POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Studio e valutazione di protocolli industriali per l'IoT in scenari Edge/Cloud

Relatore:
Prof. Claudio Zunino
Prof. Gianluca Cena
Prof. Stefano Scanzio

Candidato:
Davide Malena

Dicembre 2021

Ringraziamenti

Vorrei dedicare questo spazio a chi, con dedizione e pazienza, ha contribuito alla realizzazione di questo elaborato. Un ringraziamento particolare va al mio relatore, il professor Claudio Zunino che mi ha seguito, con la sua infinita disponibilità, in ogni step della realizzazione dell'elaborato, fin dalla scelta dell'argomento. Ringrazio i correlatori, il professor Gianluca Cena e il professor Stefano Scanzio per i preziosi consigli e per la disponibilità. Ringrazio i miei genitori per avermi sostenuto in questi anni, con pazienza e diligenza, dandomi il loro supporto. Infine, tutti gli amici che ho conosciuto durante il percorso di studi.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

L'avvento del paradigma Industria 4.0 ha nettamente accresciuto l'importanza dell'interconnettività, della comunicazione da macchina a macchina (M2M) e della capacità di trasmettere dati in tempo reale da e verso il Cloud. In questi contesti è spesso necessario avere latenze di trasmissione molto basse, ad esempio nei sistemi che prevedono la notifica di allarmi, oppure può essere indispensabile la garanzia di una connettività resiliente. Vista l'importanza che, in ambito Industria 4.0, riveste l'interoperabilità, un protocollo che ultimamente sta conoscendo una sempre maggiore diffusione nei sistemi reali è OPC UA (Open Platform Communications Unified Architecture) [1], a dispetto del fatto che il suo primo rilascio avvenne nel 2006. Nonostante prima della versione attuale ci furono altre versioni di OPC, che introdussero novità come la gestione di dati, eventi ed allarmi, è solo con OPC UA che è stata realizzata un'adeguata indipendenza dalla piattaforma, facilitando così lo scambio di dati anche negli scenari industriali eterogenei.

Un secondo protocollo molto usato nel campo dell'IoT, che sta trovando impiego anche nell'IoT industriale (IIoT), è MQTT (precedentemente conosciuto come Message Queue Telemetry Transport) [2], un protocollo semplice, che necessità di poca banda per la comunicazione ed inoltre prevede opportuni meccanismi per assicurare la corretta ricezione dei dati. MQTT si basa sul modello Publish/Subscribe [3], in cui i Client si connettono ad entità chiamate broker che si occupano di mediare la comunicazione fra di essi, ricevendo e inoltrando i dati in modo opportuno. Grazie a questo protocollo è possibile scambiare facilmente i dati tra macchine (anche dotate di limitate risorse computazionali) che usano sistemi operativi diversi. Di recente, OPC UA ha rilasciato una nuova specifica che permette di usare il protocollo anche in modalità Publish/Subscribe. Questo ha permesso di combinare la ricchezza delle strutture dati di OPC UA con il meccanismo di comunicazione di MQTT, in modo da sfruttare la sua semplicità ed il basso consumo di banda (tale opzione è esplicitamente prevista tra le varie possibilità nelle specifiche di OPC UA). Alla luce dell'importanza che l'IIoT riveste nei moderni impianti industriali, l'obiettivo principale di questa tesi è quindi valutare alcuni aspetti di tali protocolli. Dopo una descrizione delle loro caratteristiche salienti, che influiscono tanto sulle funzionalità quanto sulle prestazioni, vengono presentate due valutazioni sperimentali di prestazioni, nelle quali la metrica analizzata è stata la latenza introdotta dalla comunicazione in rete. La prima riguarda il protocollo OPC UA basato sul paradigma Publish/Subscribe, nel quale il trasporto dei dati avviene per mezzo di MQTT. La seconda è stata invece condotta sul protocollo OPC

UA nativo, e ha riguardato la misura e l'analisi dei tempi di notifica di tre differenti implementazioni di Server: Unified Automation [4], Free OPC-UA [5] ed open62541 [6]. Inoltre, nell'ambito della tesi è stato definito un approccio volto a migliorare la resilienza del protocollo MQTT, cui ha fatto seguito una sua implementazione prototipale. In particolare, è stata realizzata un'architettura a percorsi ridondati basata su MQTT, concettualmente simile a PRP (Parallel Redundancy Protocol) [7]. Anche di questa implementazione si sono valutate le prestazioni in termini di latenza, e se ne è verificata praticamente la capacità di fare fronte a guasti in grado di compromettere le funzionalità di parte dell'architettura di connessione sviluppando un prototipo.

Capitolo 2

OPC UA

2.1 Introduzione

Open Platform Communications Unified Architecture venne realizzato da un'organizzazione chiamata OPC Foundation [8]. Questa nacque con lo scopo di definire uno standard di comunicazione tra i vari dispositivi in ambito industriale, ma prima di OPC UA ci furono altre versioni, la prima fu *OPC Data Access* [9] rilasciata nel 1996 [10], tuttavia questa prima versione si basava sull'interfaccia COM/DCOM dei sistemi Windows. Durante gli anni, OPC Foundation continuò a lavorare fino ad arrivare nel 2006 alla realizzazione di OPC UA, che introdusse la novità di essere platform-independent, permettendo così di staccarsi dall'uso di COM/DCOM e potendo realizzare una comunicazione che fosse indipendente dal sistema operativo usato. OPC Foundation, per descrivere nel dettaglio il funzionamento e i possibili utilizzi, ha rilasciato una specifica che negli anni viene aggiornata con eventuali

correzioni o nuove funzionalità: attualmente è divisa in sedici parti principali [11] e ogni parte approfondisce una specifica di OPC UA. In Figura 2.1 sono mostrate le sedici parti più importanti pubblicate finora.

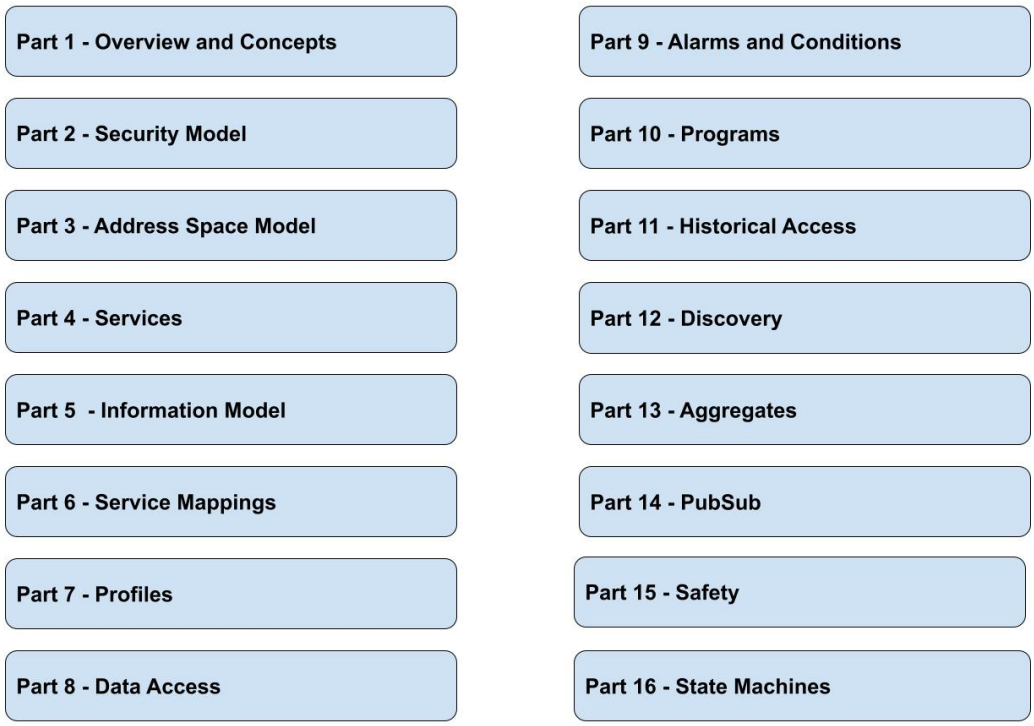

Figura 2.1. Elenco delle principali parti in cui sono suddivise le specifiche di OPC UA

2.2 Client-Server

OPC UA è applicabile in vari campi dell'industria, come sistemi di controllo, IoT, M2M e altro, offrendo inoltre un modello per scambiare facilmente informazioni tra i dispositivi che usano OPC UA. La comunicazione può

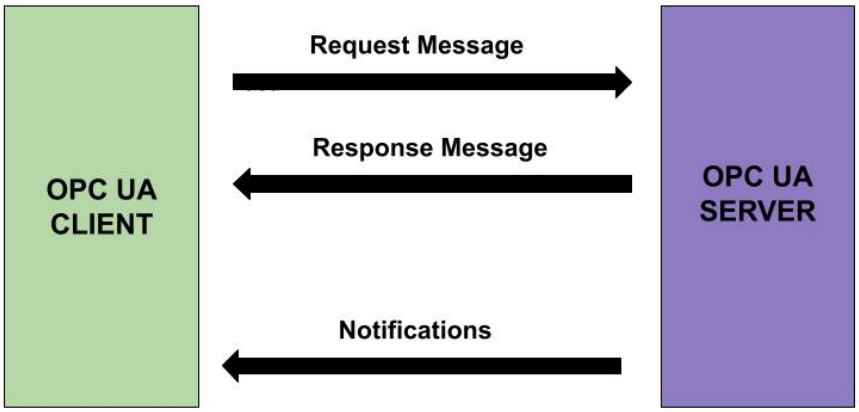

Figura 2.2. Interazione Client-Server OPC UA

avvenire usando una classica architettura Client-Server in cui si hanno il Client OPC UA e il Server OPC UA. In questo modello, la comunicazione Client-Server avviene tramite lo scambio di due messaggi: avremo prima di tutto il Client, che invierà il pacchetto *Request Message* contenente la richiesta di ciò che vuole dal Server e, dall'altra parte, dopo aver elaborato la richiesta, il Server che risponderà con *Response Message* che conterrà lo stato della richiesta, Figura 2.2. Ogni Server OPC UA mette a disposizione per i Client un insieme di informazioni ben strutturato, chiamato *AddressSpace*, che è composto da nodi connessi tramite *Reference*, cioè puntatori. I Client possono accedere sia ai dati correnti, sia a quelli storici; alcuni dei servizi messi a disposizione dal Server sono allarmi ed eventi, usati per far sapere al Client quando un certo cambiamento si verifica all'interno dell'*AddressSpace*:

infatti, non appena avviene un cambiamento in un determinato insieme di dati, verrà inviata una notifica al Client che ne ha fatto richiesta. OPC UA permette di esporre i dati in diversi formati, tra cui XML, JSON e binario.

2.3 AddressSpace

L'*AddressSpace*, Figura 2.3, è composto da nodi, Figura 2.4, e ogni nodo appartiene ad un *NodeClass*, tra i principali ci sono *Variable*, *Method* e *Object*. I Client possono far richiesta di sottoscrizione per ricevere notifiche in caso

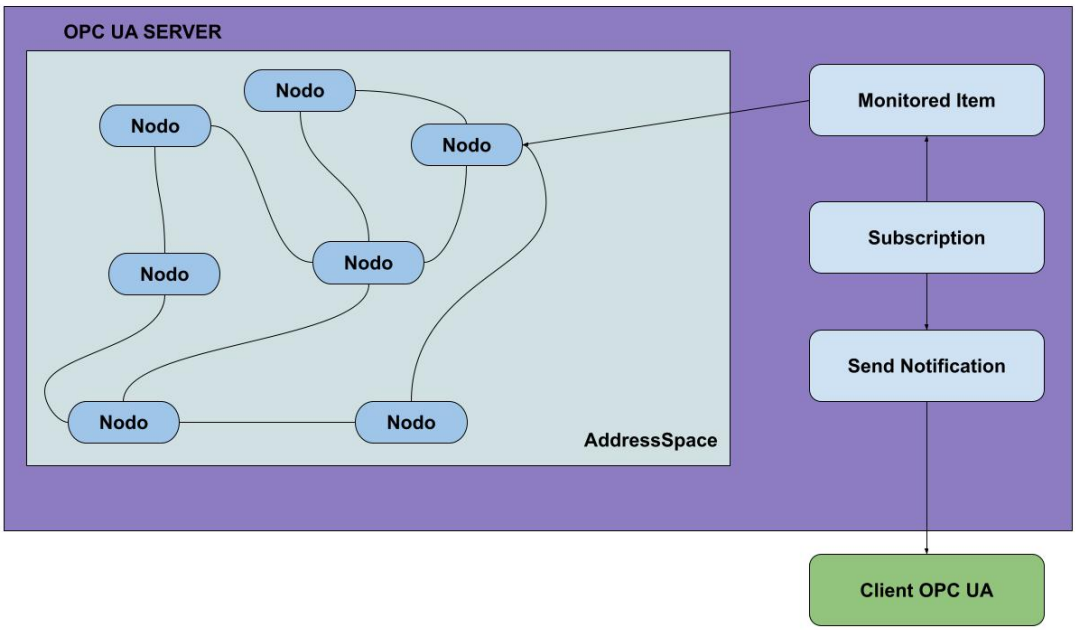

Figura 2.3. Descrizione di un AddressSpace OPC UA

qualche dato di loro interesse cambiasse o nel caso si verificassero particolari condizioni all'interno del Server. Una volta approvata la richiesta, il Server creerà un *MonitoredItem* al suo interno per controllare periodicamente

lo stato della condizione richiesta. I *MonitoredItem* sono entità all'interno del Server, create su richiesta da un Client che vuole monitorare dei nodi nell'*AddressSpace*. Non appena un cambiamento nel nodo osservato è rilevato, verrà inviata dal Server una notifica al Client. Il servizio che si occupa di inviare le notifiche è il *Publish Service*. Le interazioni non sono limitate tra Client e Server ma sono possibili anche tra Server e Server.

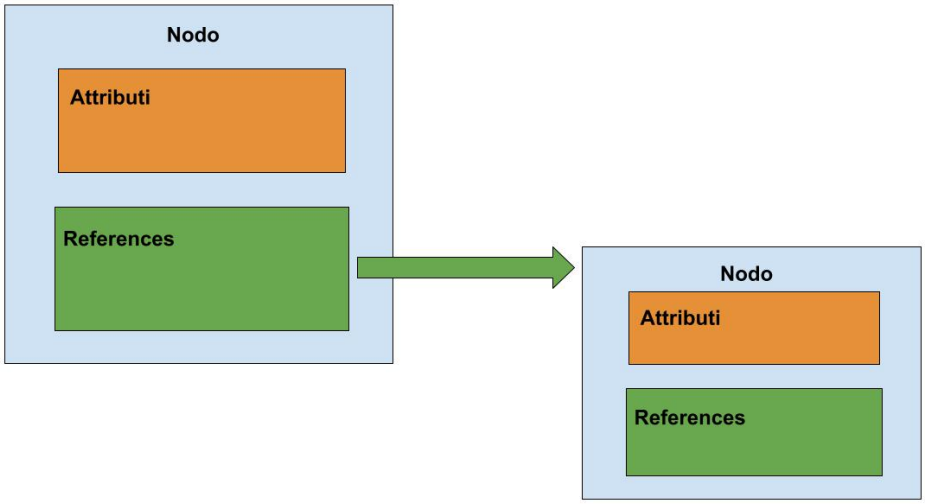

Figura 2.4. Descrizion di un Nodo OPC UA

2.4 Publish/Subscribe

Di recente è stato aggiunta la specifica numero 14 - Publish/Subscribe [12] alla documentazione, aprendo così nuovi campi di applicazione. Il sistema Publish/Subscribe non è legato a un particolare sistema di messaggi, ad esempio, si potrebbe usare UDP multicast codificando i messaggi in binario,

per quando si vogliono mandare dati di dimensioni ridotte, ma con una frequenza elevata, oppure, usando protocolli noti come AMQP o MQTT e codificando i messaggi con JSON. Tuttavia, a prescindere da come avviene la codifica, i termini e gli schemi usati di seguito valgono per una visione più astratta del meccanismo di funzionamento, non vincolata dalla codifica usata e dal *Message Oriented Middleware*. Con il modello Publish/Subscribe non occorre uno scambio diretto tra Client e Server, come con i messaggi *Request Message* e *Response Message*; si utilizza invece un meccanismo in cui i messaggi vengono inviati dal publisher al *Message Oriented Middleware*, che, a sua volta, li inoltrerà ai subscriber. Il *Message Oriented Middleware* è un software o infrastruttura hardware che supporta l'invio e la ricezione dei messaggi tra sistemi distribuiti. OPC UA Publish/Subscribe supporta 2 differenti tipi:

- **Broker-less**, il *Message Oriented Middleware* è un'infrastruttura di rete che è in grado di inoltrare messaggi, tra publisher e subscriber, usando protocolli datagram come UDP multicast.
- **Broker-based**, in questo caso, il *Message Oriented Middleware* è un broker, i publisher e subscriber possono usare protocolli come AMQP o MQTT per comunicare tramite il broker.

I messaggi che vengono spediti dai publisher e ricevuti dai subscriber vengono chiamati *NetworkMessage*, essi sono formati da un insieme di pacchetti, *DataSetMessage* che vanno a comporre il Payload, Figura 2.5. Ogni *DataSetMessage* è creato partendo da un *DataSet*, e la parte che si occupa di creare i *DataSetMessage* è chiamata *DataSetWriter*. Lo stesso publisher può

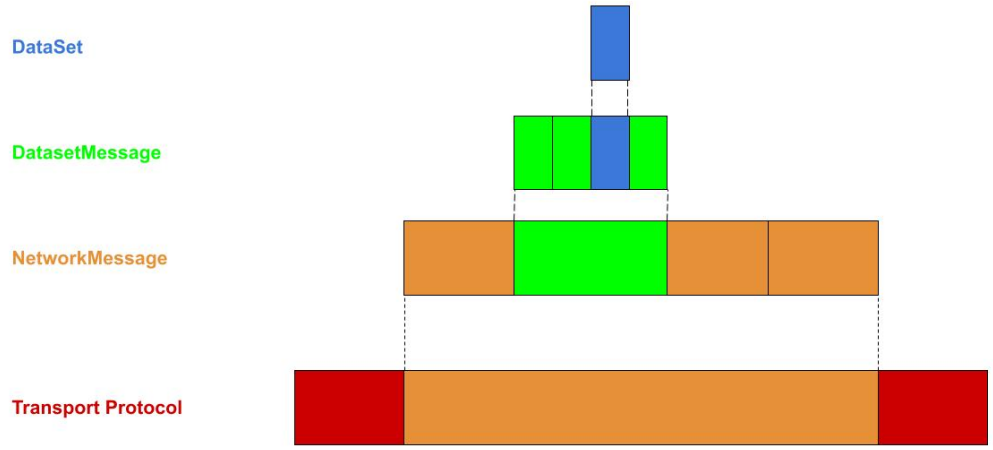

Figura 2.5. Incapsulamento del NetworkMessage

avere più di un *DataSetWriter* che crea i suoi *DataSetMessage* dai *DataSet*, Figura 2.6. I *DataSet* possono essere eventi o liste di variabili che verranno poi codificati in un *DataSetMessage*, che insieme ad altri *DataSetMessage* andranno a comporre il *NetworkMessage*. La parte di sicurezza e codifica del *DataSet* è svolta dal *DataSetMessage* e alcune impostazioni dipendono dal *Message Oriented Middleware* usato.

I subscriber usano i *DataSetMeta* per decodificare i valori contenuti nei *DataSetMessage*, i *DataSetMeta* sono ricevuti tramite un *NetworkMessage*. Solitamente i publisher sono quelli che forniscono le informazioni, mentre i subscriber quelli che ne usufruiscono, quindi lo schema solito è Server OPC UA nel ruolo di publisher e Client OPC UA come subscriber, tuttavia, si può avere anche il contrario, non si hanno vincoli al riguardo. I subscriber

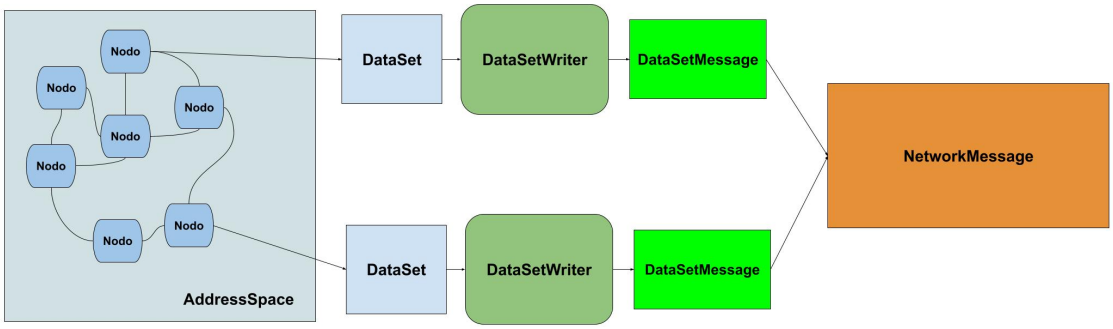

Figura 2.6. Sequenza per l'invio del NetworkMessage

ricevono dal *Message Oriented Middleware* i *NetworkMessage*, e se un *DataSetMessage* di quel pacchetto è di suo interesse, allora verrà inoltrato al corrispondente *DataSetReader*, che si occuperà di decodificarlo in un *DataSet*. Il subscriber può anche impostare un filtro in modo da far passare solo *NetworkMessage* che soddisfino alcuni requisiti, andando a leggere il solo *header*, Figura 2.7

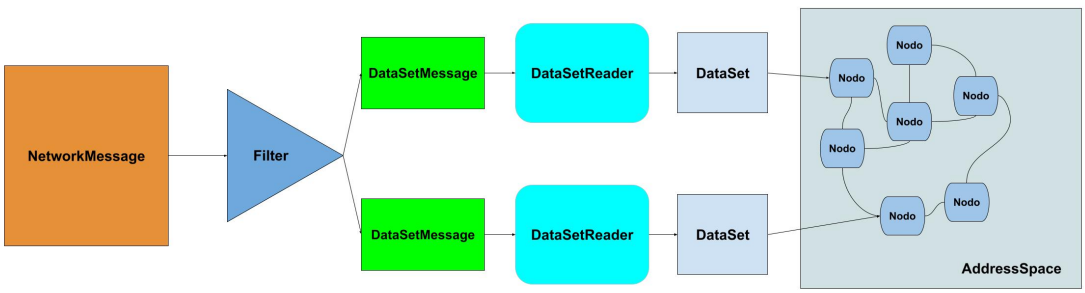

Figura 2.7. Sequenza per la ricezione del NetworkMessage

2.5 Librerie di OPC UA

Esistono diverse librerie di OPC UA, realizzate in vari linguaggi, per esempio C, C++, Python, Rust, Java. Nel linguaggio C/C++ è stata implementata la libreria che prende il nome di open62541, la cui ultima versione stabile, cioè la 1.2, è stata rilasciata nel febbraio del 2021. Ci sono altre librerie scritte in C/C++, ma open62541 risulta essere, oltre che la più conosciuta, anche ben sviluppata e molto usata nell'ambito della ricerca. Tra le altre librerie: node-opcua [13] scritto in JavaScript, mentre, in Python, la libreria denominata Free OPC-UA. Un'altra libreria, scritta in linguaggio C, ma che a differenza di open62541 non è open-source è quella della Unified Automation.

Capitolo 3

MQTT

3.1 Introduzione

Il protocollo MQTT (MQ Telemetry Transport) è stato inventato, nel 1999, da Andy Stanford-Clark e Arlen Nipper, per avere un protocollo che consumasse poca batteria e poca larghezza di banda. Il protocollo si basa su connessioni TCP/IP e sul pattern Publish/Subscribe, fornendo un'alternativa all'architettura tradizionale Client-Server, nel quale il Client comunica direttamente con il Server. Nel modello Publish/Subscribe, publisher e subscriber non sono mai connessi direttamente tra di loro, il collegamento fra i due è realizzato e gestito da un altro componente, il broker. In MQTT, per Client, si intendono tutti quei dispositivi che usano il protocollo MQTT e si connettono al broker, potendo essere publisher, subscriber o entrambi. Il principale compito del broker è quello di inoltrare i messaggi ricevuti dai publisher e distribuirli a tutti i subscriber che ne hanno fatto richiesta

in precedenza: quando un publisher manderà un messaggio al broker, sarà quest'ultimo che dovrà inoltrare il messaggio a tutti i subscriber interessati, così che il publisher non dovrà perdere tempo a spedire nuovamente lo stesso messaggio, in caso di più Client, come avverrebbe in un'architettura Client/Server. Uno svantaggio del modello Publish/Subscribe è che non si può sapere quanti subscriber interessati a quel messaggio siano collegati al broker. Per quanto riguarda il filtraggio dei messaggi, il broker sa già a chi mandare i messaggi grazie al topic. Ogni messaggio in MQTT ha il campo topic, che consiste in una stringa con struttura gerarchica, dove ogni livello è suddiviso da "/". Il subscriber, dopo essersi connesso al broker, dovrà dire su quali topic vorrà ricevere i messaggi. In questo modo, il broker, quando riceverà un messaggio da qualsiasi publisher, che contenga quel determinato topic, lo inoltrerà a tutti i subscriber che ne hanno fatto richiesta. È quindi chiaro che il broker gioca un ruolo chiave nella consegna e filtraggio dei messaggi, infatti, in caso di guasti al broker, non sarebbe più possibile consegnare messaggi. Una possibile soluzione a questo problema è usare i cluster broker [14], un insieme di broker che si comportano come fossero uno solo, in cui il carico di lavoro dei messaggi da inoltrare viene distribuito tra i vari broker e, in caso di malfunzionamento di un broker, ci sarebbero gli altri. Tuttavia, se nel momento del guasto ci fossero dei pacchetti già ricevuti da un broker e non ancora inoltrati, questi andrebbero persi, anche se il sistema continuerebbe a funzionare, visto che il traffico verrebbe spostato su un altro nodo del cluster. Un altro difetto è la mole di dati scambiati tra i broker del cluster per la gestione del sistema.

Da notare che nel modello Publish/Subscribe i Client non sanno se l'altro è connesso, quindi può capitare che un publisher mandi messaggi con un determinato topic, ma che non ci sia nessun subscriber che abbia fatto la sottoscrizione a quel topic, pronto a riceverlo. Stesso discorso dal lato dei subscriber, in cui si potrebbe fare la sottoscrizione a un topic, senza sapere se e quando qualcuno manderà dei messaggi sul quel topic.

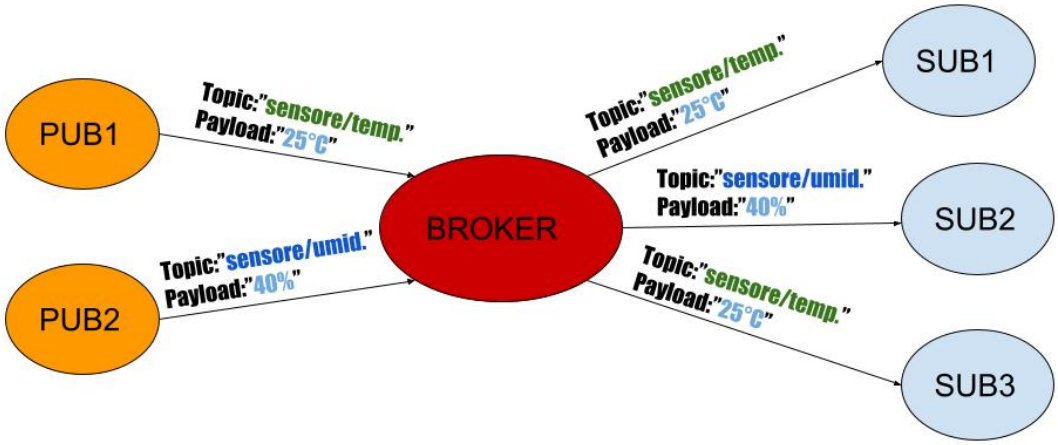

Figura 3.1. Schema di base del funzionamento di MQTT

Per dare un livello di affidabilità ai messaggi, in MQTT ci sono 3 livelli di qualità del servizio, detto QoS (Quality of Service). Ogni messaggio spedito ha un livello di QoS che può essere 0, 1 o 2. La scelta del QoS dipende sia dal publisher, sia dal subscriber. Tuttavia, se un publisher spedisce un messaggio con un certo livello, il subscriber non potrà fare una sottoscrizione con un livello maggiore, ma solo minore o uguale, quindi, in caso un publisher inviasse messaggi con QoS pari a zero e un subscriber abbia una sottoscrizione con QoS pari a 2, quest'ultima verrà gestita come un QoS con

valore 0.

Nella Figura 3.1, è rappresentato uno schema del funzionamento di MQTT con 2 publisher, 3 subscriber e 2 topic: “sensore/temp.”, “sensore/umid.”. Ciascun Client dovrà prima di tutto connettersi al broker, dopodichè, i subscriber si registreranno ai topic di loro interesse: in questo caso sub1 e sub3 faranno richiesta di iscriversi al topic “sensore/temp.”, mentre sub2 al topic “sensore/umid.”. A questo punto i publisher potranno spedire i messaggi, pub1 manderà il messaggio con topic “sensore/temp.” al broker, che lo inoltrerà a tutti i subscriber di quel topic, cioè, sub1 e sub3. Discorso analogo per pub2 e sub2.

3.2 Broker

Il broker è la parte fondamentale nel protocollo MQTT e per collegarsi, nel caso in cui sia abilitata l’autenticazione, è necessario fornire le credenziali di accesso. Nella configurazione base è invece sufficiente conoscere indirizzo IP e porta TCP. Per iniziare la connessione con il broker, il Client dovrà mandare, prima di tutto, la richiesta *Connect Command*, dopodiché, il broker risponderà con lo stato della richiesta. Una volta stabilita, la connessione, se andata a buon fine, rimarrà aperta finché non ci sarà una richiesta di disconnessione o un problema di comunicazione. Ogni Client che si collega al broker avrà un *ClientId* per essere identificato univocamente tra i Client del broker. In Figura 3.2, si vede lo scambio di messaggi che avviene tra broker, publisher e subscriber per poter inviare un messaggio con QoS=0 al subscriber.

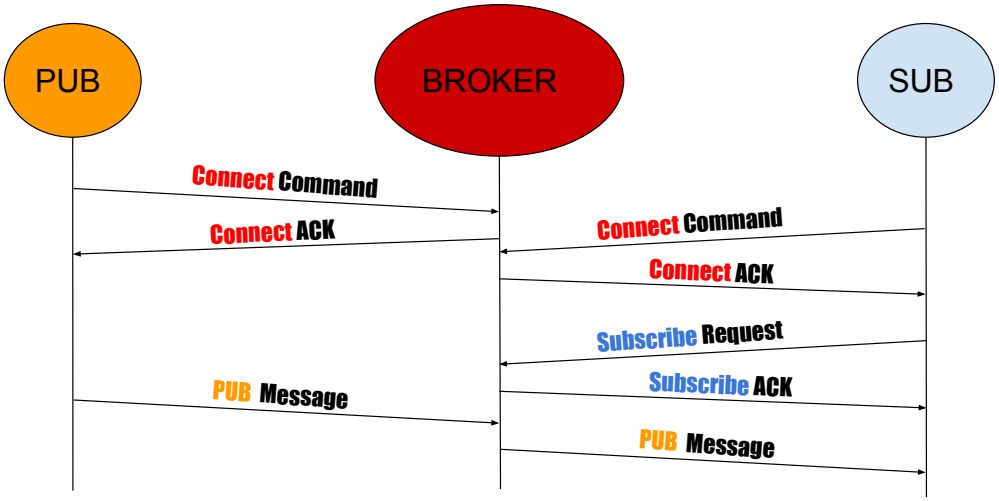

Figura 3.2. Esempio di interazione e messaggi scambiati con QoS=0.

Per prima cosa, ogni Client dovrà fare richiesta di connessione specificando le impostazioni volute nel messaggio *Connect Command*, una di queste impostazioni è il flag *Clean Session*, che di default è su *True*. Se settato su *False*, il broker dovrà memorizzare tutte le sottoscrizioni che il Client farà e tutti i messaggi destinati ad esso con QoS uguale a 1 o 2 che non sono ancora stati consegnati. Se invece il flag è impostato su *True*, allora il broker non memorizzerà nulla, e cancellerà tutte le informazioni riguardanti quel Client una volta disconnesso. Sempre in *Connect Command*, è previsto l'inserimento, se necessari, di username e password: in questo caso si consiglia di criptare il messaggio tramite TLS o con altri protocolli di sicurezza, in quanto sarebbero dati in chiaro.

Keep alive è un valore presente in *Connect Command*, espresso in secondi, che determina dopo quanto tempo Client e broker si scambieranno un Ping per controllare lo stato della connessione.

Una volta spedito il messaggio *Connect Command* con le impostazioni volute, il broker risponderà con *Connect ACK*, comunicando lo stato della richiesta, che può essere andata a buon fine oppure no. Ad esempio, può capitare, in caso di broker con credenziali, password o username errati e quindi nel *Connect ACK* risulterà che la connessione non è andata a buon fine per questo errore.
Esistono diverse implementazioni di broker, realizzati in diversi linguaggi di programmazione, alcuni sono open-source, mentre altri sono commerciali.

3.3 Publisher e Subscriber

Una volta stabilita la connessione, il publisher potrà spedire messaggi. Ogni pacchetto conterrà: PacketId, topic, QoS, RetainFlag, Dupflag e Payload. Come mostrato in un esempio nella Figura 3.3.

- **TopicName** è una stringa del tipo *Topic/Sottotopic2/Sottotopic3*, con lo slash che indica la sottocategoria del Topic, così da avere una gerarchia.
- **QoS** è un numero che indica il livello di qualità del servizio del messaggio.
- **Retainflag** definisce se il messaggio dovrà essere salvato dal broker, come l'ultimo messaggio per quel topic.
- **Payload** Il contenuto del messaggio, che può essere al massimo 268.435.455 bytes.

- **PacketId** Identificativo del messaggio univoco per Client, in modo da identificare i vari messaggi di un Client con QoS maggiore di 0. In caso di QoS pari a zero questo campo vale 0.
- **Dupflag** per identificare i messaggi duplicati, cioè i messaggi che sono stati rispediti in quanto non è stato ricevuto ACK di conferma ricezione. Il publisher rispedisce quello specifico messaggio, dato per perso, dopo un timeout (usato solo se QoS è maggiore di 0).

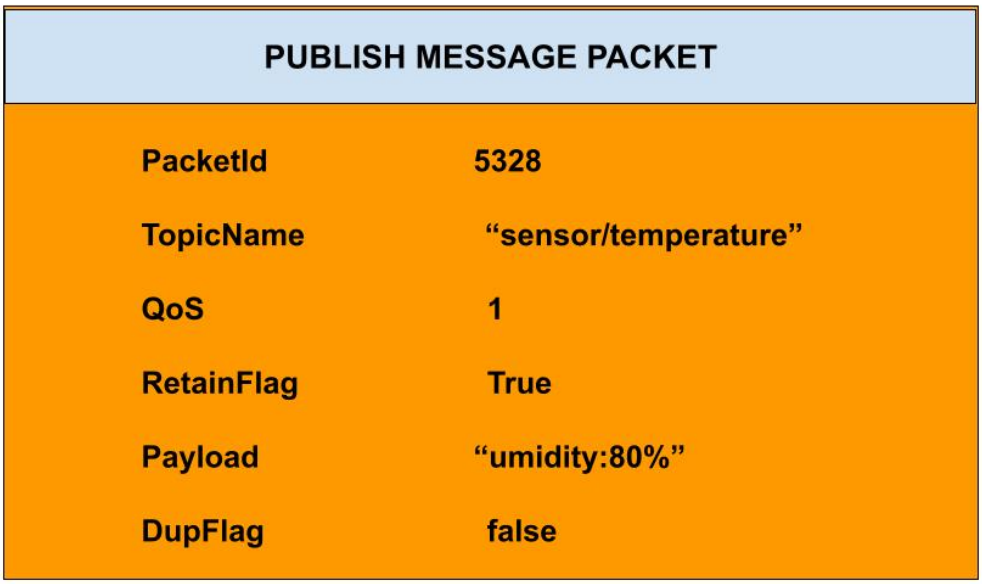

Figura 3.3. Esempio pacchetto contenente un messaggio Publish

Una volta spedito il messaggio dal publisher, e ricevuto dal broker, sarà compito di quest'ultimo consegnarlo ai subscriber. Il publisher, in base al QoS scelto, potrà solo sapere, tramite ACK, se il messaggio è giunto al broker, ma non saprà se il messaggio è arrivato a tutti i subscriber e a quanti. Un publisher senza nemmeno un subscriber non ha senso, in quanto, non

ci sarebbe nessun subscriber pronto a ricevere il messaggio, quindi è logico attendersi che ci sia almeno un Client che faccia la sottoscrizione per quel topic. Per effettuare la richiesta di sottoscrizione al topic, bisognerà mandare il messaggio *Subscribe Request*. Per notificare una conferma dell'avvenuta sottoscrizione, il broker risponderà con *Subscribe ACK*, che conterrà lo stato della richiesta con il livello di QoS. Da questo momento il subscriber riceverà tutti i messaggi spediti su quel topic da qualunque publisher connesso al broker. È inoltre possibile fare una richiesta di *Unsubscribe* in cui si specifica da quali dei topic si vuole cancellare la propria sottoscrizione. Si riceverà così un *UNSUBACK*, contenente solo il packetId del *UNSUBACK*.

3.4 Topic

In MQTT il topic è una stringa UTF-8 usata per categorizzare i messaggi. Il topic è costituito da livelli separati dal simbolo "/", il Client non deve creare il topic, ma semplicemente usarlo. Il broker accetterà ogni tipo di stringa come topic che contenga almeno un carattere che non sia lo spazio, il topic è anche case-sensitive. Per i subscriber ci sono alcuni caratteri speciali da usare (wildcards), ad esempio, un subscriber può iscriversi esattamente a un topic *Topic1/sottotopic*, oppure può iscriversi a più topic simultaneamente. Infatti, con il simbolo "+" si prendono tutti i topic di quel singolo livello. Prendiamo in considerazione il seguente scenario, con un broker, che al suo interno ha i seguenti topic che stanno venendo usati dai Client:

1. *Casa/soggiorno/temperatura*

2. *Topic1/garage/topic2*

3. *Topic1/cucina/topic2*

4. *Topic1/bagno/boiler*

Se il subscriber facesse la sottoscrizione con *Topic1/+/topic2*, il broker dovrà inoltrargli i messaggi che abbiano come topic il 2 e 3. Se invece fosse interessato a più livelli di topic, potrebbe usare il simbolo "#", ma solo alla fine del topic, in modo da comprendere tutti i possibili sottolivelli da lì in avanti. Ad esempio, con *Topic1/#* la sottoscrizione comprenderebbe il topic 2, 3 e 4. Volendo è anche possibile sottoscriversi a tutti i topic presenti usando */#* in modo da ricevere tutti i messaggi che transitano nel broker. Esistono dei topic che iniziano con */$*, questi sono topic particolari che non vengono presi in considerazione quando si usa il simbolo "#", sono riservati per statistiche interne del broker, i Client non possono pubblicare messaggi con questo tipo di topic, però possono sottoscriversi e ricevere le statistiche del broker usando "SYS".

3.5 QoS

I Client hanno la possibilità di scegliere un livello di affidabilità di consegna del messaggio: possono scegliere di non dare importanza alla consegna, di essere sicuri che quel messaggio arrivi a destinazione, eventualmente inviando copie dello stesso, oppure, di volere che arrivi esattamente una sola volta. Ci sono 3 livelli di affidabilità (QoS), dal più basso partendo da zero, e più veloce, al più alto, ma più lento.

- (0) Invio solamente una volta (At most once)
- (1) Invio almeno una volta (At least once)
- (2) Ricevuto esattamente una volta (Exactly once)

Il messaggio va dal publisher al broker e dal broker al subscriber. In queste due fasi il QoS può essere diverso, in base anche alla scelta del subscriber. Dato che la fase di invio di un messaggio, che parte da un publisher e giunge ai subscriber, è divisa in due fasi: publisher->broker e broker->subscriber. Con il termine sender si intende sia publisher che broker, invece, con receiver subscriber e broker.

3.5.1 Livello Zero

QoS=0 Figura 3.2, è il livello più basso, in cui non si hanno garanzie sulla ricezione, il messaggio non verrà memorizzato o ritrasmesso dal sender, quindi l'unica garanzia è quella offerta dal livello sottostante, TCP.

3.5.2 Livello Uno

QoS=1, Figura 3.4, con questo livello viene garantita la ricezione del messaggio. Il sender mantiene in memoria il messaggio finché non riceverà un *PUB ACK* e, se non lo ricevesse entro un determinato tempo, il messaggio dato per perso verrebbe rispedito. È quindi possibile che lo stesso messaggio venga spedito più di una volta, e consegnato due volte, nel caso in cui il primo non fosse andato perso, ma ci fosse stato solo un ritardo. In questo livello, a differenza del livello zero, si usa il campo packetId così da

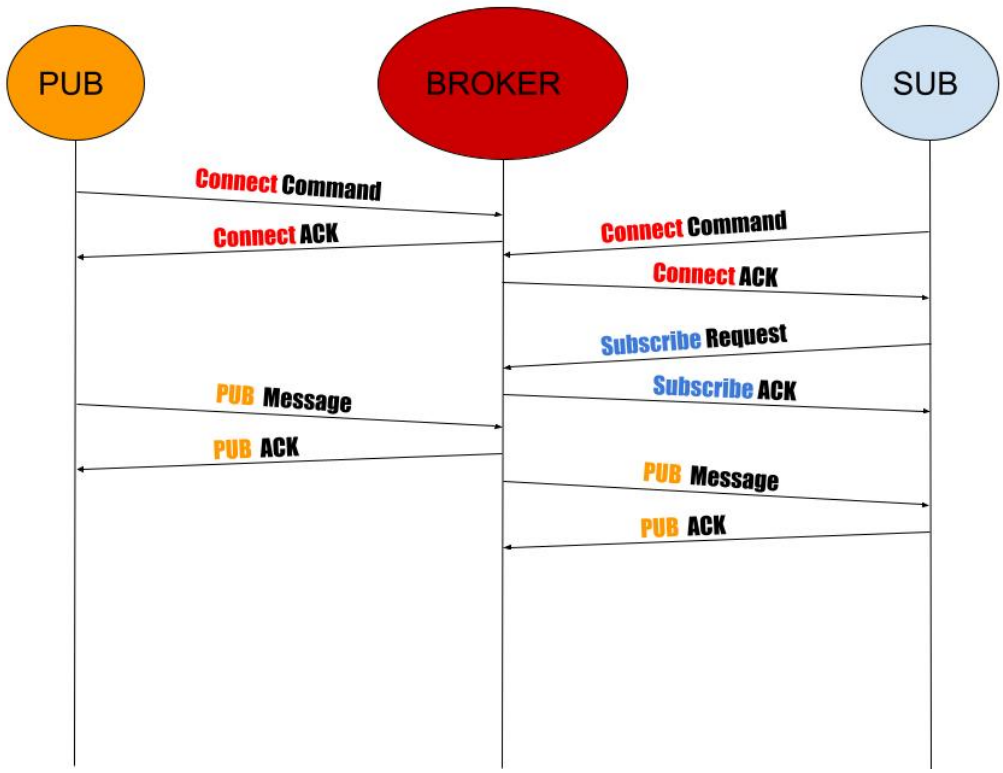

Figura 3.4. Esempio di interazione e messaggi scambiati con QoS=1.

identificare i pacchetti e sapere quali sono stati ricevuti. In caso scadesse il timeout, il sender spedirebbe nuovamente il messaggio e questo verrebbe contrassegnato con il *Dupflag*, anche se in QoS=1 non è utilizzato.

3.5.3 Livello Due

QoS=2 Figura 3.5, questo è il livello di affidabilità più alto. Con questo livello si garantisce che il pacchetto venga inoltrato ai subscriber solamente una volta, tuttavia, questo livello è il più lento, in quanto qui si avrà uno scambio di 4 pacchetti invece di 2, come nel QoS=1. Quando il receiver riceve

un messaggio con QoS=2, risponderà con un *PUB Received*, se il sender non dovesse ricevere il *PUB Received* entro un determinato tempo, allora rinvierà il messaggio contrassegnandolo però con *Dupflag* per indicare che è una copia. Una volta ricevuta la conferma di ricezione, *PUB Received*, il sender potrà scartare dalla propria memoria il messaggio e salvare il pacchetto *PUB Received*, in quanto, il sender saprà per certo che almeno una copia del messaggio è giunta a destinazione. Ora il sender può rispondere con *PUB Release*, il receiver, quando riceverà *PUB Release* saprà che il sender non spedirà altre copie del messaggio, visto che ha ricevuto il *PUB Release* e che quindi è possibile processare il messaggio in sicurezza e inoltrare il messaggio ai subscriber. Se il sender non dovesse ricevere il *PUB Release*, dovrà rispedire il *PUB Received*. L'ultimo messaggio spedito dal receiver è il *PUB Complete* per confermare la ricezione del *PUB Release*. Il sender, invece, se non dovesse ricevere l'ultimo pacchetto, dovrà rimandare il *PUB Release*, tuttavia, se un primo *PUB Release* è già stato ricevuto e quindi, è solo andato perso il *PUB Complete*, l'inoltro dei pacchetti sarà già stato eseguito e non verrà eseguito una seconda volta.

Terminato lo scambio dei 4 pacchetti, e quindi quando il sender riceverà il *PUB Complete*, il valore usato dal message packetId sarà di nuovo reso disponibile per un altro pacchetto. Da notare che packetId è univoco per la coppia Client e broker, inoltre il valore massimo per packetId è 65535.

In quest'ultimo livello di QoS, si ha questo secondo scambio di messaggi, *PUB Release* e *PUB Complete*, per essere sicuri di non inoltrare lo stesso messaggio ai subscriber più volte. Infatti, in QoS=1, in caso di *PUB ACK*

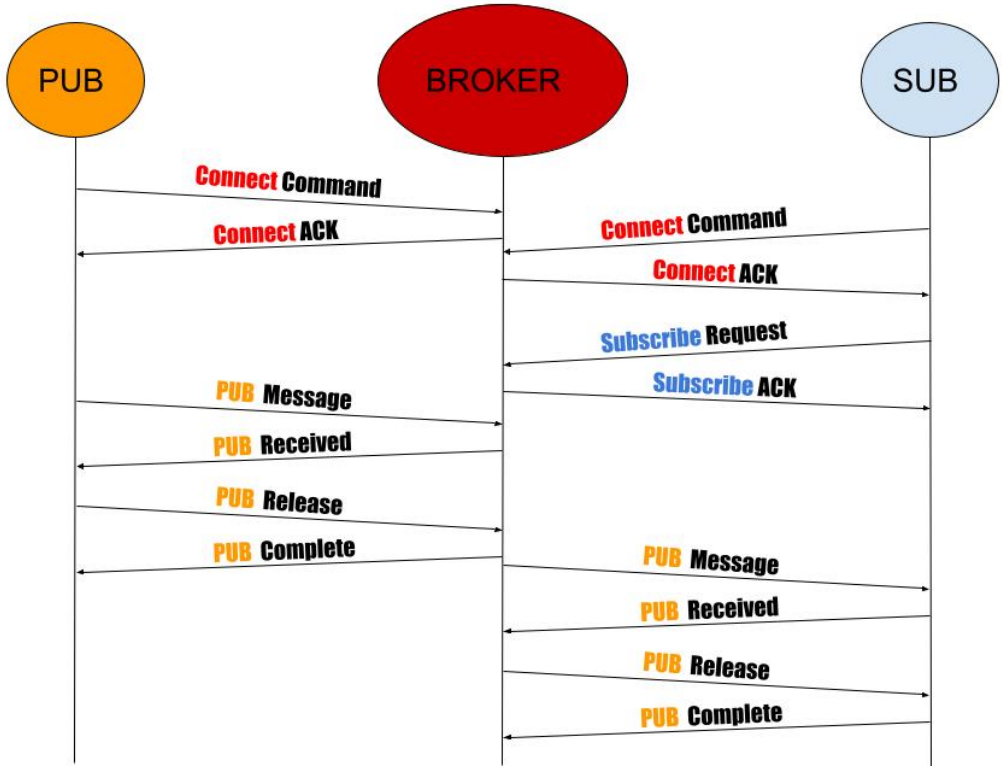

Figura 3.5. Esempio di interazione e messaggi scambiati con QoS=2.

perso, il publisher rispedirebbe il pacchetto e il broker, quindi, inoltrerà una seconda volta, ai subscriber, lo stesso messaggio. In QoS=2 invece, dopo aver ricevuto il messaggio *PUB Received*, il broker non inizierà la fase di inoltro, ma aspetterà la ricezione di *PUB Release*, per essere sicuro che il sender non spedirà altre copie di quel messaggio. Un'altra differenza tra QoS=1, QoS=2 e QoS=0 riguarda il salvataggio dei messaggi sul broker, cioè, se il Client ha una *Persistent Session* e ha fatto una sottoscrizione, ma ora è off-line, quel pacchetto verrà tenuto in memoria e non appena sarà online verrà consegnato.

3.6 Persistent Session

Quando un Client si connette al broker, può richiedere una *Persistent Session* settando il flag *Clean Session* a *False*. Questa impostazione avviene durante la fase iniziale di connessione al broker. Se non esplicitato, di default è *True* e il broker non terrà in memoria alcuna informazione che riguarda il Client, una volta disconnesso. Ad esempio, in caso un subscriber dovesse essere off-line, il broker non terrà in memoria i pacchetti destinati a quel Client non ancora consegnati, oppure, la lista dei topic a cui si è sottoscritto. In una connessione con *Persistent Session*, il broker dovrà invece ricordarsi dei topic a cui il Client si sottoscriverà e non sarà necessario, in caso di riconnessione del Client, mandare nuovamente richieste di sottoscrizioni, in quanto il broker avrà mantenuto le informazione a riguardo. Verranno tenuti in memoria anche i messaggi non ancora inviati, ma solo quelli con con QoS 1 e 2 destinati a quel Client o che non hanno ricevuto conferma di ricezione anche se già inviati.

3.7 Retaired Message

Nel protocollo MQTT esistono dei messaggi chiamati *Retaired Message*, per ogni topic viene tenuto in memoria un solo messaggio. Quando un publisher pubblica un messaggio, può essere settato come *Retaired Message*, in modo che il broker terrà in memoria, per quel topic, l'ultimo messaggio ricevuto con il flag *Retaired Message* settato a *True*. In questo modo, ogni volta che un subscriber si sottoscrive a un topic, esplicitando di volere il *Retaired*

Message, il broker spedirà subito il *Retaired Message* che ha in memoria per quel topic. Questo non ha nulla a che fare con le Persistent Session. Il *Retaired Message* è utile per far sapere ai nuovi subscriber, qual è stato l'ultimo messaggio spedito in un topic, senza aspettare che ci sia un nuovo messaggio pubblicato su quel topic.

3.8 Keep alive

Un grosso problema del TCP su cui si appoggia MQTT è quando uno dei 2 punti smette di funzionare, e in questo caso si ha "half-open connection", in cui il lato funzionante continua a essere connesso e a mandare messaggi aspettando ACK del TCP. Per questo problema, in MQTT, esiste il keep alive, il quale assicura che la connessione tra broker e Client sia ancora aperta, e che entrambi siano funzionanti. Se broker e Client si scambiassero messaggi con un tempo minore del keep alive impostato, allora non sarebbe necessario usare i messaggi di Ping, in caso contrario sì. Se il Client non inviasse nulla per un tempo maggiore del keep alive, sarà necessario inviare un *Ping Request* al broker, il quale, per confermare che è ancora operativo, risponderà con *Ping Response*. Se il Client non ricevesse nessuna risposta la connessione verrebbe considerata chiusa. Il keep alive viene specificato durante la richiesta di connessione, se il valore di keep alive è zero allora non verrà eseguito nessun Ping.

Capitolo 4

Valutazione sperimentale dell'uso combinato di OPC UA e MQTT

4.1 Introduzione

Vista l'importanza che sta avendo OPC UA nel campo dell'Industria 4.0, in particolare, grazie alla sua peculiarità nel semplificare lo scambio di dati con macchinari diversi, si è deciso di analizzare le tempistiche nella comunicazione utilizzando questo protocollo. La versione classica di OPC UA si basa sul modello Client/Server, tuttavia, in questo capitolo, ci si è focalizzati sul valutare le prestazioni della nuova specifica di OPC UA, cioè quella del capitolo 14 Publish/Subscribe. Questa nuova specifica non è strettamente

vincolata e permette vari approcci, ad esempio, il *Message Oriented Middleware*, il quale, nel modello Publish/Subscribe svolge il ruolo di inoltrare i messaggi tra publisher e subscriber, può essere un'infrastruttura, chiamata broker, oppure si può semplicemente usare una comunicazione UDP multicast, senza l'ausilio di un broker per inoltrare messaggi. Nella versione basata su broker è previsto l'uso di alcuni protocolli tra cui MQTT. Si è deciso di testare la nuova specifica di OPC con il protocollo MQTT, inoltre, i messaggi che sono stati trasmessi in questo esperimento con MQTT, sono codificati con JSON, cosa del tutto permessa dalla specifica del capitolo 14 di OPC UA.

Esistono diverse librerie di OPC UA, anche open-source, tuttavia, anche se si basano tutte sulla documentazione di OPC UA pubblicata da OPC Foundation, non tutte si comportano esattamente nello stesso modo, e l'unica libreria open-source che attualmente permette l'uso del modello Publish/-Subscribe è quella denominata open62541. Per questo motivo, la libreria utilizzata per OPC UA, per questi esperimenti, è stata proprio quest'ultima, sviluppata con il linguaggio C. Anche se non è del tutto completa, avendo ancora la parte di Publish/Subscribe nella fase di "working in progress". Nella libreria open62541, la versione che attualmente è meglio implementata e che quindi si è potuta utilizzare per questi esperimenti, è quella del Server OPC UA, mentre la versione Client OPC UA non è attualmente sviluppata a sufficienza da permettere di utilizzarla nell'esperimento. Per questo motivo, si è scelto di usare, sia per il publisher che per il subscriber, un Server OPC UA invece di utilizzare un Client OPC UA come subscriber, come un

tipico scenario produttori e consumatori potrebbe suggerire.
L'idea, alla base di questo esperimento consiste nell'inviare, con un intervallo di 2 secondi, un pacchetto *NetworkMessage* dal publisher al subscriber, che saranno in esecuzione sulla stessa macchina, e di analizzare le tempistiche di trasmissione. Il *DataSet* contenuto nel *NetworkMessage* è semplicemente una variabile di tipo `Int32`.
Dato che il protocollo di MQTT prevedere lo scambio di messaggi con tre livelli di QoS (Quality of Service), per avere un confronto completo, si è preferito ripetere l'esperimento cambiando solamente il QoS, ciascuno con QoS pari a 0, 1 e 2.

4.2 Setup sperimentale

Per questo esperimento, le macchine utilizzate per mandare in esecuzione i Server OPC UA e il broker sono state tre schede Raspberry di diverse versioni (2, 3 e 4), tutte collegate tramite cavo Ethernet ad uno switch e tutte con il sistema operativo Raspbian GNU/Linux 10 (buster). Nella Raspberry 3 si è installato il broker mosquitto [15], indispensabile per poter utilizzare la comunicazione con il protocollo MQTT, il quale avrà il ruolo di *Message Oriented Middleware* per OPC UA, mentre nelle Raspberry 2 e 4, si sono mandati in esecuzione i Server OPC UA.
All'interno della Raspberry 4, sono stati mandati in esecuzione due Server OPC UA, che si comportano rispettivamente da publisher e da subscriber, mentre nella Raspberry 2, è presente solo un Server OPC UA, che si comporta da subscriber, ma che, contemporaneamente, inoltra tutti i messaggi

su di un altro topic. La comunicazione ha inizio dalla Raspberry 4, precisamente dal Server OPC UA che si comporta da publisher. Questo spedisce un *NetworkMessage* ogni 2 secondi. Nel mentre, nella Raspberry 2, un altro Server OPC UA, che si comporta da subscriber, rimarrà in attesa di ricevere il pacchetto. Allo stesso tempo, il codice del subscriber, presente nella Raspberry 2, è stato modificato, così che possa inoltrare tutti i pacchetti ricevuti, su di un topic diverso da quello usato dal mittente. Il messaggio inoltrato verrà poi ricevuto dall'altro Server OPC UA, presente nella Raspberry 4, che, come detto in precedenza, si comporta da subscriber. Per la cattura dei pacchetti, assieme alle loro tempistiche, ci si è serviti del programma "tcpdump" [16], che è stato appositamente installato ed eseguito, in contemporanea, sulle tre Raspberry. Il programma tcpdump, una volta avviato, cattura i pacchetti che transitano sulla scheda di rete scelta, in questo caso, quella della rete Ethernet. Terminato l'esperimento si avrà per ogni Raspberry il file di estensione `.pcap` che contiene i pacchetti con le loro tempistiche.

In Figura 4.1 si vede come avviene lo scambio di un pacchetto con le tre Raspberry, si può notare come il compito della Raspberry 2 sia semplicemente quello di rispedire indietro tutti i pacchetti.

Quindi avremo che ogni pacchetto partirà dalla Raspberry 4, per poi arrivare al broker situato sulla Raspberry 3, il quale spedirà il suddetto pacchetto all'unico subscriber di quel topic, in questo caso, quello posto nella Raspberry 2. Il pacchetto verrà poi inoltrato su un topic diverso, da quello di partenza, in modo da essere ricevuto solamente dall'altro Server OPC UA,

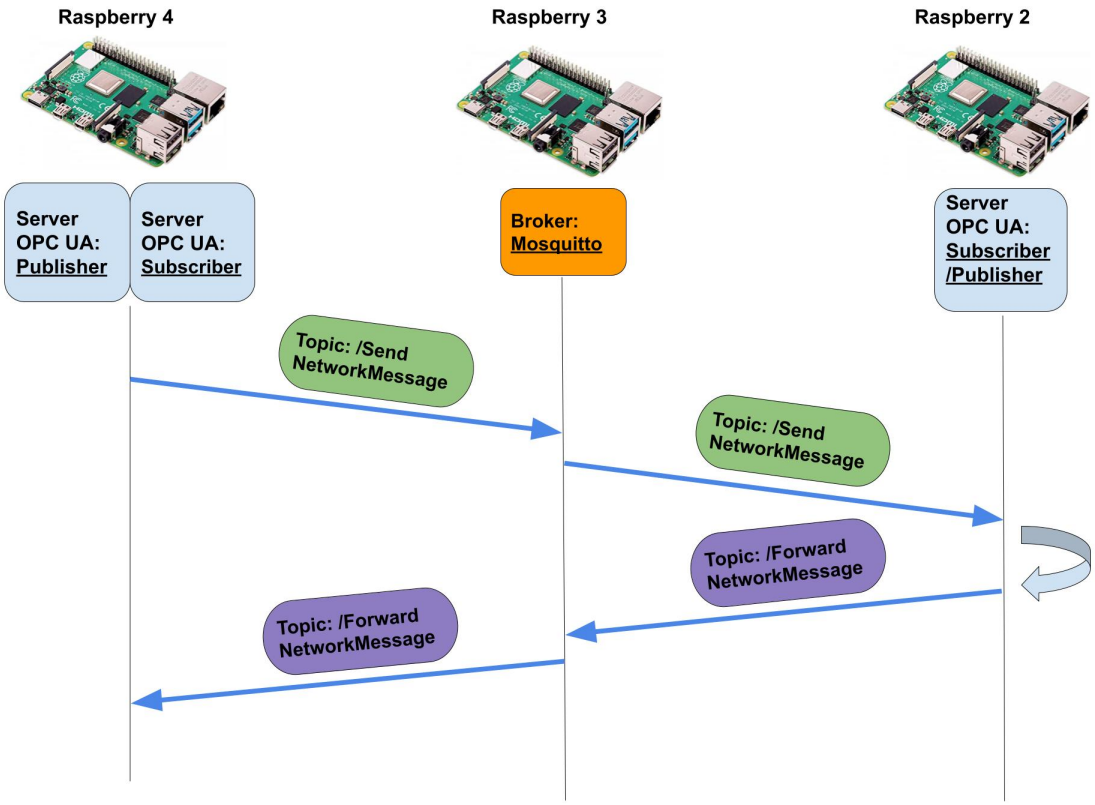

Figura 4.1. Setup sperimentale

presente nella Raspberry 4.

Si è scelto questo sistema con i pacchetti che tornano indietro, in modo da poter analizzare con precisione le tempistiche. Infatti, anche se le tre Raspberry sono sincronizzate tramite NTP (Network Time Protocol) [17], avremo comunque un margine di imprecisione. Per questo motivo si è deciso di seguire quest'altro approccio in cui, facendo tornare lo stesso pacchetto nuovamente sulla Raspberry di partenza, è possibile eseguire i calcoli con delle misurazioni che hanno la stessa base dei tempi.

Nelle successive tre sezioni verrà spiegato nel dettaglio il codice dei vari

Server OPC UA.

4.2.1 Server OPC UA Publisher su Raspberry 4

Di seguito è mostrata parte del codice usato.

```
int main(int argc, char **argv)
{
  UA_Server *server = UA_Server_new();
  UA_ServerConfig *config = UA_Server_getConfig(server);
  UA_ServerConfig_setDefault(config);

  addressSpace(server);
  addPubSubConnection(server, addressUrl);
  addPublishedDataSet(server);
  addDataSetField(server);
  addDataSetWriter(server, topic);

  UA_Server_run(server, &running);

  return 0;
}
```

con le prime 3 righe di codice, si crea il Server OPC UA e si configura con le impostazioni di default.

- `addressSpace` si imposta nell'AddressSpace un nodo che contiene un valore di tipo `Int32`.

- `addPubSubConnection` si stabilisce la connessione con il broker, il cui indirizzo è contenuto nella variabile `addressUrl`.
- `addPublishDataSet` si prepara il *DataSet* con tutte le informazioni necessarie.
- `addDataSetField` si aggiunge un campo al *DataSet* creato in precedenza, e si passa un riferimento al nodo creato in precedenza nell'*AddressSpace*.
- `addDataSetWriter` crea il *DataSetWriter*.
- `UA_Server_run` con questa funzione si manderà in esecuzione il Server OPC UA che trasmetterà tramite MQTT, ogni 2 secondi, quel determinato nodo del proprio *AddressSpace*.

4.2.2 Server OPC UA Subscriber su Raspberry 4

Come già detto in precedenza, sulla Raspberry 4, oltre al Server OPC UA, nel ruolo di publisher, ci sarà in esecuzione, su un processo diverso, un altro Server OPC UA che si comporterà solo da subscriber, il quale farà la sottoscrizione a un topic diverso da quello dell'altro Server OPC UA sulla stessa Raspberry. Di seguito il codice

```
int main(int argc, char **argv)
{
  UA_Server *server = UA_Server_new();
  UA_ServerConfig *config = UA_Server_getConfig(server);
  UA_ServerConfig_setMinimal(config, 4850, NULL);
```

```
    addPubSubConnection(server, addressUrl);
    addDataSetReader(server);

    addSubscribedVariables(server, readerIdentifier);

    UA_Server_run(server, &running);
    return 0;
}
```

Come per la versione del publisher, anche qui le prime 3 righe sono per creare e impostare il Server OPC UA, con la sola differenza che si sceglie una porta diversa, che in questo caso è la 4850.

- `addPubSubConnection` stabilisce la connessione con il broker, il cui indirizzo è contenuto nella variabile `addressUrl`.

- `addDataSetReader` si prepara il *DataSetReader* che dovrà occuparsi della decodifica del *NetworkMessage*.

- `addSubscribedVariables` crea un nodo nell'*AddressSpace* che verrà aggiornato con il valore di tipo `Int32` inviato dal publisher.

- `UA_Server_run` con questa funzione si manderà in esecuzione il Server OPC UA.

4.2.3 Server OPC UA Subscriber/Inoltro su Raspberry 2

Qui si è usato un codice simile a quello del subscriber precedente, con l'unica differenza che all'interno del codice, esattamente dove avviene la lettura del pacchetto, si è aggiunta una ripubblicazione del pacchetto appena ricevuto, dal publisher della Raspberry 4, che verrà pubblicato su un altro topic, ovvero quello usato dal subscriber che è in esecuzione sulla Raspberry 4. In sostanza, a differenza della Raspberry 4, qui si ha un solo processo che manda in esecuzione l'unico Server OPC UA, con l'aggiunta di una modalità echo.

4.3 Risultati ed osservazioni

Terminati gli esperimenti, si sono presi i file `.pcap` generati da tcpdump e analizzati, ogni pacchetto che è transitato sulle Raspberry è stato identificato tramite un *sequence number*. Per ogni Raspberry e per ciascun pacchetto avremo quindi due tempi, quello di uscita e di entrata, l'unica eccezione si ha con la Raspberry 3, il broker, il quale avrà, per ogni pacchetto 4 tempi, 2 per l'andata e 2 per il ritorno. In Figura 4.2 sono mostrati i simboli usati per i tempi; con il simbolo T_x si intendono i tempi misurati da tcpdump, quando il pacchetto è in uscita o in entrata, mentre, con S_x si intende la differenza dei tempi in quel tratto. I tre tcpdump che girano sulle Raspberry, misurano i tempi in base a quella del sistema operativo in cui sono in esecuzione, quindi non è possibile fare la differenze tra due tempi presi da

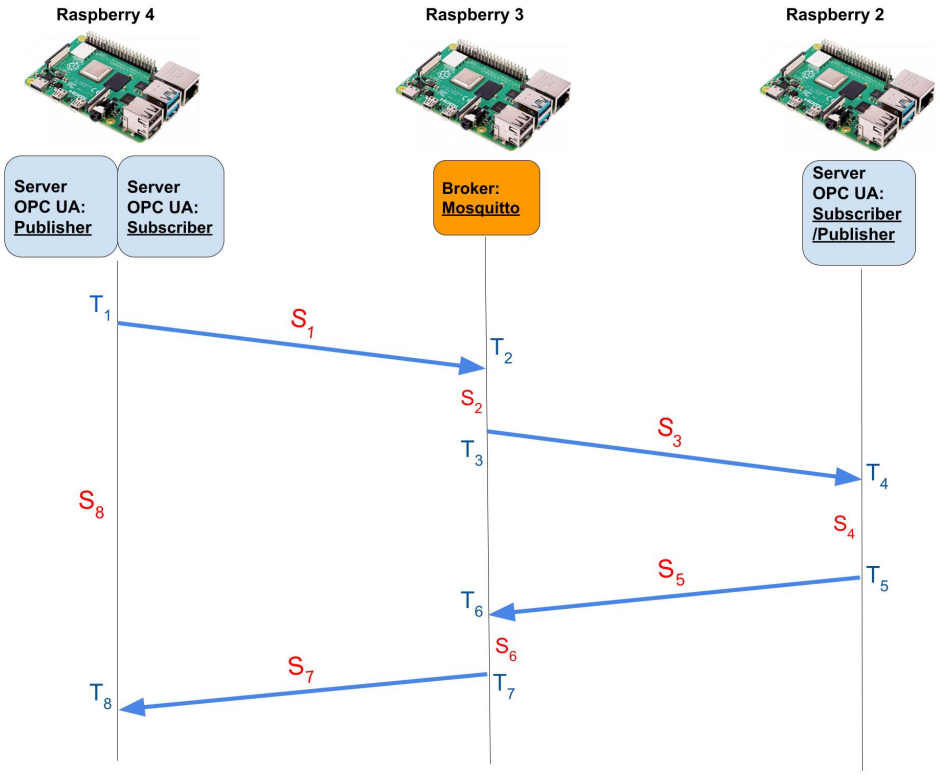

Figura 4.2. Simboli per le tempistiche

due diverse Raspberry, o meglio, si potrebbe, ma si effettuerebbe la differenza tra due tempi che hanno dell'incertezza. Di seguito sono mostrate le formule utilizzate per calcolare le latenze.

$$S_1 = \frac{(T_8 - T_1) - (T_7 - T_2)}{2}$$

$$S_2 = T_3 - T_2$$

$$S_3 = \frac{(T_6 - T_3) - (T_5 - T_4)}{2}$$

$$S_4 = T_5 - T_4$$

$$S_5 = \frac{(T_6 - T_3) - (T_5 - T_4)}{2}$$

$$S_6 = T_7 - T_6$$

$S_7 = \frac{(T_8 - T_1) - (T_7 - T_2)}{2}$

$S_8 = T_8 - T_1$

Come si può notare, i calcoli sono stati fatti solo su valori con la stessa base dei tempi, e quindi provenienti dalla stessa Raspberry. Tuttavia, per alcuni non è stato possibile, ma ci si è dovuti accontentare di effettuare una media tra l'andata il ritorno. Infatti, per S_1, S_3, S_5 e S_7 si è potuto calcolare solo la media tra l'andata e il ritorno dello stesso pacchetto. Per questo motivo le due coppie S_1, S_7, e S_3, S_5 risultano essere uguali. I tempi S_2 e S_6 sono impiegati dal broker, da quando arriva il pacchetto a quando lo spedisce. Infine, S_8 è il tempo complessivo impiegato da un pacchetto per tornare al punto di partenza.

Nelle Tabelle 4.1, 4.2, 4.3 sono mostrati, per ogni tratta, il tempo minimo, massimo e media di tutti e tre gli esperimenti.

QoS=0 Intervallo 2 secondi								
#packets 1233	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
Minimo (ms)	0.185	0.331	0.362	5.213	0.362	0.356	0.185	7.539
Massimo (ms)	0.261	2.998	0.548	14.690	0.548	4.600	0.261	17.242
Media (ms)	0.212	0.585	0.428	8.272	0.428	0.668	0.212	10.805

Tabella 4.1. QoS=0

QoS=1 Intervallo 2 secondi								
#packets 1572	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
Minimo (ms)	0.204	0.446	0.376	5.444	0.376	0.356	0.204	7.994
Massimo (ms)	0.299	4.274	0.757	59.220	0.757	4.027	0.299	61.889
Media (ms)	0.231	0.719	0.427	21.745	0.427	0.678	0.231	24.458

Tabella 4.2. QoS=1

QoS=2 Intervallo 2 secondi								
#packets 1752	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
Minimo (ms)	0.196	100.557	0.350	5.499	0.35	2.804	0.196	111.526
Massimo (ms)	0.314	103.464	0.916	18.768	0.916	60.955	0.314	174.68
Media (ms)	0.225	100.838	0.406	8.691	0.406	25.343	0.225	136.133

Tabella 4.3. QoS=2

Per valutare meglio l'aspetto latenze, importante nelle comunicazioni industriali, si è preso in esame, per ogni esperimento, il tempo massimo dato da S_8, cioè il tempo complessivo da quando viene spedito il dato dal publisher a quando viene ricevuto dal subscriber.

4.3.1 Considerazioni sul tempo massimo con QoS=0

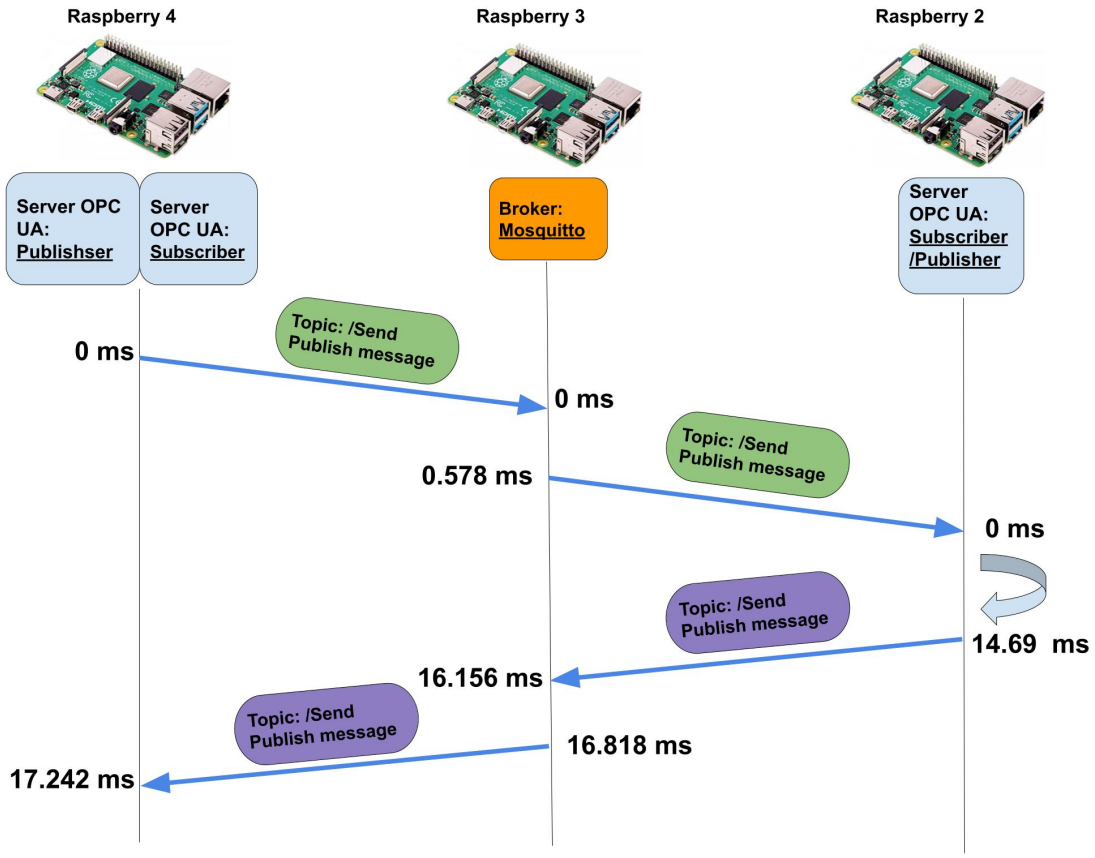

Figura 4.3. Massimo QoS=0

In Figura 4.3 si vedono nel dettaglio le tempistiche del pacchetto durante i vari passaggi. La comunicazione è abbastanza rapida, anche perché

utilizzando il livello QoS=0, non ci sono meccanismi di *handshaking* che prolungano i tempi.

4.3.2 Considerazioni sul tempo massimo con QoS=1

Qui invece i tempi di trasmissione iniziano ad aumentare, anche perché raddoppia il numero di pacchetti scambiati per ogni dato spedito.

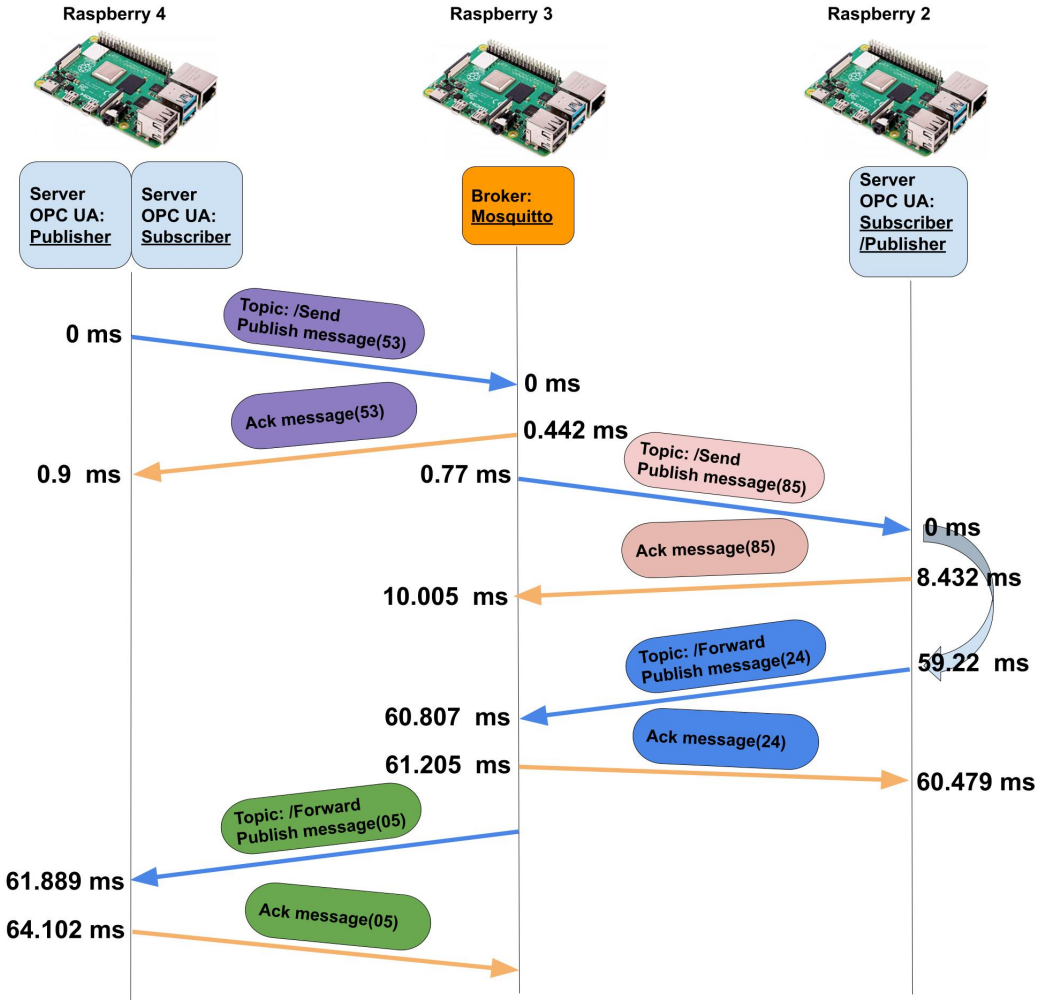

Figura 4.4. Massimo con QoS=1

4.3.3 Considerazioni sul tempo massimo con QoS=2

Il caso più insolito lo possiamo osservare qui, come mostrato dalla Figura 4.5, dove il publisher, da cui parte il pacchetto, non va mai sotto i 100 ms per mandare il pacchetto *Released Message*. Controllando il codice di open62541, per meglio comprendere questo comportamento, si è trovato che, una volta avviato il Server OPC UA, viene attivato un loop con una particolare funzione, che si occupa di mandare in esecuzione i vari processi. Tra un'iterazione e l'altra è impostato un timeout, e la causa del ritardo è dovuta proprio a questo timeout. Andando a porre un timeout pari a 1 ms i tempi cambiano, poiché le iterazioni del loop del Server OPC UA aumentano di frequenza, Tabella 4.4. Nel subscriber non si ha questo aumento di 100 ms perché, durante il tempo di timeout, il Server rimane in ascolto, sbloccandosi prima del tempo solo quando riceve un pacchetto.

QoS=2 Intervallo 2 secondi								
#packets 877	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
Minimo (ms)	0.196	2.471	0.335	4.506	0.335	2.500	0.196	11.055
Massimo (ms)	0.331	6.505	0.552	55.542	0.552	57.296	0.331	67.964
Media (ms)	0.222	2.662	0.394	10.646	0.394	14.919	0.222	29.461

Tabella 4.4. QoS=2, Timeout= 1 ms

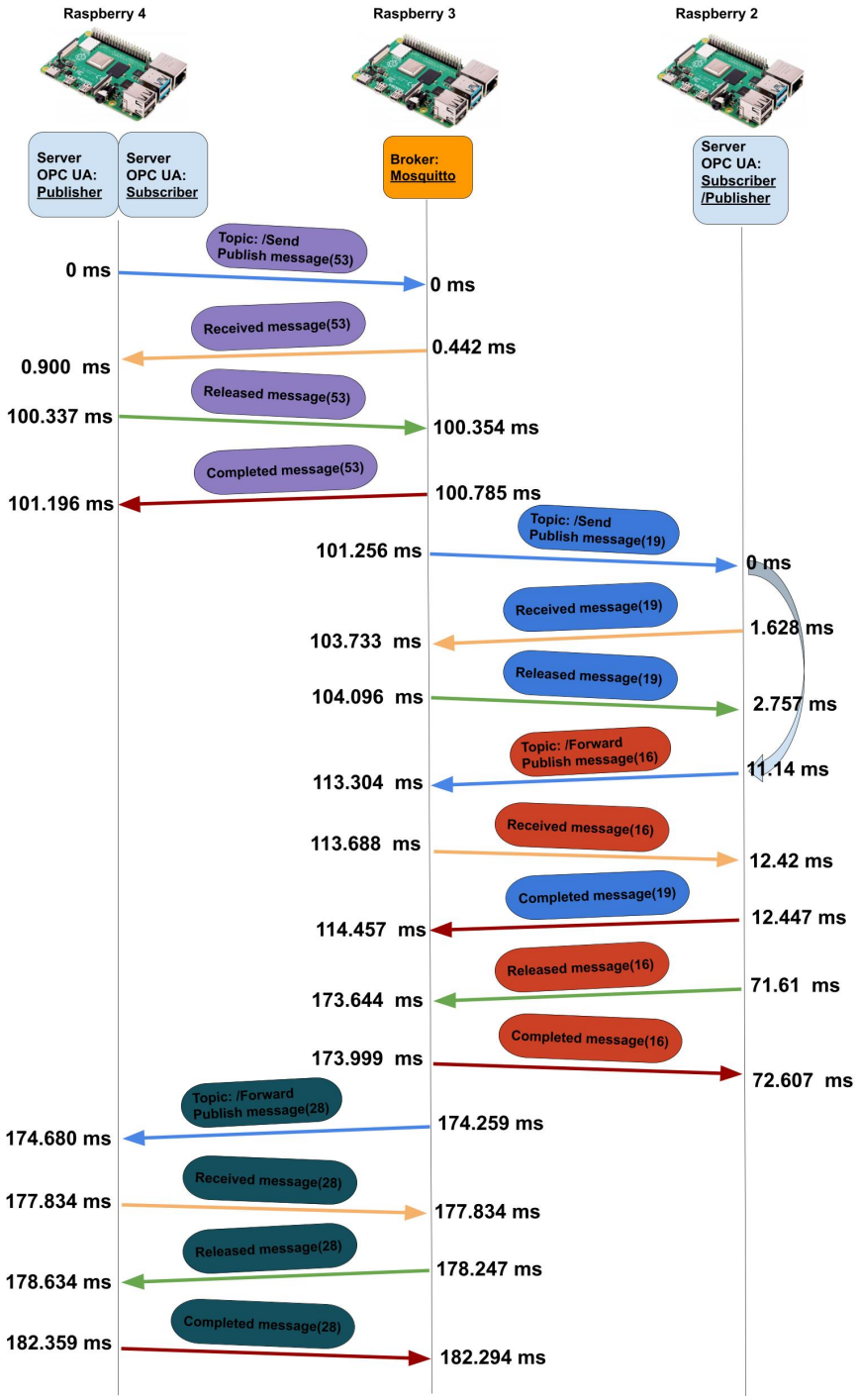

Figura 4.5. Massimo con QoS=2

Capitolo 5

Analisi della latenza di notifica di Publish-Subscribe nei Server OPC UA

5.1 Introduzione

In questo capitolo si analizzerà, in modo più approfondito, il tempo di notifica, a partire da quando il Client sorgente invia un pacchetto *WriteRequest* al Server OPC UA, per modificare un dato, a quando il Client ricevente viene avvisato tramite notifica dal Server. Come già spiegato nei capitoli precedenti, il Server OPC UA dà la possibilità ai Client OPC UA, che ne

fanno richiesta, di essere informati non appena un certo evento si verificasse all'interno del Server OPC UA. Un esempio è il cambiamento di una variabile all'interno dell'*AddressSpace* del Server su richiesta di un altro Client, in questo caso i passaggi da seguire sono i seguenti:

- invio della richiesta di *WriteRequest* dal Client sorgente al Server
- rilevata modifica della variabile osservata all'interno del Server
- invio della notifica al Client ricevente

Le latenze dovute al primo e al terzo passaggio sono causate da fattori di rete, e alle tecnologie usate, come ad esempio Ethernet o Wi-Fi. La latenza dovuta al secondo passaggio invece è causata dal solo Server OPC UA: tale tempo verrà definito in seguito come D_{srv}. L'analisi è stata eseguita a livello rete, del modello ISO/OSI, inoltre sono stati presi in considerazione, per eseguire i test, tre popolari implementazioni di Server OPC UA, vale a dire:

- Free OPC-UA (versione in Python)
- Unified Automation (versione in C)
- open62541 (versione in C)

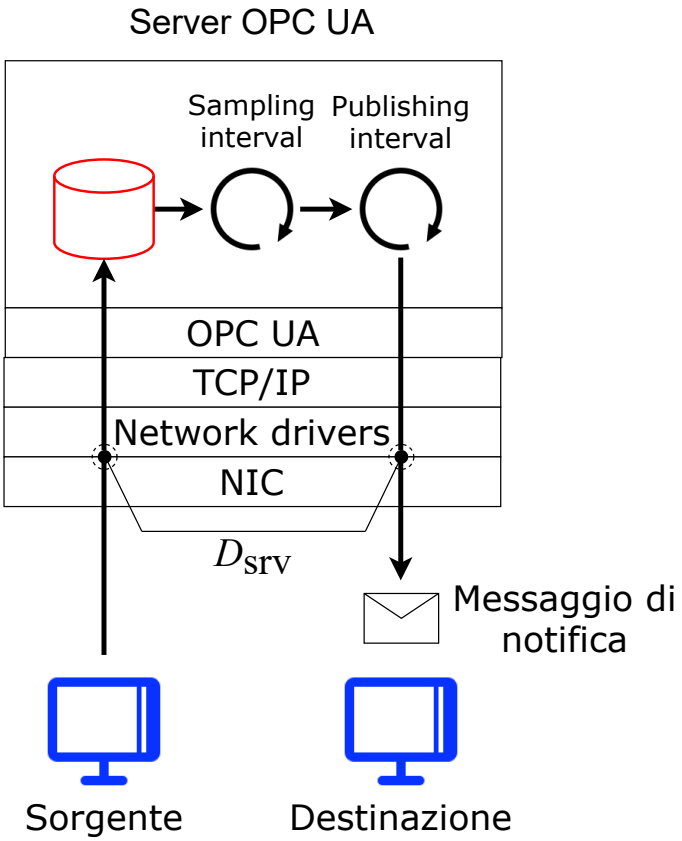

Figura 5.1. Setup sperimentale

5.2 Configurazione e setup sperimentale

Per l'esecuzione dell'esperimento è stato programmato un Client OPC UA che si occupa di aggiornare ripetutamente una variabile nel Server, inviando ciclicamente pacchetti *WriteRequest*. Un secondo Client OPC UA invece è stato programmato per far richiesta di sottoscrizione al Server, cosicché riceverà notifiche in caso quella variabile venisse modificata. Tutti i pacchetti in entrata e in uscita dal Server sono stati catturati e salvati con i loro timestamp, dopodiché, i file di log sono stati analizzati, al fine di determinare le

latenze. Come mostrato in Figura 5.1, all'interno del Server OPC UA, sono presenti due cicli: uno di sampling e uno di publishing. Il primo si occupa di controllare, con un dato intervallo di tempo, se la variabile monitorata è stata modificata, l'altro ciclo, invece, si occupa di controllare periodicamente se ci sono notifiche da inviare (in seguito ad un cambiamento nel valore della variabile, rilevato dal ciclo di sampling) a chi ha fatto una sottoscrizione. Le misurazioni sono state eseguite per valutare i valori medi e minimi di D_{srv}, variando il valore degli intervalli di publishing e sampling richiesti: ($T_{\text{pub\_req}}$ e $T_{\text{sam\_req}}$). Ogni esperimento comprende 1000 campioni ed è stato ripetuto 5 volte. Per quanto riguarda la configurazione, si è impostato nella richiesta del Client di sottoscrizione che gli intervalli di publishing e sampling fossero della stessa durata, cioè $T_{\text{pub\_req}} = T_{\text{sam\_req}}$. Come detto sopra, il processo di generazione delle notifiche sul Client è periodico. Infatti, le operazioni eseguite dal Server OPC UA per monitorare e pubblicare le modifiche della variabile sotto osservazione sono basati su processi ciclici. È importante notare che, sebbene tutti questi processi siano periodici, le basi temporali sul Client sorgente e sul Server non sono sincronizzate, il che significa che si allontanano progressivamente (e arbitrariamente) anche nel caso limite in cui i periodi sono nominalmente gli stessi. Per quanto riguarda le statistiche, se il lasso di tempo durante il quale le statistiche sono calcolate è sufficientemente ampio, possiamo presumere che i processi Client e Server siano effettivamente indipendenti. La selezione dei periodi con numeri come 1, 11, 21, 31, etc., permette di ottenere statistiche affidabili anche su intervalli di tempo relativamente brevi. Pertanto, nel seguente ragionamento,

è possibile ipotizzare che la *WriteRequest* del Client arrivi al Server in un momento uniformemente distribuito all'interno del ciclo di sampling. Ciò implica che il ritardo minimo $D_{\text{sam}}^{\min}$ introdotto dal ciclo di sampling è in teoria 0, mentre il ritardo medio è metà del periodo di sampling, $\overline{D}_{\text{sam}} = 0{,}5 \cdot T_{\text{sam}}$. Poi, c'è un secondo ritardo $\overline{D}_{\text{pub}}$ introdotto dal ciclo di publishing. In questo caso, la base temporale è la stessa del ciclo di sampling, perciò, in condizioni specifiche, i due cicli possono essere spostati di un offset fisso, e questo può portare a un comportamento contro-intuitivo, come vedremo in seguito.

5.3 Risultati

5.3.1 Analisi a scatola chiusa

Di seguito sono riportati i risultati degli esperimenti. Per chiarezza, ogni implementazione del Server viene presentata separatamente.

Unified Automation

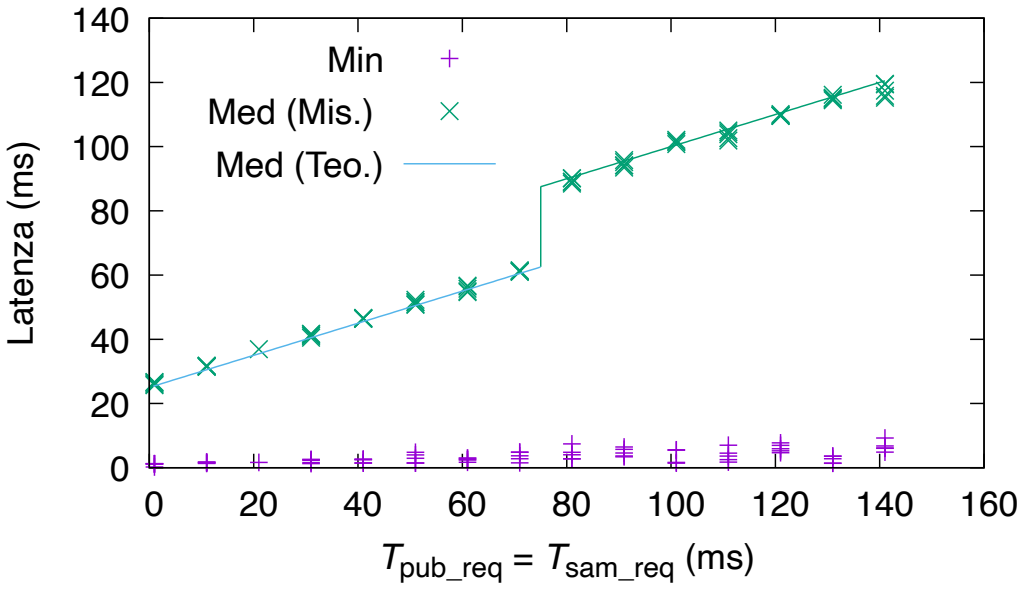

Figura 5.2. Server Unified Automation

Questo Server ha mostrato un comportamento standard. Come possiamo osservare in Figura 5.2, i valori minimi di latenza sono molto bassi e, allo stesso tempo, la latenza media aumenta proporzionalmente ai parametri specificati $T_{\text{pub\_req}}$ e $T_{\text{sam\_req}}$. Il Server Unified Automation non applica alcun valore minimo per T_{pub}, che può essere selezionato liberamente. Per

quanto riguarda T_{sam} invece, all'interno del Server è definito un vettore di 7 valori di sampling preimpostati ($[50, 100, 250, 500, 1000, 2500, 5000]$, espressi in ms) e l'intervallo di sampling T_{sam} è impostato sul valore più vicino all'intervallo di sampling richiesto $T_{\text{sam\_req}}$. Questo porta ad un divario tra i campioni relativi a $T_{\text{sam\_req}} = 71\,\text{ms}$ e $T_{\text{sam\_req}} = 81\,\text{ms}$, perché per $T_{\text{sam\_req}} = 75\,\text{ms}$ il valore di T_{sam} è impostato a 50 ms, mentre per valori maggiori di $T_{\text{sam\_req}} = 75\,\text{ms}$ è impostato a 100 ms. I valori medi sono coerenti con questo comportamento. Infatti, T_{pub} può assumere qualunque valore, mentre T_{sam} può assumere solo i valori pre-impostati nel server. Detto ciò, la latenza media è pari alla metà del periodo di sampling più la metà del periodo di publishing, $D_{\text{srv}} = 0{,}5\ (T_{\text{sam}} + T_{\text{pub}})$. Ad esempio, quando $T_{\text{pub\_req}} = T_{\text{sam\_req}} = 41\,\text{ms}$, T_{sam} viene arrotondato dal Server a 50 e la teoria dice che dovremmo ottenere $D_{\text{srv}} = (50 + 41)/2 = 45{,}5\,\text{ms}$, che concorda molto bene con i risultati numerici misurati. Si può quindi concludere che, il comportamento di questa implementazione commerciale, vista dall'esterno, è conforme a quello che ci aspettiamo dalle specifiche OPC UA.

Free OPC-UA

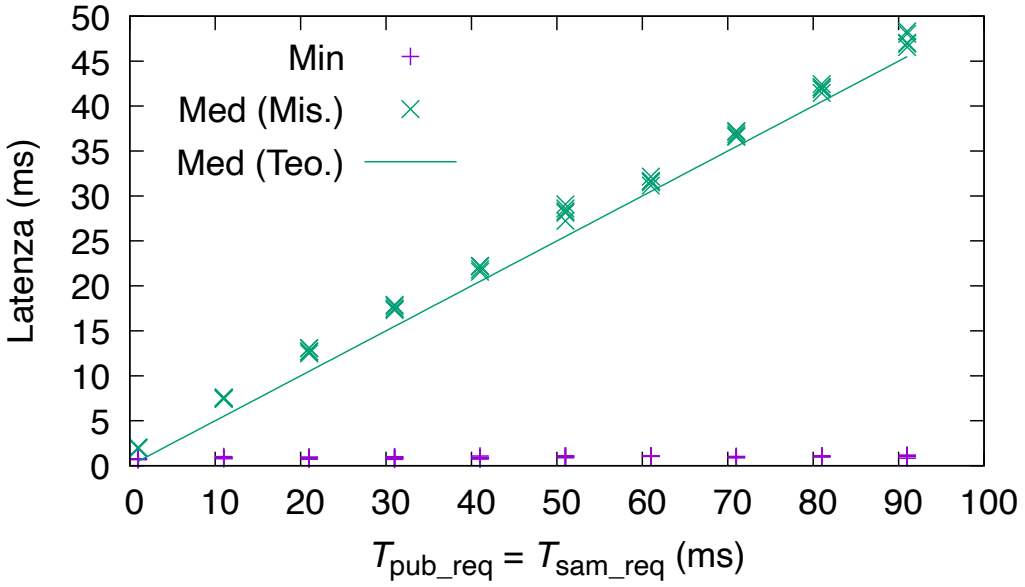

Figura 5.3. Server Free OPC-UA

Il comportamento esterno di questo Server, basato su Python, differisce da quello che ci si può aspettare da OPC UA, come spiegato in precedenza. Come possiamo osservare in Figura 5.3, i valori minimi sono molto vicini allo zero. Anche in questo caso possiamo osservare che, aumentando i tempi di sampling cresce anche la latenza media. Ciò che non è così chiaro, a prima vista, sono i valori misurati. In primo luogo, dall'analisi dei pacchetti catturati, possiamo dedurre che questa implementazione del Server non impone alcun limite minimo per gli intervalli di publishing e sampling, come dimostrato dal comportamento lineare rispetto agli intervalli richiesti dal Client. Quello che però emerge è che il ciclo di sampling non porti alcun contributo alla latenza del Server, come visto dall'esterno. Ad esempio,

quando $T_{\text{pub\_req}} = T_{\text{sam\_req}} = 41\,\text{ms}$, si è misurato $D_{\text{srv}} = 22\,\text{ms}$, che è più o meno la metà del parametro specificato. Lo stesso è stato osservato per gli altri valori di T_{pub}. Una conclusione, supportata anche dall'ispezione del codice sorgente, è che il Server si affida internamente ad una procedura semplificata, con un unico ciclo, sia per il sampling, sia per il publishing.

open62541

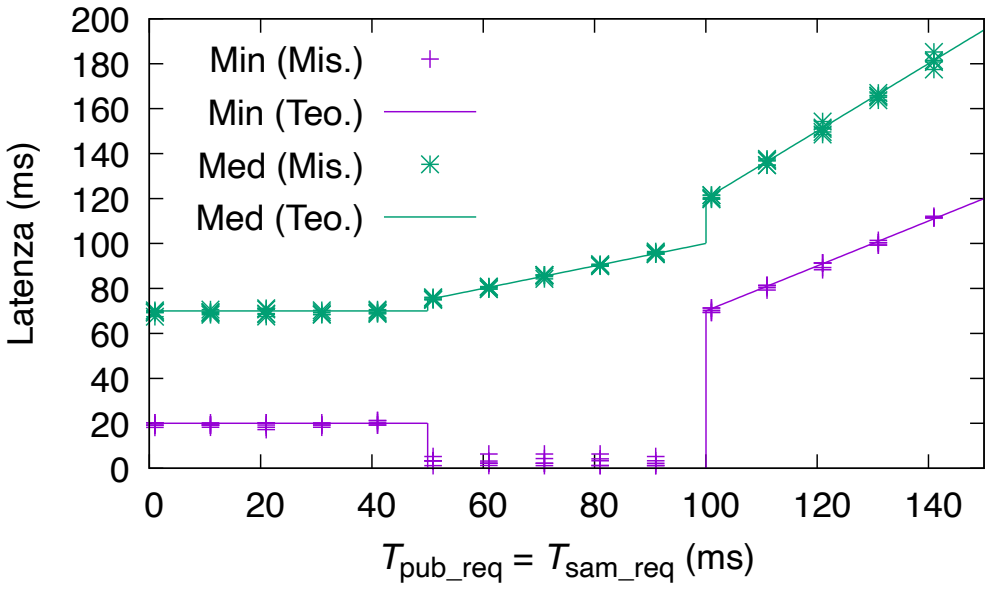

Figura 5.4. Server open62541

Questa è l'ultima implementazione dei Server analizzata, il cui comportamento differisce dai precedenti. In Figura 5.4 si evidenzia un comportamento piuttosto particolare per quanto riguarda i valori minimi della latenza misurata. Secondo questi valori (molto bassi o non trascurabili), il grafico può essere diviso in tre regioni: una prima di $T_{\text{pub\_req}} = T_{\text{sam\_req}} = 50\,\text{ms}$, una

tra 50 ms e 100 ms, e l'ultima dopo 100 ms. Per spiegare questo grafico bisogna considerare, di nuovo, i valori minimi preimpostati per gli intervalli di publishing e sampling e quelli richiesti dal Client, al momento della richiesta di sottoscrizione al Server, e che possono essere modificati da quest'ultimo in fase di negoziazione; è possibile determinare gli intervalli effettivamente impostati dal Server, osservando il contenuto dei pacchetti in risposta alla richiesta del Client. Esaminando i pacchetti si è scoperto che il valore minimo preimpostato per T_{sam} è di 50 ms, mentre per T_{pub} è di 100 ms. Questo è il punto di partenza per capire perché ci sono tre regioni. Inoltre c'è qualcosa all'interno del codice (meglio, nell'architettura interna) dell'implementazione del Server che introduce, in alcuni casi, un ritardo fisso e, quando T_{pub} è più grande di 100 ms, un ritardo minimo superiore a zero che aumenta con T_{pub}. Una descrizione dettagliata del ragionamento è presentata qui sotto.

5.3.2 Funzionamento interno del Server open62541

Dopo l'analisi esterna del Server open62541 sopra riportato, si è deciso di indagare sul codice per scoprire perché sono presenti questi ritardi all'interno del Server, in particolare, controllando il codice, seguendo il flusso di informazioni dal primo messaggio `CreateSubscriptionRequest` fino all'ultimo messaggio `CreateMonitoredItemResponse`. Una prima considerazione riguarda i meccanismi utilizzati per creare i cicli di publishing e di sampling. Entrambi i cicli vengono avviati, in questo ordine.

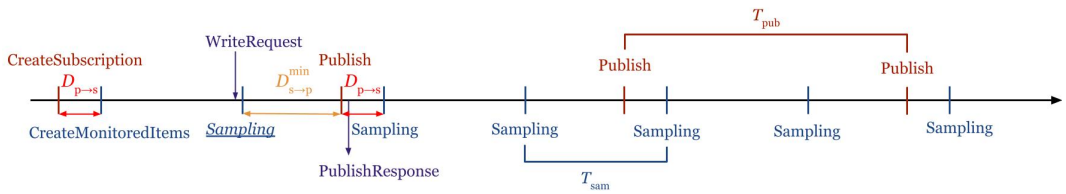

Figura 5.5. Caso quando la latenza D_{srv} si avvicina al minimo (sampling e publishing sincronizzati, $T_{\text{pub}} = 2 \cdot T_{\text{sam}}$)

Come si può vedere in Figura 5.5, l'offset iniziale, $D_{\text{p}\to\text{s}}$ tra i due eventi, è dovuto alla latenza di andata e ritorno tra il Client e il Server. Ad esempio, esaminando la fase di configurazione della sottoscrizione per questo Server, è emerso che $D_{\text{p}\to\text{s}} \simeq 30\,\text{ms}$ (Client e Server erano situati in due posizione diverse a Torino). Se i due intervalli T_{sam} e T_{pub} sono uguali, nell'ipotesi che i cicli di sampling e publishing siano sincronizzati, lo spostamento nel tempo $D_{\text{s}\to\text{p}}$ tra gli eventi correlati rimane costante. Ciò significa che tutte le operazioni sono ripetute esattamente nello stesso ordine e con gli stessi tempi. Nel migliore dei casi, l'operazione di sampling avviene subito dopo una modifica al valore monitorato, ma bisogna aspettare $D^{\text{min}}_{\text{s}\to\text{p}}$ prima dell'operazione di publishing, dove $D^{\text{min}}_{\text{s}\to\text{p}} = T_{\text{sam}} - D_{\text{p}\to\text{s}}$. Più in generale, se T_{pub} è un multiplo intero di T_{sam}, cioè $T_{\text{pub}} = K \cdot T_{\text{sam}}$, allora $D_{\text{s}\to\text{p}} = D^{\text{min}}_{\text{s}\to\text{p}} + \ell \cdot T_{\text{sam}}$, dove il valore discreto ℓ è distribuito uniformemente in $[0, K-1]$ cosicché, la latenza minima è pari a

$$D^{\text{min}}_{\text{srv}} = T_{\text{sam}} - D_{\text{p}\to\text{s}} \tag{5.1}$$

Una considerazione analoga può essere fatta per la latenza media. In questo caso il ciclo di publishing non ha alcun impatto sulla variabilità dei risultati,

in quanto segue il relativo evento di sampling di $D_{\mathrm{s}\to\mathrm{p}}$. Ciò che conta è il ciclo di sampling, e solo questa operazione può introdurre un ritardo casuale. Ergo, per un dato intervallo di sampling, la latenza del Server D_{srv} può essere modellata come una variabile casuale distribuita uniformemente tra $D^{\mathrm{min}}_{\mathrm{s}\to\mathrm{p}}$ e $D^{\mathrm{min}}_{\mathrm{s}\to\mathrm{p}} + T_{\mathrm{pub}}$ come illustrato in Figura 5.6. La latenza media coincide con il punto medio di questo intervallo, cioè

$$\overline{D}_{\mathrm{srv}} = D^{\mathrm{min}}_{\mathrm{s}\to\mathrm{p}} + 0.5 \cdot T_{\mathrm{pub}}. \tag{5.2}$$

Il comportamento visibile esterno del Server può essere spiegato ipotizzando che il Server aumenti automaticamente i valori $T_{\mathrm{pub\_req}}$ e $T_{\mathrm{sam\_req}}$ richiesti dal Client sottoscrittore in modo che non siano inferiori ai valori minimi consentiti, cioè $T_{\mathrm{sam}} = 50\,\mathrm{ms}$ e $T_{\mathrm{pub}} = 100\,\mathrm{ms}$. Se guardiamo la regione più a sinistra della Figura 5.4, quando gli intervalli richiesti $T_{\mathrm{sam\_req}}$ e $T_{\mathrm{pub\_req}}$ sono impostati sui valori 1 ms, 11 ms, 21 ms, 31 ms e 41 ms, tutti i valori negoziati sono aumentati ai minimi consentiti. La latenza minima $D^{\mathrm{min}}_{\mathrm{srv}}$, come previsto, è di circa 20 ms. Per quanto riguarda il valore medio della latenza, in teoria si ottiene che $\overline{D}_{\mathrm{srv}} = (20 + 100/2) = 70\,\mathrm{ms}$. La stessa spiegazione si applica anche alla regione più a destra della Figura 5.4, quando entrambi gli intervalli richiesti sono più grandi di 100 ms e quindi non vengono toccati dal Server. Superata tale soglia, i valori per gli intervalli di publishing e sampling sono esattamente come richiesti dal Client sottoscrittore. Quindi, il minimo teorico della latenza è di circa 30 ms inferiore al T_{pub} richiesto, mentre il valore medio teorico può essere calcolato utilizzando 5.1.

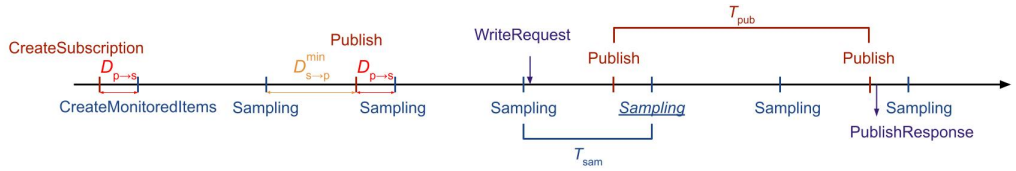

Figura 5.6. Caso quando la latenza D_{srv} si avvicina al massimo (sampling e publishing sincronizzati, $T_{\text{pub}} = 2 \cdot T_{\text{sam}}$)

L'ultimo caso rimasto è quello nella regione centrale, dove gli intervalli richiesti sono tra 50 ms e 100 ms. In questo caso, come è possibile vedere dalla Figura 5.7 i due cicli di publishing e sampling vengono creati e sincronizzati con uno sfasamento di $D_{\text{p}\to\text{s}}$, come anche nelle altre due regioni prima di 50 ms e dopo 100 ms. Tuttavia, a differenza delle altre due regioni, i tempi effettivamente impostati di T_{sam} saranno uguali a $T_{\text{sam\_req}}$, mentre T_{pub} rimarrà costante. Tenendo presente che il tempo minimo di T_{sam} è 50 ms, mentre, per T_{pub} è 100 ms, avremo che il primo cambierà in base alla richiesta $T_{\text{sam\_req}}$, poiché si avranno richieste maggiori di 50 ms, invece T_{pub} rimarrà sempre 100 ms, dato che i valori di $T_{\text{pub\_req}}$ saranno sotto il suo minimo ammissibile. Pertanto, anche se i due cicli partono sincronizzati, avremo durante il funzionamento del Server, un continuo sfasamento tra i due cicli. In queste condizioni, una latenza vicina a zero può essere sperimentata ogni volta che il sampling avviene appena prima del publishing. Anche la latenza media è coerente con questo comportamento: ad esempio, per un intervallo pari a 61 ms, si ha, in teoria, $D_{\text{srv}} = (61 + 100)/2 = 80{,}5$ ms, che corrisponde molto bene a quello che si è misurato negli esperimenti. Da notare: anche se, nella prima regione, i tempi di T_{sam} e T_{pub} sono diversi,

non si ha uno sfasamento continuo e $D_{\mathrm{p}\to\mathrm{s}}$ rimarrà costante, poiché T_{pub} è multiplo di T_{sam}, per questo motivo si riscontra lo stesso fenomeno di non disallineamento che mantiene costante la latenza minima.

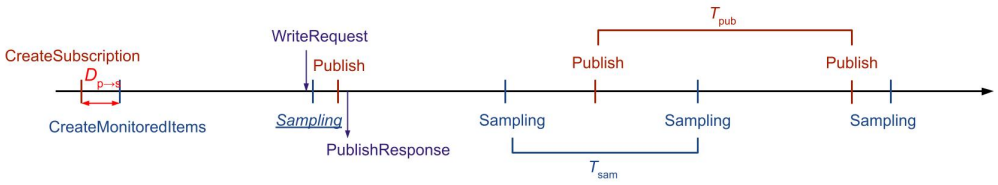

Figura 5.7. Caso quando la latenza D_{srv} si avvicina al minimo (sampling e publishing non sincronizzati

Capitolo 6

Ridondanza dei broker

6.1 Introduzione

In quest'ultimo capitolo viene descritta la realizzazione di un sistema di comunicazione resiliente, basato sul protocollo MQTT, che sfrutta un approccio simile al protocollo PRP (Parallel Redundancy Protocol). Il funzionamento base del protocollo PRP è quello di spedire lo stesso pacchetto su più percorsi diversi, in modo tale che il destinatario possa ricevere almeno una copia del messaggio e, in caso di ricezioni multiple, scartare i duplicati. Questo protocollo è solitamente utilizzato nelle reti Ethernet, in cui non c'è un meccanismo che garantisca l'avvenuta ricezione, come, per esempio, nei livelli applicativi basati su TCP. Inoltre viene gestito in modo ottimale anche il caso in cui si verificasse un guasto, in quel caso i percorsi dovranno essere ricalcolati e bisognerà attendere che il protocollo *Spanning Tree* finisca di calcolare i nuovi percorsi, prima di poter riprendere la comunicazione, con

il PRP invece, non si dovrà attendere. Il punto di forza di questo protocollo è la resilienza, infatti, anche se dovessero verificarsi problemi su uno dei percorsi, gli altri continuerebbero a funzionare (i percorsi devono essere il più possibile indipendenti). In questo modo diminuisce anche la probabilità di non ricevere il dato inviato, in quanto, una tale eventualità, significherebbe un guasto su tutti i percorsi ridondati utilizzati e nella stessa finestra temporale. Un altro possibile vantaggio è l'avere una latenza di sistema con valori massimi migliori, rispetto ai singoli percorsi, dato che il destinatario elabora sempre il pacchetto che per primo giunge a destinazione.
Per questa parte della tesi è stato realizzato un sistema che usa l'approccio del protocollo PRP, ma sulla rete Internet, usando il protocollo MQTT. Anche se non si ha il controllo della rete, e quindi non è possibile scegliere il percorso che faranno i nostri dati una volta inviati usando l'*Internet Service Provider*, con MQTT è tuttavia possibile veicolare parte dell'instradamento tramite il broker: visto che non si ha una connessione end-to-end tra publisher e subscriber, ma in mezzo ai due c'è un broker che fa da intermediario. Per creare un sistema abbastanza esteso e volendo tenere i publisher e subscriber sulla stessa macchina (per elaborare in modo semplice, e con la stessa base dei tempi, le latenze) situata in Italia, i broker sono stati posizionati in paesi europei diversi, cercando di sceglierli in modo da avere latenze simili. Le città scelte, usando quelle disponibili sulla piattaforma Amazon Web Services, in cui posizionare i broker sono state Londra e Stoccolma, con il publisher e il subscriber a Torino, per una serie di misure, e a Milano, per un altro esperimento. In Figura 6.1 è possibile osservare lo schema di

funzionamento di questa implementazione con due broker. Si vede come il publisher manderà nello stesso istante un dato su due percorsi diversi, ma che alla fine dovranno giungere al subscriber, sempre posizionato sulla stessa macchina (reale o virtuale) su cui si trova il publisher stesso. Il compito del filtro sarà quello di scartare pacchetti già ricevuti.

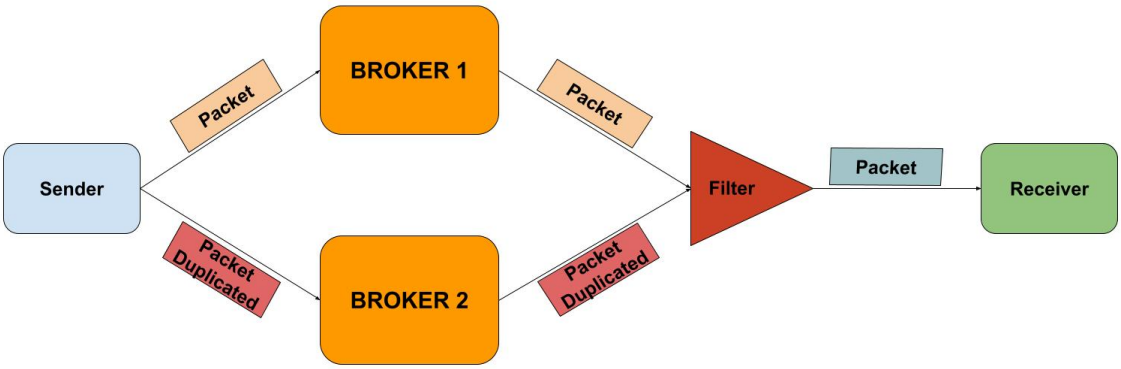

Figura 6.1. Sistema ridondato

6.2 Publisher e Subscriber

Per la realizzazione del sistema, si sono realizzati due programmi in Python, un publisher e un subscriber, entrambi fanno uso di una classe chiamata `multi_mqtt`, realizzata appositamente per gestire la connessione ridondata. La classe `multi_mqtt` si basa sulla libreria paho-mqtt [18], la quale viene solitamente utilizzata per realizzare una singola connessione MQTT con il broker. Di seguito è mostrato il codice del costruttore della classe `multi_mqtt`.

```
def __init__(self, broker, _callback=0):
    self.lock = threading.Lock()
```

```
self.clients = broker
self.nclients = len(self.clients)
self.ID_c = str(uuid.uuid4())
self.multi_conne()
self.tocall = _callback
self.seq = 1
self.seq_max = 2**32
```

Il costruttore accetta due paramatri, `broker` e `_callback`: il primo contiene la lista dei broker con i quali bisogna instaurare la connessione, mentre il secondo, `_callback`, è la funzione che verrà richiamata ogni volta che un pacchetto verrà ricevuto. Nello specifico, quest'ultima verrà chiamata solo con i nuovi pacchetti, e non con i duplicati, inoltre, nel caso si volesse solo realizzare un publisher, e quindi trasmettere dati senza riceverli, è possibile inserire nel costruttore solo il primo parametro, che contiene la lista dei broker, senza necessariamente inserire il secondo paramento, cioè la funzione che verrebbe chiamata in caso di ricezione.
Nella variabile `self.ID_c`, si salva un identificativo univoco, generato dalla libreria *uuid* [19]. La variabile `self.seq`, invece, è un contatore che viene posto sui pacchetti trasmessi per identificarli.
Per gestire le sezioni critiche del codice si è utilizzato un *lock*, che verrà usato dai vari thread. Sempre nel costruttore viene chiamato il metodo `multi_conne()`, che si occuperà di realizzare le connessioni con i broker scelti. Di seguito è riportato il codice.

```
def multi_conne(self):
```

```
mqtt.Client.connected_flag = False
for i in range(self.nclients):
    cname = self.ID_c + "_connection" + str(i)
    client = mqtt.Client(cname,False)
    self.clients[i]["client"] = client
    self.clients[i]["connection_id"] = cname
    broker = self.clients[i]["broker"]
    port = self.clients[i]["port"]
    client.on_connect = self.on_connect
    client.on_disconnect = self.on_disconnect
    client.on_message = self.on_message
    try:
        client.connect(broker, port)
    except:
        print("Connection failed to broker ", broker)
        continue
    client.loop_start()
    while not client.connected_flag:
        time.sleep(0.05)
self.th = threading.Thread(target=self.clean)
self.th2 = threading.Thread(target=self.clean_2)
self.th.start()
self.th2.start()
```

Come si vede dal codice, esattamente nella prima riga, viene aggiunta alla

classe Client di paho-mqtt un flag, chiamato `connected_flag` e inizializzato a *False*. Tale flag servirà per sapere lo stato di ciascuna connessione.
All'interno del ciclo `for` si andrà a creare una nuova istanza della classe Client di paho-mqtt; in ogni Client viene abilitata la *Persistent Session*, e il numero di Client istanziati sarà uguale al numero di broker a cui ci si vuole connettere. Inoltre, per ogni Client, si registrano tre callback:

- `on_connect`
- `on_message`
- `on_disconnect`

La prima verrà chiamata ogniqualvolta il Client si connette al broker: in questo modo verrà chiamata la callback `on_connect`, che aggiornerà la stato del flag `connected_flag`, impostandolo su *True*. Procedura analoga con `on_disconnect`, tuttavia, in questo caso, verrà impostato un timer nel quale ritentare la connessione al broker. Questo punto è particolarmente importante perché gestisce i tentativi di riconnessione ai broker che possono essere diventati temporaneamente non raggiungibili. Se dopo una serie di timeout, finalmente, la connessione tornerà attiva, verrà chiamata `on_connect` che cambierà lo stato di `connected_flag` a *True*, rendendo di nuovo disponibile quel broker per la comunicazione. Di seguito è riportata la funzione `on_disconnect`.

```
def on_disconnect(self, client):
        print("Disconnected")
        client.connected_flag = False
```

```
client.reconnect_delay_set(min_delay=5, max_delay=6)
```

L'ultima callback, `on_message`, è quella che verrà chiamata ogni volta che si riceve un pacchetto da uno dei broker.
Infine, viene chiamato il metodo `loop_start()`, per ciascun Client: chiamando tale metodo si creerà un thread che avrà il compito di gestire il traffico di quel determinato Client, oltre ad occuparsi della riconnessione.
Terminato il ciclo `for`, vengono avviati due thread, il cui compito è la gestione della struttura dati, nella quale si tiene traccia dei pacchetti ricevuti non ancora elaborati, che verrà spiegata più avanti.
Una volta finita la fase di connessione, il Client potrà decidere, tramite le funzioni `multi_pub` e `multi_sub`, se trasmettere dati, e quindi essere un publisher, oppure, riceverli ed essere un subscriber. Lo stesso programma Python, quindi, può essere sia publisher, sia subscriber, tuttavia, ai fini della valutazione di prestazioni del sistema ridondato, si è preferito mandare in esecuzione i programmi su due processi diversi, due programmi distinti, che si comportano o solo da publisher o solo da subscriber. Per questo motivo verrà illustrato il codice del publisher e del subscriber separatamente nelle seguenti sezioni.

Publisher

In questa sezione si approfondirà il funzionamento del publisher, con l'uso della classe `multi_mqtt`. Di seguito è riportato il codice usato per i test di valutazione, che fanno uso della classe `multi_mqtt`.

```
broker = [{"broker": "18.134.154.212", "port": 1883, "qos": 0},
```

```
            {"broker": "13.53.188.114", "port": 1883, "qos": 0}]
n = multi_mqtt(broker)
conta = 1
conta_max=8400
print(n.ID_c)
while conta<=conta_max:
    print(conta)
    n.multi_pub("test/broker_ridondato", "Hello "+str(conta))
    conta = conta+1
    time.sleep(0.5)
n.multi_disc()
print("finished")
```

Nella variabile `broker` si salveranno gli indirizzi IP, le porte e la QoS scelta di ciascun broker a cui ci si vuole connettere. Una volta chiamato il costruttore, che è già stato descritto in precedenza, si potrà iniziare a trasmettere usando la funzione `multi_pub`: tale funzione necessita del topic e del payload, che, in questo caso, è semplicemente una stringa con la parola "Hello" concatenata ad un intero crescente. La funzione `multi_pub` dovrà spedire a tutti broker connessi lo stesso payload. Di seguito è riportato il metodo `multi_pub` nel dettaglio.

```
def multi_pub(self, pub_topic, msg):
    for i in range(self.nclients):
        js = {"message": msg,
              "seq":self.seq,
```

```
                "id_client":self.ID_c,
                "connection":i}
            to_send = json.dumps(js)
            client = self.clients[i]["client"]
            Qos = self.clients[i]["qos"]
            if client.connected_flag:
                client.publish(pub_topic, to_send, qos=Qos)
        self.seq=(self.seq+1)%(self.seq_max)
```

All'interno di `multi_pub` avremo un ciclo che spedirà, ad ogni broker, un pacchetto JSON. Il pacchetto JSON conterrà i seguenti valori:

- **message**, il messaggio che si vuole spedire
- **seq**, un *sequence number*
- **Client_ID**, un codice che identifica univocamente il Client
- **connection**, il numero della connessione, che identifica a quale broker verrà spedito il messaggio

Una volta preparato, il pacchetto JSON verrà spedito al broker a cui quel Client è connesso. Se il broker non dovesse essere connesso, allora il pacchetto non verrà spedito: ciò si può verificare semplicemente dal flag `connected_flag`. Terminato il ciclo `for`, e, quindi, conclusa la fase di invio su tutti i broker, si incrementa la variabile `self.seq` e si va avanti.

Subscriber

In questa parte verrà illustrato più nel dettaglio il funzionamento del subscriber, il quale non dovrà solo ricevere i pacchetti provenienti dal publisher, ma dovrà anche filtrare i pacchetti duplicati, quindi scartando quelli ridondati provenienti dagli altri broker. Di seguito è riportato il codice del subscriber realizzato.

```
def my_callback(client, userdata, message):
    print(message)


broker = [{"broker": "18.134.154.212", "port": 1883, "qos": 0},
          {"broker": "13.53.188.114", "port": 1883, "qos": 0}]
n = multi_mqtt(broker,my_callback)
n.multi_sub("test/broker_ridondato")
try:
    while True:
        pass
except:
    n.multi_disc()
    print("finished")
```

Il codice è molto simile a quello del publisher, tuttavia qui si ha qualche aggiunta. Prima di tutto, si può notare che nel costruttore `multi_mqtt` si va ad aggiungere la funzione `my_callback`. Tale funzione verrà richiamata ogni volta che un nuovo pacchetto giungerà al subscriber, e sarà compito della classe `multi_mqtt` occuparsi di filtrare i duplicati. Con la funzione

`multi_sub`, si andrà a richiedere la sottoscrizione dello stesso topic, a tutti i broker. Fatto ciò, si è pronti a ricevere i pacchetti dai broker.
Come visto in precedenza, il costruttore, dopo aver istanziato un Client, crea un thread, il quale gestirà la connessione. Quando un pacchetto giungerà al subscriber, il thread di quel Client, dovrà chiamare il metodo `on_message`: tale funzione si occupa di decomporre il pacchetto appena ricevuto, verificare se è un duplicato o no e, in caso non lo fosse, chiamare la callback dichiarata dallo user nel costruttore. Di seguito è riportato il codice di `on_message`.

```
def on_message(self, client, userdata, message):
    self.lock.acquire()
    timestamp_receiver=time.time_ns()
    msg_json = str(message.payload.decode("utf-8"))
    y = json.loads(msg_json)
    topic=message.topic
    msg = y["message"]
    id_p = int(str(y["seq"]))
    client_id=y["id_client"]
    client_id_conn=str(y["connection"])
    if topic in self.storage.keys():
        Clients_json=self.storage[topic]
    else:
        Clients_json={}
        self.storage[topic]=Clients_json
    if client_id in Clients_json.keys():
        pack=Clients_json[client_id]
    else:
        pack={"waiting_for":id_p}
        Clients_json[client_id]=pack
    if id_p in pack.keys():
        lista=pack[id_p].keys()
        nuova_lista = [item for item in lista if type(item)==int]
        numero_attuale=max(nuova_lista)
        numero_attuale=numero_attuale+1
        pack[id_p][numero_attuale]=timestamp_receiver
    else:
        if id_p>=self.storage[topic][client_id]["waiting_for"]:
            pack[id_p]={1:timestamp_receiver,
                        "payload":msg}
        else:
            pack[id_p]={1:timestamp_receiver}
    if self.storage[topic][client_id]["waiting_for"]==id_p:
```

```
            self.tocall(self.storage[topic][client_id][id_p]["payload"])
            tmp=(int(self.storage[topic][client_id]["waiting_for"])+1)%self.seq_max
            self.storage[topic][client_id]["waiting_for"]=tmp
            del self.storage[topic][client_id][id_p]["payload"]
            try:
                cerca=self.storage[topic][client_id]["waiting_for"]
                for x in range(cerca,self.seq_max-cerca):
                    if topic in self.storage.keys() and
                                client_id in self.storage[topic].keys() and
                                x in self.storage[topic][client_id].keys():
                        self.tocall(self.storage[topic][client_id][x]["payload"])
                        tmp_2=(int(self.storage[topic][client_id]["waiting_for"])+1)%self.seq_max
                        self.storage[topic][client_id]["waiting_for"]=tmp_2
                        del self.storage[topic][client_id][x]["payload"]
                    else:
                        break
            except:
             pass
        self.lock.release()
```

La prima operazione eseguita è prendere possesso del *lock*, visto che la funzione potrebbe essere chiamata dagli altri thread. Dopodiché, si estraggono le informazioni contenute nel pacchetto, cioè: *message*, Client_ID e *sequence number*, inoltre, si tiene traccia del timestamp di ricezione del pacchetto.

I pacchetti spediti dovranno essere elaborati dal subscriber con il medesimo ordine, rispettando quindi il *sequence number*, tuttavia, in caso di eventuali pacchetti giunti in anticipo, non potranno essere elaborati, ma nemmeno scartati, se non sono già stati elaborati, quindi, si dovrà tenerli in memoria, in attesa che arrivino quelli precedenti, cioè con il *sequence number* minore. Per tenere traccia di questi pacchetti si è utilizzata la struttura dati mostrata in Figura 6.2, tale struttura è realizzata mediante dizionari annidati, strutture fornite dal linguaggio Python. Prima di tutto, si controlla se il topic del pacchetto appena ricevuto è già presente nel dizionario dei topic,

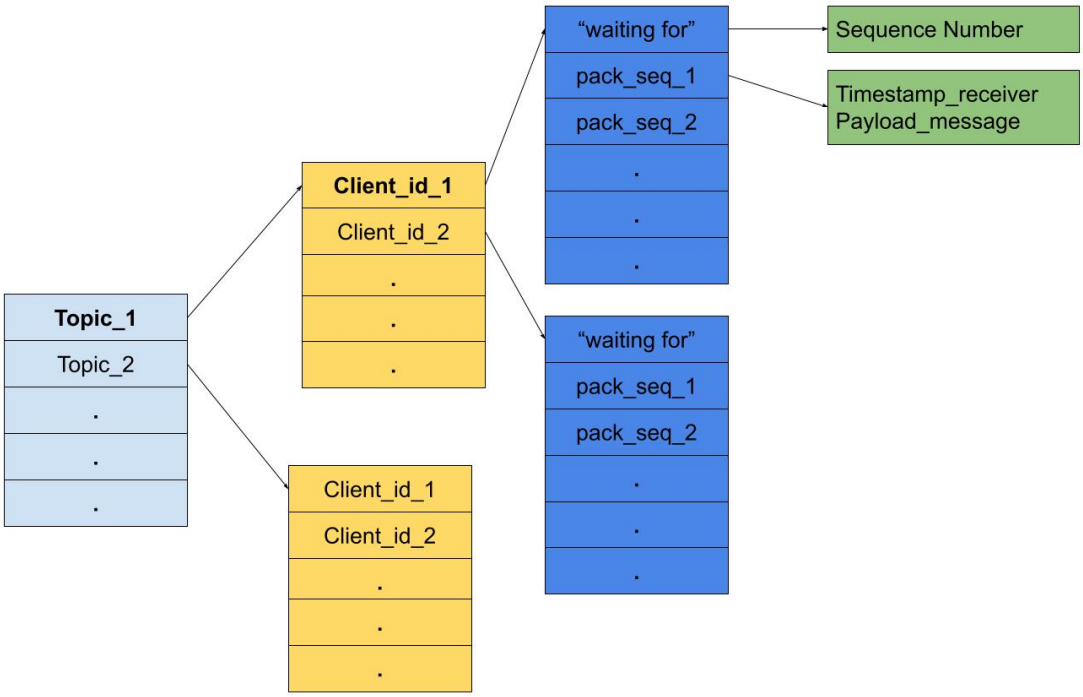

Figura 6.2. Salvataggio dei pacchetti ricevuti

in caso non lo fosse, verrà aggiunto; stessa operazione viene eseguita per il valore Client_ID. A questo punto, per ogni Client_ID, si terrà in memoria, sotto il nome di "waiting for", il *sequence number* del pacchetto che dovrà essere elaborato non appena arriva il suo turno, cioè, quello che si sta aspettando da quel determinato Client_ID. Se il *sequence number* appena ricevuto è maggiore di quello che si sta aspettando, allora si salva insieme al suo timestamp, se invece fosse un valore minore, e quindi che si è già elaborato in precedenza, verrà scartato. Solo quando giunge il pacchetto che si sta aspettando verrà chiamata la callback definita dallo user. Alla fine si dovrà rilasciare il *lock*. A questo punto i casi possibili per il pacchetto appena ricevuto sono tre:

- **seq=waiting_for**, ho ricevuto il pacchetto che aspettavo, elaboro
- **seq<waiting_for**, pacchetto già ricevuto, scarto
- **seq>waiting_for**, pacchetto non ancora elaborato, ma non quello che aspettavo, salvo

Ricapitolando: nel caso di pacchetto duplicato, questo viene semplicemente scartato, mentre, nel caso di pacchetto giunto in anticipo, viene salvato.
Per la gestione della struttura dati, vengono eseguiti dei thread che si occupato di ripulirla (in caso di eventuali campi vuoti) e di controllare da quanto tempo sono in attesa i pacchetti salvati. Se ci dovessero essere eventuali pacchetti salvati, vuol dire che il pacchetto che si stava aspettando, da un determinato Client, è in ritardo o, peggio, perso. Se dopo un certo lasso di tempo, che il pacchetto salvato è in attesa, non dovesse arrivare il pacchetto atteso, allora verrà considerato definitivamente perso e verrà elaborato quello in memoria.
Questa mansione è affidata al thread `clean_2`, che si occupa di controllare se sono presenti eventuali pacchetti in attesa. Nello specifico, il thread verifica da quanto tempo ogni pacchetto sta aspettando in memoria. Se il tempo d'attesa dovesse essere maggiore di 60 secondi (valore pre-impostato nel codice mostrato), il pacchetto in attesa verrà elaborato e il *sequence number*, del pacchetto atteso verrà cambiato.
Di seguito il codice del thread `clean_2`.

```
def clean_2(self):
    while(True):
        self.lock.acquire()
        riferimento = int(time.time_ns())
        confronto = 1*60*1*1000000000
```

```
for topic in  self.storage.keys():
    for client in self.storage[topic].keys():
        for seq_packet in sorted ([item for item in self.storage[topic][client].keys()
                                    if type(item)==int and
                                    item > self.storage[topic][client]["waiting_for"] ]):
            time_waiting=self.storage[topic][client][seq_packet][1]
            flag=0
            if (riferimento-time_waiting)>confronto:
                self.storage[topic][client]["waiting_for"]=seq_packet
                flag=1
            if seq_packet==self.storage[topic][client]["waiting_for"]:
                self.tocall(self.storage[topic][client][seq_packet]["payload"])
                tmp=(int(self.storage[topic][client]["waiting_for"])+1)%self.seq_max
                self.storage[topic][client]["waiting_for"]=tmp
                del self.storage[topic][client][seq_packet]["payload"]
                flag=1
            if flag==0:
                break
self.lock.release()
time.sleep(30)
```

Nel dettaglio, si può vedere come il thread `clean_2` scansiona l'intera struttura dati per verificare se sono presenti pacchetti salvati. Per ogni topic, Client_id e *sequence number*, il thread controlla, in ordine di *sequence number* da quanto tempo è in attesa quel pacchetto. Se il tempo è maggiore di 60 secondi, si chiamerà la callback predisposta e si cancellerà quel pacchetto dalla struttura dati, incrementando il valore di "waiting_for", e continuando a cercare se sono presenti altri pacchetti da elaborare.

L'altro thread, `clean` viene chiamato meno frequentemente, visto che si occupa semplicemente di cancellare eventuali dati inutilizzati nella struttura: ad esempio, potrebbe capitare che un certo Client o un topic non vengano più utilizzati. Il payload dei pacchetti viene cancellato appena questi sono stati elaborati, ma potrebbe rimanere ad esempio un Client_ID senza contenuto o un topic non più utilizzato, quindi, in quel caso, si devono cancellare quelle informazioni per mantenere pulita la struttura dati. Di seguito

il codice del thread `clean`

```
def clean(self):
    self.testora=0
    while True:
        self.lock.acquire()
        riferimento = int(time.time_ns())
        confronto = 1*60*60*1000000000
        ora=time.time_ns()
        for topic in  self.storage.keys():
            if len(self.storage[topic])==0:
                    del self.storage[topic]
            else:
                for client in self.storage[topic].keys():
                    if len(self.storage[topic][client])==0:
                        del self.storage[topic][client]
                    else:
                        for seq_packet in sorted ([item for item in self.storage[topic][client].keys()
                                                                    if type(item)==int]):
                            time_waiting=self.storage[topic][client][seq_packet][1]
                            if (riferimento-time_waiting) > confronto and
                            self.storage[topic][client]["waiting_for"]!=seq_packet:
                                    del self.storage[topic][client][seq_packet]
                            else:
                                break
        dopo=time.time_ns()
        if (dopo-ora)>self.testora:
            self.testora=(dopo-ora)
        self.lock.release()
        time.sleep(60*60)
```

6.3 Setup sperimentale

Il publisher si connette a entrambi i broker, Londra e Stoccolma, dopodiché invia, con un intervallo di 500 ms, un pacchetto su entrambi i broker. Anche il subscriber si connette a entrambi i broker ed esegue la sottoscrizione su entrambi, in modo da ricevere i dati trasmessi dal publisher; in questo modo il subscriber riceverà da entrambi i broker lo stesso pacchetto, ma accetterà

solo il primo scartando il secondo che giungerà a destinazione, come illustrato nella Figura 6.3.

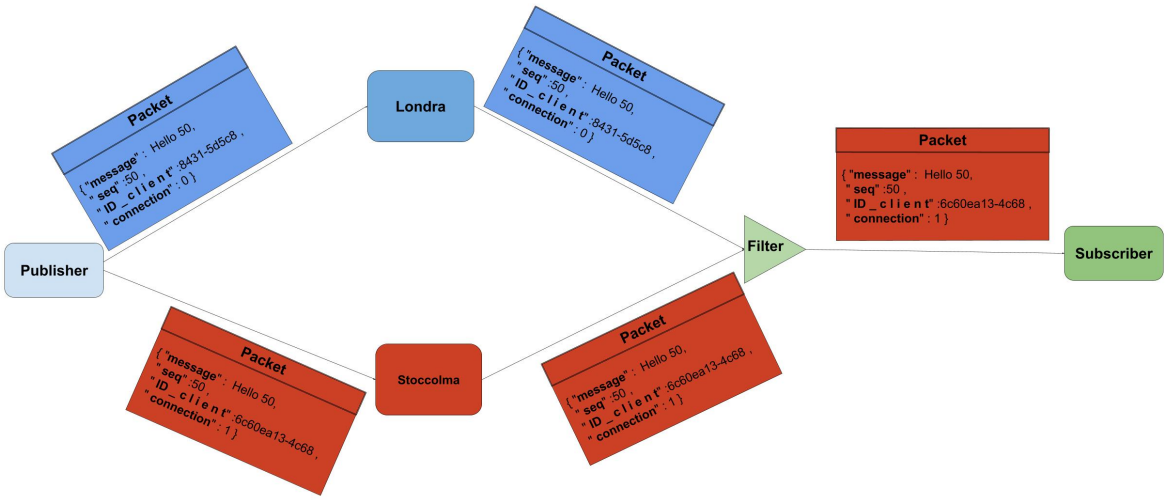

Figura 6.3. Sistema con broker a Londra e Stoccolma

Sia il publisher, sia il subscriber sono stati mandati in esecuzione, impostando il QoS=0.

Per poter usufruire di broker posizionati in diverse città in Europa, come accennato precedentemente, si è utilizzato il servizio offerto da Amazon chiamato AWS (Amazon Web Services) [20]. Tramite questo servizio è possibile realizzare macchine virtuali situate in diverse città nel mondo. Il sistema operativo scelto per le macchine virtuali è stato Ubuntu Server 20.04 LTS. Una volta create le macchine virtuali su Londra e Stoccolma si è installato il sistema operativo scelto, dopodiché si è installato il broker mosquitto, per l'utilizzo di MQTT, ed infine tcpdump per la cattura dei pacchetti in entrata e uscita dalle macchine virtuali, in modo da poter valutare le latenze dovute al broker. Finito di preparare i broker di Londra e Stoccolma si è

passati al Client. Per quest'ultimo si è utilizzata una Raspberry 4 posizionata a Torino, in cui si sono mandati in esecuzione entrambi i programmi in Python, sia il publisher, sia il subscriber; per la cattura dei pacchetti si è utilizzato, come nei due broker, tcpdump, in Figura 6.4 è possibile vedere la posizione dei vari componenti.

Per una prima valutazione, che ha riguardato solo le prestazioni del sistema in condizioni normali, cioè senza simulare guasti, si sono spediti un totale di 8400 pacchetti. In Tabella 6.1 sono riportate le statistiche di questa prima valutazione. Lo stesso esperimento, sempre con 8400 pacchetti, è stato ripetuto, ma simulando un guasto sul broker di Londra. Il guasto è stato simulato spegnendo per 10 minuti il broker mosquitto, per poi riaccenderlo. Nella Tabella 6.2 sono mostrate le statistiche.

Un'ultima misurazione è stata fatta cambiando la posizione del Client: nello specifico, si è scelto di usare una macchina virtuale, anch'essa creata tramite il servizio AWS come nel caso dei broker, con lo stesso sistema operativo e situata a Milano. Da quest'ultima si sono spediti in totale 18000 pacchetti, inoltre, si è simulata una serie di guasti, sulla macchina di Londra, in cui, ogni 30 minuti, si spegneva il broker per la durata di 10 minuti. Nella Tabella 6.3 è possibile vedere le statistiche di quest'ultima valutazione, mentre in Figura 6.5 si vede, per ogni pacchetto, il tempo impiegato da ogni pacchetto giunto al sistema, e la media mobile con finestra pari a 100 campioni.

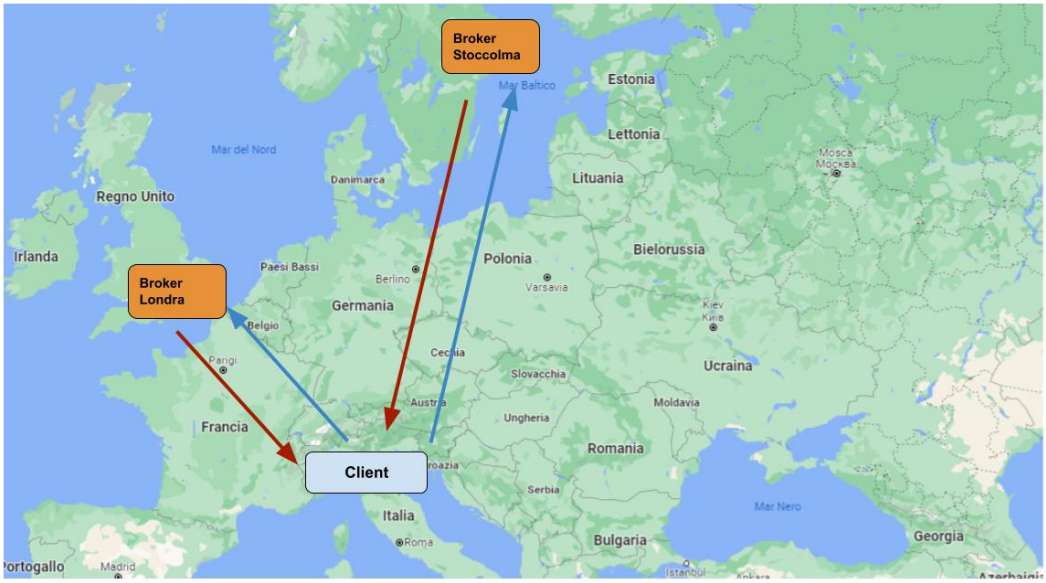

Figura 6.4. Rappresentazione geografica della posizione dei broker

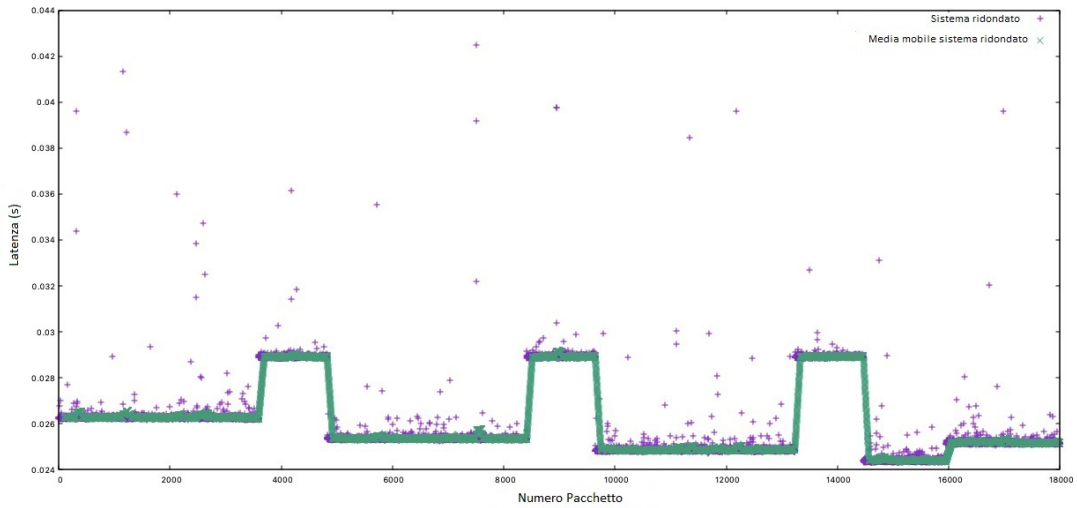

Figura 6.5. Simulazione di tre guasti con Client a Milano e due broker: Londra e Stoccolma

6.4 Risultati

Nella Tabella 6.1 vengono presentati alcuni dei risultati ottenuti con il primo esperimento, quello in cui non è stato spento nessun broker. Com'è possibile vedere, a volte si verificano ritardi nelle due connessioni viste singolarmente, infatti i tempi massimi sono alti, tuttavia, nel sistema ridondato, il quale prende il messaggio che arriva per primo dalle due connessioni, ha un tempo massimo inferiore rispetto alle connessioni di Londra e Stoccolma. Un altro miglioramento è possibile vederlo nella Deviazione Standard.
Nella Tabella 6.2 si vede come il sistema continui ad avere delle buone prestazioni, anche con il broker di Londra che viene disattivato per 10 minuti. Un'ulteriore osservazione può essere fatta considerando che le latenze minori sono, in media, quelle che si ottengono usando la connessione con il broker di Londra: la maggior parte dei pacchetti che raggiungono per primi il sistema sono quelli da Londra. Da qui si può vedere come questo abbia influito, anche se di poco, sulla media. Nella Tabella 6.3, in cui si è posizionato il Client a Milano, si vedono latenze più basse: ciò è dovuto principalmente a una connessione migliore offerta per i sistemi di Amazon, inoltre, anche con una serie di guasti, il sistema ha continuato a funzionare bene, senza mostrare dei tempi di latenza massima eccessivi.

#packets 8400	Sistema (Torino)	Londra	Stoccolma
Minimo (ms)	35.186	35.186	40.511
Media (ms)	36.789	37.304	42.049
Massimo (ms)	44.362	1082.369	291.855
Percentile 90	37.648	37.639	42.466
Percentile 95	39.962	39.944	42.599
Percentile 99	41.927	45.471	43.352
Dev.Stand.	1.256	14.298	8.087
Ricevuto (%)	100	100	100

Tabella 6.1. Sistema

#packets 8400	Sistema (Torino)	Londra	Stoccolma
Minimo (ms)	33.569	33.569	40.005
Media (ms)	36.148	35.201	41.259
Massimo (ms)	44.491	282.881	284.975
Percentile 90	41.172	35.874	41.929
Percentile 95	41.674	36.062	42.065
Percentile 99	42.087	36.796	42.479
Dev.Stand.	2.351	3.011	2.725
Ricevuto (%)	100	84.25	100

Tabella 6.2. Guasto simulato su Londra

#packets 18000	Sistema (Milano)	Londra	Stoccolma
Minimo (ms)	24.307	24.307	28.820
Media (ms)	26.097	25.365	28.923
Massimo (ms)	42.503	42.573	56.398
Percentile 90	28.903	26.274	28.970
Percentile 95	28.941	26.296	28.999
Percentile 99	29.019	26.403	29.137
Dev.Stand.	1.594	0.759	0.418
Ricevuto (%)	100	79.46	100

Tabella 6.3. Sistema Milano con 3 guasti

Capitolo 7

Conclusione e futuri sviluppi

Nella prima valutazione sperimentale, in cui si sono valutati alcuni aspetti dell'uso combinato di OPC UA e MQTT, è stato possibile osservare le prestazioni di questa nuova specifica aggiunta in OPC UA, che è stata introdotta ancora in versione preliminare nella libreria open62541. Nonostante il fatto che sia ancora un "working in progress", la parte attualmente implementata mostra delle tempistiche con QoS=0 e QoS=1 decisamente buone, discorso diverso per il QoS=2 dove, a causa di tempi di timeout interni al Server OPC UA, nei quali attende eventuali dati in ricezione, la comunicazione del publisher ne risente, portando dei ritardi aggiuntivi.
Nella seconda valutazione sperimentale sono state effettuate alcune misurazioni e considerazioni per modellare accuratamente il ritardo di elaborazione

interno di tre popolari Server OPC UA, analizzando e spiegando il loro peculiare comportamento. Anche se questa può sembrare un'analisi molto specifica, il punto è che, a volte, i dettagli di implementazione (dovuti, ad esempio, alla negoziazione dei parametri per la connessione) potrebbero influenzare, da un punto di vista quantitativo, le prestazioni di protocolli con comportamenti ben noti. Questo è particolarmente vero, dalle valutazione effettuate, per la libreria open-source open62541.

Nell'ultimo capitolo, a differenza dei primi due, si è usato principalmente il protocollo MQTT, dove si è realizzato un sistema ridondato e in grado di fornire un comportamento resiliente. Come si è potuto osservare, il sistema mostra delle statistiche migliori, anche se non di molto, rispetto alle singole connessioni. Ma garantisce, nel caso si verifichino guasti su una delle connessioni, che il sistema continui a funzionare usando le altre. Questo porta anche ad avere tempi massimi, a livello di sistema, decisamente più bassi rispetto alle connessioni singole. È interessante notare come il sistema riesca a mantenere un tempo massimo basso anche durante i guasti, ciò è dovuto al fatto che il percorso ridondato viene utilizzato sempre, non solo in caso di guasti. In questo modo, se un percorso dovesse rompersi, non ci sono tempi di setup per una connessione alternativa, quindi non si hanno tempi di latenza alti per quel pacchetto in corso di invio, poiché sta già viaggiando sul percorso parallelo.

Sviluppi futuri di questo lavoro di tesi sui server studiati per il protocollo OPC UA potrebbero riguardare un miglioramento di alcune scelte implementative della libreria open62541.

Per quello che riguarda l'architettura ridondata per MQTT, un possibile miglioramento potrebbe essere quello di aggiungere meccanismi di resilienza anche a livello del singolo broker.

Bibliografia

[1] *OPC-UA*. https://opcfoundation.org/about/opc-technologies/opc-ua/. Accessed: 2021-07-16.

[2] *MQTT*. https://mqtt.org/. Accessed: 2021-06-16.

[3] *Publish/Subscribe*. https://it.wikipedia.org/wiki/Publish/subscribe. Accessed: 2021-07-16.

[4] *Unified Automation*. https://www.unified-automation.com/. Accessed: 2021-07-17.

[5] *FreeOpcUa*. https://github.com/FreeOpcUa/python-opcua. Accessed: 2021-07-17.

[6] *open62541*. https://open62541.org/. Accessed: 2021-07-17.

[7] Hubert Kirrmann, Mats Hansson e Peter Muri. «IEC 62439 PRP: Bumpless Recovery for Highly Available, Hard Real-Time Industrial Networks». In: *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)*. 2007, pp. 1396–1399. DOI: 10.1109/EFTA.2007.4416946.

[8] *OPC-Foundation*. https://opcfoundation.org/. Accessed: 2021-07-17.

[9] *OPC Data Access*. https://opcfoundation.org/developer-tools/specifications-classic/data-access/. Accessed: 2021-07-16.

[10] *OPC*. https://opcfoundation.org/about/what-is-opc/. Accessed: 2021-07-16.

[11] *OPC UA specifications and information models*. https://reference.opcfoundation.org/. Accessed: 2021-07-17.

[12] *OPC-UA-Cap-14*. https://reference.opcfoundation.org/v104/Core/docs/Part14/. Accessed: 2021-07-16.

[13] *node-opcua*. https://node-opcua.github.io/. Accessed: 2021-07-16.

[14] *cluster mqtt*. https://www.hivemq.com/blog/clustering-mqtt-introduction-benefits/. Accessed: 2021-07-16.

[15] *mosquitto*. https://mosquitto.org/. Accessed: 2021-07-16.

[16] *tcpdump*. https://www.tcpdump.org/. Accessed: 2021-07-16.

[17] *Network Time Protocol*. http://www.ntp.org/. Accessed: 2021-07-16.

[18] *paho-MQTT*. https://pypi.org/project/paho-mqtt/. Accessed: 2021-07-16.

[19] *uuid*. https://docs.python.org/3/library/uuid.html. Accessed: 2021-07-16.

[20] *AWS-Amazon*. https://aws.amazon.com/it/. Accessed: 2021-07-16.