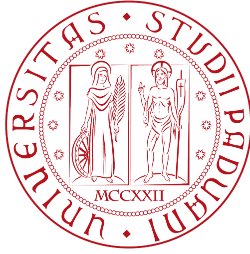


Università degli studi di Padova
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Analisi dell'algoritmo SCHC per la compressione di pacchetti IP su reti Lo-Ra

Relatore
Prof. Lorenzo Vangelista

Laureando
Claudio Tommasi
matr. 1128714

4 Dicembre 2017
Anno Accademico 2016/2017

Alla mia famiglia.

Aequam memento rebus in arduis servare mentem,
non secus in bonis ab insolenti temperatam laetitia.
Orazio

Abstract

This work aims to provide a deep analysis of the Static Context Header Compression (SCHC) algorithm, that represents the state-of-the-art for the compression and fragmentation of IP packets over the Low-Power Wide-Area Networks (LPWANs).
After an initial description of LoRa technology, the problems arising from the lack of IP support and the need to introduce the SCHC, the work focuses on how SCHC works and compresses the header of the internet stack protocols.
Finally, it is presented the implementation that has been developed and that is positioned as one of the first available to date. Then, the great benefits that have been achieved in terms of efficiency are highlighted.
The work ends with a reflection on the limits and the future of SCHC, that is a key point for the growing adoption of the Internet of Things.

Sommario

Questo lavoro si pone l'obiettivo di fornire un'analisi approfondita dell'algoritmo Static Context Header Compression (SCHC), che rappresenta lo stato dell'arte per quanto riguarda la compressione e la frammentazione di pacchetti IP su reti Low-Power Wide-Area Network (LPWAN).
Dopo la descrizione della tecnologia LoRa e dei problemi che causano il mancato supporto al protocollo IP e quindi la necessità di introdurre questo nuovo meccanismo, il lavoro si focalizza sulla descrizione del funzionamento di SCHC per la compressione dell'header dei diversi protocolli dello stack internet. Viene poi presentata l'implementazione che è stata realizzata e che si posiziona come una delle prime disponibili ad oggi. Sono quindi evidenziati i grandi benefici che si sono riusciti ad ottenere in termini di efficienza.
Il lavoro si conclude con una riflessione sui limiti e sul futuro di SCHC, che rappresenta un fattore chiave per la crescita dell'Internet of Things.

Indice

Abstract I

Sommario III

Elenco delle figure VII

Elenco delle tabelle VIII

1 Introduzione **1**

1.1 Introduzione . . . 1

2 LoRa **5**

2.1 LoRa . . . 5

2.2 LoRaWAN . . . 7

2.2.1 Regolamentazione . . . 8

2.2.2 Classi LoRaWAN . . . 10

2.2.3 Struttura dei messaggi LoRaWAN . . . 11

2.2.4 Attivazione dei nodi della rete . . . 14

3 SCHC **17**

3.1 Static Context Header Compression . . . 17

3.2 Compressione . . . 18

3.2.1 Terminologia e regole nel context . . . 20

3.2.2 Header pacchetto IPv6 . . . 22

3.2.3 Matching Operator . . . 24

3.2.4 Azioni di compressione/decompressione . . . 25

3.2.5 Estensione a CoAP . . . 28

3.2.6 Layered SCHC . . . 32

3.3 Frammentazione 34
3.3.1 Livelli di affidabilità 34
3.3.2 Terminologia e formati dei pacchetti 35
3.3.3 Funzionamento della frammentazione 39
3.3.4 Funzionalità aggiuntive 43
3.3.5 Sicurezza 44

4 Implementazione di SCHC **47**
4.1 Implementazione di SCHC 47
4.1.1 Tipi di dato 48
4.1.2 Caricamento delle regole nel context 50
4.1.3 Metodo per la compressione 51
4.1.4 Metodo per la decompressione 54
4.1.5 Metodi utili per la creazione e gestione dei pacchetti 56
4.2 Performance del sistema 57

5 Conclusioni e sviluppi futuri **61**

Appendice **63**
A Codice schc.h 63
B Codice comp_decomp.cpp 65

Bibliografia **79**

Elenco delle figure

1.1 Tecnologie legate al mondo IoT [3] 2

2.1 Stack protocollo LoRa 5
2.2 Spettrogramma di un segnale LoRa [7] 6
2.3 Architettura rete LoRaWAN [10] 7
2.4 LoRaWAN Classe A 10
2.5 LoRaWAN Classe B [15] 11
2.6 LoRaWAN Classe C 11
2.7 Struttura di un messaggio LoRaWAN [9] 12
2.8 Struttura del MACPayload [7] 13
2.9 Struttura messaggio di join request [9] 15
2.10 Struttura messaggio di join accept [9] 16

3.1 Schema architettura SCHC [5] 19
3.2 Regole nel context [5] 21
3.3 Esempio di pacchetto compresso 22
3.4 Header pacchetto IPv6 23
3.5 Esempio di regola di SCHC [5] 24
3.6 Azioni di compressione e decompressione 25
3.7 Estratto regola con mapping-sent [5] 27
3.8 Estratto regola con LSB [5] 28
3.9 Struttura del protocollo CoAP. 29
3.10 Struttura messaggio protocollo CoAP. 30
3.11 Esempio di regola SCHC per CoAP [16] 32
3.12 Regole con uguale contenuto IPv6 in SCHC [19] 33
3.13 Contesti multipli in Layered SCHC [19] 33
3.14 Struttura di un frammento. 36

3.15 Struttura header frammenti in modalità No ACK 37
3.16 Struttura header frammenti in modalità Window 37
3.17 Struttura messaggi ACK 38
3.18 Esempio ACK con bitmap [5] 39
3.19 Diagramma trasmissione frammenti in modalità No ACK [5] 40
3.20 Diagrammi con trasmissione frammenti in modalità Window-"on error" [5] 41
3.21 Diagrammi con trasmissione frammenti in modalità Window-"always" [5] . 42
3.22 Diagramma buffer reservation attack [20] 44
3.23 Diagramma duplication attack di frammenti [20] 45
3.24 Esempio di content-chaining schema 45

4.1 Istogramma performance di compressione 58
4.2 Istogramma space saving con SCHC 58

Elenco delle tabelle

2.1 Bande di frequenza adottata da LoRa nelle diverse regioni del mondo [11] 6
2.2 Range copertura e Tx data rate in base allo SF in reti LoRa 7
2.3 Limitazioni del duty cycle [12]. 9
2.4 Descrizione delle opzioni offerte da MType. 12
2.5 Descrizione dei comandi MAC 14

3.1 Payload e Tx data rate in base allo SF in reti LoRa 18
3.2 Descrizione codici di risposta CoAP 31

4.1 Struttura del tipo di dato regola 49
4.2 Struttura del tipo di dato campo di una regola 49
4.3 Esempio regola di tipo Not Sent 50
4.4 Esempio regola di tipo Mapping Sent 51
4.5 Pseudocodice algoritmo di compressione 52
4.6 Pseudocodice algoritmo di decompressione 54
4.7 Tempi di compressione e decompressione 59

1

Introduzione

Il capitolo descrive il contesto generale in cui si inserisce questo lavoro di tesi, esponendo il problema delle reti LPWAN nel supportare il protocollo IP, cosa che rende necessaria l'adozione di una nuova strategia di compressione chiamata SCHC.

1.1 Introduzione

Negli ultimi anni il numero di persone in grado di connettersi ad Internet è aumentato in maniera considerevole grazie alla diffusione della connessione a banda larga e ai progressi tecnologici nella produzione di componenti elettronici sempre più a basso costo. Un nuovo fenomeno sta ora prendendo piede in maniera esponenziale ed è quello che va sotto il nome di Internet of Things (IoT). Si stima che nel 2021 i dispositivi IoT potenzialmente connessi alla rete saranno oltre 16 miliardi a dispetto delle sole 4 miliardi e mezzo di persone collegate [1]. Questo significa che se Internet è ciò che ha permesso all'uomo di comunicare, l'Internet of Things rappresenta quel nuovo paradigma tecnologico che consentirà agli "oggetti" di diventare a pieno titolo i principali protagonisti della rete [2]. Dietro al termine IoT si nascondono tutta una serie di tecnologie e di standard con caratteristiche diverse che competono tra loro per aggiudicarsi il mercato. Tra queste troviamo, per comunicazioni a cortissimo raggio, NFC (Near Field Communication) e RFID (Radio Frequency IDentification), e per quanto riguarda distanze corte e medie Bluetooth, ZigBee, 6LoWPAN e Wi-Fi. Infine, per quei dispositivi che necessitano di trasmettere su lunghe distanze si possono trovare tecnologie che lavorano su frequenze

licenziate come 3G, 4G e il futuro 5G, oppure su bande libere come fatto da alcune delle Low Power Wide Area Network (LPWAN) quali, ad esempio, LoRa e SIGFOX.

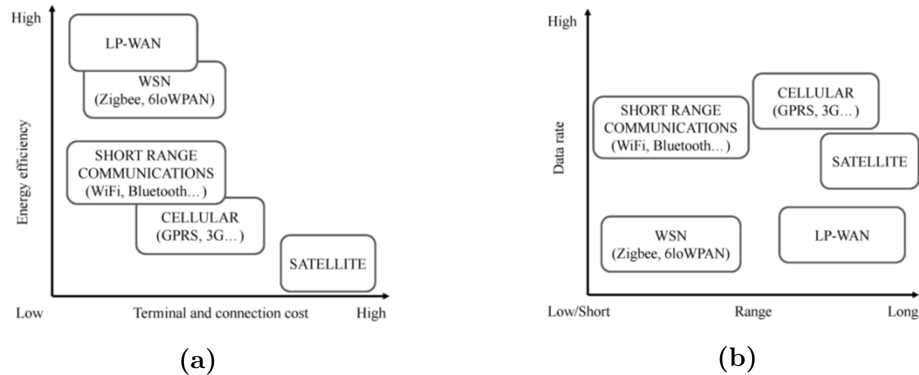


Figura 1.1: Rappresentazione delle caratteristiche delle tecnologie legate al contesto IoT. Con (a) raffigurante l'andamento dei costi e dell'efficienza energetica a seconda della tecnologia utilizzata e (b) la copertura raggiunta rispetto al data-rate. [3]

In questo lavoro di tesi ci si è concentrati sulle reti LPWAN che, come evidenziato in figura 1.1, hanno un costo relativamente basso e un'efficienza energetica elevata. Quest'ultima caratteristica è molto importante in quanto consente di avere dei dispositivi con una durata delle batterie che può raggiungere i dieci anni. Oltre a ciò, queste reti offrono una grande scalabilità, permettendo di gestire migliaia di device dispiegati ad una distanza che può arrivare a qualche decina di chilometri [4]. Nonostante queste caratteristiche positive, queste tecnologie soffrono di un data rate basso e di conseguenza di un frame, a livello data link, di dimensioni così ridotte da non supportare quei protocolli Internet fondamentali come IP (Internet Protocol) e UDP (User Datagram Protocol). Poiché ciò rappresenta una grave limitazione all'adozione di queste nuove tecnologie di rete, in un mondo che di fatto comunica attraverso questi protocolli, all'interno della comunità di IETF (Internet Engineering Task Force), organizzazione internazionale responsabile dello sviluppo di numerosi standard alla base di Internet, tra cui quelli citati in precedenza, nasce a ottobre 2016 il gruppo di lavoro LPWAN WG che ha proposto come soluzione un nuovo algoritmo chiamato Static Context Header Compression (SCHC) [5] che rappresenta l'oggetto di studio della tesi. Questa tecnica permette di

comprimere l'header dei protocolli utilizzati per i diversi strati dello stack IoT come IP per la rete, UDP per il trasporto e CoAP (Constrained Application Protocol) per il lato applicazione, cercando di ridurre il tutto ad un semplice identificatore di pochi bit, rappresentante la regola di compressione utilizzata. In questo modo è possibile rientrare nelle dimensioni del frame offerto dalle reti LPWAN. Questa tesi si occupa di approfondire l'algoritmo SCHC con particolare attenzione alle reti LoRa, andando a verificare come questo meccanismo possa essere inserito su questa promettente tecnologia ed i benefici in termini di performance che si possono ottenere.

Questo lavoro si compone di un *secondo capitolo* in cui verranno presentate le reti LoRa e le loro caratteristiche, dopodiché nel *terzo capitolo* sarà ampiamente descritto l'algoritmo SCHC e le strategie da esso utilizzate per la compressione e la frammentazione dei pacchetti IP. Nel *quarto capitolo* verrà mostrata l'implementazione che è stata sviluppata e ne verranno discussi i risultati ottenuti. Infine nell'*ultimo capitolo* verranno esposte le conclusioni a cui si è giunti attraverso questo lavoro di tesi e le idee con cui si potrà procedere ad ampliare il progetto.

2

LoRa

In questo secondo capitolo viene data una descrizione completa di LoRa, una delle tecnologie più promettenti all'interno della categoria delle Low-Power Wide-Area Network (LPWAN).

2.1 LoRa

Quando si parla di LoRa, acronimo di Long Range, bisogna porre attenzione a ciò a cui ci si riferisce in quanto lo stack LoRa si compone di due livelli, lo strato fisico che utilizza una modulazione proprietaria [8] derivata dal Chirp Spread Spectrum (CSS) e il protocollo per il livello MAC chiamato LoRaWAN [9].

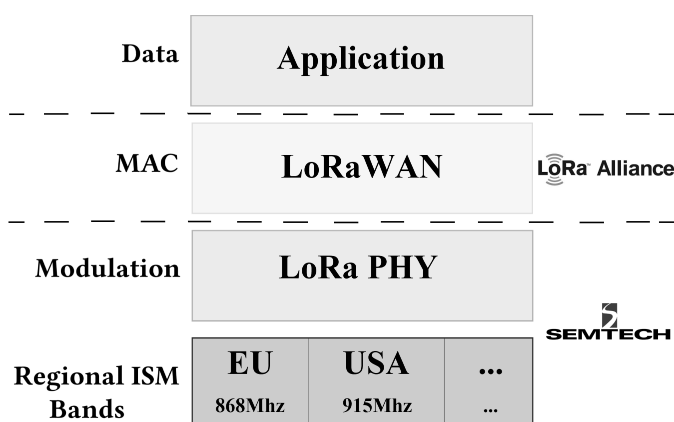


Figura 2.1: Stack del protocollo LoRa.

Per quanto riguarda il physical layer la tecnologia è stata sviluppata e brevettata da Cycleo, una società francese, che nel 2012 è stata acquisita dalla californiana Semtech.

LoRa utilizza una velocità di trasmissione variabile, in maniera da poter controllare il data rate e il consumo di potenza ottimizzando di conseguenza le prestazioni della rete in base alle necessità. Essa adotta un meccanismo per la rilevazione e la successiva correzione degli errori (FEC) ed aumenta la sensibilità del ricevitore usando per intero la larghezza di banda offerta, rendendo più robusto il sistema al rumore di canale.

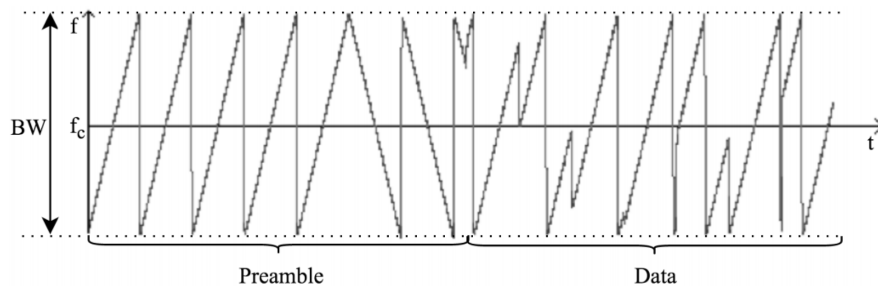


Figura 2.2: Spettrogramma di un segnale LoRa. [7]

Per quanto riguarda le frequenze LoRa utilizza le bande ISM, ossia quelle bande non licenziate e utilizzabili per finalità industriali, scientifiche e mediche, ma non commerciali. In particolare, a seconda dell'area geografica e delle sue regolamentazioni, LoRa adotta delle frequenze differenti, che sono gli 868 MHz per l'Europa, i 915 MHz per il Nord America e i 780 MHz per la Cina come descritto nella tabella 2.1 [11].

Regione	Banda di frequenza [MHz]
Asia	923
Australia	915–928
Cina	779–787 e 470-510
Europa	863–870 e 433
India	865-867
USA	902–928

Tabella 2.1: Bande di frequenza adottata da LoRa nelle diverse regioni del mondo. [11]

Le prestazioni che si possono ottenere dipendono da un parametro, chiamato spreading factor, il quale può variare dal valore 7 al 12, nonostante le specifiche LoRaWAN lo limitino entro il 10. Con uno SF pari a 7 si può raggiungere, come descritto nella tabella 2.2, un bitrate elevato dell'ordine dei 5 kbit/s, con un basso consumo di energia ma con

un range di trasmissione intorno ai 2 km. Aumentando lo spreading factor fino a 12, viceversa, si possono raggiungere anche i 14 km di copertura, ma con un bitrate che scende a soli 250 bit/s ed un alto dispendio dal punto di vista energetico.

Spreading Factor	Tx data rate	Range copertura
SF 7	5470 bit/s	2 km
SF 8	3125 bit/s	4 km
SF 9	1760 bit/s	6 km
SF 10	980 bit/s	8 km
SF 11	440 bit/s	11 km
SF 12	250 bit/s	14 km

Tabella 2.2: Range copertura e Tx data rate in reti LoRa a seconda dello Spreading Factor.

2.2 LoRaWAN

A differenza di quanto descritto per lo strato fisico, il protocollo che copre i livelli più alti, chiamato LoRaWAN [9], è di tipo aperto. Le specifiche sono descritte dalla LoRa Alliance, associazione no-profit fondata da aziende leader del settore con la missione di standardizzare il protocollo e diffonderlo. L'architettura proposta è quella in figura 2.3, con una topologia a stella di stelle. Gli elementi principali all'interno della rete sono tre: gli end device, i gateway e il Net Server.

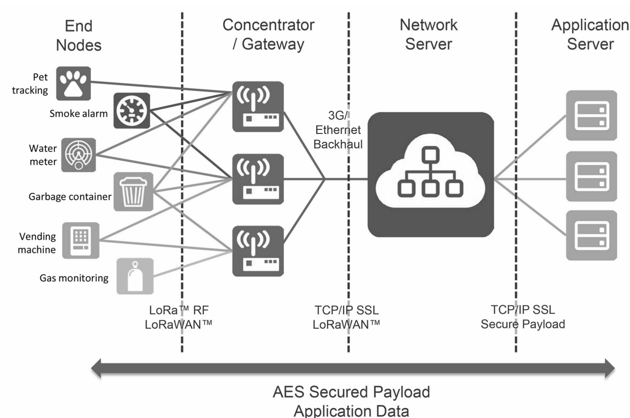


Figura 2.3: Architettura di una rete LoRaWAN [10].

Gli end device comunicano con i gateway attraverso la modulazione proprietaria LoRa. Questi a loro volta, in maniera logicamente trasparente, spediscono attraverso una connessione IP, instradata su Ethernet, Wi-Fi o 3G, i messaggi al Network Server aggiungendo solamente delle informazioni riguardanti la qualità della comunicazione. Più gateway possono captare uno stesso messaggio spedito dell'end device e sarà poi il Net Server a doversi preoccupare della gestione dei duplicati e della selezione del miglior gateway da utilizzare in caso ci dovesse essere un successivo messaggio di risposta al dispositivo. La comunicazione è di tipo bidirezionale, ma la trasmissione in uplink è quella più frequente, considerata la natura degli end device che solitamente hanno l'obiettivo di raccogliere dati per poi mandarli al server. La comunicazione tra end-device e gateway, quindi le frequenze utilizzate ed il data-rate si modificano in base alle esigenze e a seconda della distanza utilizzando il meccanismo dell'adaptive data rate (ADR) che consente oltretutto di aumentare l'efficienza energetica e quindi di conseguenza la durata delle batterie.

2.2.1 Regolamentazione

Gli end device devono rispettare i vincoli stabiliti dai diversi paesi per quanto riguarda il duty-cycle e la durata della trasmissione. Inoltre il canale selezionato va cambiato in maniera casuale ad ogni nuova comunicazione, in modo da rendere la rete meno soggetta alle interferenze. Infatti, nonostante si utilizzino bande non licenziate, questo non significa che ci si trovi liberi di fare ciò che si vuole, si deve invece sottostare ad una serie di regole e limitazioni. Ci sono diversi livelli di regolamentazione per quanto riguardo l'utilizzo dello spettro elettromagnetico: a livello globale troviamo l'Unione internazionale delle telecomunicazioni, con acronimo ITU (International Telecommunication Union) che è un'organizzazione internazionale che si occupa di definire gli standard nelle telecomunicazioni e nell'uso delle onde radio. Dal 1947 l'ITU è diventata una delle agenzie specializzate delle Nazioni Unite, con sede a Ginevra, e coordina gli organismi a livello regionale e nazionale. A livello europeo poi troviamo tre organismi che in

coordinazione tra loro hanno voce in materia di telecomunicazioni e sono l'European Telecommunications Standards Institute (ETSI) che si occupa di stabilire gli standard di comunicazione europei, l'Electronic Communications Committee (ECC) che è parte dell'European Conference of Postal and Telecommunications Administrations (CEPT) ed infine la Commissione Europea stessa. A livello nazionale, i singoli stati definiscono come allocare le frequenze e a chi destinare, anche attraverso delle licenze, le bande. Per quanto riguarda l'Italia l'organo preposto è il Ministero dello Sviluppo Economico (MISE) che con la pubblicazione sulla Gazzetta Ufficiale del Piano Nazionale di Ripartizione delle Frequenze (PNRF) [13] regola le bande di frequenza comprese tra 0 e 3000 GHz e stabilisce in tempo di pace l'attribuzione delle bande di frequenza ai diversi servizi ed indica di ciascun servizio l'autorità governativa atta alla gestione di tale frequenze e le principali utilizzazioni civili [14]. Per quanto riguarda le regole imposte da ETSI, in ambito europeo per le frequenze usate da LoRa, come descritto in precedenza nella tabella 2.1 quelle tra gli 868 MHz e gli 870 MHz, troviamo differenti vincoli sul duty cycle [12]. Con quest'ultimo termine si intende il rapporto, espresso in termini percentuali, del tempo massimo in cui il dispositivo può trasmettere nell'arco di un'ora. I diversi limiti di duty cycle dipendono dalla sottobanda in cui ci troviamo, e un riepilogo è rappresentato in tabella 2.3.

Banda di frequenza [MHz]	**Duty cycle**
(g) 867-868	1%
(g1) 868-868.6	1%
(g2) 868.7-869.2	0.1%
(g3) 869.4-869.65	10%
(g4) 869.7-870	1%

Tabella 2.3: Limitazioni del duty cycle [12].

Nel caso in cui il dispositivo utilizzi una frequenza il cui limite sul duty cycle è l'1%, il tempo massimo per una trasmissione singola è di 36 secondi in un'ora, e tra una trasmissione e l'altra il vincolo da rispettare in termini di tempo di attesa T_{wait} è legato al tempo di trasmissione T_{tx} e al valore di $DutyCycle = 0.01$ attraverso l'equazione

$T_{wait} = (\frac{T_{tx}}{DutyCycle}) - T_{tx}$. La formula rimane valida anche con gli altri valori di duty cycle. Questi limiti si applicano a livello di sottobanda, quindi un dispositivo che trasmette ad esempio sulla sottobanda g1, descritta in tabella 2.3, una volta inviato un messaggio potrebbe subito mandare il successivo utilizzando una delle altre sottobande, ovviamente dovendone rispettare le limitazioni, aggirando così il T_{wait} di g1.

2.2.2 Classi LoRaWAN

Le specifiche di LoRaWAN definiscono tre tipologie di end-node: di classe A, di classe B e di classe C. Tutti i dispositivi compatibili LoRaWAN devono implementare la classe A, dopodiché le classi B e C rappresentano delle estensioni. Una descrizione di ciascuna classe è fornita in seguito:

– *Classe A:* rappresenta la modalità di default dei nodi. I dispositivi supportano la comunicazione bidirezionale con il gateway. I messaggi in uplink, ossia dal dispositivo al server, possono essere inviati in qualsiasi momento. A seguito di un messaggio di questo tipo il dispositivo apre due finestre di ricezione. Il server può rispondere in una delle due finestre. Solitamente la prima finestra è aperta sullo stesso canale utilizzato nella trasmissione in uplink, viceversa la seconda finestra, come già accordato in precedenza con il server, viene aperta su una banda differente per migliorare la resistenza alle oscillazioni del canale. Questa classe è la più efficiente dal punto di vista energetico ed è utilizzata da quei dispositivi che cercano di mantenersi spenti per il più lungo tempo possibile, ed in cui le comunicazioni in uplink sono le più frequenti, come ad esempio nei sensori.

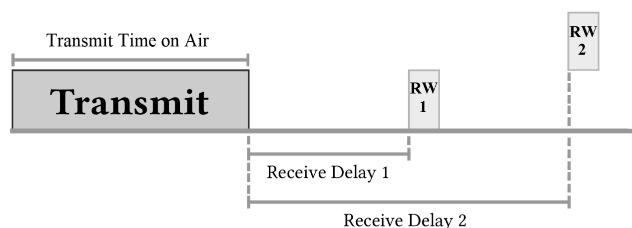


Figura 2.4: Finestra di trasmissione e ricezione in dispositivi di classe A.

– *Classe B:* questi dispositivi sono sincronizzati con il Net Server attraverso un meccanismo che sfrutta dei pacchetti, detti beacon, trasmessi dai gateway. Un beacon contiene uno specifico tempo di riferimento in cui far aprire ai nodi della rete una finestra di ricezione extra, chiamata ping slot. In questo modo i dispositivi possono ricevere dati in downlink o comandi indipendentemente dai momenti di uplink. Questa classe trova impiego ad esempio negli attuatori, che hanno la necessità di ricevere degli ordini da parte di un server.

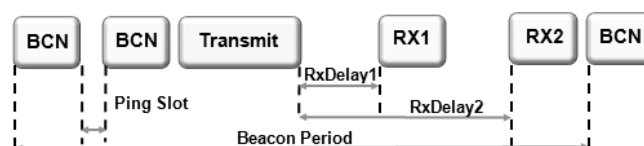


Figura 2.5: Finestra di trasmissione e ricezione in dispositivi di classe B [15].

– *Classe C:* questa tipologia di nodi rappresenta dispositivi sempre in ascolto, ad eccezione dei momenti in cui stanno comunicando, pensati per quei casi in cui la comunicazione deve rispettare vincoli dal punto di vista temporale. Questa operatività si traduce in un elevato consumo energetico che rende solitamente necessario che questi dispositivi siano connessi alla rete elettrica.

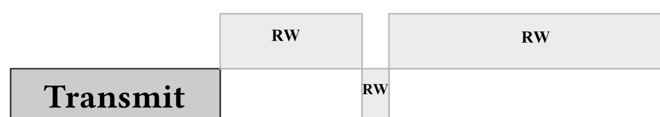


Figura 2.6: Finestra di trasmissione e ricezione in dispositivi di classe C.

2.2.3 Struttura dei messaggi LoRaWAN

LoRaWAN si occupa di definire anche la struttura dei pacchetti a livello fisico e MAC e tutti i parametri necessari per il corretto funzionamento della rete. La composizione dei messaggi è visibile in figura 2.7 nei suoi diversi strati. Il pacchetto a livello fisico si compone di un preambolo che serve a identificare il segnale, un Physical Header (PHDR) e il rispettivo Physical Payload (PHYPayload) in cui sono inserite le informazioni dello

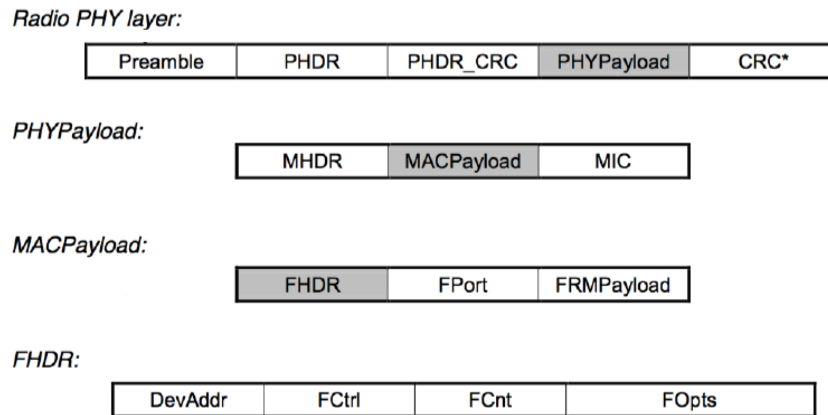


Figura 2.7: Struttura di un messaggio LoRaWAN [9].

strato superiore. Vi sono inoltre dei codici CRC per il controllo degli errori, che nei messaggi in downlink riguardano solo l'header (PHDR_CRC), mentre per quelli in uplink anche il payload (CRC). Passando poi al contenuto del PHYPayload, che corrisponde quindi al frame di livello MAC, esso si compone di tre parti. La prima consiste nel MAC Header (MHDR), formato da 8 bit, dei quali 3 bit sono usati dal campo interno chiamato MType, che ppecifica il tipo di messaggio, e le cui tipologie sono descritte in tabella 2.4. Altri 3 bit sono dichiarati reserved for future usage (RFU), ossia lasciati liberi per eventuali esigenze future, ed infine gli ultimi due bit appartengono al Major che specifica la versione LoRaWAN usata.

MType	Descrizione
000	Richiesta di Join
001	Join accettato
010	Messaggio Uplink non confermato
011	Messaggio Downlink non confermato
100	Messaggio Uplink confermato
101	Messaaggio Dowlink confermato
110	RFU (riservato per uso futuro)
111	Proprietario/Riservato

Tabella 2.4: Descrizione delle opzioni offerte da MType.

Le altre due parti di cui si compone il PHYPayload sono il MAC Payload e il MIC

(Message Integrity Code), quest'ultimo è atto a garantire l'integrità del messaggio. A sua volta il MAC Payload, rappresentato in figura 2.8, è suddiviso in tre parti:

- *Frame Header (FHDR):* descrive attraverso il campo DevAddr l'indirizzo da 32 bit che identifica l'end device. Contiene inoltre il campo FCtrl che serve a gestire correttamente la funzionalità di Adaptive Data Rate (ADR). Il campo Frame Counter (FCnt) contiene invece due contatori, uno dei quali che conta i frame spediti in uplink e tenuto aggiornato dal nodo stesso, e l'altro che conta i frame ricevuto in downlink e incrementato dal server.

- *Frame Port (FPort):* definisce la porta di una specifica applicazione che dovrà ricevere il messaggio. Nel caso in cui il valore risulti essere 0 significa che il FRMPayload contiene solo un MAC command. Il campo è opzionale.

- *Frame Payload (FRMPayload):* rappresenta il campo fondamentale in cui vengono inseriti i dati da trasmettere attraverso il messaggio. Il campo è crittografato ed è opzionale.

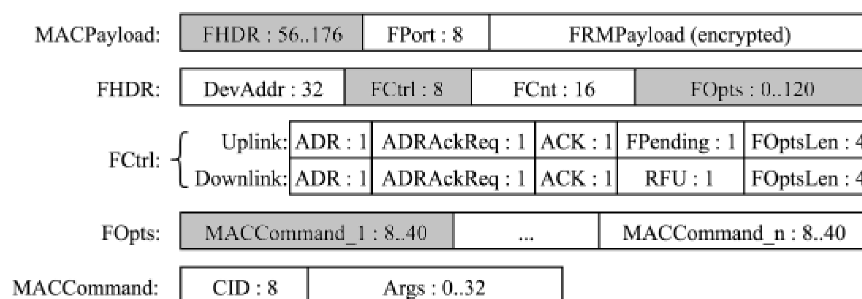


Figura 2.8: Struttura del MACPayload [7].

I comandi MAC (MACCommand), descritti nella tabella 2.5, servono per l'amministrazione della rete, e sono scambiati tra il Network Server e gli end device a livello MAC, rimanendo nascosti al livello applicativo. Questi possono essere trasportati dal campo FOpts del Frame Header, oppure inseriti nel campo FRMPayload, ed in questo secondo caso FPort deve essere impostata a zero. Se trasportati nel campo FOpts non sono criptati e possono avere una dimensione massima di 15 ottetti, nel caso invece siano

inseriti nel Payload i comandi sono criptati, diventando quindi una scelta obbligata nel caso si cerchi una maggiore sicurezza. I comandi a disposizione permettono di conoscere il livello di batteria del dispositivo, di cambiare il data rate, il canale di comunicazione, il duty-cycle e il timing degli slot di comunicazione e di settare tanti altri parametri come spiegato nella tabella 2.5 sottostante.

Comando	**Trasmesso da**	**Descrizione**
LinkCheckReq	End-device	Usato dall'end-device per validare la sua connessione alla rete
LinkCheckAns	Gateway	Risponde ad un comando LinkCheckReq inviando dati sulla qualità della connessione (potenza del segnale e link margin)
LinkADRReq	Gateway	Richiede all'end-device di cambiare il data-rate, la potenza di trasmissione o il canale
LinkADRAns	End-device	Ack in risposta al comando LinkRateReq
DutyCycleReq	Gateway	Setta il massimo duty-cycle aggregato di trasmissione del dispositivo
DutyCycleAns	End-device	Ack in risposta al comando DutyCycleReq
RXParamSetupReq	Gateway	Setta i parametri degli slot di ricezione
RXParamSetupAns	End-device	Ack in risposta al comando RXSetupReq
DevStatusReq	Gateway	Richiede lo stato all'end-device
DevStatusAns	End-device	Risponde con lo stato dell' end-device (livello di batteria e margine di demodulazione)
NewChannelReq	Gateway	Crea o modifica l'impostazione di un canale radio
NewChannelAns	End-device	Ack in risposta al comando NewChannelReq
RXTimingSetupReq	Gateway	Setta il timing degli slot di ricezione
RXTimingSetupAns	End-device	Ack in risposta al comando RXTimingSetupReq

Tabella 2.5: Descrizione dei comandi MAC.

2.2.4 Attivazione dei nodi della rete

Ogni dispositivo all'interno di una rete LoRaWAN dispone di queste informazioni: il proprio indirizzo identificativo del device chiamato DevAddr, un identificativo di applicazione (AppEUI), una chiave di sessione di rete (NwkSKEY) ed infine una chiave di

sessione per l'applicazione (AppSKey). Per quanto riguarda il DevAddr esso si compone di 32 bit di cui i sette più significativi identificano la rete mentre i rimanenti sono assegnati in maniera arbitraria dal network manager. L'AppEUI è un ID globale composto da 64 bit che identifica l'applicazione utilizzata dal nodo e che essendo di solito specifica è settata a priori nel dispositivo. La NwkSKey è una chiave AES-128 bit che viene invece utilizzata dal Network Server e dal device per generare il MIC per il controllo dell'integrità del messaggio e per criptare e decriptare il FRMPayload. Infine l'AppSKey è una chiave AES-128 bit utilizzata dall'Application Server e dall'end-node per criptare e decriptare il payload dei messaggi specifici di tale applicazione, in maniera da avere in questo modo una sicurezza end-to-end.
LoRaWAN mette a disposizione due strategie affinché un nuovo end device possa partecipare alla rete che sono la Over-The-Air Activation (OTAA) e la Activation By Personalization (ABP). Nel primo caso gli end device devono partecipare ad una procedura di join prima di poter scambiare dati con il network server. Questa procedura deve essere reiterata ogniqualvolta le informazioni riguardo la sessione siano perse. Per eseguire questa procedura il dispositivo deve possedere un DevEUI che identifica globalmente il device, l'identificatore di applicazione (AppEUI) ed una chiave AES-128 come AppKey. Questa chiave sarà utilizzata per generare le due chiavi di sessione AppSKey e la NwkSKEY. La procedura di join consiste in un messaggio MAC di join request, rappresentato in figura 2.9, in cui un dispositivo invia anche il proprio DevEUI e l'AppEUI dopodiché il

Size (bytes)	8	8	2
Join Request	AppEUI	DevEUI	DevNonce

Figura 2.9: Struttura del messaggio di join request [9].

network server risponde con un messaggio di join accept, come in figura 2.10, mandando al mittente il DevAddr e un valore casuale chiamato AppNonce utilizzato dal dispositivo per ricavare l'AppSKey e la NwkSKEY. Nel caso invece dell'Activation By Personalization (ABP), il nodo non deve utilizzare la procedura di join request-join accept descritta in precedenza, ma il DevAddr, la NwkSKey e l'AppSKey vengono memorizzate diretta-

Size (bytes)	3	3	4	1	1	(16) Optional
Join Accept	AppNonce	NetID	DevAddr	DLSettings	RxDelay	CFList

Figura 2.10: Struttura del messaggio di join accept [9].

mente nel nodo rispetto al DevEUI, all'AppEUI e all'AppKey. In questo modo il nodo è già equipaggiato delle informazioni necessarie per partecipare alla specifica rete fin da subito. Per una questione di sicurezza sarebbe opportuno che queste chiavi fossero diverse per ogni dispositivo prodotto e che l'algoritmo utilizzato dalla casa produttrice fosse segreto e non fosse in alcun modo riconducibile a informazioni pubbliche del nodo.

3

SCHC

In questo capitolo viene descritto in maniera approfondita l'algoritmo SCHC, andando a spiegare il funzionamento e le strategie adottate dall'algoritmo per la compressione degli header e la frammentazione dei pacchetti.

3.1 Static Context Header Compression

L'algoritmo SCHC (Static Context Header Compression), sviluppato da IETF [5], rappresenta un meccanismo per la compressione degli header e per la frammentazione dei pacchetti. Esso si pone l'obiettivo di rispondere alle particolari esigenze delle reti LP-WAN (Low Power Wide Area Network). Queste nuove reti hanno delle caratteristiche molto vincolanti dal punto di vista della frame size offerta dal livello data link, in quanto sono pensate per connettere dispositivi che trasmettono qualche decina di byte, a basse velocità, nell'arco di un'intera giornata. In questa maniera è reso possibile da una parte inviare dati su distanze dell'ordine dei chilometri e dall'altra avere un bassissimo consumo energetico. Queste caratteristiche permettono ai dispositivi di raggiungere un'autonomia anche di dieci anni nel caso di un'alimentazione a batteria. La dimensione del payload risulta tuttavia nell'ordine di un centinaio di byte, e nel caso specifico di LoRa questo si attesta tra i 51 e i 242 byte in base allo Spreading Factor scelto, come riportato in tabella 3.1. Questi valori non riescono a soddisfare le richieste dei protocolli dello stack internet, tra cui IPv6, il quale richiede un minimo di 40 byte per l'header e un MTU (Maximum Transmission Unit) di almeno 1280 byte per evitare la frammentazione

dei pacchetti [6]. Per questa ragione è nata la necessità di trovare in primo luogo un metodo di compressione per l'header, in modo da rendere i dispositivi interoperabili con i protocolli UDP/IP, ma anche una formula per compiere una frammentazione intermedia dei pacchetti per tutte quelle reti LPWAN che già non la prevedessero in maniera nativa. Inoltre la tecnica SCHC si differenzia dagli schemi di compressione già presenti in letteratura come 6LoWPAN HC1/HC2 o IPHC/NHC, pensati per le reti IEEE 802.15.4, oltre che per la maggiore efficacia anche e soprattutto per la capacità di gestire l'header dei protocolli di livello applicativo quali ad esempio CoAP.

Spreading Factor	**Tx data rate**	**Payload massimo**
SF 7	5470 bit/s	242 byte
SF 8	3125 bit/s	242 byte
SF 9	1760 bit/s	115 byte
SF 10	980 bit/s	51 byte
SF 11	440 bit/s	51 byte
SF 12	250 bit/s	51 byte

Tabella 3.1: Payload massimo e Tx data rate in reti LoRa a secondo dello Spreading Factor

3.2 Compressione

L'algoritmo SCHC sfrutta due condizioni iniziali, comuni a tutte le reti LPWAN, ovvero:

1. la topologia a stella, per cui i pacchetti seguono sempre il medesimo percorso, come nella rete LoRaWAN vista in figura 2.3, che vanno dal Network Server all'end device passando da un gateway.
2. i dispositivi eseguono applicazioni costanti e specifiche, per cui la tipologia di traffico è conosciuta a priori.

Il sistema, avendo tale conoscenza della rete e del traffico che vi andrà a circolare, utilizza un contesto (context), ossia un insieme di regole, per comprimere l'header dei pacchetti. Questo contesto è statico, quindi i valori dei campi non cambiano nel tempo, poiché un'eventuale continua sincronizzazione comporterebbe una complessità e un consumo di

risorse troppo elevato. La configurazione di questo contesto può essere svolta attraverso un protocollo di provisioning, oppure, vista che la finalità del dispositivo è conosciuta a priori, settata in partenza direttamente sul device. Il sistema trasforma l'header di un pacchetto IP/UDP/CoAP in una Rule ID di pochi bit, che identifica una delle regole presenti nel contesto, ed eventualmente altre informazioni aggiuntive richieste dalla regola. Il meccanismo di compressione e decompressione, chiamato SCHC C/D (Compressor/Decompress), come mostrato nello schema in figura 3.1, contiene il context ed è posizionato nella rete LWPAN sia all'interno dei device, che a monte come parte del Network Gateway (NGW) o collegato ad esso. Tutte le regole presenti nel context dei dispositivi devono essere presenti anche nel context del NGW. Quando un device vuole comunicare con un Application Server (App) con pacchetti IP, l'SCHC C/D nel device deve selezionare dal contesto la regola di compressione corretta, e la Rule ID che la identifica sarà inserire nel frame da inviare al Radio Gateway, il quale a sua volta lo trasmetterà al Network Gateway. A questo punto l'SCHC C/D del NGW ricercherà la regola nel contesto, ed essa avrà le informazioni necessarie per ricostruire l'header del pacchetto originale. Il pacchetto sarà quindi spedito all'Application Server che a sua volta potrà rispondere al dispositivo seguendo il processo inverso appena descritto.

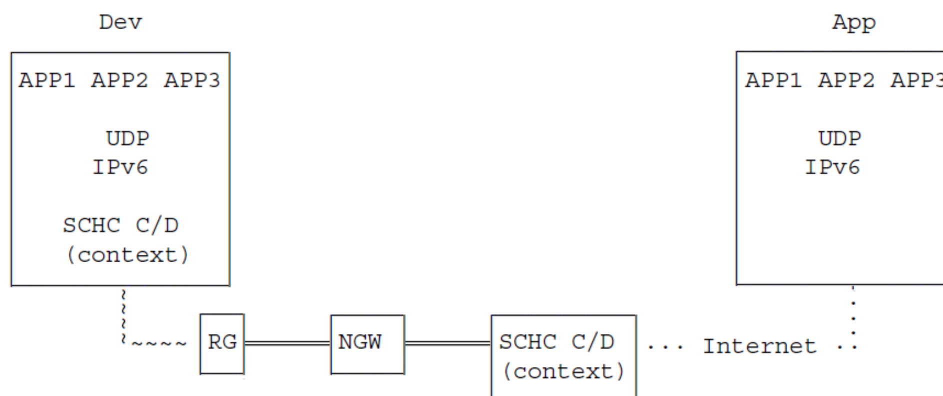


Figura 3.1: Schema architettura SCHC [5].

3.2.1 Terminologia e regole nel context

Le regole del context, identificate da una Rule ID, come descritto in figura 3.2 contengono una serie di parametri. Nelle regole la descrizione dei campi dell'header è fatta seguendo l'ordine dato dal formato del pacchetto. I parametri presenti sono i seguenti:

- *Field ID (FID):* un valore univoco che definisce il campo dell'header.
- *Field Position (FP):* dichiara, se esistono diverse istanze dello stesso campo nell'header, a quale di questi ci si riferisce. Il valore di default è 1.
- *Direction Indicator (DI):* indica quale direzione ha il pacchetto, e può essere settato UP LINK (Up) quando il campo è presente solo nel pacchetto che è inviato dal device all'Application Server. Viceversa il valore può essere DOWN LINK (Dw) quando il pacchetto viaggia in direzione contraria. Infine BIDIRECTIONAL (Bi) quando il campo è presente nei pacchetti scambiati in entrambe le direzioni.
- *Target Value (TV):* è il valore da comparare con quello presente nell'header del pacchetto. Questo può essere espresso sia come valore intero, sia come stringa oppure anche attraverso strutture più complesse come array, liste, JSON e CBOR.
- *Matching Operator (MO):* indica quale operatore verrà utilizzato, durante la fase di compressione, per confrontare il valore nel campo dell'header con quello presente nel Target Value. L'operatore può richiedere dei parametri aggiuntivi.
- *Compression Decompression Action (CDA):* descrive l'azione da adottare in fase di compressione e in quella di decompressione. Anche in questo caso potrebbero essere necessari ulteriori parametri.

La dimensione della Rule ID non è fissata ma dipende dall'implementazione e dalla tecnologia di rete LPWAN sul quale viene adottato SCHC. Alcuni valori di Rule ID possono essere riservati per la parte di frammentazione. In particolare le Rule ID sono specifiche per ciascun dispositivo e per identificare la Rule ID corretta va combinato la Rule ID

con l'identificatore di livello L2, chiamato Dev-ID, che è solitamente rappresentato dal MAC address del dispositivo.

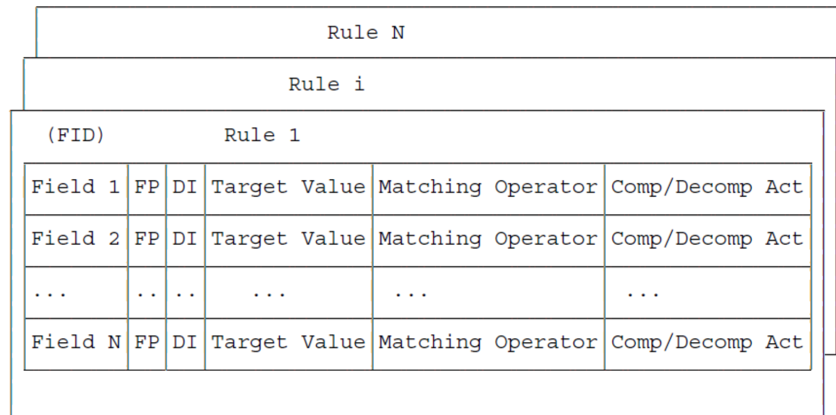


Figura 3.2: Rappresentazione delle regole nel context [5].

La compressione e decompressione dei pacchetti si compone di tre fasi: compressione, spedizione e decompressione. Durante la parte di compressione viene selezionata la Rule ID corretta da utilizzare per la compressione dell'header ed è l'SCHC C/D ad occuparsi di identificare il Dev-ID e la corretta Rule ID. Nel caso di comunicazioni solo in Uplink il Dev-ID non è necessario. Guardando il Director Indicator (DI) si escludono dalle regole quei campi non concordanti con questo valore. A questo di verificano che i campi identificati attraverso il field identifier (FID) coincidano con quelli presenti nell'header pacchetto, facendo attenzione però che anche la field position (FP) corrisponda, altrimenti la regola selezionata viene scartata e si passa alla successiva, finché sia FID che FP non corrispondono. A quel punto si dovrà andare a verificare il Matching Operator (MO). Se il risultato del MO dà esito positivo per tutti i campi, allora la regola selezionata è quella corretta e i campi vengono elaborati secondo quanto descritto nelle Compression Decompression Action (CDA) in modo da ottenere il pacchetto compresso. Altrimenti se il confronto da anche solo un esito negativo si deve procedere a testare la regola successiva. Nel caso in cui nessuna regola soddisfi la verifica il pacchetto non può essere compresso ed allora l'alternativa sarà utilizzare la frammentazione. La Rule

ID della regola selezionata viene spedita al destinatario del pacchetto assieme alle informazioni risultanti dalla procedura di compressione dell'header e al payload. Al frame vengono inoltre aggiunti dei bit detti di padding per rendere la dimensione un multiplo di 8 bit.

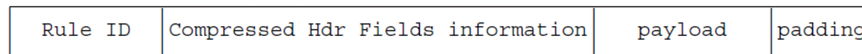


Figura 3.3: Esempio di pacchetto compresso.

Infine la parte di decompressione consiste nel ricevitore che identifica il mittente attraverso il Device ID, che come già dichiarato solitamente corrisponde con il MAC address, seleziona la regola corretta combinandolo con la Rule ID ricevuta. Attraverso l'azione descritta nel campo CDA il contenuto dell'header può essere correttamente ricostruito.

3.2.2 Header pacchetto IPv6

In figura 3.4 vediamo da quali campi è formato l'header di un pacchetto IPv6, che sono poi gli stessi che dobbiamo ritrovare all'interno delle regole del contesto, sotto il campo Field, come visibile nell'esempio in figura 3.5. I campi che caratterizzano l'header dei pacchetti IPv6 sono:

- *Version (4 bit):* indica la versione del protocollo IP. Nel caso di IPv6 esso assume il valore 6.
- *Traffic Class/DiffServ (8 bit):* descrive la classe di traffico utilizzata per controllo della congestione, assegnando una priorità ai pacchetti in base al livello scelto.
- *Flow Label (20 bit):* utilizzato per etichettare una sequenza di pacchetti come appartenenti allo stesso flusso. Viene usata per la gestione del QoS (Quality of Service).
- *Payload Length (16 bit):* indica la dimensione del payload, ossia la quantità di byte del pacchetto non utilizzate dall'header e che corrispondono ai dati proveniente

dagli strati superiori dello stack. Anche le eventuali estensioni dell'header utilizzate per l'instradamento o per la frammentazione sono conteggiate nella lunghezza del payload.

- *Next Header (8 bit):* indica quale tipo di header segue l'header di base IPv6. Nel caso in cui siano richiesti dei campi ulteriori per le opzioni, essi vengono indicati in uno o più header aggiuntivi che seguono l'header IPv6.
- *Hop Limit (8 bit):* indica il limite di salti consentito. Questo valore decrementa di uno ogni volta che un nodo inoltra il pacchetto. Quando il valore arriva a zero il pacchetto viene scartato.
- *Source Address (128 bit):* indica l'indirizzo IP del mittente del pacchetto.
- *Destination Address (128 bit):* indica l'indirizzo IP del destinatario del pacchetto.

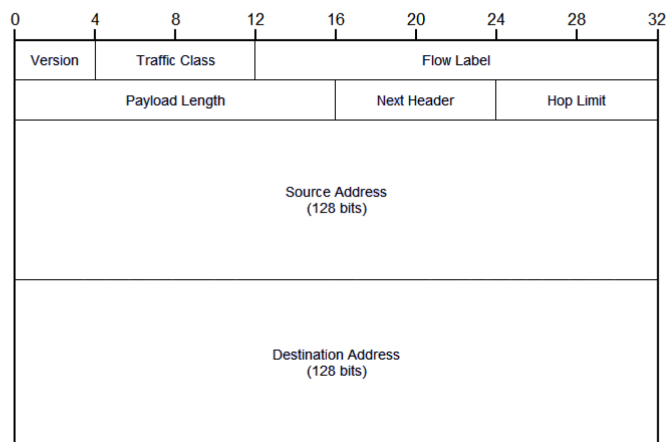


Figura 3.4: Header di un pacchetto IPv6.

All'interno della regola, come in figura 3.5, notiamo delle differenze per quanto riguarda l'addressing, in particolare a differenza dei campi Source e Destination Address troviamo invece il concetto di Dev e di App. Questa scelta è stata fatta per evitare di dover creare due regole che differiscono solamente per gli indirizzi scambiati. In questo modo si fa leva sul ruolo e non sulla posizione degli indirizzi nel frame. Sarà quindi

l'SCHC C/D a preoccuparsi di selezionare il campo corretto in base al fatto che ci si trovi in una comunicazione in upstream o in downstream.

Rule 0

Field	FP	DI	Value	Match Opera.	Comp Decomp Action	Sent [bits]
IPv6 version	1	Bi	6	equal	not-sent	
IPv6 DiffServ	1	Bi	0	equal	not-sent	
IPv6 Flow Label	1	Bi	0	equal	not-sent	
IPv6 Length	1	Bi		ignore	comp-length	
IPv6 Next Header	1	Bi	17	equal	not-sent	
IPv6 Hop Limit	1	Bi	255	ignore	not-sent	
IPv6 DEVprefix	1	Bi	FE80::/64	equal	not-sent	
IPv6 DEViid	1	Bi		ignore	DEViid	
IPv6 APPprefix	1	Bi	FE80::/64	equal	not-sent	
IPv6 APPiid	1	Bi	::1	equal	not-sent	
UDP DEVport	1	Bi	123	equal	not-sent	
UDP APPport	1	Bi	124	equal	not-sent	
UDP Length	1	Bi		ignore	comp-length	
UDP checksum	1	Bi		ignore	comp-chk	

Figura 3.5: Esempio di regola di SCHC [5].

3.2.3 Matching Operator

Il campo Matching Operators (MO) indica quale operatore verrà utilizzato dall'SCHC C/D durante la compressione per verificare il valore del campo nell'header rispetto a quanto presente nel Target Value (TV) per la selezione della Rule ID corretta. Questi operatori possono essere applicati a diversi tipi di dati come: interi, stringhe e strutture dati più complesse fornendo sempre come risultato i valore Vero o Falso. I Matching Operator previsti al momento sono:

- *Equal:* il valore del campo deve corrispondere a quello del Target Value.
- *Ignore:* nessun controllo viene fatto tra il valore del campo e quello del Target Value. Il risultato è quindi sempre Vero.

- *MSB(length):* il risultato è Vero se i bit più significativi del campo length dell'header sono uguali a quelli del TV. L'operato ha bisogno di un parametro ulteriore che definisca il numero di bit da prendere in considerazione.

- *Match-mapping:* il risultato è positivo quando il valore del campo corrisponde a uno dei valori del Target Value. Questo operatore è utilizzato quando il TV contiene una lista di valori, e ciascuno di esse è identificato da un piccolo ID. Questo operatore è utilizzato assieme all'azione di mapping-sent per inviare al posto del valore del Target Value un indice risparmiando in questo modo dello spazio.

3.2.4 Azioni di compressione/decompressione

Action	Compression	Decompression
not-sent	elided	usa valori salvati nel ctxt
value-sent	send	usa il valore ricevuto
mapping-sent	send index	valore da indice tabella
LSB(length)	send LSB	TV o valore ricevuto
compute-length	elided	calcola lunghezza
compute-checksum	elided	calcola UDP checksum
Deviid	elided	usa IID da L2 Dev addr
Appiid	elided	usa IID da L2 App addr

Figura 3.6: Azioni di compressione e decompressione.

Una volta che una regola è stata selezionata, quindi tutti i Matching Operator hanno dato un esito positivo, si passa poi a quelle che sono le operazioni di compressione e decompressione. In figura 3.6 sono rappresentate le azioni che si possono ritrovare nel campo Action, con la descrizione di ciò che va fatto nel caso in cui il pacchetto sia nella fase si compressione oppure di decompressione. Le possibili azioni sono le seguenti:

- *Not-sent:* indica che nella fase di compressione il valore del campo va semplicemente eliminato dall'header compresso, in quanto il valore è già conosciuto dai soggetti in comunicazione e si ritrova nel TV della regola. In fase di decompressione basterà

semplicemente ricavarlo da esso. L'utilizzo di questa azione è solitamente associato all'operatore equal, il quale offrirà la certezza che il dato nella regola, e che quindi verrà recuperato dal destinatario, coincide con quanto inviato dal mittente. Viceversa, l'utilizzo per esempio del MO ignore potrebbe causare la mancata coerenza tra i dati trasmessi e ricevuti.

- *Value-sent:* con questa azione si indica che il valore contenuto nel campo va inserito nell'header compresso, e al contrario di not-sent il MO da utilizzare è solitamente ignore. Il decompressore dovrà quindi utilizzare il valore ricevuto. Il valore della dimensione del dato deve però essere conosciuta sia dal compressore che dal decompressore. Nel caso in cui la size sia fissa si suppone come essa sia già conosciuta, nel caso contrario la dimensione va indicata esplicitamente inserendola nel pacchetto compresso seguendo le regole descritte successivamente.

- *Mapping-sent:* è utilizzato per inviare un valore di indice che si riferisce ad uno dei valori presenti nel TV descritti sotto forma di lista. Come anticipato in precedenza questa azione ha come MO di riferimento match-mapping il quale risponde con esito positivo solamente se uno dei valori del TV corrisponde a quanto presente nel campo. In questo modo non si spreca spazio e si manda solamente un indice. Il decompressore una volta conosciuto l'indice può recuperare il valore dal TV della regola. Il numero di bit utilizzati sarà il numero di bit necessario per codificare i possibili indici. In figura 3.7 è possibile vedere un esempio di regola in cui è utilizzato il mapping-sent. Il campo DEVPrefix ha due valori possibili, quindi i bit inviati necessari per codificare le due alternative sono solamente uno, invece nel caso di APPprefix in cui le alternative sono tre sono necessari 2 bit, i quali sarebbero stati sufficienti anche nel caso in cui le scelte fossero state quattro. Il numero di bit necessari è quel valore che posto ad esponente di due è uguale o maggiore al numero di scelte da codificare.

Rule 1

Field	FP	DI	Value	Match Opera.	Comp Decomp Action	Sent [bits]
----	-	-	---	---	---	---
IPv6 DEVprefix	1	Bi	[alpha/64,	match-	mapping-sent	[1]
	1	Bi	fe80::/64]	mapping		
IPv6 DEViid	1	Bi		ignore	DEViid	
IPv6 APPprefix	1	Bi	[beta/64,	match-	mapping-sent	[2]
			alpha/64,	mapping		
			fe80::64]			
IPv6 APPiid	1	Bi	::1000	equal	not-sent	
----	-	-	---	---	---	---

Figura 3.7: Estratto regola con mapping-sent [5].

- *LSB:* l'azione Least Significant Bits serve per evitare di mandare l'intero contenuto di un campo del pacchetto quando una parte è già conosciuta a priori. Questa azione è utilizzata insieme all'operatore di MSB, il quale si occupa di verificare che i bit più significativi coincidano. Nell'esempio in figura 3.8 l'azione viene utilizzata per i campo UDP DEVport e APPport. Ricordiamo che il numero delle porte sono 65536, quindi per codificarle vi è bisogno di 16 bit. In questo caso vengono trasmessi solamente i 4 bit meno significati, poiché gli altri 12 bit sono recuperati dal valore nel campo TV. Nell'esempio in figura 3.8 il valore 8720, che in binario si traduce in 0010001000010000, agisce come maschera e perciò il ricostruito risultante sarà 001000100001 per i primi 12 bit e i bit trasmessi come ultimi 4 bit. Il valore finale recuperato sarà quindi compreso tra 8720 e 8735. Il decompressore combina quindi i bit del Target Value con i bit ricevuti. Il numero di bit da inviare può essere di tre tipi: fisso e descritto nella regola, come nel caso appena descritto, oppure ricavato dal campo Length a cui viene sottratto il valore espresso dall'operatore MSB, se nulla è segnalato. Infine nel caso in cui il pacchetto abbia una dimensione del campo Length variabile la dimensione residua va dichiarata in precedenza.

- *DEViid e APPiid:* queste azioni sono usate per il calcolo del Device Interface Identifier e dell'Application Interface Identifier dell'indirizzo IPv6. L'APPiid di solito non è utilizzato, invece il valore del DEViid è calcolato a partire dal Device

Rule 2

Field	FP	DI	Value	Match Opera.	Comp Decomp Action	Sent [bits]
----	-	-	---	---	---	---
UDP DEVport	1	Bi	8720	MSB(12)	LSB(4)	[4]
UDP APPport	1	Bi	8720	MSB(12)	LSB(4)	[4]
UDP Length	1	Bi		ignore	comp-length	
UDP checksum	1	Bi		ignore	comp-chk	
----	-	-	---	---	---	---

Figura 3.8: Estratto regola con LSB [5].

ID presente nell'header del frame di livello 2 secondo un metodo che dipende dalla tecnologia di rete LPWAN e che va quindi gestito a livello implementativo. Al momento della compressione l'indirizzo viene eliminato e poi nel momento della decompressione viene recuperate dal Dev-ID del livello L2.

- *Compute-legth:* calcola la lunghezza assegnata al campo. Nel caso però questa azione sia usata per il campo IPv6 length e UDP length questo calcola la rispettiva lunghezza del pacchetto.
- *Compute-checksum:* viene calcolato il checksum sui dati ricevuti, in particolare l'azione trova il suo impiego principale nel calcolo del checksum di UDP.

3.2.5 Estensione a CoAP

Il protocollo HTTP (HyperText Transfer Protocol), che è alla base dell'architettura client-server di internet, non è ottimale per le reti con risorse limitate come quelle LP-WAN. Per questo la IETF si è occupata attraverso il gruppo CoRE di sviluppare il protocollo CoAP (Constrained Application Protocol) [16] che permette se unito ad un proxy di convertire le richieste HTTP proveniente da Internet in una richiesta più leggera diretta ai dispositivi. CoAP però non è solo una versione ridotta di HTTP con un overhead ridotto ed una complessità più limitata per adattarsi ai device, ma è un protocollo web che implementa utilizzando il paradigma REST (Representational State Transfer) un set di funzionalità appositamente studiate per il mondo delle applicazione

Machine to Machine(M2M). CoAP è un fattore chiave per integrare il mondo IoT con il Web, per creare il cosiddetto Web of Things (WoT) [17]. Il paradigma REST definisce il significato di risorsa e modella le interazioni tra client e server come uno scambio di rappresentazioni di risorse, con lo scopo di realizzare un'infrastruttura di gestione delle risorse remote tramite alcune funzioni di accesso e interazione come quelle di HTTP: PUT, POST, GET, DELETE.

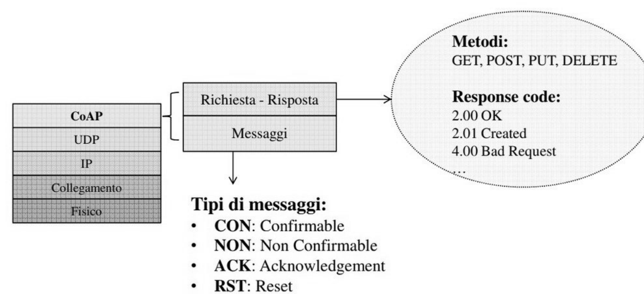


Figura 3.9: Struttura del protocollo CoAP.

Le caratteristiche chiave di questo protocollo sono le seguenti:

- Protocollo web per nodi di rete con risorse limitate;
- Utilizzo del protocollo UDP per il livello di trasporto con gestione opzionale dell'affidabilità;
- Gestione asincrona dei messaggi;
- Utilizzo di un header limitato e di un parsing semplificato;
- Gestione delle risorse attraverso un Uniform Resource Identifier (URI);
- Mapping semplice, attraverso un proxy, per permettere ai nodi HTTP l'accesso a risorse CoAP e viceversa;
- Uso della memorizzazione in cache per la gestione rapida delle risposte.

Il protocollo CoAP, come rappresentato in figura 3.9, si compone di una parte di messaggistica, che riguarda la gestione dello scambio dei messaggi che avvengono in

maniera asincrona utilizzando UDP, ed una seconda parte di interazione request-replay, che utilizza dei metodi appositi per formulare delle richieste e dei codici come risposta.

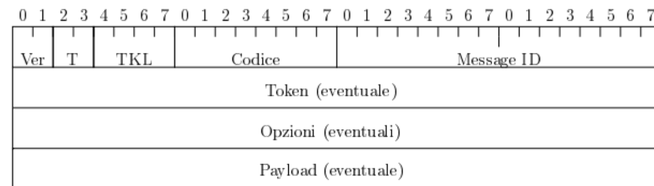


Figura 3.10: Struttura messaggio protocollo CoAP.

La struttura di un pacchetto è rappresentato in figura 3.10 e si compone di un header di soli 4 byte a cui poi si possono aggiungere una serie di altri parametri. I primi due bit indicano la versione in uso del protocollo. I successivi due bit dichiarano il tipo di pacchetto che può essere di 4 tipi: Confirmable (CON), che prevede che il destinatario mandi un ACK per confermare l'avvenuta ricezione, Non Confirmable (NON) per il viceversa, Acknowledge (ACK), ed infine Reset (RST) che indica al mittente che il server non è riuscito a elaborare il pacchetto ricevuto. I 4 bit seguenti compongono l'Option Count, che indica il numero di opzioni che il pacchetto contiene dopo l'header. Gli ulteriori 8 bit formano il Code. I valori da 1 a 31 sono riservati per i tipi di richiesta, i valori da 64 a 191 per i codici di risposta. Infine gli ultimi 16 bit formano il message ID che è un numero che consente di associare correttamente un ACK al pacchetto a cui si riferisce e per identificare i duplicati. Le altre parti, di dimensioni variabili, sono poi il token, le opzioni e il payload.

In base al valore del campo Code nell'header del pacchetto il messaggio può essere di richiesta oppure di risposta. Per quanto riguarda le richieste, come detto in precedenza CoAP utilizza il paradigma REST ed implementa 4 tipologie: GET segnalato dal codice 1, PUT con il valore 2, POST con 3 e DELETE con 4. Per quanto riguarda invece i codici di risposta, alcuni di essi sono rappresentati in tabella 3.2 dove sono riportate anche le analogie con il protocollo HTTP.

La tecnica di SCHC per la compressione dell'header si differenzia dagli schemi come 6LOWPAN HC1/HC2 o IPHC/NHC, pensate per le reti IEEE 802.15.4, oltre che per

Code	Descrizione	Corrispondente HTTP
64	2.00 OK	242 200 OK
65	2.01 Created	201 Created
66	2.02 Deleted	204 No Content
67	2.03 Valid	203 Not Modified
68	2.04 Changed	204 No Content
128	4.00 Bad Request	400 Bad Request
129	4.01 Unauthorized	400 Bad Request
130	4.02 Bad Option	400 Bad Request
131	4.03 Forbidded	403 Forbidden
132	4.04 Not Found	404 Not Found
133	4.05 Method Not Allowed	405 Method Not Allowed

Tabella 3.2: Descrizione dei codici di risposta CoAP e corrispondenze con HTTP.

la maggiore capacità di soddisfare i vincoli delle reti LPWAN anche e soprattutto per la capacità di gestire gli header dei protocolli di livello applicativo come CoAP. Quest'ultimo può essere compresso, tenendo però conto di una serie di differenze importanti, che rendono l'operazione più complessa rispetto a quando accadeva per IPv6/UDP, che sono:

1. Una lista flessibile di campi (token e opzioni) e delle loro dimensioni,

2. Asimmetria tra gli header dei messaggi di Request e di Response che contengono campi differenti,

3. Gli oggetti possono agire da client e da server,

4. Livelli differenti di acknowledgement.

Tuttavia adottando una serie di stratagemmi, come descritto nel documento sviluppato [18] da IETF per la compressione di CoAP con SCHC, è possibile riuscire a superare queste criticità e a formare un set di regole, come quella di esempio in figura 3.11, ma che non saranno in questo paragrafo approfondite in maniera ulteriore.

rule id 1

Field	TV	MO	CDF	dir	Sent
CoAP version	01	equal	not-sent	bi	
CoAP Type	CON	equal	not-sent	dw	
CoAP Type	ACK	equal	not-sent	up	
CoAP TKL	0	equal	not-sent	bi	
CoAP Code	ML2	match-map	mapping-sent	dw	CCCC C
CoAP Code	ML3	match-map	mapping-sent	up	CCCC C
CoAP MID	0000	MSB(5)	LSB(11)	bi	M-ID
CoAP Uri-Path	path	equal 1	not-sent	dw	

ML1 = {CON : 0, ACK:1} ML2 = {POST:0, 2.04:1, 0.00:3}

Figura 3.11: Esempio di regola SCHC per la compressione del protocollo CoAP [16].

3.2.6 Layered SCHC

Nelle sezioni precedenti è stato descritto come SCHC riesca a gestire tutto lo stack IoT, permettendo dunque di ridurre l'overhead dell'header creato dai diversi strati, 40 ottetti per IPv6, 8 ottetti per UDP e 4 ottetti di CoAP, trasformando il tutto in una Rule ID di pochi bit perfetta per le limitate dimensioni del frame delle reti LPWAN come LoRa. In questo modo viene aumentata la velocità di trasmissione dei pacchetti. Dall'altra parte però non viene tenuta in considerazione la crescente richiesta di spazio necessaria a gestire tutte queste regole del contesto che dovranno essere immagazzinate dai device, che soffrono anch'essi di risorse limitate. Da questa esigenza nasce una versione ottimizzata dello schema di compressione, chiamata Layered SCHC che sfruttando un contesto diviso in strati permette di salvaguardare la memoria dei dispositivi andando ad aggiungere maggiore flessibilità allo schema di compressione [19]. La soluzione nasce dal fatto che SCHC utilizza un solo contesto in cui vengono salvate tutte le regole, ciascuna delle quali riguarda i differenti strati dello stack di rete. Questa struttura però induce facilmente a molte situazioni in cui più regole condividono gli stessi valori per la parte riguardante la parte di IPv6 e magari si differenziano solo per la parte di UDP oppure viceversa. Questa situazione è mostrata dall'esempio in figura 3.12, dove due regole si differenziano solamente per la porta UDP utilizzata. Layered SCHC (LSCHC) propone di adottare più

Rule one

Field ID	Pos	Dir	Target Value	Matching Operator	C/D Function
IPv6 V	0	B	6	equal	not-sent
IPv6 TF	0	B	0	equal	not-sent
IPv6 FL	0	B	0	equal	not-sent
IPv6 L	0	B		ignore	comp-legth
IPv6 NH	0	B	17	Equal	not-sent
IPv6 HL	0	B	255	equal	not-sent
IPv6 S Prefix	0	U	Alpha::/64	equal	not-sent
IPv6 S IID	0	U		Ignore	DEViid-DID
IPv6 D Prefix	0	U	Beta::/64	equal	not-sent
IPv6 D Prefix	0	U	::1000	equal	not-sent
UDP S Port	0	B	5683	equal	not-sent
UDP D Port	0	B	5683	equal	not-sent
UDP L	0	B		ignore	comp-length
UDP C	0	B		ignore	comp-check

Rule two

Field ID	Pos	Dir	Target Value	Matching Operator	C/D Function
IPv6 V	0	B	6	equal	not-sent
IPv6 TF	0	B	0	equal	not-sent
IPv6 FL	0	B	0	equal	not-sent
IPv6 L	0	B		ignore	comp-legth
IPv6 NH	0	B	17	Equal	not-sent
IPv6 HL	0	B	255	equal	not-sent
IPv6 S Prefix	0	U	Alpha::/64	equal	not-sent
IPv6 S IID	0	U		Ignore	DEViid-DID
IPv6 D Prefix	0	U	Beta::/64	equal	not-sent
IPv6 D Prefix	0	U	::1000	equal	not-sent
UDP S Port	0	B	5230	equal	not-sent
UDP D Port	0	B	5230	equal	not-sent
UDP L	0	B		ignore	comp-length
UDP C	0	B		ignore	comp-check

Figura 3.12: Due regole che condividono lo stesso contenuto nei campi IPv6 in SCHC [19].

contesti, rispetto al singolo usato da SCHC, in cui ciascuno di essi contiene le regole per un determinato strato dello stack. Vi è dunque un contesto con le regole che gestiscono la compressione del livello di rete (NLC), uno per quello di trasporto (TLC) ed infine un altro per quello applicativo (APL). In questa maniera si ottiene una soluzione come quella in figura 3.13 in cui la Rule ID finale è composta di tre segmenti che sono il frutto della composizione delle Rule ID provenienti dai tre contesti. La dimensione dei tre segmenti dipenderà dal numero di regole di ciascun contesto. Quindi in riferimento al problema da cui si era partiti, descritto in figura 3.12, si potrà ora utilizzare una singola regola presa dal contesto NLC e due regole dal TLC. In questo modo si potrà descrivere il primo flusso con la Rule ID: ALC=0, TLC=1, NLC =1, mentre il secondo con la Rule ID: ALC=0, TLC=2, NLC=1.

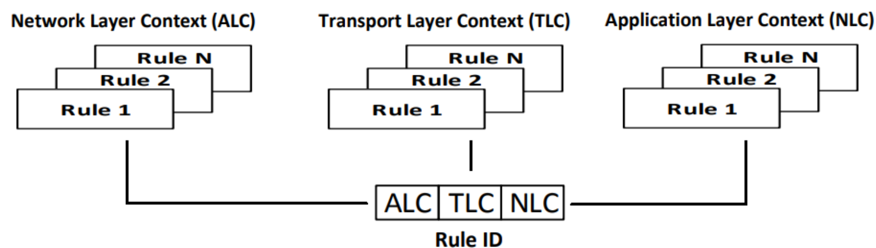


Figura 3.13: Rappresentazione dei contesti multipli del Layered SCHC C/D e della formazione della Rule ID [19].

3.3 Frammentazione

Il supporto alla frammentazione si rende necessario quando la tecnologia LPWAN adottata non è in grado di soddisfare i requisiti in termini di MTU richiesti dal protocollo che si vuole utilizzare. Una volta che è stata applicata la compressione dell'header attraverso SCHC ed il risultato ha una dimensione superiore al payload del livello L2 il pacchetto da inviare può essere frammentato utilizzando la strategia descritta in seguito. Come visto nei capitoli precedenti per LoRa, queste tecnologie LPWAN hanno delle limitazioni sul traffico, e perciò la ritrasmissione dei singoli pacchetti andati persi va considerata in maniera opzionale a secondo dell'interesse ad avere un certo grado di affidabilità della rete oppure si sia focalizzati su considerazioni di tipo energetico. E' inoltre importante ricordare come in questo tipo di reti con topologia a stella non avvenga il riordino dei pacchetti ed è proprio grazie a questa assunzione di base che si riduce la complessità del meccanismo che altrimenti soffrirebbe di un ulteriore overhead.

3.3.1 Livelli di affidabilità

La frammentazione permette di definire tre diversi livelli di affidabilità sulla consegna dei frammenti che sono descritte in seguito. La stessa scelta deve essere adottata per ciascun frammento appartenente ad uno stesso pacchetto. La scelta del determinato grado di affidabilità e del particolare utilizzo per specifici pacchetti è una decisione che va presa al momento dell'implementazione in base alle esigenze ricercate. Le opzioni disponibili sono:

1. *No ACK:* il ricevente quando ricevi i frammenti di un pacchetto non deve mandare alcun messaggio di ACK al mittente.

2. *Window mode - ACK "always":* dopo che una finestra di frammenti è stata spedita il ricevente manda un ACK al mittente. Con il termine finestra di frammenti si intende un sottoinsieme di tutti quei frammenti necessari a trasmettere il pacchetto.

L'ACK conferma la corretta ricezione dei frammenti oppure informa il mittente dei frammenti mancanti che dovranno quindi essere ritrasmessi. Nel caso in cui il mittente non riceva alcun ACK questo ritrasmetterà un frammento che avrà la funzione di richiedere un ACK. Questa procedura di richiesta potrà essere reiterata fino ad un valore, scelto a livello implementativo, denominato MAX_ACK_REQUEST.

3. *Window mode - ACK "on error":* diversamente dal caso precedente, un ACK viene trasmesso dal ricevente dopo una finestra solamente nel caso sia stato perso almeno un pacchetto al suo interno. In questo modo la segnalazione al mittente è fatta solamente in caso di frammenti mancanti. Questo a sua volta si dovrà occupare di ritrasmettere i frammenti andati persi. Il numero di volte che un ricevente può mandare un ACK per una singola finestra è settato a livello implementativo dal parametro MAX_ACKS_PER_WINDOW.

La modalità *No ACK* è quella che, dal punto di vista dell'efficienza, non appesantisce la comunicazione, ma d'altra parte non offre nessun grado di affidabilità. La terza modalità invece cerca di mantenere leggera la comunicazione puntando sul fatto che le perdite saranno moderate e perciò anche il numero di ACK non sarà elevato. Questa modalità inoltre si configura positivamente in quei casi in cui la tecnologia LPWAN utilizzata è asimmetrica, ossia offre un'ampia capacità in termini di uplink ma non altrettanto in downlink. Viceversa la modalità *ACK always*, a scapito di un grosso overhead causato dal continuo invio di ACK, offre un controllo della rete di un livello nettamente superiore.

3.3.2 Terminologia e formati dei pacchetti

Un frammento di un pacchetto, come visibile in figura 3.14 si compone di due parti: l'header e il payload. Quest'ultimo non sarà altro che una parte del pacchetto compresso. Ricordiamo che il frammento a sua volta rappresenta il payload del frame di livello L2.

L'header dei frammenti e degli ACK si compone, a seconda della modalità di frammentazione scelta tra quelle presentate in precedenza, di diverse campi che sono quelli

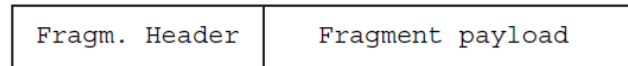


Figura 3.14: Struttura di un frammento.

descritte in seguito:

- *Rule ID:* è utilizzato sia dai frammenti che per gli ACK ed il suo valore serve ad distinguere questi pacchetti da quelli non frammentati. Può inoltre indicare quale livello di affidabilità è utilizzato e le dimensioni delle window size nel caso ne siano supportate più di una. A livello implementativo si dovrà quindi scegliere un certo set di ID sufficientemente largo per rappresentare le diverse possibilità previste.

- *Window bit (W):* è utilizzato quando viene scelta una delle frammentazione di tipo Window mode, e il suo valore serve ad identificare un gruppo di frammenti appartenenti ad una stessa finestra mantenendone un costante valore del Window bit.

- *Fragment Compressed Number (FCN):* è incluso in tutti i frammenti e rappresenta una parziale numerazione identificativa del pacchetto. All'interno di una Window mode esso è utilizzato in associazione al Window bit descritto in precedenza. Un valore speciale è utilizzato per segnalare l'ultimo frammento di un pacchetto IP.

- *Datagram Tag (DTag):* se presente, serve a identificare i frammenti di uno stesso pacchetto IP, marcandoli con un ugual valore.

- *Message Integrity Check (MIC):* viene calcolato dal mittente sull'intero pacchetto IP prima della sua frammentazione e serve poi al ricevente per verificare l'integrità del messaggio ricomposto.

- *Bitmap:* è una sequenza di bit inclusi in un ACK, per una determinata finestra, che servono a segnalare al mittente se tutti i frammenti sono stati correttamente ricevuti dal ricevente o quali sono andati persi.

Quando ci si trova nella modalità chiamata No ACK ciascun header di un frammento è composto dai campi visibili in figura 3.15, in particolare avviene la distinzione tra l'ultimo frammento inviato e tutti gli altri. I primi si compongono di R bit che comprendono la Rule ID, il Datagram Tag (DTag) composto da T bit e il Fragment Compressed Number (FCN) di N bit. La quantità effettiva di bit dipenderà poi dalla particolare implementazione. All'ultimo frammento trasmesso si aggiungono M bit che riguardano il Message Integrity Check (MIC) e gli N bit dell'FCN avranno il particolare valore scelto per segnalare che il frammento considerato è l'ultimo.

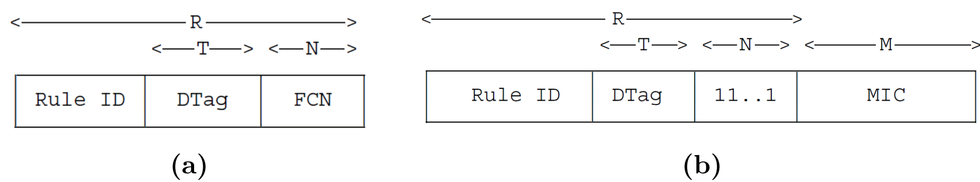


Figura 3.15: Rappresentazione della struttura dell'header di un frammento nella modalità No ACK. Con (a) raffigurante tutti i frammenti eccetto l'ultimo descritto dalla figura (b).

Quando invece si è in una delle modalità Window l'header dei frammenti assume una configurazione leggermente differente, in quanto viene introdotto il Window bit come rappresentato in figura 3.16. Anche in questo caso l'ultimo frammento differisce dagli altri e vengono aggiunti gli M bit del MIC del pacchetto IP e gli N bit assumono il valore di riferimento per segnalare il messaggio.

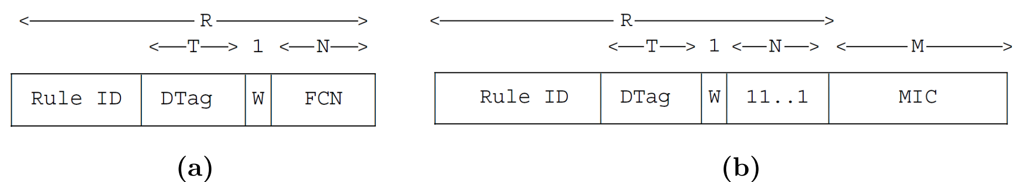
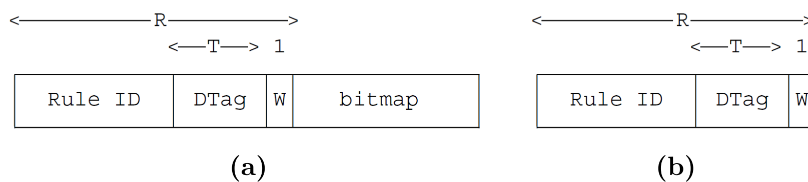


Figura 3.16: Rappresentazione della struttura dell'header di un frammento nella modalità Window. Con (a) raffigurante tutti i frammenti eccetto l'ultimo descritto dalla figura (b).

Riguardo ai valori assunti dai vari campi dell'header, indifferentemente dalla modalità in cui ci troviamo il DTag, che potrebbe anche non essere presente e che serve ad identificare i frammenti di uno stesso pacchetto IP, dovrà essere settato in maniera in-

crementale da 0 fino a al valore 2^T-1. Il campo Fragment Compressed Number (FCN) che invece è sempre presente, nella modalità No ACK sarà composto da un solo bit, il quale avrà il valore 1 nel caso si tratti nell'ultimo frammento e il valore 0 per tutti gli altri. Invece nelle varie modalità Window esso dovrà utilizzare almeno tre bit e per quanto riguarda il valore, il primo frammento dovrà partire dal valore massimo, chiamato MAX_WIND_FCN, che dovrà essere minore di 2^N-1, decrementare fino al valore 0, per poi di nuovo cominciare dal MAX_WIND_FCN e riscendere fino a quando non si è giunti all'ultimo frammento il quale sarà denotato dal valore pari a 2^N-1 ossia tutti i bit saranno posti ad 1. Il Window bit infine, composto da un solo bit, segue una regola molto semplice, inizia con il valore 1 con la prima finestra di un nuovo pacchetto IP e poi si alterna con il valore 0 nelle successive.

I messaggi di ACK, che sono utilizzati solo nelle modalità Window, si compongono come descritto in figura 3.17 dei campi Rule ID, Datagram Tag e del Window bit in uno spazio di R bit, a cui si può aggiungere il campo di bitmap quando l'ACK invia al mittente la mappa dei frammenti trasmessi correttamente, oppure andati persi o errati.



(a) **(b)**

Figura 3.17: Rappresentazione della struttura di un messaggio di ACK. Con (a) raffigurante un ACK con bitmap e (b) senza.

Il campo DTag, trasporta lo stesso valore del DTag presente nei frammenti del pacchetto IP per cui l'ACK è richiesto e si compone di T bit. Il Window bit, in quanto gli ACK come spiegato in precedenza sono utilizzati solo nelle modalità Window, sarà sempre presente e assumerà il valore dei frammenti della finestra a cui si riferisce l'ACK. Infine la bitmap, eventualmente presente, si compone del numero di ottetti necessario a contenere il numero dei frammenti della finestra. In questo modo avremo che l'n-esimo bit della sequenza avrà un valore pari ad 1 se l'n-esimo frammento della finestra è stato

correttamente ricevuto, 0 altrimenti. I rimanenti bit per terminare l'ottetto assumeranno il valore 0 a parte l'ultimo il quale posto a 1 indicherà che l'ultimo frammento è stato correttamente ricevuto oppure 0 altrimenti. Un esempio di ACK con il rispettivo bitmap, in una comunicazione in modalità Window in cui i frammenti hanno un numero N=3 di bit per il campo Fragment Compressed Number (FCN), è mostrato in figura 3.18.

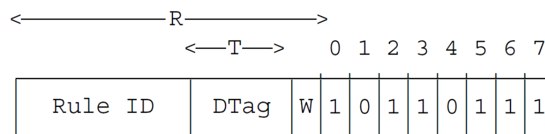


Figura 3.18: Esempio di ACK con bitmap in modalità Window con N=3. [5]

Quando la bitmap non è presente nell'ACK significa che tutti i frammenti sono stati ricevuti correttamente. Bisogna comunque prestare attenzione ai limiti imposti dal payload del livello L2, e il rapporto tra la grandezza della bitmap e le caratteristiche del Fragment Compressed Number (FCN) nell'header dei pacchetti, infatti se la bitmap ha per esempio una dimensione inferiore a 2^N, con N il numero di bit del campo FCN dei frammenti, la dimensione della window in uso dovrà essere anch'essa minore di 2^N-1 di frammenti, quindi se la dimensione a disposizione della bitmap fosse 60 bit, N dovrà essere settato a 6, la window size potrà essere settata ad un massimo di 59 bit e quindi la MAX_WIND_FCN avrà valore 59.

3.3.3 Funzionamento della frammentazione

In questa sezione vediamo come si compone la trasmissione di un pacchetto IP frammentato tra un nodo mittente e un ricevitore. Il ricevitore per identificare correttamente i frammenti che appartengono ad un determinato pacchetto può sfruttare le informazioni fornite dall'indirizzo L2 del mittente, dall'indirizzo L2 del destinatario, dalla Rule ID e dal DTag nei frammenti. Inoltre, se la Rule ID è stata codificata seguendo una certa regola sarà possibile determinare anche il livello di affidabilità adottata dalla comuni-

cazione. Nel diagramma in figura 3.15 è raffigurata come avviene la trasmissione di un pacchetto suddiviso in 11 frammenti e che adotta la modalità No ACK. In questo caso, trattandosi nella modalità più semplice abbiamo che il mittente spedisce 11 frammenti che hanno come unica distinzione il Fragment Compressed Number (FCN) che per i primi dieci frammenti sarà pari a 0, e nell'ultimo avrà valore 1 per segnalare che si tratta dell'ultimo frammento del pacchetto IP. Arrivati a quel punto, il ricevitore ricostruirà il pacchetto seguendo l'ordine di ricezione dei frammenti e verificherà attraverso il Message Integrity Check (MIC) di aver ricevuto correttamente tutti i frammenti e di aver ricostruito correttamente il pacchetto IP.

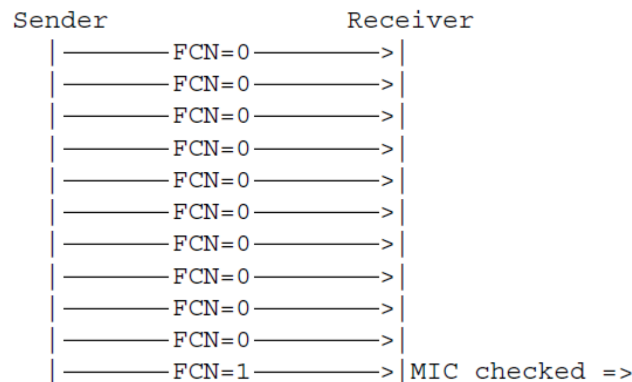


Figura 3.19: Diagramma della trasmissione di un pacchetto IP composto da 11 frammenti in modalità No ACK [5].

Nel caso in cui il pacchetto debba essere trasferito attraverso la Window mode - ACK "on error" la procedura seguita si compone di una serie di passaggi più complessi. Quando al ricevitore arriva il primo frammento di un nuovo pacchetto IP, esso fa partire un timer chiamato ACK on Error Timer il quale tuttavia riparte ogni qualvolta viene ricevuto un nuovo frammento dello stesso pacchetto e si stoppa solamente una volta che l'ultimo frammento dell'ultima finestra è stato ricevuto. Viceversa se un frammento non viene ricevuto entro il tempo definito a livello di implementazione dal timer, il ricevitore fa partire un ACK in cui segnala al mittente attraverso la bitmap quali sono i frammenti della finestra correttamente ricevuti e quali no che dovranno essere per questi rimandati. Un comportamento analogo avviene, come rappresentato in figura 3.20b nel

caso in cui dopo che l'ultimo frammento di una finestra, marcato con FCN=0 oppure con FCN=2^N-1 se la finestra è l'ultima, è stato ricevuto ma alcuni di essi sono stati persi o nel caso in cui il MIC abbia riscontrato errori. Nel caso in cui i frammenti di tutta la finestra siano stati correttamente ricevuti, come nel diagramma in figura 3.20a, nessun messaggio di ACK è mandato al mittente. Quest'ultimo continuerà ininterrottamente a trasmettere i frammenti delle finestre successive, fino a trasmettere l'ultimo frammento identificato da FCN=2^N-1, ossia con tutti i bit settati ad 1. Dopodiché, se anche il MIC non dovesse riscontrare problemi, il sistema prosegue con i pacchetti successivi. Ricordiamo infine come il primo frammento di ogni finestra debba partire sempre da un FCN =MAX_WIND_FCN.

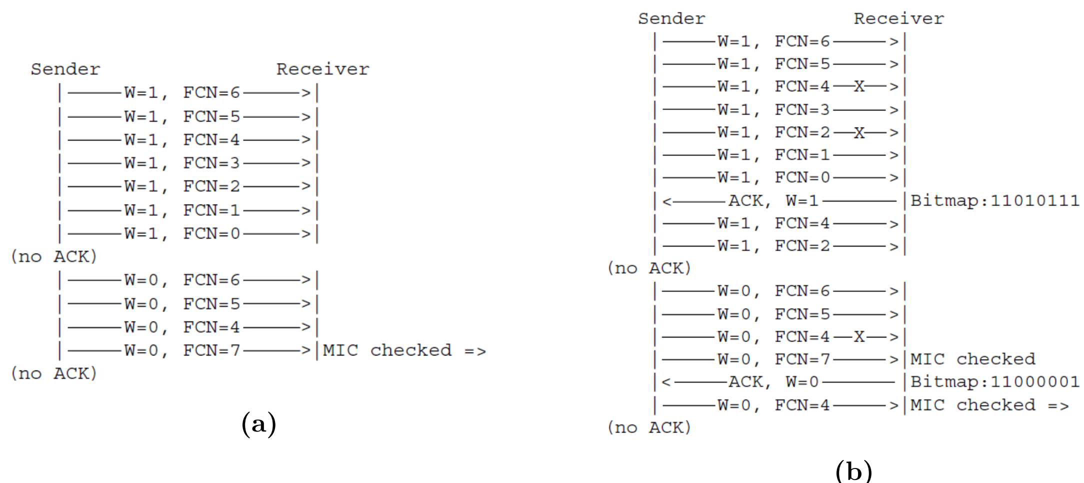


Figura 3.20: Diagramma della trasmissione di un pacchetto IP composto da 11 frammenti in modalità Window - "on error". In figura (a) caso con N=3, MAX_WIND_FCN=6 e senza perdite di frammenti. In (b) caso analogo ma con tre frammenti persi. [5]

La trasmissione in modalità ACK "always" fa uso anch'essa di un timer chiamato "ACK Always Timer" che appartiene però al sender dei frammenti. Questo timer si attiva al primo tentativo di trasmissione dell'ultimo frammento della finestra, ossia di quel frammento marcato da FCN=0 o FCN=2^N-1, e si riattiva dopo la ritrasmissione del frammento con più basso FCN richiesto dal ricevitore in seguito ad un ACK. Allo scadere del timer, ossia passato un certo lasso di tempo, deciso a priori, senza che il sender abbia più ricevuto un ACK, se ci si trova in una delle finestre che non siano l'ultima, il mittente

rispedisce l'ultimo frammento inviato al momento dello scadere del timer, viceversa se il sistema si trova nell'ultima finestra il mittente invia di nuovo il frammento con FCN=2^N-1 per cercare di ottenere un qualche ACK dal ricevitore. Nel diagramma in figura 3.21a vediamo una trasmissione ACK "always" in cui non avvengono perdite di pacchetti, e che prevede l'invio di un ACK senza bitmap alla fine della ricezione della prima finestra di frammenti e dopo la verifica del MIC. Nel caso in figura 3.21b vediamo invece come in caso di mancata ricezione di un frammento, analogamente a quanto succedeva per la modalità ACK "on error", il ricevitore invia un ACK con allegata la bitmap in cui vengono segnalati i frammenti mancati con bit settato a 0 nella posizione di riferimento e che dovranno poi essere rispediti. Quando poi i frammenti sono stati correttamente ricevuti ed il MIC ha dato esito positivo viene spedito l'ACK finale di conferma senza bitmap.

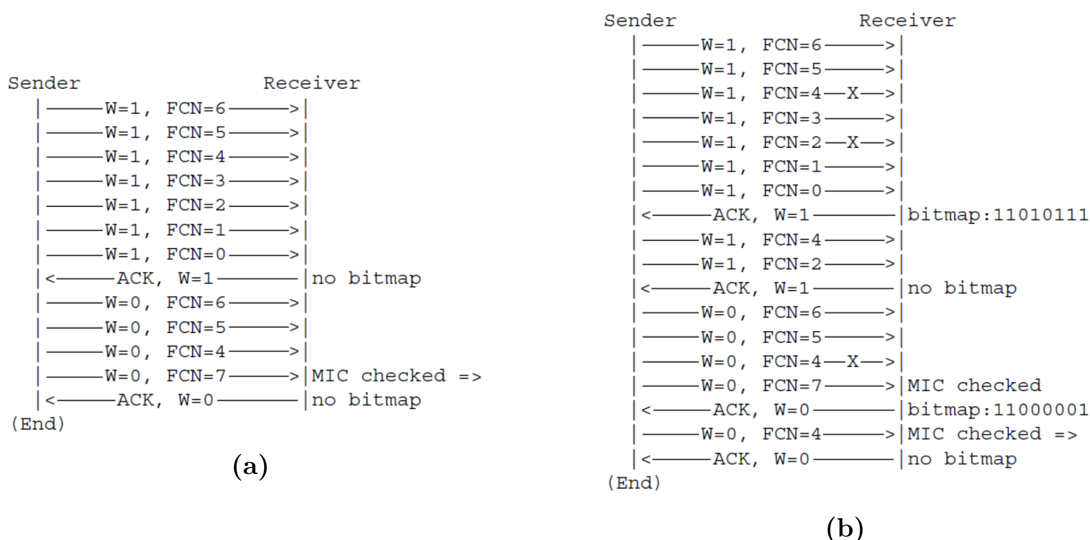


Figura 3.21: Diagramma della trasmissione di un pacchetto IP composto da 11 frammenti in modalità Window - "always". In figura (a) caso con N=3, MAX_WIND_FCN=6 e senza perdite di frammenti. In (b) caso analogo ma con tre frammenti persi. [5]

Dal punto di vista della salvaguardia delle risorse, quando un nodo ricevitore si dovesse disconnettere dalla rete, questo dovrà eliminare tutti i frammenti delle parti di pacchetto parzialmente assemblate e allo stesso modo i nodi mittenti dovranno distruggere tutti

quei frammenti di pacchetti parzialmente inviati. Inoltre come ulteriore sicurezza un ricevitore dovrà implementare un reassembly timer che tiene conto dell'inizio della ricezione del pacchetto e che passato un determinato tempo senza che il pacchetto non sia stato ricostruito elimina tutti i frammenti ricevuti fino a quel momento. Adottando queste strategie si evita che i dispositivi, che solitamente non dispongono di grosse memorie saturino lo spazio a loro disposizione.

3.3.4 Funzionalità aggiuntive

Al fine di ottimizzare le prestazioni si possono adottare alcune strategie nel momento dell'implementazione, come per esempio utilizzare window size differenti in maniera da sfruttare finestre di dimensioni elevate quando i pacchetti sono di grandi dimensioni, in modo da diminuire il numero di frammenti utilizzati, viceversa usare delle finestre più piccole in caso di pacchetti di dimensioni inferiori. In quest'ultimo modo si va ad evitare lo spreco di bit da dedicare alla bitmap nei messaggi di ACK a all'header dei frammenti per bit non utilizzati veramente. Il modo da segnalare questo supporto a finestre di dimensione multiple potrebbe essere svolto attraverso una particolare Rule ID, usata per i pacchetti che ne volessero fare uso. Una seconda possibilità, per evitare innumerevoli sprechi, potrebbe essere l'inserimento di un messaggi per segnalare la necessita di abortire la trasmissione dei frammenti rimanenti del pacchetto. Anche in questo caso la strategia potrebbe essere messa in atto attraverso l'utilizzo di una Rule ID predefinita, che consentirebbe in questo modo di terminare la trasmissioni di dati non più desiderati. Infine, considerato la natura delle tecnologie LPWAN, in cui solitamente la trasmissione in downlink si verifica solamente dopo una in uplink, come è il caso dei nodi LoRaWAN di classe A presentati nella sottosezione 2.2.2, per evitare che il downlink di frammenti impieghi tempi troppo lunghi, si potrebbe implementare un un sistema a livello L2 o superiore che inneschi una trasmissione in uplink immediata ogni qualvolta il frammento ricevuto non sia quello finale.

3.3.5 Sicurezza

La frammentazione come ogni altro protocollo non è immune a problematica dal punto di vista della sicurezza. In particolare, si possono facilmente verificare due tipologie di attacco come il buffer reservation attack e lo spoofing attack. Il primo provoca un Denial of Service(DOS), ossia un rallentamento o addirittura un blocco del servizio. Esso si verifica quando un nodo, come nell'esempio in figura 3.22, invia un primo frammento al destinatario da attaccare, il quale a questo punto riserverà un certo spazio di buffer per la ricezione degli altri frammenti del pacchetto, e per un certo intervallo di tempo prima del timeout per l'assemblaggio, i nuovi frammenti provenienti da altri pacchetti saranno scartati. L'attaccante può nuovamente mandare un primo frammento ed in questo modo continuerà a tenere occupato l'attaccato. La difficoltà nel portare a termine l'attacco sarà proporzionale al numeri di buffer di cui può disporre il destinatario. Possono inoltre essere intraprese una serie di strategie atte a rendere più difficoltoso il successo dell'attacco come per esempio permettere a frammenti di pacchetti diversi essere inseriti in uno stesso buffer di assemblaggio.

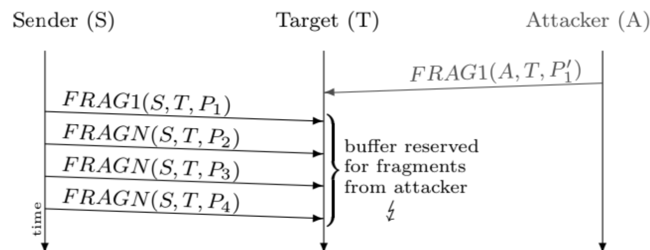


Figura 3.22: Diagramma di un buffer reservation attack. [20]

La seconda tipologia di attacco, visibile in figura 3.23, è invece composta da un nodo che può intercettare un frammento in transito sulla rete e può quindi mandare un frammento duplicato, detto spoofed, con un payload differente al destinatario, il quale ritrovandosi a quel punto con due frammenti con lo stesso header non sapendo quale di questi scegliere per la ricostruzione elimina il pacchetto che a questo punto è identificato come corrotto.

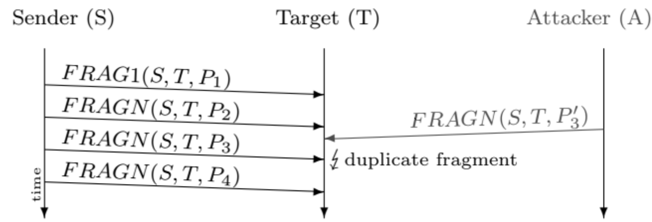


Figura 3.23: Diagramma di un duplication attack di frammenti. Il target si trova per lo stesso frammento il payload P_3 mandato dal sender e il payload $P_{3'}$ dell'attacker. [20]

Come contromisure a questo tipo di attacco si possono attivare dei meccanismi di identificazione del mittente o di conteggio dei frame qualora la tecnologia LPWAN sottostante lo preveda. Si può inoltre adottare la strategia già proposta per 6LoWPAN [20] che prevede di stabilire un binding tra i frammenti applicando uno schema chiamato content-chaining, rappresentato in figura 3.24, in cui a catena viene applicato una funzione hash crittografica sul frammento precedente.

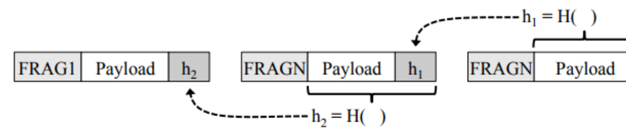


Figura 3.24: Esempio di content-chaining per un pacchetto composto da tre frammenti. [20]

4

Implementazione di SCHC

In questo capitolo viene descritto il lavoro di implementazione dell'algoritmo SCHC nella sue parti di compressione e decompressione in linguaggio C++ e i risultati che sono stati ottenuti testandolo attraverso un insieme di regole caricate nel context.

4.1 Implementazione di SCHC

L'implementazione dell'algoritmo SCHC da me svolta si è basata sull'utilizzo del linguaggio C++. Questa scelta è stata motivata dal fatto che un compilatore per questo linguaggio è solitamente presente su qualsiasi piattaforma hardware che si potrebbe scegliere per la realizzazione della propria rete LPWAN. Perciò, nonostante la grande quantità di dispositivi esistenti attualmente o futuri, questa implementazione si presta facilmente ad essere adattata e riutilizzata. Inoltre, nel panorama mondiale, questa si presenta come una delle prime implementazioni di SCHC. Il lavoro è stato strutturato in maniera modulare nelle seguenti parti:

- ***schc.h:*** contiene la descrizione dei tipi di dati creati per la modellazione delle regole del context di SCHC.
- ***comp_decomp.cpp/h:*** sono descritte le tre funzioni principali di compressione, di decompressione, e di creazione dei pacchetti finali compressi con l'eventuale bit padding.
- ***rules.cpp/h:*** sono inserite un insieme di regole di prova ed il metodo che le carica nel context di SCHC.

- ***pkt_utility.cpp/h:*** contiene una serie di metodi utili per la creazione di pacchetti IPv6/UDP, per il loro salvataggio e caricamento.

Nelle prossime pagine verrà descritta in maniera approfondita ogni parte e le scelte che sono state prese per realizzare l'algoritmo. Da segnalare inoltre l'utilizzo della libreria libtins [21] in particolare per la gestione degli indirizzi IPv6 e per la costruzione e il salvataggio di pacchetti UDP/IPv6. La libreria, molto leggera ed efficiente ha permesso di rendere ancora più facile l'eventuale utilizzo dell'implementazione in uno scenario reale in quanto essa ha consentito di interfacciarsi direttamente con il formato PCAP che è anche il formato predefinito per i software che si occupano di analizzare le reti, tra cui ad esempio il famoso Wireshark.

4.1.1 Tipi di dato

La prima cosa da fare per creare la struttura dell'algoritmo è stata quella di realizzare quei tipi di dato indispensabile alla trattazione del problema. In particolare, all'interno del file `schc.h` sono state inserite le definizioni delle regole del context di SCHC. Una regola sarà quindi un tipo di dato, creato attraverso la parola chiave `typedef` messa a disposizione dal linguaggio C++, seguita dal termine `struct` per indicare una struttura composta e si comporrà, come rappresentato nel listato 4.1, dalla Rule ID, ossia l'identificatore della regola ed un array contenente la descrizione dei vari campi, detti field. La Rule ID è espressa attraverso il tipo di dato bitset fornito dalla libreria standard e non è altro che una serie di bit di valore zero o uno, in una quantità arbitraria ma che dovrà comunque essere legata al numero di regole di cui si prevedere si avrà necessità all'interno del context.

```
typedef struct{
    std::bitset<N_RULE_ID_BIT>      rule_id;
    SCHC_Rule_field                 rule_field[N_FIELD_MAX];
}SCHC_Rule;
```

Listing 4.1: Definizione struttura di una regola del context.

Per quanto riguarda invece l'array contente i campi della regola a sua volta questi sono espressi da un tipo di dati creato ad hoc per l'algoritmo e definito come descritto nel listato 4.2. Esso si compone di quei campi già discussi nel sottocapitolo 3.2.1, quindi dal nome del campo, da un intero che rappresenta la field positon, un valore indicante la direzione della comunicazione per cui si applica il campo, un valore target che nel nostro caso può essere un valore intero, oppure un indirizzo IPv6, grazie al supporto fornito dalla libreria libtins, e addirittura un vettore di indirizzi IPv6. Questi ultimi saranno utilizzati in quei campi che lo necessitano come per esempio i campi APPprefix, DEVprefix e i corrispettivi APPiid e DEViid. Infine gli ultimi due termini riguardano il Matching Operator utilizzato dal campo e l'azione di compressione e decompressione da utilizzare.

```
typedef struct {
    SCHC_FieldName               field_name;
    int                          field_position;
    SCHC_DirectionIndicator      direction_indicator;
    int                          target_value_int;
    Tins::IPv6Address            target_value_ip;
    std::vector<Tins::IPv6Address> target_value_ip_vec;
    SCHC_MatchingOperator        mo_field;
    SCHC_CDA                     cda_field
}SCHC_Rule_field;
```

Listing 4.2: Definizione struttura campo di una regola.

I valori che può assumere il nome sono descritti attraverso il tipo `enum` messo a disposizione dal linguaggio C++ e che permette di elencare una serie di possibili scelte. La stessa modalità è stata utilizzata per il direction indicator, per il matching operator e l'azione di compressione e decompressione con tutti i campi già descritti nei capitoli in cui è stato presentato l'algoritmo.

4.1.2 Caricamento delle regole nel context

Attraverso il metodo *load_rules* descritto nel file rules.cpp vengono invece create una serie di regole e inserite all'interno del context, che a livello implementativo è strutturato come un vettore di regole. Un esempio di regola è mostrata nel listato 4.3.

```
SCHC_context[3].rule_id=3;
SCHC_context[3].rule_field[0].field_name=IPv6_Version;
SCHC_context[3].rule_field[0].field_position=1;
SCHC_context[3].rule_field[0].direction_indicator=Bi;
SCHC_context[3].rule_field[0].target_value_int=6;
SCHC_context[3].rule_field[0].mo_field=Equal;
SCHC_context[3].rule_field[0].cda_field=Not_sent;
.......................................................
SCHC_context[3].rule_field[6].field_name=IPv6_DEVprefix;
SCHC_context[3].rule_field[6].field_position=1;
SCHC_context[3].rule_field[6].direction_indicator=Bi;
SCHC_context[3].rule_field[6].target_value_ip="fe80::";
SCHC_context[3].rule_field[6].mo_field=Equal;
SCHC_context[3].rule_field[6].cda_field=Not_sent;
```

Listing 4.3: Esempio di regola.

Nell'esempio vediamo l'assegnazione del nome del campo, come la versione di IPv6 utilizzata, con il valore inserito nel target value, e le altre informazioni che completano la descrizione del campo della regola. Per questi campi vediamo che il compressore dovrà solamente verificare il corretto matching dei valori del pacchetto da comprimere con la rispettiva target value senza dover inviare alcun dato aggiuntivo al ricevitore. Vediamo che il campo DEVprefix e APPprefix possono contenere un solo indirizzo IPv6 oppure una lista di indirizzi, come invece descritto in una delle altre regole di test caricate nel context e visibili nel listato 4.4. In questo secondo caso il Matching Operator è settato nella modalità Match Mapping e quindi l'algoritmo andrà a verificare se il campo del pacchetto da comprimere abbia uno dei valori inseriti nelle target value della regola e procederà come previsto dall'azione Mapping Sent espressa dal campo CDA ad inviare il

valore, dell'indice del vettore, necessario al momento della decompressione per selezionare il valore corretto per la ricostruzione del pacchetto.

```
SCHC_context[2].rule_field[8].field_name=IPv6_APPprefix;
SCHC_context[2].rule_field[8].field_position=1;
SCHC_context[2].rule_field[8].direction_indicator=Bi;
SCHC_context[2].rule_field[8].target_value_ip_vec.push_back("ae50:2222::");
SCHC_context[2].rule_field[8].target_value_ip_vec.push_back("aa50::");
SCHC_context[2].rule_field[8].target_value_ip_vec.push_back("df34::");
SCHC_context[2].rule_field[8].target_value_ip_vec.push_back("ab50:1010::");
SCHC_context[2].rule_field[8].target_value_ip_vec.push_back("ff50::");
SCHC_context[2].rule_field[8].mo_field=Match_mapping;
SCHC_context[2].rule_field[8].cda_field=Mapping_sent;
```

Listing 4.4: Esempio di campo APPprefix con con Match Mapping.

4.1.3 Metodo per la compressione

L'algoritmo SCHC per la compressione dei pacchetti è scritto all'interno del file denominato comp_decomp.cpp in Appendice B e che segue la struttura mostrata nello pseudocodice riportato nell'Algoritmo 1. Nello pseudocodice non si è scesi nel dettaglio di ogni operazione in quanto vuole essere una traccia di quanto poi effettivamente svolto in linguaggio C++. In ingresso abbiamo il pacchetto IP da comprimere, il context contenente le regole conosciute dal sistema ed infine l'indicatore della direzione del flusso della comunicazione che può essere di tipo upstream, da un nodo verso il network server, oppure viceversa di downstream. I risultati prodotti saranno la Rule ID selezionata dall'algoritmo e l'insieme di informazioni aggiuntive richieste dalla regola da aggiungere al pacchetto compresso. Il metodo tuttavia ritorna un valore booleano, che sarà true quando una regola adatta è trovata oppure false se nessuna regola presente nel context può essere utilizzata. L'algoritmo scorrendo ad una ad una le regole presenti nel context verifica in un primo momento se tutti quei campi della regole, il cui indicatore di direzione concorda con quello in ingresso, rispondono in maniera positiva al confronto svolto

dal matching operator. In caso di esito negativo, l'algoritmo passa alla regola successiva, viceversa, significa che la regola è stata trovata e l'algoritmo comincia la compressione del pacchetto. Questo è fatto andando ad applicare per ciascun campo l'azione di compressione descritta dalla regola stessa. Il caso più fortuito si ha quando nessun ulteriore dato oltre alla Rule ID deve essere trasmesso. Tuttavia l'implementazione gestisce correttamente anche i casi più complessi in cui entra in gioco l'azione LSB, che prevede l'invio dei bit meno significati, oppure Mapping sent, che determina la spedizione dell'indice del valore corretto di target value. Queste informazioni aggiuntive andranno poi inserite nel pacchetto compresso. Una volta effettuate le operazioni di compressione l'algoritmo avrà finalmente portato a termine il suo lavoro e verrà restituito dal metodo il valore true. Nel caso in cui, come detto in precedenza, ci si sia spostati sulla regola successiva, una volta esaurite le regole il valore restituito sarà a quel punto false, senza che sia stato generato alcun dato dall'algoritmo.

La formazione vera e propria del pacchetto compresso è poi delegata al metodo *composer* che si occupa di prendere la Rule ID e le informazioni aggiuntive necessarie alla regola, generate con l'algoritmo appena descritto, e di metterle insieme in un unico pacchetto aggiungendo, in caso lo si scelga, quei bit di padding necessari a completare l'ottetto. Alla fine di tutto questo, una volta aggiunto il payload, il pacchetto compresso è finalmente pronto e può essere trasmesso.

Algorithm 1: Pseudocodice algoritmo di compressione SCHC

```
   Data: IP packet pkt, SCHC context ctx and the Direction Indicator di
   Result: Rule ID r_id and Compressed header informations comp_info
   Output: Bool found, true if a rule was found, false otherwise
 1 found ← false;
 2 mo_miss ← false;
 3 ctx_rules_num ← ctx.size();
 4 for rule ← 0 to ctx_rules_num do
 5     max_field ← ctx[rule].rule_field.size();
 6     for field ← 0 to max_field do
 7         if di == ctx[rule].rule_field[field].direction_indicator
 8         or ctx[rule].rule_field[field].direction_indicator == Bi then
 9             if ctx[rule].rule_field[field].mo_field == Equal
10             or Match_mapping
11             or MSB then
                   /* Verify, depending of which type of operator is used,
                   if the pkt target value correspond to the value stored
                   in the rule */
12                 if ctx[rule].rule_field[field].tv! = pkt.field.tv then
13                     mo_miss == true   /* Matching operator has failed */
14                 end
15             end
16         end
17     end
18     if mo_miss == false then         /* No mismatches, rule found!  */
19         found ← true;
20         r_id ← ctx[rule].rule_id;
21         for field ← 0 to max_field do
                /* Follow the compression action expressed in the rule */
22             if ctx[rule].rule_field[field].cda == Value_Sent then
23                 comp_info ← pkt.field.tv;
24             end
25             if ctx[rule].rule_field[field].cda == Mapping_sent then
26                 comp_info ← index(ctx[rule].rule_field[field].tv);
27             end
28             if ctx[rule].rule_field[field].cda == LSB then
29                 comp_info ← lsb(pkt.field.tv);
30             end
31         end
32         return found                   /* Rule found, return true */
33     end
34 end
35 return found                       /* Rule not found, return false */
```

4.1.4 Metodo per la decompressione

L'algoritmo SCHC per la decompressione è anch'esso implementato all'interno del file comp_decomp.cpp in Appendice B e che segue la struttura riportata in pseudocodice nell'Algoritmo 2. In ingresso abbiamo il pacchetto da decomprimere, il context contenente le regole conosciute dal sistema ed infine l'indicatore della direzione del flusso della comunicazione che può essere di tipo upstream, da un nodo verso il network server, oppure viceversa di downstream. Il risultato prodotto è ovviamente il pacchetto originale ricostruito. Il metodo tuttavia ritorna un valore booleano, che sarà true quando il pacchetto sarà stato correttamente decompresso, oppure false nel caso in cui non sia stato possibile. L'algoritmo fa un primo controllo per determinare se effettivamente la regola è presente nel context, dopodiché in caso di esito positivo ha inizio il processo di ricostruzione del pacchetto originario. A questo punto si vanno ad analizzare tutti quei campi descritti nella regola che concordano con la direzione del flusso di comunicazione e si seguono le indicazioni descritte dalla regola di decompressione espressa dalla CDA. In base all'azione di volta in volta verranno intraprese le operazioni necessarie atte a ricostruire tutti i campi del nostro pacchetto IP di partenza. Una volta terminato il ciclo il pacchetto è pronto ed è coerente con quanto il mittente voleva trasmettere.

Algorithm 2: Pseudocodice algoritmo di decompressione SCHC

```
    Data: Compressed packet comp_pkt, SCHC context ctx and the Direction
          Indicator di
    Result: Decompressed IP packet pkt_decomp
    Output: Bool decompressed, true if a pkt is reconstructed, false otherwise
 1  found ← false;
 2  decompressed ← false;
 3  rule ← 0;
 4  for i ← 0 to cxt.size() do
 5  |   if context[i].rule_id == comp_pkt.rule_id then
 6  |   |   rule ← i;
 7  |   |   found ← true;
 8  |   end
 9  end
10  if found == false then   /* Pkt not reconstructed, return false */
11  |   return decompressed
12  else
13  |   max_field ← context[rule].rule_field.size();
14  |   for i ← 0 to max_field do
15  |   |   if di == ctx[rule].rule_field[field].direction_indicator
16  |   |   or ctx[rule].rule_field[field].direction_indicator == Bi then
    |   |   |      /* Follow the decompression action expressed in the
    |   |   |   rule */
17  |   |   |   if ctx[rule].rule_field[field].cda == Not_Sent then
18  |   |   |   |   pkt_decomp.field ← ctx[rule].rule_field[field].tv;
19  |   |   |   end
20  |   |   |   if ctx[rule].rule_field[field].cda == Value_Sent then
21  |   |   |   |   pkt_decomp.field ← comp_pkt.field_value;
22  |   |   |   end
23  |   |   |   if ctx[rule].rule_field[field].cda == Mapping_sent then
24  |   |   |   |   field_value ←
    |   |   |   |     ctx[rule].rule_field[field].tv[comp_pkt.index];
25  |   |   |   |   pkt_decomp.field ← field_value;
26  |   |   |   end
27  |   |   |   if ctx[rule].rule_field[field].cda == LSB then
28  |   |   |   |   field_value ← comp_pkt.lsb_value
29  |   |   |   |                + ctx[rule].rule_field[field].msb_value;
30  |   |   |   |   pkt_decomp.field ← field_value;
31  |   |   |   end
32  |   |   end
33  |   end
34  |   decompressed ← true;
35  |   return decompressed          /* Pkt reconstructed, return true */
36  end
```

4.1.5 Metodi utili per la creazione e gestione dei pacchetti

Oltre ai metodi principali che eseguono l'algoritmo SCHC sono stati anche sviluppati tutta una serie di strumenti, inseriti nel file pkt_utility.cpp, per consentire la formazione e la gestione dei pacchetti, ed atti a essere utilizzati nello scenario simulativo ma che si prestano poi ad essere applicati anche in contesti reali. Il metodo più importante è quello denominato *ipv6_pkt_creator* usato per la creazione dei pacchetti IP in maniera facile partendo dagli indirizzi IP di mittente e destinatario, e dalle porte UDP di invio e ricezione. Questo metodo crea da prima il datagramma UDP con le determinate caratteristiche scelte e lo inserisce poi nel payload del pacchetto IP. A questo punto il pacchetto è coerente con le specifiche IP ed ogni campo dell'header può essere facilmente modificato attraverso un accesso diretto al campo a cui si è interessati. Questo pacchetto può essere eventualmente salvato come file PCAP in caso si abbia intenzione di riutilizzarlo successivamente attraverso il metodo *ipv6_pkt_saver*. Questo prevede in input il nome che si intende utilizzare per il file e il riferimento all'oggetto rappresentante il pacchetto. Dopodiché questi pacchetti possono essere recuperati a partire dai file salvati utilizzando il metodo *ipv6_pkt_loader* il quale richiede solamente la path del file e restituisce il pacchetto come oggetto. Questo metodo è molto interessante in quanto apre il software alla possibilità di recuperare direttamente i pacchetti da quei programmi che svolgono analisi di rete, come per esempio Wireshark, che di norma salvano il traffico intercettato proprio nel formato PCAP. Poi, il metodo *print_pkt_info* è stato fatto per ottenere in maniera semplice la stampa a video di tutti i campi dell'header del pacchetto. Infine, *ipv6_pkt_verify* permette di verificare se due pacchetti IP hanno lo stesso header, ed è quindi stato utilizzato per controllare se effettivamente il pacchetto decompresso coincidesse con quello inviato prima che subisse il processo di compressione. Grazie a tutti questi metodi è stato possibile testare ed espandere l'utilizzo dell'algoritmo SCHC.

4.2 Performance del sistema

In questa sezione vengono presentate e discusse le performance ottenute dal sistema. In figura 4.1 vediamo i livelli di compressione raggiunti attraverso l'uso dell'algoritmo SCHC in diversi scenari. Le situazioni che sono state prese in esame sono state tre, considerate le più rappresentative. Il caso espresso dalla classe di sinistra è quella in cui la regola selezionata dal compressore contiene tutti i campi settati a Not Sent, che è dunque la situazione più fortunata in quanto il pacchetto compresso si compone solamente dei bit rappresentanti la Rule ID. Dall'istogramma si vede come partendo da un pacchetto si compone di ben 384 bit, ossia 48 byte, di cui 40 dovuti all'header IPv6 e 8 dovuti all'header UDP, una volta compresso il pacchetto diventa di soli 3 bit. Questa è la dimensioni scelte nelle prove che consente di inserire nel context fino ad otto regole. Questo valore ovviamente dovrà essere scelto in base alle esigenze del sistema in cui andrà ad essere adottato l'algoritmo. Nel caso in cui poi si scelga di applicare il bit padding, il risultato aumenta di cinque bit, rimanendo tuttavia entro una dimensione veramente minima. Nella seconda classe, ci troviamo nel caso in cui la regole selezionata contiene un campo settato a Mapping Sent, cioè prevede che sia trasmesso il valore di indice necessario a identificare il corretto valore di target value. Nel nostro esempio il campo era quello dell'APPprefix, con cinque indirizzi possibili, e ciò ha reso necessario l'invio di altri tre bit per indicare il valore dell'indice, da cui si spiegano le dimensione del pacchetto compresso ottenuto di sei bit. Anche in questo caso il bit padding provoca l'aumento di dimensione fino agli otto bit. Infine, l'ultima classe a destra dell'istogramma raffigura il caso in cui due campi della regola selezionata sono settati a LSB, ossia solo un certo numero di bit significati del valore dei campi sono trasmessi. Nel nostro esempio i campi interessati sono stati l'UDP APPport e DEVPort che richiedevano l'invio di quattro bit ciascuno. Questo ha determinato un pacchetto compresso finale di undici bit, composto dai tre bit della Rule ID e gli otto bit dovuti alle due porte. Dopodiché applicando il bit padding la dimensione sale a sedici bit, rimanendo tuttavia ottima.

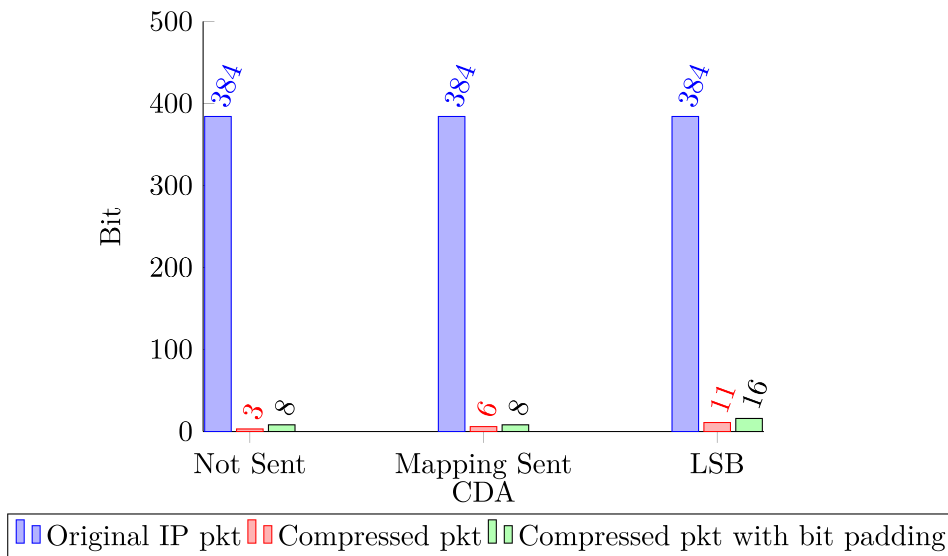


Figura 4.1: Istogramma rappresentante i livelli di compressione raggiunti nei test con differenti CDA e adottando o meno il bit padding del pacchetto compresso.

Vediamo inoltre i risultati ottenuti ragionando in termini di Space Savings, definito come $Space\,Savings = 1 - \frac{Compressed\ size}{Uncompressed\ size}$. I valori ricavati sono rappresentati nell'istogramma in figura 4.2 in termini percentuali e si attestano al di sopra del 95%. Questo dimostra ancora una volta l'impressionante efficacia dell'algoritmo SCHC.

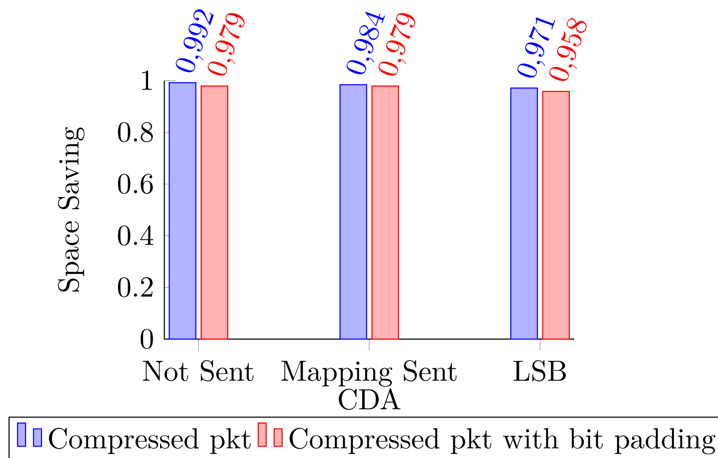


Figura 4.2: Istogramma rappresentante lo space saving raggiunto nei test attraverso la compressione dei pacchetti con differenti CDA adottando o meno il bit padding.

L'algoritmo è stato infine testato in termini di efficienza temporale. In questo caso è importante segnalare che le prove sono state svolte su un notebook Asus con sistema operativo Ubuntu Linux 14.04, dotato di processore Intel i7-2670QM@2.20GHz. La piattaforma non si può comparare lato device a quella di un dispositivo IoT, ma tuttavia i valori ottenuti forniscono comunque una valida indicazione su quella che è la velocità del sistema. I test dimostrano un tempo di esecuzione inferiore agli 0.7 millisecondi sia per la compressione che per la decompressione. Come ci si poteva aspettare, vista la struttura dell'algoritmo mostrata attraverso gli pseudocodici nelle sezioni precedenti, la parte di decompressione si rivela essere nettamente più veloce della compressione, con valori intorno agli 0.2 millisecondi. L'algoritmo SCHC si dimostra dunque essere sia efficace in termini di capacità di compressione sia efficiente in termini temporali.

METODO	**CDA**	**TEMPO [ms]**
COMPRESSIONE	Not Sent	0.483
	Mapping Sent	0.622
	LSB	0.539
DECOMPRESSIONE	Not Sent	0.209
	Mapping Sent	0.211
	LSB	0.214

Tabella 4.7: Tempi di compressione e decompressione ottenuti con differenti CDA.

5

Conclusioni e sviluppi futuri

Attraverso il presente lavoro di tesi è stata effettua un'analisi approfondita dell'algoritmo SCHC, che rappresenta lo stato dell'arte per quanto riguarda la compressione di pacchetti IP su reti LPWAN e di questo è stata sviluppata una delle primissime implementazioni in linguaggio C++.
Dopo aver presentato il nuovo paradigma chiamato Internet of Things, il lavoro si è soffermato ad analizzare le nuove tecnologie Low-Power Wide-Area Network (LPWAN), le quali permettono l'interconnessione di oggetti posti a distanza di decine di chilometri e assicurano un'autonomia di diversi anni anche nel caso di dispositivi alimentati a batteria. Tra queste reti è stato approfondito il sistema LoRa, che si compone dello strato fisico LoRa PHY e della componente MAC descritta dal protocollo LoRaWAN, e che per le determinate caratteristiche si configura come la più promettente.
Tuttavia LoRa, come anche le altre LPWAN, non garantisce un bit rate sufficiente a poter supportare quei protocolli dello stack Internet come IP ed UDP che, di fatto, sono indispensabili per poter comunicare direttamente con il resto delle reti. Da questa mancanza, nasce il lavoro promosso da IETF e portato avanti dall'LPWAN Working Group, il quale si prefigge l'obiettivo di produrre quegli standard necessari ad abilitare la connettività IP su queste reti. Raggiungere tale scopo è di primaria importanza per la futura adozione di massa delle tecnologie LPWAN.
La tesi si è dunque focalizzata su quello che è il nocciolo della questione, rappresen-

tato dalla necessità di comprimere l'header dei pacchetti IP e la cui risposta è fornita dall'algoritmo Static Context Header Compression (SCHC). A partire dalla descrizione formale dei suoi meccanismi, è stata dunque sviluppata una delle prime implementazioni di questo futuro standard. Il lavoro fatto avrà dunque un ruolo di primo piano su quello che sarà l'IoT di domani. Lo sviluppo ha tenuto ben presente le caratteristiche e i limiti dell'hardware per cui il software è pensato e per questo è stato scelto il linguaggio di programmazione C++, il cui compilatore è presente anche sui dispositivi più minimali. I risultati ottenuti hanno confermato la grande efficacia dell'algoritmo nel saper ridurre i pacchetti IP in pochi bit e di saperlo fare con grande efficienza sia nella fase di compressione che nella successiva di decompressione. Infine nel software realizzato sono stati inseriti dei metodi utili per potersi interfacciare con suite di analisi di rete, quali per esempio Wireshark, rendono in questo modo il programma realizzato più aperto e pronto ad essere utilizzato anche in scenari differenti.

Il lavoro è aperto a numerosi sviluppi futuri, in quanto seppure l'algoritmo SCHC per quanto riguarda la parte di compressore dell'header IPv6 e UDP sia pressoché completo ed in via di standardizzazione, le altre parti che compongono il disegno finale del progetto portato avanti da IETF sono ancora aperte e presentano tuttora diverse problematiche, come per esempio la gestione del protocollo ICMPv6 e dunque anche del Neighbor Discovery Protocol (NDP) utilizzato dalle reti IP per individuare i nodi nella rete, i router disponibili e per l'autoconfigurazione dei nodi. Per quanto riguarda l'implementazione fatta, un lavoro comunque importante sarà quello di ampliare il software al supporto del protocollo CoAP, già presentato all'interno della tesi, così da inglobare uno dei più importanti protocolli di livello applicazione del mondo IoT.

Appendice

A Codice schc.h

```
/***********************************************************
*                   University of Padova                   *
*                       A.Y. 16/17                         *
*                                                          *
*   THESIS: Analysis of the SCHC algorithm for             *
*           IP packets compression in Lo-Ra networks       *
*                                                          *
*   TRACK:                                                 *
*   Provide an efficient implementation of                 *
*   SCHC algorithm and its compressor/decompressor         *
*   mechanism for UDP/IPv6 packets.                        *
*                                                          *
*   AUTHOR:                                                *
*   Tommasi Claudio     tommasi.claudio[AT]gmail.com       *
*                                                          *
**********************************************************/

#ifndef SCHC_TESI_SCHC_H
#define SCHC_TESI_SCHC_H

#include <vector>
#include <bitset>
#include <iostream>
#include <tins/tins.h>

/**
 *  Definition of SCHC rule field names
 */

#define N_RULE_ID_BIT   3       //Bits of a Rule ID
#define N_FIELD_MAX     14      //Max fields in a Rule

typedef enum{
    IPv6_Version,       // IPv6 Version
    IPv6_DiffServ,      // IPv6 Traffic class
    IPv6_FlowLabel,     // IPv6 Flow label
    IPv6_Length,        // IPv6 Length
    IPv6_NextHeader,    // IPv6 Next header
    IPv6_HopLimit,      // IPv6 Hop limit
    IPv6_DEVprefix,     // IPv6 Device prefix
    IPv6_DEViid,        // IPv6 Device iid
    IPv6_APPprefix,     // IPv6 Application prefix
    IPv6_APPiid,        // IPv6 Application iid
    UDP_DEVport,        // UDP Device port
    UDP_APPport,        // UDP Application port
    UDP_Length,         // UDP Length
    UDP_Checksum,       // UDP Checksum
}SCHC_FieldName;


typedef enum{
    Up,         // Upstream
    Dw,         // Downstream
    Bi          // Bidirectional
}SCHC_DirectionIndicator;


typedef enum{
    Equal,          // Equal
    Ignore,         // Ignore
    MSB,            // Most Significant Bit
    Match_mapping   // Match Mapping
}SCHC_MatchingOperator;


typedef enum{
    Not_sent,       // Not Send
    Value_sent,     // Value Sent
    Mapping_sent,   // Mapping Sent
    LSB,            // Least Significant Bit
    DEViid,         // Reconstruct DEV-IID from L2 header
    APPiid,         // Reconstruct APP-IID from L2 header
    Comp_length,    // Compute Length
```

```
74      Comp_chk        // Compute Checksum
75  }SCHC_CDA;
76
77
78  /**
79   * Description of a Rule field in a SCHC rule
80   */
81
82
83  typedef struct {
84      SCHC_FieldName          field_name;                 // Field Name
85      int                     field_position;             // Field Position
86      SCHC_DirectionIndicator direction_indicator;        // Director Allocator
87      int                     target_value_int;           // Target Value
88      Tins::IPv6Address       target_value_ip;            // Target Value for IPv6
        address
89      std::vector<Tins::IPv6Address> target_value_ip_vec; // Target Value for IPv6
        address vec
90      SCHC_MatchingOperator   mo_field;                   // Matching Operator
91      SCHC_CDA                cda_field;                  // Compression-Decompression
        Action
92  }SCHC_Rule_field;
93
94  /**
95   * Description of a rule in the SCHC context
96   */
97
98  typedef struct{
99      std::bitset<N_RULE_ID_BIT>      rule_id;                    // Rule id (bit
        format)
100     SCHC_Rule_field                 rule_field[N_FIELD_MAX];    // Rule field
101 }SCHC_Rule;
102
103 #endif //SCHC_TESI_SCHC_H
104
```

B Codice comp_decomp.cpp

```
1  /*********************************************************
2   *                  University of Padova                 *
3   *                      A.Y. 16/17                       *
4   *                                                       *
5   *   THESIS: Analysis of the SCHC algorithm for          *
6   *           IP packets compression in Lo-Ra networks    *
7   *                                                       *
8   *   TRACK:                                              *
9   *   Provide an efficient implementation of              *
10  *   SCHC algorithm and its compressor/decompressor      *
11  *   mechanism for UDP/IPv6 packets.                     *
12  *                                                       *
13  *   AUTHOR:                                             *
14  *   Tommasi Claudio     tommasi.claudio[AT]gmail.com    *
15  *                                                       *
16  *********************************************************/
17
18 #include "../include/comp_decomp.h"
19
20 using namespace std;
21 using namespace Tins;
22
23 /**
24  * SCHC Compressor
25  * Compress a UDP/IPv6 packet using the rules in the SCHC context.
26  * @param ipv6_pkt The IPv6 packet to compress. <input>
27  * @param context The SCHC context with the rules. <input>
28  * @param di The communication direction. <input>
29  * @param rule_id Rule id selected by the compressor. <output>
30  * @param hdr_info Other header info of the compressed packet. <output>
31  * @return bool True if a rule ID is found by the compressor, False otherwise.
32  */
33 bool compress(IPv6 &ipv6_pkt, std::vector<SCHC_Rule> context,
   SCHC_DirectionIndicator di,
34               bitset<N_RULE_ID_BIT> &rule_id, vector<bool> &comp_hdr_info) {
35
36     bool found = false;
37
38     cout << "Rule in the context: " << endl;
39     for (int i=0; i<context.size(); i++){
40         cout << "context[" << i <<"].rule_id= " << context[i].rule_id << endl;
41     }
42
43     int ctx_rules_num = (int)context.size(); //Number of rules in the context
44     UDP udp_pkt = ipv6_pkt.rfind_pdu<UDP>(); //UDP inner pdu
45
46     Tins::IPv6Address prefix_mask = "ffff:ffff:ffff:ffff::";           //prefix mask
47     Tins::IPv6Address src_prefix= ipv6_pkt.src_addr() & prefix_mask;   //src address
       prefix
48     Tins::IPv6Address dst_prefix= ipv6_pkt.dst_addr() & prefix_mask;   //dst address
       prefix
49
50     Tins::IPv6Address iid_mask = "::ffff:ffff:ffff:ffff";          //iid mask
51     Tins::IPv6Address src_iid= ipv6_pkt.src_addr() & iid_mask;   //src address iid
52     Tins::IPv6Address dst_iid= ipv6_pkt.dst_addr() & iid_mask;   //dst address iid
53
54     for (int rule=0; rule<ctx_rules_num; rule++){       //For all rules in the context
55         int max_field = sizeof(context[rule].rule_field)
56                         /sizeof(context[rule].rule_field[0]);  //Number of fields in
                           a rule
57
58         cout << "Rule: " << rule << "\tNumber of Field: " << max_field << endl;
59         bool mo_missing=false;  //Initialise mo_missing
60
61         for (int field=0; field < max_field; field++){  //Check Matching Operators
62
63             if (di == context[rule].rule_field[field].direction_indicator
64                 || context[rule].rule_field[field].direction_indicator==Bi) //Check
                   direction indicator
65             {
66
67                 if (context[rule].rule_field[field].mo_field == Equal)  //if MO=Equal
68                 {
```

```
69
70              switch (context[rule].rule_field[field].field_name)
71              {   //Check if TV==IP_pkt_field_value
72                  case (IPv6_Version):
73                      if (context[rule].rule_field[field].target_value_int !=
74                          ipv6_pkt.version())
75                      {
76                          mo_missing = true;
77                      }
78                      break;
79                  case (IPv6_DiffServ):
80                      if (context[rule].rule_field[field].target_value_int !=
81                          ipv6_pkt.traffic_class())
82                      {
83                          mo_missing = true;
84                      }
85                      break;
86                  case (IPv6_FlowLabel):
87                      if (context[rule].rule_field[field].target_value_int !=
88                          ipv6_pkt.flow_label())
89                      {
90                          mo_missing = true;
91                      }
92                      break;
93                  case (IPv6_Length):
94                      if (context[rule].rule_field[field].target_value_int !=
95                          ipv6_pkt.payload_length())
96                      {
97                          mo_missing = true;
98                      }
99                      break;
100                 case (IPv6_NextHeader):
101                     if (context[rule].rule_field[field].target_value_int !=
102                         ipv6_pkt.next_header())
103                     {
104                         mo_missing = true;
105                     }
106                     break;
107                 case (IPv6_HopLimit):
108                     if (context[rule].rule_field[field].target_value_int !=
109                         ipv6_pkt.hop_limit())
110                     {
111                         mo_missing = true;
112                     }
113                     break;
114                 case (IPv6_DEVprefix):
115                     if (di == Up)
116                     {
117                         if (context[rule].rule_field[field].target_value_ip
                            != src_prefix)
118                         {
119                             mo_missing = true;
120                         }
121                     }
122                     if (di == Dw)
123                     {
124                         if (context[rule].rule_field[field].target_value_ip
                            != dst_prefix)
125                         {
126                             mo_missing = true;
127                         }
128                     }
129                     break;
130                 case (IPv6_DEViid):
131                     if (di == Up)
132                     {
133                         if (context[rule].rule_field[field].target_value_ip
                            != src_iid)
134                         {
135                             mo_missing = true;
136                         }
137                     }
138                     if (di == Dw)
```

```
139                         {
140                             if (context[rule].rule_field[field].target_value_ip
                                != dst_iid)
141                             {
142                                 mo_missing = true;
143                             }
144                         }
145                         break;
146                     case (IPv6_APPprefix):
147                         if (di == Up)
148                         {
149                             if (context[rule].rule_field[field].target_value_ip
                                != dst_prefix)
150                             {
151                                 mo_missing = true;
152                             }
153                         }
154                         if (di == Dw)
155                         {
156                             if (context[rule].rule_field[field].target_value_ip
                                != src_prefix)
157                             {
158                                 mo_missing = true;
159                             }
160                         }
161                         break;
162                     case (IPv6_APPiid):
163                         if (di == Up)
164                         {
165                             if (context[rule].rule_field[field].target_value_ip
                                != dst_iid)
166                             {
167                                 mo_missing = true;
168                             }
169                         }
170                         if (di == Dw)
171                         {
172                             if (context[rule].rule_field[field].target_value_ip
                                != src_iid)
173                             {
174                                 mo_missing = true;
175                             }
176                         }
177                         break;
178                     case (UDP_DEVport):
179                         if (di == Up)
180                         {
181                             if (context[rule].rule_field[field].target_value_int
                                !=
182                                 udp_pkt.sport())
183                             {
184                                 mo_missing = true;
185                             }
186                         }
187                         if (di == Dw)
188                         {
189                             if (context[rule].rule_field[field].target_value_int
                                !=
190                                 udp_pkt.dport())
191                             {
192                                 mo_missing = true;
193                             }
194                         }
195                         break;
196 
197                     case (UDP_APPport):
198                         if (di == Up)
199                         {
200                             if (context[rule].rule_field[field].target_value_int
                                !=
201                                 udp_pkt.sport())
202                             {
203                                 mo_missing = true;
```

```
204                                 }
205                             }
206                             if (di == Dw)
207                             {
208                                 if (context[rule].rule_field[field].target_value_int
                                    !=
209                                     udp_pkt.dport())
210                                 {
211                                     mo_missing = true;
212                                 }
213                             }
214                             break;
215                         case (UDP_Length):
216                             if (context[rule].rule_field[field].target_value_int !=
217                                 udp_pkt.length())
218                             {
219                                 mo_missing = true;
220                             }
221                             break;
222                         case (UDP_Checksum):
223                             if (context[rule].rule_field[field].target_value_int !=
224                                 udp_pkt.checksum())
225                             {
226                                 mo_missing = true;
227                             }
228                             break;
229                     }//end switch
230                 }//end if MO=Equal
231 
232                 if (context[rule].rule_field[field].mo_field == Match_mapping) //If
                    MO==Match mapping
233                 {
234                     switch (context[rule].rule_field[field].field_name)
235                     { //Check if TV==IP_pkt_field_value
236                         case (IPv6_DEVprefix):
237                             if (di == Up)
238                             {
239                                 if
                                    (std::find(context[rule].rule_field[field].target_valu
                                    e_ip_vec.begin(),
240 
                                                context[rule].rule_field[field].target_v
                                                alue_ip_vec.end(),
241                                             src_prefix) ==
242 
                                        context[rule].rule_field[field].target_value_ip_ve
                                        c.end())
243                                 {
244                                     mo_missing = true;
245                                 }
246                             }
247                             if (di == Dw)
248                             {
249                                 if
                                    (std::find(context[rule].rule_field[field].target_valu
                                    e_ip_vec.begin(),
250 
                                                context[rule].rule_field[field].target_v
                                                alue_ip_vec.end(),
251                                             dst_prefix) ==
252 
                                        context[rule].rule_field[field].target_value_ip_ve
                                        c.end())
253                                 {
254                                     mo_missing = true;
255                                 }
256                             }
257                             break;
258                         case (IPv6_APPprefix):
259                             if (di == Up)
260                             {
261                                 if
                                    (std::find(context[rule].rule_field[field].target_valu
```

```
                    e_ip_vec.begin(),
262
                                context[rule].rule_field[field].target_v
                                alue_ip_vec.end(),
263                             dst_prefix) ==
264
                        context[rule].rule_field[field].target_value_ip_ve
                        c.end())
265                 {
266                     mo_missing = true;
267                 }
268             }
269             if (di == Dw)
270             {
271                 if
                    (std::find(context[rule].rule_field[field].target_valu
                    e_ip_vec.begin(),
272
                                context[rule].rule_field[field].target_v
                                alue_ip_vec.end(),
273                             src_prefix) ==
274
                        context[rule].rule_field[field].target_value_ip_ve
                        c.end())
275                 {
276                     mo_missing = true;
277                 }
278             }
279             break;
280         }//end switch
281     }//end if MO=Match Mapping
282
283     if (context[rule].rule_field[field].mo_field == MSB)     //If MO==MSB
284     {
285         switch (context[rule].rule_field[field].field_name)
286         {
287             case (UDP_DEVport):
288             if (di == Up)
289             {
290                 std::bitset<16> tv_bit(
291                         (unsigned long)
                        context[rule].rule_field[field].target_value_i
                        nt);
292                 std::bitset<16> port_bit(udp_pkt.sport());
293                 std::cout << "tv_bit: " << tv_bit << std::endl;
294                 std::cout << "port_bit: " << port_bit << std::endl;
295                 for (int i = 16; i > 0; i--)
296                 {
297                     if (tv_bit[i - 1] == 1 && port_bit[i - 1] == 0)
298                     {
299                         mo_missing = true;
300                     }
301                 }
302             }
303             if (di == Dw)
304             {
305                 std::bitset<16> tv_bit(
306                         (unsigned long)
                        context[rule].rule_field[field].target_value_i
                        nt);
307                 std::bitset<16> port_bit(udp_pkt.dport());
308                 for (int i = 16; i > 0; i--)
309                 {
310                     if (tv_bit[i - 1] == 1 && port_bit[i - 1] == 0)
311                     {
312                         mo_missing = true;
313                     }
314                 }
315             }
316             break;
317
318             case (UDP_APPport):
319             if (di == Up)
```

```
320                     {
321                         std::bitset<16> tv_bit(
322                                 (unsigned long)
                                context[rule].rule_field[field].target_value_i
                                nt);
323                         std::bitset<16> port_bit(udp_pkt.sport());
324                         for (int i = 16; i > 0; i--)
325                         {
326                             if (tv_bit[i - 1] == 1 && port_bit[i - 1] == 0)
327                             {
328                                 mo_missing = true;
329                             }
330                         }
331                     }
332                     if (di == Dw)
333                     {
334                         std::bitset<16> tv_bit(
335                                 (unsigned long)
                                context[rule].rule_field[field].target_value_i
                                nt);
336                         std::bitset<16> port_bit(udp_pkt.dport());
337                         for (int i = 16; i > 0; i--)
338                         {
339                             if (tv_bit[i - 1] == 1 && port_bit[i - 1] == 0)
340                             {
341                                 mo_missing = true;
342                             }
343                         }
344                     }
345                     break;
346                 }//end switch
347             }//end if MSB
348 
349             cout << "MatchingOperator[Rule=" << rule << "][Field=" << field <<
            "]="
350                 << (mo_missing ? "wrong" : "ok") << endl;
351 
352             if (mo_missing) break; //if there is an unsatisfied mo go to the
            next rule
353         }//end if dir field is correct
354     }//end for checking MO of different field
355 
356     if (!mo_missing){ //if no MO was wrong, rule found!!!
357         found = true;
358         rule_id = context[rule].rule_id;
359 
360         cout << "Rule ID found: " << context[rule].rule_id << endl;
361 
362         for (int field=0; field < max_field; field++)
363         {  //do Compression actions
364             if (context[rule].rule_field[field].cda_field == Not_sent)
365             { //Not sent
366                 //Do nothing
367             }
368             if (context[rule].rule_field[field].cda_field == Value_sent)
369             { //Value sent
370                 std::vector<bool> vec;
371                 int i = context[rule].rule_field[field].target_value_int;
372                 while (i)
373                 {
374                     vec.push_back(i & 1);
375                     i >>= 1;
376                 }
377                 std::reverse(vec.begin(), vec.end());
378                 comp_hdr_info.insert(comp_hdr_info.end(), vec.begin(), vec.end());
379             }
380             if (context[rule].rule_field[field].cda_field == LSB)
381             { //Least Significant Bit
382                 std::bitset<16> tv_bit(
383                         (unsigned long)
                        context[rule].rule_field[field].target_value_int);
384 
385                 std::bitset<16> port_bit;
```

```
386                         std::vector<bool> lsb_vec;
387                         int count_lsb=0;
388
389                         switch (context[rule].rule_field[field].field_name)
390                         {
391                             case (UDP_DEVport):
392
393                                 if (di == Up){
394                                     port_bit=udp_pkt.sport();
395                                     cout << "Port_bit= "<< port_bit << endl;
396                                 }
397                                 else {
398                                     port_bit=udp_pkt.dport();
399                                     cout << "Port_bit= "<< port_bit << endl;
400                                 }
401
402                                 for (int i = 0; i <16; i++){
403                                     if (tv_bit[i]==0) { //read the first 0bits which
                                    mask the lsb
404                                         count_lsb++;    //increase the counter
405                                     }
406                                     else break;         // when 1 is read exit from cycle
407                                 }
408
409                                 for (int i=0; i<count_lsb; i++){//create a vector with
                                the lsb
410                                     lsb_vec.push_back(port_bit[i]);
411                                 }
412                                 std::reverse(lsb_vec.begin(), lsb_vec.end());
413                                 comp_hdr_info.insert(comp_hdr_info.end(),
                                lsb_vec.begin(), lsb_vec.end());
414                                 break;
415
416                             case (UDP_APPport):
417                                 if (di == Up){
418                                     port_bit=udp_pkt.dport();
419                                     cout << "Port_bit= "<< port_bit << endl;
420                                 }
421                                 else {
422                                     port_bit=udp_pkt.sport();
423                                     cout << "Port_bit= "<< port_bit << endl;
424                                 }
425
426                                 for (int i = 0; i <16; i++){
427                                     if (tv_bit[i]==0) { //read the first 0bits which
                                    mask the lsb
428                                         count_lsb++;    //increase the counter
429                                     }
430                                     else break;         // when 1 is read exit from cycle
431                                 }
432
433                                 for (int i=0; i<count_lsb; i++){//create a vector with
                                the lsb
434                                     lsb_vec.push_back(port_bit[i]);
435                                 }
436                                 std::reverse(lsb_vec.begin(), lsb_vec.end());
437                                 comp_hdr_info.insert(comp_hdr_info.end(),
                                lsb_vec.begin(), lsb_vec.end());
438                                 break;
439
440                         }//end switch
441
442                     }//end if LSB
443
444
445                     if (context[rule].rule_field[field].cda_field == Mapping_sent)
446                     { //Mapping sent
447                         int n_elem =
                        (int)context[rule].rule_field[field].target_value_ip_vec.size();
448                         int n_bit =
                        (int)ceil(log2(context[rule].rule_field[field].target_value_ip_vec
                        .size()));
449                         int i=0;
```

```
450                     for (i; i<n_elem; i++){
451                         bool case1 =
                            (((context[rule].rule_field[field].field_name==IPv6_DEVprefix
                            && di==Up)
452                                 ||
                                    (context[rule].rule_field[field].field_name==IPv6_APPp
                                    refix && di==Dw))
453                                 &&
                                    context[rule].rule_field[field].target_value_ip_vec[i]=
                                    =ipv6_pkt.src_addr());
454                         bool case2 =
                            (((context[rule].rule_field[field].field_name==IPv6_DEVprefix
                            && di==Dw)
455                                  ||
                                     (context[rule].rule_field[field].field_name==IPv6_APP
                                     prefix && di==Up))
456                                  &&
                                     context[rule].rule_field[field].target_value_ip_vec[i]
                                     ==ipv6_pkt.dst_addr());
457                         std::cout << "i=" << i << "\tcase1: " << case1 << "\tcase2:
                            " << case2 << std::endl;
458                         if (case1 || case2) {
459                             break;
460                         }
461
462                     }//end for
463
464                     std::cout << "mapping value: " << i << std::endl;
465                     std::vector<bool> vec;
466
467                     while (i) // i is the mapping value
468                     {
469                         vec.push_back(i & 1);
470                         i >>= 1;
471                     }
472                     vec.resize(n_bit);
473                     std::reverse(vec.begin(), vec.end());
474                     std::cout << "vec: " ;
475                     for (int j=0; j<vec.size(); j++){
476                         std::cout << vec[i];
477                     }
478                     std::cout << std::endl;
479
480                     comp_hdr_info.insert(comp_hdr_info.end(), vec.begin(), vec.end());
481                 }
482             }//end for fields
483
484             return found;   //found==true
485
486         }//end if rule found
487     }// end for rules
488
489     return found;   //found==false
490 }//end compress
491
492
493 /**
494  * SCHC Decompressor
495  * @param compr_pkt Compressed packet to decompress.
496  * @param context The SCHC context with the rules.
497  * @param di The communication direction.
498  * @param ipv6_pkt_decomp Reconstructed IPv6 packet.
499  * @return bool True if decompression is successful, False otherwise.
500  */
501 bool decompress(std::vector<bool> &compr_pkt, std::vector<SCHC_Rule> context,
502                 SCHC_DirectionIndicator di, IPv6 &ipv6_pkt_decomp){
503
504     bitset<N_RULE_ID_BIT> rule_id;
505
506     std::string dev_prefix;
507     std::string app_prefix;
508     std::string dev_iid;
509     std::string app_iid;
```

```
510
511         UDP* udp_pkt_decomp = ipv6_pkt_decomp.find_pdu<UDP>(); //UDP inner pdu
512
513         //std::reverse(compr_pkt.begin(), compr_pkt.end());
514
515         /** Extract rule ID from the compressed packet **/
516
517         std::cout << "\nStarting decompressing phase..." << std::endl;
518
519         for (int i=0; i< N_RULE_ID_BIT; i++){
520             rule_id[N_RULE_ID_BIT-1-i]=compr_pkt[i];
521         }
522
523         std::cout << "Rule_id decompressed: " << rule_id << std::endl;
524
525         bool found = false;
526         int rule = 0;
527
528         for (int i=0; i<context.size(); i++) {
529             if(context[i].rule_id==rule_id) {
530                 found = true;
531                 rule = i;
532                 break;
533             }
534         }
535
536         if(!found){
537             return found;   //found=false
538         }
539         else{
540
541             int index=3;
542
543             int max_field = sizeof(context[rule].rule_field)
544                             /sizeof(context[rule].rule_field[0]);  //Number of fields in
                            the rule
545             for (int field=0; field < max_field; field++){
546                 if (di == context[rule].rule_field[field].direction_indicator
547                     || context[rule].rule_field[field].direction_indicator==Bi) //Check
                    direction indicator
548                 {
549                     switch (context[rule].rule_field[field].field_name)
550                     {
551                         case (IPv6_Version):
552                             if (context[rule].rule_field[field].cda_field==Value_sent){
553                                 int n_bit = 3;
554                                 int ip_value = 0;
555                                 int exp=n_bit-1; //exponent var
556                                 for (int i=0; i<n_bit; i++)
557                                 {
558                                     if (compr_pkt[index+i])
559                                     {
560                                         ip_value += pow(2, exp);
561                                     }
562                                     exp--;  //decrease exponent
563                                 }
564                                 index=index+n_bit;
565                                 ipv6_pkt_decomp.version(ip_value);
566                             }
567                             else if (context[rule].rule_field[field].cda_field==Not_sent){
568                                     ipv6_pkt_decomp.version(
569
                                    (Tins::small_uint<4>)context[rule].rule_field[fiel
                                    d].target_value_int);
570                             }
571                             break;
572
573                         case(IPv6_DiffServ):
574                             if (context[rule].rule_field[field].cda_field==Not_sent){
575
                                ipv6_pkt_decomp.traffic_class(context[rule].rule_field[fie
                                ld].target_value_int);
576                             }
```

```
577                             break;
578 
579                         case(IPv6_FlowLabel):
580                             if (context[rule].rule_field[field].cda_field==Not_sent){
581 
                                    ipv6_pkt_decomp.flow_label(context[rule].rule_field[field]
                                    .target_value_int);
582                             }
583                             break;
584 
585                         case(IPv6_Length):
586                             break;
587 
588                         case(IPv6_NextHeader):
589                             if (context[rule].rule_field[field].mo_field==Equal
590                                 && context[rule].rule_field[field].cda_field==Not_sent){
591 
                                    ipv6_pkt_decomp.next_header(context[rule].rule_field[field
                                    ].target_value_int);
592                             }
593                             break;
594 
595                         case(IPv6_HopLimit):
596                             if (context[rule].rule_field[field].cda_field==Not_sent){
597 
                                    ipv6_pkt_decomp.hop_limit(context[rule].rule_field[field].
                                    target_value_int);
598                             }
599                             break;
600 
601                         case(IPv6_DEVprefix):
602                             if(context[rule].rule_field[field].cda_field==Not_sent){
603                                 dev_prefix =
                                    context[rule].rule_field[field].target_value_ip.to_string(
                                    );
604                                 dev_prefix.erase(dev_prefix.end()-2,dev_prefix.end());
                                    //delete last "::"
605                             }
606                             else if
                                (context[rule].rule_field[field].cda_field==Mapping_sent){
607                                 int n_bit =
                                    (int)ceil(log2(context[rule].rule_field[field].target_valu
                                    e_ip_vec.size()));
608                                 int DEVprefix_index=0;
609                                 int exp=n_bit-1; //exponent var
610                                 for (int i=0; i<n_bit; i++)
611                                 {
612                                     if (compr_pkt[index+i])
613                                     {
614                                         DEVprefix_index += pow(2, exp);
615                                     }
616                                     exp--;  //decrease exponent
617                                 }
618                                 index=index+n_bit;
619                                 dev_prefix =
                                    context[rule].rule_field[field].target_value_ip_vec[DEVpre
                                    fix_index-1].to_string();
620                                 dev_prefix.erase(dev_prefix.end()-2,dev_prefix.end());
                                    //delete last "::"
621 
622                             }
623                             break;
624 
625                         case(IPv6_DEViid):
626                             if (context[rule].rule_field[field].cda_field==Not_sent){
627                                 dev_iid =
                                    context[rule].rule_field[field].target_value_ip.to_string(
                                    );
628                             }
629                             break;
630 
631                         case(IPv6_APPprefix):
632                             if(context[rule].rule_field[field].cda_field==Not_sent){
```

```
633                     app_prefix =
                        context[rule].rule_field[field].target_value_ip.to_string(
                        );
634                     app_prefix.erase(app_prefix.end()-2,app_prefix.end());
                        //delete last "::"
635                 }
636                 else if
                    (context[rule].rule_field[field].cda_field==Mapping_sent){
637                     int n_bit =
                        (int)ceil(log2(context[rule].rule_field[field].target_valu
                        e_ip_vec.size()));
638                     int APPprefix_index=0;
639                     int exp=n_bit-1; //exponent var
640                     for (int i=0; i<n_bit; i++)
641                     {
642                         if (compr_pkt[index+i])
643                         {
644                             APPprefix_index += pow(2, exp);
645                         }
646                         exp--;  //decrease exponent
647                     }
648                     index=index+n_bit;
649                     app_prefix =
                        context[rule].rule_field[field].target_value_ip_vec[APPpre
                        fix_index-1].to_string();
650                     app_prefix.erase(app_prefix.end()-2,app_prefix.end());
                        //delete last "::"
651
652                 }
653                 break;
654
655             case(IPv6_APPiid):
656                 if (context[rule].rule_field[field].cda_field==Not_sent){
657                     app_iid =
                        context[rule].rule_field[field].target_value_ip.to_string(
                        );
658                 }
659                 break;
660
661             case(UDP_DEVport):
662                 if (context[rule].rule_field[field].cda_field==Not_sent){
663                     if (di==Up)
664
                            udp_pkt_decomp->sport(context[rule].rule_field[field].
                            target_value_int);
665                     else
666
                            udp_pkt_decomp->dport(context[rule].rule_field[field].
                            target_value_int);
667                 }
668                 else if (context[rule].rule_field[field].cda_field==LSB){
669                     std::bitset<16> tv_bit(
670                             (unsigned long)
                                context[rule].rule_field[field].target_value_int);
671
672                     std::vector<bool> lsb_vec;
673                     int devport_value=0;
674                     int count_lsb=0;
675
676                     for (int i = 0; i <16; i++){
677                         if (tv_bit[i]==0) { //read the first 0bits which
                            mask the lsb
678                             count_lsb++;    //increase the counter
679                         }
680                         else break;         // when 1 is read exit from cycle
681                     }
682
683                     int exp=count_lsb-1;
684
685                     for (int i=0; i<count_lsb; i++)
686                     {
687                         if (compr_pkt[index+i])
688                         {
```

```
689                         devport_value += pow(2, exp);
690                     }
691                     exp--;  //decrease exponent
692                 }
693 
694                 index=index+count_lsb;
695                 int DEVport_recon= devport_value +
                    context[rule].rule_field[field].target_value_int;
696 
697                 if (di==Up){
698                     udp_pkt_decomp->sport(DEVport_recon);
699                 }
700 
701                 else {
702                     udp_pkt_decomp->dport(DEVport_recon);
703                 }
704             }
705             break;
706 
707         case(UDP_APPport):
708             if (context[rule].rule_field[field].cda_field==Not_sent){
709                 if (di==Up)
710 
                        udp_pkt_decomp->dport(context[rule].rule_field[field].
                        target_value_int);
711                 else
712 
                        udp_pkt_decomp->sport(context[rule].rule_field[field].
                        target_value_int);
713             }
714 
715             if (context[rule].rule_field[field].cda_field==LSB){
716                 std::bitset<16> tv_bit(
717                         (unsigned long)
                        context[rule].rule_field[field].target_value_int);
718 
719                 std::vector<bool> lsb_vec;
720                 int APPport_value=0;
721                 int count_lsb=0;
722 
723                 for (int i = 0; i <16; i++){
724                     if (tv_bit[i]==0) { //read the first 0bits which
                        mask the lsb
725                         count_lsb++;    //increase the counter
726                     }
727                     else break;         // when 1 is read exit from cycle
728                 }
729 
730                 int exp=count_lsb-1;
731 
732                 for (int i=0; i<count_lsb; i++)
733                 {
734                     if (compr_pkt[index+i])
735                     {
736                         APPport_value += pow(2, exp);
737                     }
738                     exp--;  //decrease exponent
739                 }
740 
741                 index=index+count_lsb;
742                 int APPport_recon= APPport_value +
743 
                                   context[rule].rule_field[field].target_
                                   value_int;
744 
745                 if (di==Up){
746                     udp_pkt_decomp->dport(APPport_recon);
747                 }
748                 else {
749                     udp_pkt_decomp->sport(APPport_recon);
750                 }
751             }//end if LSB
752             break;
```

```
753
754                         case(UDP_Length):
755                             break;
756
757                         case(UDP_Checksum):
758                             break;
759                     }//switch field name
760                 }//if dir
761
762         }//for field
763
764         /* Set ipv6 pkt dst and src addresses */
765         if (di==Up){
766             Tins::IPv6Address dst_address = app_prefix+app_iid;
767             Tins::IPv6Address src_address = dev_prefix+dev_iid;
768             ipv6_pkt_decomp.dst_addr(dst_address);
769             ipv6_pkt_decomp.src_addr(src_address);
770         }
771         else{ //di==Dw
772             Tins::IPv6Address dst_address = dev_prefix+dev_iid;
773             Tins::IPv6Address src_address = app_prefix+app_iid;
774             ipv6_pkt_decomp.dst_addr(dst_address);
775             ipv6_pkt_decomp.src_addr(src_address);
776         }
777
778     }//else
779
780     return found;   //found=true
781 }
782
783
784 /**
785  * SCHC Compressed Packet composer with or without padding
786  * @param rule_id Rule ID of the choosen rule.
787  * @param comp_pkt_hdr_info Compreesed header info generated by compression.
788  * @param padding If True padding bits are added to the final compressed pkt.
789  * @return Final compressed packet with or without padding bits.
790  */
791 std::vector<bool> composer(bitset<N_RULE_ID_BIT> rule_id, vector<bool>
    comp_pkt_hdr_info, bool padding)
792 {
793
794     std::vector<bool> compr_pkt;
795
796     for (int i = 0; i < N_RULE_ID_BIT; i++)
797     {
798         compr_pkt.push_back(rule_id[i]);
799     }
800
801     std::reverse(compr_pkt.begin(), compr_pkt.end());
802
803     compr_pkt.insert(compr_pkt.end(), comp_pkt_hdr_info.begin(),
    comp_pkt_hdr_info.end());
804
805     if (padding){
806         int mod = (int) compr_pkt.size() % 8;
807         //cout << "mod= " << mod << endl;
808
809         for (int i = 0; i < (8-mod); i++)
810         {
811             compr_pkt.push_back(0);
812         }
813     }
814
815     return compr_pkt;
816 }
```

Bibliografia

[1] Ericsson Mobility Report, November 2016, https://www.ericsson.com/res/docs/2016/ericsson-mobility-report-2016.pdf. Ultima visita: 10/08/2017.

[2] L. Atzori, A. Iera and G. Morabito, “The internet of things: A survey”, in Computer networks 54.15, pp. 2787-2805, 2010.

[3] R. Sanchez-Iborra and M. D. Cano, “State of the Art in LP-WAN Solutions for Industrial IoT Services”, in Sensors 16(5), 708, 2016.

[4] M. Centenaro, L. Vangelista, A. Zanella, and M. Zorzi, “Long-Range Communications in Unlicensed Bands: the Rising Stars in the IoT and Smart City Scenarios”, in IEEE Wireless Communications, Vol. 23, Oct. 2016.

[5] A. Minaburo, L. Toutain et al., “LPWAN Static Context Header Compression (SCHC) and fragmentation for IPv6 and UDP”, draft 6, 12 Sept. 2017.

[6] RFC 4260 Internet Protocol Version 6 (IPv6) Specification, https://tools.ietf.org/html/rfc2460. Ultima visita: 22/08/2017.

[7] A. Augustin, J. Yi, T. Clausen and W. M. Townsley, “A study of LoRa: Long range & low power networks for the internet of things”, in Sensors, 16(9), 1466, Basel, Switzerland, 2016.

[8] SemTech, “AN1200.22 LoRa^{TM} Modulation Basics”, www.semtech.com/images/datasheet/an1200.22.pdf. Ultima visita: 10/08/2017

[9] N. Sornin et al., “LoRaWAN^{TM} Specification 1.0.2”, July 2016.

[10] LoRa Alliance Technical Marketing Workgroup, “LoRaWAN What is it? A technical overview of LoRa and LoRaWAN”, November 2015.

[11] LoRa Alliance Technical committee, “LoRaWAN^{TM} Regional Parameters”, version 1.0.2 rev. B, February 2017.

[12] ETSI EN 300 220-2 V3.1.1, “Short Range Devices (SRD) operating in the frequency range 25 MHz to 1 000 MHz”, November 2016.

[13] http://www.sviluppoeconomico.gov.it/index.php/it/comunicazioni/radio/pnrf-piano-nazionale-di-ripartizione-delle-frequenze. Ultima visita: 15/09/2017.

[14] http://www.sviluppoeconomico.gov.it/images/stories/documenti/radio/Tabella_B_PNRF_2015.pdf. Ultima visita: 15/09/2017.

[15] J. de Carvalho Silva et al., “LoRaWAN - A low power WAN protocol for Internet of Things: A review and opportunities”, 2nd International Multidisciplinary Conference on Computer and Energy Science, pp. 1-6, Split, 2017.

[16] Z. Shelby, K. Hartke, C. Bormann and B. Frank, “RFC 7252, Constrained application protocol (CoAP)”, 2014.

[17] E. Mingozzi, G. Tanganelli and C. Vallati, “CoAP Proxy Virtualization for the Web of Things”, 2014 IEEE 6th International Conference on Cloud Computing Technology and Science, pp. 577-582, Singapore, 2014.

[18] A. Minaburo, L. Toutain et al., “LPWAN Static Context Header Compression (SCHC) for CoAP”, draft 2, 6 September 2017.

[19] K. Abdelfadeel, V. Cionca and D. Pesch, “LSCHC: Layered Static Context Header Compression for LPWANs”, in Proceedings of the 12th ACM Workshop on Challenged Networks, at Snowbird, Utah, USA, October 2017.

[20] R. Hummen et al., “6LoWPAN fragmentation attacks and mitigation mechanisms”, in Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks, New York, USA, 2013.

[21] Libtins, a packet crafting and sniffing library, http://libtins.github.io/. Ultima visita: 25/08/2017.