ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria
Laurea Magistrale in Ingegneria Informatica

Implementazione di un protocollo Header-compression a livello applicazione per scenari IoT

Tesi in
Multimedia Services and Applications M

Studente:
ANTONIO IANNACCONE

Relatore:
Prof.
DANIELE TARCHI
Co-supervisor:
Dott.
FRANCESCO PANDOLFI

SESSIONE Luglio
ANNO ACCADEMICO 2021–2022

scrivi qui
la dedica

KEYWORDS

Internet of Things

Protocolli a livello applicativo

Codifica Huffman

HTTPC

HPACK

Sommario

Il fenomeno noto come Internet of Things costituisce oggi il motore principale dell'espansione della rete Internet globale, essendo artefice del collegamento di miliardi di nuovi dispositivi. A causa delle limitate capacità energetiche e di elaborazione di questi dispositivi è necessario riprogettare molti dei protocolli Internet standard. Un esempio lampante è costituito dalla definizione del Constrained Application Protocol (CoAP), protocollo di comunicazione client-server pensato per sostituire HTTP in reti IoT. Per consentire la compatibilità tra reti IoT e rete Internet sono state definite delle linee guida per la mappatura di messaggi CoAP in messaggi HTTP e viceversa, consentendo così l'implementazione di proxies in grado di connettere una rete IoT ad Internet. Tuttavia, questa mappatura è circoscritta ai soli campi e messaggi che permettono di implementare un'architettura REST, rendendo dunque impossibile l'uso di protocolli di livello applicazione basati su HTTP.

La soluzione proposta consiste nella definizione di un protocollo di compressione adattiva dei messaggi HTTP, in modo che soluzioni valide fuori dagli scenari IoT, come ad esempio scambio di messaggi generici, possano essere implementate anche in reti IoT. I risultati ottenuti mostrano inoltre che nello scenario di riferimento la compressione adattiva di messaggi HTTP raggiunge prestazioni inferiori rispetto ad altri algoritmi di compressione di intestazioni (in particolare HPACK), ma più che valide perchè le uniche applicabili attualmente in scenari IoT.

Indice

Introduzione

Prima dell'avvento di Internet, le reti telefoniche a commutazione di circuito erano il principale mezzo di comunicazione. Negli anni '60 cominciarono a comparire le prime reti a commutazione di pacchetto, come ARPANET. L'uso commerciale di queste reti fu però possibile solo dopo l'introduzione di TCP/IP negli anni '80, fatto che segnò l'inizio di un'evoluzione tecnologica che dura ancora oggi. In appena quarant'anni, rivoluzioni come il World Wide Web, la telefonia mobile e il cloud computing hanno guidato un processo di espansione della rete Internet caratterizzato non solo da un aumento dei dispositivi connessi, ma anche da una forte differenziazione della tipologia di dispositivi connessi e di servizi offerti, che ha richiesto l'implementazione di una grande varietà di architetture e protocolli di comunicazione, i quali compongono l'insieme degli standard internet di oggi.

Il fenomeno Internet of Things (IoT) promette una trasformazione radicale della rete Internet, connettendo miliardi di nuovi dispositivi con carat-

teristiche tecniche molto diverse da quelle per cui erano stati pensati gran parte dei protocolli standard. La grande eterogeneità degli scenari IoT rende difficile trovare una soluzione che ottenga sempre risultati ottimali, di conseguenza si è venuto a creare un vasto ecosistema di protocolli particolarmente ampio a livello applicazione, in quanto il protocollo HTTP, standard internet di livello applicazione, si è rivelato inadeguato nella quasi totalità degli scenari IoT. In questi scenari i dispositivi sono wireless, alimentati a batteria e la fonte di maggior consumo energetico è costituita dell'uso dell'antenna: maggiore il numero di pacchetti trasmessi, maggiore il consumo. La dimensione dei pacchetti invece influisce in minima parte [1]. I fattori critici che hanno determinato il fallimento di HTTP sono essenzialmente due:

- HTTP tende a trasmettere messaggi lunghi, che saranno frammentati in molti pacchetti a livello trasporto e rete.

- HTTP è basato su TCP, un protocollo pesante in fase di connessione, mentre in scenari IoT le disconnessioni sono frequenti.

Il protocollo CoAP è un protocollo aperto e leggero per dispositivi IoT; è simile al linguaggio HTTP ma è stato riprogettato per soddisfare i requisiti dei dispositivi alimentati a batteria con risorse CPU e RAM limitate. IL protocollo CoAP è standard IoT per comunicazione client-server e risolve alcune

delle criticità citate precedentemente essendo basato su UDP e definendo messaggi codificati.

Lo scopo di questa tesi è proporre una strategia di riadattamento a scenari IoT di protocolli esistenti, basata su compressione adattiva dei messaggi trasmessi. É presentato un prototipo di protocollo di compressione di messaggi HTTP/1.1, chiamato HTTP Compressed o HTTPC. I vantaggi derivanti dall'uso di HTTPC rispetto a CoAP sono una maggior compressione dei messaggi, che quindi provoca un risparmio energetico, e una piena compatibilità con i protocolli basati su HTTP che non è possibile ottenere con CoAP. Una strategia di compressione analoga potrebbe inoltre essere applicata anche ad altri protocolli di livello applicazione, ad esempio Session Initiation Protocol (SIP), molto usato nelle reti mobili.

Nel capitolo 1 di questa tesi saranno nominati i protocolli standard e le problematiche principali introdotte dall'Internet of Things in ogni livello dello stack protocollare. Sarà inoltre definito lo scenario di riferimento per il quale HTTPC è stato progettato. Nel capitolo 2 sono riportati alcuni dei protocolli che lavorano a livello applicativo (HTTP 2.0, HTTP 3.0, COAP, ROHC) con le relative strategie per il *compression-header*.

Nel capitolo 3 si definirà la strategia di compressione adottata da HTTPC e si fornirà dello pseudocodice per l'implementazione dell'algoritmo di codifica.

Nel capitolo 4 si valuteranno le prestazioni di HTTPC in tre scenari IoT. Ed infine, nel capitolo 5 si valuteranno le prestazioni della strategia di compressione di HTTP 2 (HPACK) nei tre scenari applicativi a confronto con le prestazione di HTTPC.

Capitolo 1

Dispositivi e reti IoT

L'Internet di qualche anno fa, prima che esplodesse il fenomeno IoT, poteva essere definita una *Internet of People*, in quanto solo le persone eseguivano operazioni tramite applicazioni connesse, sia di propria iniziativa, sia perché notificate dalle applicazioni stesse [2]. Con il termine *Internet of Things* si indica una futura Internet che connetta applicazioni usate da qualsiasi cosa possa essere monitorata o controllata: automobili, edifici, animali, persone, ecc. I casi d'uso sono innumerevoli e coprono svariate aree di interesse (Figura 1.1).

Il punto di forza di questa tecnologia è la possibilità di generare una gran quantità di dati da settori diversi e, analizzandoli attraverso le tecniche emergenti dai *Big Data*, creare modelli che permettano ad una nuova generazione di applicazioni di prevedere il comportamento di un sistema o reagire ad esso

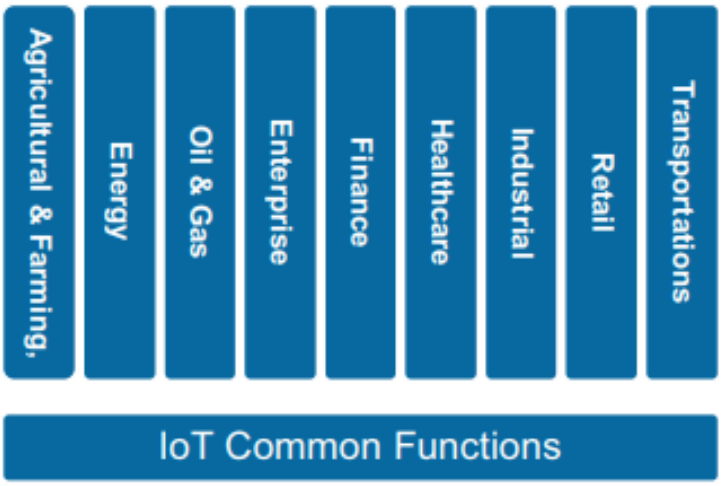


Figura 1.1: Aree di applicazione della tecnologia IoT (IoT Verticals) [3]

nel modo più efficace possibile.

Lo scopo della ricerca in ambito IoT è produrre uno stack protocollare standard da installare sui dispositivi, allo stesso modo di come lo stack TCP/IP è diventato standard di Internet. Come si vedrà nella prossima sezione, lo stack TCP/IP non è adatto a reti IoT e non è facile trovarne uno nuovo a causa della grande eterogeneità degli scenari in cui questo stack protocollare standard dovrà operare. Inoltre, qualsiasi sia la soluzione adottata, essa deve essere compatibile con TCP/IP per poter essere connessa a Internet.

In genere, quando i dati raccolti dai sensori, o più in generale quelli che transitano nella rete, sono di natura multimediale si parla di *Multimedia Internet of Things* [4], per evidenziare come in questi casi la richiesta di risorse sia maggiore rispetto ai casi con dati *scalari*.

Risorsa richiesta	Dati scalari	Dati multimediali
RAM	KB - MB	MB - GB
Frequenza CPU	KHz - MHz	MHz - GHz
Disco	KB - MB	GB
Banda	KB	MB
Sensibilità ai ritardi	bassa	alta
Consumo energetico	basso	alto

Tabella 1.1: Confronto tra dati scalari e multimediali

Le soluzioni proposte in letteratura per la gestione di dati multimediali in architetture IoT prevedono spesso l'introduzione di tecniche di data mining e sviluppo di reti neurali per gestire efficientemente la mole di dati [5, 6, 7].

Una qualsiasi soluzione IoT può essere scomposta in 4 livelli (Figura 1.2):

1. Device, comprende tutti i sensori e gli attuatori
2. Network, comprende tutti i dispositivi che consentono la connessione ad internet
3. Management Services Platform, comprende tutti i servizi e le astrazioni che consentono l'integrazione nella rete IoT di dispositivi e applicazioni eterogenee
4. Application, comprende tutte le applicazioni che operano nel sistema

In questa tesi si considereranno principalmente i primi due.

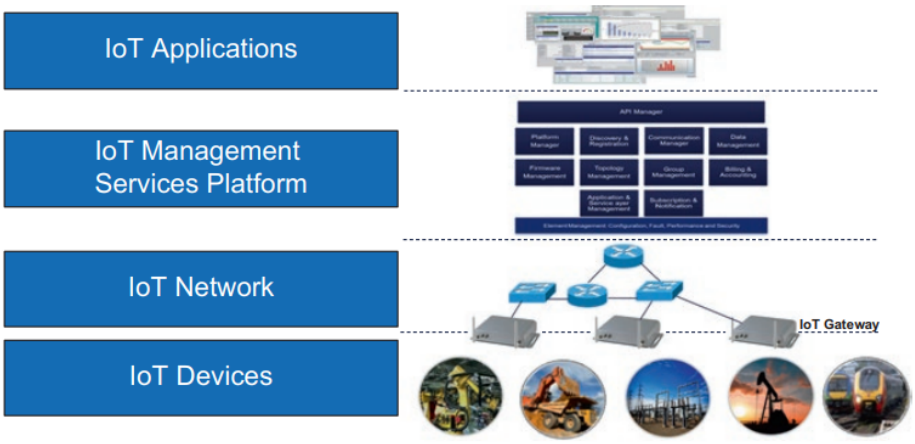


Figura 1.2: livelli di una soluzione IoT [2]

1.1 Sfide tecnologiche e protocolli standard

In questa sezione sono menzionate alcune tra le sfide tecnologiche poste dall'Internet of Things e alcuni tra i protocolli oggi più in uso. Per approfondimenti consultare [8].

1.1.1 Livello datalink

A livello 2 esistono quattro fattori da tenere in considerazione:

1. I dispositivi connessi ad una rete IoT possono avere caratteristiche tecniche e risorse molto differenti. La RFC 7228 [9] definisce 3 classi di dispositivi con vincoli su capacità di memoria RAM e Flash sempre più stringenti (Tabella 1.2). Arduino è un ottimo esempio di disposi-

tivo vincolato in memoria. Tuttavia non è raro l'utilizzo di dispositivi non vincolati, come Raspberry PI.

2. Le qualità del servizio (QoS) richieste in una rete IoT variano da caso a caso a seconda dell'applicazione. In alcuni casi, ad esempio in un'applicazione per il controllo di un motore jet, sono richieste una bassa latenza, una bassa perdita di pacchetti e una bassa variazione di latenza (jitter). In altri casi, ad esempio in un'applicazione per il monitoraggio meteorologico, questi requisiti possono essere più rilassati, nonostante il fatto si possano riutilizzare gli stessi componenti (sensori di umidità, pressione e temperatura).

3. L'applicazione a cui una rete IoT è destinata influisce anche sulle modalità di accesso alla rete. I nodi possono essere fissi o mobili e la copertura richiesta può essere a lungo, corto o medio raggio.

4. Una rete IoT deve essere scalabile. La trasmissione dati wireless determina un aumentare delle collisioni[1] e della latenza al crescere della rete. D'altro canto, una rete cablata richiederebbe costi di implementazione e distribuzione decisamente elevati.

I protocolli di livello datalink più usati sono ZigBee (IEEE 802.15.4 [11]), Bluetooth (IEEE 802.15.1 [12] e Bluetooth Low-Energy) e le varie versioni

[1] Si vedano problemi *hidden node* e *exposed node*[10]

di Wi-Fi (IEEE 802.11.x).

Nome	Memoria RAM	Memoria Flash
Classe 0	$\ll$ 10KB	$\ll$ 100KB
Classe 1	$\sim$ 10KB	$\sim$ 100KB
Classe 2	$\sim$ 50KB	$\sim$ 250KB

Tabella 1.2: Classificazione dispositivi per vincoli di memoria [9]

1.1.2 Livello rete

A livello 3 si ripropone il problema dei vincoli energetici e sulla memoria. Un protocollo di rete deve quindi essere ottimizzato al fine di richiedere la minor memoria possibile per le tabelle di routing e affrontare frequenti cambiamenti nella topologia della rete. Infatti, una delle soluzioni più usate per risparmiare energia è attivare e disattivare l'antenna periodicamente: un nodo sarà invisibile agli altri se l'antenna è disattivata. Inoltre, si aggiungono le difficoltà dovute ad un'alta probabilità di perdita dei pacchetti e alla necessità di supportare comunicazioni unicast, broadcast e multicast.

Uno dei protocolli più usati è 6LowPAN [13], basato su IPv6 [14] e pensato per operare su reti ZigBee. Si consulti RFC 4919 [15] per una trattazione più approfondita delle problematiche e degli obiettivi di 6LowPAN.

1.1.3 Livello trasporto

Il protocollo di livello 4 più utilizzato in scenari IoT è probabilmente UDP, su cui sono basati molti dei protocolli di livello applicazione. TCP è poco utilizzato e uno dei motivi è la pesantezza del protocollo, sopratutto in fase di handshake e stabilimento della connessione. Nonostante questi difetti TCP è divenuto standard internet perché le connessioni sono stabili in reti cablate. Ma in scenari in cui i nodi non mantengono a lungo una connessione la pesantezza di TCP risulta evidente.

Un'alternativa che potrebbe diffondersi in futuro è costituita dal protocollo QUIC [16], una versione più leggera di TCP.

Esistono poi protocolli, come ROHC [17], che implementano tecniche di compressione degli headers dei pacchetti. Queste tecniche, in realtà già usate a livello rete da 6LowPAN, si rivelano efficaci grazie alla frammentazione in pacchetti dei messaggi: dato un messaggio di livello applicazione, al momento della trasmissione esso verrà suddiviso in datagrammi del protocollo di livello trasporto usato (UDP, TCP, RTP...), i quali a loro volta saranno frammentati in pacchetti IP. Ad ognuno di questi datagrammi e pacchetti viene aggiunto in testa un header. Nonostante il fatto che la dimensione massima del body di un datagramma o pacchetto sia molto maggiore della dimensione del header, in reti con alte probabilità di perdita dei pacchetti

è consigliato non trasmettere pacchetti con bodies grandi, in modo da non dover ritrasmettere molti dati in caso di errore. La percentuale di bit utilizzati per la trasmissione degli headers sarà così comparabile a quella dei bit usati per la trasmissione dei bodies dei pacchetti, quindi una compressione degli headers permetterà di risparmiare una quantità di bit significativa. Le informazioni contenute negli headers saranno ripetute con alta probabilità tra gli headers dei datagrammi o pacchetti ottenuti da uno stesso messaggio di livello applicazione. 6LowPAN e ROHC sfruttano queste ripetizioni per riscrivere gli headers con meno bit.

1.1.4 Livello applicazione

I protocolli standard di livello applicazione sono divisi in due modelli: *publish-subscribe* e *client-server*. Due implementano un modello di comunicazione publish-subscribe (MQTT e AMQP), mentre gli altri due implementano un modello client-server (HTTP e CoAP). Per un confronto dettagliato si consulti [18].

Constrained Application Protocol (CoAP, RFC 7252 [19]) è stato sviluppato in seguito alla necessità di protocolli di comunicazione client-server più leggeri rispetto a HTTP. CoAP è basato su UDP, definisce messaggi codificati, in modo da ridurne le dimensioni, i quali possono essere mappati su

messaggi HTTP. Si prevede infatti l'uso di uno o più proxy HTTP/CoAP che connettano una rete IoT (in cui si usa CoAP) con internet (in cui si usa HTTP) convertendo un messaggio HTTP in CoAP e viceversa[2], in modo da mantenere la compatibilità con la rete internet. Tuttavia questa mappatura non comprende tutti i metodi e gli headers descritti da HTTP, ma è limitata ai soli metodi ed headers che consentono di definire architetture software conformi allo standard REpresentational State Transfer (REST) [21]. Dunque la mappatura non è perfetta, il che rende inutilizzabili protocolli basati su HTTP come HTTPS [22] e DASH [23]. Per questo motivo si sono sviluppate soluzioni per la sicurezza e lo streaming multimediale basate su CoAP e UDP, come DTLS [24] e DASCo [25]. Inoltre, CoAP necessita della definizione di politiche di ritrasmissione dei datagrammi a livello applicazione, in modo da poter offrire qualità del servizio superiore alla semplice trasmissione senza ricevuta di consegna offerta da UDP.

[2]RFC 8075 [20] contiene linee guida per la mappatura HTTP/CoAP

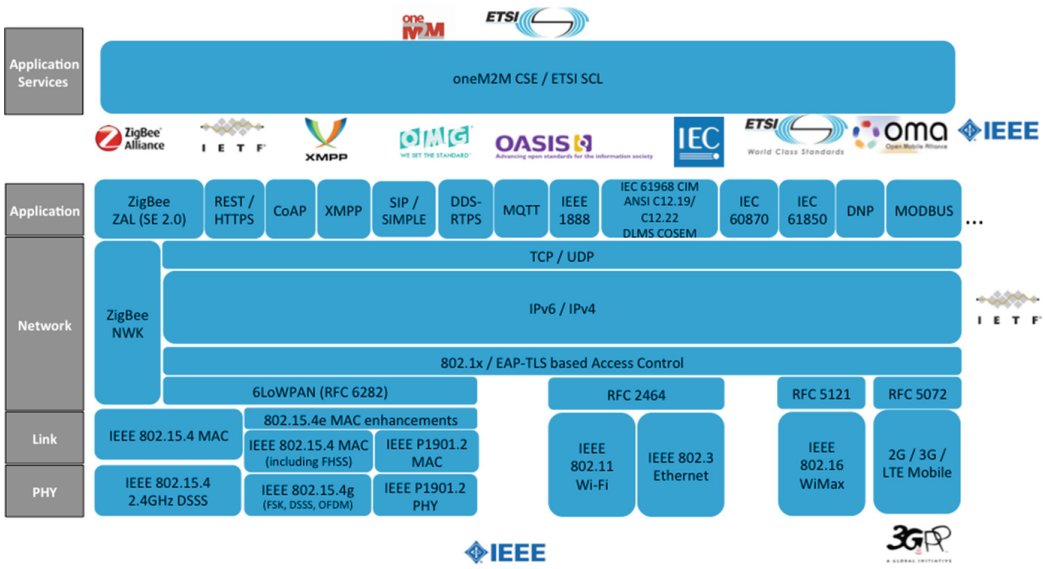


Figura 1.3: Protocolli standard nell'ecosistema IoT [26]

Capitolo 2

Protocolli a livello applicativo

2.1 Scenario di riferimento e obiettivi

Lo scopo di questa tesi è proporre una soluzione che consenta di ridurre il consumo energetico dei dispositivi e allo stesso tempo mantenere una piena compatibilità con HTTP. L'idea che si vuole promuovere consiste nel definire un protocollo di livello presentazione ISO/OSI che consenta una compressione adattiva dei messaggi HTTP. Le ultime due versioni di HTTP, HTTP/2.0 [27] e HTTP/3.0 [28] prevedono già due protocolli di compressione, rispettivamente HPACK [29] nella versione 2.0 e QPACK [30] nella 3.0, molto simili tra loro. Il protocollo proposto, chiamato HTTPC, raggiunge tassi di compressione maggiori. É progettato per comprimere messaggi HTTP/1.1 ma può essere facilmente esteso alle versioni successive. Uno sviluppo inte-

ressante potrebbe essere l'implementazione di HTTPC su HTTP/3.0, il quale prevede delle ottimizzazioni per essere usato su QUIC.

HTTPC è pensato per una rete costituita da più sensori (vincolati) connessi via wireless ad un unico master della rete (non vincolato) secondo una topologia a stella (Figura 2.1). Nel caso d'uso ideale, il master esegue un'applicazione che in determinati momenti (in seguito ad una richiesta di un utente, ad orari prefissati o ad intervalli regolari) invia una richiesta ad ogni sensore, il quale risponde trasmettendo la propria misurazione. In questo scenario di riferimento, in cui oggi si usa principalmente CoAP, non saranno considerati i problemi relativi alla sicurezza. Si assume inoltre che i dispositivi siano fissi e possiedano indirizzi IP statici.

Gli obiettivi saranno due:

1. La compressione deve produrre messaggi di lunghezza media comparabile o inferiore a quella dei messaggi CoAP, HTTP/2.0 e HTTP/3.0.

2. La compressione e la decompressione non devono richiedere elevate risorse computazionali, in quanto saranno eseguite da dispositivi vincolati. Inoltre il ritardo e l'overhead di comunicazione devono essere limitati.

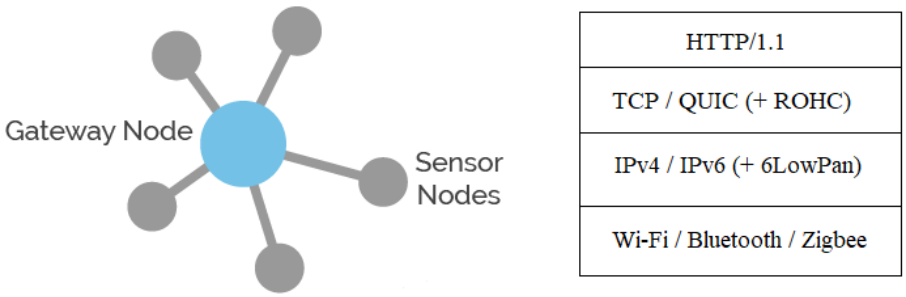


Figura 2.1: Esempio di rete a stella e stack protocollare dello scenario di riferimento

2.2 HTTP 2.0

Una normale connessione HTTPS è in realtà una sovrapposizione di diverse connessioni nel modello multistrato. Connessione TCP (il livello di trasporto), in aggiunta la connessione TLS (mix di livelli di trasporto/applicazione) e infine la connessione HTTP (livello di applicazione). Le pagine Web sono cresciute fino a richiedere decine o centinaia di richieste, i campi di intestazione ridondanti in queste richieste consumano inutilmente la larghezza di banda, aumentando in modo misurabile la latenza.

In passato, la compressione HTTP veniva eseguita nel livello TLS, utilizzando gzip. Sia le intestazioni che il corpo sono stati compressi indiscriminatamente, poiché il livello TLS inferiore non era a conoscenza del tipo di dati trasferito. In pratica significava che entrambi erano compressi con l'algoritmo DEFLATE.

Poi è arrivato SPDY con un nuovo algoritmo di compressione dell'in-

testazione dedicato. Sebbene progettato specificamente per le intestazioni, incluso l'uso di un dizionario preimpostato , utilizzava ancora DEFLATE, inclusi codici Huffman dinamici e corrispondenza di stringhe. Sfortunatamente entrambi sono risultati vulnerabili all'attacco CRIME , che può estrarre cookie di autenticazione segreti dalle intestazioni compresse: poiché DEFLATE utilizza corrispondenze di stringhe all'indietro e codici Huffman dinamici, un utente malintenzionato che può controllare parte delle intestazioni delle richieste, può recuperare gradualmente l'intero cookie modificando parti della richiesta e osservando come cambia la dimensione totale della richiesta durante la compressione. La maggior parte delle reti perimetrali ha disabilitato la compressione dell'intestazione a causa di CRIME [29]. Questo finché non è arrivato HTTP 2.0.

HTTP 2.0 supporta un nuovo algoritmo di compressione dell'intestazione dedicato, chiamato HPACK. HPACK è stato sviluppato pensando ad attacchi come il CRIME ed è quindi considerato sicuro da usare.

HPACK è resiliente a CRIME, perché non utilizza corrispondenze di stringhe all'indietro parziali e codici Huffman dinamici come DEFLATE. Invece, utilizza questi tre metodi di compressione:

1. Dizionario statico: un dizionario predefinito di 61 campi di intestazione di uso comune, alcuni con valori predefiniti.

2. Dizionario dinamico: un elenco di intestazioni effettive che sono state rilevate durante la connessione. Questo dizionario ha dimensioni limitate e quando vengono aggiunte nuove voci, le vecchie voci potrebbero essere eliminate.

3. Codifica Huffman: un codice Huffman statico può essere utilizzato per codificare qualsiasi stringa: nome o valore. Questo codice è stato calcolato specificamente per le intestazioni di risposta/richiesta HTTP: le cifre ASCII e le lettere minuscole hanno codifiche più brevi. La codifica più breve possibile è lunga 5 bit, quindi il rapporto di compressione più alto ottenibile è 8:5 (o il 37,5% più piccolo).

Quando HPACK deve codificare un'intestazione nel formato *name:value*, cercherà prima nei dizionari statici e dinamici. Se il *full name:value* è presente, farà semplicemente riferimento alla voce nel dizionario. Questo di solito richiede un byte e nella maggior parte dei casi due byte saranno sufficienti! Un intero header codificato in un singolo byte. Poiché molte intestazioni sono ripetitive, questa strategia ha un tasso di successo molto alto. Quando HPACK non riesce a trovare una corrispondenza con un'intera intestazione in un dizionario, tenterà di trovare un'intestazione con lo stesso nome. La maggior parte dei nomi di intestazione popolari sono presenti nella tabella statica, ad esempio: content-encoding, cookie, etag. Il resto è probabile che

sia ripetitivo e quindi presente nella tabella dinamica.Se il nome è stato trovato, può essere nuovamente espresso in uno o due byte nella maggior parte dei casi, altrimenti il nome verrà codificato utilizzando la codifica grezza o la codifica Huffman: la più breve delle due. Lo stesso vale per il valore dell'intestazione. Sebbene HPACK esegua la corrispondenza delle stringhe, affinché l'attaccante trovi il valore di un'intestazione, deve indovinare l'intero valore, invece di un approccio graduale che era possibile con la corrispondenza DEFLATE ed era vulnerabile a CRIME.

2.3 HTTP 3.0

HTTP 3.0 è un nuovo standard in fase di sviluppo che influenzerà il modo in cui i browser Web e i server comunicano, con aggiornamenti significativi per l'esperienza dell'utente, comprese prestazioni, affidabilità e sicurezza. Una differenza importante in HTTP/3 è che funziona su QUIC, un nuovo protocollo di trasporto. QUIC è progettato per l'utilizzo di Internet da dispositivi mobili in cui le persone portano con sé smartphone che passano costantemente da una rete all'altra mentre si spostano durante la giornata. Questo non era il caso quando furono sviluppati i primi protocolli Internet: i dispositivi erano meno portatili e non cambiavano rete molto spesso. L'uso di QUIC significa che HTTP/3 si basa sul protocollo UDP (User Datagram Protocol),

non sul protocollo TCP (Transmission Control Protoco). Il passaggio a UDP consentirà connessioni più veloci e un'esperienza utente più rapida durante la navigazione online.

Il protocollo QUIC è stato sviluppato da Google nel 2012 ed è stato adottato dall'Internet Engineering Task Force (IETF), un'organizzazione di standard indipendente dal fornitore, quando ha iniziato a creare il nuovo standard HTTP/3. Dopo aver consultato esperti di tutto il mondo, l'IETF ha apportato una serie di modifiche per sviluppare la propria versione di QUIC. QUIC aiuterà a correggere alcune delle maggiori carenze di HTTP/2:

1. Sviluppo di una soluzione alternativa per le prestazioni lente quando uno smartphone passa dal Wi-Fi ai dati cellulari (come quando si esce di casa o dall'ufficio)

2. Diminuzione degli effetti della perdita di pacchetti: quando un pacchetto di informazioni non arriva a destinazione, non bloccherà più tutti i flussi di informazioni (un problema noto come "blocco di testa")

Altri vantaggi includono:

1. Stabilimento della connessione più rapido: QUIC consente la negoziazione della versione TLS contemporaneamente all'handshake crittografico e di trasporto

2. Tempo di andata e ritorno zero (0-RTT): per i server a cui si sono già connessi, i client possono saltare il requisito di handshake (il processo di riconoscimento e verifica reciproca per determinare come comunicheranno)

3. Crittografia più completa: il nuovo approccio di QUIC all'handshake fornirà la *crittografia per impostazione predefinita*, un enorme aggiornamento da HTTP/2, e aiuterà a mitigare il rischio di attacchi

Come in HTTP 2.0, i campi delle richieste e delle risposte vengono compressi per la trasmissione. La richiesta della crittografia all'interno del livello di trasporto, anziché a livello dell'applicazione, ha importanti implicazioni per la sicurezza. Significa che la connessione sarà sempre crittografata. In precedenza, in HTTPS, la crittografia e le connessioni a livello di trasporto avvenivano separatamente. Le connessioni TCP potevano trasportare dati crittografati o non crittografati e l'handshake TCP e l'handshake TLS erano eventi distinti. Tuttavia, QUIC imposta le connessioni crittografate per impostazione predefinita a livello di trasporto: i dati a livello di applicazione saranno sempre crittografati. QUIC realizza questo combinando i due handshake in un'unica azione, riducendo la latenza poiché le applicazioni devono attendere il termine di un solo handshake prima di inviare i dati. Crittografa anche i metadati su ciascuna connessione, inclusi i numeri di pacchetto e

alcune altre parti dell'intestazione, per aiutare a mantenere le informazioni sul comportamento degli utenti fuori dalle mani degli aggressori ("attackers' hands"). Questa funzionalità non è stata inclusa in HTTP 2.0. HTTP 3.0 sostituisce HPACK con QPACK. QPACK utilizza flussi unidirezionali separati per modificare e seguire lo stato della tabella dei campi, mentre le sezioni di campo codificate fanno riferimento allo stato della tabella senza modificarlo. senza modificarlo. Mentre lo standard è ancora in fase di sviluppo, i proprietari di siti Web e i visitatori possono iniziare a ottenere supporto per HTTP 3.0 tramite browser, sistemi operativi e altri client. Naturalmente, ci sono probabilmente altre modifiche in vista dello standard, che ha già subito diverse implementazioni. Dopo il rilascio di HTTP 3.0, l'intero Web non passerà immediatamente. Molti siti non sono ancora nemmeno su HTTP 2.0.
Un potenziale ostacolo per il nuovo protocollo è che richiede un maggiore utilizzo della CPU sia per il server che per il client. Questo probabilmente diminuirà di impatto nel tempo man mano che la tecnologia si evolve.

2.4 COAP

Il Constrained Application Protocol (CoAP) è stato progettato con un duplice obiettivo: è un protocollo applicativo specializzato per ambienti vincolati ed è facilmente utilizzabile in architetture basate su Representational State

Transfer (REST), come il web. Quest'ultimo obiettivo ha portato a definire CoAP in modo che possa facilmente interoperare con HTTP attraverso un proxy intermediario che esegue la conversione interprotocollo.[19]

Il lavoro sui Constrained RESTful Environments (CoRE) mira a realizzare l'architettura REST in una forma adatta ai nodi più vincolati (ad esempio, microcontrollori a 8 bit con RAM e ROM limitate) e alle reti (ad esempio, 6LoWPAN]). Le reti vincolate come 6LoWPAN supportano la frammentazione dei pacchetti IPv6 in piccoli frame di livello link e livello di collegamento; tuttavia, ciò causa una significativa riduzione della probabilità di consegna dei pacchetti. probabilità di consegna dei pacchetti. Uno degli obiettivi della progettazione di CoAP è stato quello di mantenere messaggio, limitando così la necessità di frammentazione.

Uno degli obiettivi principali di CoAP è quello di progettare un protocollo web generico per i requisiti speciali di questo ambiente limitato, in particolare considerando l'energia, l'automazione degli edifici e altre applicazioni machine-to-machine (M2M). L'obiettivo di CoAP non è quello di comprimere ciecamente HTTP, ma piuttosto di realizzare un sottoinsieme di REST comune a HTTP ma ottimizzato per le applicazioni M2M (Figura 2.2). Sebbene CoAP possa essere utilizzato per rimodellare semplici interfacce HTTP in un protocollo più compatto, è più importante che offra anche caratteristiche per l'M2M, come la scoperta integrata, il supporto multicast e gli scambi

di messaggi asincroni.

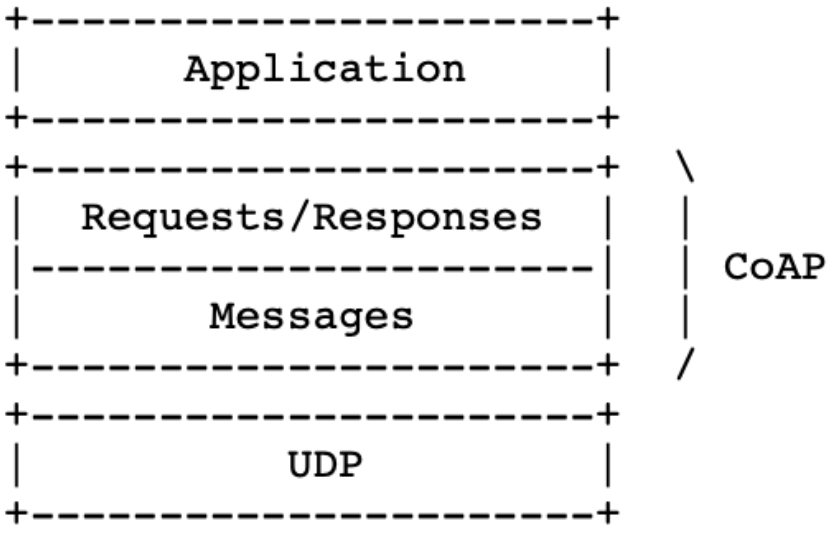


Figura 2.2: Abstract Layering of CoAP

CoAP ha le seguenti caratteristiche principali:

1. Protocollo web che soddisfa i requisiti M2M in ambienti vincolati
2. Legame UDP con affidabilità opzionale che supporta richieste unicast e multicast.
3. Scambi di messaggi asincroni.
4. Basso overhead di intestazione e complessità di analisi.
5. Supporto per URI e Content-type.
6. Semplici funzionalità di proxy e caching.

7. Una mappatura HTTP stateless, che consente di costruire dei proxy che forniscono accesso alle risorse CoAP tramite HTTP in modo uniforme o di realizzare interfacce HTTP semplici interfacce da realizzare in alternativa su CoAP (Figura 2.3)

8. Legame di sicurezza con Datagram Transport Layer Security (DTLS).

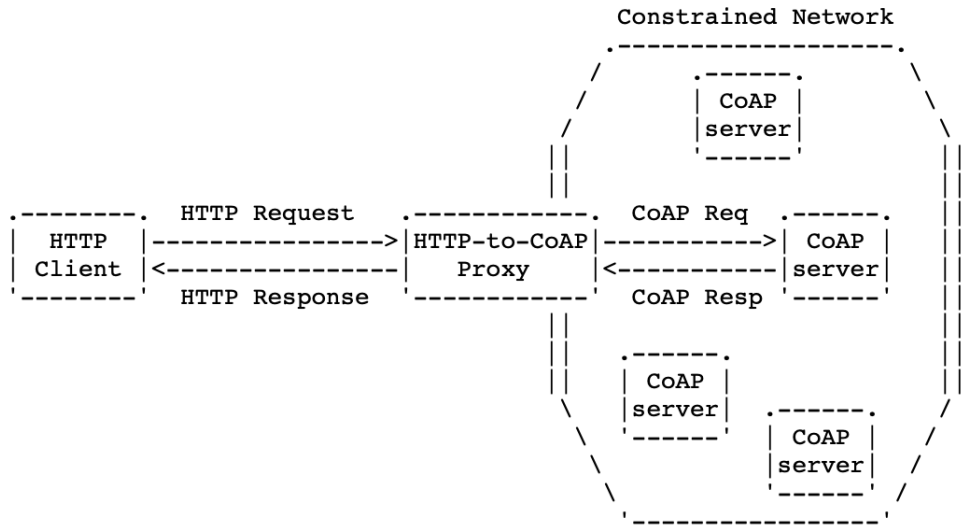


Figura 2.3: HTTP-To-CoAP Proxy

2.5 ROHC

Il protocollo Robust Header Compression (ROHC) fornisce un concetto di compressione delle intestazioni efficiente, flessibile e a prova di futuro. È stato progettato per funzionare in modo efficiente e robusto su varie tecnologie di collegamento con caratteristiche diverse. ROHC è un tipo di algoritmo

per comprimere l'intestazione di vari pacchetti IP. Nel caso di IPv4, la dimensione dell'intestazione IP non compressa è di 40 byte e nel caso di IPv6, la dimensione dell'intestazione IP non compressa è di 60 byte. Se si tratta di una normale applicazione a pacchetto, come il trasferimento di file o la navigazione, non sarebbe un grosso problema, poiché la dimensione dei dati trasferiti sarebbe molto grande rispetto a quella dell'intestazione. Quindi l'overhead creato dall'intestazione IP non sarebbe un grosso problema. Ma in alcune applicazioni (ad esempio VoIP, messaggi brevi, giochi, ecc.) le dimensioni dei dati trasferiti tendono a essere piccole e generano transazioni molto frequenti; in questo caso l'overhead creato dall'intestazione IP diventa molto grande. In questo caso, sarebbe un grande vantaggio se si riuscisse a trovare un metodo per ridurre le dimensioni dell'intestazione IP e ROHC è uno di questi metodi definiti da RFC 3095. Il tasso di compressione ideale di ROHC è quello di ridurre la dimensione dell'intestazione (40 o 60 byte nella dimensione originale) a solo 1 o 2 byte. L'idea di base di questo metodo di compressione è piuttosto semplice e può essere descritta come segue (figura 2.4).

1. All'inizio di una sessione (lo stato di iniziazione), il trasmettitore e il destinatario inviano l'intestazione completa senza alcuna compressione.

2. Dalla fase 1), sia il trasmettitore che il destinatario estraggono tutte le

informazioni dall'intestazione e le memorizzano.

3. Dopo la transazione iniziale, il trasmettitore invia solo le informazioni diverse dall'intestazione scambiata nella transazione iniziale. (Poiché molte informazioni dell'intestazione non cambiano durante l'intera sessione, la dimensione della parte che cambia diventa molto piccola. Quindi, trasmettendo solo la parte che cambia si ottiene un effetto simile alla compressione dei dati).

4. Eseguire ora la compressione vera e propria dei dati rimasti dopo il passo 3.

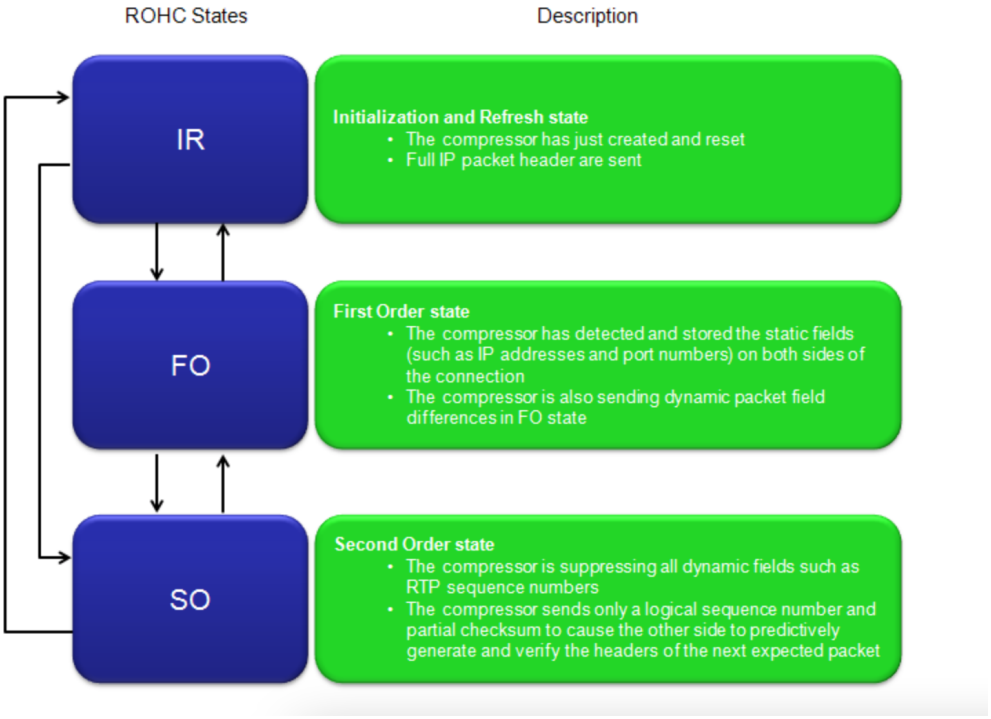


Figura 2.4: Stati di ROHC in compressione

Capitolo 3

Protocollo HTTP Compressed (HTTPC)

In questo capitolo è presentato l'algoritmo di codifica su cui si basa il funzionamento di HTTPC , noto come algoritmo V [31] (sezione 3.2), il quale è una versione adattiva della codifica di Huffman [32]. La tecnica di compressione utilizzata rientra nella categoria *lossless*.

3.1 Codifica di Huffman

La codifica di Huffman è un algoritmo che restituisce un codice prefisso ottimo o sub-ottimo a partire dalla distribuzione di probabilità dei simboli da codificare. La codifica è ottima se le probabilità dei simboli sono potenze

(negative) di base 2, sub-ottima altrimenti. Questo comportamento dipende dal fatto che i codici di Huffman (come qualsiasi altra codifica prefisso) associano un numero intero di bit ad ogni simbolo. Se l'autoinformazione di quel simbolo non corrisponde ad un numero intero (ad esempio, se ha probabilità $p = 1/3$, e quindi autoinformazione pari a $-\log(1/3) = 1.585$ bit) sarà impossibile per la codifica di Huffman raggiungere il numero di bit ottimale perché non intero. Tuttavia, la codifica di Huffman può essere generalizzata ad alberi non binari: supponendo di utilizzare codifiche a x lettere, la codifica di Huffman sarà ottima se le probabilità dei simboli sono potenze (negative) di base x, perché la loro autoinformazione misurata tramite logaritmo in base x sarà un numero intero. Ad ogni modo, dato che i calcolatori operano in bit, x dovrebbe essere una potenza di due affinché la codifica sia utilizzabile. Ciò significa che si potrà sempre trovare una codifica binaria equivalente.

La codifica di Huffman consente di creare un albero di codifica seguendo un approccio bottom-up: per prima cosa si memorizzano in un array i simboli da codificare, insieme alla loro probabilità o numero di occorrenze. Ad ogni simbolo è associato un nodo che alla fine dell'algoritmo sarà una foglia dell'albero di codifica. Caricato l'array, si prendono i due nodi con probabilità minore, si eliminano dall'array e si crea un terzo nodo che sarà loro padre: a questo nodo viene associata una probabilità che sarà la somma di quella dei figli e poi lo si reinserisce nello stack. Si ripete il procedimento

finché lo stack non conterrà un unico nodo, il quale sarà la radice dell'albero di codifica. Se si usano n lettere, ad ogni passaggio si prendono i n nodi con probabilità minore e si uniscono sotto un unico nodo padre. In Figura 3.1 è presentato un esempio di utilizzo della codifica di Huffman sul messaggio "CODIFICA". Ovviamente, affinché un messaggio possa essere decompresso, l'albero di codifica deve essere noto in fase di decodifica.

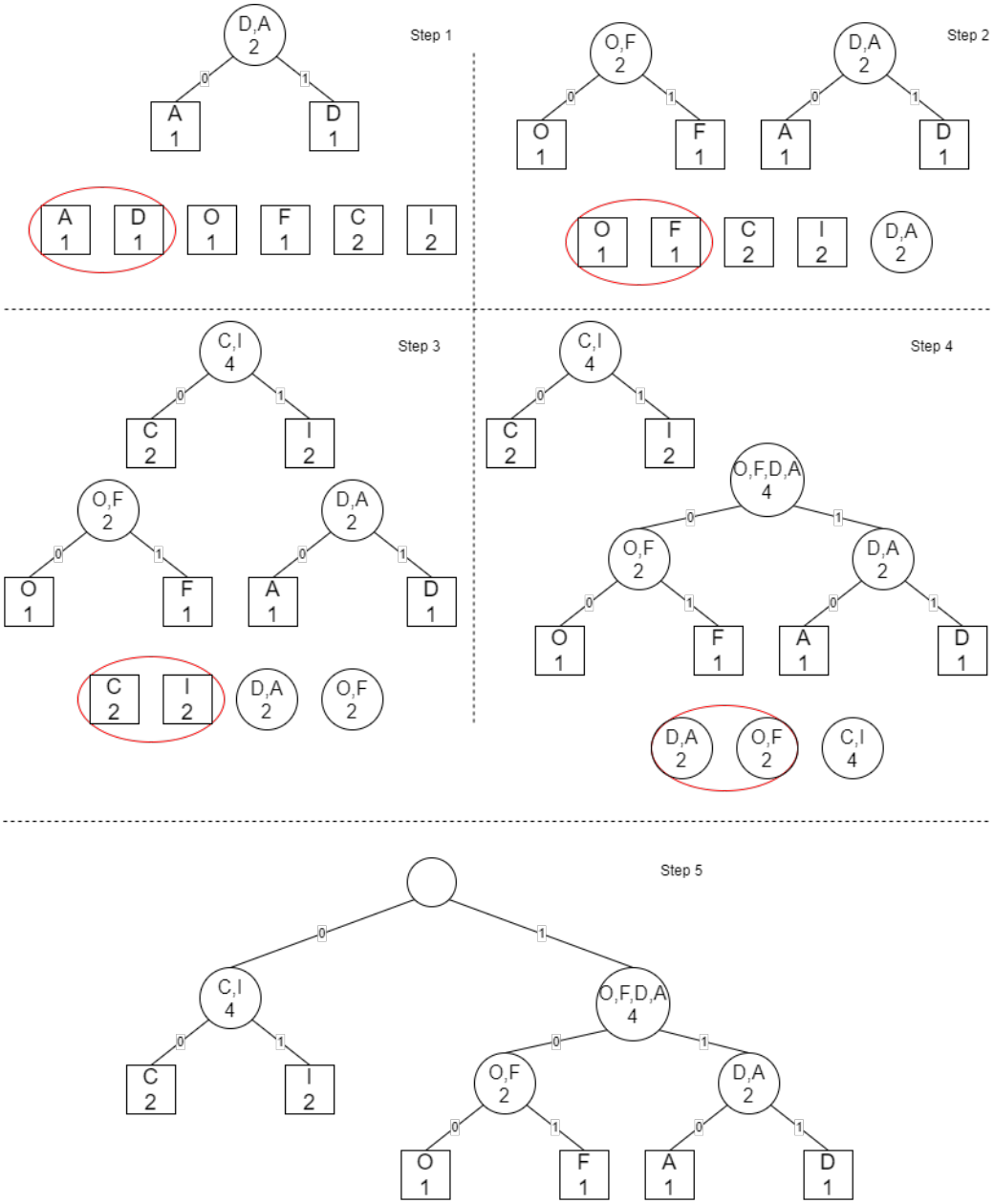


Figura 3.1: Passaggi per trovare una codifica di Huffman per la stringa "CODIFICA"

3.2 Algoritmo V

La codifica di Huffman, sebbene permetta di raggiungere dimensioni dei messaggi codificati molto vicine al limite teorico dato dal teorema di codifica di sorgente, ha il grande svantaggio di richiedere la conoscenza a priori della distribuzione probabilistica (o il numero di occorrenze) dei simboli da codificare. Dato che questa condizione non può essere sempre soddisfatta sono stati prodotti algoritmi che restituiscono codifiche ottime in modo adattivo. L'albero di codifica viene aggiornato man mano che i simboli si presentano e, in fase di decodifica, può essere ricostruito dal messaggio codificato, in modo analogo agli algoritmi Lempel-Ziv. Questa caratteristica porta all'ulteriore vantaggio di non dover memorizzare l'albero di codifica. Tuttavia, gli algoritmi adattivi raggiungono tassi di compressione leggermente inferiori rispetto ai non adattivi per due motivi: il primo è che ogni occorrenza di nuovo valore non viene codificata, ma anzi gli viene aggiunto in testa una parola riservata ad indicare l'occorrenza di un nuovo valore. Il secondo motivo è che la distribuzione dei simboli può essere disomogenea, fatto che inizialmente porta ad assegnare le parole più brevi a simboli che alla fine del messaggio saranno meno frequenti di altri, provocando un'inefficienza.

L'algoritmo V fu proposto da Jeffrey Vitter nel 1987 come miglioramento dell'algoritmo FGK, uno dei primi ad implementare una codifica di Huffman

adattiva. L'idea che contraddistingue l'algoritmo V è stabilire delle convenzioni nel processo di costruzione dell'albero di codifica che ne semplificano le modifiche.

3.3 Strategia di compressione

A differenza dei protocolli 6LowPAN e ROHC, che possono contare sulla ripetizione degli headers dovuta alla frammentazione operata dai protocolli di livello di rete e trasporto, il protocollo HTTPC dovrà comprimere un unico header per messaggio. Per questo motivo la tecnica di compressione sarà fondamentalmente diversa da quelle sfruttate a livelli inferiori.
Una qualsiasi tecnica di compressione punta a sfruttare le ridondanze del messaggio. Il primo passo per definire la strategia di compressione di HTTPC è dunque identificare le ridondanze presenti nei messaggi HTTP.

- Ridondanze spaziali: tipiche dei file di testo e delle immagini, in HTTP/1.1 si presentano come ripetizioni di uno stesso carattere all'interno del messaggio.

- Ridondanze temporali: tipiche dei video e dei suoni, in HTTP/1.1 si presentano come ripetizioni di uno stesso campo in messaggi successivi.

Le ridondanze spaziali saranno sfruttate attraverso una codifica di Huffman dei caratteri, dunque è necessario costruire un modello probabilistico della distribuzione dei caratteri. Attraverso il formato ISO-8859-1 (standard http per il tipo MIME text) ogni carattere è rappresentato tramite un byte, quindi questa codifica statica userà i bytes come simboli.

Le ridondanze temporali, invece, saranno sfruttate mantenendo in memoria una struttura dati in cui saranno memorizzate parti dei messaggi precedenti. Ad ognuna di esse sarà associata una parola che verrà usata per codificare quella parte la prossima volta che si presenterà. Le parti memorizzate costituiranno quindi i simboli di questa codifica dinamica.

Il protocollo HPACK usa una strategia simile: definisce una codifica di Huffman ottenuta tramite la distribuzione probabilistica dei caratteri che compongono un messaggio HTTP e mantiene i campi più frequenti in 2 code di lunghezza massima nota, una per le richieste e una per le risposte. L'indice di coda costituisce la parola con la quale il campo è codificato, quindi ogni parola codificata avrà lunghezza fissa pari al logaritmo della lunghezza massima della coda. Le code sono aggiornate durante la comunicazione secondo una filosofia FIFO.

Una coda, tuttavia, non è lo strumento ideale perché non riesce a rappresentare eventuali interdipendenze tra campi: ad esempio, se una risposta possiede l'header *Content-Length*, che indica la lunghezza del body, allora

è molto probabile che contenga anche l'header *Content-Type* che specifica il tipo di body trasmesso. Viceversa, se il primo header è assente allora è molto probabile che manchi anche il secondo. Se questi due headers fossero mantenuti in una coda e il primo non fosse presente nel messaggio, si sprecherebbe una codifica per un header (il secondo) che molto probabilmente non si presenterà. Lo stesso ragionamento può essere fatto anche tra campi diversi. Ad esempio, una richiesta GET probabilmente conterrà headers diversi da una richiesta POST, anche se entrambe le richieste fossero effettuate sul medesimo URL. L'uso di una coda implicherebbe la memorizzazione di tutti gli headers, quindi nel caso in cui arrivasse una richiesta GET alcune codifiche sarebbero occupate da headers che in realtà non si sono mai visti in una richiesta GET e che probabilmente non compariranno.

HTTPC propone l'uso di alberi come strutture dati per memorizzare i campi trasmessi, i quali saranno aggiornati durante la comunicazione tramite algoritmo V. É fondamentale che entrambi gli interlocutori mantengano i propri alberi sincronizzati. L'aggiornamento avviene dopo ogni trasmissione di un campo e costituisce la principale fonte di ritardi di comunicazione. La necessità di sincronizzazione tra i contesti degli interlocutori richiede una gestione dei casi di errore. Gli errori di trasmissione saranno gestiti dal protocollo di livello trasporto: nello scenario di riferimento, TCP e QUIC implementano la semantica exactly-once, che garantisce che ogni messaggio

sia processato a livelli superiori una sola volta. Si dovranno quindi gestire i casi in cui il messaggio viene trasmesso correttamente ma la connessione cade prima che il destinatario sia riuscito a leggerlo e ad aggiornare il suo contesto.

Una semplice soluzione a questo problema può essere contare i messaggi scambiati durante una connessione, memorizzare il contatore e, non appena una nuova connessione con lo stesso interlocutore viene aperta, confrontare il proprio contatore con quello dell'interlocutore. Se i contatori sono uguali i contesti sono sincronizzati e la comunicazione può proseguire, altrimenti i contesti saranno cancellati e la comunicazione avverrà come se fosse la prima volta. La grandezza del contatore deve essere tarata in funzione di quello che si ritiene essere il numero massimo di messaggi che possono essere inviati alla volta. Se il client attende sempre una risposta prima di inviare la richiesta successiva, il contatore può essere anche di un solo bit, in quanto la desincronizzazione massima è di un solo messaggio. Viceversa, se il client tende a mandare tutte le richieste insieme, in caso di errore la desincronizzazione può essere di un numero variabile di messaggi: un contatore troppo piccolo potrebbe non rilevare la desincronizzazione, mentre un contatore troppo grande aumenterebbe l'overhead di comunicazione. In generale, un contatore di un byte dovrebbe costituire un buon compromesso.

Dato un messaggio in chiaro, esso viene analizzato un campo alla volta: se

il campo è presente nell'albero allora viene codificato con la parola associata dall'albero, il quale deve poi essere aggiornato, altrimenti viene codificato secondo la codifica statica definita in HPACK e inserito come nuova foglia dell'albero di codifica. Ad ogni foglia è associato un altro albero che servirà alla codifica del campo successivo. HTTPC definisce una gerarchia tra i campi: l'albero di codifica di un campo di grado n può associare alle sue foglie solo alberi di codifica di campi di grado n o n+1. La struttura dati risultante è quindi un grande albero che costituisce il contesto della comunicazione, sostituendo le code di HPACK. La frammentazione dell'albero di contesto in sottoalberi per campi porta due vantaggi fondamentali:

1. Se ogni sottoalbero contiene pochi elementi rispetto al numero di elementi totali mantenuti dall'albero di contesto, le codifiche associate ad un campo saranno più brevi di quelle ottenute per mezzo di code. Inoltre il tempo di accesso ai nodi dell'albero sarà inferiore, .

2. Durante la codifica e la decodifica non è necessario mantenere in memoria RAM tutto l'albero di contesto ma solo il sottoalbero relativo al campo in elaborazione. Ciò è molto utile se la memoria disponibile è limitata.

Tuttavia, la frammentazione implica anche l'impossibilità di parallelizzare la codifica e la decodifica dei campi di un messaggio in quanto non si conosce

l'albero da usare per un campo finché non si codifica o decodifica il campo precedente.

3.4 Campi

Basandosi sulla definizione dei campi di HTTP/1.1 [], HTTPC definisce 6 campi per le richieste e 4 per le risposte (3.1). Essi sono suddivisi in 3 tipi e per ciascun tipo è adottata una tecnica di compressione diversa. Il campo body non è classificato in nessuno dei tre tipi perché HTTPC comprime solo l'intestazione di un messaggio HTTP. Dato un messaggio HTTP, il protocollo HTTPC dovrà per prima cosa scomporlo nei suoi campi costituenti che saranno poi compressi in ordine gerarchico.

3.4.1 Tipo 0

I valori dei campi di tipo 0 costituiscono le foglie dei sottoalberi, quindi la codifica dei tipo 0 è una semplice sostituzione con la parola data dal sottoalbero. I campi HTTPC di tipo 0 sono Version, Method e Status.

3.4.2 Tipo 1

L'unico campo di questo tipo è URI. Esso è una concatenazione di valori che iniziano o terminano con un carattere noto. La codifica di tipo 1 è trattata

Grado gerarchia	Richiesta	Risposta
1	Version	Version
2	Method	Status
3	URI	Headers
4	Parameters	Body
5	Headers	
6	Body	

Tabella 3.1: Campi HTTPC

come una sequenza di codifiche di tipo 0, quindi un sottoalbero di tipo 1 è in realtà ottenuto per composizione di alberi di tipo 0, proprio come l'albero di contesto.

3.4.3 Tipo 2

I campi di tipo 2 sono composti da una lista di coppie nome-valore. Per ogni elemento della lista si applica una codifica di tipo 0 prima sul nome, poi sul valore. Come per il tipo 1, un sottoalbero di tipo 2 è una composizione di alberi di tipo 0 che alternativamente codificano nomi o valori. I campi di tipo 2 sono Parameters e Headers.

3.5 Sessioni

L'albero di contesto serve a mantenere informazioni relative la sessione di comunicazione tra due interlocutori. Il tempo di vita della sessione coincide con il tempo di vita dell'albero di contesto. Idealmente, ad ogni nuova

apertura di una connessione TCP andrebbe creato un nuovo albero di contesto. Questo è il motivo per cui si è scelto di basare HTTPC su HTTP/1.1 e non HTTP/1.0: nella prima versione di HTTP si prevedeva che su una connessione TCP viaggiassero solo una richiesta ed una risposta. In questo modo HTTPC non avrebbe potuto sfruttare la sua codifica adattiva. HTTP/1.1 introduce invece la possibilità di inviare più richieste e ricevere più risposte attraverso la stessa connessione. L'albero di contesto potrebbe poi essere inizializzato con valori di default in modo da aumentare l'efficienza della codifica dei primi messaggi.

Ad ogni modo, non è detto che su una stessa connessione siano trasmessi molti messaggi, quindi una valida alternativa è rendere persistente l'albero ottenuto al termine di una connessione e quindi estendere il tempo di vita della sessione a più connessioni TCP. Nello scenario di riferimento, l'albero di contesto può essere associato all'indirizzo IP dell'interlocutore. Se gli indirizzi non sono fissi è necessario prevedere in fase di connessione un protocollo di autenticazione che permetta di individuare il contesto da utilizzare.

Se si usano contesti persistenti è importante controllare la crescita del contesto. Man mano che l'albero di contesto cresce in dimensioni si ha:

- un peggioramento del tasso di compressione, in quanto le parole del codice diventano più lunghe;

- un peggioramento del ritardo introdotto, in quanto aumentando il numero di nodi aumenta il tempo richiesto per navigare l'albero e quindi il tempo necessario al suo aggiornamento;

- una maggiore richiesta di spazio disco per memorizzare il contesto.

La dimensione massima raggiungibile dal contesto deve essere calibrata in funzione delle prestazioni richieste e delle risorse disponibili secondo lo scenario di applicazione. Può essere definita impostando un limite allo spazio disco utilizzabile oppure alla dimensione e al numero dei sottoalberi. Nello scenario di riferimento, in cui i messaggi trasmessi hanno poche variazioni dei campi, tale controllo potrebbe anche non essere implementato perché il contesto non dovrebbe crescere così tanto da peggiorare sensibilmente le prestazioni.

Il formato di memorizzazione del contesto utilizzato da un nodo è invisibile agli altri. Ogni nodo può quindi usare il formato che preferisce e non è necessario che due interlocutori adottino lo stesso formato durante una sessione.

Capitolo 4

Scenario di riferimento in ambito IoT

In questo capitolo descriveremo lo scenario IoT, le componenti software utilizzate e l'analisi del comportamento del protocollo di compressione HTTPC (capitolo 3) ai diversi scenari IoT realizzati.

L'idea alla base è la simulazione di un'applicazione meteo che invia messaggi con dati contenenti temperatura, pressione e umidità. Per la realizzazione dello scenario IoT è stato creato uno script python che simulava le richieste HTTP di tipo GET o POST. Esempio di trasmissione HTTP in figura 4.1. Tutto questo è stato possibile facendo uso di una liberia python *Requests* (`https://requests.readthedocs.io/en/latest/`) facilitandone la realizzazione e l'invio delle sudette richieste.

Ogni singola richiesta è stata inviata ad un semplice *HTTP Request & Response Service "httpbin.org"* (`https://httpbin.org`). Così il lato client era rappresentato dallo script python e il lato server con annesse risposte da *httpbin.org*. Per il monitoraggio delle richieste e delle risposte è

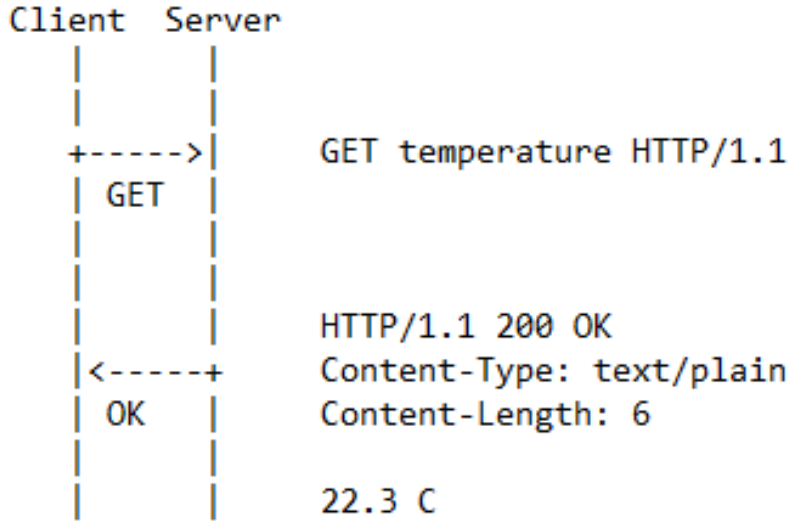


Figura 4.1: Esempio di trasmissione HTTP

stato utilizzato *Fiddler Everywhere* (`https://www.telerik.com/fiddler/fiddler-everywhere`). *Fiddler Everywhere* è un software per analisi di protocollo o packet sniffer utilizzato per la soluzione di problemi di rete, per l'analisi e lo sviluppo di protocolli o di software di comunicazione, possedendo tutte le caratteristiche di un analizzatore di protocollo standard. Il software di packet sniffer per ogni richiesta HTTP restituisce un file .txt contenente header e body e fa lo stesso anche per la corrispettiva risposta dal

server. Quindi avremo che il numero dei file di testo sarà il doppio rispetto alle richieste effettuate attraverso lo script python. Prima di essere utilizzati, i file di testo restituiti dal software hanno bisogno di passare per un lavoro di formattazione. Per ogni singolo file di testo, che sia di richiesta o risposta, sarà mantenuto solo l'header e salvato in un nuovo file di testo e a loro volta ancora salvati un una cartella. Questa cartella verrà data in pasto all'eseguibile del protocollo di compressione HTTPC che ne farà il suo lavoro di compressione.

Il protocollo HTTPC è stato valutato in 3 scenari di applicazione, tutti e 3 costituiti da un dataset di 2400 messaggi. Ogni scenario si differenzia dall'altro per la frequenza di cambiamento dei dati nelle richieste. Gli scenari sono : Cambio ad ogni richiesta, quindi un cambio costante; Cambio ogni 600 messaggi; Cambio ogni 24 messagi. Per ogni scenario, i tassi di compressione sono espressi secondo la formula

$$1 - \frac{\text{bit messaggio codificato}}{\text{bit messaggio originale}} \tag{4.1}$$

4.1 Cambio costante

In questo scenario è stato realizzato e ipotizzato un possibile cambiamento di tutti i dati (di temperatura,pressione e umidità) o solo di alcuni ad ogni richiesta dello script python, cercando di creare uno scenario quanto più simile alla realtà. Il tutto è stato pensato utilizzando la funzione random in python scegliendo un valore tra [-1,0,1] e sommandolo al valore precedente, cosi da aumentare o diminuire di 1 il valore o facendolo restare lo stesso. Come si può vedere nella figura 4.2 il protocollo HTTPC ha risposto ad ogni richiesta sempre meglio, adattandosi sempre di più per quanto riguarda il *compression rate*. Nonostante la diversità del'header per ogni richiesta e la possibilità di avere dei dati sempre diversi dal messaggio precendete, non hanno inficiato negativamente sul comportamento del protocollo che è riuscito ad adattarsi sempre meglio con l'avanzare delle richieste dal punto di vista della compressione.

Figura 4.2: Tassi di compressione medi calcolati tramite finestra scorrevole di 10 messaggi

4.2 Cambi ogni 600 messaggi

L'idea alla base è sempre la stessa, in questo scenario si ha sempre un dataset di 2400 messaggi tra richieste e risposte. L'unica differenza è che, se l'header da un messaggio all'altro varia perchè richieste e risposte diverse, il dato non varia sempre ma è stato progettato che cambiasse dopo un numero di richieste-risposte ben definito (600 messaggi). Come si vede dalla figura 4.3, il comportamento del protocollo è ottimale e varia in un intervallo dove il *compression rate* è compreso tra 0,65-070 e ha picchi inferiori solo nei

casi dove è stato deciso che c'era la possibilità che il dato cambiasse. Seppur a priori era previsto che il cambiamento del dato portasse ad un valore inferiore del *compression rate*, si può notare l'adattamento del protocollo al cambiamento migliorando la risposta alle due possibili variazioni avendo un *compression rate* migliore rispetto a quello della prima variazione.

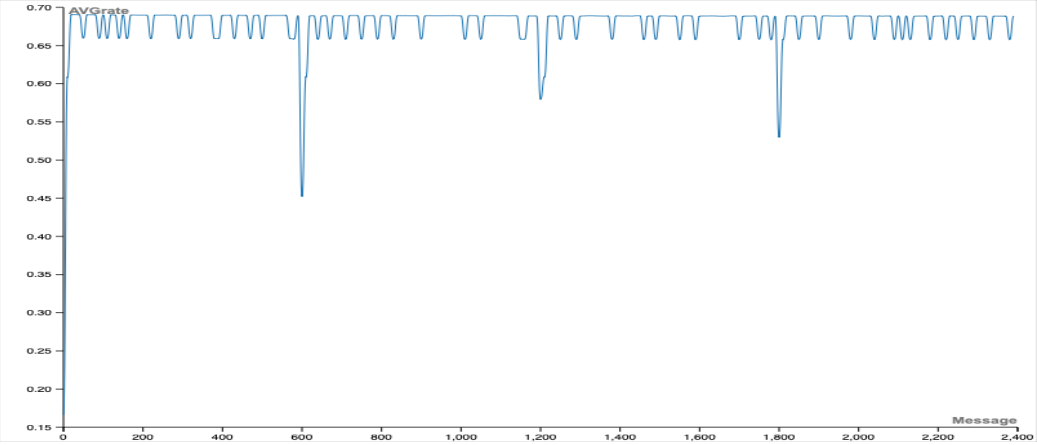


Figura 4.3: Tassi di compressione medi calcolati tramite finestra scorrevole di 10 messaggi

4.3 Cambio ogni 24 messaggi

Anche per quest ultimo scenario le caratteristiche di base sono le stesse degli scenari presentati precedentemente. Ovvero dataset costituito da 2400 messaggi tra richieste-risposte con dati di valori metereologici contenuti negli header dei messaggi.

A differenza dei precedenti è stata scelta un'altra frequenza di cambiamento

per vedere come si comportasse il protocollo in questa situazione. É stato scelto di dare la possibilità al dato di cambiare ogni 24 messaggi. É stato scelto questo passo perchè si voleva valutare il *compression rate* con un passo molto vicino alla frequenza di campionamento per il calcolo della media del *compression rate*. Il protocollo si è comportato allo stesso modo dei casi precedenti, ovvero adattandosi (figura 4.4). I valori peggiori sono nella fase iniziale con valori nell'intervallo [0,35-0,40], cioè quando il dizionario si sta ampliando, seppur nelle richieste successive ci sono delle variazioni di dato il protocollo non raggiunge mai un *compression rate* peggiore della fase iniziale con a volte anche ottime risultati nell'intervallo [0,65-0,70].

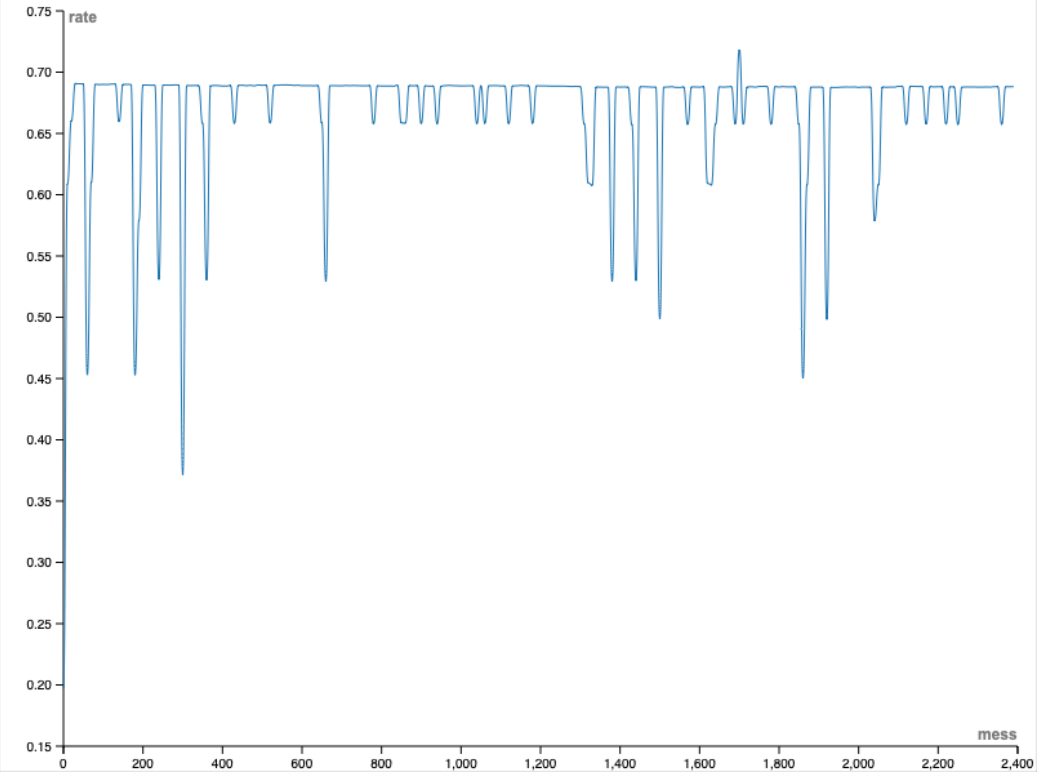
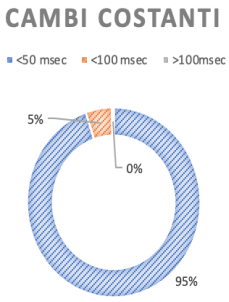


Figura 4.4: Tassi di compressione medi calcolati tramite finestra scorrevole di 10 messaggi

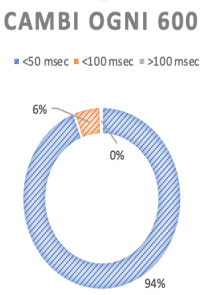
4.4 Tempo di elaborazione

Il *compression rate* non è la sola caratteristica che è stata valutata, ma anche il tempo di elaborazione per la compressione del header di ogni messaggio. Questo perchè stiamo sempre lavorando in scenari IOT, e come sappiamo essendo *constrained devices*, bisogna valutare diversi fattori e rispettare vincoli tal volte anche scendendo a compromessi. Da come si evince dalla figura 4.5, la maggior parte delle richieste, in tutti e tre gli scenari, vengono elebaorate in un intervallo che ha come massimo 50 msec. Con precisione il 94/95%

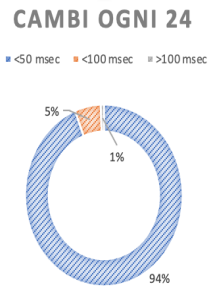
delle richieste è inferiore ai 50 msec. Solo il 5% tra i 50 msec e i 100 msec e l'1% eleborate in più di 100msec.



(a) Cambiamenti ad ogni richiesta



(b) Cambiamenti ogni 600 richieste



(c) Cambiamenti ogni 24 richieste

Figura 4.5: Tempi di elaborazione

Capitolo 5

Implementazione di Hpack

Come trattato nel capitolo 2.2, HTTP 2.0 supporta un nuovo algoritmo di compressione dell'intestazione dedicato, chiamato HPACK.
É stato realizzato un ambiente che emula richieste di tipo HTTP 2.0 e ad ogni richiesta è stato applicato l'algortimo di compressione dell'intestazione *Hpack*. Come per lo scenario IoT del Capitolo 4, il tutto è stato realizzato tramite uno script python con l'utilizzo di diverse liberie:

1. La libreria *Requests* per l'invio di richieste al server di test "*http-bin.org*".

2. La libreria di *Hpack* per la codifica degli headers (`https://python-hyper.org/projects/hpack/en/latest/`)

Le intestazioni delle richieste ottengono una compressione migliore, a causa di una duplicazione molto più elevata nelle intestazioni, perciò è stato scelto di lavorare sulle intestazioni delle risposte cosi da escludere elementi duplici. Grazie alla libreria requests con *"r.headers"*, che permette di accedere facilmente ad ogni singolo attributo dell'headers della risposta, è stato possibile creare un dizionario cosi da poter associare ad ogni campo il corrispettivo valore. A questo punto, il dizionario di ongi singola risposta è codificato dall'encoder della liberia python HPACK.

Come per HTTPC, la codifica dell'intestazione delle ripsoste da parte di HPACK è stata valutata per 3 scenari di un applicazione meteo, che si differenziavano tra loro nella frequenza della possibilità di cambiamento dei valori dell'invio delle richieste:

1. Richieste con cambio costante
2. Richieste con cambio ogni 600 messaggi
3. Richieste con cambio ogni 24 messaggi

5.1 Prestazioni

Per ogni scenario è stato valutato il comportamento della compressione dell'intestazione da parte di HPACK.É stato osservato il comportamento sia

generale che in media tramite una finestra di 10 compressioni di intestazioni di risposte. Come si puo vedere nelle tre figure 5.2, 5.4, 5.6, seppure gli scenari siano diversi, l'algoritmo di compressione in linea generale si comporta in maniera analoga in tutti e tre gli scenari. Nella parte destra di ogni figura vi è il grafico che riporta il comportamento generale e le prestazioni per ogni singola compressione, mentre nella parte di sinistra è riportato il comportamento in media tramite una finestra di 10 compressioni.

Tutti e 3 gli scenari sono costituiti da un dataset di 1200 messaggi di risposta (la metà rispetto al totale dei 2400 messaggi richiesta-risposta analizzati in HTTPC) Per ogni scenario, i tassi di compressione sono espressi secondo la formula

$$1 - \frac{\text{bit messaggio codificato}}{\text{bit messaggio originale}} \tag{5.1}$$

La compressione è ottimale in tutti e 3 gli scenari, come si può vedere dalle figure 5.2, 5.4, 5.6, hanno un picco massimo del *compression-rate* di 0,867 e la maggior parte delle richieste hanno un valore di *compression-rate* compreso tra 0,6 e 0,867. Cosa molto importante che si evince dal grafico generale di ogni figura, che questi tipi di algoritmi hanno sempre una fase di train per poi arrivare al massimo delle prestazioni. E che dopo 120/150 richieste elaborate l'algoritmo raggiunge una fase di saturazione dove le richieste successive saranno elaborate come se fossero le prime, perciò ci

sono dei picchi negativi cosi in basso. Ovviamente questi picchi negativi sono dati spuri e se ragionassimo in media, grafico sinistro di ogni figura, possiamo notare che le richieste vengono elebarate con un *compression-rate* in un intervallo [0,56-0,8].

5.1.1 Cambio costante

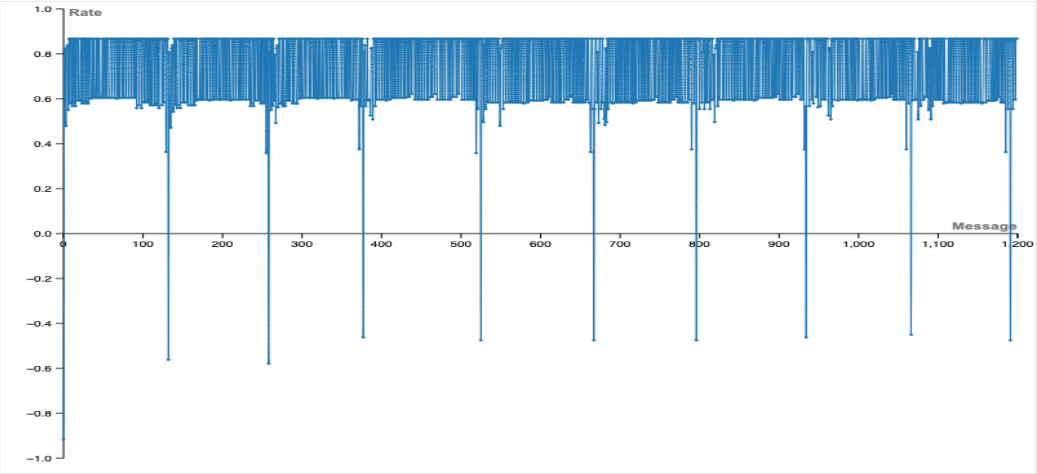


Figura 5.1: Cambio costante—compression-rate

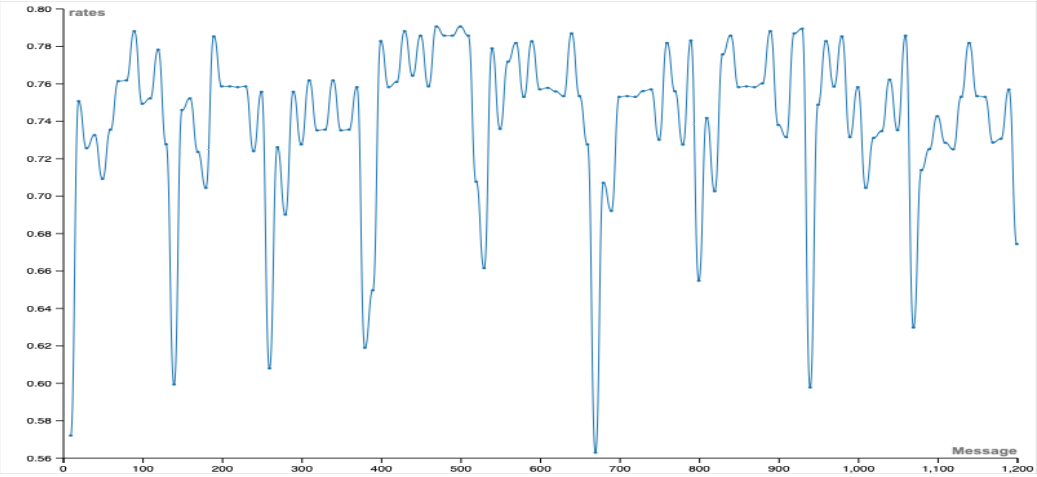


Figura 5.2: Cambio costante—Average compression-rate

5.1.2 Cambio ogni 600

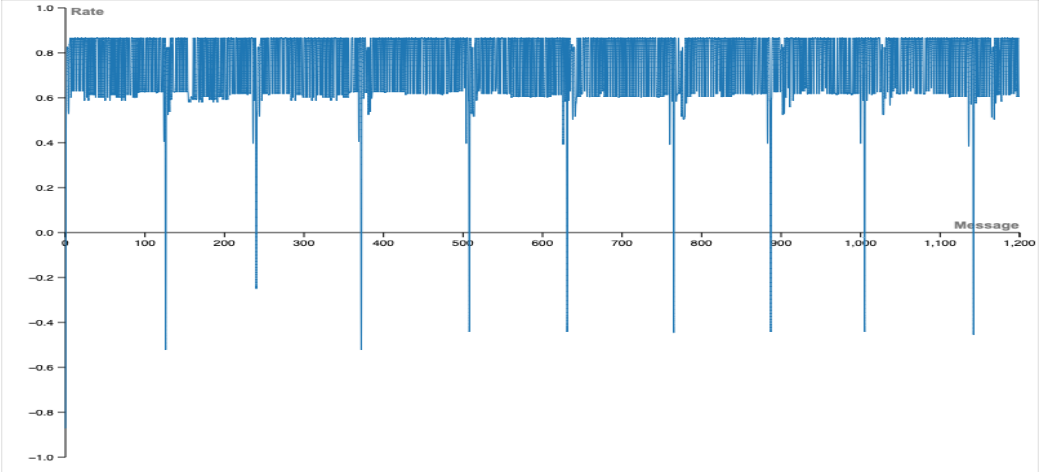


Figura 5.3: Cambio ogni 600—compression-rate

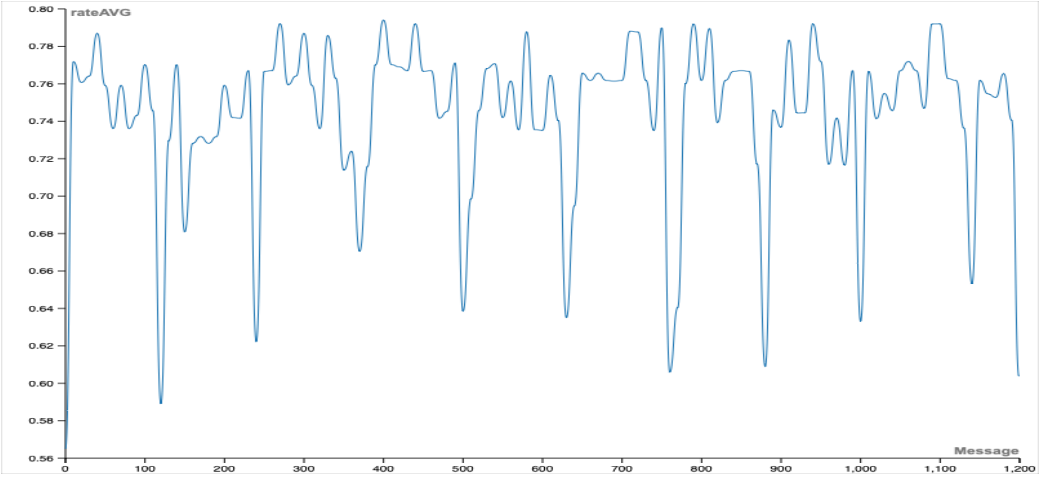


Figura 5.4: Cambio ogni 600—Average compression-rate

5.1.3 Cambio ogni 24

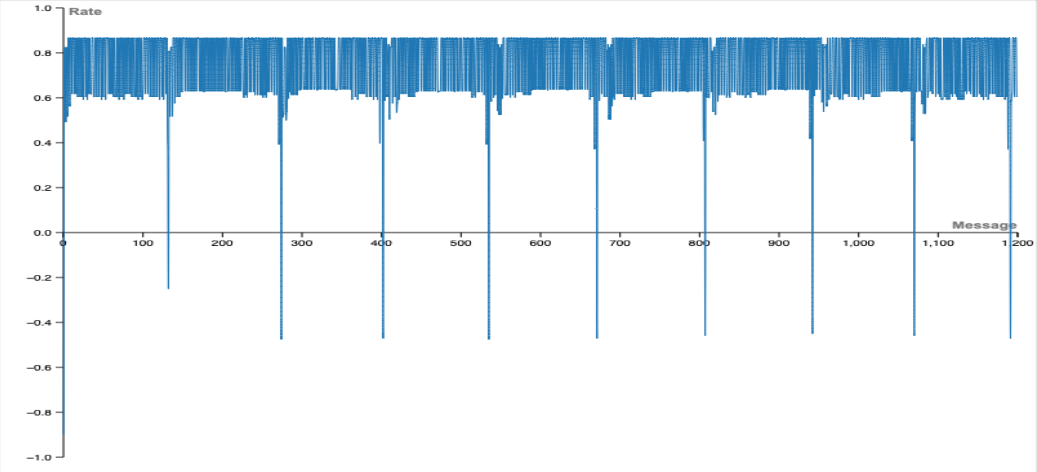


Figura 5.5: Cambio ogni 24—compression-rate

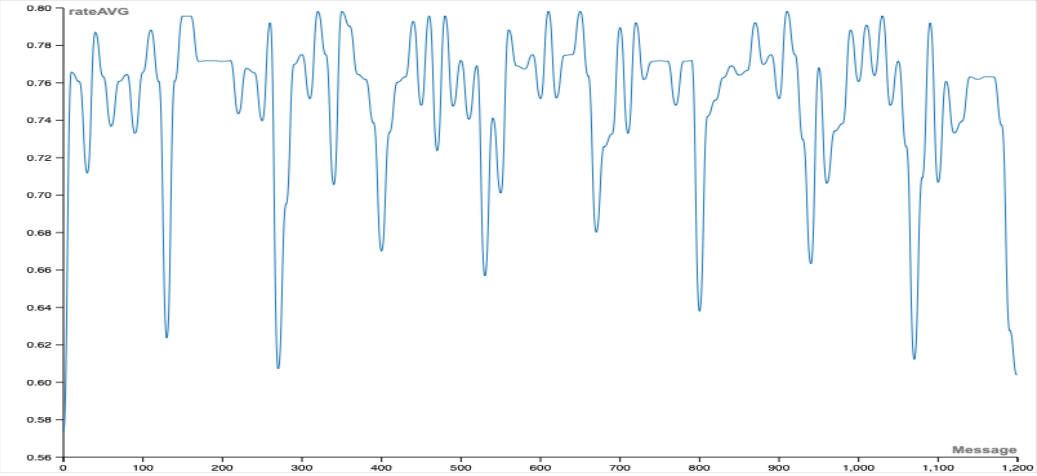


Figura 5.6: Cambio ogni 24—Average compression-rate

5.2 Confronto con HTTPC

Le tre figure :

1. Costanti 5.7
2. Ogni 600 5.8
3. Ogni 24 5.9

sono frutto del confronto tra le prestazioni(in media) dei due algoritmi di compressione delle intestazioni per i dataset costituiti dalle 1200 risposte di messaggi. Notiamo che HTTPC ha prestazioni che rientrano in un intervallo del *compression-rate* più piccolo rispetto ad HPACK. Per HTTPC la maggior parte delle intestazioni di risposta hanno un valore compreso tra lo 0,3 e lo 0,4, quindi una differenza di 0,1 tra caso milgiore e peggiore. Mentre per HPACK ha una differenza tra valore milgiore e peggiore di 0,25 circa. Però è anche vero che HPACK ha un *compression-rate* migliore rispetto ad HTTPC anche confrontando il suo caso peggiore con il caso migliore di HTTPC.

Cambio costante

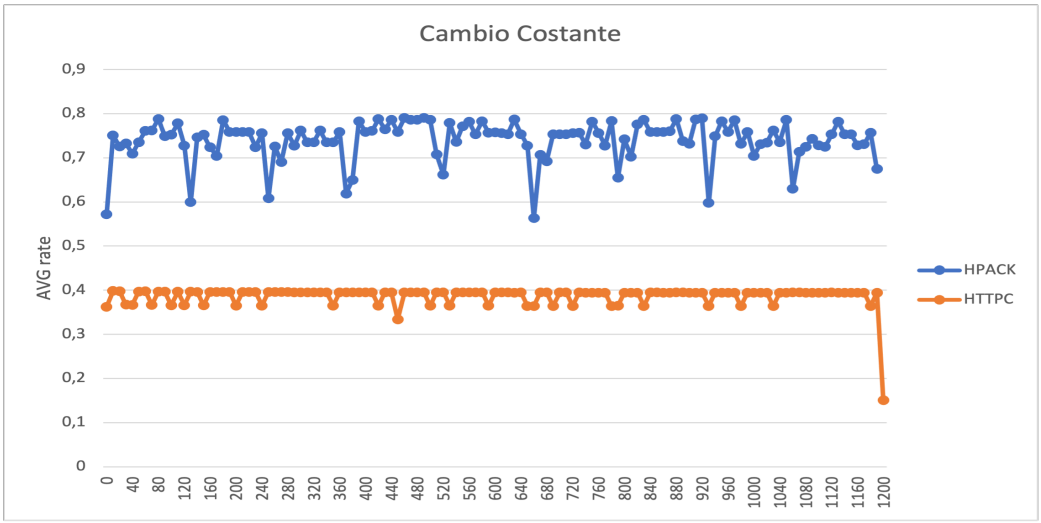


Figura 5.7: COSTANTI—Hpack e HTTPC a confronto

Cambio ogni 600

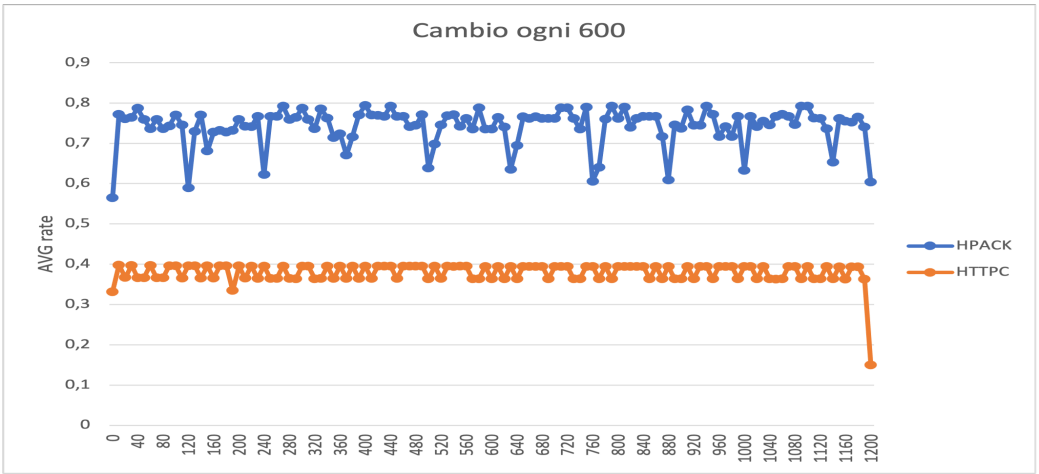


Figura 5.8: OGNI 600—Hpack e HTTPC a confronto

Cambio ogni 24

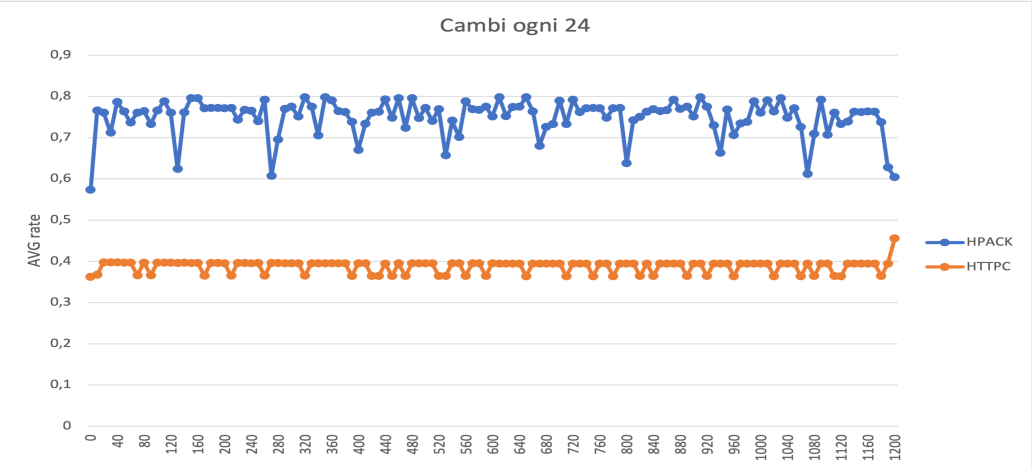


Figura 5.9: OGNI 24—Hpack e HTTPC a confronto

Capitolo 6

Conclusioni

L'obiettivo di questa tesi è stato proporre una strategia di riadattamento a scenari IoT di protocolli esistenti, basata su compressione adattiva dei messaggi trasmessi. La grande eterogeneità degli scenari IoT rende difficile trovare una soluzione che ottenga sempre risultati ottimali, di conseguenza si è venuto a creare un vasto ecosistema di protocolli particolarmente ampio a livello applicazione, in quanto il protocollo HTTP, standard internet di livello applicazione, si è rivelato inadeguato nella quasi totalità degli scenari IoT. Quello che ci ha spinto a lavorare ad una soluzione è che in questi scenari i dispositivi sono wireless, alimentati a batteria e la fonte di maggior consumo energetico è costituita dell'uso dell'antenna: maggiore il numero di pacchetti trasmessi, maggiore il consumo. Proprio questo ha determinato un fallimento per HTTP, che tende a trasmettere messaggi lunghi, che saranno

frammentati in molti pacchetti a livello trasporto e rete.

La soluzione presentata è un prototipo di protocollo di compressione di messaggi HTTP/1.1, chiamato HTTP Compressed o HTTPC. I vantaggi derivanti dall'uso di HTTPC rispetto a CoAP sono una maggior compressione dei messaggi, che quindi provoca un risparmio energetico, e una piena compatibilità con i protocolli basati su HTTP che non è possibile ottenere con CoAP. La soluzione è stata valutata in tre scenari IoT per valutarne le prestazioni (Capitolo 4) e confrontato con un altro protocollo che ha lo stesso scopo. In particolare si nota che HTTPC abbia delle performance inferiori rispetto ad altri algoritmi che svolgono lo stesso lavoro, in particolare HPACK di HTTP 2.0. D'altro canto però, molti siti non sono ancora su HTTP 2.0 quindi è impensabile che allo stato attuale delle cose si possa valutare uno scenario IoT implementato con scambi di messaggi su HTTP 2.0. Da ciò si evince che, seppure HTTPC, valutato sullo stesso scenario, abbia prestazioni nettamente inferiori ad HPACK, è l'unico che attualmente può essere preso in considerazione per la realizzazione di uno scenario IoT in quanto HTTP 1.1 è attualmente utilizzato nella gran parte dei siti internet.

Bibliografia

[1] F. Michelinakis, A. Al-selwi, M. Capuzzo, A. Zanella, K. Mahmood, and A. Elmukashfi, "Dissecting energy consumption of nb-iot devices empirically," 04 2020.

[2] A. Rayes and S. Salam, *Internet of Things (IoT) Overview*. Cham: Springer International Publishing, 2019, pp. 1–35. [Online]. Available: https://doi.org/10.1007/978-3-319-99516-8_1

[3] ——, *IoT Vertical Markets and Connected Ecosystems*. Cham: Springer International Publishing, 2019, pp. 239–268. [Online]. Available: https://doi.org/10.1007/978-3-319-99516-8_9

[4] A. Nauman, Y. A. Qadri, M. Amjad, Y. B. Zikria, M. K. Afzal, and S. W. Kim, "Multimedia internet of things: A comprehensive survey," *IEEE Access*, vol. 8, pp. 8202–8250, 2020.

[5] S. A. Alvi, B. Afzal, G. A. Shah, L. Atzori, and W. Mahmood, "Internet of multimedia things: Vision and challenges," *Ad Hoc Networks*, vol. 33, pp. 87–111, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870515000876

[6] A. Rego, A. Canovas, J. M. Jiménez, and J. Lloret, "An intelligent system for video surveillance in iot environments," *IEEE Access*, vol. 6, pp. 31 580–31 598, 2018.

[7] K. Seng and L. Ang, "A big data layered architecture and functional units for the multimedia internet of things," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, pp. 500–512, 2018.

[8] A. Rayes and S. Salam, *IoT Protocol Stack: A Layered View*. Cham: Springer International Publishing, 2019, pp. 103–154. [Online]. Available: https://doi.org/10.1007/978-3-319-99516-8_5

[9] C. Bormann, M. Ersue, and A. Keränen, "Terminology for Constrained-Node Networks," RFC 7228, May 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7228.txt

[10] A. Jayasuriya, S. Perreau, A. Dadej, and S. Gordon, "Hidden vs. exposed terminal problem in ad hoc networks," *Proceedings of the Australian Telecommunication Networks and Applications Conference*, 01 2004.

[11] "Ieee standard for low-rate wireless networks," *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pp. 1–709, 2016.

[12] Y. Xiao and Y. Pan, *Overview of IEEE 802.15.1 Medium Access Control and Physical Layers*, 2009, pp. 105–134.

[13] G. Montenegro, J. Hui, D. Culler, and N. Kushalnagar, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," RFC 4944, Sep. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4944.txt

[14] D. S. E. Deering and B. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 8200, Jul. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8200.txt

[15] G. Montenegro, C. Schumacher, and N. Kushalnagar, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals," RFC 4919, Aug. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4919.txt

[16] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-34, Jan. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-34

[17] L.-E. Jonsson, K. Sandlund, and G. Pelletier, "The RObust Header Compression (ROHC) Framework," RFC 5795, Mar. 2010. [Online]. Available: https://rfc-editor.org/rfc/rfc5795.txt

[18] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, 2017, pp. 1–7.

[19] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252, Jun. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7252.txt

[20] A. P. Castellani, S. Loreto, A. Rahman, T. Fossati, and E. Dijk, "Guidelines for Mapping Implementations: HTTP to the Constrained Application Protocol (CoAP)," RFC 8075, Feb. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8075.txt

[21] R. T. Fielding, "REST: architectural styles and the design of network-based software architectures," Doctoral dissertation, University of California, Irvine, 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[22] E. Rescorla, "HTTP Over TLS," RFC 2818, May 2000. [Online]. Available: https://rfc-editor.org/rfc/rfc2818.txt

[23] "Information technology — Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats," International Organization for Standardization, Geneva, CH, Standard, Aug. 2019.

[24] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347, Jan. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6347.txt

[25] P. Krawiec, M. Sosnowski, J. Mongay Batalla, C. Mavromoustakis, and G. Mastorakis, "Dasco: dynamic adaptive streaming over coap," *Multimedia Tools and Applications*, pp. 1–20, Jun. 2017.

[26] A. Rayes and S. Salam, *Industry Organizations and Standards Landscape*. Cham: Springer International Publishing, 2019, pp. 297–313. [Online]. Available: https://doi.org/10.1007/978-3-319-99516-8_11

[27] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," RFC 7540, May 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7540.txt

[28] M. Bishop, "Hypertext Transfer Protocol Version 3 (HTTP/3)," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-http-

34, Feb. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34

[29] R. Peon and H. Ruellan, "HPACK: Header Compression for HTTP/2," RFC 7541, May 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7541.txt

[30] C. B. Krasic, M. Bishop, and A. Frindell, "QPACK: Header Compression for HTTP/3," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-qpack-21, Feb. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-qpack-21

[31] J. Vitter, "Design and analysis of dynamic huffman codes," *Journal of the ACM*, vol. 34, 10 1994.

[32] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, September 1952.