UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Elettronica

Studio e sviluppo di protocolli real-time per sistemi di Early Warning

Study and development of real-time protocols for Early Warning systems

Candidato:

Alessandro Zacchilli

Relatore:

Ing. Paola Pierleoni

Correlatore:

Dott. Lorenzo Palma

Anno Accademico 2021-2022

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Elettronica

Studio e sviluppo di protocolli real-time per sistemi di Early Warning

Study and development of real-time protocols for Early Warning systems

Candidato:

Alessandro Zacchilli

Relatore:

Ing. Paola Pierleoni

Correlatore:

Dott. Lorenzo Palma

Anno Accademico 2021-2022

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Elettronica
Via Brecce Bianche – 60131 Ancona (AN), Italy

Ringraziamenti

Voglio sfruttare questo spazio per ringraziare tutte le persone che ho avuto accanto in questi anni. In primo luogo la mia famiglia, che mi ha dato la possibilità di compiere questo percorso universitario, sostenendomi ogni qual volta ne avessi bisogno. Un grazie speciale va a alla mia fidanzata Maria Vittoria per il costante supporto esibito in qualsiasi momento. Ringrazio infinitamente i miei amici per avermi regalato momenti di spensieratezza ma anche per aver condiviso giornate intense di studio in biblioteca. Un particolare ringraziamento va alla prof.ssa Pierleoni, per avermi dato la possibilità di svolgere questo lavoro di tesi magistrale, e ai suoi collaboratori Lorenzo, Alberto e Roberto per l'aiuto che mi hanno fornito in questi mesi.

Ancona, Dicembre 2021

Alessandro Zacchilli

Indice

1 Introduzione **1**

2 Stato dell'arte **3**

2.1 Sistemi di Earthquake Early Warning 3

2.1.1 Sistemi di EEW nel mondo 6

2.1.2 Il sistema di EEW in Italia 8

2.2 Formato miniSEED 11

2.2.1 Architettura convenzionale della rete 12

2.3 Protocollo Seedlink 13

2.3.1 Descrizione del protocollo 14

2.3.2 Ringserver 15

2.4 Protocolli per l'Internet of Things 16

2.4.1 Il protocollo MQTT 19

2.5 Rappresentazione dei dati 24

2.5.1 XML 25

2.5.2 JSON 25

2.5.3 CBOR 27

2.5.4 Integrità dei dati 31

2.6 La piattaforma Docker 34

3 Materiali e metodi **39**

3.1 Introduzione 39

3.2 Confronto tra MQTT e SeedLink 40

3.2.1 Metriche di comparazione 43

3.3 Confronto tipologie di formato dati in MQTT 44

3.3.1 miniSEED con serializzazione JSON 47

3.3.2 miniSEED con serializzazione binaria 48
3.3.3 Il formato μSEED 50
3.3.4 μSEED con serializzazione JSON e CBOR 52
3.3.5 Implementazione struttura di analisi 55

4 Risultati **61**
4.1 Confronto tra MQTT e SeedLink 61
4.2 Confronto tipologie di formato dati 63
4.2.1 Dimensioni del pacchetto 63
4.2.2 Tempi di codifica 67
4.2.3 Simulazioni su Docker 69
4.2.4 Simulazioni con Raspberry e client Docker 77

5 Conclusioni e sviluppi futuri **83**

Elenco delle figure

2.1 Esempio di una forma d'onda sismica registrata dalla stazione MMUR durante un terremoto. Le linee verticali blu ed arancione indicano i tempi di arrivo rispettivamente delle onde P e delle onde S. t_{p-s} indica il tempo tra i due arrivi. L'asse dei tempi è in secondi relativi al 26-10-2016T19:18:04 UTC. 4

2.2 Tempi di arrivo dell'onda P e dell'onda S rispetto alla distanza dall'epicentro del sisma. 5

2.3 Lo stato dei sistemi di EEW nel mondo. Sullo sfondo viene presentato il rischio sismico inteso come accelerazione massima al suolo con probabilità del 10% di superamento in 50 anni. 7

2.4 Interfaccia grafica PRESTo. 10

2.5 Rappresentazione grafica di un pacchetto miniSEED. 12

2.6 Elenco delle sezioni dell'intestazione di un pacchetto miniSEED. . . 12

2.7 Struttura convenzionale di una rete sismica. 13

2.8 Elenco protocolli IoT distribuiti sui livelli di appartenenza. 17

2.9 Comparazione delle caratteristiche dei principali protocolli per l'IoT. 18

2.10 Trend di utilizzo dei protocolli di messaggistica in ambito IoT tra il 2016 e il 2018. 19

2.11 Esempio di utilizzo del protocollo MQTT. 20

2.12 Esempio di struttura del topic. 21

2.13 Handshake per i diversi livelli di QoS. 22

2.14 Formato del pacchetto MQTT. 23

2.15 Esempio serializzazione XML. 26

2.16 Rappresentazione delle tipologie di dati supportati da JSON. 27

2.17 Rappresentazione dello schema di un oggetto JSON. 27

2.18 Header della codifica CBOR. 28
2.19 Pacchetto firmato con JWS. 32
2.20 Schema di confronto tra pacchetto firmato/cifrato con JOSE e COSE. 33
2.21 Confronto tra pacchetto firmato con JSS e con COSE. 33
2.22 Confronto tra struttura Docker e quella delle Virtual Machine. . . . 35
2.23 Creazione di un Container da un Dockerfile. 36

3.1 Schema della metodologia per la valutazione dei tempi di trasmissione delle forme d'onda mediante il protocollo SeedLink ed il protocollo MQTT. 42
3.2 Riepilogo metriche considerate nella comparazione tra il protocollo SeedLink e MQTT. 44
3.3 Rappresentazione della struttura di comunicazione MQTT implementata. 45
3.4 Intervalli temporali presi in esame all'interno di una comunicazione MQTT con una $QoS = 1$. 45
3.5 Processo di realizzazione del formato miniSEED con serializzazione JSON. 48
3.6 Processo di realizzazione del formato miniSEED con serializzazione binaria. 49
3.7 Rappresentazione grafica dei campi selezionati di un pacchetto miniSEED per la nuova struttura dati μSEED. 51
3.8 Processo di realizzazione del formato μSEED con serializzazione JSON. 53
3.9 Processo di realizzazione del formato μSEED con serializzazione CBOR. 54
3.10 Schema dell'implementazione su container Docker del sistema di simulazione. 57
3.11 Interfaccia grafica dell'applicativo Jupyter Notebook. 57

4.1 Distribuzione di probabilità dei T_l per i protocolli MQTT e SeedLink. Gli istogrammi sono stati normalizzati. 62
4.2 Confronto tra le dimensioni dei pacchetti per i diversi formati dati. . 64
4.3 Confronto tra le dimensioni dei pacchetti per i diversi formati dati con un numero di campioni ridotto. 66

4.4 Rappresentazione del guadagno percentuale, in termini di dimensioni, dei formati CBOR e Pickle rispetto a JSON. 66
4.5 Tempi medi di codifica in modalità offline. 68
4.6 Tempi medi di codifica del publisher su piattaforma docker. La macchina di riferimento è la 1. 69
4.7 Tempi medi di trasmissione su piattaforma docker. La macchina di riferimento è la 1. 71
4.8 Tempi medi di decodifica del subscriber su piattaforma docker. La macchina di riferimento è la 1. 71
4.9 Distribuzione dei tempi, nei diversi formati, per un pacchetto da 2000 campioni su piattaforma docker. La macchina di riferimento è la 1. . 72
4.10 Tempi medi totali su piattaforma docker. La macchina di riferimento è la 1. 73
4.11 Rappresentazione delle distribuzioni temporali in percentuale dei quattro formati. 74
4.12 Tempi medi di codifica del publisher su piattaforma docker. La macchina di riferimento è la 2. 75
4.13 Tempi medi di trasmissione su piattaforma docker. La macchina di riferimento è la 2. 75
4.14 Tempi medi di decodifica del subscriber su piattaforma docker. La macchina di riferimento è la 2. 76
4.15 Confronto distribuzione dei tempi tra la macchina 1 e 2, nei diversi formati, per un pacchetto da 2000 campioni, su piattaforma docker. 77
4.16 Tempi medi totali su piattaforma docker. La macchina di riferimento è la 2. 77
4.17 Tempi medi di codifica del publisher su Raspberry Pi. 78
4.18 Tempi medi totali utilizzando Raspberry Pi come publisher e la macchina 2 come subscriber. 78
4.19 Distribuzione dei tempi nei diversi formati, per un pacchetto da 2000 campioni, utilizzando Raspberry Pi come publisher e la macchina 2 come subscriber. 79

4.20 Differenza tra il formato mB e μSC in termini di tempi di codifica,decodifica e totali medi. simulazione realizzata utilizzando Raspberry Pi come publisher e la macchina 2 come subscriber. 80
4.21 Differenza dei tempi totali tra le simulazioni effettuate nella macchina 1 rispetto alla configurazione con Raspberry e macchina 2. 81
4.22 Distribuzione di probabilità dei tempi totali per un pacchetto da 250 campioni. Gli istogrammi sono stati normalizzati. 82
4.23 Distribuzione di probabilità dei tempi totali per un pacchetto da 64000 campioni. Gli istogrammi sono stati normalizzati. 82

Elenco delle tabelle

3.1 Confronto dei tempi di codifica e decodifica delle diverse implementazioni python di CBOR. I tempi sono in ms. 56

4.1 Statistiche ottenute dalle simulazioni per le latenze dei protocolli MQTT e SeedLink. I tempi sono in ms. 62

Capitolo 1

Introduzione

Le conseguenze catastrofiche che derivano dai terremoti con elevata intensità e l'impossibilità di prevederli, hanno portato negli ultimi decenni allo sviluppo di reti di monitoraggio sismico sempre più articolate, soprattutto nelle regioni del mondo ritenute a maggior rischio. Attraverso il progresso tecnologio dei dispositivi che costituiscono le stazioni di misura, è stato possibile sviluppare tecniche progressivamente più efficaci per un monitoraggio sismico in tempo reale. Questo ha permesso alle reti sismiche di elaborare automaticamente i segnali registrati dai nodi sensori e fornire stime rapide ed affidabili dei parametri di un evento sismico, già durante la sua fase di sviluppo e prima dell'arrivo delle onde sismiche più distruttive. Tale procedura è nota come “earthquake early warning”, abbreviato con la sigla EEW. Il principio dell'EEW è semplice e si basa sulla differente velocità di propagazione delle onde sismiche rispetto a quella dei segnali elettromagnetici, trasmessi via radio o cavo, che veicolano l'informazione sul terremoto. In questi sistemi, una maggiore concentrazione di stazioni nell'area di generazione di terremoti per l'acquisizione in tempo reale delle forme d'onda, potrebbe portare a benefici sia in termini di correttezza dei risultati sia di tempi di rilevamento. Fino a diversi anni fa, rendere una rete più densa in termini di dispositivi, comportava sia costi economici elevati della strumentazione che difficoltà nella gestione dei tempi di comunicazione dei molteplici nodi attraverso le consuete reti di telecomunicazione. Tuttavia, i recenti sviluppi nell'Internet of Things (IoT) hanno aperto la strada a scenari promettenti anche per applicazioni di EEW [1][2]. I dispositivi IoT, grazie alle loro caratteristiche, possono essere integrati all'interno di reti sismiche già esistenti, al fine di consentire un monitoraggio real-time sul territorio più capillare e contribuire ad una rilevazione

più rapida di un evento sismico. I limiti che contraddistinguono i dispositivi IoT sono rappresentati dalle risorse compuazionali, di storage ed eventualmente energetiche, che hanno a disposizione. Allo scopo di ridurre il carico di lavoro ai sistemi IoT viene spesso demandato ad una piattaforma Cloud, che ha virtualmente infinite capacità elaborative e di storage, il compito di eseguire le operazioni più onerose in termini computazionali. Le tecnologie basate sull'IoT dispongono di un panorama applicativo estremamente ampio e questo implica l'utilizzo di protocolli di comunicazione diversificati tra loro. Tra questi, un protocollo che si è ampiamente diffuso in ambito IoT, per le sue caratteristiche, risulta essere MQTT (Message Queuing Telemetry Transport Protocol). Nella prima parte di questo lavoro vengono confrontati, in termini di latenza sulla consegna di un pacchetto, il principale protocollo per lo scambio di rilevazioni sismiche (SeedLink) con il protocollo MQTT. Nelle reti sismiche attuali sono scambiati pacchetti in un formato standard chiamato miniSEED. Il differente protocollo utilizzato (MQTT) però, non permette l'invio diretto di tale formato. Nella seconda parte del lavoro vengono quindi analizzati nuovi diversi formati dati, capaci di contenere le stesse informazioni, valutandone le perfomance mediante simulazione di due client della rete, su diversi dispositivi. In modo da rendere ripetibili tali simulazioni, i client sono stati implementati su piattaforma Docker. Infine, vista l'importanza delle informazioni diffuse da un nodo di misura in applicazioni EEW, è stato effettuato un approfondimento sull'integrità dei dati e ciò che essa comporta.

Capitolo 2

Stato dell'arte

2.1 Sistemi di Earthquake Early Warning

Un terremoto è una vibrazione o assestamento della crosta terrestre, provocato dallo spostamento improvviso di una massa rocciosa nel sottosuolo. I terremoti più distruttivi sono causati da una dislocazione improvvisa di segmenti di crosta che liberano energia sotto forma di onde, le cosiddette onde sismiche. Queste onde, generate nell'ipocentro del terremoto, hanno differenti caratteristiche sia a livello di velocità di propagazione che di potenziale distruttivo. Tali onde sismiche vengono suddivise in onde di volume ed onde di superficie. Le onde di volume si dividono in:

- **Onde primarie (o onde P)**: queste onde sono le più veloci, quindi le prime a raggiungere una stazione di rilevamento. Sono onde compressionali o longitudinali e viaggiano lungo la cosiddetta direzione di propagazione dell'onda. La loro velocità è compresa tra i 4 e gli 8 km/s e la loro energia è relativamente bassa da considerarle generalmente non distruttive;

- **Onde secondarie (o onde S)**: queste onde viaggiano con un movimento di taglio perpendicolare lungo la direzione di propagazione ad una velocità circa il 60% inferiore rispetto alle onde P. Le onde S possono avere energia sufficiente da disporre di potenziale distruttivo anche a diversi chilometri di distanza dall'epicentro del sisma.

Sebbene le due velocità cambino in base allo strato interno della Terra che stanno attraversando, il loro rapporto rimane pressoché costante [3]. Le onde di superficie

possono essere raggruppate in onde di Love e onde di Rayleigh, esse si identificano come quelle maggiormente distruttive, ma sono caratterizzate da una velocità inferiore rispetto alle onde di volume. In Fig.2.1 viene riportato un sismogramma registrato presso la stazione denominata MMUR dell'Istituto Nazionale di Geofisica e Vulcanologia (INGV), durante l'evento del 26-10-2016 di magnitudo (M_w) 5.9 con epicentro a Visso (MC, Italia), nel quale sono evidenziati i tempi di arrivo delle onde P ed S. Misurando il tempo di ritardo che intercorre tra l'arrivo dell'onda P e quello

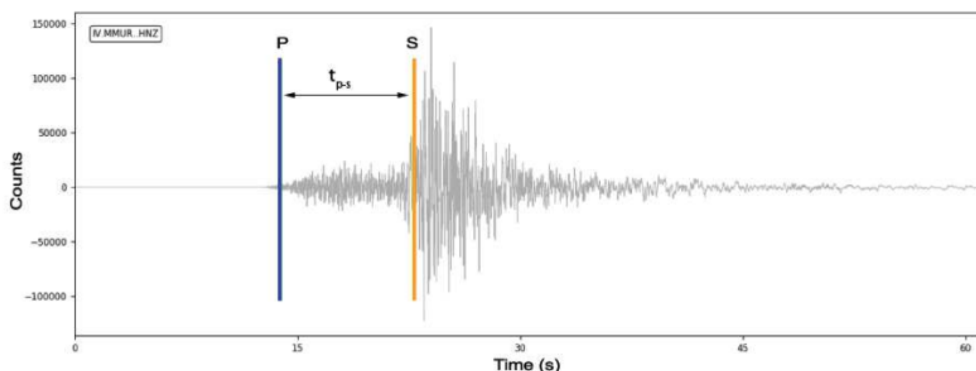


Figura 2.1: Esempio di una forma d'onda sismica registrata dalla stazione MMUR durante un terremoto. Le linee verticali blu ed arancione indicano i tempi di arrivo rispettivamente delle onde P e delle onde S. t_{p-s} indica il tempo tra i due arrivi. L'asse dei tempi è in secondi relativi al 26-10-2016T19:18:04 UTC.

dell'onda S è possibile stimare la distanza a cui ci si trova dall'epicentro. Tramite triangolazione di almeno tre misurazioni dislocate sul territorio si risale con precisione alla posizione esatta. Dalla presenza di questo ritardo temporale tra l'arrivo delle due onde(P ed S) nascono i sistemi di Early warning, progettati per fornire un allerta "istantanea" in seguito ad una stima della magnitudo e dell'epicentro del terremoto basata sulla rilevazione dell'onda P[4]. Il precursore di questi stistemi è stato Cooper nel 1868 la cui idea era basata sul concetto per cui un'onda elettromagnetica viaggia a velocità molto superiori a quelle delle onde sismiche e quindi l'informazione che può trasportre arriva in anticipo [5]. Il termine "earthquake early warning systems" (EEWs) è usato per classificare sistemi real-time in grado di fornire un'allerta anticipata rispetto ad un significativo evento sismico [6]. Avendo le due onde velocità diversa, in funzione della distanza dall'epicentro del sisma, il delay-time, che può essere sfruttato per inviare un allarme ad uno o più target con alcuni secondi (o

decine di secondi) di anticipo rispetto all'arrivo delle onde di ampiezza più rilevante, aumenta. Possono essere sviluppati quindi due tipi di sistemi di allerta: regionali o in loco(direttamente sulla zona ad alto rischio). L'approccio regionale è il più completo, poiché fa leva sulle informazioni provenienti da una rete sismica, dispiegata vicino all'epicentro, raccogliendole in tempo reale in un data center che si occupa di determinare i parametri del sisma e fornire una stima del movimento del suolo regionale mediante l'utilizzo di leggi di attenuazione. L'approccio in loco è più veloce, poiché si basa su una misurazione locale del movimento del suolo sismico, e può fornire un utile allarme tempestivo nei siti a breve distanza dall'epicentro. Questo è reso possibile dall'utilizzo di leggi di scala tra, ad esempio, il picco iniziale di ampiezza dello spostamento registrato e la severità dello scuotimento del suolo atteso. Un l'allarme tempestivo è spesso maggiormente necessario rispetto ad una precisa stima dei parametri sismici[7]. Se l'epicentro quindi è molto vicino alla zona ad alto

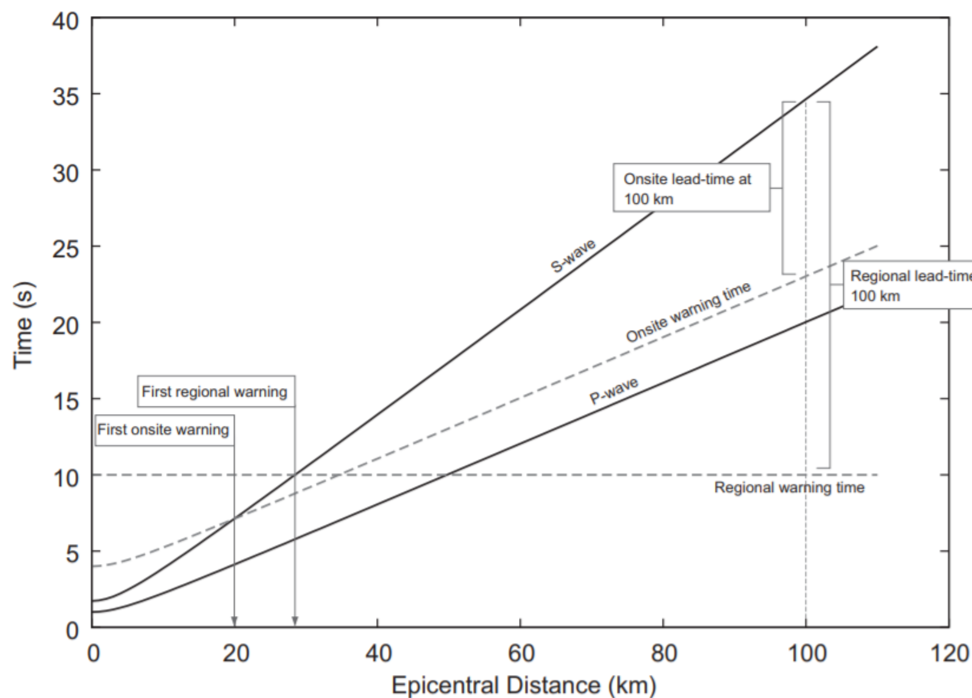


Figura 2.2: Tempi di arrivo dell'onda P e dell'onda S rispetto alla distanza dall'epicentro del sisma.

rischio è più conveniente un sistema in loco che consegue una riposta più rapida, ma in caso contrario, come è riscontrabile dalla Fig.2.2, la strategia regionale permette di avere un maggior tempo di allerta, non dovendo aspettare che l'onda P arrivi

nelle vicinanze dei target da allertare. Quando possibile questi due approcci sono utilizzati contemporaneamente fornendo un grado di sicurezza superiore. Nel cercare di implementare questi meccanismi di allerta, un lasso temporale di pochi secondi può fare la differenza salvando vite umane ed evitando disastri naturali. Appunto per questo, ad alcuni luoghi o infrastrutture, deve essere tenuto un occhio di riguardo classificandoli come più a rischio e predisponendo opportune azioni immediate in seguito alla ricezione di allerta da parte dei sistemi di EEW.

2.1.1 Sistemi di EEW nel mondo

Lo sviluppo di sistemi di EEW è stato guidato dal'evoluzione tecnologica dei dispositivi all'interno delle reti sismiche ma anche da diversi terremoti chiave. Questi fattori hanno portato alla diffusione e al potenziamento dei sistemi di EEW in tutto il mondo. Questi ultimi posso essere suddivisi in funzione della tipologia di allerta fornita: *Public distribution*, *Limited distribution* e $Real - time\ testing$. Mentre nei *Public distribution* l'allerta viene inviata in modalità broadcast attraverso diversi canali verso tutta la popolazione locale, nei *Limited distribution* gli allarmi vengono diramati solamente a determinati gruppi di utenti (organismi pubblici per la gestione dell'emergenza, edifici strategici, ecc.). Infine, in diverse regioni a livello mondiale, dei sistemi sono in fase di costruzione e/o testing e le relative allerte non vengono inoltrate a nessuna tipologia di utenti diversa dalla comunità di sviluppo. Un'analisi dei sistemi di EEW a livello mondiale [8], condotta nel 2019, ha portato alla rappresentazione di tali sistemi identificandoli per modalità di distribuzione dell'allerta(Fig.2.3). Altri studi invece sono stati focalizzati sulla presentazione di questi sistemi e sul loro attuale sviluppo sul territorio Europeo [9].

Il primo sistema a livello mondiale è stato sviluppato in Messico, dove è operativo dal 1991. Questo sistema utilizza il principio della valutazione della più probabile magnitudo di un terremoto rilevato attraverso singole stazioni. Nel momento in cui 2 stazioni triggerano un evento, viene diramata una allerta verso le città dove lo squotimento atteso è superiore rispetto ad una soglia prefissata. Gli avvisi vengono emessi mediante migliaia di ricevitori radio dedicati, distribuiti nelle scuole e negli uffici governativi. A Città del Messico, ad esempio, un allarme pubblico suona attraverso 12.000 sirene dislocate in tutta la città in modo tale da poter essere

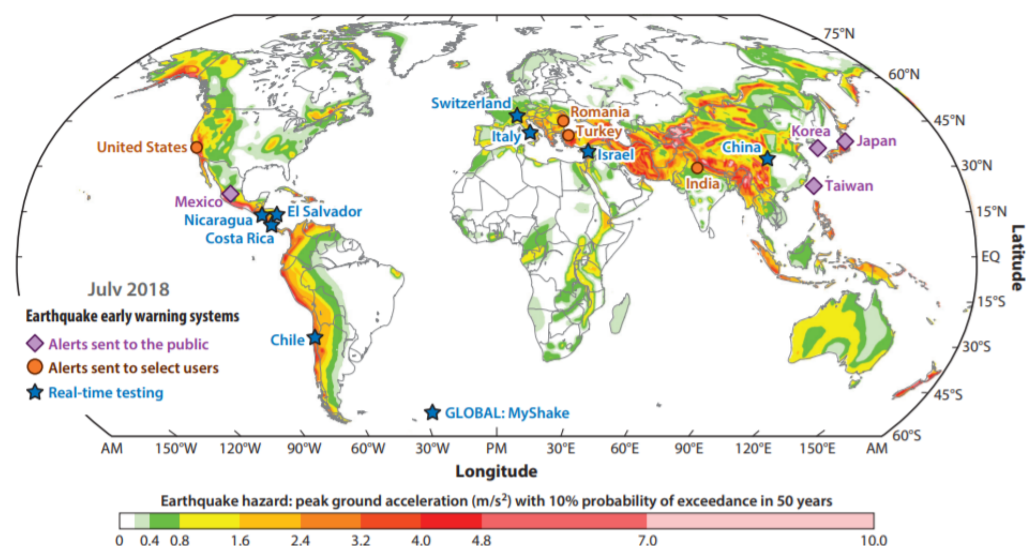


Figura 2.3: Lo stato dei sistemi di EEW nel mondo. Sullo sfondo viene presentato il rischio sismico inteso come accelerazione massima al suolo con probabilità del 10% di superamento in 50 anni.

udite dalla maggior parte dei residenti. La distanza della faglia dai centri abitati consente al sistema di diramare un'allerta anche fino a 60 secondi prima dell'arrivo delle onde distruttive. Per l'evento di M_w 7.3 del 14 settembre 1995, l'allarme è arrivato a Città del Messico con 72 s di anticipo rispetto all'arrivo delle onde S, consentendo l'evacuazione di edifici pubblici come scuole e l'interruzione delle linee metropolitane. In Giappone, il sistema utilizza una combinazione dell'approccio on-site e regionale. Quando una stazione rileva un picco di accelerazione che supera i 100 cm/s^2, viene triggerato un evento ed inizia la caratterizzazione dei parametri di sorgente. I tempi delle stazioni che hanno rilevato un evento vengono confrontati con le stazioni che ancora non lo hanno triggerato, in modo tale da correggere la stima della localizzazione. L'allerta viene diramata qualora l'evento superi una certa intensità attraverso canali dedicati ed anche attraverso app su smartphone, TV e radio. La valutazione di un algoritmo denominato ElarmS è stata condotta in Corea del Sud a partire dal 2008. Attualmente il sistema è in utilizzo e dirama allarmi alla popolazione mediante messaggi SMS di tipo broadcast per terremoti superiori a magnitudo 4. Anche in Taiwan viene utilizzato un sistema basato sull'analisi dei primi secondi delle onde P per determinare una stima della magnitudo dell'evento. La generazione dell'allerta avviene in circa 15 secondi e viene inviato un messaggio a

tutti i telefoni cellulari localizzati ad una distanza minima di 50 km dall'epicentro. Per quanto riguarda i sistemi che invece ancora non inviano un'allerta a tutta la popolazione, ma ad un numero ristretto di utenti, si segnala il sistema ShakeAlert in uso sulla costa Ovest degli Stati Uniti d'America. Si tratta di un sistema distribuito formato da diversi componenti interconnessi. Questa architettura di sistema consente lo sviluppo indipendente di singoli componenti, dalle origini dati agli algoritmi, dall'associazione di eventi alla generazione ed invio di allarmi. Attualmente è stata distribuita un'app su smartphone solamente in alcune regioni. Una distribuzione dell'allerta limitata viene effettuata anche nei sistemi in uso nel nord dell'India ed in Romania. La Cina ha da poco iniziato la fase implementativa di un sistema a livello nazionale con l'impiego di quasi 15000 sensori ed ingenti investimenti a livello economico. In fase di test, infine, si collocano i sistemi del Cile, Israele, Svizzera ed il sistema sviluppato per l'Italia Meridionale, che viene analizzato nella prossima sezione.

2.1.2 Il sistema di EEW in Italia

In Italia è in sperimentazione un sistema di EEW denominato PRESTo (PRobabilistic and Evolutionary early warning SysTem)[10]. Il sistema attualmente è operativo in tempo reale sulla rete sismica dell'Irpinia (ISNet) ed è in fase di test sulla KIGAM network in Corea del Sud, sulla RoNet in Romania, sulla KOERI newtork nella regione della Marmara ed, infine, in un progetto che vede impegnate le regioni a nord-est dell'Italia, la Slovenia e l'Austria sulla CE3N network. La ISNet è composta da 30 stazioni che in tempo reale trasmettono i segnali registrati a dei sistemi di controllo dedicati. Le stazioni sono organizzate in subnets, ognuna delle quali è composta da un massimo di 6 o 7 stazioni connesse a dei Local Control Centers (LCC), connessi a loro volta a un Network Control Center (NCC). Il sistema a livello architetturale è basato su 5 moduli.

1. Acquisizione dati: in ogni LCC è presente un server SeiscomP. Il NCC si collega via protocollo SeedLink (protocollo che verrà illustrato nelle prossime sezioni) ai server ed acquisisce le tre componenti accelerometriche registrate dalle stazioni. Per ogni secondo che un nuovo pacchetto di dati viene ricevuto, il sistema

assegna un fattore di qualità ad ogni stazione, ignorando nelle elaborazioni successive tutte le stazioni che sono considerate di qualità non sufficiente.

2. Picking della fase P ed associazione: per ogni pacchetto ricevuto, il sistema effettua il picking della fase P sulla componente verticale dell'accelerazione mediante un algoritmo denominato FilterPicker. Ogni pick viene memorizzato e se ne controlla la coerenza temporale con altri pick precedentementi memorizzati. Nel momento in cui il numero di pick coerenti supera un numero configurabile di stazioni, si procede alla fase successiva.

3. Localizzazione: per tale fase, PRESTo utilizza una tecnica di localizzazione evolutiva e real-time denominata RTLoc. Tale implementazione permette di stimare la posizione ipocentrale di un evento in base alle stazioni che hanno effettuato il triggering della fase P e quelle che ancora non effettuato tale operazione. Vengono utilizzate griglie 3D precalcolate per ogni stazione contenenti i modelli di velocità ed i travel-time da ogni punto della griglia alla stazione stessa, sia per quanto riguarda le onde P che per le onde S.

4. Stima della magnitudo: le componenti accelerometriche dall'istante di rilevamento della fase P, vengono filtrate e viene ricavato il rispettivo spostamento del suolo mediante doppia integrazione. Successivamente, mediante un algoritmo denominato RTMag, inizia la stima della densità di probabilità del valore di magnitudo.

5. Stima dell'accelerazione di picco verso uno o più punti target: quando si hanno a disposizione le stime dei parametri di sorgente dell'evento è possibile, mediante l'utilizzo di leggi di attenuazione note, stimare il valore più probabile dell'accelerazione di picco ad una certa distanza (i punti target sono anch'essi configurabili) dall'epicentro.

Il sistema può essere utilizzato sia in modalità real-time che in modalità simulazione. Nella prima modalità il sistema si interfaccia direttamente con le rete sismica sottostante come descritto in precedenza, mentre in modalità simulazione è possibile inserire nel sistema dei dati in formato SAC precedentemente acquisiti o generati da una rete sismica. In questo tipo di modalità, PRESTo converte i files in stream

SeedLink della durata di 1 secondo ed è possibile simulare una latenza di rete mediante configurazione. La Fig. 2.4 riporta l'interfaccia grafica dell'applicativo PRESTo, dove nella parte sinistra vengono visualizzate in real-time le forme d'onda delle stazioni configurate. Nella parte destra, la finestra viene suddivisa in due sezioni, in quella superiore è presente una mappa della zona con triangoli verdi o rossi in corrispondenza delle stazioni attive o meno. In modalità simulazione vengono visualizzate in semi trasparenza le stazioni che non sono state considerate dall'utente, per le quali cioè non sono presenti file con registrazioni. Nella parte inferiore, viene riservata una sezione che verrà utilizzata in caso di evento sismico. Nel momento in cui ciò si verifica, nei sismogrammi vengono evidenziate linee verticali rosse in corrispondenze delle fasi P rilevate, mentre nella parte in basso a destra inizia la stima evolutiva di localizzazione e magnitudo. In contemporanea nella mappa viene riportato l'epicentro ed il lead time verso uno o più punti target. Durante il suo

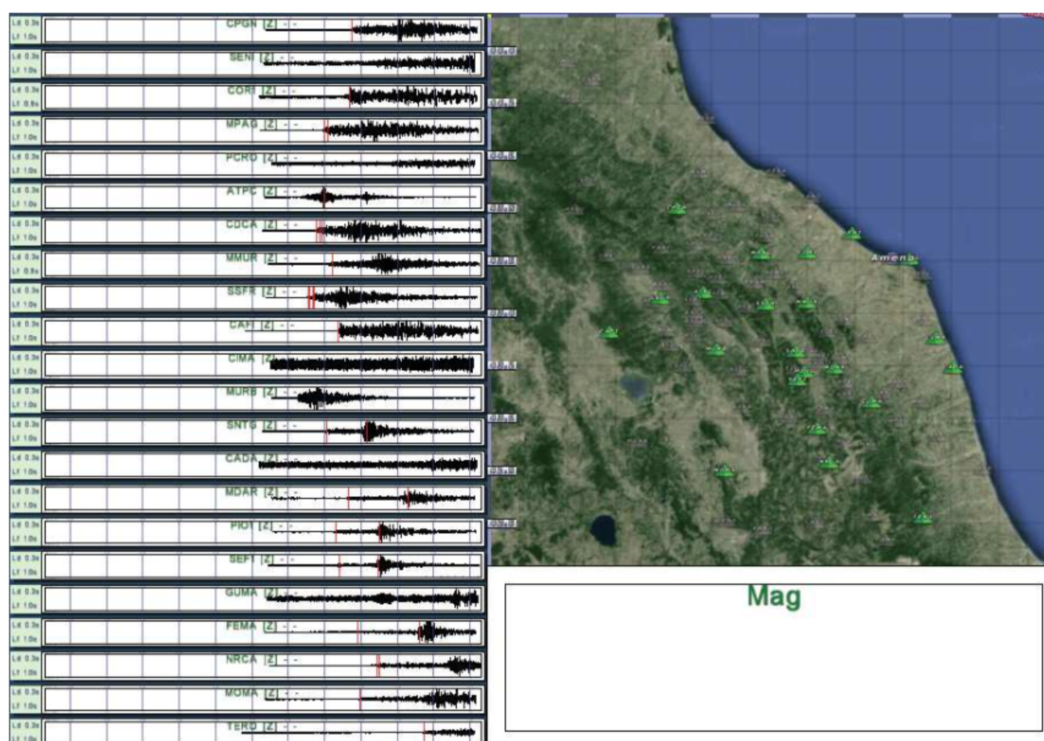


Figura 2.4: Interfaccia grafica PRESTo.

funzionamento, PRESTo genera un rapporto sia dell'effettivo funzionamento delle componenti sia dei calcoli effettuati in un file di log. Tale file contiene l'indicazione temporale di generazione in ogni riga ed è utile sia per il debug dell'applicazione che

per fini statistici in merito alle prestazioni del sistema.

2.2 Formato miniSEED

Il formato miniSEED è un sottoinsieme del più articolato formato SEED (Standard for the Exchange of Earthquake Data), che nasce negli anni ottanta con lo scopo di standardizzare lo scambio di dati, relativi al campo sismologico, tra le varie comunità scientifiche. Il formato SEED si scompone nella parte contenente la serie temporale di dati chiamata miniSEED e la restante parte chiamata dataless SEED. All'interno della dataless SEED vengono inserite informazioni riguardanti la rete,le stazioni di rilevazione con relativa posizione e le risposte degli strumenti di misura. Visto che i file di dataless SEED sono spesso ridondanti, in quanto le infomazioni che contengono restano inalterate per lunghi periodi di tempo, vengono conservati separatamente ai pacchetti miniSEED e sostituiti solamente quando necessario. Ogni file miniSEED è formato una serie di valori misurati, di una certa lunghezza potenza di 2, fissata in precedenza. Le lunghezze comuni sono: 512 byte (per i flussi in tempo reale) e 4096 byte (per l'archiviazione), altre lunghezze vengono utilizzate per scenari particolari. Ogni pacchetto miniSEED inizia con un'intestazione di 48 byte di informazioni descrittive della serie di dati sottostante. Immediatamente dopo questa sezione fissa vengono inseriti ulteriori sezioni chiamate Blockette("blocchetti" di 8 byte), per un totale di 16 byte, che aggiungono ulteriori informazioni sui dati sismici o su come questi devono essere letti ed interpretati. Il miniSEED offre infatti diverse modalità di rappresentazione del dato(16, 24 o 32 bit) e diversi livelli di compressione dell'informazione a seconda dell'algoritmo impiegato, tra cui STEIM1 e STEIM2(modalità di compressione senza perdita) [11]. La compressione del dato dipendente dalla velocità con cui il segnale varia, per cui il numero di campioni all'interno di un pacchetto non è costante e può quindi coprire un intervallo temporale di diversa durata, non necessariamente un multiplo esatto di un secondo. Nel formato SEED complessivo esistono molti tipi di "blocchetti" ma in miniSEED ne vengono utilizzati solo due: blockette 1000 e blockette 1001. I blocchetti 1000 e 1001 ci informano sul tipo di compressione utilizzata, sulla grandezza del pacchetto dati e su offset temporali da applicare alla lettura delle misure. Una rappresentazione grafica

di un pacchetto miniSEED è disponibile in Figura 2.5. Le dimensioni delle varie

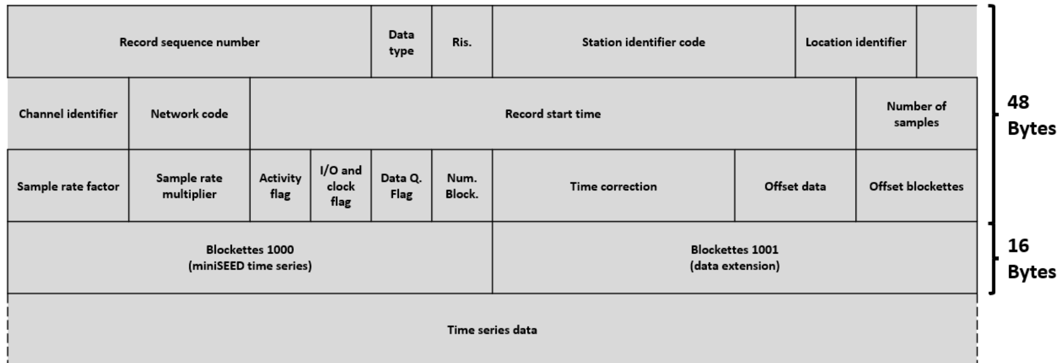


Figura 2.5: Rappresentazione grafica di un pacchetto miniSEED.

sezioni presenti nei primi 48 byte dell'intestazione del pacchetto sono specificate in Fig.2.6.

Fixed section of (waveform) data header		48 total
1) Record sequence number	(^^^^^1 etc.)	6
2) Data header/quality indicator	['D' | 'R' | 'Q' | 'M']	1
3) Reserved byte	(0)	1
4) Station identifier code	('SSSSS')	5
5) Location identifier	('LL')	2
6) Channel identifier	('CCC')	3
7) Network Code	('NN')	2
8) Record start time	(BTIME)	10
9) Number of samples	(UWORD)	2
10) Sample rate factor	(200)	2
11) Sample rate multiplier	(1)	2
12) Activity flags	(UBYTE)	1
13) I/O and clock flags	(UBYTE)	1
14) Data quality flags	(UBYTE)	1
15) Number of blockettes that follow	(UBYTE)	1
16) Time correction	(0)	4
17) Beginning of data	(48)	2
18) First block	(1)	2

Figura 2.6: Elenco delle sezioni dell'intestazione di un pacchetto miniSEED.

2.2.1 Architettura convenzionale della rete

Una rete sismica è convenzionalmente organizzata con una struttura ad albero composta da:

- **Rete**;
- **Stazione**;
- **Posizione**;
- **Canale**.

Il nome di una stazione si riferisce a un luogo fisico in cui sono raggruppati diversi strumenti di misura e viene abbreviato con un massimo di 5 lettere (per esempio, NRCA sta per stazione sismica di Norcia). Un gruppo di più stazioni è definito come rete e il suo acronimo è di massimo due lettere assegnate dalla FDSN(International Federation of Digital Seismic Networks). Un esempio pertinente al precedente è l'abbreviazione IV che rappresenta la rete sismica italiana. Il canale per una determinata serie temporale è rappresentato da un singolo flusso di dati. Mediante l'utilizzo di tre caratteri viene descritto un certo canale al fine di esprimere la relativa banda di frequenza, la frequenza di campionamento, il codice dello strumento e il codice di orientamento dello stesso. Infine la posizione, abbreviata con due caratteri o numeri, viene utilizzata per distingure strumentazioni o canali simili all'interno della stessa stazione[12]. In Figura 2.7 è rappresentato schematicamente un esempio di come viene convenzionalmente organizzata una rete sismica.

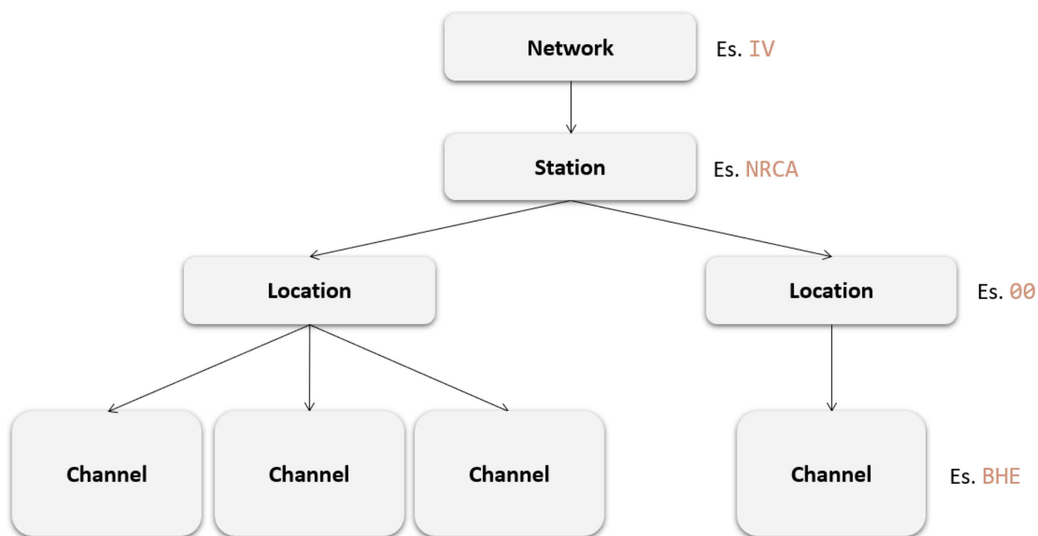


Figura 2.7: Struttura convenzionale di una rete sismica.

2.3 Protocollo Seedlink

SeedLink è composto da un protocollo di acquisizione dati in tempo reale e un software client-server che implementa lo stesso. Il protocollo SeedLink è basato su TCP(Transmission Control Protocol) e tutte le connessioni vengono avviate da parte del client. Durante la fase di handshake il client può sottoscrivere stazioni e flussi

specifici utilizzando semplici comandi in codifica ASCII. Al termine dell'handshaking, al client viene inviato un flusso di "pacchetti" SeedLink costituito da un'intestazione di 8 byte (contenente il numero di sequenza) seguito da un record miniSEED da 512 byte, dimensione standard per lo streaming. Uno degli svantaggi più evidenti del protocollo riguarda proprio il fatto che i dati vengano inviati in pacchetti miniSeed da 512 byte. Nei datalogger pertanto possono essere scelte due vie alternative, cioè attendere il riempimento completo del pacchetto oppure attendere un intervallo temporale ed inviare il pacchetto inserendo contenuto nullo fino alla sua dimensione massima. Se da un lato la prima modalità consente di sfruttare a pieno la banda disponibile, dall'altro la seconda modalità risulta essere più appropriata in un contesto di Early Warning, mantenendo i ritardi dovuti alla pacchettizzazione sull'ordine del secondo. Sono state realizzate nel tempo varie implementazioni di questo protocollo, non tutte compatibili tra loro in quanto in alcune non è stata implementata la versione integrale.

2.3.1 Descrizione del protocollo

Una sessione Seedlink inizia con un apertura di connessione TCP/IP e termina con la chiusura della connessione TCP/IP. Una volta stabilita la connessione il server attende l'inizio della comunicazione(handshaking) da parte del client senza inviargli nulla. Durante questa fase il client invia al server dei comandi di setup per configurarlo, poi eventualmente inizia la trasmissione dati. Ad ogni comando ricevuto il server risponde con una conferma o un errore senza i quali il client non inoltra altri comandi. Tra i comandi che il client può inviare al server SeedLink è inclusa la selezione del flusso secondo il Seed Station Naming Convention (SSNC). Questa convenzione assegna codici appropriati per identificare un flusso attraverso la sequenza di codice di rete, codice stazione, codice posizione, codice canale. Ovviamente il client attende la risposta per un certo tempo dopodiché, in caso di timeout o di errore della rete, la sessione viene chiusa e ne viene iniziata una nuova. Una volta completato l'handshaking vengono inviati i pacchetti Seedlink il cui header di 8 byte è una stringa ASCII che inizia con le due lettere "SL" seguite da 6 cifre esadecimali rappresentanti la numerzione di pacchetto propria della stazione(server). C'é la possibilità che più stazioni comunichino sullo stesso canale TCP, in questo caso il client, per risalire al

mittente del pacchetto, deve controllare il contenuto dell'header miniSEED. I dati vengono trasferiti come flusso continuo senza rilevamento di errori o controllo del flusso perché queste funzioni sono eseguite dal protocollo TCP. Ciò garantisce la massima velocità di trasferimento dati possibile con lo specifico hardware. Quando il server invia l'ultimo pacchetto, aggiunge in coda ad esso la stringa ASCII "END" per poi attendere che il client la riceva e chiuda di conseguenza la connessione[13]. La caratteristica fondamentale del protocollo è costituita dalla robustezza, deve infatti tollerare client che possono disconnettersi e riconnettersi in maniera aleatoria e provvedere affinché non vi sia perdita di dati. Le eventuali trasmissioni perse possono essere recuperate purché esse siano ancora presenti nei buffer del server. Il server di una connessione Seedlink può essere implementato attraverso l'utilizzo di Ringserver.

2.3.2 Ringserver

Ringserver è un buffer circolare basato su TCP progettato per lo streaming di dati in pacchetti. Le caratteristiche principali di questa implementazione sono la scalabilità e la stabilità. Questo viene utilizzato principalmente nel trasporto di pacchetti(quasi real-time) contenenti serie di dati temporali. Il buffer circolare è implementato con architettura FIFO(First In First Out), non è richiesto inoltre un formato dati specifico. I protocolli supportati sono tutti basati su TCP: DataLink, SeedLink e HTTP/WebSocket. Il server è configurato direttamente da riga di comando o utilizzando un file di configurazione ringserver (o entrambi). Sebbene l'architettura implementata da ringserver sia completamente generica, sono state apportate alcune aggiunte per supportare il protocollo SeedLink. In particolare per i pacchetti in cui i dati sono record miniSEED da 512 byte, il server può essere configurato per servire i vari stream tramite SeedLink. Questa opzione deve essere abilitata, l'ascolto delle connessioni SeedLink non viene eseguito come impostazione predefinita. In generale, i client si connettono al server e inviano una "sottoscrizione" per flussi di dati specifici inviati da altri client. Utilizzando un identificativo per ogni pacchetto nel buffer, per il client è possibile selezionare in modo univoco il punto di partenza del buffer cosi che, in caso di connessioni interrotte, lo straming possa riprendere senza perdita di dati.

2.4 Protocolli per l'Internet of Things

L'Internet of Things (IoT) consiste in un insieme di reti e dispositivi connessi tramite Internet. Nasce con l'obbiettivo di far comunicare tra loro dispositivi in grado di scambiare ed analizzare informazioni sull'ambiente che li circonda. Le tecnologie basate su IoT sono in continuo sviluppo in campi come l'ingegneria, l'agricoltura[14], la medicina[15] e tanti altri, alcuni tutt'ora da esplorare. Tramite l'IoT ogni oggetto reale ha un proprio identificativo unico che permette di individuarlo nella rete. Il numero di dispositivi connessi ad internet continuerà sempre più ad aumentare e con loro anche i dati che vengono scambiati. Ci si aspetta che tutto ciò porterà ad un incremento costante dell'intelligenza dei sistemi connessi tra loro e di conseguenza dei servizi disponibili. Inoltre, anche i dispositivi affacciati alla rete sono in continuo sviluppo con potenze di calcolo sempre superiori e dimensioni sempre più piccole. La miniaturizzazione rende possibile l'equipaggiamento di svariati sensori ed attuatori e l'aggiunta di potenza di calcolo rende il dispositivo parte integrante della rete. Questi elementi della rete quindi, per poter essere efficienti, devono essere affiancati da tecniche di comunicazione M2M(Machine-to-Machine) all'avanguardia. Infatti,uno dei principali fattori che determinano le prestazioni della comunicazione M2M(Machine-to-Machine), è il protocollo messaggistica appositamente progettato per applicazioni IoT. Per poterlo scegliere è necessario conoscere approfonditamente il caso d'uso in cui ci si trova relativamente ai suoi particolari requisiti. Rispetto al Web che utilizza un unico protocollo di messaggistica comune(HTTP), nell'IoT esistono centinaia di protocolli per soddisfare diverse richieste. Ognuno di essi si colloca ad un certo livello della pila protocollare di rete con precisi compiti[16].

Questo, in una visione globale nella quale le "smart things" popolano la rete, potrebbe rappresentare un limite impedendo la comunicazione tra protocolli(proprietari e non) non compatibili [17]. Andando ad evidenziare quelli che operano a livello trasporto e superiori troviamo: MQTT (Message Queuing Telemetry Transport Protocol),CoAP (Constrained Application Protocol),AMQP (Advanced Message Queuing Protocol),HTTP (Hyper Text Transport Protocol) e altri. Nella tabella seguente vengono comparate le caratteristiche dei protocolli in ambito IoT appena citati[18].

Come illustrato, HTTP nato come protocollo Web e non per Iot, ha la dimensione

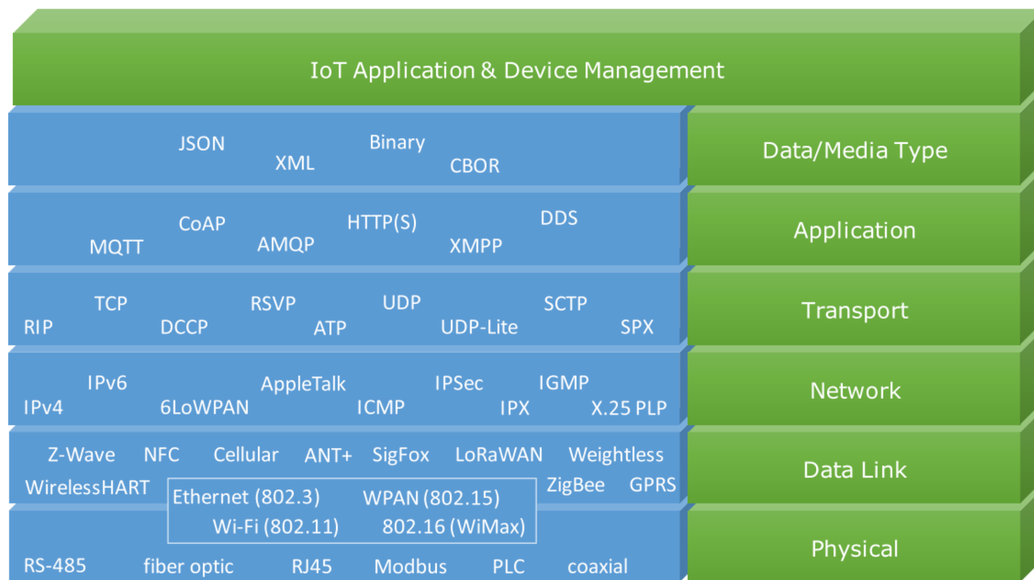


Figura 2.8: Elenco protocolli IoT distribuiti sui livelli di appartenenza.

più grande per quanto riguarda l'intestazione(overhead) e la dimensione del pacchetto mentre, all'opposto CoAP,utilizzando come protocollo di trasporto UDP(User Datagram Protocol), ha complessivamente le dimensioni più piccole. C'é da notare infatti che MQTT,AMQP e HTTP si basano su TCP a livello trasporto e ciò implica un overhead maggiore per la creazione e la chiusura della connessione. Tra queste tre quindi MQTT è la più leggera rispetto all'overhead per messaggio di soli 2 byte. AMQP è un protocollo binario leggero caratterizzato dalla capacità di fornire sicurezza,affidabilità e interoperabilità che comportano però un aumento di overhead e grandezza del messaggio. MQTT e CoAP sono stati sviluppati per essere utilizzati in dispositivi con largezza di banda e risorse limitate e soprattutto con bassa latenza. Per quanto riguarda la Quality of Services (QoSs) e l'interoperabilità MQTT offre le prestazioni migliori, tra i protocolli citati, utilizzando a livello trasporto TCP, che garantisce la consegna del pacchetto. Come verrà illustrato nella prossima sezione, MQTT dispone di più livelli di QoS. A livello di sicurezza MQTT fornisce però un basso livello di sicurezza attraverso una semplice autenticazione con username e password, rispetto agli altri in cui risulta superiore. Da questa breve descrizione dei protocolli utilizzati per l'Iot è comprensibile il perchè, MQTT, si stia sempre più impiegando come protocollo M2M, anche se ancora non è diventato uno standard mondiale come HTTP, il quale però non ha le migliori performance in IoT[18]. Rispetto a HTTP e

Criteria	MQTT	CoAP	AMQP	HTTP
1. Year	1999	2010	2003	1997
2. Architecture	Client/Broker	Client/Server or Client/Broker	Client/Broker or Client/Server	Client/Server
3. Abstraction	Publish/Subscribe	Request/Response or Publish/Subscribe	Publish/Subscribe or Request/Response	Request/Response
4. Header Size	2 Byte	4 Byte	8 Byte	Undefined
5. Message Size	Small and Undefined (up to 256 MB maximum size)	Small and Undefined (normally small to fit in single IP datagram)	Negotiable and Undefined	Large and Undefined (depends on the web server or the programming technology)
6. Semantics/ Methods	Connect, Disconnect, Publish, Subscribe, Unsubscribe, Close	Get, Post, Put, Delete	Consume, Deliver, Publish, Get, Select, Ack, Delete, Nack, Recover, Reject, Open, Close	Get, Post, Head, Put, Patch, Options, Connect, Delete
7. Cache and Proxy Support	Partial	Yes	Yes	Yes
8. Quality of Service (QoS)/ Reliability	QoS 0 - At most once (Fire-and-Forget), QoS 1 - At least once, QoS 2 - Exactly once	Confirmable Message (similar to At most once) or Non-confirmable Message (similar to At least once)	Settle Format (similar to At most once) or Unsettle Format (similar to At least once)	Limited (via Transport Protocol - TCP)
9. Standards	OASIS, Eclipse Foundations	IETF, Eclipse Foundation	OASIS, ISO/IEC	IETF and W3C
10. Transport Protocol	TCP (MQTT-SN can use UDP)	UDP, SCTP	TCP, SCTP	TCP
11. Security	TLS/SSL	DTLS, IPSec	TLS/SSL, IPSec, SASL	TLS/SSL
12. Default Port	1883/ 8883 (TLS/SSL)	5683 (UDP Port)/ 5684 (DLTS)	5671 (TLS/SSL), 5672	80/ 443 (TLS/SSL)
13. Encoding Format	Binary	Binary	Binary	Text
14. Licensing Model	Open Source	Open Source	Open Source	Free
15. Organisational Support	IBM, Facebook, Eurotech, Cisco, Red Hat, Software AG, Tibco, ITSO, M2Mi, Amazon Web Services (AWS), InduSoft, Fiorano	Large Web Community Support, Cisco, Contiki, Erika, IoTivity	Microsoft , JP Morgan, Bank of America, Barclays, Goldman Sachs, Credit Suisse	Global Web Protocol Standard

Figura 2.9: Comparazione delle caratteristiche dei principali protocolli per l'IoT.

CoAP che utilizzano un protocollo di tipo request/response, MQTT, non utilizzando una connessione diretta tra i client, può essere scelto in situazioni in cui la rete non è costantemente disponibile. I risultati dell'indagine svolta annualmente da Eclipse Foundation e Eclipse IoT Working Group tra il 2016 e il 2018 su un gruppo di partecipanti, ai quali è stato chiesto quale protocollo di messaggistica utilizzassero per le loro soluzioni IoT, mostra come HTTP rimanga il protocollo più utilizzato anche se in lieve calo negli anni, mentre all'opposto MQTT, nel 2017 e 2018, riscontra un significativo aumento[19]. Un altro aspetto evidente è la differenza di utilizzo dei due protocolli appena citati rispetto a tutti gli altri(figura 2.10).

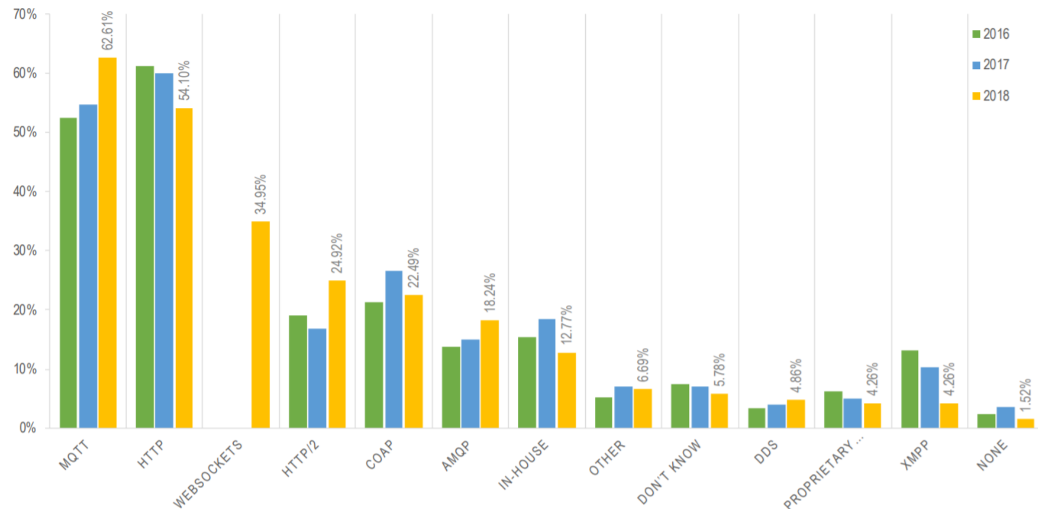


Figura 2.10: Trend di utilizzo dei protocolli di messaggistica in ambito IoT tra il 2016 e il 2018.

2.4.1 Il protocollo MQTT

Il protocollo di messaggistica MQTT è stato per la prima volta introdotto nel 1999 da Andy Stanford-Clark di IBM e Arlen Nipper. Nel 2013, l'MQTT divenne standard protocollare dell'Organizzazione OASIS (Organization for the Advancement of Structured Information Standards). L'idea di partenza fù quella di creare un protocollo che minimizzasse la banda di rete utilizzata, le risorse computazionali necessarie e i consumi di energia, conservando contemporaneamente affidabilità nella consegna dei pacchetti. Inoltre,essendo leggero in termini di dimensione delle librerie,open source e semplice anche da implementare, risulta ideale in contesti più vincolati come nelle comunicazioni M2M e più in generale, nell'internet delle cose. Come già detto, MQTT si appoggia sul protocollo TCP/IP, il quale gli fornisce ordine di consegna senza perdite e una connessione bidirezionale. Le porte che utilizza sono la 8883 per connessioni SSL(Secure Sockets Layer)/TLS(Transport Layer Security) o la 1883 per connesioni non TLS. MQTT è composto da tre figure fondamentali:

1. **Publisher**

2. **Broker**

3. **Subscriber**

dove Publisher e Subscriber sono visti come client MQTT mentre il broker rappresenta il server MQTT. I due client quindi non instaurano una connessione diretta ma comunicano attraverso il broker/server, una volta che viene pubblicato un pacchetto, questo arriva al server che a sua volta lo inoltra al secondo client. Il client che crea messaggi e li pubblica(aprendo una connessione) al broker prende il nome di Publisher mentre tutti i client che desiderano ricevere questi messaggi vengono chiamati Subscriber. Publisher e Subscriber sono quindi disaccoppiati, non si conoscono tra loro e non hanno bisogno di essere sincronizzati nella comunicazione. Questo

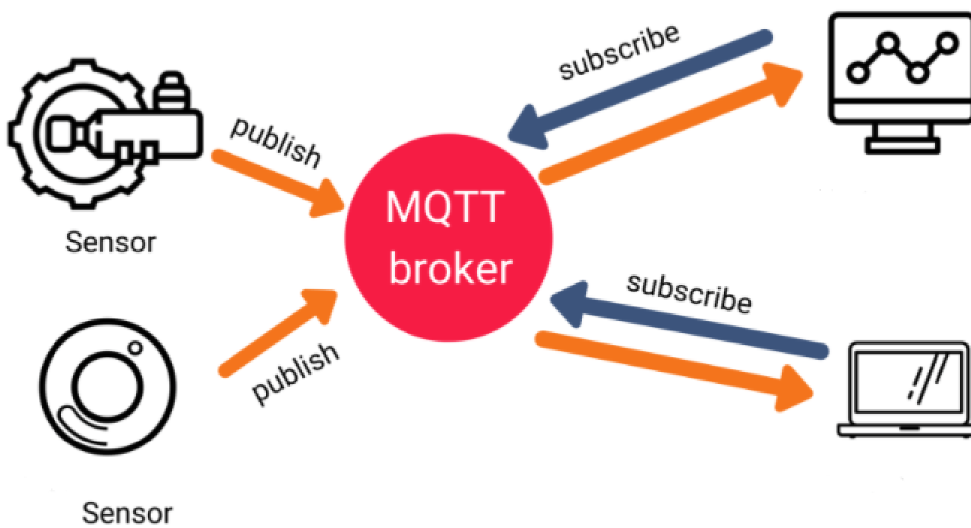


Figura 2.11: Esempio di utilizzo del protocollo MQTT.

approccio, rispetto a quello client-server tradizionale, ci permette di avere maggiore scalabilità della rete. In scenari come nell'IoT dove le reti assumono grandi dimensioni, un solo broker potrebbe non essere sufficiente, creando un collo di bottiglia della rete, è stata così implementata anche una versione dell'MQTT multi-broker chiamata MQTT-ST, che fornisce una gestione ad albero dei broker[20]. Il Subscriber ha la possibilità di filtrare i messaggi che vuole ricevere, ovvero quelli che il broker non deve inoltrargli, attraverso il topic. Quest'ultimo è una stringa, con codifica UTF-8, che il broker utilizza per filtrare ogni connessione da parte dei client. Il topic è impostato dal Publisher come allegato del messaggio mentre, lato Subscriber, viene utilizzata per sottoscriversi alla connessione. Come il Publisher può inviare messaggi in differenti topic, il Subscriber sceglie tramite un operazione iniziale di sottoscrizione, solo quelli di suo interesse. In fase di configurazione, il topic,

viene strutturato gerarchicamente a più livelli sfruttando il carattere "/" per la loro suddivisione, la Fig.2.12 ne rappresenta un esempio. Nel momento in cui un client va

Figura 2.12: Esempio di struttura del topic.

ad effettuare una sottoscrizione al broker, sfruttando la struttura gerarchica, basta inserire un carattere di '#' in un determinato livello, per ricevere messaggi da tutti i topic sottostanti. Il broker quindi, non necessita di una inizializzazione da parte dei client, è sufficiente che il topic sia una stringa valida. L'organizzazione dei nodi Publisher risulta quindi molto semplice per mezzo della struttura gerarchica del topic. MQTT offre tre livelli di QoS per la consegna dei messaggi:

- **QoS = 0**: conosciuto come "at most once message delivery service" dove il messaggio viene inoltrato almeno una volta con la possibilità reale che venga perso;
- **QoS = 1**: conosciuto come "at least once message delivery service" dove il messaggio viene inoltrato e ricevuto almeno una volta, ciò implica che il client Subscriber potrebbe ricevere più volte lo stesso messaggio;
- **QoS = 2**: conosciuto come "exactly once message delivery service" dove il messaggio viene ricevuto, lato Subscriber, una sola volta.

Come si nota dalla figura 2.13, in base al livello di QoS desiderato, vengono scambiati più o meno pacchetti per ogni messaggio pubblicato [21]. La $QoS = 1$ richiede due scambi di pacchetti per una singola pubblicazione di un messaggio, questo perchè dopo la pubblicazione il client attende un acknowledgement (PUBACK) da parte del destinatario(in questo caso è il broker ma lato Subscriber i ruoli si invertono). Una volta ricevuto il PUBACK il client elimina il messaggio dalla coda di uscita, in caso contrario, rispedisce il messaggio con il flag DUP(Duplicate Flag) attivato. Il messaggio continuerà a essere reinviato a intervalli regolari, fino a quando il mittente

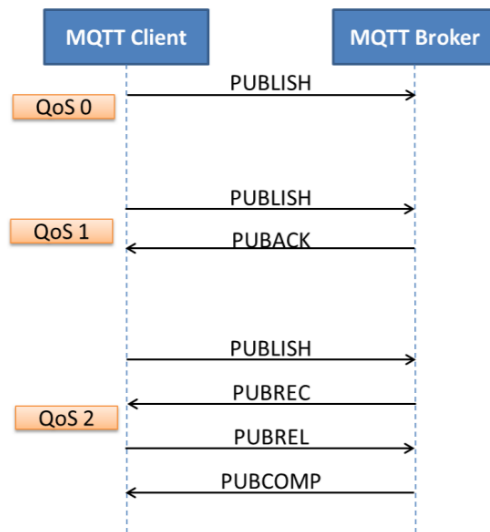


Figura 2.13: Handshake per i diversi livelli di QoS.

non riceverà una conferma. Con la $QoS = 2$ la procedura rallenta richiedento 4 pacchetti scambiati. In questo caso, quando il broker riceve il messaggio, non lo inoltra direttamente agli iscritti al topic, ma invia prima un messaggio di ricezione al client (PUBREC). Dopodiché lo stesso client, ricevendo questa conferma, invia un pacchetto di rilascio del messaggio (PUBREL), dicendo al broker che può inoltrare il messaggio a tutti i Subscriber. Il broker, ricevuto il PUBREL, manda un pacchetto di conferma al client(PUBCOMP) che può quindi cancellare il pacchetto dalla coda di uscita. Appare evidente che il ritardo end-to-end aumenti con una QoS superiore, come verificato inoltre in [22] e di pari passo andranno anche i consumi energetici [23]. Questi aspetti ci permettono di sottolineare l'importanza nella scelta della QoS per dispositivi con risorse limitate. Come accennato nella sezione precedente, MQTT, permette uno scambio di messaggi anche quando la connessione alla rete non è costantemente disponibile. Questo grazie ad un flag (RETAIN), settato per informare il broker che il messaggio deve essere conservato nel caso nessun client sia sottoscritto al topic(potrebbe essere temporaneamente offline), per poi essere inoltrato in un momento successivo. Il formato del pacchetto nel protocollo MQTT è composto, nella sua forma più piccola, da un header fisso di 2 byte, al quale possono essere aggiunti altri byte di header per specifiche funzioni, seguiti dal payload che può avere una dimensione massima di 256 MB. Nel tempo sono state standardizzate alcune versioni del protocollo MQTT : MQTT-SN v1.2 standardizzata da IBM, MQTT

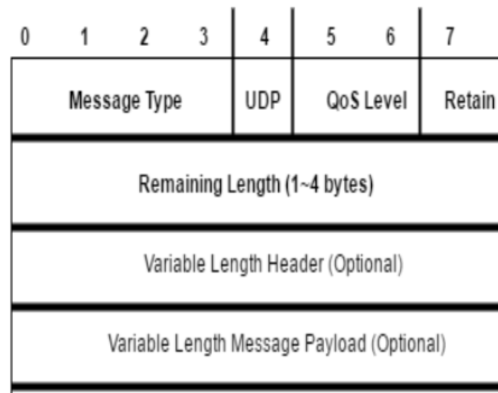


Figura 2.14: Formato del pacchetto MQTT.

v3.1 standardizzata da Eurotech e IBM, MQTT v3.1.1 e MQTT v5.0 standardizzate da OASIS. Da sottolineare è la variante MQTT-SN(MQTT for sensor networks) sviluppata per essere utilizzata principalmente in reti wireless. Si differenzia da MQTT per la riduzione del payload del messaggio e per impiegare il protocollo UDP a livello trasporto. Di questi standard ne sono state implementate diverse versioni, tra cui alcune anche open-source, che integrano tutte o solo alcune delle versioni di MQTT [24]. Tra queste è presente Mosquitto, un implementazione del broker MQTT open-source con a disposizione le versioni 5.0, 3.1.1 e 3.1 sviluppato da Eclipse Foundation [25]. Mosquitto è scritto in linguaggio C, risulta leggero in termini computazionali e di memoria, quindi è possibile impiegarlo sia in dispositivi a bassa potenza che in server di grandi dimensioni. Le caratteristiche di Mosquitto gli consentono di essere uno tra i broker MQTT più scaricati in rete. Per quanto riguarda la realizzazione del client, vengono fornite svariate librerie, scritte in diversi linguaggi, tra cui una delle più famose è Eclipse Paho project. Il progetto Eclipse Paho fornisce implementazioni open source affidabili di protocolli di messaggistica aperti e di standard destinati ad applicazioni nuove, esistenti ed emergenti per Machine-to-Machine (M2M) e Internet of Things (IoT). Paho contiene implementazioni client di Publisher/Subscriber MQTT da utilizzare su piattaforme embedded, insieme al corrispondente supporto server.

2.5 Rappresentazione dei dati

I sistemi informatici possono variare nella loro architettura hardware, nel sistema operativo e nei meccanismi di indirizzamento. Anche le rappresentazioni interne dei dati variano di conseguenza in ogni ambiente. L'archiviazione e lo scambio di dati tra ambienti così diversi richiede un formato indipendente dalla piattaforma e dal linguaggio, in modo tale da essere comprensibile a tutti i sistemi. I dati di un computer sono generalmente organizzati in strutture come array, grafici, classi o altre configurazioni efficienti. Quando le strutture dati devono essere archiviate o trasmesse ad un'altra posizione, ad esempio attraverso una rete, devono passare attraverso un processo chiamato serializzazione. Il termine serializzazione indica il processo di conversione di un oggetto in un flusso di byte, allo scopo di archiviare tale oggetto o trasmetterlo. Nella programmazione ad oggetti, gli oggetti svolgono la funzione di racchiudere in un'unica unità organizzativa sia i dati che le funzioni. Il fine principale della serializzazione è salvare lo stato di un oggetto per consentirne la ricreazione in caso di necessità. Il processo inverso è denominato deserializzazione. Nel caso della trasmissione in rete, questa elaborazione dei dati svolge un ruolo importante nella riduzione delle dimensioni del payload del messaggio, che aiuta a ridurre i tempi di trasferimento e con essi il rischio di perdida dei pacchetti. Ogni tipo di serializzazione ha caratteristiche differenti, che devono essere tenute in considerazione durante il processo di sviluppo in un determinato ambiente. Infatti, devono essere analizzati gli aspetti riguardanti:

1. **Dimensione**: la dimensione del flusso di byte risultante dalla serializzazione, diventa sempre più importante in dipositivi portatili con memoria limitata, che devono salvare grandi quantità di dati, o in quelli che hanno risorse limitate e necessitano di inviare in rete pacchetti di dimensione minima. La dimensione, oltre che dal tipo di serializzazione utilizzato, dipende dalla tipologia di oggetto che teniamo in considerazione.

2. **Tempo di serializzazione/deserializzazione**: per poter trasformare un oggetto in un flusso di dati vengono utilizzati opportuni schemi che permettono di rendere un oggetto come una sequenza di byte. Diversi tipi di serializzazione hanno conseguentemente diversi schemi di processamento che richiedono tempi

più o meno estesi. Per esempio, nel caso di comunicazioni real-time, questi tempi devono risultare più piccoli possibile.

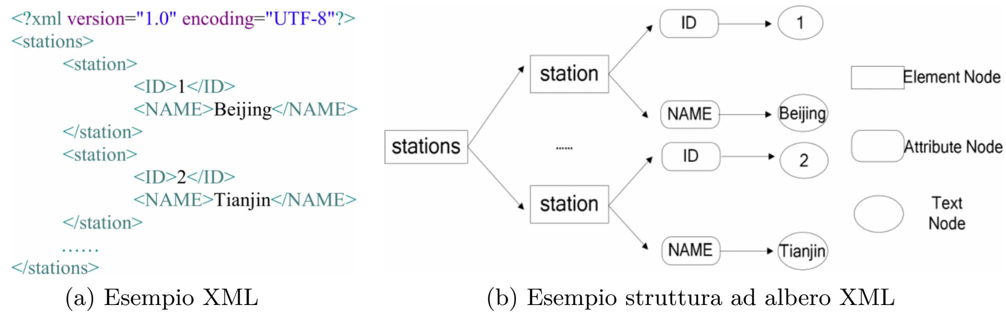
Esistono due tipi di serializzazione, basata su testo e binaria. La prima tipologia porta con se il vantaggio di essere un formato leggibile dall'uomo, strutturando i dati in coppie chiave-valore. La serializzazione binaria, come suggerisce il nome, rappresenta le coppie chiave-valore con un formato binario non comprensibile dall'uomo. Spesso i risultati di una serializzazione di tipo binario evidenziano una notevole riduzione della dimensione del flusso dati. In alcune tipologie di serializzazione binaria, mediante l'utilizzo di schemi per la codifica e decodifica dei dati, è possibile rimuovere le chiavi delle coppie. L'utilizzo di uno schema consente un'ulteriore riduzione delle dimensioni, può tuttavia risultare un limite nella diffusione di un formato dati.

2.5.1 XML

XML è l'abbreviazione di Extensible Markup Language, un formato di testo semplice e molto flessibile derivato da SGML(Standard Generalized Markup Language). Il "progetto XML", ebbe inizio alla fine degli anni novanta e nel febbraio del 1998 è stato riconosciuto come standard W3C(World Wide Web Consortium) con il nome di Extensible Mark-up Language, versione 1.0. Ben presto ci si accorse che XML non era limitato al solo contesto web ma era qualcosa di più: uno strumento che permetteva di essere utilizzato nei più diversi contesti, dalla definizione della struttura di documenti, allo scambio delle informazioni tra sistemi diversi, dalla rappresentazione di immagini alla definizione di strutture dati. XML utilizza elementi e attributi per descrivere i dati organizzandoli con una struttura ad albero. Nel processo di trasferimento dei dati, XML ha sempre mantenuto relazioni gerarchiche che mostrano la relazione subordinata tra gli elementi in modo chiaro e semplice[26], come rappresentato in Fig.2.15 .

2.5.2 JSON

JSON(JavaScript Object Notation) è un formato di interscambio dati leggero, di facile lettura e scrittura. Le caratteristiche più importanti che lo contraddistinguono da XML sono la semplicità di generazione e analisi della struttura, attraverso le



(a) Esempio XML (b) Esempio struttura ad albero XML

Figura 2.15: Esempio serializzazione XML.

quali è possibile presentare i dati in maniera più fluida da parte delle applicazioni. Generalmente JSON per rappresentare la stessa struttura XML richiede meno caratteri. Infatti,JSON è un formato autodescrittivo, come XML, ovvero non è necessario uno schema o una definizione esterna per interpretare i dati codificati, ma rispetto a quest'ultimo risulta molto meno complesso. Questo formato si basa su un sottoinsieme di JavaScript (Programming Language, Standard ECMA-262 3Rd Edition December, 1999), utilizzando un formato di testo che è completamente indipendente dal linguaggio di programmazione. Queste proprietà hanno reso JSON il formato principale per lo scambio di informazioni sul web. Dal punto di vista sintattico, JSON è molto simile alla sintassi letterale degli oggetti di JavaScript. Gli oggetti JSON iniziano con l'apertura di una parentesi graffa e terminano con la sua chiusura. Tra le parentesi graffe sono presenti coppie chiave/valore chiamate membri. I membri tra loro sono delimitati da virgole mentre, per separare la chiave dal corrispondente valore, si utilizza il carattere ":". JSON supporta molte delle tipologie dati native di JavaScript, in particolare: numeri, stringhe, booleani, array, oggetti e null[27]. Un esempio generico di oggetto JSON viene qui riportato:

$$\{"key1" : value1, "key2" : value2, ..., "keyN" : valueN\}$$

mentre lo schema di rappresentazione degli oggetti in JSON è illustrato in figura 2.17.

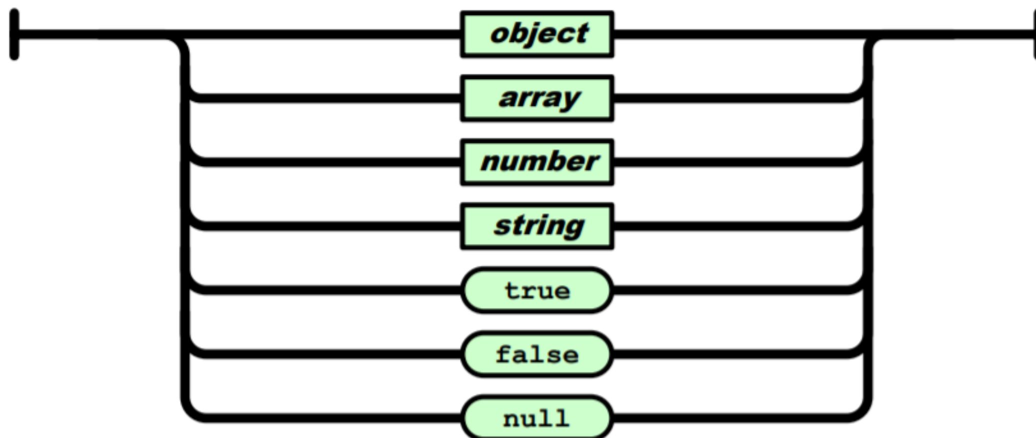


Figura 2.16: Rappresentazione delle tipologie di dati supportati da JSON.

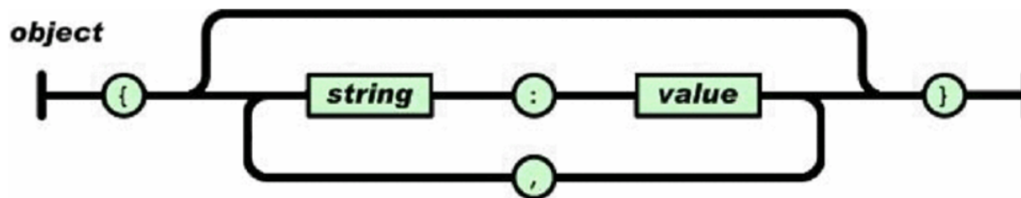


Figura 2.17: Rappresentazione dello schema di un oggetto JSON.

2.5.3 CBOR

Concise Binary Object Representation (CBOR) è un formato di serializzazione dati binario, pubblicato da Internet Engineering Task Force (IETF) in RFC 8949[28]. Come JSON, consente la trasmissione di oggetti che contengono coppie chiave-valore, ma in modo più conciso. Ciò aumenta la velocità di elaborazione e trasferimento a discapito della leggibilità umana. Anche CBOR è stato implementato nella maggior parte dei linguaggi di programmazione sia con implementazioni private che open-source. Il formato CBOR è stato sviluppato per conseguire i seguenti obbiettivi:

1. La rappresentazione deve essere in grado di codificare in modo univoco i formati dati più comuni utilizzati negli standard Internet.

2. Il codice di un codificatore o decodificatore deve essere compatto per supportare sistemi con memoria, potenza del processore e set di istruzioni molto limitati.

3. I dati devono poter essere decodificati senza una descrizione dello schema ovvero, come in JSON, il formato codificato deve essere autodescrittivo.

4. I dati serializzati devono essere compatti ma questa compattezza è secondaria rispetto a quella del codice di codifica e decodifica. Come limite inferiore di compattezza si considera quella di JSON.

5. Il formato deve essere implementabile in nodi con risorse limitate ma anche in applicazioni che richiedono la codifica di grandi volumi di dati.

6. Il formato deve supportare tutti i formati di dati JSON così che sia possibile la conversione da e verso JSON.

7. Il formato deve essere estensibile e consentire la decodifica anche a decodificatori precedenti. Un decodificatore che non comprende un'estensione deve poter comunque decodificare il messaggio.

CBOR ha un proprio modello di dati generico che definisce l'insieme di tutti gli elementi che si possono rappresentare con questo formato. Questo modello, come detto sopra, è estensibile mantenendo la stessa formattazione ed aggiungendo valori e tag. Viene quindi a crearsi la possibilità di creare sottoinsiemi di questo modello generale così da soddisfare le esigenze di specifiche applicazioni. Per poter soddisfare questi requisiti, in fase di codifica viene aggiunto un byte iniziale che contiene la descrizione dell'elemento codificato. Nello specifico, i primi 3 bit(valori da 0 a 7) identificano la tipologia principale del dato mentre, nei successivi 5 bit, vengono inserite informazioni aggiuntive.

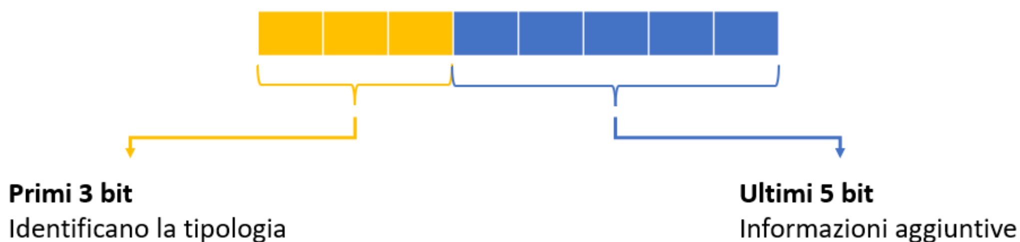


Figura 2.18: Header della codifica CBOR.

Le informazioni aggiuntive, avendo a disposizione 5 bit, possono rappresentare 32 "casi"(0-31), nella maggior parte dei casi però, essendo utilizzate per esprimere la lunghezza dell'elemento in byte, potrebbe non bastare questo range, vengono quindi aggiunti eventualmente byte che vanno a completare le informazioni aggiuntive. Per

comprendere quando questi byte di header sono aggiunti è possibile interpretare i 5 bit finali del primo byte come segue:

- **Valore inferiore a 24**: l'argomento è contenuto nei 5 bit di informazioni aggiuntive.
- **24,25,26 o 27**: l'argomento è contenuto nei successivi 1,2,4 o 8 byte rispettivamente.
- **28,29 o 30**: valori riservati per sviluppi di CBOR futuri.
- **31**: valore indeterminato.

Le informazioni aggiuntive variano in alcuni casi la propria funzione a seconda della tipologia principale che le precede. Le tipologie principali sono 8 e i bit di informazioni aggiuntive dipendono strettamente dalla tipologia di dato da codificare in cui ci troviamo:

- **Tipo 0**: interi senza segno;
- **Tipo 1**: interi negativi;
- **Tipo 2**: stringa di byte;
- **Tipo 3**: stringa di testo codificata in UTF-8;
- **Tipo 4**: array di elementi;
- **Tipo 5**: mappa di coppie chiave-valore(oggetto in JSON);
- **Tipo 6**: tag;
- **Tipo 7**: Numeri in virgola mobile e tipologie di dati semplici speciali;

Nel tipo 0, per valori interi($0..2^{(64)} - 1$) al di sotto di 24, le informazioni aggiuntive contengono direttamente il valore dell'intero da codificare mentre, per numeri maggiori, informano sul numero di byte successivi che contengono l'intero(stessa comportamento si ha per il tipo 1). Diversamente accade per il tipo 2 dove nel contenuto delle informazioni aggiuntive(i primi 5 bit eventualmente seguiti da altri byte di descrizione) viene descritta solamente la lunghezza in byte della stringa da

codificare. Nel tipo 4, le informazioni aggiuntive rendono noto il numero di elementi costituenti l'array, i quali tra loro non devono essere necessariamente dello stesso tipo. Il tipo 5 permette di creare, come in JSON, coppie chiave-valore e nell'argomento dell'header è contenuto il loro numero. In questo caso, trattandosi di coppie, il numero di elementi successivi all'header deve essere pari e la loro lettura è eseguita in ordine, riscontrando rispettivamente prima la chiave e poi il valore. Il numero di tag(tipo 6), fornisce ulteriori informazioni al decodificatore sul tipo dell'elemento seguente, oltre a quelle che può fornire il tipo principale a 3 bit. Ad esempio, un tag di 1 indica che il numero seguente è un valore temporale Unix. Un tag di 2 indica che la seguente stringa di byte codifica un bignum senza segno. Tutti i tag e le tipologie di dati semplici speciali sono definiti in un registro IANA(Internet Assigned Numbers Authority). Attraverso la registrazione di nuovi dati semplici speciali è stato possibile rendere questo formato estensibile come dichiarato negli obbiettivi. L'esempio seguente mostra un oggetto composto da coppie chiave-valore di tipologia mista (caretteri e numeri):

```
{
'key1': 1,
'key2': 2,
'key3': 3
}
```

La sua codifica CBOR è la seguente:

```
{
A3                # map(3)
   44             # bytes(4)
      6B657931 # "key1"
   01             # unsigned(1)
   44             # bytes(4)
      6B657932 # "key2"
   02             # unsigned(2)
   44             # bytes(4)
      6B657933 # "key3"
```

```
    03              # unsigned(3)
}
```

L'oggetto che è stato codificato, essendo formato da coppie chiave-valore, viene espresso in CBOR come tipologia principale 4(mappa di coppie) e il primo byte(header della mappa) è espresso in esadecimale con 'A3' che in binario corrisponde a '10100011'. Come è visibile, i primi tre bit ci informano sulla tipologia mentre, i secondi 5, rendono noto il numero di coppie presenti(3). Internamente alla mappa poi ogni valore/coppia è descritta singolarmente dal proprio header.

2.5.4 Integrità dei dati

Per IoT si intende un sistema complesso, strutturato in diversi livelli di astrazione, nel quale un pacchetto che viene inviato da un nodo sensore, viaggerà attraverso una moltitudine di sistemi prima di raggiungere l'applicazione. Ad esempio, nel suo percorso, il pacchetto attraverserà nodi con risorse limitate, router, Internet e database che potrebbero essere in parte sistemi distribuiti nel cloud. Più un pacchetto ha un percorso esteso prima di arrivare al livello applicazione di destinazione, più le probabilità di un attacco dannoso alla sua integrità aumentano. Un'applicazione che prende decisioni basate sulle informazioni che gli vengono fornite dai nodi sensori, come il caso di reti per EEW, necessita di lavorare su informazioni valide e non manomesse. Devono quindi essere seguite delle pratiche di sicurezza come l'integrità crittografica dei dati e l'autenticazione del mittente. Avendo numerosi intermediari nel percorso della comunicazione, è necessario utilizzare una tipologia di protezione end-to-end. Le firme digitali sono lo strumento crittografico per ottenere una tale protezione dell'integrità di un messaggio. La firma viene generata attraverso un algoritmo che ha come ingressi il messaggio stesso e una chiave segreta (privata). Il destinatario a sua volta applica al pacchetto arrivato, composto da messaggio e firma, un algoritmo di verifica della firma utilizzando la chiave pubblica del mittente. Se questa operazione ha esito positivo allora siamo sicuri che il messaggio non sia stato alterato e che l'origine sia autenticata. JSON Web Signature (JWS) rappresenta il contenuto del messaggio con firma digitale utilizzando strutture di dati basate su JSON[29]. I meccanismi crittografici JWS forniscono protezione dell'integrità per una sequenza arbitraria di ottetti di bit. Il pacchetto firmato, realizzato con JWS, è

composto di tre parti separate da un punto: un header che contiene le informazioni riguardanti l'algoritmo crittografico utilizzato per la firma, il payload e la firma stessa. L'header è conosciuto come JOSE Header(JSON Object Signing and Encryption Header), il suo nome deriva dal gruppo IETF che ha lavorato alla standardizzazione della rappresentazione dei dati protetti dall'integrità, mediante il formato JSON. Tutte e tre queste componenti devono essere codificate tramite BASE64URL, che

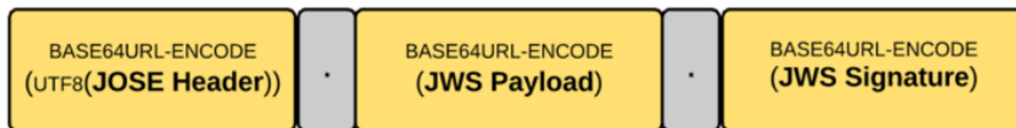


Figura 2.19: Pacchetto firmato con JWS.

traduce ottetti binari in stringhe in formato ASCII. Questa codifica, necessaria per il processo di firma del pacchetto, ha come svantaggio la perdita della leggibilità umana del payload. Il formato di messaggio JSS(JSON Sensor Signatures) è stato sviluppato proprio per evitare questo, consente infatti di avere garanzia di integrità end-to-end per applicazioni di IoT[30] ed inoltre mantiene in chiaro il payload in formato JSON. Gli obbiettivi di JSS sono:

- mantenere i dati dei messaggi firmati ancora accessibili;
- mantenere la semplicità di JSON;
- limitare l'aumento delle dimensioni dei messaggi dovute alla firma;
- la generazione e la verifica della firma devono poter essere effettuate in dispositivi con risorse limitate;

In un pacchetto quindi, il payload che mantiene leggibilità umana, è racchiuso tra un header protetto JOSE e la firma generata, quest'ultima è codificata in BASE64URL. I parametri dell'header protetto sono definiti per seguire la specifica JSON Web Algorithms [31]. Rimane comunque necessario codificare header e payload in BASE64URL per poter generare la firma. Invece di utilizzare la codifica BASE64URL di JSON, COSE(CBOR Object Signing and Encryption) propone di utilizzare il formato CBOR. Attraverso il formato COSE, non è più necessario impiegare la

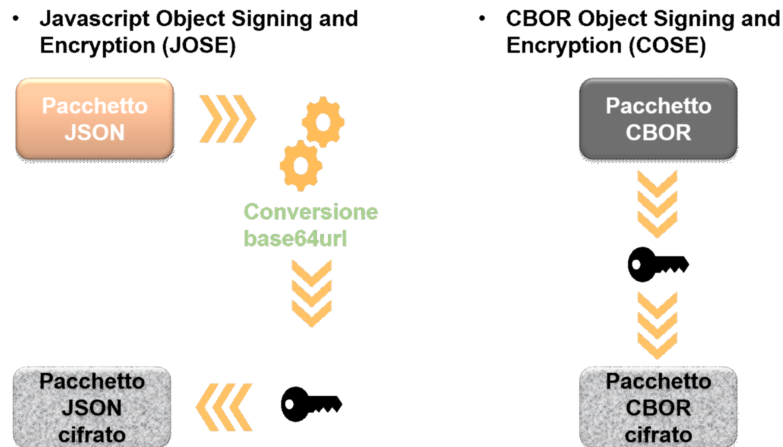


Figura 2.20: Schema di confronto tra pacchetto firmato/cifrato con JOSE e COSE.

codifica BASE64URL prima di generare la firma, bensì basta utilizzare CBOR come funzione di serializzazione.

L'obbiettivo generale di COSE, in conseguenza a quello di CBOR, è quello di mantenere un codice dell'algoritmo relativamente ristretto, cosi come le dimensioni del messaggio codificato. In [32], oltre ad altri formati, vengono comparate le dimenioni di due messaggi, uno firmato con JSS mentre l'altro con COSE. Come si può vedere dalla figura 2.21, perdendo la leggibilità umana, per l'utilizzo di CBOR, è possibile ridurre sensibilmente la dimensione del messaggio.

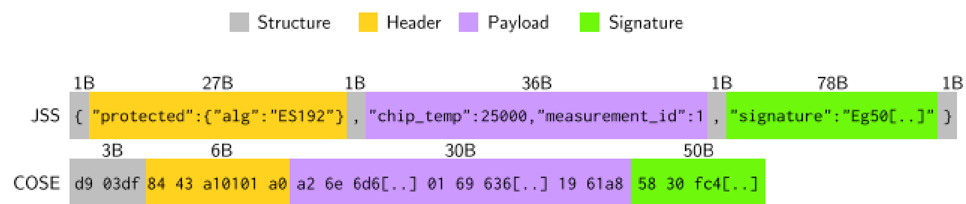


Figura 2.21: Confronto tra pacchetto firmato con JSS e con COSE.

2.6 La piattaforma Docker

Docker è una piattaforma open source che esegue applicazioni e semplifica lo sviluppo e la distribuzione di un processo sotto forma di "contenitori"(container) leggeri, portabili e autosufficienti, che possono essere eseguiti su cloud (pubblici o privati) o in locale. Le applicazioni costruite nel docker sono impacchettate con tutte le dipendenze di supporto di cui necessitano in un modulo standard chiamato container. Questi container continuano a funzionare in modo isolato sul kernel del sistema operativo ospitante. La tecnica dei container esiste da oltre 15 anni ma con docker vengono inserite delle nuove capacità tecnologiche non presenti in precedenza. Prima dell'avvento di Docker, il compito di semplificare la gestione del ciclo di vita del software è stato affidato alle tecnologie di virtualizzazione. Tuttavia, la pratica ha dimostrato come la virtualizzazione abbia in sé una serie di limitazioni che mal si coniugano con le esigenze delle applicazioni moderne. Se da una parte esse offrono un elevato grado di isolamento (ogni virtual machine è di fatto un'unità a sé), dall'altra comportano un significativo spreco di risorse di calcolo, dato che ogni VM(virtual machine) impegna l'hardware ospitante ad eseguire il codice del sistema operativo virtualizzato oltre quello delle applicazioni che esegue. Ciò limita fortemente il numero di VM che uno stesso host può ospitare senza degradare le prestazioni generali del sistema. Con docker è possibile creare e controllare container che risultano molto leggeri. All'interno di essi è si possono inserire applicazioni di qualsiasi genere da parte dello sviluppatore. Una volta creato un container è possibile utilizzarlo su una qualsiasi macchina senza che l'applicazione abbia alterazioni, questa caratteristica molto importante permette la loro condivisione. La condivisione di un container può risultare fondamentale in ambito di ricerca dove, grazie alla non alterazione dell'applicazione, diventa possibile garantire la riprodibilità di un risultato, che è un aspetto importante[33]. Un approccio basato su Docker funziona in modo simile all'immagine di una macchina virtuale. Una differenza fondamentale tra le immagini Docker e altre macchine virtuali è che le immagini Docker condividono il kernel Linux con la macchina host. Per l'utente finale, la conseguenza principale di ciò è che qualsiasi immagine Docker deve essere basata su un sistema Linux con software compatibile con Linux. La condivisione del kernel Linux rende Docker più leggero e più performante rispetto alle macchine virtuali complete, un computer desktop

potrebbe eseguire non più di poche macchine virtuali contemporaneamente ma centinaia di contenitori Docker. Questa caratteristica ha reso Docker particolarmente attraente per l'industria e molto popolare. I container di Docker sono l'insieme dei dati di cui necessita un'applicazione per essere eseguita: librerie, altri eseguibili, rami del file system, file di configurazione, script, ecc. Il processo di distribuzione di un'applicazione si riduce quindi alla semplice creazione di una Immagine Docker, ovvero di un file contenente tutti i dati appena citati. L'immagine viene utilizzata da Docker per creare un Container, un'istanza dell'immagine che eseguirà l'applicazione in essa contenuta. I container sono quindi autosufficienti perché contengono già tutte le dipendenze dell'applicazione e non richiedono quindi particolari configurazioni sull'host, ma sono soprattuto portabili, perché essi vengono distribuiti in un formato standard (le immagini appunto), che può essere letto ed eseguito da qualunque server Docker. Utilizzando docker si elimina l'Hypervisor, lo strato software in esecuzione nella macchina ospitante, che si occupa di gestire le risorse allocate a ciascuna VM e che adotta (anche con l'ausilio dell'hardware) tutte le politiche necessarie per isolare i processi in esecuzione su VM differenti. Le funzionalità dell'Hypervisor sono assolte dal kernel del sistema operativo ospitante. Linux per questo scopo utilizza Control Groups (o cgroups) e Namespaces rispettivamente per gestire l'utilizzo delle risorse di calcolo da parte di un gruppo specifico di processi e garantire l'isolamento dei processi in esecuzione in container differenti. In figura 2.22 vengono comparate la struttura docker a destra e la struttura delle virtual machine a sinistra. Una peculiarità delle

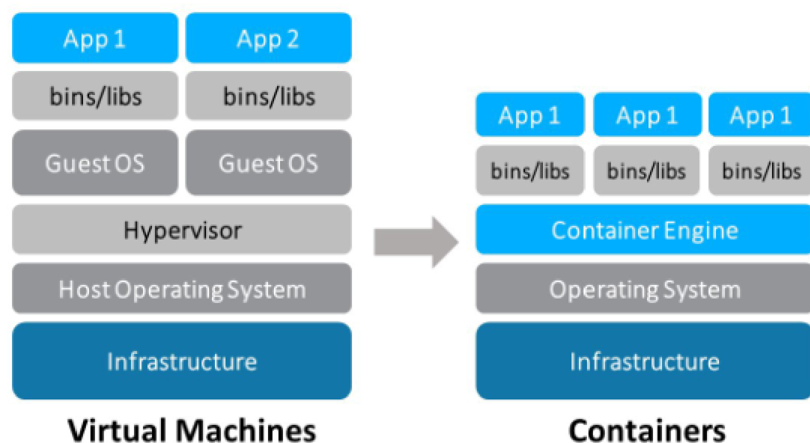


Figura 2.22: Confronto tra struttura Docker e quella delle Virtual Machine.

immagini Docker è la loro stratificazione in layer, ognuno dei quali contribuisce alla definizione di quello che sarà poi il file system del container. Particolare di questi layer è la loro immutabilità: i layer sono infatti accessibili in sola lettura e non sono modificabili direttamente. Questa, che potrebbe sembrare una limitazione, è invece un grande punto di forza di Docker, che in questo modo offre la possibilità a più immagini di condividere un layer comune. Un'immagine rappresenta, di fatto, una serie di layer immutabili chiamati layer immagine. Una pila di layer accessibili in sola lettura, però, non ha molto senso di esistere, ragion per cui quello che avviene nel momento in cui si chiede a Docker di istanziare un container è la creazione, in cima a tutti gli altri layer, di un singolo layer scrivibile. Questo layer è proprio il container in cui verranno memorizzate tutte le modifiche apportate. Le modifiche effettuate in un container, nella fase successiva alla sua creazione, potrebbero rendere difficile la sua ripetibilità. Viene per questo scopo utilizzzato un Dockerfile, un semplice file di testo che, con una sintassi semplice e concisa, ci permette di esprimere le personalizzazioni che vogliamo apportare nella creazione del nostro container. Inoltre, la condivisione

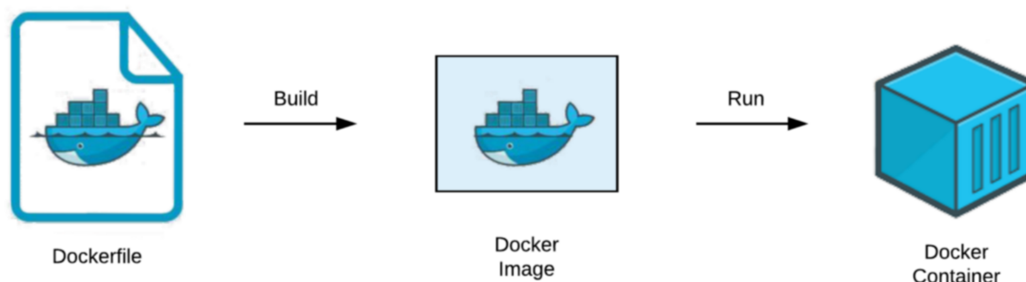


Figura 2.23: Creazione di un Container da un Dockerfile.

di immagini, è semplificata dal Docker Hub, servizio gratuito per la distribuzione e archiviazione di immagini, per il download e il riutilizzo da parte di altri. I principali vantaggi di docker quindi sono:

- Velocità: vista la loro ridotta dimensione i container sono relativamente veloci da costruire,testare,modificare e condividere.;

- Portabilità: l'applicazione può facilmente essere spostata da un sistema all'altro senza che il suo comportamento sia alterato.;

- Scalabiltà: i container possono facilmente essere spostati da piccoli host locali a grandi piattaforme cloud.;
- Formato standardizzato: i docker hanno una struttura standardizzata in modo da semplificare il lavoro dello sviluppatore.;
- Densità: l'eliminazione del layer di hypervisor consente ad una macchina locale di utilizzare contemporaneamente molti container.;

Queste caratteristiche permettono di avere maggiori performance rispetto ad una virtual machine[34].

Capitolo 3

Materiali e metodi

3.1 Introduzione

In un EEWS maggiore è il tempo a disposizione prima che un evento sismico colpisca un punto target, maggiori e più efficaci potranno essere le azioni che possono essere intraprese. Studiare quindi i ritardi che ogni componente di un sistema aggiunge riveste notevole importanza. Partendo dal lavoro presentato in [35] si evince come i contributi più importanti in termini di tempo vengono introdotti dalla data latency, intesa come somma di tempi di pacchettizzazione e di invio dei campioni rilevati dal dispositivo. Potendo paragonare un sistema di monitoraggio sismico ad un'applicazione tipica dell'IoT a livello architetturale, ci siamo chiesti se l'adozione del protocollo MQTT, uno dei protocolli più utilizzati in questo ambito, potesse esser utile ai fini di un minore impatto sul tempo di rilevazione di un'onda sismica e, più in generale, sull'invio di un segnale di allerta. Diversi studi in letteratura hanno dimostrato i migliori risultati di MQTT in termini di latenza end-to-end e consumo di larghezza di banda riferiti ad altri protocolli utilizzati nelle applicazioni IoT come HTTP , AMQP o COAP. Grazie alla struttura semplice, MQTT può essere facilmente implementato in dispositivi con processori a bassa potenza e basse prestazioni e si sta rapidamente diffondendo. Per questi motivi si è deciso di implementarlo e valutarne il comportamento in ambito EEW, comparandolo al protocollo utilizzato attualmente per lo scambio di rilevazioni sismiche(SeedLink). L'obbiettivo della prima analisi è stato quello di verificare le latenze dei due protocolli in esame, intesa come la differenza temporale tra l'istante di ricezione e l'istante di invio di un medesimo pacchetto miniSEED, inviato tramite SeedLink o come payload

di un messaggio MQTT. L'idea non è quella di sostituire il protocollo standard SeedLink, ma di studiare la possibilità di implementazione del protocollo MQTT parallelamente agli standard attuali utilizzati nei datalogger(dispositivi di raccolta dati). In questa parte del lavoro ci si è focalizzati nell'individuare quale protocollo di comunicazione garantisse minori latenze nella trasmissione del pacchetto, ma è necessario tenere in considerazione che anche altri aspetti possono contribuire alla latenza. Tra questi, le strutture di impacchettamento dei dati con i loro rispettivi tempi e dimensioni risultanti del pacchetto. Nella seconda parte sono state quindi comparate le performance di diversi formati di dati per l'invio di pacchetti attraverso il protocollo MQTT.

3.2 Confronto tra MQTT e SeedLink

In questo paragrafo viene illustrata l'implementazione della prima architettura di misura che ha lo scopo di verificare se, il protocollo MQTT, sia in grado di ridurre i tempi di pacchettizzazione e trasmissione del sistema di comunicazione rispetto all'utilizzo del protocollo SeedLink. Al fine di confrontare le latenze dei due protocolli, sono stati trasmessi identici pacchetti miniSEED tramite SeedLink e come payolad di un messaggio MQTT. Utilizzando il protocollo MQTT però, non è possibile inserire direttamente un pacchetto in formato miniSEED all'interno del payload di un messaggio MQTT, come verrà approfondito nel prossimo capitolo. La soluzione proposta per questa prima analisi è quella di effettuare una pacchettizzazione utilizzando una struttura dati, differente da miniSEED ma con lo stesso contenuto informativo, in formato JSON per inviare i pacchetti mediante il protocollo MQTT. Nel caso in esame, per ridurre l'arbitrarietà del client publisher (il dispositivo di rilevamento sismico), è stato scelto di utilizzare i topic secondo il SSNC descritto in precendenza,pertanto con la forma:

$$Network/Station/Location/Channel$$

Per quanto riguarda invece la struttura del pacchetto JSON, è stata organizzata in questo modo:

```
{
"timestamp" : <Integer>,
"values" : [Array],
"encoding" : <String>,
"sps" : <Integer>
}
```

dove nel campo *timestamp* viene inserito il valore in millisecondi del timestamp del primo valore contenuto nell'array *values*. Questo array può essere di lunghezza arbitraria, può cioè contenere un numero configurabile di campioni. I dati possono essere codificati secondo diversi formati (ad esempio INT16,INT32, FLOAT32, FLOAT64, STEIM1, STEIM2) in conformità al pacchetto miniSEED e questa informazione è contenuta nel campo *enconding*. Il campo *sps* contiene il numero di campioni per secondo, da cui è possibile quindi ottenere la frequenza di campionamento. Tramite l'utilizzo di questo schema è possibile scegliere il numero di campioni da inserire all'interno dell'array. Questa struttura dati in formato JSON ha come obbiettivo quello di trasportare le stesse informazioni contenute all'interno di un pacchetto miniSEED(o abbreviato con mSEED). Infatti, possone essere aggiunti ulteriori campi informativi se necessario, tenendo in considerazione che la descrizione del dispositivo sismico di misura(Publisher), all'interno della rete sismica, è già disponibile nel topic MQTT. Come mostrato in Fig.3.1, partendo da tracce sismiche, sono state create finestre temporali mobili di un secondo, al fine di simulare quindi il comportamento real-time di un datalogger collegato al sistema PRESTo. Ogni secondo il pacchetto viene inviato ad un SeedLink server ed a un broker MQTT che risiedono in una stessa macchina. Il dispositivo simulato e l'architettura descritta sono quindi perfettamente conformi ad un sistema composto da datalogger e acquisitore di una seismic network generica, aggiungendone la funzionalità di server MQTT. Sono stati generati i due client SeedLink ed MQTT, mendiante l'utilizzo della libreria Python Obspy [36] e della libreria Paho, all'interno di un'altra macchina e tramite un analizzatore di rete sono stati registrati i tempi di ricezione del pacchetto. La differenza tra l'istante di ricezione e l'istante di invio di un medesimo pacchetto permette di confrontare le latenze dei due protocolli, in quanto le condizioni di analisi sono le medesime (larghezza di banda, traffico di rete, tempo dedicato alla pacchettizzazione, etc.).

Particolare attenzione è stata dedicata alla sincronizzazione della macchina che simula il dispositivo sismico ed della macchina dove risiedono i client SeedLink ed MQTT mediante l'utilizzo del protocollo NTP [37]. I client sono stati simulati in una macchina

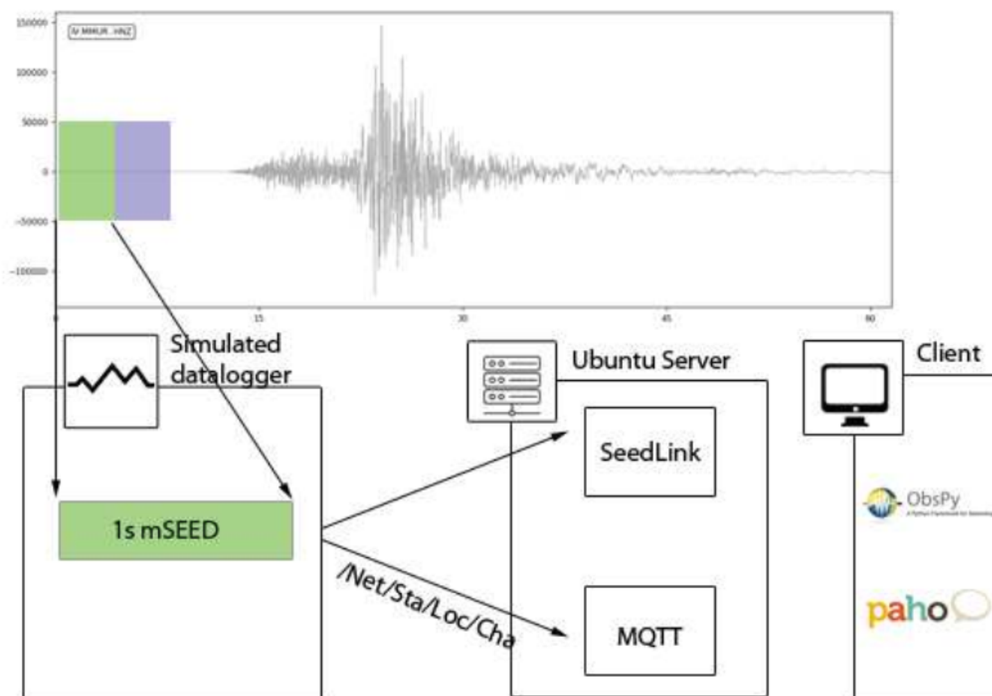


Figura 3.1: Schema della metodologia per la valutazione dei tempi di trasmissione delle forme d'onda mediante il protocollo SeedLink ed il protocollo MQTT.

con le seguenti caratteristiche: Intel Xeon X5650 (x2) CPU, 12 MB cache, 2.66 GHz, 16 GB RAM con sistema operativo Linux Ubuntu 18.04.1 LTS. Come si nota dalle caratteristiche, si tratta di una macchina molto performante. Al fine di verificare la fattibilità pratica della soluzione in casi reali, dove gli acquisitori hanno solitamente limitate capacità di elaborazione, abbiamo simulato un dispositivo tramite l'utilizzo di una Raspberry Pi 3 Model B con CPU Quad Core 1.2 GHz Broadcom BCM2837 64 bit ed 1 GB RAM, installando al suo interno lo script per la lettura di una sola traccia accelerometrica e relative chiamate ai due processi descritti in precedenza, un SeedLink server ed un broker MQTT. Per quanto riguarda il server SeedLink abbiamo installato e configurato il ringserver distribuito da Incorporated Research Institutions for Seismology (IRIS) [38]. Per quanto riguarda il broker, abbiamo utilizzato il broker MQTT Mosquitto visto che si tratta di un implementazione leggera ed open source che si adatta perfettamente alla nostra installazione su un dispositivo

a ridotte capacità computazionali. I risultati ottenuti sul Raspberry evidenziano un utilizzo della CPU mai andato oltre il 40% durante le simulazioni, confermando pertanto la possibilità di implementazione anche su dispositivi meno performanti. Per quanto riguarda il client SeedLink è stato utilizzata la classe EasySeedLinkClient in obspy.clients.seedlink.easyseedlink, mentre per il client MQTT abbiamo utilizzato la libreria Paho-MQTT. Il protocollo MQTT supporta, come illustrato in precedenza, 3 livelli di QoS. Le simulazioni sono state effettuate con una QoS pari ad 1 (at least once), al fine di riprodurre il comportamento del protocollo SeedLink. Tutte le misurazioni sono state condotte al termine della fase di connessione e sottoscrizione del protocollo MQTT e successivamente alla fase di apertura della connessione e di handshake del protocollo SeedLink.

3.2.1 Metriche di comparazione

Al fine di studiare le differenze temporali nell'utilizzo delle due diverse modalità, andiamo a definire le metriche da comparare. Facendo riferimento alla Fig.3.2, per quanto riguarda il datalogger, definiamo come

$$T_{pack,i} = t_{ep,i} - t_{sp,i} \tag{3.1}$$

dove $T_{pack,i}$ è il tempo di pacchettizzazione dell'i-esimo pacchetto, ottenuto da $t_{sp,i}$ e $t_{ep,i}$, cioè l'istante del primo ed ultimo campione nel pacchetto stesso. $T_{pack,i}$ dipende dal sampling rate e dal numero di campioni contenuti in ogni pacchetto. Dal punto di vista del client definiamo $t_{r,i}$ come il tempo di arrivo del pacchetto i al client. Pertanto il data latency $T_{l,i}$ del pacchetto i-esimo è definito come la differenza temporale tra l'istante dell'ultimo campione nel pacchetto ed l'istante di ricezione dello stesso:

$$T_{l,i} = t_{r,i} - t_{ep,i} \tag{3.2}$$

Questa quantità contiene anche il tempo necessario al datalogger per l'invocazione della funzione di pacchettizzazione.

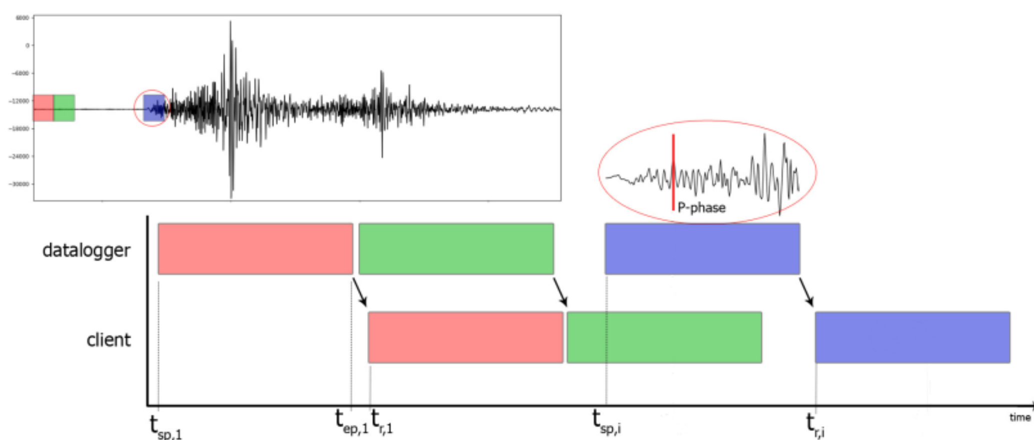


Figura 3.2: Riepilogo metriche considerate nella comparazione tra il protocollo SeedLink e MQTT.

3.3 Confronto tipologie di formato dati in MQTT

Una volta implementato il sistema di simulazione di un datalogger reale ed eseguite le misure di confronto tra i due protocolli, ci si è chiesti quale formato dati potesse ottimizzare i tempi di ritardo nella comunicazione. Utilizzando il protocollo MQTT infatti, non è possibile effettuare l'invio diretto di un pacchetto miniSEED, a meno che questo non sia serializzato in modo tale da poter essere inserito all'interno del payload del messaggio. Per sopperire a questa problematica, nelle misurazioni precedenti, le informazioni contenute all'interno del pacchetto miniSEED sono state inserite come campi di una struttura serializzata con JSON. Lo scopo di questa seconda parte del lavoro è quella di andare ad analizzare altri formati dati utilizzabili in situazioni real-time, come nel caso in esame dell'EEW. La struttura di comunicazione MQTT implementata per questa analisi, cerca il più possibile di simulare l'architettura reale di una rete sismica ed è rappresentata in Fig.3.3. Infatti, da un lato troviamo un nodo Publisher, che invia pacchetti contenenti campioni rilevati da sensori accelerometrici per esempio, mentre dall'altro abbiamo un nodo(Subscriber) che riceve i pacchetti che gli vengono inoltrati dal broker e tramite la libreria Obspy salva questi in locale in formato miniSEED. Per poter caratterizzare il comportamento di un certo protocollo di serializzazione dati mediante l'utilizzo di MQTT sono state suddivise le fasi della comunicazione in intervalli temporali come nello schema in Fig.3.4. Vengono quindi tralasciati gli istanti iniziali di connessione del client Publisher al Broker e

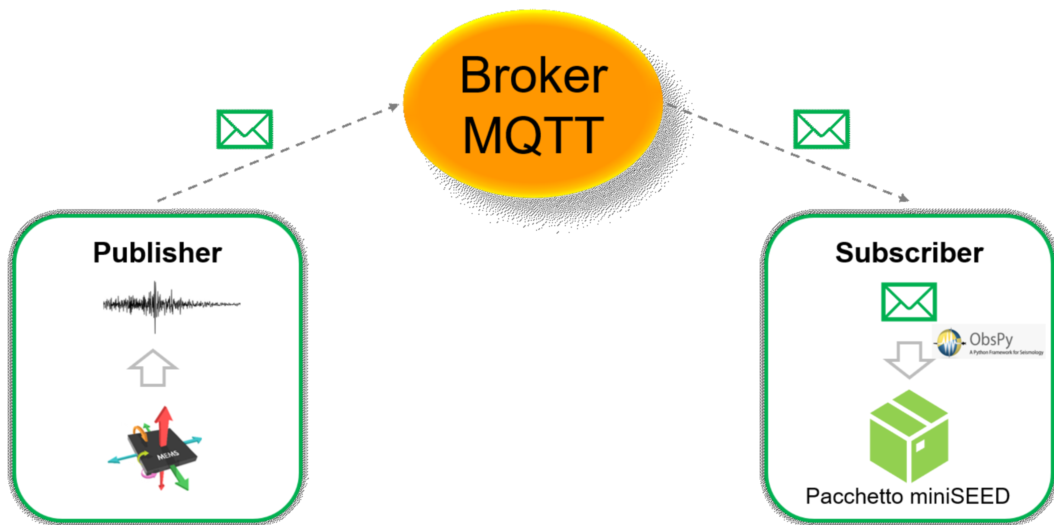


Figura 3.3: Rappresentazione della struttura di comunicazione MQTT implementata.

gli stessi da parte del client Subscriber che deve anche sottoscriversi al topic. Il

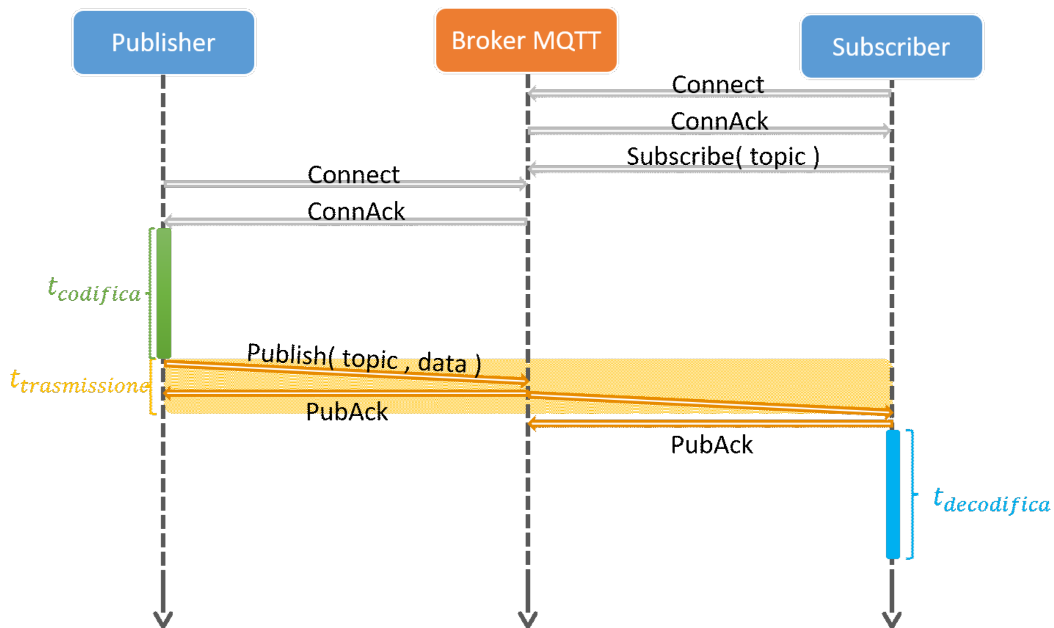


Figura 3.4: Intervalli temporali presi in esame all'interno di una comunicazione MQTT con una $QoS = 1$.

nodo Publisher necessita inizialmente di un tempo per raccogliere le misurazioni costante, dipendente dal numero di campioni richiesti all'interno di un pacchetto e ovviamente dalla fraquenza di campionamento scelta. Successivamente questi valori devono essere inseriti nel payload di un messaggio attraverso un operazione di pacchettizzazione(o codifica). Il tempo necessario alla raccolta dati non risulta

essere di nostro interesse dato che deriva da una scelta progettuale del sistema. Viene quindi chiamato $t_{codifica}$ il tempo necessario alla codifica dei dati, che deve essere ottimizzato per ridurre la latenza tra l'ultimo campione misurato e l'invio del pacchetto. Una volta inviato il pacchetto, questo arriva al broker dove viene immediatamente inoltrato a tutti i Subscriber del topic(nel nostro caso è soltanto uno). L'intervallo temporale, che inizia con la pubblicazione del messaggio e termina con la ricezione del messaggio da parte del Subscriber, è stato denominato $t_{trasmissione}$. Ad ogni ricezione di un messaggio, come definito dal protocollo MQTT per una QoS di 1, il Subscriber ed il broker notificano il mittente con un $PUBACK$. Lato Subscriber il pacchetto subisce una procedura di deserializzazione e successiva creazione di un nuovo pacchetto in formato miniSEED, conforme allo standard attualmente in uso nelle reti sismiche, tramite l'utilizzo della libreria Python Obspy. Questo tempo, che parte dalla ricezione del messaggio fino al raggiungimento del formato miniSEED, è stato denominato $t_{decodifica}$. La tipologia di formato dati scelta influisce su tutti questi intervalli temporali. Ogni protocollo infatti necessita di tempi di codifica/decodifica dei dati differente e il rispettivo dato serializzato potrà riscontrare una diversa dimensione in termini di byte. Questa differente dimensione del dato serializzato condiziona la latenza derivata dal $t_{trasmissione}$ in quanto, un pacchetto di dimensione più grande richiede tempi di trasmissione mediamente maggiori in base alla condizione della rete in cui ci si trova ed oltretutto è soggetto ad una probabilità packet loss superiore. Inoltre, la scelta del formato dati migliore, non è universale per un qualsiasi pacchetto processato dall'algoritmo di codifica, in quanto risultati diversi possono essere riscontrati in funzione della tipologia di informazioni contenute e questo implica che le scelte siano orientate rispetto al caso in esame. Per questo motivo Sono per questo stati implementati 4 modelli di formato dati differenti, in particolare, due di essi sono basati sulla serializzazione di pacchetti miniSEED, mentre altri due si basano su un concetto differente, come nella struttura dati utilizzata nell'analisi precedente. Questi ultimi si basano sull'idea di spostare la generazione del pacchetto miniSEED lato Subscriber, che si suppone avere capacità computazionali superiori(es. piattaforma Cloud) a chi effettua e trasmette le misurazioni.

3.3.1 miniSEED con serializzazione JSON

Il primo formato dati implementato per essere utilizzato con il protocollo MQTT si basa su una serializzazione di tipo JSON e permette ad un pacchetto miniSEED di essere inserito all'interno del payload del messaggio. Utilizzando il protocollo Seedlink il pacchetto miniSEED veniva scritto all'interno di un buffer del ringserver per poi essere inoltrato al client, questo processo permette di trasferire il pacchetto tra due o più client della rete. Con MQTT non è possibile inserire tale struttura dati in un buffer del broker ma è necessario applicare una 'conversione', da parte del nodo Publisher affinchè l'informazione possa essere contenuta all'interno del payload del messaggio. Questa 'conversione' non è frutto di una procedura diretta ma, la struttura dei dati in formato miniSEED, necessita di opportune eleaborazioni prima di poter essere serializzata con JSON. Questo formato dati è stato chiamato miniSEED serializzato JSON(mSJ) e il processo che porta alla sua realizzazione è illustrato in Fig.3.5. Seguendo il flusso di operazioni di questo processo, vengono per prima cosa raccolte le misure all'interno di una lista, a questo punto, tramite le funzioni della libreria Obspy, si crea un oggetto python che rappresenta il pacchetto miniSEED vero e proprio. L'oggetto in questione è già serializzato di per se in una rappresentazione python che non può pero essere esportata in altri ambienti direttamente. Un metodo per la sua conversione è fornito da python attraverso un suo modulo di serializzazione chiamato Pickle. Il modulo pickle implementa protocolli binari per la serializzazione e la deserializzazione al fine di rappresentare una struttura di oggetti Python. "Pickling" è il processo mediante il quale una gerarchia di oggetti Python viene convertita in un flusso di byte e "unpickling" è l'operazione inversa, per cui un flusso di byte (da un file binario o da un oggetto simile a byte) viene riconvertito in una gerarchia di oggetti. Il formato dei dati utilizzato da Pickle è Python-specific ovvero, il risultato del processo di pickling può essere letto e interpretato solo da un programma python. Questo ha il vantaggio di non avere restrizioni imposte da standard esterni come JSON, tuttavia comporta che i programmi non Python potrebbero non essere in grado di ricostruire oggetti Python serializzati con Pickle [39]. Deserializzando con questo modulo l'oggetto miniSEED, attraverso una semplice conversione della codifica dei caratteri in ASCII, è possibile inserire l'informazione ottenuta nella struttura serializzata con JSON. Le operazioni

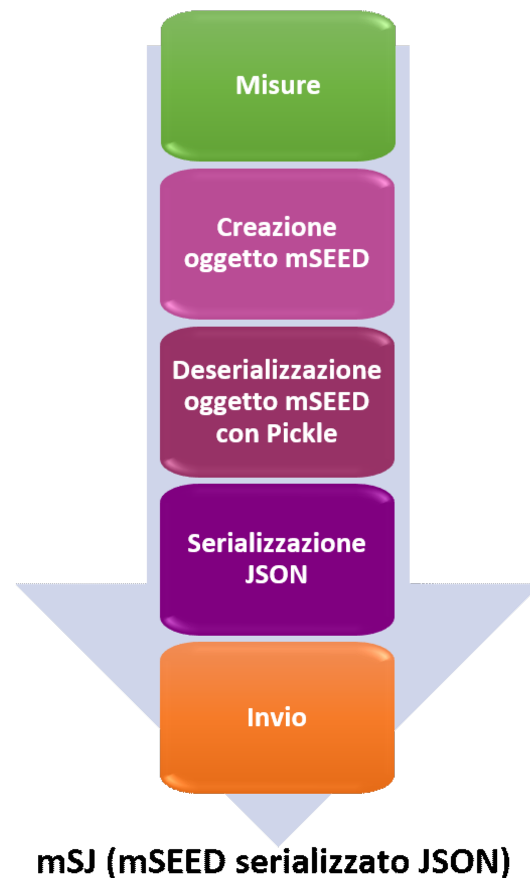


Figura 3.5: Processo di realizzazione del formato miniSEED con serializzazione JSON.

appena descritte generano un pacchetto sul quale non si dispone di leggibilità umana in quanto si parte da un oggetto miniSEED. In fase di decodifica, lato Subscriber, è sufficiente percorrere il processo inverso al fine di ottenere il pacchetto in formato miniSEED.

3.3.2 miniSEED con serializzazione binaria

Il secondo formato, sempre derivante dal pacchetto miniSEED, è stato implementato per non dover incorrere a tutte le operazioni di conversione della struttura dati necessarie in mSJ. Restando all'interno dell'ambiente python non è infatti possibile evitare di utilizzare tali operazioni trattandosi di un oggetto. Per superare questo limite è stata sfruttata la libreria Obspy per il salvataggio in locale di un file in formato miniSEED contenente il pacchetto che deve essere inviato dal Publisher. Con Seedlink il pacchetto veniva salvato sul buffer del ringserver, in questo caso

invece direttamente in una locazione di memoria locale del Publisher(Il ringserver può essere installato sullo stesso dispositivo del Publisher).Una volta salvato, il file, non è altro che una stringa binaria e può essere letta e caricata come tale da una qualsiasi funzione di lettura per file binari. Utilizzando questo approccio, nel payload del messaggio viene inserita la stringa binaria rappresentente le informazioni del pacchetto miniSEED. Questo formato dati è stato denominato miniSEED con 'serializzazione' binaria(mB) e il suo flusso procedurale è evidenziato in fig.3.6. Attraverso

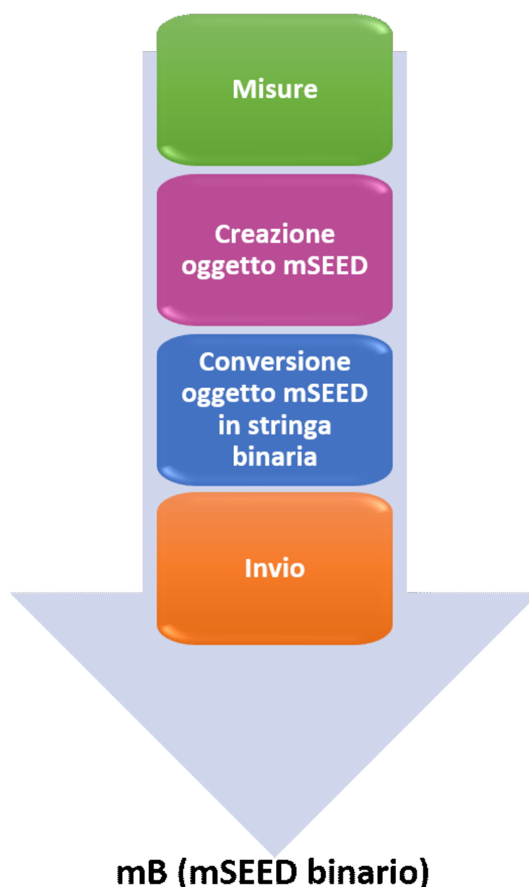


Figura 3.6: Processo di realizzazione del formato miniSEED con serializzazione binaria.

l'utilizzo di Obspy, nel momento in cui si procede al salvataggio mediante il modulo dedicato *write*, deve essere specificata la dimensione desiderata del file. Questa è stata impostata a 512 byte in quanto, nello standard miniSEED, tale dimensione è stabilita per lo streaming. Il pacchetto trasmesso, come nel formato mSJ, non dispone di leggibilità umana ed un ulteriore svantaggio è rappresentato dall'operazione di salvataggio in locale che potrebbe rappresentare un problema in dispositivi

con limitata memoria a disposizione. D'altra parte però bisogna considerare che la dimensione di un pacchetto, con un limitato numero di campioni, è solitamente molto contenuta. Un importante vantaggio di questo formato è riscontrabile lato Subscriber dove, all'arrivo del messaggio, è sufficiente salvare il contenuto del payload per avere a disposizione il pacchetto miniSEED, senza eseguire alcuna operazione di conversione.

3.3.3 Il formato μSEED

Fino ad ora si è fatto riferimento al formato standard miniSEED utilizzando procedure che ci permettano la sua pacchettizzazione in messaggi MQTT. Partendo dalla struttura del pacchetto JSON utilizzata nella prima analisi, per il confronto dei due protocolli di comunicazione, sono stati sviluppati due ulteriori formati. La caratteristica che gli accomuna è quella di spostare la generazione del pacchetto miniSEED lato Subscriber. L'utilizzo di questa tipologia di formato del pacchetto nella comunicazione tra client, consente:

1. Interoperabilità sulle strutture dati nella comunicazione;
2. Mantenimento dello standard miniSEED;
3. Rimozione della libreria python Obspy sul dispositivo Publisher;

Il payload del messaggio ricevuto conterrà tutte le informazioni attraverso le quali il Subscriber sarà in grado di generare il pacchetto in formato miniSEED. Analizzando i campi all'interno dello standard miniSEED si è riscontrato che molti di essi risultavano ridontanti nella comunicazione in quanto, per ogni pacchetto, viene descritto il dispositivo di misura e l'array di campioni. Utilizzando il protocollo MQTT c'è la possibilità di sfruttare il campo topic per inserire informazioni ridondanti, per ora era stato utilizzato solo per descrivere il dispositivo all'interno della rete sismica. Informazioni come la frequenza di campionamento, la codifica utilizzata e tante altre, rimangono costanti nel tempo ed inserendole direttamente nel topic sono comunque rese fruibili al Subscriber che per esempio, non dovrà più leggere all'interno del payload quale *Sps* ha utilizzato uno specifico sensore di una stazione che ha inviato il messaggio. Utilizzando la struttura gerarchica del topic, un Subscriber può

sottoscriversi a tutti i dispositivi di una rete mediante una semplice Wildcard('#') nelle fasi iniziali di sottoscrizione. Nel caso una stazione aggiungesse uno o più dispositivi per arricchire la sue rete di monitoraggio, il Subscriber non riscontrerebbe alcuna difficoltà avendo tutte le informazioni necessarie nel topic. Stessa situazione si verrebbe a creare in una circostanza in cui uno dei parametri, normalmente costanti, dovesse subire una variazione. Il Publisher in questo caso non farà altro che aggiornare il contenuto del topic mentre il Subscriber vedrà recapitarsi un messaggio con un nuovo topic, come se fosse un nuovo dispositivo, dal quale prenderà le informazioni necessarie alla generazione del pacchetto miniSEED. Con l'utilizzo della wildcard, grazie alla struttura gerarchica del topic, la sottoscrizione a topic di livello sottostante sarà automatica. Partendo dai principi appena illustrati, sono stati analizzati i campi all'interno dell'intestazione di un pacchetto miniSEED per determinare cosa potesse essere inserito nel topic e cosa invece necessitava di essere inviato in ogni messaggio. In Fig.3.7 è rappresentato un pacchetto miniSEED nel quale sono stati evidenziati

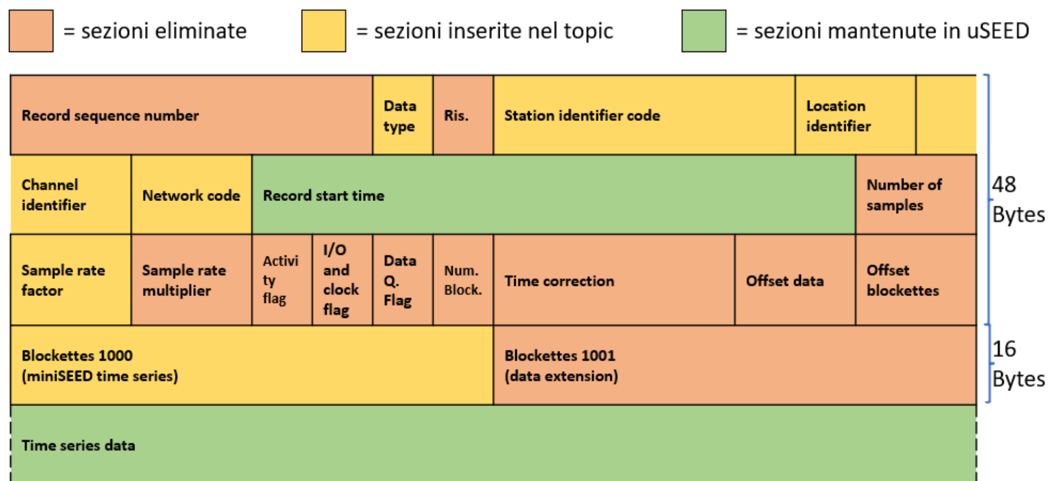


Figura 3.7: Rappresentazione grafica dei campi selezionati di un pacchetto miniSEED per la nuova struttura dati μSEED.

i vari campi con diversi colori(come in legenda). Nel nostro studio non sono state considerate tutte le sezioni, reputando alcune di esse ricavabili da altre o poco significative, nulla vieta comunque di utilizzarle all'interno della nuova struttura dati con piccole integrazioni insignificanti nei confronti dei risultati. Per esempio, la sezione che informa sul numero di campioni contenuti nel pacchetto, non facendo uso dello standard miniSEED, diviene inutile nella nuova struttura dati in cui il

Subscriber ha piena visibilità sul numero di campioni contenuti nell'array. Altre sezioni più importanti all'interno dell'intestazione sono state inserite direttamente nella struttura del topic, nell'ultimo livello utile a descrivere il dispositivo e la sua misurazione. Il topic utilizzato nelle simulazioni è il seguente:

$$Network/Station/Location/Channel_Dataquality_Sps_Encoding$$

dove sono state aggiunte informazioni, separate mediante il carattere '_', a livello *channel*. La descrizione può essere facilmente estesa di ulteriori campi, utili in una situazione reale. Le restanti sezioni contengono lo stretto necessario, ovvero i campioni e l'istante temporale di inizio misurazione, due sezioni chiave che subiscono un cambiamento per ogni pacchetto dello streaming. Questa nuova struttura dati è stata denominata μSEED per evidenziare la riduzione ulteriore di informazioni fornite rispetto a miniSEED. In python la struttura dati utilizzata è rappresentata come coppie chiave-valore in questo modo:

```
{
"timestamp" : <Integer>,
"values" : [Array]
}
```

Questa struttura necessita di una serializzazione per poter essere inserita all'interno del payload del messaggio MQTT, nella prossima sezione vengono quindi sviluppati i due ulteriori formati di dati citati in precedenza.

3.3.4 μSEED con serializzazione JSON e CBOR

Gli ultimi due formati, utilizzati in termini di confronto rispetto a mSJ e mB, derivano dalla struttura μSEED illustrata nella sezione precedente e si occupano di codificarla in modo tale da poterla inviare in un messaggio MQTT. Esistono numerosi protocolli di codifica dei dati, ognuno di questi con le proprie peculiarità e difetti. In questa analisi ne sono stati presi in considerazione due, JSON e CBOR. Il primo formato è stato scelto per la sua diffusione nello scambio di informazioni sul web, per la sua leggerezza ed inoltre per consentire leggibilità umana sui dati. Come secondo formato è stato selezionato CBOR perchè consente di avere una codifica dei dati

più coincisa a discapito della leggibilità umana. CBOR infatti è stato sviluppato partendo proprio JSON, cercando di mantenere le caratteristiche che lo hanno reso celebre e migliorarne i difetti. L'architettura di codifica del pacchetto risulterà quindi più snella in quanto, successivamente alla raccolta dei campioni in un array ed inserito il *timestamp*, sarà sufficiente applicare la specifica funzione di serializzazione per ottenere il payload desiderato pronto per essere inviato. Infatti l'header del formato μSEED è composto solamente dal *timestamp*, un valore generato dalla classe DateTime per descrivere gli istanti nel tempo, definito come il numero di nanosecondi trascorsi dalla mezzanotte del Coordinated Universal Time(UTC) di giovedì 1 gennaio 1970. Il processo di pacchettizzazione JSON è rappresentato in Fig.3.8 e viene chiamato μSEED con serializzazione JSON(μSJ).

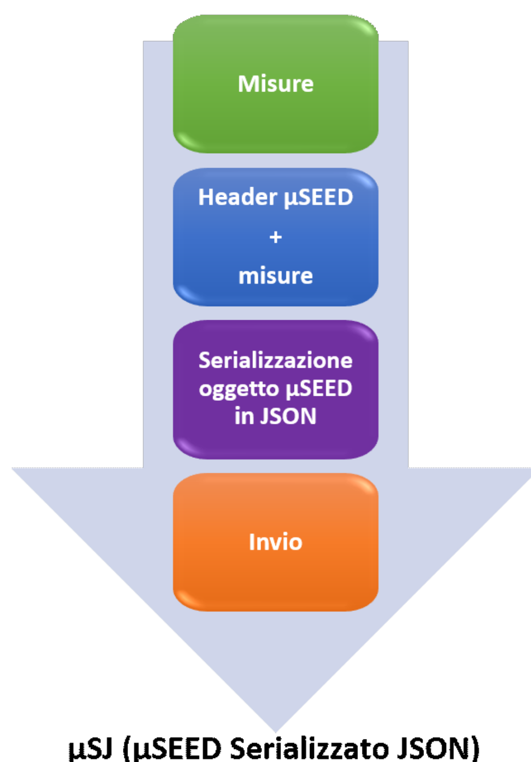


Figura 3.8: Processo di realizzazione del formato μSEED con serializzazione JSON.

Il secondo formato analizzato in questa sezione risulta pressochè identico al primo tranne che per il protocollo di serializzazione CBOR(Fig.3.9), viene quindi definito come μSEED con serializzazione CBOR(μSC). Lato Subscriber sarà necessario per prima cosa deserializzare il payload del messaggio, poi, utilizzando anche le informazioni contenute nel topic, viene generato il pacchetto in formato miniSEED

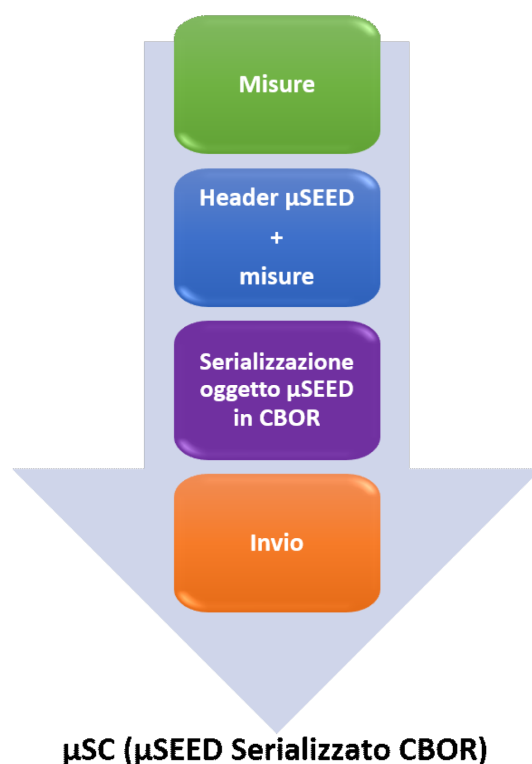


Figura 3.9: Processo di realizzazione del formato μSEED con serializzazione CBOR.

standard. Queste due tipologie di pacchetto sono state sviluppate considerando di dover ridurre il carico di lavoro al dispositivo di acquisizione, che ha limitate capacità computazionali rispetto al client che riceve i messaggi. Quest'ultimo si presume essere una piattaforma cloud o comunque un nodo di rete relativamente performante, così da non risentire dell'aumento delle operazioni da eseguire. Un altro aspetto da tenere in considerazione è quello dell'interoperabilità fornita dal non utilizzo dello standard miniSEED, la quale permette l'interazione fra sistemi differenti, nonché lo scambio e il riutilizzo delle informazioni anche fra sistemi informativi non omogenei. Nello sviluppo dei sistemi di EEW con dispositivi IoT questa caratteristica potrebbe risultare fondamentale. Un altra tematica che sta con gli anni diventando sempre più importante in rete e soprattuto con i dispositivi direttamente interfacciati ad internet, è quella legata alla sicurezza. Questa può essere incrementanta ad ogni livello della pila protocollare ISO/OSI attraverso opportuni protocolli e algoritmi. In una rete di EEW risulta essenziale avere certezza sul nodo mittente del messaggio dato che, sulle informazioni ricevute, vengono prese importanti decisioni. Per garantire una protezione dei dati end-to-end è necessario fornire sicurezza già dal livello applicazione

utilizzando schemi di cifratura o firma digitale sul pacchetto. In questo caso avendo a disposizione strutture di dati JSON e CBOR è possibile utilizzare i loro meccanismi crittografici rispettivamente JOSE e COSE. COSE rispetto a JOSE applica gli algoritmi crittografici direttamente ai dati del pacchetto sfruttando la codifica binaria di CBOR mentre JSON ha bisogno di una codifica BASE64URL prima di poter essere utilizzato. Alle caratteristiche di questi due formati dati è quindi possibile aggiungere anche questo ulteriore vantaggio.

3.3.5 Implementazione struttura di analisi

Una prima valutazione è stata dedicata alle dimensioni del pacchetto dati, nei 4 formati, per un diverso numero di campioni al loro interno. Lo sviluppo di appositi script ha permesso di generare in sequenza pacchetti mSJ,mB,μSJ e μSC con un numero crescente di campioni al fine di compararne le dimensioni in termini di byte. L'elenco dei valori utilizzati nella prova è il seguente:

$$[5, 10, 20, 50, 100, 250, 500, 1000, 2000, 4000, 8000, 16000, 32000, 64000]$$

Del protocollo di serializzazione CBOR esistono diverse implementazioni, è stato necessario quindi selezionare la migliore prima di effettuare le successive misure. In rete sono disponibili quattro implementazioni python open-source di CBOR: flunn [40], flynn [41], cbor [42] e cbor2 [43]. Ognuna di queste implementa correttamente il protocollo, fornendo lo stesso output a parità di input, le differenze si trovano sulle metodologie utilizzate nella scrittura del codice. Una breve verifica ha permesso di capire quele tra queste fosse di gran lunga la più veloce ad effetture le fasi di codifica e decodifica. Questa prova è stata eseguita misurando il tempo strettamente necessario alla sola serializzazione di una generica struttura μSEED e alla rispettiva deserializzazione, i risultati sono esposti nella tabella 3.1. In entrambi i tempi in esame l'implementazione cbor2, sviluppata da Alex Grönholm, è più rapida nell'eseguire tali operazioni. Cbor2 nei confronti del tempo totale necessario alla codifica e decodifica, ha un guadagno che va dal 22% fino a circa il 50% rispetto le altre implementazioni, la scelta è quindi ricaduta su tale implementazione. Per le implementazioni di JSON e Pickle sono state utilizzate le versioni standard fornite da python mentre per la

	Tempi		
Implementazione	**Codifica**	**Decodifica**	**Totale**
cbor	0.300	0.318	**0.618**
cbor2	0.185	0.292	**0.477**
flunn	0.231	0.381	**0.612**
flynn	0.285	0.655	**0.940**

Tabella 3.1: Confronto dei tempi di codifica e decodifica delle diverse implementazioni python di CBOR. I tempi sono in ms.

creazione del formato miniSEED, come già detto, è stata utilizzata la libreria Obspy. Prima di procedere con la parte di simulazione dei client MQTT è stato eseguito un test ’offline’, ovvero implementando un client Publisher che generava le quattro tipologie di pacchetto senza inviarle al broker, così da andare a rilevare i tempi di codifica utilizzando un numero di campioni variabile come in precedenza. Al fine di rendere questi valori più realistici, in questo caso è stato considerato come numero minimo di campioni il valore di 250. In questo test la generazione di pacchetti avviene in modo continuativo senza pause intermedie, per ogni valore del numero di pacchetti sono state eseguite 100 misurazioni così da rendere i risultati attendibili. Lo step successivo è stato quello di sviluppare i client Publisher e Subscriber sulla piattaforma open source Docker, la quale permette di semplificare lo sviluppo e la distribuzione di applicazioni sotto forma di ’container’. Il container Docker è stato creato mediante un Dockerfile dove vengono elencate tutte le librerie python necessarie al Publisher/Subscriber in aggiunta al sistema operativo Linux Ubuntu 20.04. Oltre al sistema operativo e le librerie necessarie vengono installati anche gli script del Publisher e Subscriber che, utilizzando l’implementazione del protocollo MQTT Paho(ideale per piattaforme embedded), comunicano con un broker, installato sulla macchina locale, mediante la porta 1883. Il broker utilizzato è lo stesso del primo studio, ovvero Mosquitto. L’architettura di misurazione è quindi composta dai due client che lavorano in modo isolato in un container docker sulla stessa macchina in cui è installato il broker. Può quindi essere rappresentata in Fig.3.10, dove per semplicità è stato evidenziato solo il client Subscriber ma in realtà all’interno dello stesso docker è contenuto anche il client Publisher che invia pacchetti verso il broker attraverso la porta 1883. All’interno del container è stato installato, sempre tramite Dockerfile,

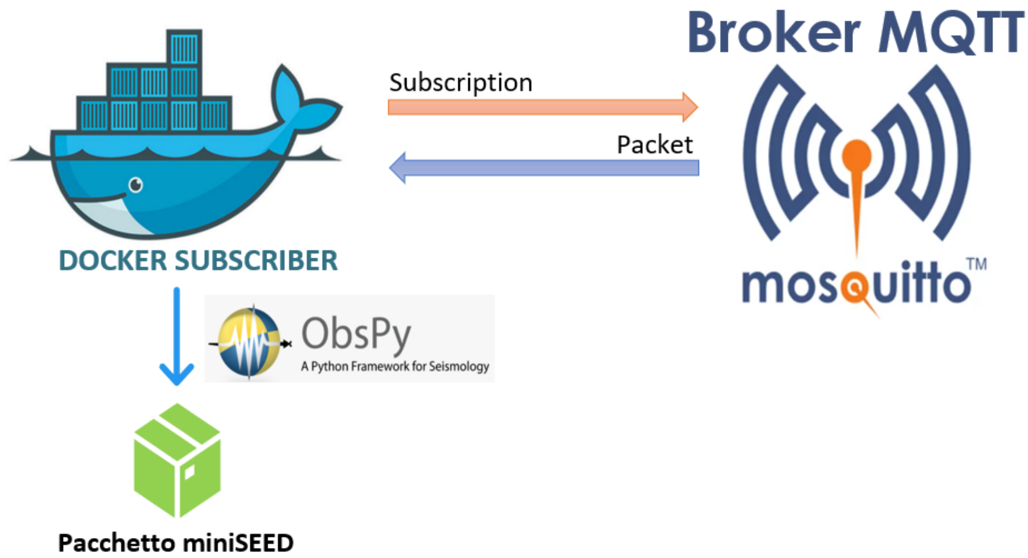


Figura 3.10: Schema dell'implementazione su container Docker del sistema di simulazione.

Jupyter Notebook per semplificare l'ambiente di sviluppo degli script di simulzione. Jupyter Notebook estende l'approccio basato su console all'elaborazione interattiva fornendo un'applicazione web-based così da rendere la scrittura del codice rapida attraverso un apposito editor testuale multilinguaggio. Mediante questo applicativo lo scambio di file tra container docker e macchina locale è stato rapido grazie all'interfaccia di visualizzazione grafica rappresentata in Fig.3.11 Jupyter Notebook

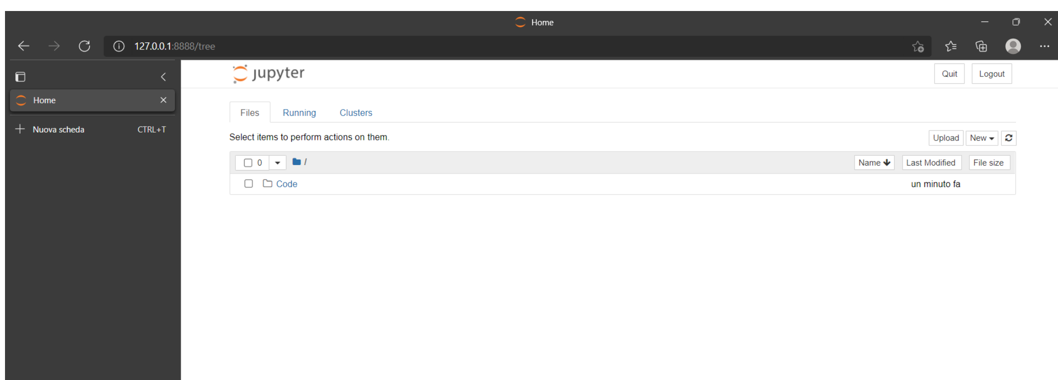


Figura 3.11: Interfaccia grafica dell'applicativo Jupyter Notebook.

sfrutta la porta 8888 e all'avvio del container docker, viene fornito un link ed un token che permettono la visualizzazione grafica dell'interfaccia attraverso il browser. Nelle simulazioni vengono fatti scambiare pacchetti tra i due client, contenenti il range di

campioni citato, per valutare i tre intervalli temporali:$t_{codifica}$,$t_{trasmissione}$ e $t_{decodifica}$. Per sincronizzare i due client, visto che ci trovavamo sulla stessa macchina, è stato utilizzato il modulo *SharedMemory* della classe python *multiprocessing* il quale permette di utilizzare una memoria condivisa, etichettata da un nome, alla quale chiunque può interfacciarsi. Un ciclo di misurazioni ha inizio con la generazione del pacchetto e termina con il salvataggio dello stesso in formato miniSEED. I campioni all'interno del pacchetto sono stati generati randomicamente con valori interi ma, per rendere le misurazioni consistenti, l'inizio di $t_{codifica}$ è stato spostato subito dopo la fine della generazione random. Questo ha permesso di evitare tempi diversificati per la generazione dei campioni per ogni pacchetto, infatti in un'applicazione reale i tempi di campionamento sono ben definiti. Per rendere le simulazioni più coerenti possibile è necessario che la macchina non esegua operazioni che possano andare a limitare le risorse al container docker per un certo tempo. Si è cercato quindi di sospendere all'interno del sistema operativo tutte le applicazioni non necessarie ai fini della prova. Come ulteriore tutela, per ogni valore del numero di campioni, sono stati eseguiti consecutivamente 100 cicli di misura così che la media non risenta di eventuali task temporanei del sistema operativo. La macchina sulla quale è stato eseguita questa simulazione ha le seguenti caratteristiche:CPU Intel Core i3-4000M, 8MB cache, 2.40 GHz, 8 GB RAM e sistema operativo Windows 10. A questa macchina è stato assegnato il nome 'macchina 1' per poterla identificare nel proseguo del testo. Sulla stessa macchina sono stati misurati anche i tempi di codifica iniziali in modalità 'offline'. In modo tale da verificare la correttezza dei risultati sperimentali sono state eseguite le stesse operazioni, condividendo il container docker, su un altra macchina più performante con CPU AMD Ryzen 9 5900HX, 16MB cache, 3.30 GHz, 16 GB RAM e sistema operativo Windows 10. Mediante questa prova ci si aspetta di riscontrare risultati proporzionalmente simili ai precedenti ma scalati nel tempo, vista la differenza generazionale tra i due processori. A questa macchina è stato assegnato il nome 'macchina 2' per poterla identificare nel proseguo del testo. Per simulare invece una situazione reale, cioè dove i due client non hanno parità di risorse, è stata utilizzata una Raspberry Pi 2 Model B v1.1 con CPU ARM Cortex-A7 Quad Core 900 MHz Broadcom BCM2836 32-bit ed 1 GB RAM. In particolare è stato collocato lo script Publisher nella Raspberry così da riprodurre

un nodo di misurazione sismica della rete EEW(il broker Mosquitto risiede nella scheda) mentre il Subscriber è stato lasciato all'interno del docker della seconda macchina più performante(macchina 2). Rispetto alle due prove precedenti, in questa i tempi di codifica e decodifica saranno molto diversi vista la disuguaglianza nelle componenti hardware dei due dispositivi, uno con basse capacità computazionali e l'altro con una CPU molto performante, anche se lontano dalle prestazioni possibili con piattaforme cloud. La sincronizzazione tra dispositivo Publisher(Raspberry) e dispositivo Subscriber(macchina 2) è avvenuta mediante l'utilizzo del protocollo NTP come nel confronto tra il protocollo MQTT e SeedLink. Tutte le misurazioni sono state condotte al termine della fase di connessione e sottoscrizione previste dal protocollo MQTT.

Capitolo 4

Risultati

Vengono ora illustrati i risultati delle misurazioni descritte nel capitolo precedente, partendo dal confronto delle latenze tra i protocolli MQTT e SeedLink per poi passare ad analizzare i formati dati che garantiscano le migliori prestazioni in termini di dimensione del pacchetto e tempi di codifica/decodifica.

4.1 Confronto tra MQTT e SeedLink

Lo scopo di questa misurazione è quello di verificare le latenze dei due protocolli in esame, intesa come la differenza temporale tra l'istante di ricezione e l'istante di invio di un medesimo pacchetto miniSEED, inviato tramite SeedLink o come payload di un messaggio MQTT. Per quanto riguarda il client MQTT, per istante di ricezione si intende l'istante in cui il payload del messaggio viene estratto e può essere processato come pacchetto miniSEED, così da non incorrere in errori dovuti all'utilizzo di differenti tecniche di processing di ogni pacchetto ricevuto. In Tabella 4.1 sono riportate le statistiche dei risultati ottenuti dai due procolli. Il protocollo MQTT presenta i risultati migliori in termini di latenza, attestandosi su un tempo di consegna medio di 33.17 ms con una deviazione standard pari a 17.03 ms, mentre il protocollo SeedLink risulta avere un ritardo medio pari a 696.13 ms con deviazione standard di 439.90 ms. Per completezza di informazioni abbiamo riportato anche le misurazioni di mediana e 90th percentile. Nel grafico in Fig. 4.1 sono riportate le distribuzioni di probabilità ottenute dalle simulazioni, dove è stato suddiviso l'asse delle ascisse in bin di 50 ms. Dal grafico risulta evidente come le distribuzioni siano molto differenti. La distribuzione riguardante il protocollo MQTT presenta

Protocol	Mean	St.Dev.	Median	90th perc.
MQTT	33.17	17.03	30.79	33.37
SeedLink	696.13	439.90	592.83	1346.64

Tabella 4.1: Statistiche ottenute dalle simulazioni per le latenze dei protocolli MQTT e SeedLink. I tempi sono in ms.

una asimmetria positiva, pertanto è indicato calcolare anche il range interquartile che risulta essere di 1.074 ms. Tale valore confrontato con il 90th percentile ed il valore di mediana in Tabella 4.1 ci porta ad affermare che le prestazioni del protocollo sono molto stabili durante tutte le simulazioni, con sporadici outliers. Analizzando la distribuzione per quanto riguarda il protocollo SeedLink si evidenzia la approssimazione ad una funzione multimodale con un primo valore di moda nel range 250-300 ms, un secondo nel range 600-650 ms ed un terzo valore nel range 1150-1220 ms. Cercando quindi un fit con una distribuzione con tali caratteristiche, abbiamo ricavato tre valori di media e deviazione standard: per l'approssimazione alla prima distribuzione un valor medio di 280 ms con deviazione standard pari a 78 ms, per la seconda valor medio di 597 ms e deviazione standard di 89 ms, mentre per la terza un valore di 1187±235 ms.

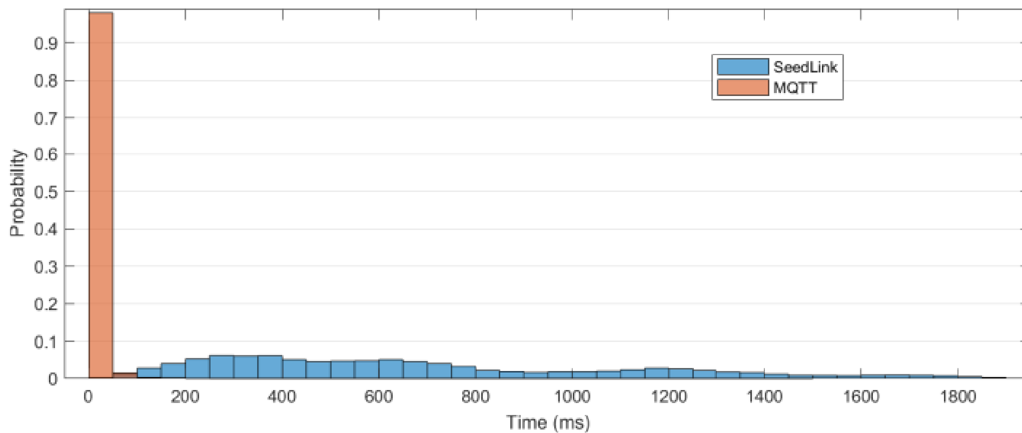


Figura 4.1: Distribuzione di probabilità dei T_l per i protocolli MQTT e SeedLink. Gli istogrammi sono stati normalizzati.

Le simulazioni quindi hanno evidenziato un comportamento non uniforme del protocollo SeedLink, con una grande variabilità di risultati visto il range di valori ottenuti. Al fine di confermare che tali valori non venissero influenzati da altri fattori,

abbiamo installato il client SeedLink nella stessa macchina che simula i dispositivi, eliminando in questo modo qualsiasi latenza dovuta alla rete. Abbiamo ottenuto distribuzioni identiche rispetto a quelle ottenute dalle simulazioni precedenti, spostate verso sinistra (spostamento dovuto proprio al mancato invio dei pacchetti in rete), confermando quindi la relativamente alta instabilità del protocollo in confronto al protocollo MQTT, nonché tempi di latenza medi notevolmente più alti. Pertanto, inviando medesimi pacchetti miniSEED, il protocollo MQTT risulta essere più performante in termini di latenza e più sicuro in termini di stabilità, candidandosi quindi ad una possibile compresenza con il protocollo SeedLink, e ad un utilizzo in applicazioni di EEW dove tempo ed affidabilità sono requisiti fondamentali.

4.2 Confronto tipologie di formato dati

Avendo riscontrato un beneficio in termini di latenza a parità di informazione trasmessa, mediante l'utilizzo del protocollo MQTT, in questa sezione vengono illustrate le caratteristiche di ogni tipologia di formato dati implementata, al fine di migliorare ulteriormente le performance del sistema nel suo complesso. L'obbiettivo è quello di trovare il formato dati che garantisca le dimensioni minori del pacchetto e contemporaneamente dei tempi di elaborazione minimi per applicazioni di EEW. Il numero di campioni che vengono trasmessi per ogni messaggio in questo ambito non è molto grande per via della frequenza di campionamento che è dell'ordine del centinaio di Hz e della necessità di avere minori ritardi possibili nell'andare a fornire un allerta rapida. Per esempio, supponendo di avere un dispositivo di misurazione che campiona a 200Hz ed un messaggio contenente 2000 campioni, allora verranno trasmessi frame temporali di 10 secondi. I risultati che verranno esposti non possono essere considerati assoluti ma sono relativi ad una specifica applicazione, infatti le caratteristiche di una codifica dipendono anche dalla tipologia di dati sotto esame.

4.2.1 Dimensioni del pacchetto

La dimensione del pacchetto risultante dalla serializzazione nei diversi formati è l'oggetto di analisi di questa sezione. Questa dimensione sarà effettivamente quella occupata nel payload del messaggio MQTT. Per ognuno dei valori di campioni

selezionati, sono stati serializzati i pacchetti mediante i quattro formati e ne sono state misurate le dimensioni in Kbyte. Allo scopo di fornire maggiore chiarezza sui risultati ottenuti, questi ultimi sono stati suddivisi in: un confronto tra pacchetti di dimensioni relativamente grandi ed uno su quelli con un numero ristretto di campioni all'interno. In Fig.4.2 sono rappresentati i risultati di questo confronto per pacchetti da 250 fino

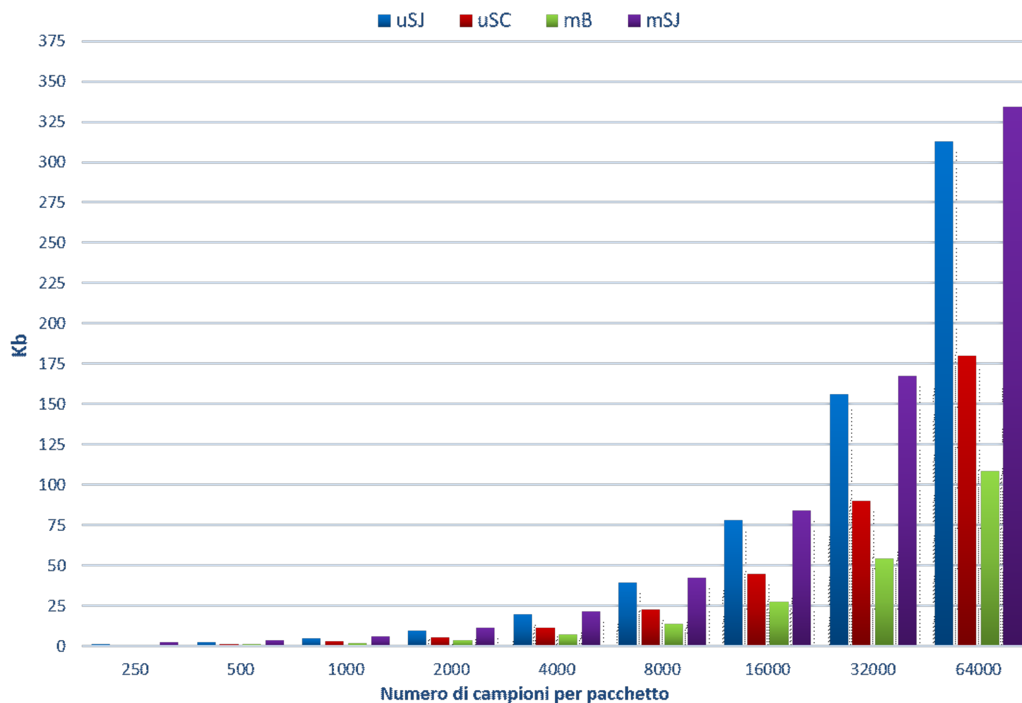


Figura 4.2: Confronto tra le dimensioni dei pacchetti per i diversi formati dati.

a 64000 campioni. Quest'ultimo valore è relativamente grande per sistemi di allerta rapida, in questo studio però si è deciso di effettuare un analisi un pò più generale in modo tale da conoscere il comportamento dei sistemi in circostanze "straordinarie". Come è facilmente riscontrabile dal grafico, il formato mSJ(in viola) è quello che necessita delle dimensioni maggiori rispetto agli altri, per ogni valore del numero di campioni. Tale effetto è dovuto al tipo di informazione che esso va a rappresentare, infatti non vengono serializzati direttamente i dati, bensì il flusso di byte generato è una descrizione di un oggetto astratto dell'ambiente python, ciò necessita di un overhead maggiore sul pacchetto. All'opposto, il formato mB riesce sempre a garantire la dimensione minori in questo range di valori. Tra mB e mSJ era auspicabile tale differenza dato che mB rappresenta il pacchetto miniSEED direttamente con la sua forma binaria. Facendo riferimento invece ai due formati che sfruttano la struttura

del pacchetto μSEED, si nota come costantemente la loro dimensione si collochi in un intervallo delimitato dalle due precedenti, mB e mSJ. Paragonando i formati μSJ e μSC è evidente il vantaggio in termini di dimensioni del formato μSEED serializzato con CBOR. Negli obbietti di sviluppo dello standard CBOR infatti veniva evidenziata la volontà di rappresentare l'informazione in maniera coincisa rispetto a JSON a discapito della leggibilità umana. Facendo un focus su quello che accade con un numero di campioni piccolo, ci si accorge come il comportamento descritto finora non si mantiene propriamente valido anche in questo range di valori, che va da 5 a 500 campioni. Fino a 250 campioni le dimensioni del pacchetto miniSEED binario si mantengono costanti e ciò accade perchè è stata fissata la dimensione standard del pacchetto per lo streaming. In pratica, nel pacchetto da 512 byte vengono inseriti campioni fino al raggiungimento della capienza massima senza l'applicazione di alcuna compressione. Se la dimensione di un pacchetto miniSEED non è sufficiente a contenere il numero di campioni desiderato, mediante la libreria Obspy, vengono creati all'interno dello stesso file, due o più pacchetti miniSEED affinchè sia possibile contenere i campioni richiesti. Come si vede dalla Fig.4.3, fino al valore di 250 è bastato un unico pacchetto per contenere tutte le misure intere generate in modo random, nel momento in cui i campioni sono diventati 500 non era più sufficiente un unico pacchetto ma ne servivano due, per questo la dimensione è raddoppiata. Questo non è vero per i restanti formati che mostrano un espansione della dimensione proporzionale al numero di valori in ingresso e grazie a questo, il formato basato su CBOR permette di avere le dimensioni minori fino ad un numero di campioni inferiore a 250. Facendo riferimento alla struttura dati μSEED, a quanto corrisponde il guadagno in termini di dimensioni che l'utilizzo di CBOR porta con se rispetto a JSON? Per ripondere a tale quesito è stato calcolato in percentule il guadagno per ogni numero di campioni. Si è voluto comparare nella stessa analisi il comportamento di Pickle nella serializzazione della stessa struttura μSEED. Pickle fino a questo momento era stato utilizzato come strumento utile alla generazione del pacchetto in formato mSJ, mentre in questo caso viene verificato un suo possibile utilizzo nella serializzazione del μSEED. Il formato Pickle, essendo sviluppato proprio per il linguaggio python, risulta molto ottimizzato nei tempi di codifica e decodifica delle informazioni. Come CBOR è un formato di serializzazione binario e quindi si è

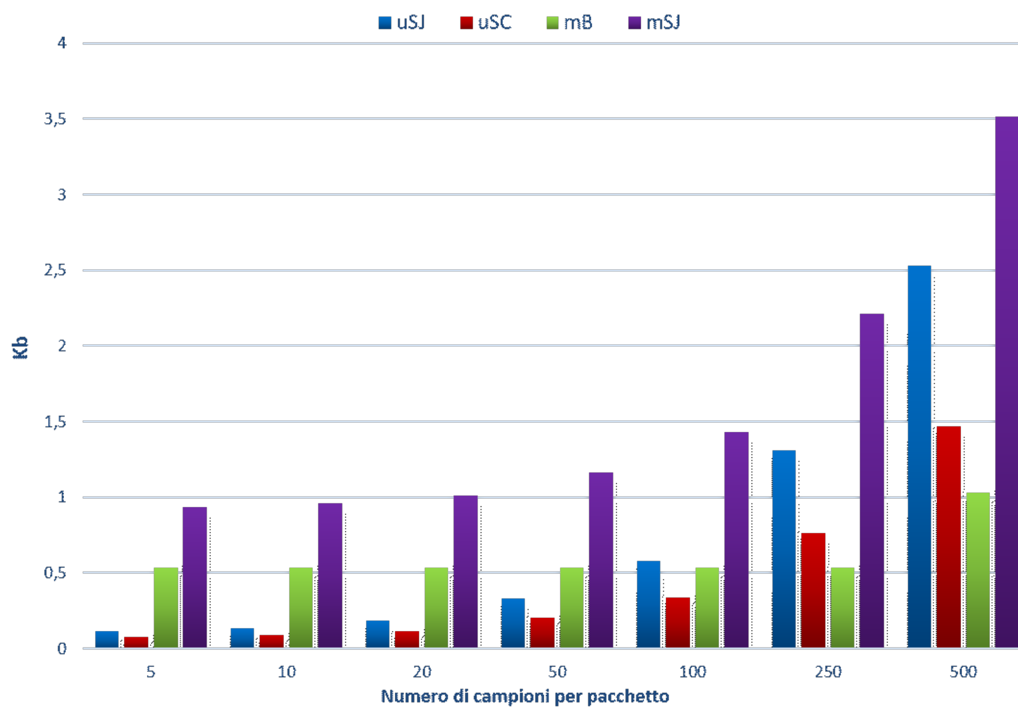


Figura 4.3: Confronto tra le dimensioni dei pacchetti per i diversi formati dati con un numero di campioni ridotto.

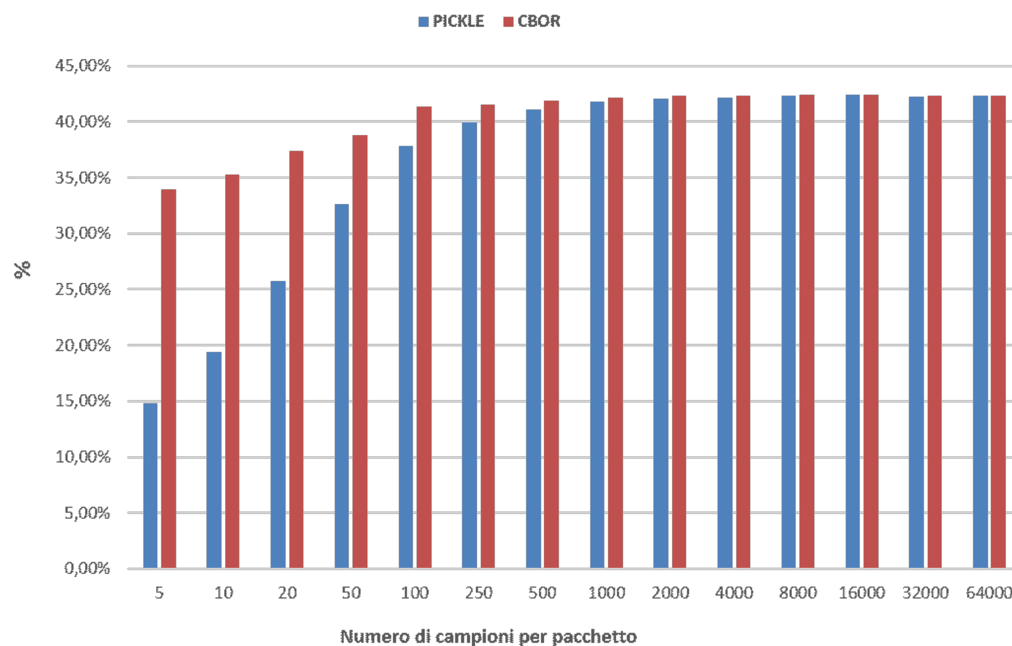


Figura 4.4: Rappresentazione del guadagno percentuale, in termini di dimensioni, dei formati CBOR e Pickle rispetto a JSON.

voluto confrontare il suo comportamento con quello di JSON e CBOR. In Fig.4.4 è riscontrabile come, per questa tipologia di informazioni, CBOR consenta una rappresentazione più coincisa già da valori piccoli del numero di campioni. Al loro aumentare Pickle si avvicina asintoticamente ai risultati ottenuti da CBOR. Questo è soltanto uno degli aspetti per cui si è deciso di non utilizzare Pickle. L'unico vantaggio riscontrato è quello riguardante i tempi di elaborazione, mentre gli svantaggi legati al suo utilizzo in questa applicazione possono essere così riassunti:

- Formato Python-specific.
- Non è sicuro per le trasmissioni in rete.
- CBOR ha prestazioni migliori rispetto alle dimensioni.

Già il vincolo riguardo al linguaggio di programmazione è rilevante di per se, a questo, oltre le dimensioni del dato serializzato, si aggiunge la problematica legata alla sicurezza. Attraverso la serializzazione di un oggetto con Pickle (pickling) possono essere inserite funzioni che, in fase di deserializzazione, permettono di eseguire codice arbitrario all'interno della macchina. La sicurezza dei dispositivi all'interno di una rete in questo caso sarebbe esposta ad un rischio di attacchi man-in-the-middle. Ritornando al grafico in Fig.4.4, una caratteristica evidente di μSC è la sua costanza nella riduzione delle dimensioni rispetto a JSON, si parte infatti da un 34% per pacchetti contenenti alcuni campioni fino ad arriavare al 42% per pacchetti molto più grandi. Per fare un esempio, scegliendo un valore di 64000 si ha un risparmio di 132 Kb non trascurabile, mentre con 5 campioni il guadagno risulta già essere di 38 byte.

4.2.2 Tempi di codifica

Dopo una valutazione delle dimensioni del pacchetto, l'altro aspetto fondamentale riguarda il tempo necessario affinchè esso sia pronto per essere trasmesso. In questo arco temporale viene creata la struttura dati miniSEED/μSEED, comprensiva di intestazione e campioni, per poi essere elaborata dalla funzione di serializzazione. Prima di passare allo studio degli intervalli temporali dei diversi formati all'interno della comunicazione MQTT, si è proceduto a considerare i soli tempi di codifica. Queste misure sono state fatte da un client di tipo Publisher senza che questo sia connesso al broker, generato quindi un pacchetto per ogni formato questo viene

scartato. Per ognuno dei 4 formati sono state effettuate 100 misurazioni consecutive per ciascun valore di campioni contenuti e,i risultati illustrati, considerano un tempo medio di queste. Visto che il client non è connesso al broker, il test può essere considerato in modalità offline. Ciò che la differenzia dai succesivi test, oltre l'utilizzo del protocollo MQTT, è il ritardo temporale non considerato per la raccolta dei campioni di un pacchetto. Nei risultati delle prossime sezioni infatti, questo intervallo è stato fissato a 2 secondi ma, volendo simulare una situazione reale, si dovrebbero considerare il numero di campioni e la frequenza di campionamento per impostare automaticamente il ritardo, da aggiungere precedentemente alla creazione della struttura dati. Per non rendere le simulazioni troppo dispendiose in termini di tempo è stato selezionato questo valore che può rispecchiare una situazione reale. Tempi di simulazione troppo estesi aumentano la probabilità di attivazione di eventuali task da parte della macchina che rendono i risultati inattendibili. In Fig.4.5 sono

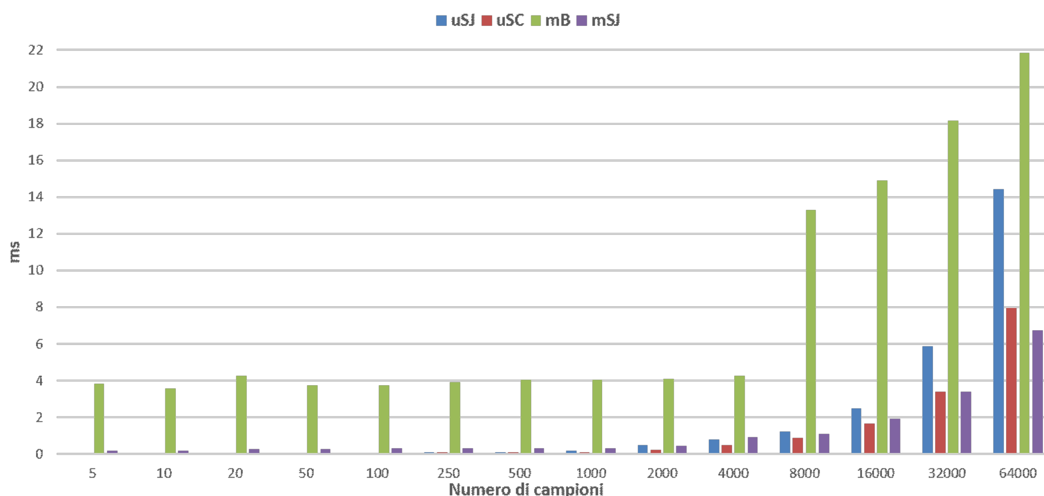


Figura 4.5: Tempi medi di codifica in modalità offline.

riportati gli andamenti dei tempi di codifica medi per tutto il range di valori preso in considerazione nella precedente analisi delle dimensioni. Come si evince dal grafico, i tempi necessari alla codifica mB si dimostrano sempre superiori, in particolare da 8000 campioni e superiori. Mentre, confrontando i due formati basati su μSEED, si nota come con CBOR, all'aumentare del numero di campioni, sia presente un guadagno temporale sempre più ampio. Fino ad un numero di campioni di 250, le misurazioni relative al μSJ e μSC, appaiono impercettibili per la scala adottata nell'istogramma a barre. Nel range da 5 a 100 campioni inoltre, il periodo necessario

alla codifica per i formati mB e mSJ possono essere considerati quasi invarianti. Per questo motivo nelle seccessive prove in modalità online, questi valori del range non saranno considerati, facendo partire la nostra analisi da 250 campioni contenuti in un pacchetto.

4.2.3 Simulazioni su Docker

Nella simulazione vera e propria su piattaforma docker, come vedremo, i risulati dei tempi di codifica tendono a cambiare. Questo accade non per via dello sviluppo in docker dell'architettura, bensì per la natura stessa della simulazione. Andando infatti cercare di implementare una struttura il più realistica possibile, si è tenuto conto del ritardo di raccolta dei campioni già discusso. In questa sezione verranno esposti i risultati raccolti, inizialmente utilizzando la prima macchina con modeste capacità computazionali descritta nel capitolo precedente, in seguito verrà mostrato come, avendo maggiore potenza computazionale(macchina 2), i tempi di processamento si riducano in maniera considerevole. In entrambi i casi i due client sono contenuti sullo stesso docker mentre il broker è installato direttamente nella macchina. I risultati dei tempi di codifica medi $t_{codifica}$ durante la simulazione sulla macchina 1 sono riportati in Fig.4.6. Come è facilmente riscontrabile confrontanto questo grafico con

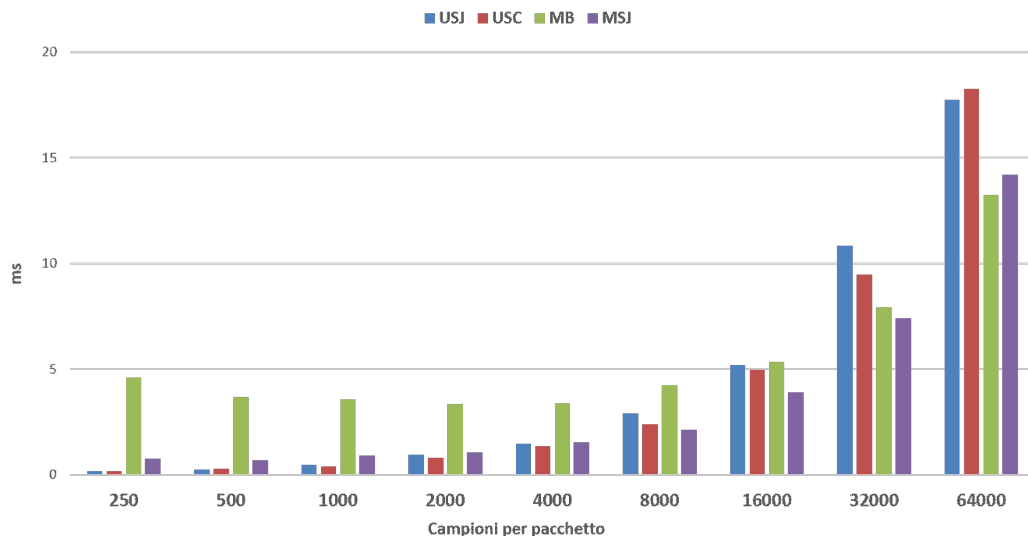


Figura 4.6: Tempi medi di codifica del publisher su piattaforma docker. La macchina di riferimento è la 1.

quello precedente effettuato in modalità offline, i risultati evidenziano differenze di

comportamento tra i formati. Tali diseguaglianze sono dovute a diversi fattori:

- Il sistema operativo ospitante i client è cambiato da Windows a Linux.
- Tra pacchetti consecutivi è presente un ritardo temporale di 2 secondi per la raccolta dei campioni.

Infatti, il formato mB, per valori di campioni di 8000 e superiori, non subisce un brusco aumento del tempo di codifica, aumenta invece linearmente con tempi perfino inferiori ai formati che utilizzano μSEED per un numero di campioni di 32000 e 64000. Il comportamento degli altri 3 formati invece, segue abbastanza fedelmente quello del test precedente, si riscontra soltanto una piccola differenza tra μSJ e μSC a 64000 campioni dove, il formato basato su CBOR, tende a prevalere. Si deve però tenere in considerazione la grande mole di campioni per pacchetto e la possibilità di errore nella misura da parte della macchina durante la simulazione(differenza di $0.5ms$). All'interno della simulazione sono stati misurati anche i tempi di trasmissione $t_{trasmissione}$ tra i due client che, lavorando in locale ovvero senza tempi effettivi di attraversamento della rete, corrisponono al tempo necessario affinchè il pacchetto arrivi al broker e questo lo inoltri all'altro client. Nel tempo di trasmissione è compreso anche l'intervallo necessario alla pubblicazione e alla ricezione del messaggio. Questo parametro, strutturando così la misura, sarà condizionato soltanto dalle capacità della macchina ospitante e dalle dimensioni del pacchetto scambiato. Tali risultati per $t_{trasmissione}$ sono riportati in Fig.4.7. Com'era logico aspettarsi, gli andamenti seguono abbastanza fedelmente quelli riportati per le dimensioni del pacchetto. I pacchetti codificati con mSJ e μSJ, disponendo del pacchetto di dimensioni maggiori, necessitano di più tempo per il suo trasferimento. Per quanto riguarda i tempi di decodifica $t_{decodifica}$, in Fig.4.8 viene chiaramente evidenziato come, trasmettendo direttamente la stringa binaria di un pacchetto miniSEED, il Subscriber necessiti di tempi minimi per salvarlo, in quanto non deve processarlo in nessun modo. Questo porta un grande vantaggio in termini di tempo rispetto agli altri formati, i quali hanno bisogno di operazioni di deserializzazione inverse a quelle effettuate in fase di codifica. Il formato mSJ sfrutta la caratteristica di generare il formato miniSEED lato Publisher, descrivendolo attraverso il particolare processo di serializzazione nel payload del messaggio, per poi deserializzarlo una volta ricevuto e salvarlo. Questa

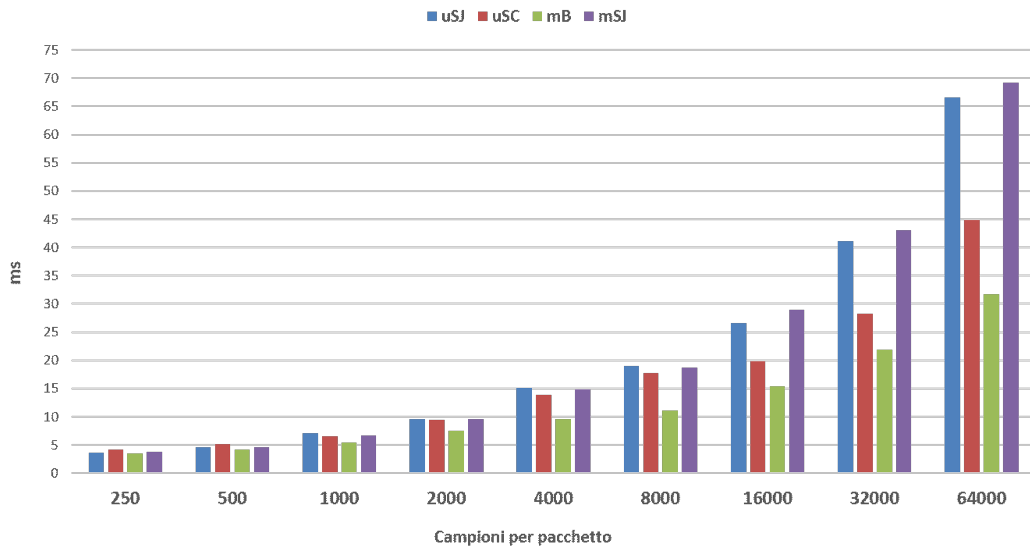


Figura 4.7: Tempi medi di trasmissione su piattaforma docker. La macchina di riferimento è la 1.

tecnica consente a mSJ di essere più rapido in decodifica rispetto ai formati che sfruttano μSEED. Questi ultimi infatti, dovendo generare il pacchetto in formato miniSEED dalla parte del client ricevente, richiedono un tempo superiore. L'unica operazione che differenzia questi due è quella di deserializzaione, uno utilizza JSON e l'altro CBOR. Come si evince da Fig.4.8, JSON risulta essere più rapido in questa fase. A titolo di esempio, in Fig.4.9, è rappresentata la distribuzione di $t_{codifica}$,

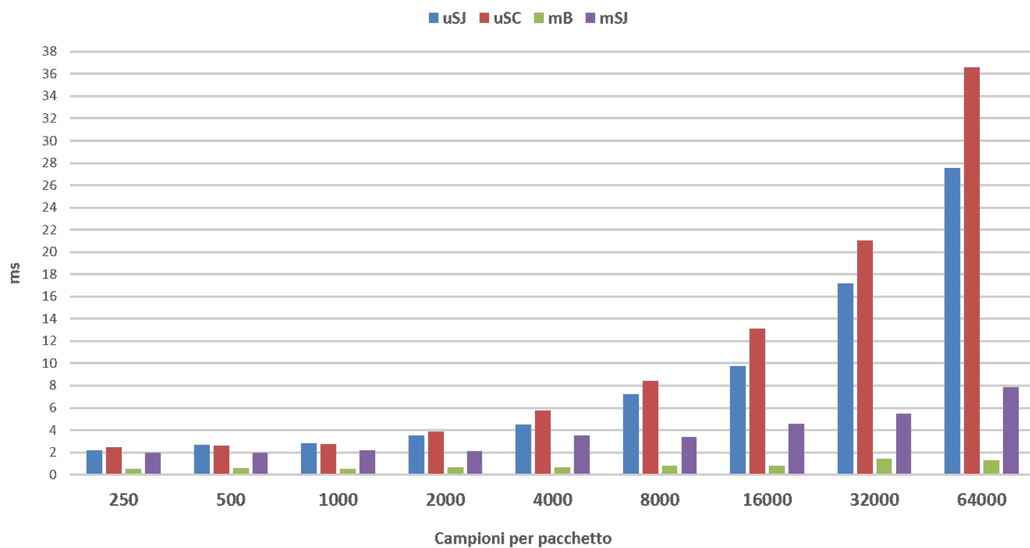


Figura 4.8: Tempi medi di decodifica del subscriber su piattaforma docker. La macchina di riferimento è la 1.

$t_{trasmissione}$ e $t_{decodifica}$ dei quattro formati dati, per un pacchetto contenente 2000 campioni(con una frequenza di campionamento di 200Hz il pacchetto racchiude 10s di registrazione). In questo grafico vengono racchiuse tutte le considerazioni fatte fin ora

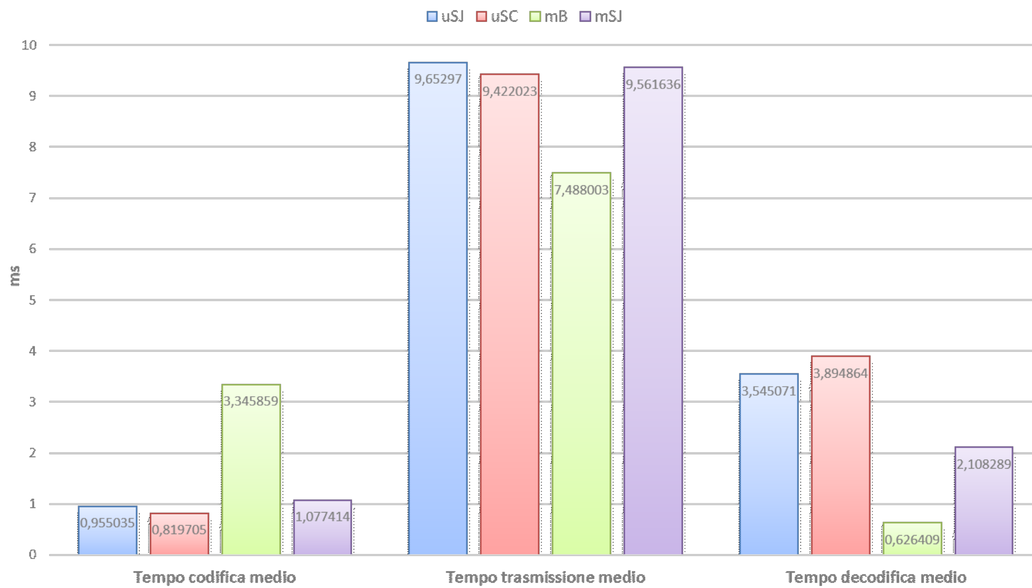


Figura 4.9: Distribuzione dei tempi, nei diversi formati, per un pacchetto da 2000 campioni su piattaforma docker. La macchina di riferimento è la 1.

riguardo le dimensioni e la velocità di elaborazione di ogni formato. Una valutazione complessiva è stata realizzata sommando i tre intervalli temporali misurati, così da confrontare il tempo totale medio impiegato da ogni formato dati:

$$t_{totale} = t_{codifica} + t_{trasmissione} + t_{decodifica} \tag{4.1}$$

Nel grafico in Fig.4.10 si può constatare come, fino a 500 campioni sia più rilevante la componente del $t_{codifica}$ riscontrando risultati equivalenti se non migliori per i formati μSEED. Da 1000 campioni in poi, il guadagno apportato da mB in decodifica diventa predominante rispetto agli altri. Tra i due formati μSEED, quello serializzato con CBOR, mostra risultati complessivi migliori rispetto all'altro, per pacchetti via via più grandi. Per 64000 campioni, μSC porta ad un vantaggio di tempo totale medio di 13ms paragonandolo a μSJ mentre, rispetto a mB, ha un ritardo di 52ms. Ogni formato ha caratteristiche tali da distribuire, in modo diverso dagli altri, le operazioni nei determinati intervalli temporali. Per analizzare questa

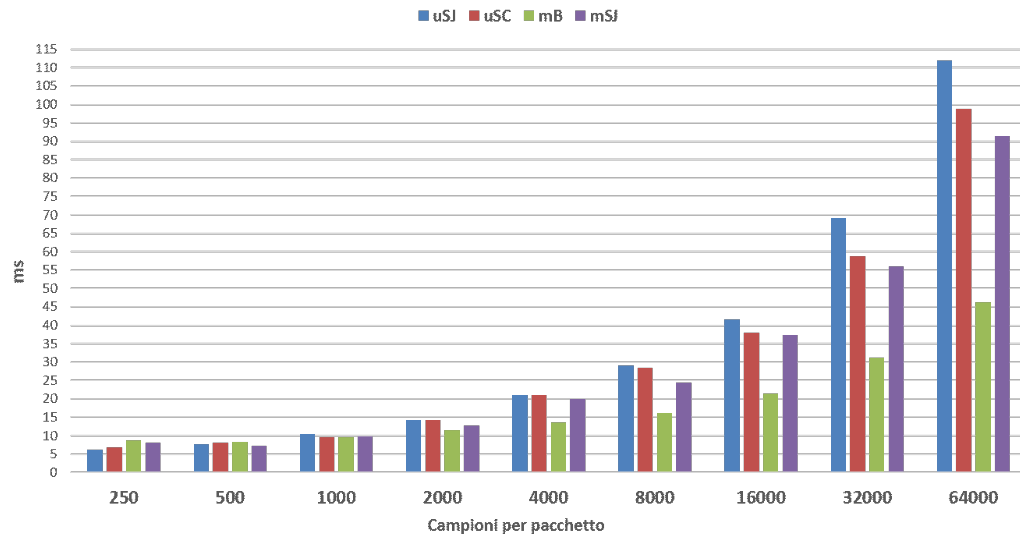


Figura 4.10: Tempi medi totali su piattaforma docker. La macchina di riferimento è la 1.

distribuzione sono stati calcolati in percentuale i tre intervalli temporali di riferimento rispetto al totale. lo studio di questa ripartizione è stato sottoposto a tutto il range di valori dei campioni per ogni formato dati. Si è riscontrata un'evoluzione di queste percentuali dipendente dal numero di campioni e al fine di svolgere una caratterizzazione generale ne son stati considerati dei valori medi. Questi sono stati poi inseriti in un grafico a torta per ciascun formato(Fig.4.11): I grafici a torta risultano utili a comprendere il partizionamento del carico di lavoro tra i due client del sistema. Le distribuzioni di μSC(Fig.4.11c) e μSJ(Fig.4.11d) sono molto simili, l'unica piccola differenza riscontrabile è dovuta alla dimensione del pacchetto del 40% superiore per la serializzazione JSON, la quale comporta un tempo di trasmissione superiore. La caratteristica fondamentale che le accomuna è quella di avere una percentuale minima rispetto al totale per la fase di codifica. Questa può risultare decisiva in un ambiente reale, nel quale un dispositivo a basse capacità computazionali si deve occupare della fase di codifica mentre ad un centro di raccolta dati(es. piattaforma Cloud) è assegnato il compito di effettuare la decodifica. Aver effettuato le simulazioni sulla stessa macchina ci ha permesso di poter raffigurare queste distribuzioni, avendo fornito pari capacità computazionali ai due client. Per quanto riguarda gli ulteriori due formati basati su miniSEED i risultati a confronto sono ovviamente diversi. Il formato mB(Fig.4.11a) mostra

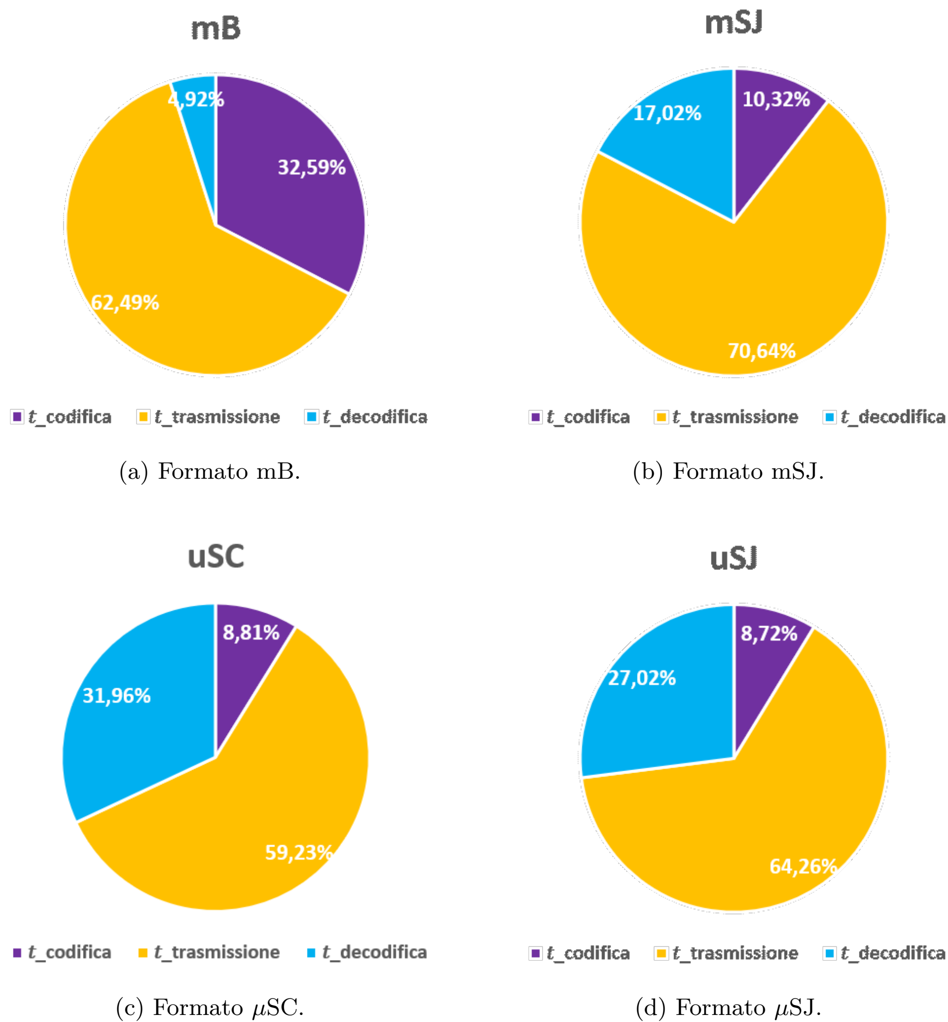


Figura 4.11: Rappresentazione delle distribuzioni temporali in percentuale dei quattro formati.

un carico di lavoro completamente sbilanciato verso il nodo di misurazione mentre mSJ(Fig.4.11b) risulta essere il più equilibrato ma allo stesso tempo produce il pacchetto di dimensioni maggiori. Mediante la condivisione dello stesso container docker si son potute svolgere le stesse simulazioni sulla macchina 2. I risultati illustrati di seguito mostrano una notevole riduzione dei tempi di processamento per via dell'architettura hardware della macchina decisamente più evoluta. Per quanto riguarda l'intervallo temporale di codifica Fig.4.12, il trend del formato mB viene rispettato e con esso anche il rapporto che intercorre con il formato mSJ. L'unica differenza riscontrabile si ha per pacchetti relativamenti grandi con μSC dove, pur mantenendo un guadagno positivo rispetto a μSJ, i risultati tendono ad essere

migliori rispetto al mB, cosa che non accadeva in fase di codifica nelle simulazioni sulla macchina 1 (Fig.4.6). Da notare come mB, per pacchetti relativamente piccoli, passi

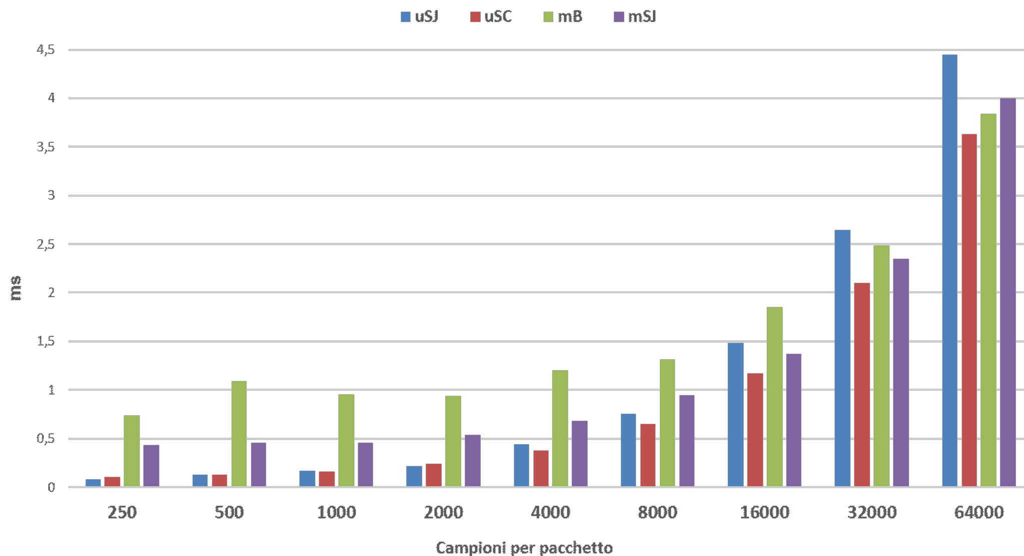


Figura 4.12: Tempi medi di codifica del publisher su piattaforma docker. La macchina di riferimento è la 2.

da un tempo necessario di $4/5ms$ nella prima macchina a circa $1ms$ nella seconda. Le dimensioni dei pacchetti ovviamente rimangono costanti e con loro le proporzioni

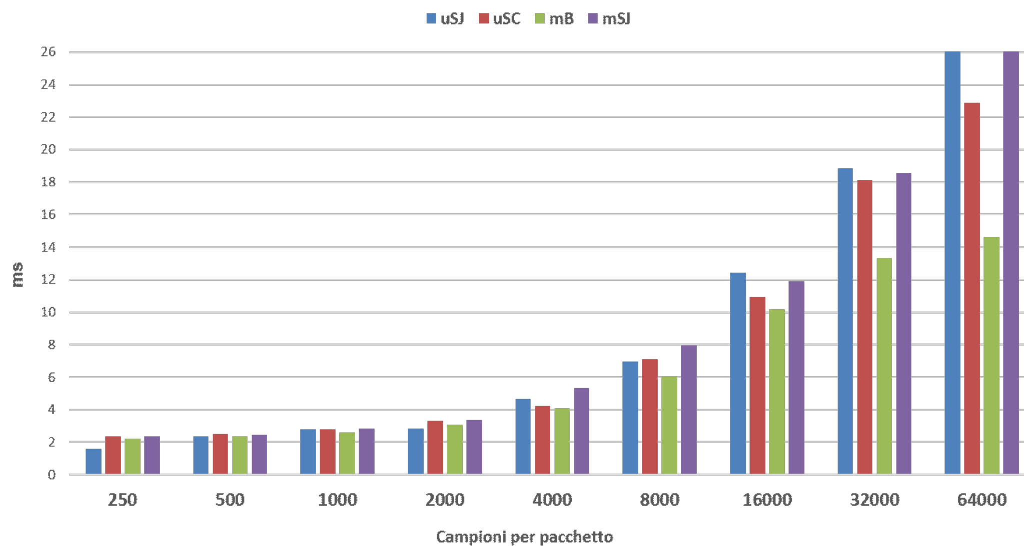


Figura 4.13: Tempi medi di trasmissione su piattaforma docker. La macchina di riferimento è la 2.

dei tempi di trasmissione tra i vari formati, come raffigurato in Fig.4.13. Anche i tempi di decodifica mantengono lo stesso andamento semplicemente scalato nel

tempo, Fig.4.14. Per 64000 campioni il formato μSC passa da quasi 37ms necessari alla decodifica a circa 5,6ms e la differenza rispetto al più rapido mB è di 5,4ms. Lo stesso confronto sulla macchina 1 manifesta una differenza di 37ms. Questo ci dimostra il beneficio ottenuto dallo spostare il carico di lavoro lato Subscriber, visto che dispositivi performanti assottigliano la disuguaglianza temporale dei diversi formati. In modo tale da chiarire come la scaltura temporale tra le due macchine

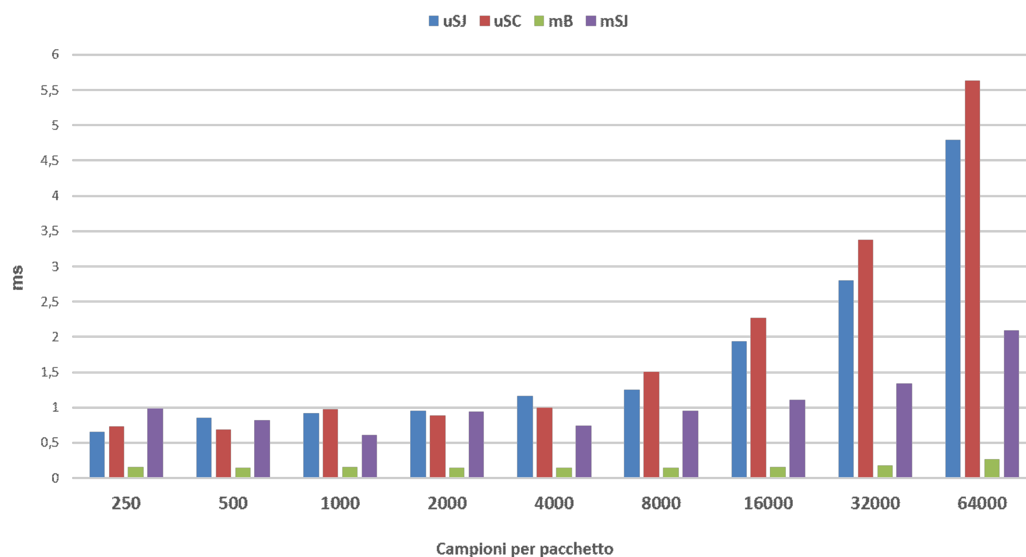


Figura 4.14: Tempi medi di decodifica del subscriber su piattaforma docker. La macchina di riferimento è la 2.

mantenga comunque gli stessi rapporti tra i formati, in Fig.4.15, è stata rappresentata la distribuzione dei tempi medi per un pacchetto contenente 2000 campioni. Per quanto riguarda invece tutto il range di valori, in Fig.4.16, vengono rappresentati i tempi medi totali derivanti dalla somma dei risultati precedenti. Un confronto diretto nei trend dei tempi totali medi delle due macchine fa riscontrare piccole differenze nei rapporti tra formati di pacchetto, bisogna però tenere in considerazione che, nella macchina 2, gli intervalli temporali sono molto più ristretti(anche sotto il ms), questo comporta una difficoltà maggiore nel riportare misurazioni puntuali. Infatti, nell'arco delle 100 simulazioni di scambio di pacchetto per ogni formato dati, è sufficiente che una misura sia inesatta per varie motivazioni che la media complessiva ne risenta, per valori di questi ordini di grandezza.

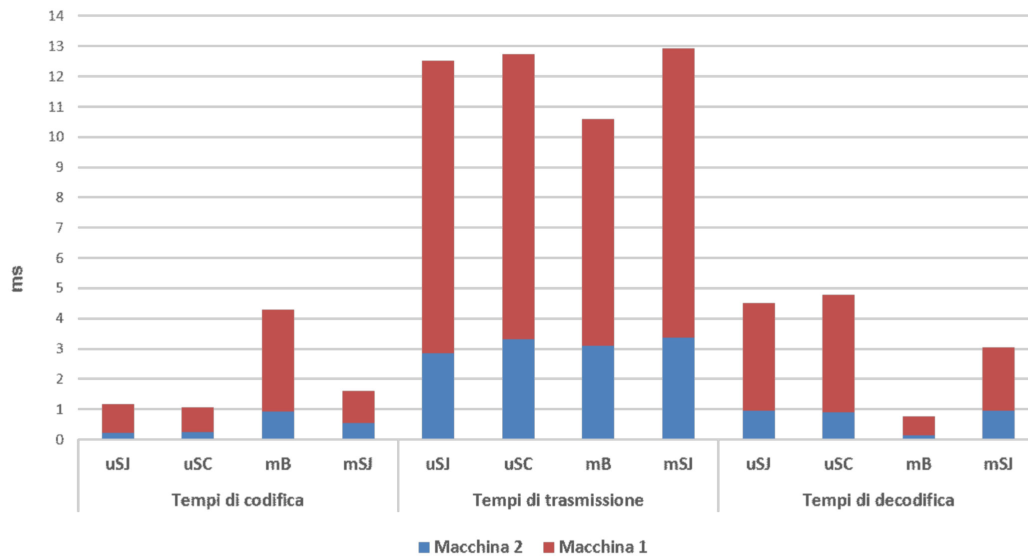


Figura 4.15: Confronto distribuzione dei tempi tra la macchina 1 e 2, nei diversi formati, per un pacchetto da 2000 campioni, su piattaforma docker.

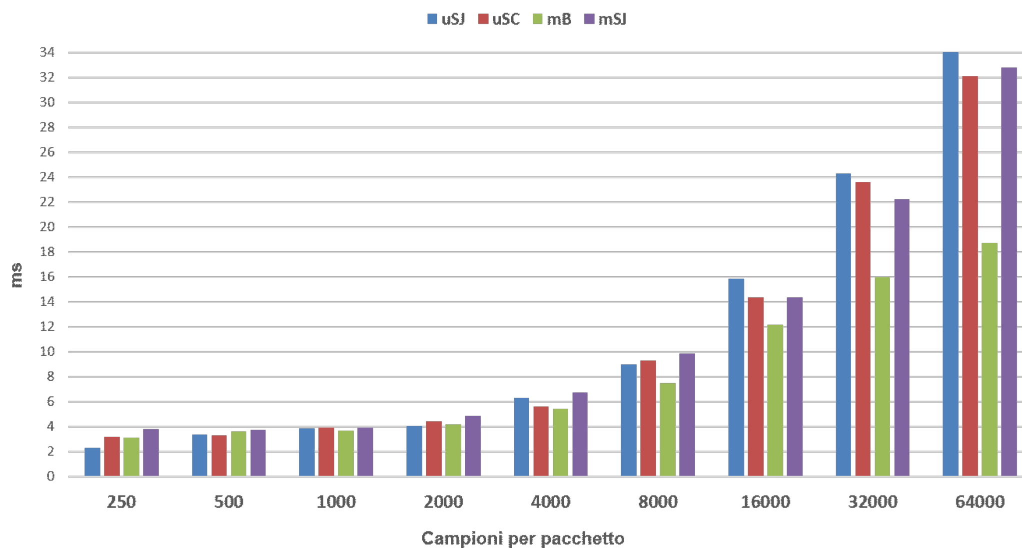


Figura 4.16: Tempi medi totali su piattaforma docker. La macchina di riferimento è la 2.

4.2.4 Simulazioni con Raspberry e client Docker

In questa sezione verranno esposti gli effetti derivanti dall'inseriemento di un dispositivo a limitate capacità computazionali lato Publisher, mantenendo la macchina 2 lato Subscriber. Come è evidente in Fig.4.17, l'intervallo temporale $t_{codifica}$ si è dilatato di un ordine di grandezza rispetto ai risultati ottenuti in fase di codifica sulla macchina 1. Il trend si mantiene analogo con il formato mB, che per un

numero di campioni inferiore a 8000, passa da un tempo di $1ms$ sulla macchina 2 a $10ms$ su Raspberry. Facendo a meno di riportare nuovamente i grafici dei tempi

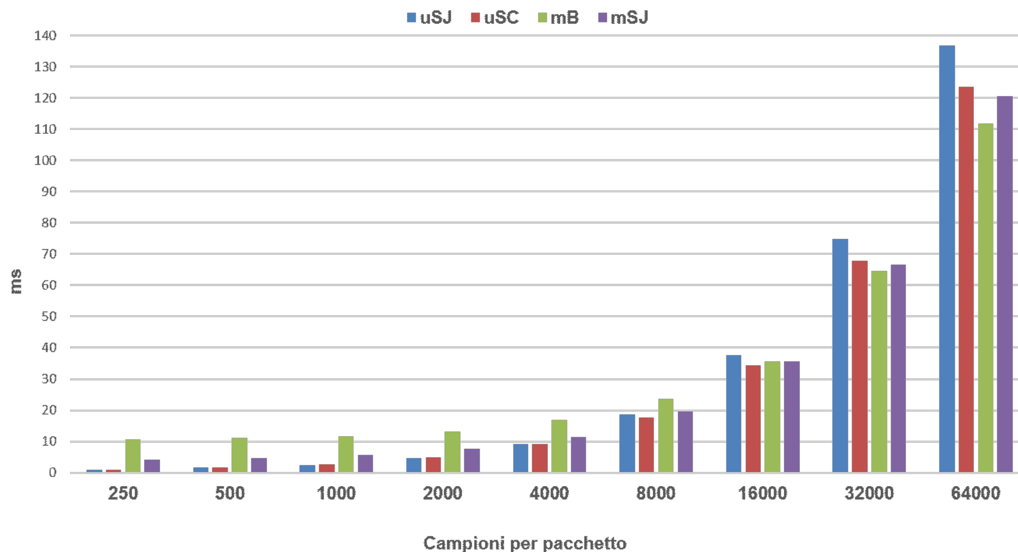


Figura 4.17: Tempi medi di codifica del publisher su Raspberry Pi.

di trasmissione,dipendenti dalla dimensione del pacchetto la quale rimane in questa prova inalterata rispetto alle precedenti e il grafico dei tempi di decodifica, invarianti avendo utilizzato la stessa macchina(2), vengono rappresentati direttamente i tempi medi totali in Fig.4.18. I risultati di questa simulazione, che ha caratteristiche più

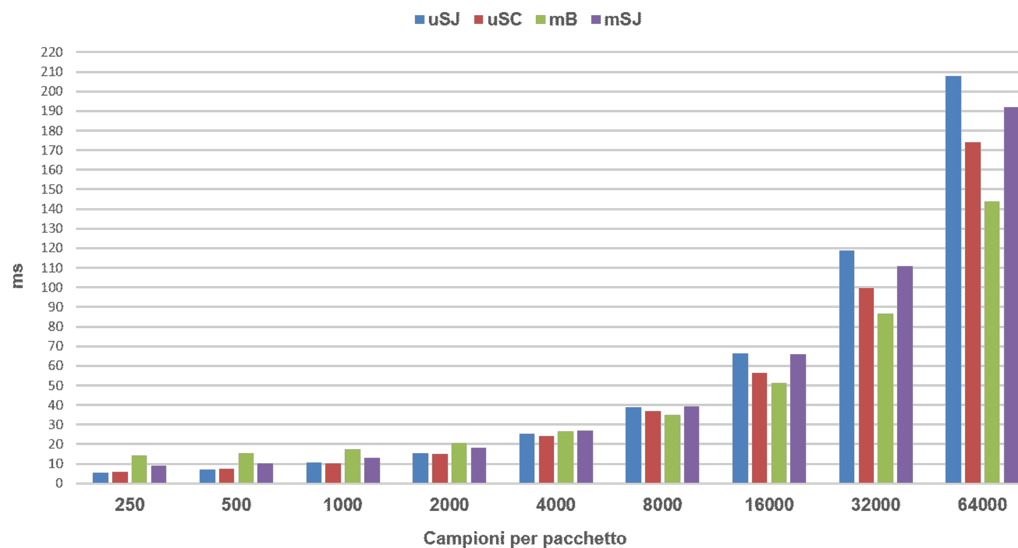


Figura 4.18: Tempi medi totali utilizzando Raspberry Pi come publisher e la macchina 2 come subscriber.

vicine alla rappresentazione di uno scenario reale, mostrano un guadagno in termini di *ms* dei formati che utilizzano μSEED, rispetto a quelli basati su miniSEED, in un range di valori che ha come limite superiore 8000 campioni per pacchetto. Fino ad ora, gli esiti delle simulazioni, con i client sulla stessa macchina, hanno messo in evidenza come il formato mB risultasse la scelta migliore in termini di t_{totali}, indipendentemente dai campioni del pacchetto. In questa simulazione ciò non risulta più corretto in quanto, per pacchetti che contengono fino a 4000 campioni, la codifica con mB evidenzia i tempi peggiori. Questo comportamento è causato da due fattori: espansione dei tempi lato Publisher e inversamente una diminuzione del gap temporale tra i formati in fase di decodifica sul Subscriber (rispetto alla macchina 1). Infatti, per i formati basati su μSEED, il vantaggio che si veniva a creare per pacchetti relativamente piccoli in codifica, veniva completamente perso e superato in decodifica. Non avendo più parità di risorse tra i client, nella fase di decodifica lato Subscriber il gap temporale viene assottigliato, permettento maggiore efficienza globale di questi formati rispetto a quelli che utilizzano miniSEED, fino ad un certo valore di campioni compreso tra i 4000 e gli 8000. Per esplicitare

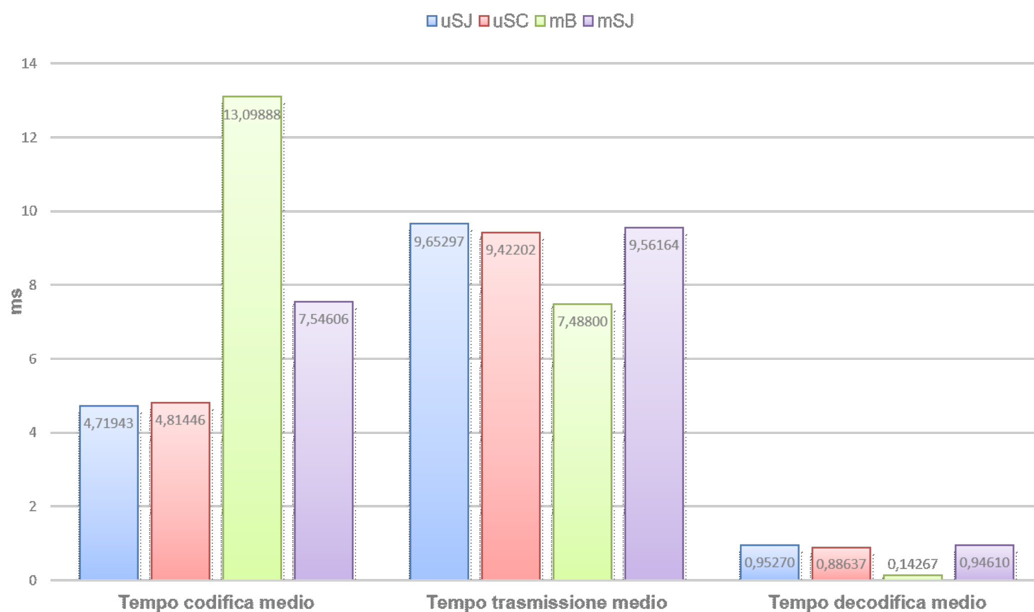


Figura 4.19: Distribuzione dei tempi nei diversi formati, per un pacchetto da 2000 campioni, utilizzando Raspberry Pi come publisher e la macchina 2 come subscriber.

quanto detto, è stata riportata in Fig.4.19 la distribuzione dei tre intervalli temporali per un pacchetto da 2000 campioni. Un fattore poco considerato fin ora è quello riguardante il tempo di trasmissione $t_{trasmissione}$, questo dipende dalle condizioni e dalla tipologia della rete, è chiaro però come il formato mB ci fornisca le dimensioni minori del pacchetto da 250 campioni in su e conseguentemente garantirà in media le minori latenze in trasmissione. Il guadagno temporale del formato μSC rispetto a quello mB è illustrato in Fig.4.20 dove sono considerati: $t_{codifica}$, $t_{decodifica}$ e t_{totale}. Questa rappresentazione ci permette di comprendere la stretta dipendenza di t_{totale} da $t_{codifica}$, tale guadagno si mantiene positivo fino a 4000 campioni passando da $8ms$ a $2ms$, poi riducendosi troppo il vantaggio del $t_{codifica}$, il guadagno sul totale diventa negativo a favore del mB. Con questa configurazione della prova, si sono

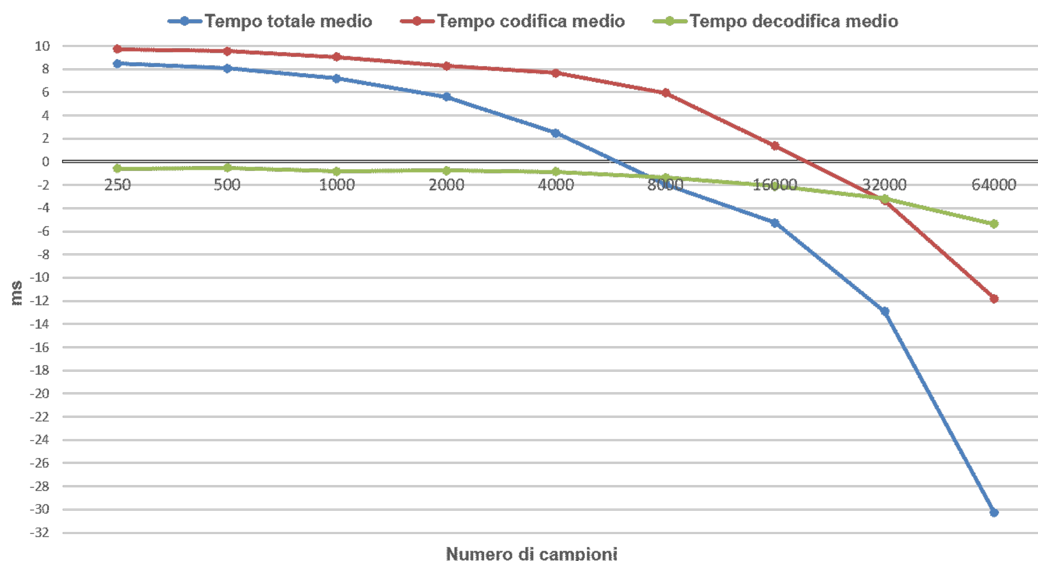


Figura 4.20: Differenza tra il formato mB e μSC in termini di tempi di codifica,decodifica e totali medi. simulazione realizzata utilizzando Raspberry Pi come publisher e la macchina 2 come subscriber.

ottenuti alcuni tempi totali medi perfino migliori di quelli riscontrati con i due client collocati sulla macchina 1. In particolare, sono stati sottratti i tempi medi totali delle simulazioni con Raspberry e macchina 2, a quelli osservati sulla macchina 1. Il risultato è illustrato in Fig.4.21 dove è evidente come la maggior parte dei tempi siano inferiori lavorando in locale sulla macchina 1, fatta eccezione per il numero di campioni del pacchetto di 250 e 500. Questi ultimi due casi sottolineano un vantaggio, in termini di tempo rispetto alla macchina 1, nell'utilizzare il formato

μSEED su un dispositivo a basse capacità computazionali lato Publisher, spostando il carico di lavoro verso una macchina più performante lato Subscriber. Infine, vengono

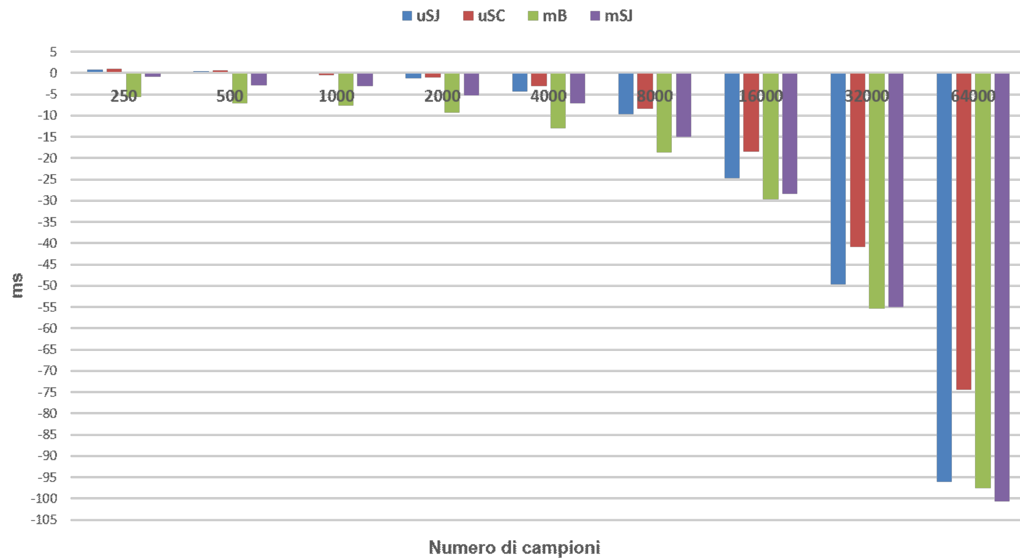


Figura 4.21: Differenza dei tempi totali tra le simulazioni effettuate nella macchina 1 rispetto alla configurazione con Raspberry e macchina 2.

rappresentate le distribuzioni di probabilità dei tempi totali nei successivi due grafici, in cui sono stati considerati i valori estremi del range di misura, vale a dire 250 e 64000 campioni. Nel primo caso in Fig.4.22, si è suddiviso l'asse delle ascisse in bin da 0.2 ms, mentre nel secondo caso in Fig.4.23, la suddivisione è stata effettuata in bin da 2 ms. Nel caso di 250 campioni è ben evidente come i formati μSEED forniscano un t_{totale} minore rispetto agli altri due. Le distribuzioni di probabilità di μSJ e μSC appaiono abbastanza simili e disgiunte da quelle di mB e mSJ. Nel secondo caso ovviamente, le distribuzioni vengono traslate verso tempi decisamente superiori. Il formato mB questa volta mostra dei risultati migliori, immediatamente seguito da μSC. Il formato μSJ, che evidenziava un comportamento leggermente migliore per pacchetti relativamente piccoli, mostra la distribuzione di probablità del t_{totale} peggiore per pacchetti contenenti 64000 campioni.

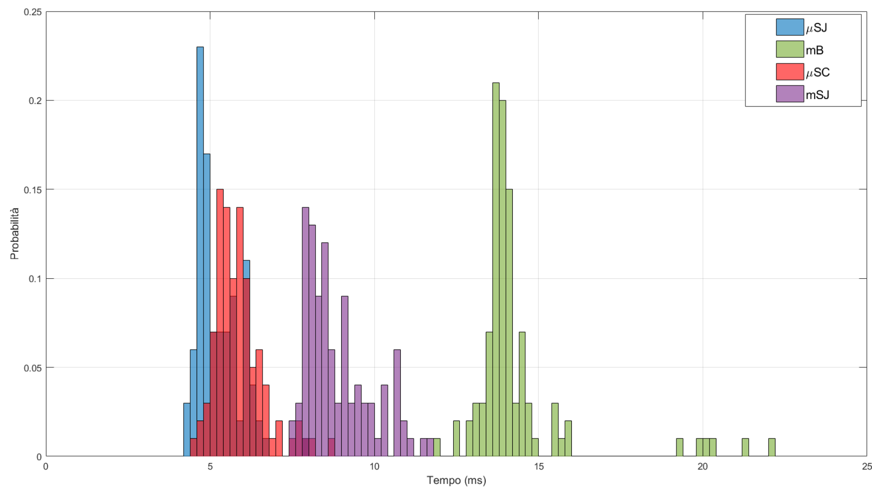


Figura 4.22: Distribuzione di probabilità dei tempi totali per un pacchetto da 250 campioni. Gli istogrammi sono stati normalizzati.

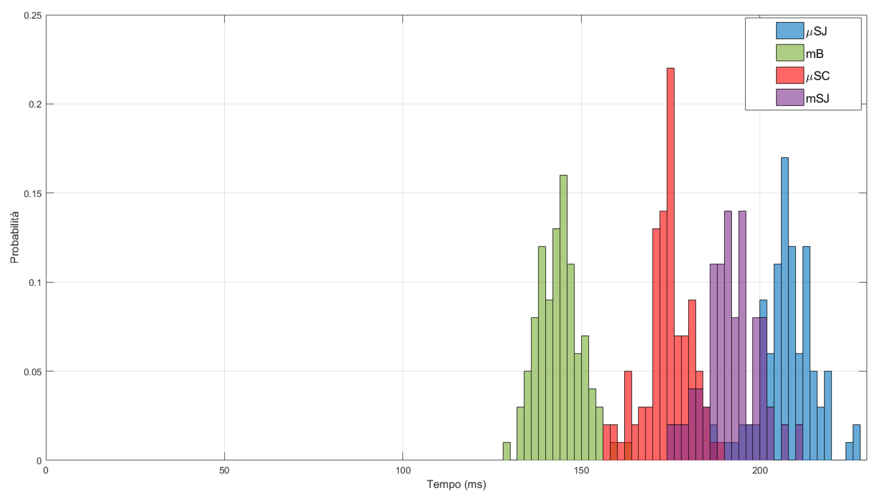


Figura 4.23: Distribuzione di probabilità dei tempi totali per un pacchetto da 64000 campioni. Gli istogrammi sono stati normalizzati.

Capitolo 5

Conclusioni e sviluppi futuri

In questo lavoro di tesi sono stati analizzati protocolli real time attraverso i quali fosse possibile ridurre la latenza temporale nell'individuazione dell'onda P per sistemi di EEW. L'architettura di tali sistemi mostra delle analogie con il mondo IoT e per tale ragione è stato implementato uno dei protocolli più utilizzati in questo ambito, ovvero MQTT, così da confrontarlo con gli standard attualmente in uso. A parità di pacchetti scambiati, il protocollo MQTT rispetto a SeedLink, ha evidenziato un guadagno temporale nelle prove sperimentali di circa 660 ms. Inoltre, come confermato dai risultati, mediante MQTT si ottiene maggiore stabilità sui tempi di consegna del pacchetto. Tali risultati ci hanno spinto a cercare ulteriori guadagni sia in termini di tempo, sia in termini di affidabilità, attraverso l'utilizzo di MQTT. In particolare, sono stati esaminati quattro formati di impacchettamento dei dati, con lo scopo di ottenere quello che si potesse meglio adattare al caso di EEW in esame, valutandoli sotto l'aspetto del tempo di elaborazione e dimensione del pacchetto. I risultati riguardanti le dimensioni mostrano come il formato mB, derivante da miniSEED, riesca a fornire un payload più ristretto a parità di informazioni, per un range di valori compreso tra 250 e 64000 campioni per pacchetto. Il formato μSC espone dimensioni lievemente maggiori, tuttavia rispetto ai formati μSJ e mSJ il guadagno resta notevole. Per quanto riguarda i tempi, si farà riferimento alla prova in cui è stata utilizzata la Raspberry come client Publisher, in quanto reputata più in linea al comportamento riscontrabile in uno scenario reale. In questa simulazione, il tempo totale medio necessario al formato mSJ è risultato sempre tra i peggiori portando con se anche lo svantaggio, sotto il punto di vista implementativo, di essere Python-specific. Il formato mB, risulta conveniente dal punto di vista dei

tempi totali medi solo per pacchetti relativamente grandi, comportando però un maggiore carico computazionale al dispositivo Publisher, che ha limitate risorse rispetto al Subscriber (ordinariamente molto più performante). Tra i due formati che hanno alla base la struttura μSEED i risultati complessivi riguardo le tempistiche e le dimensioni, orientano una eventuale scelta verso quello che si avvale della serializzazione CBOR. Inoltre utilizzando i formati μSEED nell'edge della rete, viene offerta maggiore interoperabilità sul pacchetto anche per quanto riguarda il numero di campioni, infatti il formato miniSEED standard prevede per lo streaming una dimensione fissa di 512 byte e questo può risultare un vincolo. Una volta che il pacchetto viene ricevuto dalla piattaforma di analisi sismica, necessita della conversione illustrata nei precedenti capitoli per poter essere salvato ed esaminato in formato standard miniSEED. Altra peculiarità di questi ultimi riguarda il tema della sicurezza all'interno della rete: è possibile infatti implementare semplicemente schemi di firma sul messaggio a livello applicazione con la finalità di avere certezza del nodo mittente. Nello specifico, mediante la codifica con CBOR si hanno prestazioni migliori rispetto a JSON che necessita di una conversione dei dati preliminare alla generazione della firma. In funzione dei risultati ottenuti e delle considerazioni presentate, in uno scenario IoT dove i dispositivi di misura sono a basso costo(per garantire capillarità sul territorio) e il centro di analisi che fornisce l'allerta dispone di risorse computazionali elevate(magari utilizzando piattaforme Cloud il più possibile vicine ai nodi sensori), è possibile ottenere latenze minori sfruttando il protocollo MQTT supportato da un formato dati μSC. Sarebbe interessante in futuro, affiancare un'architettura IoT che utilizza tali protocolli ad un sistema di EEW reale, in modo tale da riscontrare il guadagno effettivo in termini di latenza che le nostre simulazioni hanno evidenziato.

Bibliografia

[1] Alphonsa A. and Ravi G. Earthquake early warning system by iot using wireless sensor networks. In *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pages 1201–1205, 2016.

[2] Young-Woo Kwon, Jae-Kwang Ahn, Jimin Lee, and Chul-Ho Lee. Earthquake early warning using low-cost mems sensors. In *IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium*, pages 6635–6637, 2020.

[3] Levent Sevgi. Electromagnetic precursors and earthquakes: Nowcasting, forecasting, and prediction [testing ourselves]. 56:319–326, 2014.

[4] Ta liang Teng, Ludan Wu, Tzay-Chyn Shin, Yi-Ben Tsai, and William H. K. Lee. One minute after: Strong-motion map, effective epicenter, and effective magnitude. 87:1209–1219, 1997.

[5] JD Cooper. Letter to editor, san francisco daily evening bulletin. *Nov*, 3:1868, 1868.

[6] R. M. Allen, P. Gasparini, O. Kamigaichi, and M. Bose. The status of earthquake early warning around the world: An introductory overview. 80:682–693, 2009.

[7] Claudio Satriano, Yih-Min Wu, Aldo Zollo, and Hiroo Kanamori. Earthquake early warning: Concepts, methods and physical grounds. 31:106–118, 2011.

[8] Richard M Allen and Diego Melgar. Earthquake early warning: Advances, scientific challenges, and societal needs. *Annual Review of Earth and Planetary Sciences*, 47:361–388, 2019.

[9] John Clinton, Aldo Zollo, Alexandru Marmureanu, Can Zulfikar, and Stefano Parolai. State-of-the art and future of earthquake early warning in the european region. *Bulletin of Earthquake Engineering*, 14(9):2441–2458, 2016.

[10] Claudio Satriano, Luca Elia, Claudio Martino, Maria Lancieri, Aldo Zollo, and Giovanni Iannaccone. Presto, the earthquake early warning system for southern italy: Concepts, capabilities and future perspectives. *Soil Dynamics and Earthquake Engineering*, 31(2):137–153, 2011.

[11] Incorporated Research Institutions for Seismology Chad Trabant. http://www.fdsn.org/pdf/seedmanual_v2.4.pdf. January 2009.

[12] Adam T. Ringler and John R. Evans. A quick seed tutorial. 86:1717–1725, 2015.

[13] IRIS. https://www.seiscomp.de/seiscomp3/doc/applications/seedlink.html.

[14] Antonis Tzounis, Nikolaos Katsoulas, Thomas Bartzanas, and Constantinos Kittas. Internet of things in agriculture, recent advances and future challenges. 164:31–48, 2017.

[15] Farahnaz Sadoughi, Ali Behmanesh, and Nasrin Sayfouri. Internet of things in medicine: A systematic mapping study. 103:103383, 2020.

[16] Eyhab Al-Masri, Karan Raj Kalyanam, John Batts, Jonathan Kim, Sharanjit Singh, Tammy Vo, and Charlotte Yan. Investigating messaging protocols for the internet of things (iot). *IEEE Access*, 8:94880–94911, 2020.

[17] Nitin Naik and Paul Jenkins. Web protocols and challenges of web latency in the web of things. In *2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 845–850, 2016.

[18] Nitin Naik. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http, 2017.

[19] https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2018.pdf. *Eclipse IoT Working Group, AGILE IoT, IEEE, and theOpen Mobile Alliance.*

[20] Edoardo Longo, Alessandro E.C. Redondi, Matteo Cesana, Andrés Arcia-Moret, and Pietro Manzoni. Mqtt-st: a spanning tree protocol for distributed mqtt brokers. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pages 1–6, 2020.

[21] Ravi Kodali and Sreeramya Soratkal. Mqtt based home automation system using esp8266. pages 1–5, 12 2016.

[22] Shinho Lee, Hyeonwoo Kim, Dong-kweon Hong, and Hongtaek Ju. Correlation analysis of mqtt loss and delay according to qos level. In *The International Conference on Information Networking 2013 (ICOIN)*, pages 714–717. IEEE, 2013.

[23] Jevgenijus Toldinas, Borisas Lozinskis, Edgaras Baranauskas, and Algirdas Dobrovolskis. Mqtt quality of service versus energy consumption, 2019.

[24] Biswajeeban Mishra and Attila Kertesz. The use of mqtt in m2m and iot systems: A survey. *IEEE Access*, 8:201071–201086, 2020.

[25] Roger A. Light. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13):265, 2017.

[26] Boci Lin, Yan Chen, Xu Chen, and Yingying Yu. Comparison between json and xml in applications based on ajax. In *2012 International Conference on Computer Science and Service System*, pages 1174–1177, 2012.

[27] Ecma-404the json data interchange syntax, December 2017.

[28] C Bormann and P Hoffman. Rfc 8949 concise binary object representation (cbor). 2020.

[29] Michael Jones, John Bradley, and Nat Sakimura. JSON Web Signature (JWS). RFC 7515, May 2015.

[30] Henrich C. Pohls. Json sensor signatures (jss): End-to-end integrity protection from constrained device to iot application, 2015.

[31] Michael Jones. JSON Web Algorithms (JWA). RFC 7518, May 2015.

[32] Henrich C. Pohls and Benedikt Petschkuhn. Towards compactly encoded signed iot messages, 2017.

[33] Carl Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, January 2015.

[34] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[35] Yannik Behr, John Clinton, Philipp Kästli, Carlo Cauzzi, Roman Racine, and Men-Andrin Meier. Anatomy of an earthquake early warning (eew) alert: Predicting time delays for an end-to-end eew system. *Seismological Research Letters*, 86(3):830–840, 2015.

[36] JM Wassermann, L Krischer, T Megies, R Barsch, and M Beyreuther. Obspy: A python toolbox for seismology. In *AGU Fall Meeting Abstracts*, volume 2013, pages S51A–2322, 2013.

[37] David Mills, Jim Martin, Jack Burbank, and William Kasch. Network time protocol version 4: Protocol and algorithms specification. 2010.

[38] "iris supported software: Ringserver," https://seiscode.iris.washington.edu/projects/ringserver.

[39] Guido Van Rossum. *The Python Library Reference, release 3.8.2*. Python Software Foundation, 2020.

[40] Sokolov Yura Fritz Conrad Grimpen. flunn. https://pypi.python.org/pypi/flunn.

[41] Fritz Conrad Grimpen. flynn. https://github.com/fritz0705/flynn.

[42] Brian Olson. cbor_py. https://github.com/brianolson/cbor_py.

[43] Alex Grönholm. cbor2. https://github.com/agronholm/cbor2.