altri protocolli (TCP, UDP, etc etc.)** indica il protocollo del pacchetto incapsulato nel payload.

1.5.7 Frammentazione in IPv6

Come già brevemente accennato nell'ultima versione dell'Internet Protocol viene rimossa la frammentazione di una datagram IPv6 da parte dei router.

A differenza della versione 4 dell'Internet Protocol, in IPv6, la frammentazione avviene esclusivamente in maniera end-to-end ovvero ciascun datagram sarà frammentato dall'host sorgente e poi riassemblato dall'host destinatario senza ulteriore frammentazione da parte dei router intermedi.
Rimuovendo la frammentazione da parte dei nodi intermedi di una trasmissione viene abbattuto l'overhead dovuto alla framentazione che era a carico dei router incrementando così il numero di pacchetti gestiti per unità di tempo.
Il protocollo IPv6 richiede che ogni link supporti la trasmissione di datagram di dimensione fino a 1280 byte.
In caso di una frammentazione viene utilizzata l'estensione precedentemente accennata *Fragment packet*.
Quando un host deve inviare un datagram IPv6 può valutare se tutti gli eventuali suoi frammenti abbiano una dimensione massima di 1280 byte oppure se utilizzare un algoritmo di discovery che gli consenta di scoprire qual è la dimensione minima supportata su uno dei link da attraversare per arrivare a destinazione. IETF suggerisce l'utilizzo del secondo approccio anche se questo potrebbe rappresentare un limite in quanto il percorso di ciascun datagram sarebbe ben definito e statico: nel caso un router non riesca per qualche motivo instradare il datagram su uno dei link predefiniti manderà un messaggio di errore all'host sorgente che calcolerà un nuove percorso.

1.6 Transport Layer

Il livello trasporto è il quarto livello dello stack di rete ISO/OSI che ha lo scopo di fornire un canale di comunicazione *end-to-end* tra processi residenti in host diversi.
Uno dei diversi protocolli di livello trasporto è il TCP (Transmission Control Protocol). TCP è un protocollo connection-oriented tramite il quale un flusso di dati inviato viene recapitato al destinatario senza errori. I frammenti dello stream vengono impacchettati e passati ai layer inferiori, sarà poi compito del ricevente provvedere a riassemblare ciascun pacchetto ed eventualmente, in caso di errori, chiederne il rinvio. TCP implementa inoltre un meccanismo per il *flow control*, non permette cioè ad una sorgente "veloce" di congestionare un ricevente "lento", assieme al *congestion control* che consente di evitare di congestionare i router che sono tra i due end-point.
Considerato lo scenario e il contesto di questa tesi il protocollo di livello trasporto che più è degno di approfondimento è il protocollo UDP.

Protocollo UDP Il protocollo UDP (User Datagram Protocol)[6] è un protocollo di livello trasporto connection-less e best-effort cioè non assicura che un pacchetto sia effettivamente consegnato a destinazione. Applicazioni in esecuzione su host differenti possono scambiarsi messaggi, detti datagram, consegnandoli su appositi socket dedicati. Essendo un protocollo inaffidabile una volta inviato un datagram verso la rete l'applicazione sorgente non potrà mai sapere se effettivamente è stato consegnato a destinazione o meno. Ciascun datagram inviato è indipendente dai restanti datagram inviati ed inoltre non viene effettuata alcun tipo di frammentazione a livello trasporto. Non viene effettuato alcun tipo di controllo nemmeno per quanto riguarda l'ordine con cui i pacchetti vengono ricevuti dall'end-system. Tutti questi motivi UDP è un protocollo molto leggero e veloce ideale per applicazioni che necessitano di interattività così come ad esempio servizi di streaming audio/video in real-time.

L'header UDP, visti i pochi controlli di cui si fa carico il protocollo, risulta essere molto snello introducendo così un overhead molto basso.

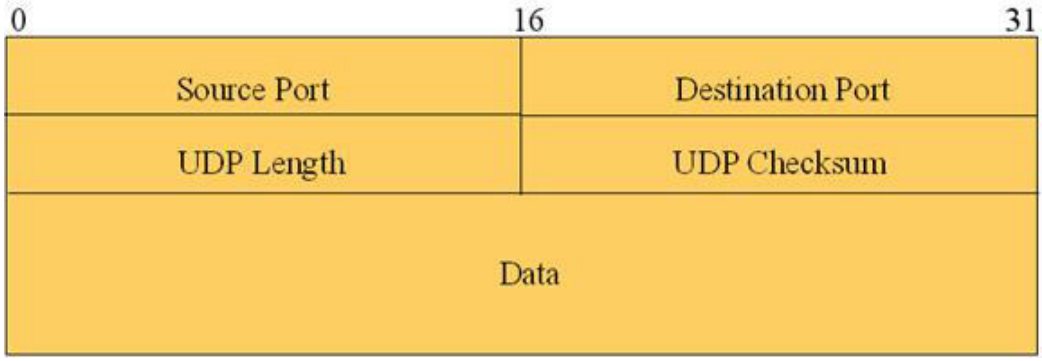

Figura 1.7: Header UDP

Nell'header UDP, oltre alle porte che indicano sostanzialmente le applicazioni (sorgente e destinataria) coinvolte nelle comunicazione, vi è il campo checksum. Questo campo è opzionale se la trasmissione avviene over IPv4 mentre obbligatorio se si utilizza IPv6 (header IPv6 non contiene campo sull'integrità datagram).

1.7 Application Layer

Il livello applicazioni si colloca nella parte più alta dello stack ISO/OSI e coinvolge tutta una serie di protocolli che si occupano di fornire servizi per i processi delle applicazioni usate dagli utenti finali. I protocolli appartenenti a questo livello sono davvero numerosissimi e utilizzati dalle più disparate applicazioni come ad esempio FTP (File Transfer Protocol) o HTTP (HyperText Transfer Protocol) alla base del World Wide Web.
Se considerato il livello Application dello standard *de jure* TCP/IP questo include anche i livelli presentazione e sessione del modello OSI. Per quanto riguarda il livello applicazione si vuole ora approfondire i protocolli coinvolti nella tecnologia VoIP in quanto la applicazioni che sfruttano questa tecnologia possono toccare con mano l'oggetto di questa tesi.

1.7.1 VoIP

VoIP (Voice over IP)[7] è una famiglia di tecnologie internet e protocolli di comunicazione per la distribuzione di comunicazioni vocali e sessioni multimediali attraverso il protocollo IP.
Il VoIP richiede due tipologie di protocolli di comunicazione in parallelo, una per il trasporto dei dati (pacchetti voce su IP), ed una per la segnalazione della conversazione che comprende la ricostruzione del frame audio, la sincronizzazione, l'identificazione del chiamante lo scambio di altri parametri di comunicazione così come indirizzp IP, porte e codec audio/video.
Per quanto riguarda la trasmissione dati la maggioranza delle implementazioni VoIP adottano il protocollo RTP (Real-time Transfer Protocol, basato su UDP). RTP viene usato assieme a RTPC (RTP Control Protocol) che monitora la *qualità del servizio* (QoS) inviando periodicamente delle statistiche ai partecipanti dello streaming multimediale senza però trasportare alcun tipo di dato relativo alla comunicazione vera e propria. È stato inoltre definito SRTP (Secure Real-time Transfer Protocol) che introduce crittografia, autenticazione e integrità in RTP.
Per quanto concerne i protocolli di segnalazione, invece, nel corso degli anni ne sono stati sviluppati diversi. H.323[9] definito dalla ITU-T (International Telecommunications Union) è stato uno dei primi protocolli pensati per il VoIP. H.323 nasce in ambito telefonico e delinea un'architettura completa per lo svolgimento di conferenze multimediali. Comprende la definizione dei formati di codifica a livello applicativo, la definizione dei formati per la segnalazione e il controllo, per il trasporto dei flussi multimediali (audio, video e dati). H.323 definisce anche alcuni meccanismi legati agli aspetti relativi alla sicurezza.
Vi è poi SIP (Session Initiation Protocol)[10] definito da IETF, operante a livello sessione del modello OSI, che gestisce in modo generale una sessione di comunicazione tra due o più entità, ovvero fornisce meccanismi per instaurare, modificare e terminare (rilasciare) una sessione. Fornisce funzionalità di instaurazione e terminazione di una sessione, operazioni di segnalazione ,

tono di chiamata, chiamata in attesa, identificazione del chiamante e molto altro ancora. Adotta un un pattern message/response e i messaggi principali di SIP sono:

- REGISTER, inviato da un client verso un server SIP che mantiene una mappa degli utenti SIP e la loro posizione (indirizzi IP).
- INVITE, utilizzato per inizializzare una sessione di comunicazione specificando l'identificativo dell'utente con la quale si vuole comunicare; inviato da un client verso un server SIP che risponde con l'indirizzo dell'altro end-node con la quale si vuole avviare la sessione assieme ad altri parametri di configurazione.
- re-INVITE, viene utilizzato quando uno dei parametri di configurazione cambia (ad esempio l'indirizzo IP).

SIP è più recente e viene largamente utilizzando riscontrando un maggiore successo di H.323.

Capitolo 2

A glance to kernel Linux and to its network modules

In questo capitolo verrà offerta una breve panoramica sul kernel Linux. Verrà inoltre descritto un aspetto del kernel Linux che risulta cruciale per lo sviluppo del progetto che questa tesi si propone di raccontare: i suoi moduli di rete.
Ma prima di addentrarci all'interno del kernel Linux, e dei suoi moduli di rete, verrà introdotto brevemente il concetto di kernel.

2.1 Cos'è un kernel?

Il kernel rappresenta il nucleo di un sistema operativo e racchiude tutte le funzioni principali del sistema stesso come gestione della memoria, gestione delle risorse, lo scheduling e il file system.
Le applicazioni in esecuzione nel sistema possono richiedere particolari servizi al kernel tramite chiamate di sistema (system call) senza accedere direttamente alle risorse fisiche.
L'accesso diretto all'hardware può risultare anche molto complesso pertanto il kernel implementa una o più astrazioni dell'hardware, il cosiddetto Hardware Abstraction Layer. Queste astrazioni servono a nascondere

la complessità e a fornire un'interfaccia pulita ed omogenea dell'hardware sottostante.
I kernel si possono classificare in quattro categorie:

- **Kernel monolitici** un unico aggregato di procedure di gestione mutuamente coordinate e astrazioni hardware.
- **Micro kernel** semplici astrazioni dell'hardware gestite e coordinate da un kernel minimale, basate su un paradigma client/server, e primitive di message passing.
- **Kernel ibridi** simili a micro kernel con la sola differenza di eseguire alcune componenti del sistema in kernel space per questione di efficienza.
- **Exo-kernel** non forniscono alcuna astrazione dell'hardware sottostante ma soltanto una collezione di librerie per mettere in contatto applicazioni con le risorse fisiche.

2.2 Il kernel Linux

Nell'aprile del 1991 Linus Torvalds, uno studente finlandese di informatica, comincia a sviluppare un semplice sistema operativo chiamato Linux.
L'architettura del kernel sviluppato da Torvalds è di tipo monolitico a discapito di una struttura più moderna e flessibile come quella basata su micro kernel.

I

Sebbene oggi il kernel possa essere compilato in modo da ottenere un'immagine binaria ridotta al minimo, e i driver possono essere caricati da moduli esterni, l'architettura originaria è chiaramente visibile: tutti i driver, infatti, devono avere una parte eseguita in kernel mode, anche quelli per cui ciò non sarebbe affatto necessario (come ad esempio i driver dei file system).
Attualmente il kernel Linux è distribuito con licenza GNU General Public

License ed è in continua evoluzione grazie ad una vastissima comunità di sviluppatori da ogni parte del mondo che contribuisce attivamente al suo sviluppo.
Il kernel Linux trova larghissima diffusione ed utilizzo: infatti grazie alla sua flessibilità viene utilizzato dai personal computer ai grandi centri di calcolo, dai più moderni sistemi embedded agli smartphone.
Android, il sistema mobile più diffuso al mondo, si basa su una versione lightweight del kernel Linux.

2.3 Linux kernel v4.0

Il 12 Aprile 2015 è stato rilasciata la versione 4.0 del kernel Linux. Il cambio di versione da 3.x a 4.0 non è dovuto a nessun particolare miglioramento del kernel: la nuova versione stabile sarebbe dovuta essere la 3.20 ma su proposta di Linus Torvalds si è deciso di incrementare la numerazione alla versione 4.0 per non creare confusione con numeri molto grandi.
La nuova versione del kernel, quindi, non introduce particolari nuove feature [11]. Una delle novità degna di nota è pero il *live patching*: ovvero la possibilità di installare pacchetti e estensioni al kernel senza dover riavviare la macchina. Questo può essere cruciale, ad esempio, per aspetti legati alla sicurezza.

2.4 Linux networking

I moduli del networking (e relativi device driver) occupano buona parte del codice del kernel Linux.
Vi sono due strutture particolarmente importanti utilizzante largamente in tutto lo stack di rete del kernel Linux.

Socket buffer La struttura sk_buff [12] mantiene le informazioni relative a ciascun pacchetto inviato o ricevuto lungo lo stack di rete di Linux.

La struttura socket buffer contiene diversi campi che memorizzano le informazioni del pacchetto come il puntatore alla queue a cui è accodato o il socket a cui è associato.
Essendo lo stack di rete implementato su più livelli, dove ciascun layer aggiunge (o rimuove) delle proprie informazioni di intestazione ad un messaggio, mantenere per ogni livello di rete una struttura diversa per identificare un certo pacchetto appartenente a quel dato layer potrebbe risultare parecchio oneroso in termini di memoria in quanto si finirebbe per copiare grandi quantità di dati da un buffer ad un altro. Anche per questo motivo, nel kernel Linux, si è deciso di mantenere un'unica struttura accessibile da qualsiasi layer dello stack di rete.
La struttura sk_buff mantiene un union field per ciascun livello di rete dello stack TCP/IP (trasporto, network, data-link) dove ciascun campo (h, nh, mac) rappresenta un header di un protocollo di comunicazione adottabile per quel livello. Ciascun campo di queste union punterà effettivamente a un header per quel livello.
Il campo *data* punta all'inizio di tutti i dati relativi al pacchetto (header compresi) variando a seconda del layer in cui il socket buffer è utilizzato: in particolare quando una funzione ad un certo livello dello stack di rete tratta una struttura sk_buff il campo data punterà all'header del messaggio per quel livello. Ad esempio in fase di ricezione di un pacchetto a livello network il campo nh di sk_buff sarà inizializzato all'header puntato dal campo data. Se si vorrà accedere, per qualche motivo al payload del datagram IP si potrà farlo calcolando l'offset a partire dal puntatore data; inoltrando il pacchetto ad un layer superiore l'header relativo al livello di rete corrente potrà essere rimosso tramite la routine **skb_pull()**.
In fase di invio il procedimento è molto simile ma con la difficoltà di dover appendere eventuali header man mano che si percorre lo stack di rete verso il basso. A tale scopo vi è una funzione di manipolazione delle strutture sk_buff chiamata **skb_reserve**, in genere invocata dopo un allocazione di un socket buffer, che consente di riservare dello spazio per gli header dei diversi

protocolli.

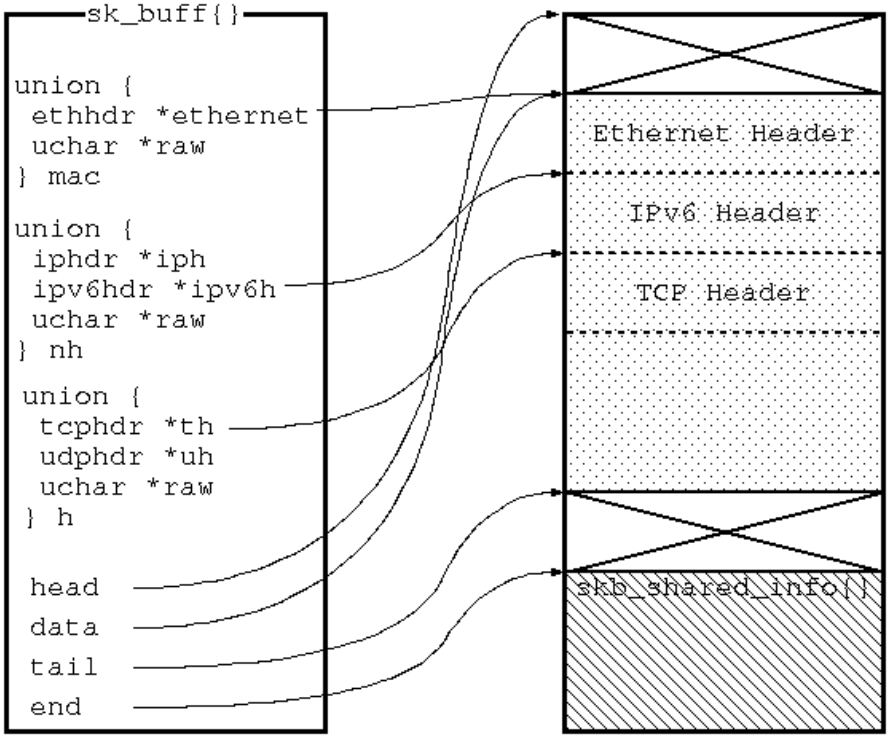

Figura 2.1: La struttura *sk_buff*.

struct net_device questa struttura rappresenta una scheda di rete, anche virtuale: può essere infatti che ci si riferisca ad una scheda di rete virtuale ottenuta dopo aver effettuato un *bonding*, ovvero è possibile assegnare un unico indirizzo IP a due o più NIC (vedendole così come un unico device) in modo da migliorare le performance.
La struttura net_device contiene campi che identificano l'interfaccia di rete come il valore massimo della sua MTU e l'indirizzo MAC.
Questa struttura viene inizializzata da un device driver non appena questo comincia la sua esecuzione.

Le operazioni offerte verso il kernel dalla interfaccia di rete sono contenute della struttura **const struct net_device_ops *netdev_ops**; la struttura net_device_ops descrive un'interfaccia generica e omogenea per la gestione di tutti i device di rete.
I campi contenuti nell'interfaccia net_device_ops non sono altro che semplici puntatori a funzione.
Il device driver, in fase di inizializzazione, si occuperà di settare i campi della struttura net_device_ops mappando l'interfaccia offerta verso l'esterno con le funzioni implementate dal device driver per l'hardware specifico.

Nello sviluppo del progetto di tesi è stato esteso il comportamento del workflow del kernel Linux per quanto riguarda la trasmissione di un datagram UDP. Per tanto, qui di seguito, verranno descritti buona parte dei moduli che rendono possibile la trasmissione di un pacchetto UDP a partire da un applicazione fino al device driver.

2.4.1 Dall'app al device driver

Quando viene invocata una routine che interagisce con la rete tramite socket (come ad esempio send, sendto o sendmsg) il controllo viene passato al *socket interface layer*: se il socket è di tipo TCP il controllo verrà dato alla funzione tcp_sendmsg altrimenti, se di tipo UDP, alla primitiva udp_sendmsg. Appena il flusso dell'esecuzione è passato alla routine udp_sendmsg vengono effettuati alcuni controlli sui parametri passati come ad esempio sulla lunghezza specificata per il payload del datagram UDP o sugli indirizzi mittente e destinatario e le porte. Dopodiché, a seconda se specificato il flag UDP_CORK o meno (che consente quando abilitato tramite la system call *setsockopt* di accumulare tutti i dati in output sul socket e inviarli in un unico datagram quando disabilitato), il controllo dell'esecuzione passerà alla primitiva di livello rete ip_append_data() oppure a ip_skb_make(). In entrambi casi, comunque, queste due funzioni si appoggeranno sulla funzione __ip_append_data() che tra le altre cose si occuperà di bufferizzare tutte le

richieste di tramissione e di generare uno o più sk_buff rappresentanti ciascun pacchetto (o frammento) IP.
A questo punto il flusso esecutivo torna a livello trasporto dove verrà invocata la primitiva udp_push_pending_frames che si occupa di generare l'header UDP e passare definitivamente il controllo a livello rete invocando la funzione ip_push_pending_frame().
In ip_push_pending_frame() verrà generato l'header IP per ogni socket buffer in coda sulla queue in uscita per quel socket e dopo diversi controlli, se il datagram IP necessita di frammentazione, la funzione ip_fragment si occuperà di frammentare il pacchetto IP in diverse strutture dati di tipo sk_buff.
Tra i compiti a carico del livello network dello stack di rete vi è quello del *routing*: in particolare a supporto del routing a livello rete vi è la funzione ip_route_output_flow().
Se la fase di routing termina con successo verrà invocata la funzione di livello *data-link* dev_queue_xmit() che, se l'interfaccia richiesta per la trasmissione è attiva, accoderà il socket buffer nella coda in uscita del net_ device associato all' sk_buff in invio e invocherà la funzione dev_hard_start_xmit per cominciare a processare l'invio di un socket buffer.
La funzione dev_hard_start_xmit utilizzerà l'interfaccia esposta dalla struttura net_device_ops per cominciare la trasmissione e, in particolare, invocherà la funzione ndo_start_xmit().
Nel caso di un interfaccia di rete *softMAC*, ovvero che il MLME (MAC Sublayer Management Entity, cioè dove viene implementata la logica del Medium Access Control) è implementato via software (vi sono dispositivi detti *fullMAC* che implementano il MLME direttamente in hardware) questa funzione si appoggerà sul sottosistema mac80211.

Sottosistema mac80211 Il sottosistema mac80211 è stato rilasciato nel 2005 e si occupa di implementare la logica MLME per device softMAC interpretando e generando frame IEEE 802.11. Prima della sua adozione all'interno del kernel Linux la gestione e l'implementazione di IEEE 802.11 era

lasciata ai device driver.
Oggi la maggior parte dei device supportano questo sistema e in molti dei vecchi dispositivi i device driver sono stati riscritti tenendo conto di esso.
Un device driver in fase di inizializzazione delle strutture net_device da lui gestite può *registrarle* presso il modulo mac80211.
In questo modo alcuni dei campi contenuti nella struttura net_device_ops possono essere settati con dei callback offerti dal modulo mac80211.
Alcuni campi della struttura net_device_ops debbono essere settati obbligatoriamente con i callback del modulo mac80211 nel caso si voglia beneficiare del sottosistema; altri sono opzionali (o più legati all'hardware specifico) e possono essere implementati dal device driver stesso.

In fase di trasmissione di un messaggio il controllo dal livello data-link viene ceduto, quindi, al sottosistema mac80211; questo è possibile in quanto, in fase di registrazione del device, la funzione ndo_start_xmit() di net_device_ops viene mappata nella funzione del sottosistema mac80211 ieee80211_monitor_start_xmit().
Verrà poi invocata la entry point **ieee80211_xmit** del modulo mac80211 che tra le altre cose si occupa dell'inizializzazione del frame 802.11 e del suo header.
Per far ciò il sottosistema mac80211 si serve di una serie di handler appositamente settati, tra cui:

- **ieee80211_tx_h_dynamic_ps()** per la gestione del power saving.
- **ieee80211_tx_h_select_key()** si occupa di selezionare una chiave di cifratura.
- **ieee80211_tx_h_michael_mic_add()** si occupa di aggiungere il Message Integrity Code al frame IEEE 802.11.
- **ieee80211_tx_h_rate_ctrl()** che seleziona il bit rate con la quale il frame IEEE 802.11 deve essere trasmesso.

- **ieee80211_tx_h_sequence()** calcola il *sequence control* del frame e lo assegna all'header 802.11.
- **ieee80211_tx_h_fragment()** si occupa, eventualmente, di frammentare il frame IEEE 802.11 in base al valore della MTU della scheda di rete wireless e a quella dell'access point.
- **ieee80211_tx_h_encrypt()** cifra il frame IEEE 802.11 con l'algoritmo designato.
- **ieee80211_tx_h_calculate_duration()** si occupa di calcolare il campo *duration* del frame IEEE 802.11 che indica per quanto tempo il canale sarà impegnato dalla trasmissione del frame.
- **ieee80211_tx_h_stats()** setta una serie di variabili utili al mantenimento di alcune statistiche di trasmissione.

Una volta che il frame IEEE 802.11 è stato trasmesso la struttura socket buffer associata non viene immediatamente deallocata: mac80211 provvederà a ritrasmettere il messaggio per un certo numero di volte, dipendente dalla NIC e dallo stato in cui si trova la rete alla quale è connessa, qualora non dovesse ricevere l'ACK relativo a quel frame.
Alla fine di questo processo viene invocata la routine ieee80211_tx_status che fornisce alcune informazioni riguardanti l'esito della trasmissione per un certo pacchetto (come ad esempio quante volte il pacchetto è stato ritrasmesso e se è stato effettivamente consegnato o meno all'access point).
A questo punto, sia che il frame IEEE 802.11 sia stato ricevuto dal first-hop o meno, il socket buffer associato viene definitivamente deallocato.

Capitolo 3

Scenario

Attualmente sempre più dispositivi mobili offrono diverse modalità di accesso ad internet: basti pensare ad uno smartphone che consente l'accesso alla rete sia tramite connessione Wi-Fi che attraverso la telefonia mobile sfruttando tecnologie come il 3G o il più recente 4G.
In questo capitolo si vuole descrivere un'architettura che consente ad un user app che fornisce un servizio multimediale in esecuzione su di un mobile device *multi-homed*, ovvero che monta più di un'interfaccia di rete, di sfruttare tutte le sue NIC (Network Interface Card) in un contesto di mobilità.
Può essere interessante studiare uno scenario che fa uso di un'app multimediale (come può essere ad esempio un'applicazione per chiamate VoIP) in quanto sono app che operano in *real-time* e generano una grande mole di traffico.
Le app appartenenti alle categorie dei servizi multimedial real-time vengono solitamente realizzate sfruttando il protocollo UDP: l'architettura presentata in questo capitolo fornisce un meccanismo in grado da rendere possibile l'inoltro di ciascun datagram UDP attraverso l'interfaccia di rete più adatta e disponibile al momento della trasmissione.
Questa architettura è detta ABPS, Always Best Packet Switching, ma prima di descriverne i suoi principali aspetti verrà descritto lo stato dell'arte per quel che riguarda la gestione della mobilità dei nodi mobili.

3.1 Seamless Host Mobility & State Of the Art

Nel corso degli ultimi anni sono state sviluppate diverse architetture che consentono ad un nodo mobile in movimento di avere accesso continuo a servizi di rete; in particolare sono stati sviluppati diversi approcci che cercano di fare in modo che, device montanti più interfacce di rete, possano effettuare un *handover* da un' interfaccia di rete ad un' altra in maniera del tutto trasparente per l' app utente in esecuzione, ovvero in maniera *seamless*.
Per handover o handoff si intende il processo di cambio di interfaccia di telecomunicazione da parte di un dispositivo multi-homed (*vertical handoff*) oppure il cambio di punto di accesso mantenendo la stessa tecnologia di telecomunicazione (*horizontal handoff*, ad esempio cambio di AP all'interno di una stessa rete WLAN).
In generale una buona architettura per la seamless mobility dovrebbe essere responsabile di identificare univocamente ciascun nodo mobile permettendogli di essere raggiungibile dall'altro nodo coinvolto nella comunicazione (Correspondent Node che potrebbe essere anch'esso un nodo mobile) e dovrebbe, inoltre, monitorare la QoS fornita dalle diverse reti a cui il nodo mobile potrebbe connettersi in modo tale da prevedere la necessità di un handoff ed eventualmente eseguirlo in maniera *seamless* assicurando la piena continuità della comunicazione.
Vediamo ora una rassegna di tutte le soluzioni sviluppate finora per implementare meccanismi di seamless handoff in un contesto di un nodo mobile che attraversa reti eterogenee.

Implementazioni a livello network Tra le architetture presenti che lavorano a livello network vi è Mobile IP version 6 e le sue ottimizzazioni come ad esempio FMIP (Fast Handover Mobile IPv6)[13], HMIP (Hierarchical Mobile IPv6)[14] e PMIP (Proxy Mobile IPv6)[15] .
Tutte queste architetture adottano un *Home Agent* ovvero un'entità aggiun-

tiva che opera all'interno della rete alla quale il nodo mobile appartiene. L'home agent ricopre il ruolo di *location registry* che offre un servizio *always on*: quando un nodo mobile cambia interfaccia di rete, e quindi indirizzo IP, lo comunica al location registry (*registration phase*) che tiene una mappa degli indirizzi. Quando un Correspondent Node vuole comunicare con il nodo mobile invia al location registry una richiesta per ottenere l'indirizzo corrente del mobile node (*lookup phase*).
Tutti i nodi coinvolti devono avere il supporto a IPv6: in particolare l'indirizzo attuale e l'identificativo univoco del nodo mobile sono trasmessi attraverso delle estensioni di IPv6. Il fatto che tutti i nodi debbano necessariamente supportare IPv6 rappresenta un limite di questo approccio architetturale. Un altro limite di questo approccio è che presso un home agent può essere registrato l'indirizzo di una sola interfaccia di rete per ogni nodo impedendo così un supporto al multihoming: la latenza introdotta dai numerosi messaggi di autenticazione procurerebbe un overhead insostenibile per la tipologia di comunicazione multimediale che dovrebbe essere veloce e snella.

Implementazioni tra livello rete e trasporto Esistono alcune possibili implementazioni di architetture per la seamless host mobility che introducono un nuovo layer posto tra il livello rete e quello trasporto: questa nuova astrazione dovrà essere aggiunta su tutti i nodi prendenti parte alla comunicazione.
Alcuni esempi possono essere HIP (Host Identity Protocol)[16] e LIN6 (Location Independent Addressing for IPv6)[17] . Il location registry si comporta in maniera simile a un server DNS mappando l'identificativo di un host e la sua attuale posizione restando però all'esterno della rete di appartenenza del nodo mobile. Un limite di questo approccio è nella necessità di modificare lo stack di rete di tutti i nodi coinvolti.

Implementazioni a livello trasporto Vi sono alcuni protocolli che operano a livello trasporto: ogni nodo coinvolto in una comunicazione si comporta in maniera pro-attiva fungendo da location registry informando diret-

tamente il proprio Correspondent Node ogni qualvolta la configurazione di rete cambia. Il limite di questo approccio sta nel fatto che se entrambi i mobile end-system cambiano contemporaneamente gli indirizzi IP a seguito di un handoff diventano mutuamente irraggiungibili. Inoltre, come nel precedente approccio, questo tipo di architettura richiederebbe una modifica delle applicazioni sia sul nodo mobile che sul suo correspondent node.

Implementazioni a livello Sessione Sono state progettato alcune soluzioni che operano a livello sessione come ad esempio TMSP (Terminal Mobility Support Protocol)[18] che sfrutta un SIP server ausiliario collocato fuori dalla rete di un nodo mobile che funge da location registry che mappa ciascun identificativo SIP di utente al suo indirizzo IP attuale. Ogni nodo mobile esegue un client SIP che manda un messaggio di tipo REGISTER per aggiornare il suo indirizzo IP. I messaggi INVITE, al solito, sono utilizzati per avviare comunicazioni con altri nodi così come i messaggi di tipo re-INVITE. Gli approcci operanti a livello Session non sembrano essere particolarmente efficienti per i ritardi introdotti dal pattern message/response dei sistemi basati SIP. In letteratura vi sono altre soluzioni alcune parziali come IEEE 802.11e[20] e IEEE 802.11r[19] che però coinvolgono solamente la gestione dell'interfaccia di rete Wi-Fi (handover orizzontali) e soluzioni come LISP di CISCO[23].

3.2 Always Best Packet Switching

Un'architettura progettata all'interno del Dipartimento di Informatica dell'Università di Bologna che supera molti dei limiti delle implementazioni precedentemente descritte è ABPS (Always Best Packet Switching)[21]. L'architettura ABPS è composta da due componenti principale:

- **Fixed proxy server**, una macchina esterna alla rete in cui si trova il mobile node; munito di IP pubblico statico e fuori da qualsiasi firewall o NAT. Il fixed proxy server gestisce e mantiene tutte le comunicazione

da un mobile node verso l'esterno e viceversa: nel caso di un handoff e quindi di una riconfigurazione delle interfacce di rete del nodo mobile il fixed proxy server nasconde questi cambiamenti al correspondent node facendo si che la comunicazione continui in maniera del tutto trasparente.

- **Proxy client**, in esecuzione su ogni mobile node, mantiene per ogni NIC una connessione verso il fixed proxy server. Applicazioni in esecuzione su un nodo mobile possono quindi sfruttare un multi-path virtuale creato tra il proxy client e il fixed proxy server per comunicare con il resto del mondo.

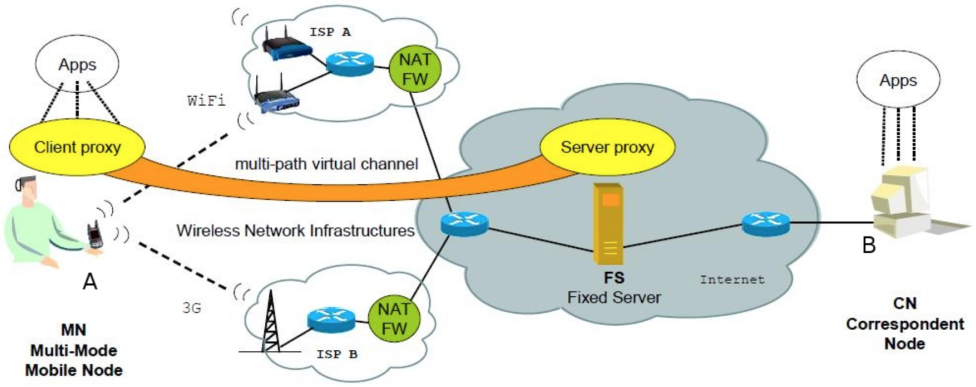

Figura 3.1: Architettura ABPS

Si immagini uno scenario in cui un nodo mobile A munito di più interfacce wireless (ad esempio NIC Wi-Fi e UMTS) stia intrattenendo una comunicazione VoIP con un altro nodo mobile B.
Il nodo A adotta il meccanismo ABPS appena descritto quindi sul dispositivo è in esecuzione un ABPS proxy client che mantiene un canale di comunicazione con un ABPS fixed proxy. Supponiamo che il nodo A sta utilizzando l'interfaccia di rete Wi-Fi e quindi risulta essere connesso a una rete WLAN. Se avviene un handoff verso un'altra interfaccia di rete, ad esempio UMTS, a

seguito di un calo delle prestazioni o per una perdita improvvisa del segnale dell'access point a cui è connesso il nodo mobile il cambio di configurazione di rete avviene in maniera del tutto trasparente al nodo B e alle applicazioni in esecuzione sul nodo A.
Il meccanismo descritto può, inoltre, decidere su quale interfaccia di rete inoltrare il singolo datagram UDP a seconda delle condizioni della rete a cui ciascuna NIC è connessa. In particolare vi è un monitoraggio del QoS per ciascuna interfaccia di rete wireless e se vi è il sospetto di una perdita di informazioni o di un ritardo di trasmissione un dato pacchetto può essere ritrasmesso attraverso un'altra NIC.
Ciascuna interfaccia di rete rimane configurata, attiva e pronta a essere utilizzata nel caso in cui l'interfaccia attualmente in uso abbia dei ritardi o delle perdite.

Qui di seguito viene illustrato un esempio di una possibile implementazione dell'architettura ABPS per il mantenimento di un servizio VoIP basato sui protocolli SIP e RTP/RTCP[22] .
Come si può vedere dalla figura 3.2 il nodo mobile, oltre alla user app VoI, mantiene attivo un client proxy: i pacchetti trasmessi dall'app vengono intercettati dal proxy client mantenendo attiva un'interfaccia di rete virtuale settata come default gateway. Per ogni interfaccia di rete reale il proxy client inizializza e mantiene attivo un socket per ogni protocollo di comunicazione e segnalazione.

Il fixed proxy server regola il traffico VoIP in invio o ricezione dal mobile node esponendo verso un SIP server (ad esempio ekiga.net) e il correspondent node un indirizzo IP pubblico e statico e delle porte associate a ciascun protocollo VoIP. La comunicazione tra il client proxy e il fixed server proxy può fare uso di un'estensione del protocollo SIP per lo scambio di ulteriori parametri di configurazione (ad esempio identificativo per individuare ciascun proxy client che comunica con ABPS proxy server).

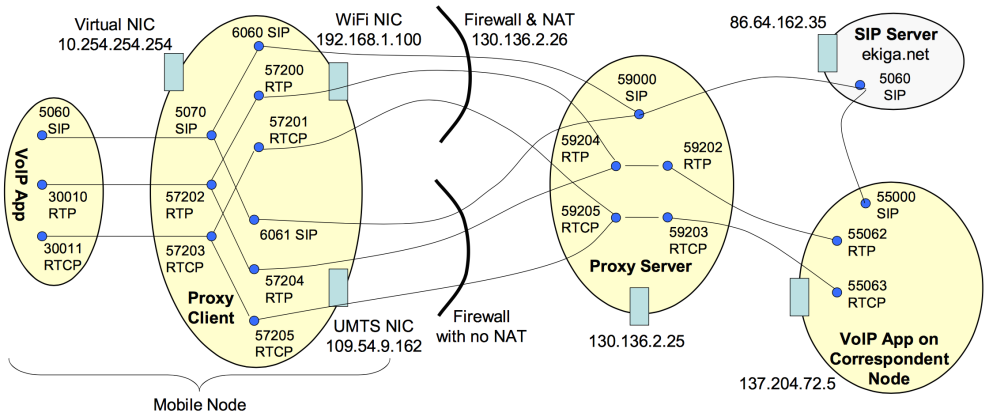

Figura 3.2: Architettura ABPS per una comunicazione VoIP via protocollo SIP e RTP/RTCP

3.2.1 Architettura

Vediamo ora brevemente le caratteristiche principali delle componenti dell'architettura Always Best Packet Switching.

Fixed proxy server Sul fixed proxy server è presente un modulo software chiamato *Policies Module*. Il compito del Policies Module, molto semplicemente, è quello di valutare su quale interfaccia di rete inoltrare un messaggio diretto verso un mobile node: verrà utilizzato l'indirizzo IP mittente dell'ultimo pacchetto ricevuto dal proxy client.

Mobile node L'architettura per supportare ABPS su un nodo mobile risulta essere piuttosto complessa. È composta da:

- **Monitor**, il suo compito principale è quello di monitorare e configurare le diverse interfacce di rete wireless presenti sul nodo mobile. Ogniqualvolta una NIC viene configurata o disabilitata il Monitor invia una notifica al proxy client di tipo *Reconfiguration Notification*.
- **TED (Transmission Error Detector)**, è il componente più importante del sistema; si occupa di monitorare l'invio dei datagram UDP

trasmessi dall'app VoIP e di notificare al client proxy se il pacchetto è stato consegnato o meno all'access point. Una volta inviato un certo pacchetto, attraverso la scheda di rete Wi-Fi, TED valuterà il relativo ACK proveniente dall'access point e tramite la notifica *First-hop Transmission Notification* notificherà al proxy client lo status di consegna di quel pacchetto. TED è implementato in maniera cross-layer nello stack di rete del kernel Linux. TED e la sua implementazione saranno largamente descritti nel capitolo successivo.

- **Wvdial** si tratta di un modulo che implementa Transmission Error Detector per UMTS.

- **Proxy client**, il cui ruolo principale è quello di inoltrare il traffico di una user app verso il ABPS fixed proxy server.
  Il proxy client al suo interno implementa il modulo *ABPS Policies* che implementa una serie di politiche e meccanismi in base alle notifiche provenienti dall'altre componenti in esecuzione sul nodo mobile e dirette verso il proxy client. Le tipologie di notifiche che possono essere ricevute sono quelle precedentemente accennate. Quando il proxy client riceve una notifica di tipo Reconfiguration Notification per ogni nuova interfaccia di rete segnalata come attiva viene creato un nuovo socket e associato a tale interfaccia; viceversa quando un'interfaccia viene segnalata come disabilitata (a seguito ad esempio di un errore di trasmissione) il relativo socket associato viene chiuso.
  Un'altro tipo di notifica che un proxy client può ricevere è quella proveniente dal TED: in base a quanto notificato da TED la componente ULB (UDP Load Balancer), un altro modulo implementato all'interno del proxy client, valuterà in base alla notifica se ritrasmettere un dato datagram e attraverso quale interfaccia di rete.
  L'ultimo tipo di notifica che un proxy client può ricevere è quella proveniente dal protocollo ICMP: semplicemente questa notifica segnala al proxy client che il fixed proxy server (o la porta sul proxy server con

la quale si vuole comunicare) è *unreachable* attraverso l'interfaccia di rete attualmente in uso.
Queste tre tipologie di notifiche permettono al proxy client di stabilire se un determinato pacchetto debba essere ritrasmesso (eventualmente attraverso un'altra interfaccia wireless) o definitivamente scartato. Permettono inoltre di realizzare un opportuno algoritmo per la selezione dinamica dell'interfaccia di rete che offre maggiori garanzie di trasmissione.

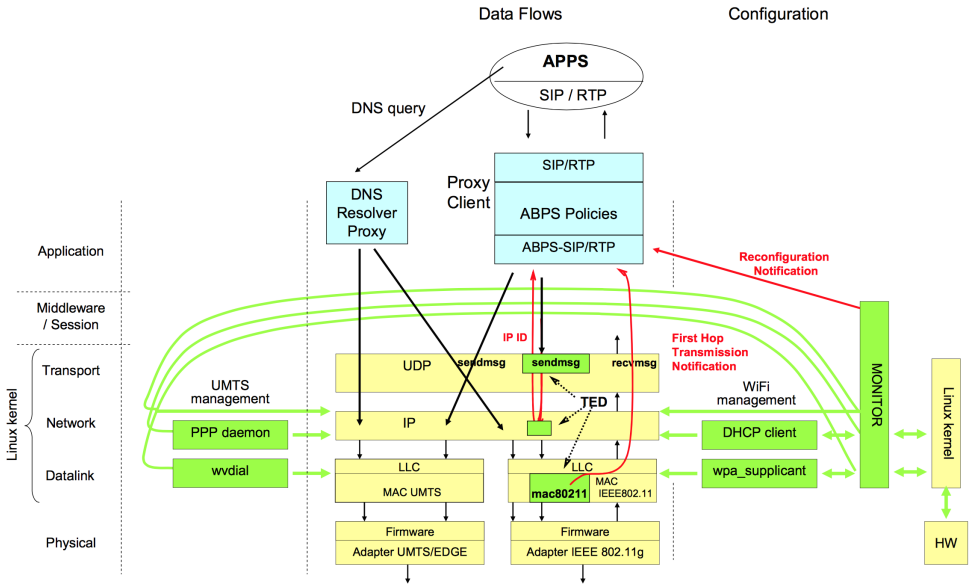

Figura 3.3: Infrastruttura Mobile Node in ABPS

3.3 Considerazioni finali

In questo capitolo è stata presentata una panoramica dell'architettura Always Best Packet Switching.
Abbiamo visto come senza modificare l'infrastruttura di rete è possibile inoltrare il flusso di dati proveniente da un certa applicazione utente su una piuttosto che su un'altra interfaccia di comunicazione di un nodo mobile a

seconda dello stato in cui si trova la rete.

Nel caso di una riconfigurazione di rete ABPS abbatte eventuali overhead introdotti da altri approcci (come ad esempio quello basato su SIP) e può far fronte ad handoff verticali senza alcun ritardo in quanto, come visto, ciascuna interfaccia di rete viene mantenuta attiva e pronta all'utilizzo.

L'utilizzo di ciascuna NIC è ottimizzato valutando pacchetto per pacchetto su quale interfaccia di rete questo dovrebbe essere inoltrato valutando le condizioni della rete e le QoS desiderate. Inoltre ABPS consente a un nodo mobile di comunicare verso l'esterno anche in presenza di NAT o firewall.

ABPS può essere utilizzato in qualsiasi contesto VoIP.

Capitolo 4

Transmission Error Detector

In questo capitolo verranno descritte e illustrate le fasi di progettazione e implementazione di un modulo Transmission Error Detector per ABPS su kernel Linux 4.0.
Nel precedente capitolo si è descritto brevemente il funzionamento di TED e il suo ruolo nel sistema ABPS ideato come supporto alla mobilità.
Il meccanismo di Transmission Error Detection potrebbe, tendenzialmente, essere applicato a qualsiasi tipo di interfaccia di rete di un dispositivo mobile. In questa capitolo verrà illustrata la progettazione e l'implementazione di un modulo TED per Wi-Fi.

4.1 Design and implementation

Transmission Error Detector è implementato in maniera *cross-layer* lungo lo stack di rete del kernel Linux e ha lo scopo di:

- monitorare ciascun datagram UDP in invio da una certa interfaccia di rete;

- notificare ad un eventuale ABPS proxy client lo stato di consegna del pacchetto al primo access point (*First-hop Transmission Notification*).

Grazie alle notifiche e ai meccanismi introdotti da TED un ABPS proxy client potrà decidere, eventualmente, di ritrasmettere un certo messaggio (attraverso la stessa NIC oppure attraverso un'altra interfaccia) oltre che valutare la QoS del collegamento di rete attualmente utilizzato,
Quando un ABPS proxy client (o un' applicazione che semplicemente sfrutta i meccanismi offerti da TED) trasmetterà un messaggio questo verrà mercato con un identificativo univoco da parte di TED.
Quando il messaggio starà per essere spedito dall'interfaccia di rete Wi-Fi il sottosistema mac80211 assegnerà al frame il sequence control tramite uno dei suoi handler: il sequence control assegnato al frame sarà associato assieme all'identificativo assegnato al messaggio in precedenza da TED. Entrambi saranno mantenuti in una struttura dati creata ad-hoc.
Una volta che il frame sarà stato effettivamente spedito, e il sottosistema mac80211 avrà ricevuto lo status di consegna di un dato frame al first-hop, si andrà alla ricerca, all'interno dell'apposita struttura dati, dell'identificativo associato al sequence control del frame di cui si è appena scoperto lo stato di consegna: l'elemento verrà rimosso dalla struttura dati e verrà sollevata la notifica First-hop Transmission Notification verso l'applicazione interessata a monitorare il messaggio. La notifica indicherà lo status di consegna del pacchetto al first-hop e altre informazioni riguardanti la trasmissione del frame.
Si è scelto di adottare questo approccio in quanto qualsiasi funzione che interagisce con la rete tramite interfaccia socket non è bloccante fino all'invio effettivo del messaggio da parte della scheda di rete; solitamente una funzione che invia dei messaggi lungo la rete resta bloccata fino a quando i dati da inviare sono stati copiati all'interno di socket buffer e questi posti in apposite code di trasmissione dello stack di rete del kernel.
Qui di seguito verrà descritto nel dettaglio TED e la sua implementazione lungo tutto lo stack di rete Linux.

4.1.1 Application Layer

Le applicazioni per le quali ABPS vuole essere un supporto alla mobilità sono le applicazioni multimedia-oriented che, come già trattato nei capitoli precedenti, sono solitamente progettate *over UDP*. Un'applicazione che utilizza UDP per le sue comunicazioni di rete sfrutta uno o più socket datagram, *connectionless*.
Per poter valutare la ritrasmissione di un dato messaggio il proxy client necessita di un meccanismo di identificazione per ciascun datagram.
A tal proposito si è esteso la system call *sendmsg* in modo tale che possa ritornare un **id** univoco per il messaggio in invio cosicché il proxy client possa mantenerlo e usarlo per riferirsi a quel preciso datagram.
Tutte le notifiche ricevute poi in seguito dal proxy client, provenienti da TED, faranno riferimento a un certo frame utilizzando lo stesso identificativo.

La system call sendmsg consente di inviare, assieme al contenuto del messaggio, delle informazioni di controllo aggiuntive dette *ancillary data* che non saranno però trasmesse lungo la rete.
Dal punto di vista implementativo i dati di tipo ancillary sono realizzati in POSIX tramite una sequenza di strutture **struct cmsghdr** contenenti le informazioni di controllo.
L'estensione di sendmsg, progettata per realizzare TED, prevede l'utilizzo di ancillary data durante la fase di invio di un messaggio; in particolare saranno utilizzati:

- Per segnalare a TED che l'applicativo invocante la system call sendmsg richiede di poter ricevere l'id per il messaggio in invio. Per far ciò viene introdotto un nuovo valore non utilizzato per il campo **cmsg_type** della struct cmsghdr.

- Per passare a TED un indirizzo di memoria in user space dove TED potrà assegnare l'identificativo generato per il datagram in invio.

Una volta creato un socket può essere possibile, inoltre, specificare particolari opzioni aggiuntive da adottare ad un certo livello di rete e protocollo tramite la system call *setsockopt*. Con questo meccanismo è possibile abilitare un socket per la ricezione di messaggi di tipo ICMP (specificando come parametro della system call setsockopt l'opzione IP_RECVERR): ad esempio quando avviene un errore di trasmissione su uno dei nodi intermedi (ad esempio un router non riesce a determinare la destinazione di un certo datagram) viene generato un messaggio di tipo ICMP e consegnato all'host mittente che sarà quindi accodato nel buffer del socket.
I messaggi di errore possono essere poi letti dall'applicazione che mantiene il socket tramite la system call **recvmsg** specificando il socket ed il flag MSG_ERRQUEUE.
In TED questo meccanismo è stato sfruttato per la ricezione delle notifiche di tipo First-hop Transmission Notification.
In particolare in Linux eventuali messaggi di errore, che possono provenire dalla rete, sono mantenuti in una struttura dati di tipo struct sock_extended_err; uno dei campi di questa struttura è il campo ee_origin che specifica l'origine del messaggio a cui si fa riferimento. È stato, quindi, aggiunto un nuovo valore (SO_EE_ORIGIN_LOCAL_NOTIFY) tra quelli disponibili per il campo ee_origin indicante la notifica proveniente da TED.
Maggiori dettagli su come la notifica viene effettivamente generata e di come la struttura struct sock_extended_err è stata utilizzata (ed estesa) per realizzare la First-hop Transmission Notification saranno forniti in seguito.

4.1.2 Transport layer

Una volta che il messaggio sarà stato inviato tramite sendmsg ed il Socket Interface Layer avrà passato il controllo alla primitiva di livello trasporto udp_sendmsg (per ulteriori dettagli vedere capitolo 2), attraverso una funzione appositamente sviluppata, si andranno ad analizzare i dati di tipo ancillary contenuti nel messaggio passato come parametro della sendmsg che sarà ricevuto della udp_sendmsg. La struttura struct msghdr mantiene il

messaggio passato alla system call dall'applicativo a livello utente e gli eventuali dati di controllo del messaggio. Una volta analizzati gli ancillary data se l'applicazione ha specificato di essere interessata a ricevere l'identificativo del messaggio inviato verrà settato un apposito flag is_identifier_required. Verrà quindi mantenuto l'indirizzo di memoria user space specificato dall'applicativo dove TED potrà settare l'identificativo calcolato per il pacchetto in invio.
Quando il controllo viene passato momentaneamente a livello rete, e verrà allocato il socket buffer relativo al messaggio in invio, sarà calcolato l'identificativo per quel pacchetto e assegnato alla struttura sk_buff (maggiore dettagli in seguito).
Non appena il flusso di esecuzione viene ripreso dalla primitiva udp_sendmsg, se l'allocazione del socket buffer è andata a buon fine, nell'indirizzo di memoria user space, precedentemente ricavato dall'ancillary data del messaggio, verrà copiato l'identificativo appena assegnato alla struttura sk_buff tramite la macro put_user (consente di copiare un certo valore presente in kernel space a partire da un certo indirizzo user space).
A questo punto, ad un'applicazione utente, sarà disponibile l'identificativo appena assegnato al messaggio in invio e potrà essere utilizzato per monitorare il messaggio stesso.

4.1.3 Network Layer

A livello rete sono state realizzate diverse modifiche.
In particolare ogni qualvolta che viene allocata una struttura dati di tipo sk_buff, per un messaggio in invio, verrà calcolato l'identificativo univoco che sarà poi passato a livello utente.
Nella precedente versione di TED, sviluppata per Kernel Linux 2.6.30-rc5, veniva utilizzato come identificativo da utilizzare a livello applicativo per monitorare i singoli messaggi spediti il campo *Identification* (per maggiori dettagli si veda il primo capitolo) del datagram IPv4: una volta che veniva calcolato e settato questo campo della struttura iphdr, che rappresenta un

header IPv4 nei moduli di rete del kernel, il valore veniva passato a livello utente.
Nella versione di TED sviluppata per kernel Linux 4.0 viene introdotto il supporto a IPv6.
L'header IPv6, supportando solamente la frammentazione di tipo end-to-end, non è caratterizzato dal campo Identification come l'header della precedente versione dell'Internet Protocol.
Per ovviare a questo problema si è definito un contatore globale: ogni volta che viene allocato un nuovo socket buffer il contatore viene incrementato (all'interno di una critical section) e associato a quel socket buffer.
La struttura sk_buff è stata quindi estesa aggiungendo il nuovo campo che manterrà l'identificativo univoco assegnatoli da TED.

```
1 struct sk_buff
2 {
3   ...
4   ...
5   uint32_t sk_buff_identifier;
6 };
```

È stata definita, inoltre, una funzione per l'assegnazione dell'identificativo al socket buffer.

Quando la funzione di livello trasporto udp_sendmsg lascia il controllo alla primitiva __ip_append_data di livello rete per ogni sk_buff allocato verrà inizializzato il suo identificativo tramite la nuova primitiva introdotta.
Non appena la funzione udp_sendmsg avrà ripreso il controllo dell'esecuzione potrà notificare all'applicativo, che ne ha fatto richiesta, l'identificativo appena assegnato al messaggio prima di continuare con il workflow di trasmissione.
A livello rete può essere necessario frammentare un pacchetto prima di inviarlo: dal punto di vista implementativo questo si traduce nell'allocare un socket buffer per ogni frammento di un datagram IP. Questo viene realizzato dal modulo di rete all'interno della routine ip_fragment.

Il meccanismo è stato esteso copiando all'interno del nuovo socket buffer allocato il valore dell'identificativo mantenuto dal datagramma originario: in questo modo ciascun fragment di uno stesso datagram IP avrà lo stesso identificativo del pacchetto iniziale.
Il meccanismo precedentemente descritto è stato equivalentemente realizzato sia per il protocollo IPv4 che IPv6.

4.1.4 The mac80211 subsystem

Quando un pacchetto a livello data-link deve essere spedito tramite interfaccia di rete wireless, come già descritto nel capitolo 2, il controllo viene lasciato al modulo mac80211 a cui è delegato il compito di generare, trasmettere e ricevere frames 802.11.
In fase di preparazione dell'header IEEE 802.11 verrà invocato, tra gli altri, un apposito handler chiamato ieee80211_tx_h_sequence che si occupa di generare e assegnare all'header 802.11 il sequence control.
In fase di invio di un frame 802.11 i dati in trasmissione sono wrappati all'interno di una struttura ieee80211_tx_data che mantiene un riferimento a una struttura socket buffer. Una volta assegnato il sequence control all'header del frame in uscita sarà possibile accedere alla struttura sk_buff (a cui i dati in trasmissione fanno riferimento) e quindi al socket associato.
TED può verificare se su quel socket è stata abilitata l'opzione di ricezione di eventuali messaggi d'errore o meno. Se l'opzione è stata attivata in precedenza a livello applicativo (tramite system call setsockopt) il sequence control calcolato viene associato all'identificativo sk_buff_identifier del socket buffer di cui si sta preparando l'header 802.11; l'identificativo del socket buffer e il sequence control saranno memorizzati assieme all'interno di una struttura dati ad-hoc mantenuta da TED.
In questa struttura vengono mantenute, per ogni frame, anche altre informazioni oltre al sequence control e l'identificativo assegnato da TED.
In particolare viene memorizzato se il messaggio è incapsulato in un datagram IPv4 o IPv6. Ciò è possibile verificando il campo Protocol ID mantenuto al-

l'interno dell'header IEEE 802.2 (per maggiori spiegazioni si veda il capitolo 1). A livello implementativo TED accede a questo campo tramite un offset a partire dall'header IEEE 802.11.
Se si è utilizzato IPv4 per la trasmissione verrà, inoltre, memorizzato se vi sono o meno altri frammenti dello stesso datagram IP, la lunghezza del frammento e il fragment offset.

Come già accennato nel capitolo 2 una volta che un frame IEEE 802.11 è stato inviato attraverso un'interfaccia di rete wireless lo stato di trasmissione sarà notificato in maniera asincrona al modulo mac80211 tramite uno specifico handler chiamato ieee80211_tx_status che fornisce alcune informazioni sulla trasmissione appena avvenuta.
Alcune delle informazioni fornite da questo handler saranno poi quelle utilizzate per il contenuto della notifica di tipo First-hop Transmission Notification che TED solleverà verso il proxy client ABPS.
L'handler ieee80211_tx_status è stato esteso in modo tale da estrarre l'header 802.11, associato al messaggio di cui l'handler sta notificando lo status, e da questo salvare il sequence control: il sequence control sarà utilizzato come chiave di ricerca all'interno della struttura dati mantenuta da TED, precedentemente menzionata, per trovare un eventuale identificativo associato a quel sequence control.
Se la ricerca si conclude con successo l'elemento verrà rimosso dalla struttura dati e TED si preparerà ad inviare una notifica all'applicazione che sta monitorando il datagram marcato dall'identificativo risultante.

4.1.5 First-hop Transmission Notification

Una volta trovato l'elemento, all'interno della struttura dati mantenuta da TED, associato al sequence control dell'header del pacchetto di cui mac80211 è stato appena notificato sullo stato di consegna si procederà con la generazione e la trasmissione della First-hop Transmission Notification verso il proxy client ABPS (o più genericamente verso un'applicazione che

sfrutta TED).
A seconda se la trasmissione dati sfrutta IPv4 o IPv6 la notifica sarà strutturata in modo diverso.
Come già brevemente accennato la notifica di tipo First-hop Transmission Notification viene realizzata estendendo la struttura di tipo struct sock_extended_err tradizionalmente utilizzata per mantenere le informazioni legate ad eventuali errori avvenuti in fase di trasmissione/ricezione di un messaggio verso/da la rete.
Verrà quindi generato un nuovo sk_buff contenente una struttura di tipo struct sock_extended_err (appesa nel campo *cb*).
Il nuovo sk_buff sarà poi appeso nella coda d'errore del socket che aveva originariamente spedito il pacchetto alla quale la notifica fa riferimento.

È stato introdotto un nuovo valore SO_EE_ORIGIN_LOCAL_NOTIFY per il campo ee_origin che identifica la notifica proveniente da TED.

```
1  struct sock_extended_err
2  {
3    __u32 ee_errno;
4    __u8  ee_origin;
5    __u8  ee_type;
6    __u8  ee_code;
7    __u8  ee_pad;
8    __u32 ee_info;
9    __u32 ee_data;
10
11   /* new value added for retry count */
12   __u8    ee_retry_count;
13 };
14
15 #define SO_EE_ORIGIN_NONE 0
16 #define SO_EE_ORIGIN_LOCAL  1
17 #define SO_EE_ORIGIN_ICMP 2
18 #define SO_EE_ORIGIN_ICMP6  3
```

```
19 #define SO_EE_ORIGIN_TXSTATUS 4
20
21 /* new value for ee_origin */
22 #define SO_EE_ORIGIN_LOCAL_NOTIFY 5
```

La struttura sock_extended_err è stata estesa con un nuovo campo chiamato ee_retry_count che verrà settato con il numero di volte che un pacchetto è stato, eventualmente, ritrasmesso.
L'identificativo alla quale la notifica fa riferimento è memorizzato, invece, nel campo ee_info utilizzato, solitamente, per mantenere informazioni aggiuntive e opzionali riguardo l'errore a cui l'intera struttura fa riferimento.
Lo stato di consegna del frame, ovvero se il frame è stato effettivamente consegnato al primo access point (ACK or NACK), viene memorizzato nel campo ee_type.
Nel caso in cui la notifica faccia riferimento a un messaggio trasmesso utilizzando IPv4 vengono memorizzate delle informazioni aggiuntive relative alla frammentazione: se vi sono altri frammenti oppure no (More Fragment) è memorizzato nel campo ee_code mentre la lunghezza del fragment e l'offset sono memorizzati nel campo ee_data della struct sock_extended_err.

4.2 Remark

In questo capitolo si è descritto la progettazione e l'implementazione di un modulo Transmission Error Detector per Wi-Fi su kernel Linux 4.0.
L'obbiettivo preposto era quello di realizzare un modulo TED che potesse monitorare tutti i datagram spediti da un applicazione tramite socket UDP.
Appena TED viene informato dal sottosistema mac80211 sullo stato del frame trasmesso notifica l'applicazione interessata sullo stato di trasmissione del messaggio.
L'implementazione, seppur possa sembrare in un primo momento un tantino ostica, risulta essere piuttosto snella ed efficiente.
Sono state riutilizzate quante più possibili strutturi già implementate all'interno dei moduli di rete kernel aggiungendo, così, solo quanto strettamente

necessario.
Il custom kernel ottenuto non presenta alcun problema di utilizzo e nessuna degradazione delle sue prestazioni o funzionalità.

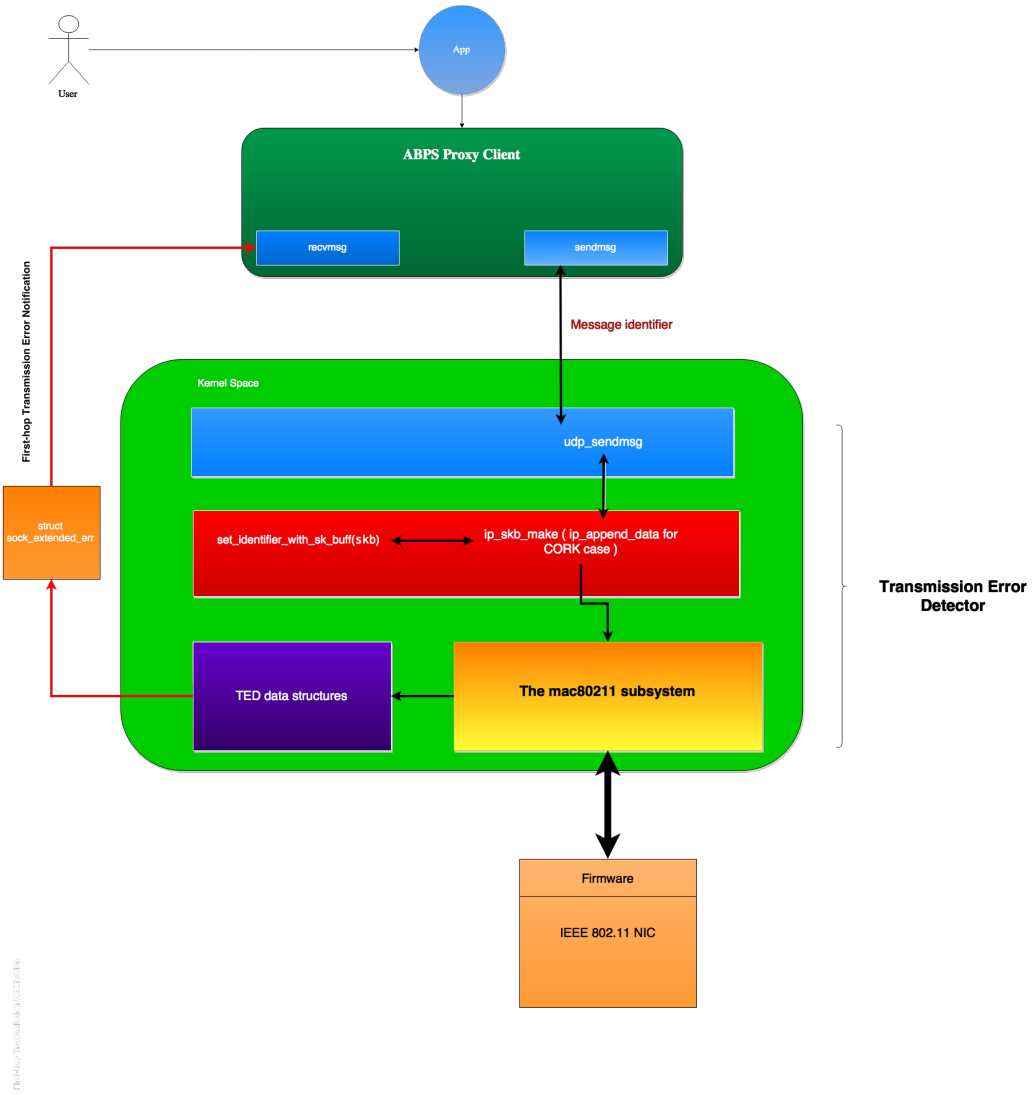

Figura 4.1: Una panoramica di insieme del modulo TED implementato.

Capitolo 5

Transmission Error Detector developer APIs

Transmission Error Detector è pensato per operare assieme all'architettura ABPS all'interno di un ABPS proxy client.
Tuttavia una qualsiasi applicazione, in esecuzione su di un kernel a cui è stato installato il modulo TED, può essere pensata e implementata per sfruttare il meccanismo di Transmission Error Detection raccontato nel capitolo precedente.
Gli unici accorgimenti che un applicativo deve adottare per beneficiare dei meccanismi forniti dalla versione di Transmission Error Detector sviluppata consistono in:

- Abilitare il socket UDP utilizzato per la trasmissione alla ricezione degli errori.

- Adottare la system call sendmsg e specificare nell'ancillary data del messaggio in invio che si è interessati a ricevere un identificativo per quel messaggio oltre che all'indirizzo di memoria dove si vuole venga memorizzato l'identificativo assegnato.

5.1 Send a message

```
1
2 /* create IPv4 socket */
3 int file_descriptor, error, option_value;
4 file_descriptor = socket(AF_INET, SOCK_DGRAM, 0);
5
6 /* Enable socket for error message */
7 error = setsockopt(*file_descriptor, IPPROTO_IP, IP_RECVERR, (
    char *)&option_value, sizeof(option_value));
```

La system call sendmsg accetta come parametro una struttura di tipo struct msghdr che descrive il contenuto del messaggio in invio.

```
1
2 struct msghdr
3 {
4     void         *msg_name;      /* optional address */
5     socklen_t     msg_namelen;       /* size of address */
6     struct iovec *msg_iov;            /* scatter/gather array */
7     size_t        msg_iovlen;         /* # elements in msg_iov
     */
8     void         *msg_control;        /* ancillary data, see
     below */
9     size_t        msg_controllen;     /* ancillary data buffer
     len */
10    int           msg_flags;          /* flags on received
     message */
11 };
```

Come già accennato la struttura struct msghdr può contenere delle informazioni di controllo che non saranno trasmesse lungo la rete; questo tipo di informazioni sono chiamate ancillary data.
Gli ancillary data sono una sequenza di strutture di tipo struct cmgshdr.
Gli ancillary data vanno sempre acceduti tramite apposite macro e mai direttamente.
Per accedere alla prima struttura struct cmsghdr di un certo messaggio di tipo

struct msghdr, ad esempio, basterà invocare la macro **CMSG_FIRSTHDR()** specificando come parametro il puntatore alla struttura msghdr.

```

struct cmsghdr
{
  socklen_t cmsg_len;    /* data byte count, including header */
  int       cmsg_level;  /* originating protocol */
  int       cmsg_type;   /* protocol-specific type */
  /* followed by unsigned char cmsg_data[]; */
};
```

Per specificare che l'app invocante sendmsg è interessata a ricevere l'identificativo da TED è sufficiente settare il campo della struttura struct cmsghdr **cmgs_type** al nuovo valore **ABPS_CMSG_TYPE**.
ABPS_CMGS_TYPE è stato definito in socket.h.
All'interno del campo data della struttura struct cmsghdr è necessario copiare il puntatore all'area di memoria in cui si desidera TED setti l'identificativo assegnato al messaggio in invio.

Vediamo un esempio di utilizzo.

```

uint32_t identifier;

uint32_t *pointer_for_identifier = &identifier;

char ancillary_buffer[CMSG_SPACE(sizeof (pointer_for_identifier)
    )];

struct iovec iov[3]; /* Scatter-Gather I/O */

struct msghdr message_header;

struct cmsghdr *cmsg;

char buffer[100];

```

```
16 memset(buffer,0,100);
17
18 strncpy(buffer,"Hello from an app built on top of TED!",100);
19
20
21 iov[0].iov_base = (void *) buffer;
22 iov[0].iov_len = strlen(buffer);
23
24
25 message_header.msg_name = (void *) &destination_address; /*
       struct sockaddr_in for destination host */
26 message_header.msg_namelen = sizeof (destination_address);
27
28 message_header.msg_iov = iov; /* message content*/
29 message_header.msg_iovlen = 1;
30
31 message_header.msg_control = ancillary_buffer;
32 message_header.msg_controllen = sizeof(ancillary_buffer);
33
34 cmsg = CMSG_FIRSTHDR(&message_header);  /* get first struct cmsg
       from message_header*/
35
36 cmsg->cmsg_level = SOL_UDP;
37 cmsg->cmsg_type = ABPS_CMSG_TYPE;  /* new type for struct
       cmsghdr defined in socket.h */
38
39 cmsg->cmsg_len = CMSG_LEN(sizeof(pointer_for_identifier));
40
41 pointer = (char *) CMSG_DATA(cmsg); /* accessing cmsg_data field
       */
42
43 memcpy(pointer, &pointer_for_identifier, sizeof(
       pointer_for_identifier)); /* copying pointer to variable in
       cmsg_data field*/
44
45 message_header.msg_controllen = cmsg->cmsg_len;
46
47
```

```
48 /* Prepare for sending. */
49 result_value = sendmsg(file_descriptor, &message_header,
       MSG_NOSIGNAL);
```

5.2 Receive a notification from TED

Per ricevere una notifica da TED è sufficiente predisporre l'app in lettura sulla coda di errore del socket sfruttato in precedenza per l'invio di un messaggio.
È possibile leggere dalla coda degli errori di un socket attraverso la system call **recvmsg** specificando il flag MSG_ERRQUEUE.
La notifica può essere ricevuta da un'app allo stesso modo in cui, l'app, potrebbe ricevere un messaggio di tipo ICMP.
Come già accennato precedentemente le informazioni della First-hop Transmission Notification sono contenute all'interno di una struttura sock_extended_err.

```
1
2 /* Fetch the notification from socket error queue */
3 return_value = recvmsg(file_descriptor, message, MSG_ERRQUEUE |
       MSG_DONTWAIT);
4
5
6
7 for(cmsg = CMSG_FIRSTHDR(message); cmsg; cmsg = CMSG_NXTHDR(
       message, cmsg))
8  {
9   if((cmsg->cmsg_level == IPPROTO_IPV6) && (cmsg->cmsg_type ==
       IPV6_RECVERR))
10   {
11     first_hop_transmission_notification = (struct
       sock_extended_err *) CMSG_DATA(cmsg);
12     switch (first_hop_transmission_notification->ee_origin)
13     {
14               /* new origin type introduced  */
15       case SO_EE_ORIGIN_LOCAL_NOTIFY:
16                     {
```

```
17                              if(
    first_hop_transmission_notification->ee_errno == 0)
18                              {
19                                uint32_t identifier =
    ted_message_identifier_from_notification(
    first_hop_transmission_notification);
20              printf("Just got a new notification for message marked
     with identifier \%" PRIu32 ". \n", identifier);
21                                .......
22              .......
23                              }
24                          }
25                          ........
26                          .......
27      }
28    }
29  }
```

5.3 Interact with First-hop Transmission Notification

Si è deciso di definire una convient API per accedere ai valori memorizzati all'interno della notifica e quindi della struttura sock_extended_err.
Le nuove macro introdotte fungono da wrapper per facilitare l'accesso ad alcuni campi della struttura stessa.
L'utilizzo di queste funzioni è consigliato per una maggiore chiarezza e astrazione. Tutte le macro sono definite assieme alla struttura sock_extended_err nel file errqueue.h.
Tutte le funzioni definite in questa nuova interfaccia accettano come unico parametro una struttura di tipo sock_extended_err.

ted_message_identifier_from_notification Utilizzata per accedere all'identificativo del messaggio di cui la notifica è associata.

Parameter: struct sock_exetended_err *notification.

Return value: un uint32_t contenente il valore dell'identificativo del messaggio a cui la notifica si riferisce.

ted_message_status_from_notification Ritorna lo status di consegna del messaggio a cui la notifica vuole fare riferimento.
Se il valore ritornato è 1 il messaggio è stato consegnato con successo al primo Access Point (ACK), se 0 altrimenti (NACK).

Parameter: struct sock_exetended_err *notification.

Return value: un uint8_t contenente lo status di consegna.

ted_message_retry_count_from_notification Ritorna il numero di volte che un messaggio è stato ritrasmesso lungo l'interfaccia di rete prima di essere consegnato.
Un frame marcato come non consegnato (NACK) avrà retry count pari a zero.

Parameter: struct sock_exetended_err *notification.

Return value: un uint8_t contenente il numero di volte che il messaggio è stato trasmesso lungo la rete.

ted_message_fragmentation_length_info_from_notification Restituisce la lunghezza del frammento a cui si riferisce la notifica.
In fase di lettura dalla coda degli errori del socket più di una notifica con lo stesso identificativo possono pervenire.
Per disambiguare a quale porzione dati fa riferimento la notifica vengono fornite alcune informazioni aggiuntive sulla frammentazione del messaggio originario.

Parameter: struct sock_exetended_err *notification.

Return value: un uint16_t contenente la lunghezza espressa in bytes del frammento.

ted_message_fragmentation_offset_info_from_notification Restituisce l'offset, rispetto al frammento originario, del fragment a cui fa riferimento la notifica.

Parameter: struct sock_exetended_err *notification.

Return value: un uint16_t indicante l'offset del frammento rispetto al frammento originario a cui si riferisce la notifica espressa in bytes del frammento.

ted_message_more_fragment_info_from_notification Ritorna 1 se vi sono altri frammenti relativi allo stesso messaggio, 0 altrimenti.

Parameter: struct sock_exetended_err *notification.

Return value: un uint8_t indicante la presenza, o meno, di ulteriori frammenti per lo stesso messaggio.

I wrapper per la l'accesso ai campi della stuct sock_extended_err appena descritti.

```
1 /* TED convenient wrapper APIs for First-hop Transmission
     Notification. */
2
3 /* Transmission Error Notification identifier of the datagram
     whose notification refers to. */
```

```
4 #define ted_message_identifier_from_notification(notification)
      ((struct sock_extended_err *) notification)->
    notification_info
5
6 /* Message status to the first hop. Return 1 if the message was
    successfully delivered to the AP, 0 otherwise. */
7 #define ted_message_status_from_notification(notification)  ((
    struct sock_extended_err *) notification)->notification_type
8
9 /* Returns the number of times that the packet, associated to
    the notification provided, was retrasmitted to the AP.  */
10 #define ted_message_retry_count_from_notification(notification)
    ((struct sock_extended_err *) notification)->
    notification_retry_count
11
12 /* Returns the fragment length */
13 #define ted_message_fragmentation_length_info_from_notification(
    notification)   (((struct sock_extended_err *) notification)
    ->notification_data >> 16)
14
15 /* Returns the offset of the current message associated with the
     notification from the original message. */
16 #define ted_message_fragmentation_offset_info_from_notification(
    notification)   ((((struct sock_extended_err *) notification)
    ->notification_data << 16) >> 16)
17
18 /* Indicates if there is more fragment with the same TED
    identifier */
19 #define ted_message_more_fragment_info_from_notification ((
    struct sock_extended_err *) notification)->notification_code
```

Capitolo 6

Test & valutazioni sperimentali

Dopo aver illustrato il funzionamento e l'implementazione del TED, passiamo ora ad analizzare i dati relativi alla trasmissione di pacchetti tramite la nostra applicazione. Lo scopo dei test effettuati è quello di analizzare la QoS del segnale e come può variare in diversi scenari. Questo ci è utile in quanto, in base ai risultati ottenuti, si può decidere se cambiare NIC per l'invio di determinati pacchetti.

Per poter analizzare

6.1 Raccolta dati

Andremo ora a mostrare quali test sono stati effettuati. In particolare mostreremo quali dispositivi sono stati utilizzati per fare le prove, i parametri di valutazione ed i risultati ottenuti.

6.1.1 Dispositivi

Per effettuare delle prove sperimentali, sono stati utilizzati diversi dispositivi. In particolare due per simulare il client (o nodo mobile) ed altri per creare traffico nella rete, in modo da impegnare il canale per avere condizioni simili ad un tipico scenario di utilizzo. Per ogni dispositivo è interessante mo-

strare per cosa è stato utilizzato. Nel caso di un nodo della rete, mostreremo anche il sistema operativo, il kernel, il processore e scheda di rete.

ZyXEL NBG4615 v2 Access point utilizzato durante i test. È stata impostata la modalità Wi-Fi 802.11b/g/n.

LB-LINK BL-WN151 Adattatore Wireless USB. Velocità fino a 150Mb/s, supporta 802.11b/g/n. È stata utilizzata principalmente sui Raspberry, visto che non dispongono di wireless integrato.

NETGEAR WG111 Adattatore Wireless USB. Velocità fino a 54Mb/s, supporta 802.11b/g. Questo adattatore è più lento ed è stato usato sul computer HP quando il wireless integrato dava dei problemi.

HP Pavilion dv6 Entertainment PC Questo notebook è stato utilizzato come client. Abbiamo montato un kernel versione 4.0.1, modificato tramite la procedura illustrata precedentemente. Le specifiche tecniche sono:

- Kernel: Linux versione 4.0.1 modificata.
- Processore: Intel Core i5 CPU M 430.
- Sistema operativo: Ubuntu 14.04 LTS 64-bit.
- Scheda di rete: Broadcom BCM43225 802.11b/g/n.

DELL Latitude E6400 Anche questo notebook è stato utilizzato come client, e vi è quindi montato un kernel versione 4.0.1 modificato. Le specifiche tecniche sono:

- Kernel: Linux versione 4.0.1 modificata.
- Processore: Intel Core 2 Duo.
- Sistema operativo: Ubuntu 14.04 LTS 32-bit.
- Scheda di rete: Intel Corporation WiFi Link 5100 802.11a/g/n.

Raspberry Pi Model 2 Abbiamo utilizzato questo raspberry per creare traffico sulla rete. Le specifiche tecniche sono:

- Kernel: Linux versione 3.18.0-20-rpi2.
- Processore: 900MHz quad-core ARM Cortex-A7.
- Sistema operativo: Raspbian.

UDOO Quad Abbiamo utilizzato la UDOO per creare traffico sulla rete, insieme ad i raspberry. Le specifiche tecniche sono:

- Kernel: Kernel Linux 3.0.35.
- Processore: Freescale i.MX 6 ARM Cortex-A9 CPU Dual/Quad core 1GHz.
- Sistema operativo: UDOObuntu.

Raspberry Pi Model B Abbiamo utilizzato questo raspberry per creare traffico sulla rete, ma a volte è stato anche utilizzato come server. Le specifiche tecniche sono:

- Kernel: Kernel Linux 3.18.7+.
- Processore: 700 MHz single-core ARM1176JZF-S.
- Sistema operativo: Raspbian Wheezy.

6.1.2 Parametri di valutazione

Per poter analizzare i test in modo ottimale e per avere dei dati su cui lavorare abbiamo deciso di controllare alcuni parametri.
I parametri che ci interessano maggiormente sono:

- **Id**: l'Id del pacchetto inviato.

- **ACK**: se è stato ricevuto un ACK o un NACK da parte dell'AP.
- **Data**: è dato dalla data e dall'orario di invio del pacchetto.
- **Tempo**: è il tempo in millisecondi tra l'invio del pacchetto e la ricezione della notifica da parte dell'access point.
- **Retry count**: è il numero di tentativi di invio di un determinato pacchetto.
- **Versione IP**: IPv4 o IPv6.
- **Configurazione**: è la configurazione dei dispositivi utilizzati durante l'esperimento.
- **Wait**: indica se la recv è bloccante.

Abbiamo scelto questi parametri perché ci permettono di poter giudicare in maniera chiara l'andamento dei pacchetti e la situazione della rete. In particolare sono molto significativi il tempo, l'ACK ed il retry count. Grazie a questi dati si può analizzare in modo dettagliato la situazione di ogni singolo pacchetto. L'applicazione può leggere l'ACK e successivamente decidere di rimandare il pacchetto in base ai millisecondi passati prima di ricevere la notifica. Il numero di retry count risulta rilevante per confrontare differenti situazioni di traffico, oppure per notare cosa succede in caso di trasmissione in movimento.

Gli altri parametri che abbiamo deciso di utilizzare hanno un valore più trascurabile per un singolo pacchetto, ma possono diventare eloquenti per analizzare i dati a posteriori. In particolare si potrebbe notare in base all'orario se c'è un evidente rallentamento della trasmissione. Ad esempio si potrebbe notare come in una zona industriale la QoS migliori durante la sera/notte.

Un altro dato che può essere utilizzato per esaminare i dati raccolti è la versione IP, si può controllare se c'è un differenza notevole tra IPv6 e IPv4 a parità di condizioni.

Le configurazioni, invece, riguardano i dispositivi utilizzati durante un test e lo scenario applicativo. Andremo a mostrare quali configurazioni sono state provate in modo più dettagliato successivamente. Per quanto riguarda la wait abbiamo deciso di fare sia una recv bloccante che una non bloccante. Abbiamo fatto dei test con entrambe e abbiamo analizzato le differenze, che andremo a descrivere più avanti.

Si potrebbero utilizzare anche altre informazioni (ad esempio la bitrate) per analizzare meglio i risultati, che saranno approfondite negli sviluppi futuri.

6.1.3 Configurazioni

Per ottenere dei risultati che potessero rispecchiare un reale utilizzo da parte di un nodo mobile abbiamo creato diverse configurazioni di dispositivi. In particolare abbiamo deciso di tenere conto di alcuni possibili scenari di utilizzo, che sono:

- Dispositivo client in movimento oppure fermo.
- Utilizzo indoor o outdoor.
- Trasmissione in linea diretta oppure trasmissione con un ostacolo tra nodo mobile ed AP.
- Assenza di traffico sulla rete in contrapposizione ad uno o più hosts wireless a creare traffico.
- Nel caso di presenza di nodi sulla rete, abbiamo utilizzato anche la distanza e la velocità come parametri.
- Trasmissioni ad un host della stessa sottorete oppure ad una rete esterna.

Dati questi possibili utilizzi, abbiamo creato alcune configurazioni per valutare la qualità del collegamento. Il nostro interesse si è focalizzato sulla

costruzione di un prodotto cartesiano tra tutte le opzioni. Abbiamo quindi provato col nodo mobile fermo, da solo nella rete ed in linea diretta con l'AP. In questa configurazione abbiamo anche valutato le possibili differenze in velocità tra recv bloccante e non bloccante. La scelta tra queste due opzioni è stata implementata nell'applicazione di prova che abbiamo creato e che abbiamo descritto precedentemente.
In contrapposizione a questa prima configurazione, abbiamo testato un nodo mobile solitario nella rete, fermo ma con un ostacolo tra lui e l'AP. Come ostacolo è stato scelto un muro, di larghezza di circa 15 centimetri.
Per completare la raccolta dati da analizzare abbiamo fatto altre due configurazioni, andando a modificare quelle precedenti usando il nodo mobile in movimento.
Queste prime configurazioni ci permettono già di raccogliere importanti dati, che però non possono riflettere un reale utilizzo di una applicazione. Questo perché difficilmente la trasmissione VoIP avverrà con il dispositivo da solo sulla rete, ma la rete potrà essere più o meno congestionata in base al luogo o all'orario.
Per rispondere a questa esigenza abbiamo deciso di fare delle prove con uno o più host attivi sulla rete, in modo che andassero a causare traffico per rallentare il nodo mobile.
Abbiamo quindi creato altre configurazioni, andando in ognuna a modificare il numero di hosts ed altri parametri. Per quanto riguarda gli host li abbiamo lasciati sempre fermi, ma abbiamo modificato la distanza in diverse prove. In questo modo gli host trasmetteranno più o meno velocemente e vogliamo andare a verificare al client dia più fastidio avere host lenti o veloci. Per controllare le velocità è stata creata una applicazione in grado di stabilire la bitrate. Questa applicazione sarà descritta successivamente.

Analizziamo ora i settaggi che sono stati implementati. In particolare avremo il nodo mobile fermo ed un host attivo, entrambi in linea diretta con l'access point. Da questa configurazione base abbiamo allontanato l'host, fino a variare la velocità in modo significativo (anche di un fattore 1/20).

Queste ultime configurazione sono state ampliate anche con l'uso della recv prima bloccante, e poi non bloccante. Invece per quanto riguarda tutte le configurazioni a seguire abbiamo deciso di testare l'applicazione solo con la modalità della recv non bloccante. Questo è stato fatto perché i primi dati empirici erano stati raccolti in quel modo e non si voleva quindi andare ad alterare il risultato. Avendo comunque analizzato separatamente il comportamento bloccante e non, possiamo dare una congettura di quello che può succedere in caso di comportamento non bloccante.
Anche in questo caso, per ogni configurazione creata, ne abbiamo create altre in cui abbiamo scelto parametri diversi. Quindi per ognuna è stata tenuta una determinata struttura e siamo andati a modificare un parametro alla volta, registrato tutti i dati ottenuti.
Grazie a questo grande numero di possibili utilizzi, abbiamo raccolto sufficienti dati empirici.

6.1.4 Dettagli implementativi

Avendo descritto quali parametri e quali configurazioni abbiamo descritto, ci concentriamo ora sulla parte implementativa dei test. I principali problemi da risolvere in questa parte erano:

- Velocità degli hosts.
- Salvataggio dei dati.
- Elaborazione dei dati.

Per quanto riguarda la misurazione della velocità degli hosts, abbiamo creato una semplice applicazione che vada a misurarla. Per realizzarla abbiamo creato una connessione tra due hosts, un client ed un server. Abbiamo scelto di realizzare la applicazione tramite socket TCP; è stata comunque implementata anche la versione UDP.
Il client trasmette un file tramite Wi-Fi ed il server si mette in ascolto e

calcola la velocità del client. Il server è stato collegato all'AP tramite ethernet, e non wireless, per evitare di introdurre eventuali ritardi ed incognite. Queste misurazioni sono state fatte all'interno della stessa sottorete, per non influenzare i risultati con la velocità della linea ADSL. Nel server il calcolo è stato realizzato tramite la divisione tra il totale dei bit ricevuti e i secondi necessari per la ricezione. Il totale dei bit è stato semplicemente ricavato dal totale di tutti i Byte ricevuti, moltiplicati per 8. I secondi, invece, sono stati ottenuti catturando il tempo prima dell'inizio della ricezione e appena finita la ricezione (tramite la *time(NULL)*). Successivamente è stata fatta la differenza tra i due time ed è stata calcolata la velocità in Mb/s. Successivamente, alla fine della ricezione di tutti i Byte, è stato catturato il tempo. La velocità che è stata rilevata non può essere considerata perfetta in quanto ci potrebbero alcuni fattori che possono andare ad alterarla. Possiamo però darla per buona perché, per ogni host, è stata misurata più volte ed è stata successivamente scelta la media. Inoltre è stata misurata, nella maggior parte dei casi, in una zona poco abitata e di sera o notte, quindi l'interferenza data da altri dispositivi era minima.

Per quanta riguarda il salvataggio dei dati, abbiamo scelto di farlo in un file in formato *JSON*. Abbiamo scelto il JSON per la facilità di leggere i dati salvati. I dati sono stati salvati come un array di pacchetti, e per ogni pacchetto è stato salvato:

- id del pacchetto.
- millisecondi tra invio pacchetto e ricezione della notifica.
- numero di tentativi di invio.
- versione IP.
- boolean per ack.
- numero della configurazione.

- data.

I dati salvati sono i parametri di valutazione che abbiamo precedentemente scelto. Per utilizzare il JSON all'interno di C abbiamo incluso una apposita libreria, la *json/json.h*.
Un esempio di file JSON è il seguente:

```
1  { "pacchetti": [
2
3    { "testId": 4713, "type": 6, "ack": true, "time":
     0, "retrycount": 0, "ipVersion": "ipv6", "date":
     "Sat Jun 20 22:53:38 2015\n" },
4    { "testId": 4714, "type": 6, "ack": true, "time":
     6, "retrycount": 0, "ipVersion": "ipv6", "date":
     "Sat Jun 20 22:53:38 2015\n" },
5    { "testId": 4715, "type": 6, "ack": true, "time":
     4, "retrycount": 0, "ipVersion": "ipv6", "date":
     "Sat Jun 20 22:53:38 2015\n" },
6    { "testId": 4716, "type": 6, "ack": true, "time":
     11, "retrycount": 0, "ipVersion": "ipv6", "date":
      "Sat Jun 20 22:53:38 2015\n" },
7    { "testId": 4717, "type": 6, "ack": true, "time":
     0, "retrycount": 0, "ipVersion": "ipv6", "date":
     "Sat Jun 20 22:53:38 2015\n" },
8    { "testId": 4718, "type": 6, "ack": true, "time":
     6, "retrycount": 0, "ipVersion": "ipv6", "date":
     "Sat Jun 20 22:53:38 2015\n" }
9    ]
10 }
```

Grazie all'utilizzo di un file JSON per il salvataggio dei dati, ne risulta più semplice l'analisi. Per poter elaborare i dati abbiamo scelto di creare vari script utilizzando il linguaggio *Python*. È stato scelto il Python perché sup-

porta il JSON e perché permette di effettuare tutte le operazioni necessarie in poche righe.

Sono stati realizzati diversi script per svolgere diversi compiti. Ogni script ha uno scopo specifico (ad esempio calcolare la media dei tempi di notifica, oppure il numero di nack ricevuti). Ad ogni script, a meno di casi particolari, viene passato il file JSON ed il numero di configurazione. Ognuno di questi script andrà ad aprire il file di log, nel quale sono salvati tutti i dati. Una volta aperto, viene convertito in una serie di elementi di un array JSON. Ora, grazie alla configurazione passato da comando, sarà possibile andare ad analizzare i dati. Ad esempio, il file Python per calcolare la media dei tempi di notifica di una certa configurazione è:

```
1  #!/usr/bin/python
2  # -*- coding: UTF-8 -*-
3  # all_avg_time.py
4  import json
5  import sys
6
7  tipo=-1
8  nPkt=0
9  numRetry=0
10 try:
11   inputFile=sys.argv[1]
12   tipo=int(sys.argv[2])
13 except:
14   sys.exit("missing argument")
15
16 with open(inputFile) as data_file:
17     data = json.load(data_file)
18
19 for el in data["pacchetti"]:
20   if el['type']==tipo:
21     if el['ack']== True :
22       numRetry=numRetry+el['retrycount']
23       nPkt=nPkt+1
24
```

```
25 avg=float(numRetry)/float(nPkt)
26 print(avg)
```

Sfruttando gli script che abbiamo realizzato siamo in grado di esprimere delle valutazioni su ogni aspetto che volevamo prendere in considerazione.

6.1.5 Test effettuati

Per effettuare i test ci siamo basati sui parametri e sulle configurazioni scelti precedentemente. La disposizione tipica della rete è quella in figura 6.1.

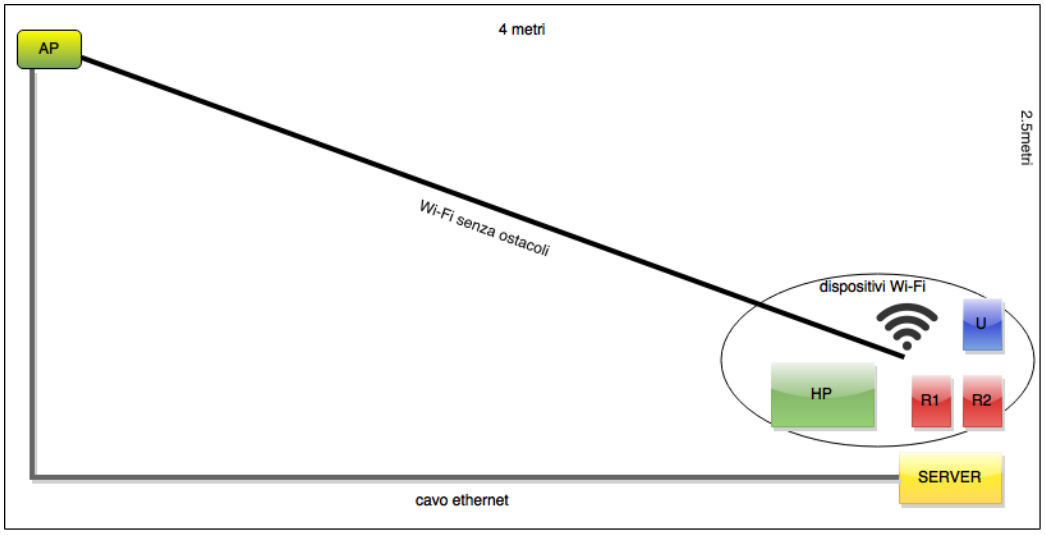

Figura 6.1: Disposizione dei dispositivi maggiormente utilizzata. R1 ed R2 sono i Raspberry, mentre U è la UDOO.

In particolare, è stato usato il computer HP per svolgere la maggior parte dei test. Per creare del traffico in rete, invece, sono stati generalmente utilizzati il Raspberry Pi 2, la UDOO ed il Raspberry Model B, con l'eventuale aggiunta di altri dispositivi (come altri portatili o tablet). Come server sono stati usati o il Raspberry Model B oppure altri computer.
Le prove sono state svolte principalmente a casa, ma anche presso il laboratorio Ercolani. Non sono state fatte prove attraverso la rete *ALMAWIFI* in quanto gli access point bloccavano la comunicazione. Per ovviare al problema è stata creata un rete locale con l'uso del ZyXEL descritto precedentemente.

6.2 Analisi dei risultati

Andiamo ora ad analizzare i risultati ottenuti. Le analisi che abbiamo effettuato riguardano la qualità della trasmissione, i tempi medi di ricezione di una notifica proveniente dall'access point, il numero medio di NACK ricevuti e la media dei retry per l'invio dei pacchetti. Andremo anche ad aggiungere una ulteriore analisi per la configurazione in movimento, che saranno i *burst NACK* (ovvero quanti NACK consecutivi sono notificati).

Per ogni parametro che vogliamo valutare andremo a stabilire quanto può incidere nella trasmissione, lasciando inalterati tutti gli altri parametri.

6.2.1 Confronto tra IPv4 e IPv6

Per quanto riguarda IPv4 ed IPv6, andiamo ad analizzare i risultati relativi ai tempi di ricezione della notifica proveniente dall'access point, la media dei *NACK* e la media dei retry. I test effettuati sono stati realizzati con pacchetti di dimensione ridotta, in modo da non avere frammentazione. Ci si aspetta che non ci sia differenza tra le due prove.
Come primo parametro andiamo a controllare i tempi, in millisecondi, di ricezione della notifica. In particolare si possono notare le medie dei tempi di IPv4 nella tabella 6.1, mentre quelle di IPv6 nella tabella 6.2.

Tabella 6.1: Media dei tempi relativi all'utilizzo di IPv4

IPv4	Conf. 5	Conf. 6	Conf. 7	Conf. 8	Conf. 9
prima	6.234	5.624	5.5952	2.8068	4.9292
seconda	7.8972	5.2736	4.7432	4.5024	2.9844
terza	7.3208	4.0812	4.0432	6.0316	4.7292
quarta	7.627	5.906	3.4704	5.7504	6.7728
Media	7.26975	5.2212	4.463	4.7728	4.8539

Guardando le medie dei tempi si può notare come, effettivamente, non ci sia una sostanziale differenza tra le due versioni. Questo è ragionevole, in quanto nessuna delle due versioni

Tabella 6.2: Media dei tempi relativi all'utilizzo di IPv6

IPv6	Conf. 5	Conf. 6	Conf. 7	Conf. 8	Conf. 9
prima	5.44	6.508	4.9788	5.338	6.2316
seconda	6.8998	6.5516	5.5684	4.5024	4.7456
terza	6.177	6.6788	5.9008	5.0316	6.1744
quarta	5.9642	5.6636	4.9396	5.5612	4.0648
Media	6.12025	6.3505	5.3469	5.1083	5.3041

in quanto sono stati testati pacchetti di dimensione ridotta, in modo da non avere frammentazione. Si possono ora controllare anche le statistiche relative al numero di retry dei pacchetti che ricevono ACK, sia per IPv4 che per IPv6. Ci si aspetta anche in questo caso che i valori siano simili.

Tabella 6.3: Media dei retry relativi all'utilizzo di IPv4

IPv4	Conf. 5	Conf. 6	Conf. 7	Conf. 8	Conf. 9
prima	0.03962	0.156	0.26683	0.13731	0.127105
seconda	0.08884	0.16	0.1312	0.077787	0.14097
terza	0.08545	0.246894	0.193264	0.1642628	0.134454
quarta	0.093838	0.1536	0.242983	0.108844	0.15459
Media	0.076937	0.1791235	0.20856925	0.12205095	0.13927975

Analizzando le medie dei retry dalle tabelle 6.3 e 6.4 possiamo osservare che effettivamente non ci sono sostanziali differenze. L'unica differenza di configurazione è ottenuta nella Conf. 6, ma è dovuta principalmente ad una prova che ha alzato il valore. Visto che una singola prova può essere rallentata da diversi fattori, possiamo valutare le medie relative alle diverse versioni come identiche in caso di mancata frammentazione.

Tabella 6.4: Media dei retry relativi all'utilizzo di IPv6

IPv6	Conf. 5	Conf. 6	Conf. 7	Conf. 8	Conf. 9
prima	0.09	0.14646	0.12902	0.11765	0.1432
seconda	0.06521	0.0748	0.11774	0.1144	0.17508
terza	0.04641	0.0556	0.096	0.11307	0.1484
quarta	0.06687	0.05682	0.08109	0.1508	0.120144
Media	0.0671225	0.08342	0.1059625	0.12398	0.146706

6.2.2 Valutazione problemi dovuti a pacchetti nella rete

Una seconda osservazione si può fare su una attenta analisi riguardo al numero di pacchetti e di hosts sulla rete. In particolare vogliamo analizzare i tempi, i NACK ed i retry relativi alla presenza di un host che trasmette N pacchetti rispetto a K hosts che trasmettono N/K pacchetti ciascuno. La prove sono state effettuate con il nodo da solo nella rete, un host sulla rete che trasmette 71 MB, due hosts che trasmettono 36 MB ciascuno e tre hosts che trasmettono 22 MB ciascuno.

6.2.3 Valutazione interferenza traffico

asdfadsfs

6.2.4 Problemi dovuti alla trasmissione in movimento

Riguardo la trasmissione in movimento, rispetto a quella da fermo, è interessante notare alcuni aspetti. In particolare

Tabella 6.5: Media dei tempi con nodo mobile fermo

	Conf. 83A	Conf. 85A	Conf. 87A
prima	0.768	12.986	26.799
seconda	0.748	9.362	56.85
terza	0.696	7.941	80.897
quarta	0.662	9.295	73.785
Media	0.7185	9.896	62.08275

Tabella 6.6: Media dei tempi con nodo mobile in movimento

	Conf. 83B	Conf. 85B	Conf. 87B
prima	0.987	17.524	45.909
seconda	1.13	15.021	36.636
terza	0.9628	50.244	57.399
quarta	1.064	45.503	96.327
Media	1.03595	21.986	59.06775

Tabella 6.7: Media dei retry con nodo mobile fermo

	Conf. 83A	Conf. 85A	Conf. 87A
prima	0.0652	0.029	0.128
seconda	0.0104	0.0706	0.0961
terza	0.0118	0.0302	0.108
quarta	0.0124	0.0981	0.102
Media	0.7185	9.896	62.08275

Tabella 6.8: Media dei retry con nodo mobile in movimento

	Conf. 83B	Conf. 85B	Conf. 87B
prima	0.227	0.273	0.072
seconda	0.218	0.202	0.179
terza	0.221	0.226	0.225
quarta	0.238	0.212	0.349
Media	0.226	0.22825	0.20625

Conclusioni

Queste sono le conclusioni.
In queste conclusioni voglio fare un riferimento alla bibliografia: questo è il mio riferimento

Appendice A

Prima Appendice

In questa Appendice non si è utilizzato il comando: `\clearpage{\pagestyle{empty}\cleardoublepage}`, ed infatti l'ultima pagina 8 ha l'intestazione con il numero di pagina in alto. METTERE COS'È E COME È FATTO UN SOCKET??

Appendice B

Seconda Appendice

B.1 Customizzazione kernel Linux

Può esserci l'esigenza di personalizzare il proprio kernel per adattarlo alle proprie esigenze. Alcuni vantaggi che si possono ottenere dalla customizzazione del kernel sono:

- Velocizzare l'avvio del sistema.
- Sfruttare un'ottimizzazione basata sul proprio processore per avere un sistema operativo un po' più reattivo.
- Ottenere una configurazione leggermente diversa e più adatta alla propria macchina.
- Cercare di migliorare le prestazioni del proprio dispositivo.
- Ridurre i consumi.
- Installare solo i moduli necessari.
- Rendere il sistema più leggero.

Per personalizzare il kernel, però, serve una buona conoscenza della propria macchina, ed in particolare del proprio hardware. Se non ci conosce il proprio

hardware, infatti, si rischia di peggiorare le prestazione e si rischia di togliere dei moduli che possono influire sul funzionamento del sistema.

Non è necessario compilare un kernel nei casi in cui l'hardware non funzioni alla perfezione o le periferiche non vengano completamente riconosciute. A volte per far riconoscere correttamente al sistema una data periferica basta caricare i moduli necessari con le dovute opzioni. È utile ricompilare il kernel solo se tali moduli non sono presenti o se si è certi che i driver della periferica sono presenti solo in una versione diversa da quella attualmente in uso. Inoltre, l'aumento di prestazioni tende a essere irrilevante, soprattutto su computer già veloci. È anche bene tenere presente che compilare un nuovo kernel significa, nella sostanza, cambiare sistema operativo, in quanto esso ne costituisce il motore; inoltre è richiesta un buona conoscenza del proprio hardware.
Dopo aver analizzato i vantaggi e gli svantaggi della personalizzazione del kernel, spieghiamo ora come si può fare la customizzazione. Per modificare il kernel bisogna determinare quali driver sono necessari, compilare i kernel ed i moduli ed infine installare l'immagine del kernel.
I passi da eseguire per portare a termine una customizzazione sono:

Scaricare una versione stabile del kernel ed estrarlo

```
# cd /directory/
# wget https://www.kernel.org/pub/linux/kernel/
v4.x/linux-4.0.1.tar.xz
# tar -xvJf linux-4.0.1.tar.xz
```

Il link indicato è quello relativo alla pagina ufficiale, dove si possono trovare le ultime versioni dei kernel. La versione scelta da noi è stata la 4.0.1 in quanto era l'ultima stabile.

Scegliere la configurazione

```
# cd linux-4.0.1
# sudo apt-get install libncurses5
```

```
# sudo apt-get install libncurses5-dev


# sudo make menuconfig
```

Per poter eseguire il comando di menuconfig è necessario installare prima i componenti *libncurses5*. Una volta eseguito il comando di configurazione, si aprirà una finestra come quella indicata in figura B.1 Nel caso in cui si volessero aggiungere o rimuovere moduli, bisognerà andare a selezionarli all'interno di questo menu.

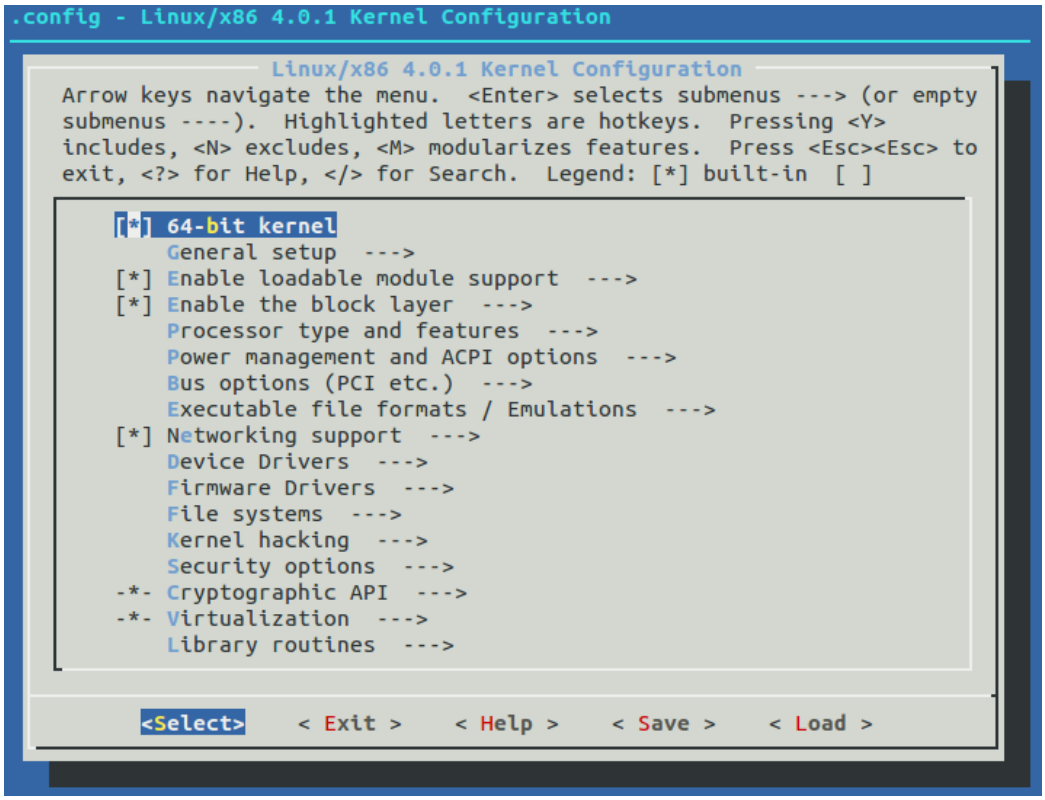

Figura B.1: Esecuzione del comando menuconfig.

Compilare il Kernel

```
# sudo make
```

Con questo comando si va a compilare il kernel. Questa operazione può richiedere diverso tempo, in particolare in dispositivi con prestazione ridotte.

È possibile anche specificare l'opzione -jN con N che indica il numero di core più uno (ad esempio *-j5* per un quad-core) per velocizzare l'operazione.

Compilare i moduli ed installarli

```
# sudo make modules
# sudo make modules_install
```

Installare il Kernel

```
# sudo make install
# sudo reboot
```

Come ultima operazione si va ad installare l'immagine del nuovo kernel. Una volta riavviato il sistema è possibile utilizzare il proprio kernel customizzato.

Bibliografia

[1] ISO 7498-2:1989 https://www.iso.org/obp/ui/#iso:std:14256:en.

[2] IEEE 802.11™ - 2012 IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications

[3] A. H. Lashkari, M. M. S. Danesh, and B. Samadi, "A survey on wireless security protocols (WEP, WPA and WPA2/802.11i)" in Proc. 2nd IEEE ICCSIT, Beijing, China, Aug. 2009.

[4] IETF RFC: 791 http://www.ietf.org/rfc/rfc791.txt

[5] IETF RFC: 2460 http://www.ietf.org/rfc/rfc2460.txt

[6] IETF RFC: 768 https://www.ietf.org/rfc/rfc768.txt

[7] B. Goode,"Voice Over Internet Protocol (VoIP)", Proceedings of the IEEE, vol.90, No.9, sept. 2002

[8] IETF RFC: 3550 http://www.ietf.org/rfc/rfc3550.txt

[9] ITU-T Recommendation H.323 (12/09)

[10] IETF RFC: 3261 http://www.ietf.org/rfc/rfc3261.txt

[11] https://www.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.0.1

[12] http://www.linuxfoundation.org/collaborate/workgroups/networking/sk_buff

[13] R. Koodli, "Mobile IPv6 Fast Handovers", IETF RFC 4068, June 2008.

[14] H. Soliman, C. Castelluccia, K. ElMalki, L. Bellier, "Hierarchical Mobile IPv6 (HMIPv6) Mobility Management", IETF RFC 5380, October 2008.

[15] S.Gundavelli, K.Leung, V.Devarapalli, K. Chowdhury, B. Patil "Proxy Mobile IPv6", IETF RFC 5213, August 2008.

[16] R. Moskowitz, P. Nikander, P. Jokela, T. Henderson "Host Identity Protocol", IETF RFC 5201, April 2008.

[17] F. Teraoka, "LIN6: A Solution to Multihoming and Mobility in IPv6", IETF Internet Draft, draft-teraoka-multi6-lin6-00.txt, 2006.

[18] T.M. Lim, Chai Kiat Yeo, Francis Bu Sung Lee, Quang Vinh Le, "TM-SP: Terminal Mobility Support Protocol", IEEE Transactions on Mobile Computing, vol. 8, no. 6, pp. 849-863, June 2009.

[19] IEEE Std 802.11r-2008; "Amendment 2: Fast Basic Service Set (BSS) Transition"; http://standards.ieee.org/getieee802/download/802.11r-2008.pdf.

[20] IEEE 802.11e: Wireless LAN Medium Access Control and Physical Layer Specification: Amendment for Quality of Service Enhancements. 2005.

[21] V. Ghini, S. Ferretti, and F. Panzieri. The "always best packet switching" architecture for sip-based mobile multimedia services. Journal of Systems and Software, 2011.

[22] V. Ghini, G. Lodi, F. Panzieri, Always Best Packet Switching: the Mobile VoIP Case Study, Journal of Communications, vol. 4, no. 9, October 2009.

[23] http://lisp.cisco.com/lisp_over.html

Ringraziamenti

Qui possiamo ringraziare il mondo intero!!!!!!!!!!
Ovviamente solo se uno vuole, non è obbligatorio.