ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria
Laurea Magistrale in Ingegneria informatica

Compressione adattiva di messaggi HTTP per scenari Internet of Things

Tesi in
Multimedia Services And Applications M

Studente:
FRANCESCO PANDOLFI

Relatore:
Professor
DANIELE TARCHI

SESSIONE Marzo
ANNO ACCADEMICO 2020–2021

KEYWORDS

Compressione adattiva

HTTP

Internet of Things

Codifica di Huffman

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 DASH per lo streaming multimediale, possano essere implementate anche in reti IoT. Le prestazioni di queste soluzioni dipenderanno dalle specifiche caratteristiche dello scenario di applicazione. I risultati ottenuti mostrano inoltre che nello scenario di riferimento la compressione adattiva di messaggi HTTP raggiunge prestazioni superiori a CoAP.

Indice

1 Introduzione **1**

2 Dispositivi e reti IoT **5**

2.1 Sfide tecnologiche e protocolli standard 8

2.1.1 Livello datalink 8

2.1.2 Livello rete 10

2.1.3 Livello trasporto 11

2.1.4 Livello applicazione 12

2.2 Scenario di riferimento e obiettivi 14

3 Codifiche di sorgenti lossless **17**

3.1 Autoinformazione ed entropia 18

3.2 Codifiche 20

3.3 Codifica di Huffman 23

3.4 Algoritmo V 26

4 Protocollo HTTP Compressed (HTTPC) 29

4.1 Strategia di compressione 30

4.2 Campi 34

4.2.1 Tipo 0 35

4.2.2 Tipo 1 35

4.2.3 Tipo 2 36

4.3 Sessioni 36

5 Implementazione 39

5.1 Codifica di campi di tipo 0 41

5.2 Codifica di campi di tipo 1 42

5.3 Codifica di campi di tipo 2 45

6 Risultati numerici 47

6.1 Scenario di riferimento 47

6.2 Scenario con traffico eterogeneo 49

6.3 Scenario con traffico DASH 55

6.4 Conclusioni 61

Capitolo 1

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 promette una trasformazione radicale della rete Internet, connettendo miliardi di nuovi dispositivi con caratteristiche 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, standard IoT per comunicazione client-server, risolve queste criticità 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 2 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 3 saranno presentati i concetti chiave della teoria dell'informazione, fondamento di un qualsiasi algoritmo di compressione, e due tecniche di compressione che saranno sfruttate da HTTPC.
Nel capitolo 4 si definirà la strategia di compressione adottata da HTTPC, mentre nel capitolo 5 si fornirà dello pseudocodice per l'implementazione dell'algoritmo di codifica.

Infine, nel capitolo 6 si valuteranno le prestazioni di HTTPC in tre scenari applicativi e si proporranno delle estensioni al protocollo per aumentarne l'efficacia.

Capitolo 2

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 2.1).

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

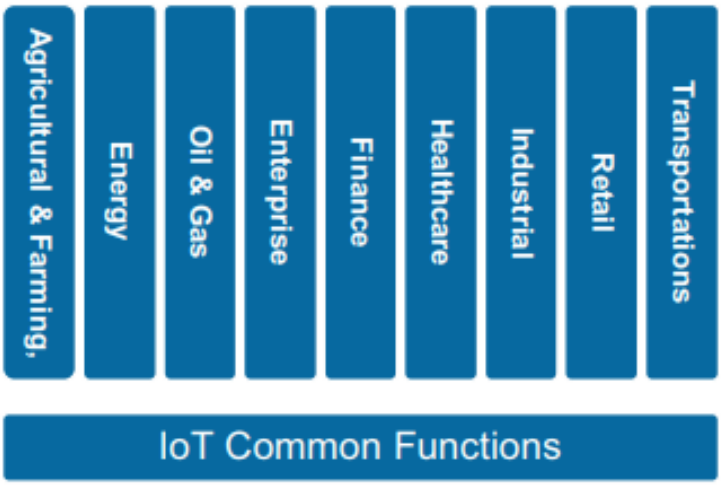


Figura 2.1: Aree di applicazione della tecnologia IoT (IoT Verticals) [3]

nel modo più efficace possibile.

Lo scopo della ricerca in ambito Internet of Things 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 2.1: 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 2.2):

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.

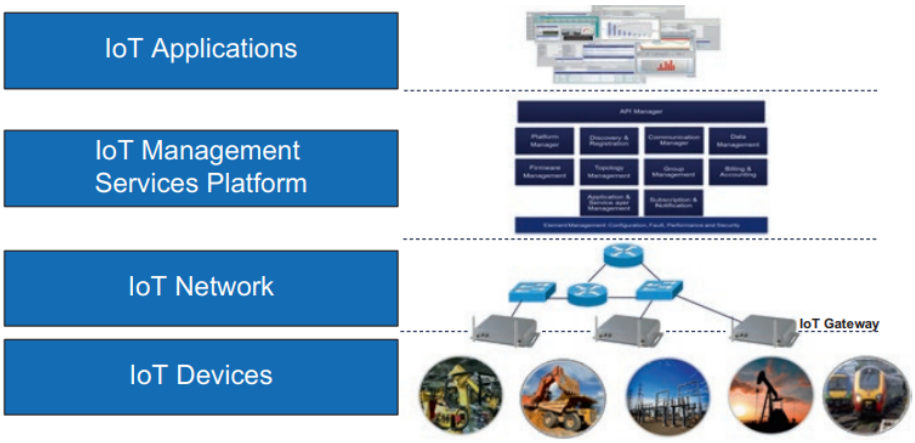


Figura 2.2: livelli di una soluzione IoT [2]

2.1 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].

2.1.1 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 2.2). 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 2.2: Classificazione dispositivi per vincoli di memoria [9]

2.1.2 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.

2.1.3 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.

2.1.4 Livello applicazione

I protocolli standard di livello applicazione sono 4: 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

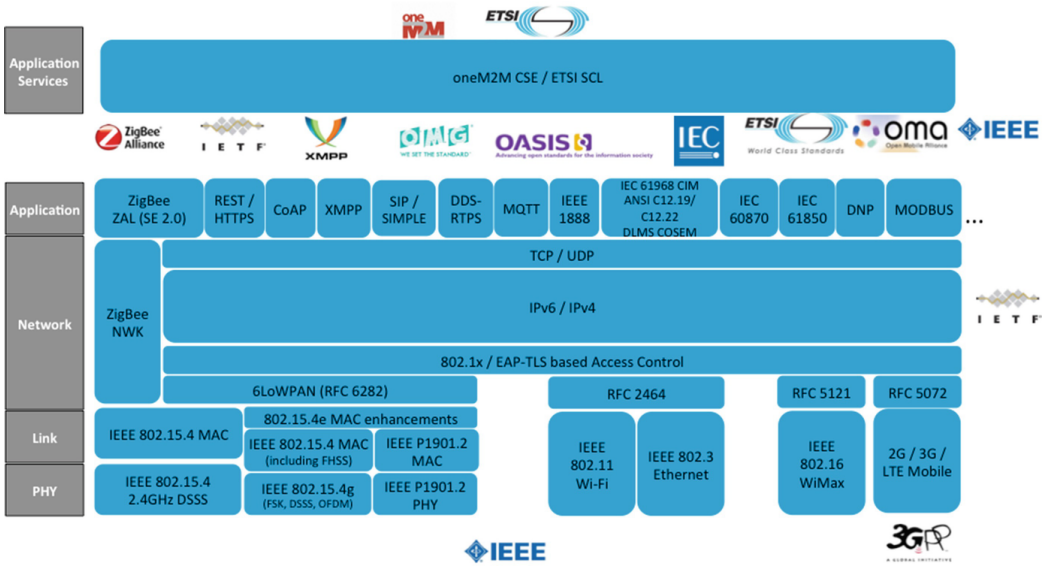


Figura 2.3: Protocolli standard nell'ecosistema IoT [26]

2.2 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 interes-

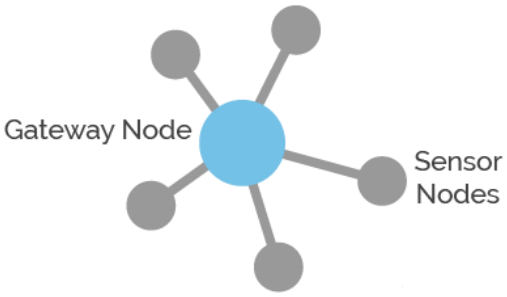
sante potrebbe essere l'implementazione di HTTPC su HTTP/3.0, il quale prevede delle ottimizzazioni per essere usato su QUIC.

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 2.4). 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.



HTTP/1.1
TCP / QUIC (+ ROHC)
IPv4 / IPv6 (+ 6LowPan)
Wi-Fi / Bluetooth / Zigbee

Figura 2.4: Esempio di rete a stella e stack protocollare dello scenario di riferimento

Capitolo 3

Codifiche di sorgenti lossless

In questo capitolo sono presentati gli elementi di teoria dell'informazione utili a capire il funzionamento di HTTPC e l'algoritmo di codifica su cui si basa, noto come algoritmo V [31] (sezione 3.4), il quale è una versione adattiva della codifica di Huffman [32], a cui è dedicata la sezione 3.3. La tecnica di compressione utilizzata rientra nella categoria *lossless*.

La teoria dell'informazione sta alla base di tutti gli algoritmi di compressione oggi esistenti e fornisce un modello per quantificare l'informazione trasportata da un messaggio [33].

3.1 Autoinformazione ed entropia

Si assume che un messaggio sia una sequenza di simboli generati da una sorgente modellata come variabile casuale e identicamente distribuita. La quantità di informazione trasportata da un singolo simbolo è strettamente legata alla probabilità del simbolo di essere emesso dalla sorgente. Sia p_i tale probabilità per il simbolo i, la quantità di informazione, o autoinformazione, è definita in (3.1)

$$I_i = -log(p_i) \tag{3.1}$$

La base del logaritmo stabilisce l'unità di misura. In questo documento, salvo diverse indicazioni, tutti i logaritmi sono intesi in base 2, per misurare l'informazione in bit.[1] Un simbolo molto probabile porterà poca informazione. Al contrario, un simbolo poco probabile porterà con sé molta informazione.

La media delle autoinformazioni dei simboli di una stessa sorgente ne costituisce l'entropia. Assumendo una sorgente discreta S che emette N simboli possibili, l'entropia della sorgente è definita in (3.2).

$$H(S) = -\sum_{i=1}^{N} p_i * log(p_i) \tag{3.2}$$

inserisci figura funzione entropia variabile binaria

[1] La base 256 misurerebbe l'informazione in byte, quella naturale (e) in nat.

L'entropia di una sorgente è massima quando i suoi simboli sono equiprobabili. Inoltre non dipende dall'ordine con il quale i simboli si presentano, ma solo dalla loro distribuzione di probabilità.

Il teorema di codifica di sorgente di Shannon stabilisce che l'entropia di una sorgente è la quantità minima di informazione necessaria a rappresentare un simbolo di quella sorgente senza errori. Quindi, per codificare un messaggio di n simboli emessi da una sorgente di entropia H saranno necessari almeno nH bit.

Si prenda come esempio un messaggio costituito di n simboli di una stessa sorgente S, con n molto grande. Per la legge dei grandi numeri, se p_i è la probabilità con cui S emette il simbolo i, il numero di occorrenze di i nel messaggio sarà $p_i^{np_i}$. Quindi, qualsiasi sia il messaggio effettivamente generato, esso sarà una tra tante permutazioni di una distribuzione di simboli nota a priori, perché propria della sorgente. Queste permutazioni sono dette sequenze tipiche. La probabilità che la sorgente generi una specifica sequenza tipica x è

$$p(x) = \prod_{i=1}^{N} p_i^{n*p_i} = \prod_{i=1}^{N} 2^{n*p_i*log(p_i)} = 2^{n*\sum_{i=1}^{N} p_i*log(p_i)} \tag{3.3}$$

Applicando (3.2) si ha

$$p(x) = 2^{-nH} \tag{3.4}$$

Assumendo che tutte le sequenze tipiche di una stessa sorgente siano equiprobabili, se T è il numero di sequenze tipiche si ha

$$p(x) = 1/T \tag{3.5}$$

Unendo (3.4) e (3.5) si ottiene

$$T = 2^{nH} \tag{3.6}$$

Quindi basteranno nH bit per rappresentare la sequenza tipica e, in media, H bit per rappresentare un simbolo.

3.2 Codifiche

In teoria dei codici, una codifica (o codice) è una funzione che associa in modo univoco una sequenza ordinata di elementi del dominio ad un elemento del codominio, il quale contiene le informazioni da rappresentare. In questo documento chiameremo *alfabeto* il dominio della funzione e *dizionario* il codominio. Gli elementi dell'alfabeto saranno detti *lettere*, mentre gli elementi del dizionario saranno detti *simboli*. Salvo diverse indicazioni, si useranno codifiche con alfabeti binari. Le sequenze di lettere saranno *parole*, le sequenze di parole saranno *messaggi*. Ad ogni parola è associato univocamente

un simbolo. Infine, il termine codifica (o codice) sarà usato per indicare, a seconda del contesto, sia l'insieme di parole, sia l'algoritmo che consente la creazione di tale insieme.

Dizionario	Codice
A	0
B	1
C	00
D	11

Tabella 3.1: Esempio di codice binario per il dizionario A,B,C,D

Un codice è detto univocamente decodificabile se può scomporre un qualsiasi messaggio in un'unica sequenza di parole. La disuguaglianza di Kraft-McMillan (3.7) costituisce una condizione necessaria e sufficiente affinché un codice sia univocamente decodificabile.

$$\sum_{i=1}^{N} 2^{-l(w_i)} \leq 1 \tag{3.7}$$

dove N indica il numero di parole del codice, mentre l(w_i) è la lunghezza in bit della i-esima parola. Il codice in Tabella 3.1 non è univocamente decodificabile. Applicando (3.7) si ottiene

$$2^{-1} + 2^{-1} + 2^{-2} + 2^{-2} = 1/2 + 1/2 + 1/4 + 1/4 = 1.5 \tag{3.8}$$

Si prenda come esempio il messaggio 0011. La sua decodifica produrrebbe 2

sequenze di simboli possibili: AABB e CD.

Esempi di codifiche univocamente decodificabili sono le codifiche prefisso, in cui ogni parola non è mai prefisso di un'altra. Questa proprietà le rende rappresentabili tramite alberi: ad ogni foglia è associato un simbolo e ad ogni ramo nel percorso dalla radice ad ogni simbolo è associato una lettera della parola con cui il simbolo è codificato.

Dizionario	Codice
A	0
B	10
C	110
D	111

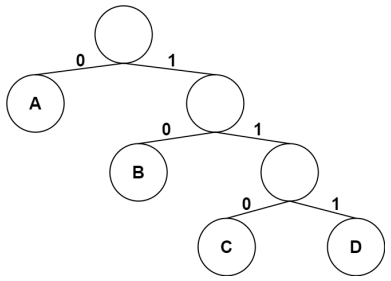


Tabella 3.2: Esempio di codice prefisso binario per il dizionario A,B,C,D. Produce il messaggio 001010 per la codifica di AABB e 110111 per CD

Data una distribuzione di simboli, un codice sarà detto ottimo, o a ridondanza minima, se associa ad ogni simbolo una parola di lunghezza tale da minimizzare la lunghezza media. Per il teorema della codifica di sorgente tale lunghezza media minima è l'entropia della sorgente. Quindi un codice ottimo associa ad ogni simbolo una parola di lunghezza pari all'autoinformazione del simbolo.

3.3 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.

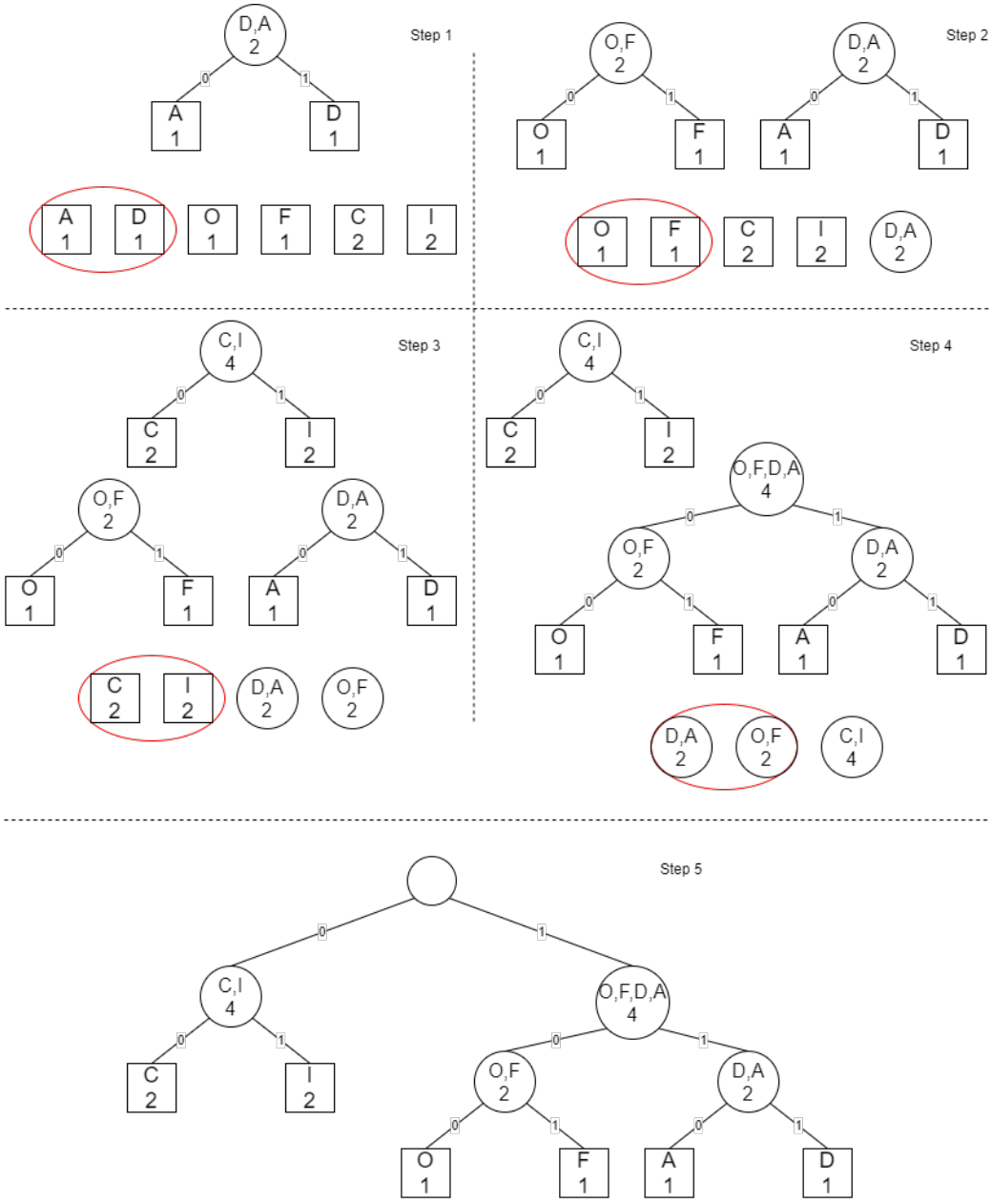


Figura 3.1: passaggi per trovare una codifica di Huffman per la stringa "CODIFICA"

3.4 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.

Capitolo 4

Protocollo HTTP Compressed (HTTPC)

In questo capitolo è presentato il protocollo HTTPC. 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.

4.1 Strategia di compressione

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 (sezione 3.4). 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.

4.2 Campi

Basandosi sulla definizione dei campi di HTTP/1.1 [], HTTPC definisce 6 campi per le richieste e 4 per le risposte (4.1). 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 protocol-

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

Tabella 4.1: Campi HTTPC

lo HTTPC dovrà per prima cosa scomporlo nei suoi campi costituenti che saranno poi compressi in ordine gerarchico.

4.2.1 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.

4.2.2 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 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.

4.2.3 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.

4.3 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 5

Implementazione

In questo capitolo è presentato lo pseudocodice per le codifiche dei tipi di campi visti nel capitolo precedente, ottenuto dal codice Java usato per le simulazioni del capitolo 6. Tutte le variabili usate nello pseudocodice sono considerate stringhe o alberi. In implementazioni reali è consigliato formalizzare anche il concetto di sequenza di bit.

Si assume che, qualsiasi sia la tecnica di memorizzazione del contesto, essa memorizzi gli alberi di codifica di campi diversi in file distinti. Sono previste 4 primitive per la memorizzazione degli alberi:

- loadTree(path), ritorna l'albero memorizzato nel file di percorso indicato.

- createTree(value), ritorna il percorso al file in cui è inizializzato un

nuovo albero per i campi successivi a value.

- saveTree(tree,path), indica che l'albero *tree* dovrà essere salvato nel file di percorso *path*. Non ritorna niente.

- flushTrees(), effettua la sovrascrittura degli alberi nei percorsi indicati tramite primitiva *saveTree*.

Ogni albero possiede sempre almeno il simbolo *Not Yet Transfered* (NYT), usato per indicare la codifica di un nuovo valore non ancora memorizzato nell'albero. In questi casi sarà dunque trasmessa la codifica del simbolo NYT seguita dalla codifica statica del nuovo valore, indicata attraverso la primitiva *staticEncode(value)*. Il valore sarà letto carattere per carattere secondo formato ISO-8859-1 e sarà aggiunto in coda un carattere "\n" per indicare la fine della codifica statica. Si usa il carattere terminatore di riga in quanto, secondo la specifica di HTTP/1.1, nessun campo può contenerlo.

Inoltre, si assume che un albero sia implementato come oggetto con i seguenti metodi:

- contains(value): restituisce un valore booleano che indica se il parametro è contenuto come foglia in questo albero.

- insert(value): permette di incrementare il contatore delle occorrenze del parametro. Se il valore è nuovo gli viene creata una foglia associata.

Successivamente, l'albero viene aggiornato secondo algoritmo V.

- link(value): restituisce il percorso del file in cui è memorizzato l'albero associato al parametro. Sarà usato per la codifica del prossimo campo.

- code(value): restituisce la sequenza di bit associata al parametro. Se non esiste una foglia associata viene ritornata la sequenza di bit associata al simbolo NYT. La nuova foglia sarà creata solo dopo l'invocazione di *insert*.

- decode(coded): restituisce il valore della foglia associata alla sequenza di bit passata come parametro, null altrimenti.

5.1 Codifica di campi di tipo 0

L'interfaccia della codifica di campi di tipo 0 prevede 3 parametri in ingresso:

- value, valore del campo da codificare

- coded, sequenza di bit a cui deve essere concatenata il valore codificato

- path, percorso del file che contiene il sottoalbero relativo al campo

La codifica ritorna un nuovo coded e un nuovo path.

codeType0(value, coded, path)

```
tree = loadTree(path);
if tree.contains(value) then
    coded += tree.code(value);
    tree.insert(value);
    newpath = tree.link(value);
else
    coded += tree.code(NYT) + staticEncode(value+"\n");
    tree.insert(value);
    newpath = createTree(value);
end
saveTree(tree,path);
return {coded,newpath};
```

5.2 Codifica di campi di tipo 1

Il protocollo HTTP definisce 3 tipi di URI:

- "*", si riferisce all'host e non ad una risorsa in particolare.
- URI assoluto, come definito in RFC 3986[].

  Esempio, "http://www.ietf.org/rfc/rfc3986.txt".
- URI relativo, non esplicita lo schema, né il server. Esempio, "/rfc/rfc3986.txt".

In caso di URI assoluto o relativo, la codifica spezza l'URI in più campi che verranno codificati come se fossero di tipo 0. Ogni campo sarà codificato usando l'albero indicato dal campo precedente. Quando l'ultimo campo è

codificato, si aggiunge la codifica di un campo vuoto "", in modo da indicare in fase di decodifica che il campo di tipo 1 è terminato.

codeType1(value, coded, path)

```
box = {coded,path};
if value == "*" then
 |  return codeType0(value,box[0],box[1]);
else
 |  if value.startsWith("/") then
 |   |  box = codeRelative(value, box[0], box[1]);
 |  else
 |   |  box = codeAbsolute(value, box[0], box[1]);
 |  end
end
return codeType0("", box[0], box[1]);
```

Gli URI relativi sono composti da un numero variabile di parti che iniziano sempre per "/".

codeRelative(value, coded, path)

```
value = value.substring(1);
index = value.indexOf("/");
/*Senza la prima istruzione indexOf avrebbe ritornato 0, dedicando una codifica al solo carattere "/" */.
box = {coded,path};
while index > −1 do
 |  part = "/"+string.substring(0,index);
 |  value = value.substring(index+1);
 |  index = value.indexOf("/");
 |  box = codeType0(part, box[0], box[1]);
end
return codeType0("/"+value, box[0], box[1]);
```

Gli URI assoluti seguono lo schema [<protocol>://]<peer><relative>

codeAbsolute(value, coded, path)

```
box = coded,path;
index = value.indexOf(":");
relative = null;
if index > -1 and value.contains("://") then
  | protocol = value.substring(0, index+3);
  | peer = value.substring(index+3);
  | box = codeType0(protocol, box[0], box[1]);
else
  | peer = value;
end
index = peer.indexOf("/");

if index > -1 then
  | relative = peer.substring(index);
  | peer = peer.substring(0,index);
end
box = codeType0(protocol, box[0], box[1]);
box = codePeer(peer, box[0], box[1]);

if relative != null then
  | box = codeRelative(relative, box[0], box[1]);
end
return box;
```

Infine, <peer> rappresenta l'indirizzo a cui connettersi, che può essere grezzo o espresso tramite DNS. In entrambi i casi può essere spezzato in corrispondenza dei caratteri ".". L'ultima parte di <peer> può contenere anche la porta, separata dal resto dell'indirizzo dal carattere ":".

codePeer(value, coded, path)

```
index = value.indexOf(".");
box = coded,path;
while index > −1 do
    part = value.substring(0,index+1);
    value = value.substring(index+1);
    index = value.indexOf(".");
    box = codeType0(part,box[0],box[1]);
end
index = value.indexOf(":");
if index > -1 then
    part = value.substring(0,index+1);
    value = value.substring(index+1);
    box = codeType0(part,box[0],box[1]);
end
return codeType0(value,box[0],box[1]);
```

5.3 Codifica di campi di tipo 2

Un campo di tipo 2 è costituito da una lista di coppie nome-valore, le quali saranno codificate mantenendo l'ordine con cui appaiono. Per ogni coppia si codifica prima il nome e poi il valore. I nomi saranno codificati secondo l'albero indicato dal valore precedente, mentre i valori saranno codificati secondo l'albero indicato dal proprio nome. In questo modo è possibile catturare eventuali interdipendenze sia tra i soli nomi delle coppie (ad esempio, la presenza o assenza di una certa coppia determina la presenza o assenza di un'altra, come nel caso degli headers *Content-Length* e *Content-Type* descritto nel capitolo 4), sia tra nomi e valori (ad esempio, il valore associato

all'header *Content-Type* determina la presenza e il valore associato all'header *Accept-Encoding*).

Dopo aver codificato l'ultima coppia si aggiunge la codifica di un campo vuoto, come per i campi di tipo 1. In fase di decodifica, questo campo vuoto sarà in corrispondenza di un nome. Dato che una coppia con nome vuoto non può esistere negli headers e parametri HTTP, questo campo indica la terminazione del campo di tipo 2. Resta comunque possibile la codifica di coppie con valore vuoto.

codeType2(couples, coded, path)

```
box = {coded,path};
foreach couple in couples do
    box = codeType0(couple.name, box[0], box[1]);
    box = codeType0(couple.value, box[0], box[1]);
end
return codeType0("", box[0], box[1]);
```

Capitolo 6

Risultati numerici

Il protocollo HTTPC è stato valutato in 3 scenari di applicazione. Per ogni scenario, i tassi di compressione sono espressi secondo la formula

$$1 - \frac{\text{bit messaggio codificato}}{\text{bit messaggio originale}} \tag{6.1}$$

6.1 Scenario di riferimento

Il primo scenario consiste in una rete di sensori conforme allo scenario di riferimento descritto in sezione 2.2 su cui è installata un'applicazione per il monitoraggio della temperatura di ogni stanza in cui è presente un sensore. In figura 6.1 è riportato un esempio dei messaggi CoAP che potrebbero essere trasmessi tra i nodi ad ogni aggiornamento dell'applicazione. La richiesta

necessita di 16 bytes per essere codificata (di cui 11 per codificare il percorso "temperature") mentre la risposta ne richiede 11 (di cui 6 per il payload "22.3 C"). In totale, questa soluzione CoAP richiede la trasmissione di 27 bytes per aggiornamento, che equivalgono a 216 bits.

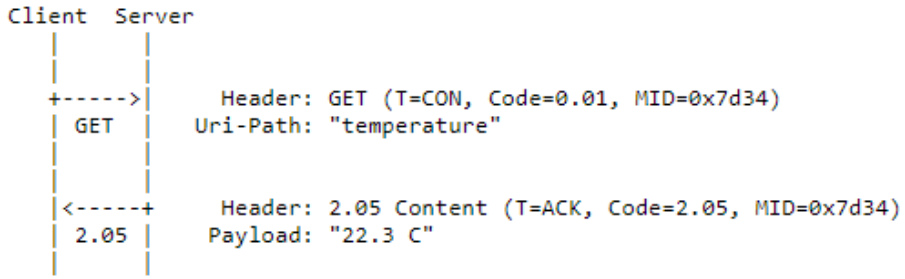


Figura 6.1: Esempio di trasmissione CoAP (RFC 7252[19], Appendice A)

I messaggi HTTP equivalenti sono riportati in figura 6.2. Sono necessari 24 bytes per la richiesta e per 66 la risposta, per un totale di 90 bytes, che equivalgono a 720 bits (ogni carattere, spazi e a capo inclusi, occupa un byte).

Supponendo che non avvengano altre trasmissioni oltre a quelle necessarie per l'aggiornamento dell'applicazione, il protocollo HTTPC comprimerà tramite codifica statica solo la prima richiesta e la prima risposta. In questa fase i messaggi codificati saranno più brevi rispetto ai messaggi HTTP in chiaro, ma i messaggi CoAP saranno ancora più brevi. I messaggi successivi potranno essere codificati utilizzando un solo bit per campo (fatta eccezione per il body della risposta). Tutti gli alberi di codifica che compongono il contesto della sessione possiederanno infatti due soli valori: NYT e quello

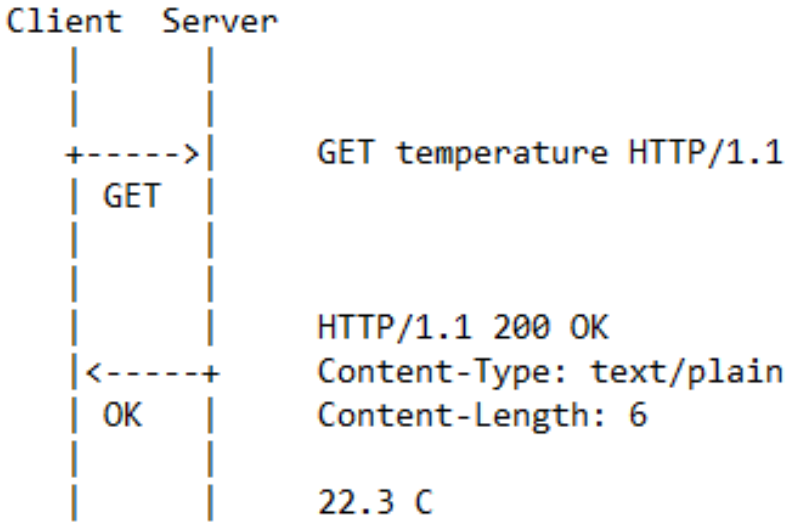


Figura 6.2: Esempio di trasmissione HTTP

memorizzato nella fase di codifica statica. La richiesta potrà quindi essere codificata con soli 3 bit, mentre la risposta con 54 (di cui 48 per il body, assumendo che il formato della temperatura rimanga lo stesso dell'esempio), per un totale di 57 bit.

Il tasso di compressione in questo scenario è del 92% rispetto ad HTTP/1.1 e del 74% rispetto a CoAP.

6.2 Scenario con traffico eterogeneo

Nel secondo scenario si è valutato HTTPC applicandolo a traffico internet generato dall'autore durante la stesura di questa tesi. L'obiettivo è stato

la generazione di traffico eterogeneo ma abbastanza ripetitivo. Il traffico è stato intercettato attraverso un proxy e memorizzato su disco. Attraverso un prototipo di implementazione di HTTPC si sono codificati i messaggi secondo l'ordine con cui sono stati intercettati, simulando così una codifica a tempo di comunicazione tra il client e il proxy. Il dataset creato per questo scenario è composto da 8096 messaggi, di cui metà sono richieste e metà risposte. Durante la generazione, durata circa 4 ore, si è evitato di accedere a contenuti multimediali, per quanto possibile. Il traffico registrato ammonta a circa 155MB, di cui solo 7.61MB (5% del totale) sono occupati dalle intestazioni dei messaggi.

In questo scenario sono state effettuate due valutazioni: nella prima si è simulato un'unica sessione in cui sono stati trasmessi tutti i messaggi. Figura 6.3 mostra i tassi di compressione ottenuti e si può notare come tendano a migliorare con il passare del tempo. Come si può vedere da figura 6.4 i tempi di codifica (espressi in millisecondi) oscillano attorno ad un valore stabile a seconda della quantità di dati da codificare staticamente. Ciò significa che gli alberi di codifica restano relativamente piccoli, ipotesi supportata dal fatto che la dimensione totale del contesto è di 8.91MB, e quindi comparabile con la dimensione originale delle intestazioni (l'eccesso deriva dal fatto di dover memorizzare anche le forme degli alberi di codifica e i contatori associati alle foglie).

Nella seconda valutazione il traffico è stato filtrato secondo il valore del parametro *Host* delle richieste e per ogni host identificato è stata creata una sessione HTTPC distinta. Così facendo ci si possono aspettare tassi di compressione migliori al costo di un maggiore spazio richiesto dai contesti. I tassi di compressione di ogni sessione sono mostrati in figura 6.5, mentre la dimensione totale dei contesti memorizzati è di 9.03MB. Il fatto che questo valore non si discosta di molto da quello ottenuto nella prima valutazione è causato da un'inefficienza della codifica dei campi di tipo 2. In alcune sessioni infatti sono stati trasmessi messaggi molto simili tra loro ma, a causa di differenze nei valori dei primi headers, i successivi sono stati memorizzati in rami diversi, nonostante avessero gli stessi valori. Inoltre, gli ultimi headers contenevano spesso dei cookie, i quali, non essendo compressi bene dall'albero di codifica statica, hanno provocato una degradazione del tasso di compressione della sessione. Alcune di queste sessioni "corrotte" sono state identificate tra quelle che in figura 6.5 hanno tasso di compressione vicino a 10%, tuttavia non è escluso che l'effetto negativo dell'inefficienza sia presente anche nelle altre sessioni. Una soluzione a questa inefficienza migliorerebbe sia il tasso di compressione, sia la dimensione del contesto, probabilmente anche nello scenario a sessione singola. Nella seconda valutazione non è riportato un grafico dei tempi di codifica in quanto molte sessioni non possiedono un numero di messaggi sufficiente a dare rilevanza statistica alla misurazione.

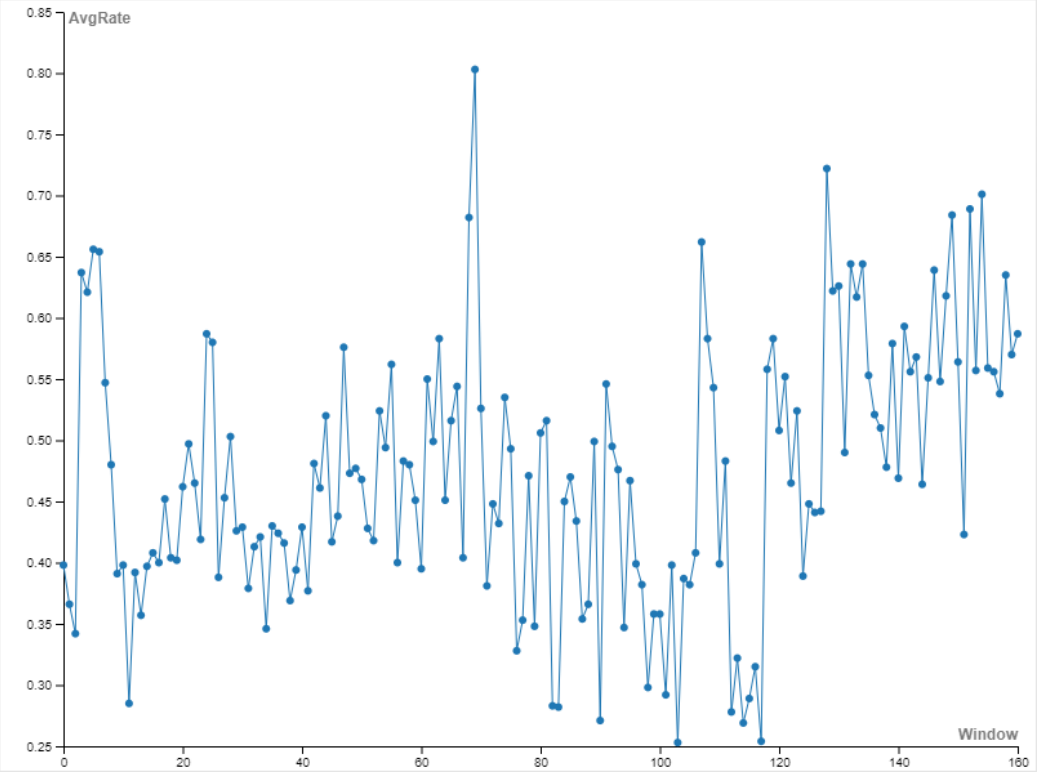


Figura 6.3: Tassi di compressione medi calcolati tramite finestra scorrevole di 50 messaggi

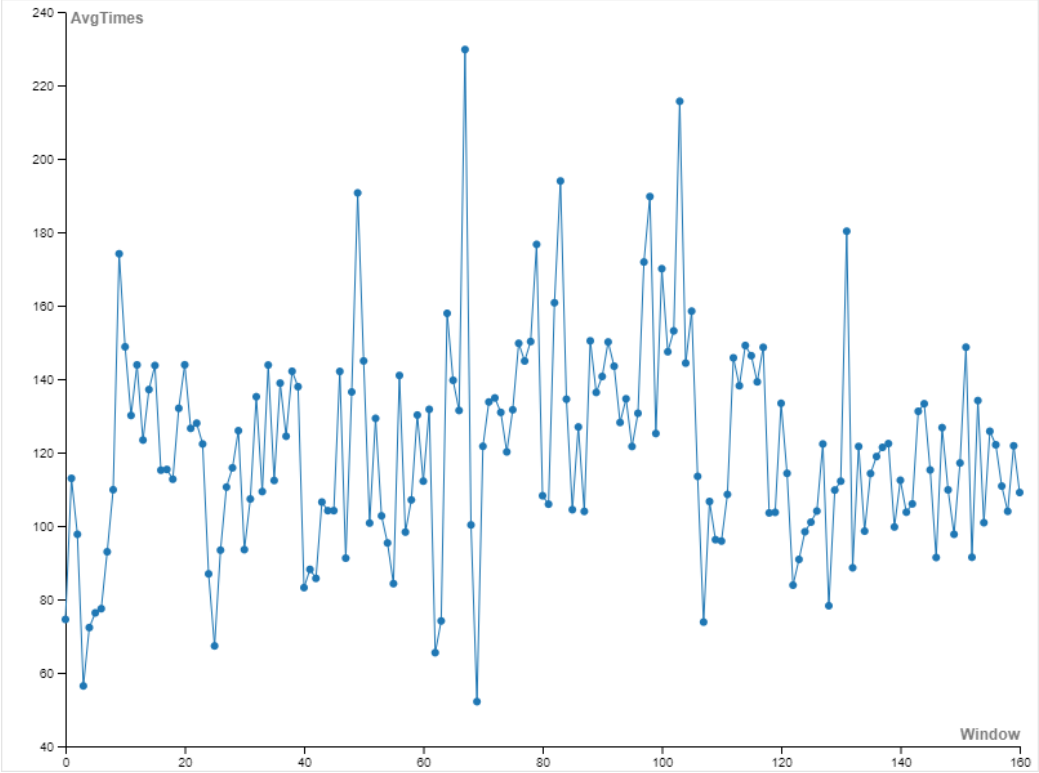


Figura 6.4: Tempi di codifica medi calcolati tramite finestra scorrevole di 50 messaggi

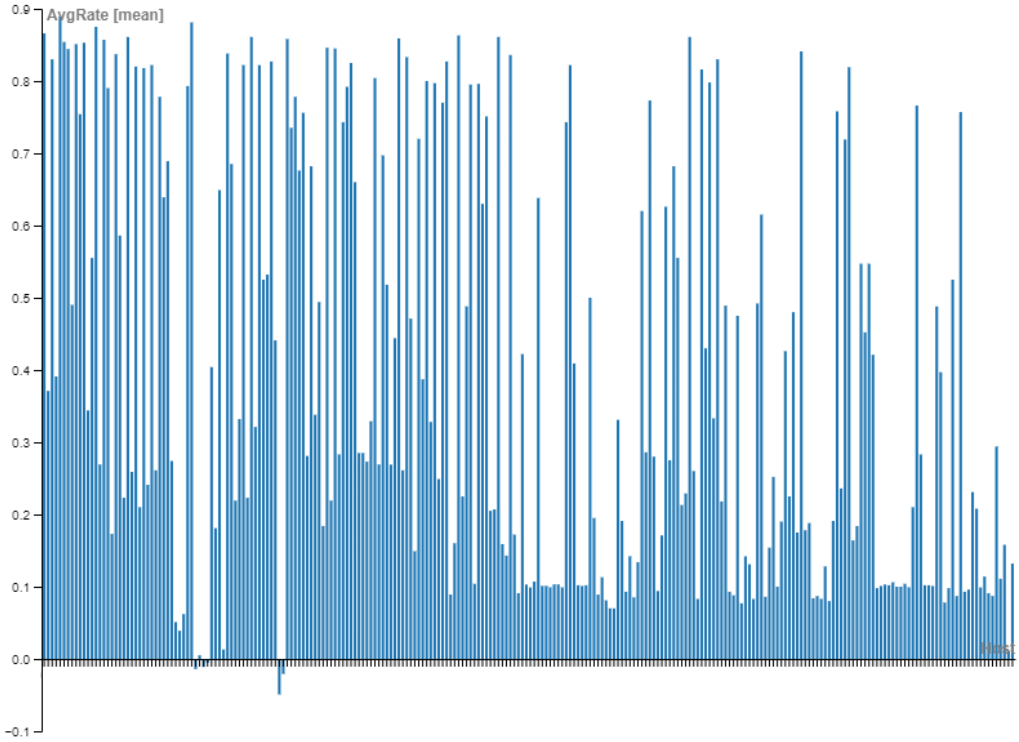


Figura 6.5: Tassi di compressione medi calcolati per ogni sessione

6.3 Scenario con traffico DASH

Il terzo scenario è molto simile al secondo ma il traffico è stato generato durante la visione di un film in streaming offerto tramite protocollo DASH. Il dataset è costituito da 7386 messaggi, di dimensione totale pari a 1.46GB. Le intestazioni dei messaggi occupano solo 5.16MB (0,35% del totale).

In questo scenario i messaggi trasmessi sono molto diversi tra loro. In particolare, sono frequenti headers con valori crescenti. Eseguendo una simulazione a sessione singola, questa caratteristica del traffico porta ad una degradazione nel tempo sia dei tassi di compressione, sia dei tempi di codifica, come mostrato in figure 6.6 e 6.7, e come previsto in sezione 4.3. La dimensione del contesto memorizzato è pari a 6.54MB, un valore che non si discosta molto da quello del secondo scenario. Tuttavia, una risoluzione dell'inefficienza della codifica di tipo 2, che ridurrebbe la dimensione del contesto del secondo scenario, non dovrebbe avere nessun effetto nel terzo.

La simulazione a sessione multipla (figura 6.8) mostra come sia possibile attenuare il degradamento del tasso di compressione. La sessione 24, utilizzata per la trasmissione del film, comprende da sola circa la metà dei messaggi del dataset e ha un tasso di compressione medio vicino al 23%, valore attorno al quale oscillano i tassi di compressione raggiunti durante la sessione, come mostrato in figura 6.9. Le sessioni con tasso intorno al 10% sono per

lo più dovute alla solita inefficienza. I tempi di codifica invece non risentono dell'approccio a sessione multipla, come mostrato da figura 6.10

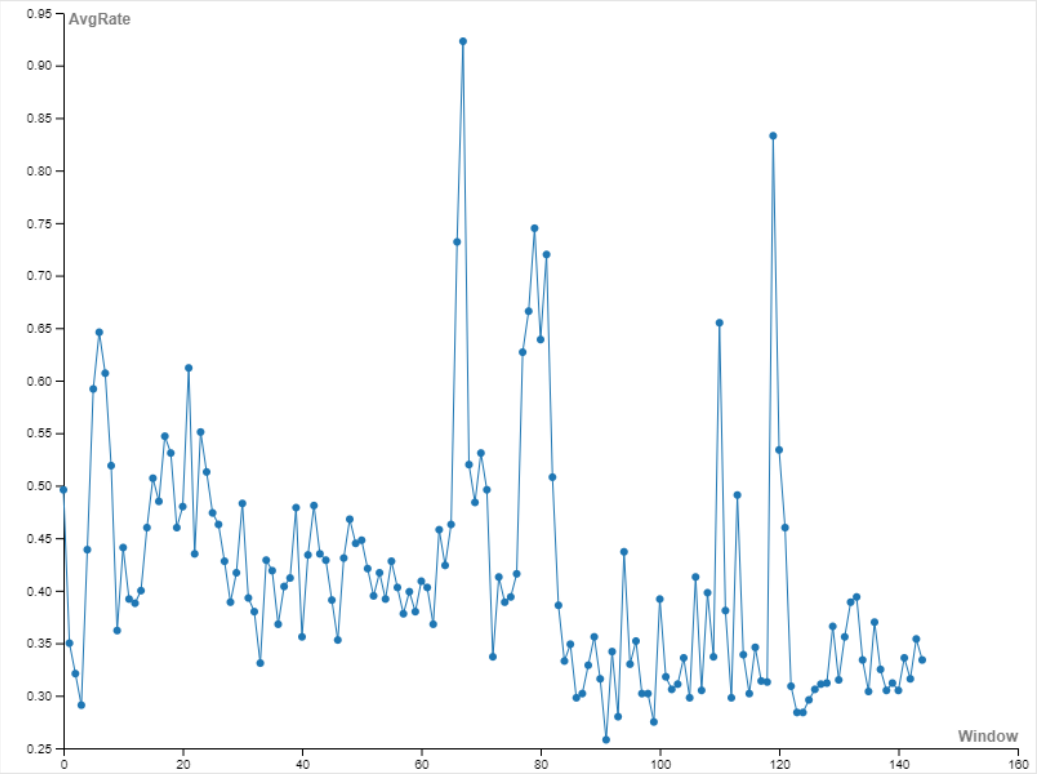


Figura 6.6: Tassi di compressione medi calcolati tramite finestra scorrevole di 50 messaggi

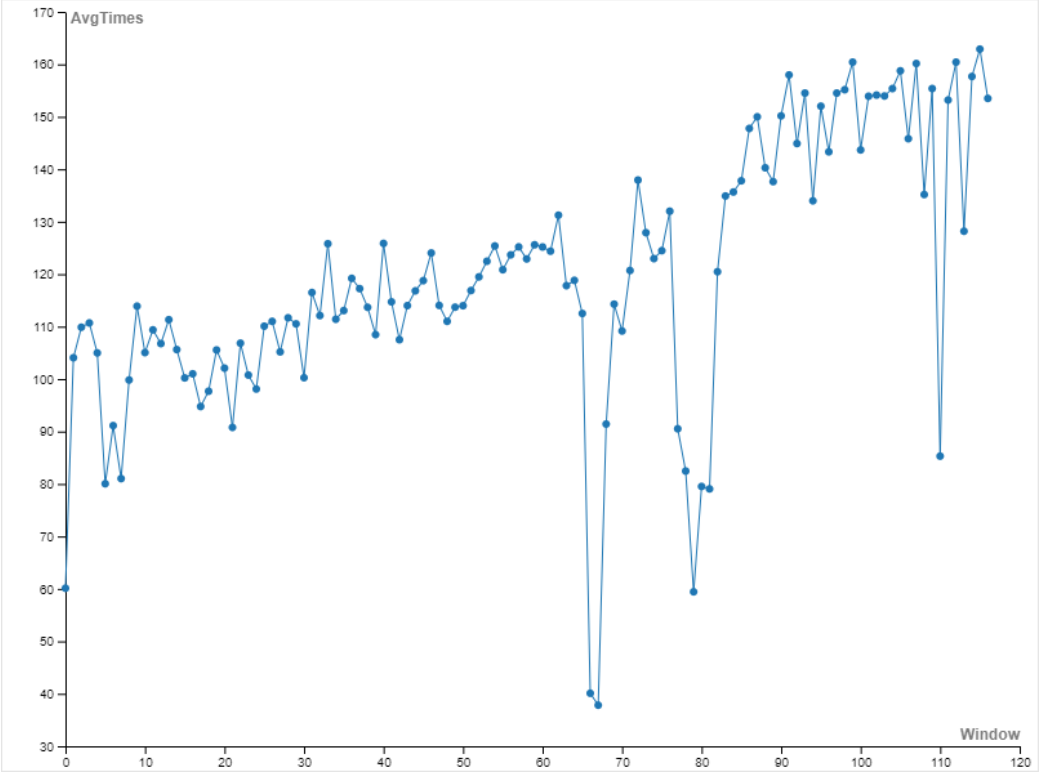


Figura 6.7: Tempi di codifica medi calcolati tramite finestra scorrevole di 50 messaggi

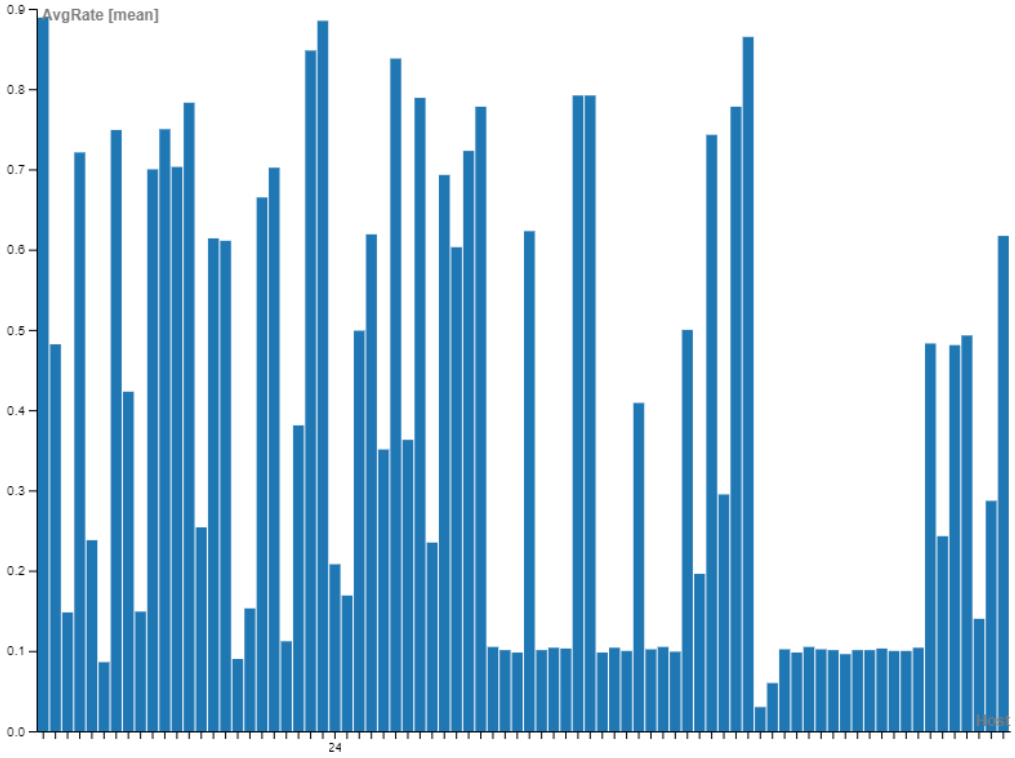


Figura 6.8: Tassi di compressione medi calcolati per ogni sessione

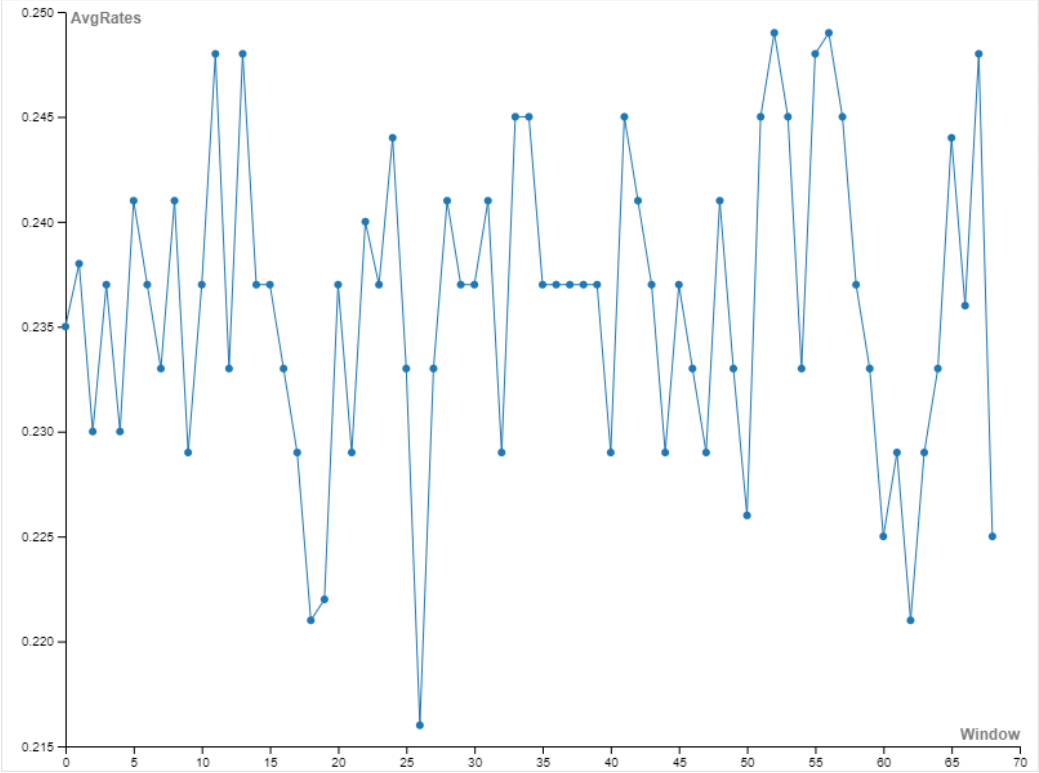


Figura 6.9: Tassi di compressione medi calcolati tramite finestra scorrevole di 50 messaggi in sessione 24

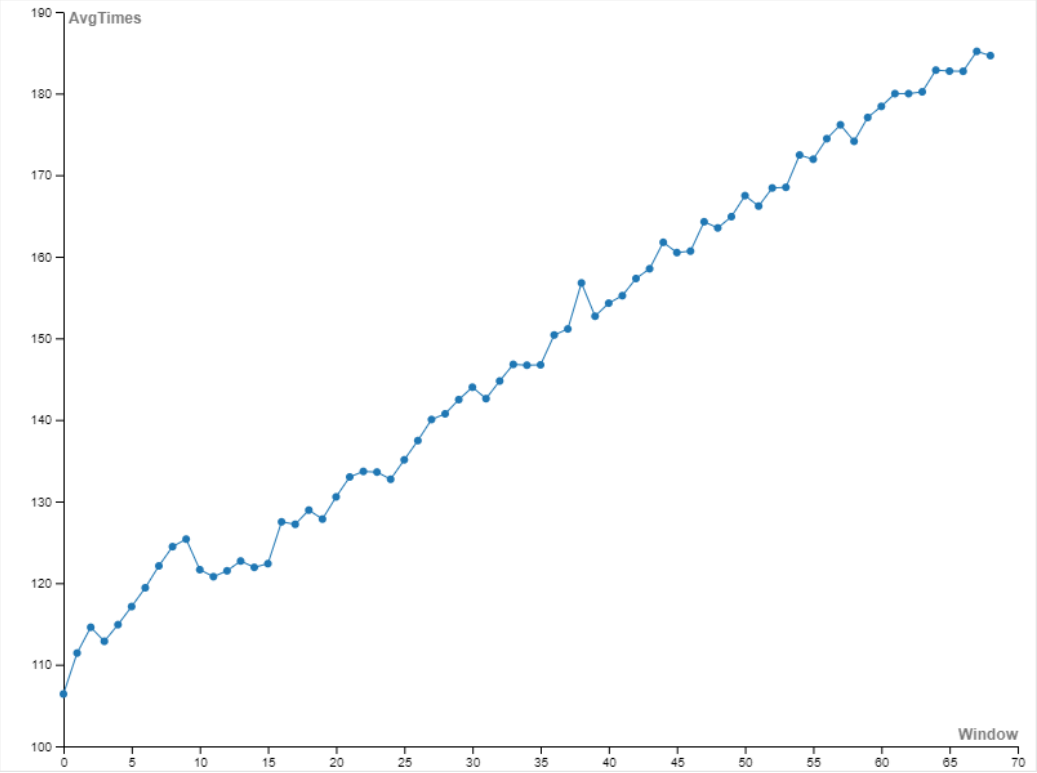


Figura 6.10: Tempi di codifica medi calcolati tramite finestra scorrevole di 50 messaggi in sessione 24

6.4 Conclusioni

La sezione 6.1 mostra quanto HTTPC sia efficace nello scenario di riferimento. Allo scopo di permettere un miglior adattamento a scenari diversi, non per forza appartenenti al mondo IoT, si propongono 5 estensioni del protocollo. Le prime tre mirano a migliorare il tasso di compressione, le ultime due mirano a ridurre lo spazio disco richiesto dal contesto della sessione.

1. Il protocollo HTTPC dovrebbe essere in grado di applicare ad uno stesso campo più codifiche possibili e di scegliere volta per volta qual'è la codifica migliore. Queste diverse tecniche per comprimere campi specifici, simili ai profili ROHC, andrebbero negoziate in fase di sincronizzazione e introdurrebbero un overhead per campo codificato proporzionale al logaritmo del loro numero.

2. Attraverso i profili, HTTPC dovrebbe prevedere tecniche di compressione dei bodies diverse a seconda del valore dell'header *Content-Type*.

3. La codifica statica di un campo per caratteri, usata la prima volta che un campo si presenta, potrebbe essere una codifica dinamica che, partendo dall'albero usato nel protocollo HPACK, si aggiorna durante la sessione tramite algoritmo V.

4. Le foglie degli alberi di codifica potrebbero contenere puntatori ad altre foglie che possiedono lo stesso valore. La dimensione e la struttura dei puntatori dipendono fortemente dalla struttura delle directory in cui i contesti sono memorizzati.

5. Invece di memorizzare in chiaro i valori delle foglie degli alberi di codifica, si possono memorizzare codificati tramite codifica statica (non compatibile con estensione 3).

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.

[33] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.