Università degli studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Sviluppo di un sistema IoT per trasmissione dati da sensori in tempo reale

Tesi di Laurea Triennale

Relatore

Prof. Marco Zanella

Laureando

Matteo Squarzoni

Matricola 2068240

ANNO ACCADEMICO 2024-2025

© Matteo Squarzoni, Luglio 2025. Tutti i diritti riservati. Tesi di Laurea Triennale: "*Sviluppo di un sistema IoT per trasmissione dati da sensori in tempo reale*", Università degli studi di Padova, Dipartimento di Matematica "Tullio Levi-Civita".

Come raggiungere un traguardo? Senza fretta ma senza sosta.

— Johann Wolfgang von Goethe

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Marco Zanella relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare i miei genitori che mi hanno sempre sostenuto e che hanno sempre creduto in me. Senza il loro supporto non sarei mai riuscito a raggiungere questo traguardo.

Inoltre ringrazio Andrea, compagno di avventure e studi da molti anni, per il supporto e la compagnia durante il percorso di studi.

Infine un ringraziamento a tutti i componenti del gruppo Tech Minds per il corso di Ingegneria del software, con cui ho condiviso dei bei momenti da ricordare.

Padova, Luglio 2025 *Matteo Squarzoni*

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, dal laureando Matteo Squarzoni presso l'azienda M31 Srl.
Il lavoro di stage ha avuto come obiettivo lo sviluppo di un sistema di trasmissione dati in tempo reale da sensori distribuiti, da ottenere grazie alla realizzazione di un'architettura efficiente e scalabile che garantisca bassa latenza, alta affidabilità e precisione nella trasmissione e gestione dei dati raccolti.

Indice

1 Introduzione **1**
1.1 L'azienda 1
1.2 L'idea 2
1.3 Organizzazione del testo 2

2 Descrizione dello stage **5**
2.1 Rapporto con l'azienda 5
2.2 Analisi preventiva dei rischi 5
2.3 Principali obiettivi 6
2.4 Pianificazione 7

3 Analisi dei requisiti **9**
3.1 Fase iniziale 9
3.2 Tracciamento dei requisiti 9

4 Protocolli e tecnologie **13**
4.1 Introduzione 13
4.2 MQTT 13
4.2.1 Architettura Publish/Subscribe 14
4.2.2 Client e Broker 15
4.2.3 Pacchetti MQTT 16
4.2.4 QoS 17
4.2.5 Altre funzionalità 18
4.3 CoAP 18
4.3.1 Caratteristiche principali 19
4.3.2 Formato dei messaggi 19
4.3.3 Altre funzionalità 21

4.3.3.1 Aspetti positivi ... 22
4.4 HTTP ... 22
4.4.1 Caratteristiche principali ... 23
4.4.2 Versioni di HTTP ... 23
4.5 Casi d'uso reali per l'utilizzo di MQTT, CoAP e HTTP ... 24
4.6 Confronto tra i protocolli ... 25
4.6.1 Performance dei protocolli ... 26
4.7 Serializzazione dei dati ... 27
4.8 Aspetti di sicurezza ... 28
4.9 Scelte progettuali ... 29

5 Architettura del sistema ... 31
5.1 Progettazione ad alto livello ... 31
5.2 Tecnologie e strumenti ... 33
5.2.1 Go ... 33
5.2.2 Docker ... 34
5.2.3 Altri strumenti ... 35
5.3 Progettazione di dettaglio ... 36
5.3.1 Dispositivi IoT ... 36
5.3.2 Server CoAP ... 36
5.3.3 Database ... 37
5.3.4 API HTTP ... 38
5.3.5 Dashboard ... 39

6 Sviluppo del prototipo ... 41
6.1 Fasi iniziali ... 41
6.1.1 Serializzazione dei dati ... 41
6.1.2 Compressione dei dati ... 43
6.2 Dati trasmessi ... 45
6.3 Dispositivo IoT ... 47
6.4 Server CoAP ... 49
6.4.1 Connessione DTLS ... 51
6.4.2 Validazione dei dati ... 52
6.4.3 Business logic ... 52

6.5 API HTTP 53
6.5.1 Comandi 54
6.5.2 Raccolta di dati aggregati 55
6.5.3 Business logic 55
6.6 Database 57
6.7 Dashboard 57
6.8 Testing 58
6.9 Limiti e criticità della soluzione 59

7 Conclusioni 61
7.1 Raggiungimento degli obiettivi 61
7.2 Margini di miglioramento 62
7.3 Conoscenze acquisite 63
7.4 Valutazione personale 63

Bibliografia i

Elenco delle figure

Figura 1 Logo di M31 Srl 1
Figura 2 Esempio di architettura Publish/Subscribe 14
Figura 3 Struttura di un pacchetto MQTT 16
Figura 4 Scambio di pacchetti con QoS 0 17
Figura 5 Scambio di pacchetti con QoS 1 17
Figura 6 Scambio di pacchetti con QoS 2 18
Figura 7 Struttura di un messaggio CoAP 20
Figura 8 Architettura del sistema 32
Figura 9 Diagramma delle classi del server CoAP 36
Figura 10 Schema del database 38
Figura 11 Diagramma delle classi dell'API HTTP 38
Figura 12 Tempo di serializzazione dei dati in diversi formati 42
Figura 13 Tempo di deserializzazione dei dati in diversi formati 43
Figura 14 Tempo di compressione con diversi algoritmi 44
Figura 15 Tempo di decompressione con diversi algoritmi 45
Figura 16 Dashboard di Grafana per la visualizzazione dei dati 58

Elenco delle tabelle

Tabella 1 Tabella pianificazione del lavoro 7
Tabella 2 Tabella del tracciamento dei requisti funzionali 10
Tabella 3 Tabella del tracciamento dei requisti qualitativi 11

Tabella 4 Tabella del tracciamento dei requisti di vincolo 11
Tabella 5 Tabella di confronto tra i protocolli MQTT, CoAP e HTTP . . 25
Tabella 6 Tabella dei packages di Go utilizzati 33
Tabella 7 Tabella delle immagini Docker utilizzate 35
Tabella 8 Tabella degli strumenti utilizzati 35
Tabella 9 Tabella di confronto della dimensione dei dati serializzati in diversi formati 42
Tabella 10 Tabella di confronto della dimensione dei dati compressi con diversi algoritmi 44
Tabella 11 Tabella dello stato dei requisti funzionali 61
Tabella 12 Tabella dello stato dei requisti qualitativi 62
Tabella 13 Tabella dello stato dei requisti di vincolo 62

Elenco dei codici sorgenti

Codice 1 Strutture dei dati biomedici trasmessi dal dispositivo IoT 45
Codice 2 Struttura dello StreamingManager del dispositivo IoT 47
Codice 3 Processo di streaming del dispositivo IoT 48
Codice 4 Inizializzazione del server CoAP 50
Codice 5 Connessione DTLS tra il server e i dispositivi IoT 51
Codice 6 Metadati per la validazione dei dati biomedici 52
Codice 7 Validazione dei dati biomedici ricevuti dal server 53
Codice 8 Esempio di richiesta per avviare lo streaming dei dati 54
Codice 9 Gestione della richiesta per l'inserimento di un comando 56
Codice 10 Query per la raccolta dei dati aggregati per l'endpoint maxmin_history 57
Codice 11 Esempio di test dell'API HTTP con Hurl 58

Acronimi e abbreviazioni

***CoAP* – Constrained Application Protocol.** 6, 7, 9, 18, 31, 33, 49

***DTLS* – Datagram Transport Layer Security.** 28, 33, 51

***HTTP* – Hypertext Transfer Protocol.** 22

***IoT* – Internet of Things.** 6, 9, 13, 31, 33, 41

***MQTT* – Message Queuing Telemetry Transport.** 6, 7, 13

***PSK* – Pre-Shared Key.** 51

***QoS* – Quality of Service.** 17

***TCP* – Transmission Control Protocol.** 14

***TLS* – Transport Layer Security.** 28

***UDP* – User Datagram Protocol.** 9, 22, 33, 51

Glossario

***CoAP* – Constrained Application Protocol**: Protocollo di comunicazione progettato per dispositivi a bassa potenza e reti a bassa larghezza di banda. Utilizza un modello di richiesta/risposta simile a HTTP, ma ottimizzato per l'uso in ambienti con risorse limitate. 13

***DTLS* – Datagram Transport Layer Security**: Estensione del protocollo TLS per la sicurezza delle comunicazioni su UDP. Fornisce autenticazione, integrità e riservatezza dei dati in ambienti con risorse limitate. 28

***HTTP* – Hypertext Transfer Protocol**: Protocollo di comunicazione utilizzato per il trasferimento di dati su reti informatiche, in particolare per il web. È un protocollo basato su richiesta/risposta che consente la trasmissione di documenti ipertestuali. 13

***IoT* – Internet of Things**: Rete di oggetti dotati di tecnologie di identificazione, collegati fra di loro, in grado di comunicare sia reciprocamente sia verso punti nodali del sistema, ma soprattutto in grado di costituire un enorme network di cose dove ognuna di esse è rintracciabile per nome e in riferimento alla posizione. 3, 13

***MQTT* – Message Queuing Telemetry Transport**: Protocollo di messaggistica leggero e standardizzato, progettato per la comunicazione tra dispositivi a bassa larghezza di banda e alta latenza. Utilizza un modello di pubblicazione/sottoscrizione per la trasmissione dei dati. 13

***PSK* – Pre-Shared Key**: Metodo di autenticazione in cui una chiave segreta condivisa viene utilizzata per autenticare le comunicazioni tra due parti. È spesso utilizzato in scenari con risorse limitate, come dispositivi IoT. 51

***QoS* – Quality of Service**: Concetto che si riferisce alla garanzia di prestazioni specifiche in una rete, come la latenza, la larghezza di banda e l'affidabilità. In MQTT, QoS determina il livello di affidabilità nella consegna dei messaggi. 15

***TCP* – Transmission Control Protocol**: Protocollo di trasmissione dati orientato alla connessione, che garantisce l'ordine e l'affidabilità nella consegna dei pacchetti. È ampiamente utilizzato per applicazioni

che richiedono una comunicazione affidabile, come il web browsing e il trasferimento file. 13

***TLS* – Transport Layer Security**: Protocollo di sicurezza che fornisce autenticazione, integrità e riservatezza dei dati nelle comunicazioni su reti TCP/IP. È ampiamente utilizzato per proteggere le comunicazioni su Internet. 28

***UDP* – User Datagram Protocol**: Protocollo di trasmissione dati senza connessione, utilizzato per applicazioni che richiedono velocità e bassa latenza, come lo streaming video o le chiamate VoIP. Non garantisce l'ordine o la consegna dei pacchetti. 19

Capitolo 1

Introduzione

1.1 L'azienda

M31 Srl [1] nasce nel 2007 ad opera di un gruppo di imprenditori, ricercatori universitari e professionisti appassionati di tecnologia, coordinati da Ruggero Frezza. La sua sede è a Padova, in via Niccolò Tommaseo 77, a pochi passi dai dipartimenti di ingegneria, fisica e matematica dell'Università di Padova. Sin dalla sua costituzione, il cuore di M31 è il suo team di ingegneria. Nei più di quindici anni di operatività, la Società ha svolto più di cento progetti di innovazione a vantaggio di startup proprie, imprese del territorio e grandi gruppi. Recentenmente M31 è diventato partner tecnologico di SMACT, uno degli 8 Centri di Competenza industria 4.0 nati in Italia su impulso del Ministero dello Sviluppo Economico, a Padova. La visione dell'azienda è quella di affiancare i propri clienti nel realizzare le loro idee di innovazione mediante l'applicazione di tecnologie allo stato dell'arte e degli output della ricerca universitaria. Tutto questo incarna il motto di M31: «With You. Beyond.»

With You. Beyond.

Figura 1: Logo di M31 Srl

1.2 L'idea

L'idea di questo progetto nasce dalla volontà di realizzare un sistema di trasmissione dati in tempo reale che elabori in modo efficiente e intelligente i dati provenienti da sensori distribuiti, garantendo al contempo bassa latenza, alta accuratezza e scalabilità. Con l'aumento del numero di dispositivi connessi, il sistema dovrà essere in grado di ottimizzare il flusso di dati, ridurre al minimo i ritardi e mantenere l'integrità dei dati, permettendo un monitoraggio e un'analisi in tempo reale senza interruzioni. Attraverso l'implementazione di meccanismi di trasmissione robusti e affidabili, l'obiettivo è quello di fornire una soluzione veloce, sicura e scalabile che soddisfi le esigenze delle applicazioni critiche.

1.3 Organizzazione del testo

Il secondo capitolo descrive l'organizzazione dello stage andando a specificare il rapporto con l'azienda e il tutor aziendale, la pianificazione del lavoro, i principali obiettivi della soluzione e un'analisi dei rischi preventiva.

Il terzo capitolo descrive in maniera dettagliata l'analisi dei requisiti.

Il quarto capitolo descrive le tecnologie esistenti per risolvere il problema, gli aspetti teorici alla base e gli strumenti scelti.

Il quinto capitolo presenta l'architettura del sistema, con una panoramica delle tecnologie e degli strumenti utilizzati, e una descrizione delle fasi di progettazione ad alto livello e di dettaglio.

Il sesto capitolo descrive lo sviluppo del prototipo incluse le fasi iniziali, la codifica e il testing.

Il settimo capitolo descrive le considerazioni finali, il raggiungimento degli obiettivi, le conoscenze acquisite e alcuni possibili margini di miglioramento.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato all'inizio del presente documento;

- per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *Internet of Things*$_G$;
- i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Descrizione dello stage

2.1 Rapporto con l'azienda

Durante il periodo di stage ho avuto modo di interagire con i membri del team di lavoro, in particolare con il tutor aziendale, quest'ultimo è stato sempre disponibile e pronto a rispondere ad ogni domanda o dubbio avessi riguardo al progetto e per facilitare le comunicazioni è stato aperto un canale diretto su piattaforma Microsoft Teams. Ogni settimana ho aggiornato il tutor riguardo il lavoro svolto, discutendo e pianificando le attività successive. Ho trovato molto utile il suo supporto, in quanto mi ha permesso di comprendere meglio le dinamiche lavorative e le aspettative dell'azienda.

2.2 Analisi preventiva dei rischi

Durante la fase di analisi iniziale sono stati individuati alcuni possibili rischi a cui si potrà andare incontro. Si è quindi proceduto ad elaborare delle possibili soluzioni per far fronte a tali rischi.

1. Apprendimento nuove tecnologie

Descrizione: il progetto prevede l'uso di tecnologie e strumenti che non sono stati precedentemente utilizzati, questo potrebbe portare a ritardi nella pianificazione.

Soluzione: dedicare del tempo all'apprendimento e alla comprensione delle tecnologie e degli strumenti necessari per il progetto. Utilizzare risorse online e documentazione ufficiale per acquisire le competenze necessarie. Chiedere supporto al tutor aziendale e ai membri del team per chiarire eventuali dubbi.

2. Architettura poco scalabile

Descrizione: l'architettura del sistema potrebbe non essere sufficientemente scalabile per gestire un numero crescente di dispositivi e flussi di dati.

Soluzione: progettare un'architettura modulare e scalabile fin dall'inizio, tenendo conto delle esigenze future. Utilizzare tecnologie e protocolli che supportano la scalabilità, come *MQTT* o *CoAP*. Eseguire test di carico per valutare le prestazioni del sistema in scenari di alta domanda.

3. Problemi legati all'integrità dei dati

Descrizione: la trasmissione dei dati in tempo reale potrebbe essere soggetta a errori o perdite di pacchetti, compromettendo l'integrità dei dati.

Soluzione: implementare meccanismi di controllo degli errori e ritrasmissione dei pacchetti. Utilizzare tecniche di aggregazione e filtraggio dei dati per ridurre il carico di trasmissione e migliorare l'affidabilità. Eseguire test approfonditi per identificare e risolvere eventuali problemi di integrità dei dati.

4. Elevata latenza

Descrizione: la latenza nella trasmissione dei dati potrebbe essere superiore alle aspettative, compromettendo le prestazioni del sistema.

Soluzione: ottimizzare il flusso di dati e le strategie di trasmissione per ridurre la latenza. Utilizzare protocolli leggeri e tecniche di compressione dei dati per migliorare le prestazioni. Eseguire test di latenza in scenari reali per identificare eventuali colli di bottiglia e ottimizzare il sistema.

5. Scadenze e pianificazione

Descrizione: il progetto potrebbe subire ritardi a causa di imprevisti o difficoltà tecniche.

Soluzione: pianificare le attività in modo realistico, tenendo conto delle possibili difficoltà. Monitorare costantemente i progressi e adattare la pianificazione se necessario. Comunicare tempestivamente eventuali ritardi al tutor aziendale per ripianificare le attività.

2.3 Principali obiettivi

Con il rapido miglioramento dei dispositivi *IoT* in settori come l'automazione industriale e la sanità, la domanda di soluzioni di trasmissione dati a bassa latenza e ad alto *throughput* è cresciuta significativamente. I sistemi esistenti spesso affrontano limitazioni chiave, tra cui:

Elevata latenza: i ritardi nella trasmissione di dati critici possono compromettere il processo decisionale in tempo reale e le risposte automatizzate, causando potenzialmente conseguenze irreversibili. È quindi necessario implementare un meccanismo che riduca al minimo i ritardi di trasmissione, consentendo la disponibilità dei dati in tempo reale per l'analisi e il processo decisionale.

Sovraccarico di dati e inefficienza: flussi di dati non ottimizzati portano ad un inutile consumo di banda e ad un sovraccarico di elaborazione. È necessario ottimizzare il flusso di informazioni riducendo i dati ridondanti o a bassa priorità, prevenendo così la congestione della rete. Il punto chiave è sviluppare strategie robuste per garantire che i dati dei sensori rimangano accurati e affidabili durante tutto il processo di trasmissione.

Vincoli di scalabilità: molte soluzioni attuali faticano a scalare in modo efficiente, influenzando l'affidabilità del sistema con l'aumentare del numero di sensori. È necessario progettare un'architettura scalabile in grado di gestire grandi volumi di dati e un numero crescente di dispositivi connessi senza compromettere le prestazioni.

Un sistema di trasmissione dati in tempo reale ben strutturato è essenziale per superare queste sfide, garantendo alta disponibilità, reattività in tempo reale e gestione efficiente del flusso di dati.

2.4 Pianificazione

Settimana	Ore	Attività
1	40	Introduzione al progetto e contesto applicativo. Studio e analisi comparativa dei protocolli di trasmissione dati in tempo reale (*MQTT*, *CoAP*, *WebSocket*). Introduzione al linguaggio *Go* e alle sue caratteristiche principali.
2	40	Continuazione dello studio dei protocolli. Prima stesura analisi dei requisiti funzionali e non funzionali. Benchmarking e comparazione di vari formati dati (*JSON*, *MsgPack*, *CBOR*).

3	40	Completamento dell'analisi dei requisiti. Progettazione dell'architettura del sistema e definizione dei moduli principali. Setup dell'ambiente di sviluppo. Inizio di implementazione di un PoC (*Proof of Concept*) per la trasmissione dati in tempo reale utilizzando il protocollo *CoAP*.
4	32	Terminazione dello sviluppo del PoC. Comunicazione tra dispositivi e server. Integrazione di un database per la memorizzazione dei dati. Gestione di varie modalità di trasmissione dati attraverso comandi impartiti grazie allo sviluppo di un *API HTTP*.
5	32	Implementazione dei server *CoAP* e *HTTP* con un architettura esagonale. Sviluppo di un sistema di gestione dei flussi di dati. Setup di un *docker compose* per fare l'hosting dei servizi esterni.
6	40	Integrazione di una dashboard per la visualizzazione dei dati in tempo reale. Sviluppo di rotte per l'*API HTTP* per il recupero di dati. Inizio dei test di unità.
7	40	Miglioramenti e ottimizzazione della trasmissione. Integrazione di un sistema di backup e successiva sincronizzazione nel caso il server dovesse essere offline. Completamento dei test di unità.
8	40	Stesura della documentazione tecnica del sistema, manuale utente e guida all'installazione. Preparazione della presentazione finale e revisione completa del progetto.

Tabella 1: Tabella pianificazione del lavoro

Capitolo 3

Analisi dei requisiti

3.1 Fase iniziale

In seguito ad uno studio preliminare dei principali protocolli di comunicazione e delle tecnologie disponibili, si è deciso di utilizzare il protocollo *CoAP* per la trasmissione dei dati tra i dispositivi *IoT* e il server di raccolta dati. La scelta è stata dettata dal fatto che i sensori da simulare sono dispositivi a bassa potenza di calcolo e con risorse estremamente limitate che sfruttano canali di comunicazione *UDP*. Nei prossimi capitoli verrà approfondito il funzionamento dei vari protocolli analizzati andando a fare delle comparazioni tra di essi al fine di dare un'introduzione al contesto applicativo, inoltre verranno discussi altri aspetti da tenere in considerazione quando si parla di trasmissione dati.

3.2 Tracciamento dei requisiti

Sono stati individuati diversi tipi di requisiti e si è quindi fatto utilizzo di un codice identificativo per distinguerli.

Il codice dei requisiti è così strutturato R(F/Q/V)(N/D/O) dove:

R = requisito

F = funzionale

Q = qualitativo

V = di vincolo

N = obbligatorio (necessario)

D = desiderabile

Z = opzionale

Nelle tabelle Tabella 2, Tabella 3 e Tabella 4 sono riassunti i requisiti delineati in fase di analisi.

Requisito	Descrizione
RFN-1	Il client deve simulare il comportamento di un dispositivo *IoT* indossabile con sensori.
RFN-2	Il client deve poter inviare dati periodici al server avviando una simulazione realistica.
RFN-3	Il server deve poter processare i dati in ingresso solamente dai client autorizzati.
RFN-4	Il client deve essere notificato di un eventuale errore di autenticazione da parte dal server nel caso non abbia i permessi di autorizzazione.
RFN-5	Il server deve poter effettuare una validazione dei dati in ingresso.
RFN-6	Il server deve poter salvare i dati ricevuti dai client in un database.
RFN-7	La configurazione di client e server deve essere gestita tramite appositi file di configurazione.
RFN-8	Il sistema deve prevedere la serializzazione/deserializzazione dati per la comunicazione in un formato condiviso.
RFN-9	Permettere l'invio di comandi impartiti attraverso degli *endpoint* forniti da un'*API HTTP*, per la gestione delle modalità di streaming del dispositivo client.
RFD-1	Il sistema deve prevedere un sistema di cifratura per proteggere la trasmissione dati.
RFD-2	L'*API HTTP* deve mettere a disposizione uno o più *endpoint* che permettano di ottenere dati aggregati a partire dai dati grezzi raccolti dal sistema.
RFD-3	Gestione del processo di *backup* e successiva sincronizzazione per evitare la perdita delle misurazioni che il client rileva nei periodi in cui il server non è raggiungibile.
RFZ-1	Gestione delle conversioni tra unità di misura per il salvataggio dei dati in maniera uniforme.
RFZ-2	L'utente deve essere in grado di visualizzare i dati inviati dai client e raccolti dal server in una dashboard.

Tabella 2: Tabella del tracciamento dei requisti funzionali

Requisito	Descrizione
RQN-1	Utilizzo del formato di dati *CBOR* per contenere la dimensione dei messaggi.
RQD-1	Condurre *benchmark* riguardo l'aspetto tecnologico della soluzione adottata e delle possibile alternative riguardo latenza, serializzazione e compressione dei dati.
RQD-2	Produrre *unit testing* per l'intero prodotto fornendo una *coverage* pari almeno al 75%.
RQD-3	Prevedere metodi di compressione dati per l'invio *batch* durante la procedura di sincronizzazione.

Tabella 3: Tabella del tracciamento dei requisti qualitativi

Requisito	Descrizione
RVN-1	Il sistema deve sfruttare il protocollo *CoAP* su *UDP* per la comunicazione.
RVN-2	Salvataggio dati su un database per permetterne estrazione ed eventuale analisi.
RVN-3	Utilizzo del formato dati *CBOR* per le capacità di calcolo limitate dei dispositivi simulati.
RVD-1	Il codice deve essere prodotto in linguaggio *Go*.
RVD-2	Utilizzare *TimescaleDB* per gestire l'aggregazione dati e l'eventuale integrazione con *Grafana*.
RVD-3	Utilizzo di *DTLS* con chiave *PSK* per le capacità di calcolo limitate dei dispositivi simulati.

Tabella 4: Tabella del tracciamento dei requisti di vincolo

Capitolo 4

Protocolli e tecnologie

In questo capitolo viene fornita una panoramica teorica delle tecnologie e dei protocolli considerati per lo sviluppo del progetto. Si tratta di un'introduzione al contesto applicativo che prepara il terreno per la comprensione delle implementazioni pratiche descritte nei capitoli successivi.

4.1 Introduzione

Esistono diversi protocolli di comunicazione per i dispositivi dell'*Internet of Things (IoT)*$_G$, ognuno con le proprie caratteristiche e vantaggi. In questa sezione, verranno presentati i protocolli più rilevanti per il progetto, con un focus particolare su *Constrained Application Protocol (CoAP)*$_G$, *Message Queuing Telemetry Transport (MQTT)*$_G$ e *Hypertext Transfer Protocol (HTTP)*$_G$. Si discuterà anche dell'importanza della serializzazione dei dati e delle tecnologie di cifratura per garantire la sicurezza e l'efficienza nella trasmissione dei dati. Sebbene *MQTT* possa essere considerato lo *standard de facto* per la comunicazione tra dispositivi *IoT*, altri protocolli come *CoAP* e *HTTP* sono valide alternative con i loro vantaggi. La scelta del protocollo dipende dalle specifiche esigenze del progetto, come la larghezza di banda disponibile, la latenza richiesta e le risorse hardware dei dispositivi coinvolti.

4.2 MQTT

MQTT [2] è un protocollo di trasporto per messaggi che utilizza un'architettura *publish/subscribe*. È leggero, aperto, semplice e progettato per essere facile da implementare. Queste caratteristiche lo rendono ideale per l'uso in molte situazioni, inclusi ambienti vincolati come la comunicazione *Machine to Machine* (M2M) e contesti *IoT* dove la larghezza di banda di rete è limitata. Utilizza una comunicazione basata sul *Transmission Control Protocol (TCP)*$_G$.

È bene ricordare che essendo il protocollo *MQTT* basato sul *TCP/IP*, significa che il client e il *broker* devono avere un *TCP/IP stack* per instaurare correttamente le connessioni.

4.2.1 Architettura Publish/Subscribe

MQTT è un protocollo che segue l'architettura *publish/subscribe*, ed utilizza un modello *"broker-based"* dove i clients si connettono ad un *broker* e i messaggi sono pubblicati su determinati *"topics"*. Dunque i *"subscribers"* possono iscriversi ad uno specifico topic e ricevere i messaggi pubblicati per quest'ultimo. Nella Figura 2 è riportato un esempio di architettura *publish/subscribe*.

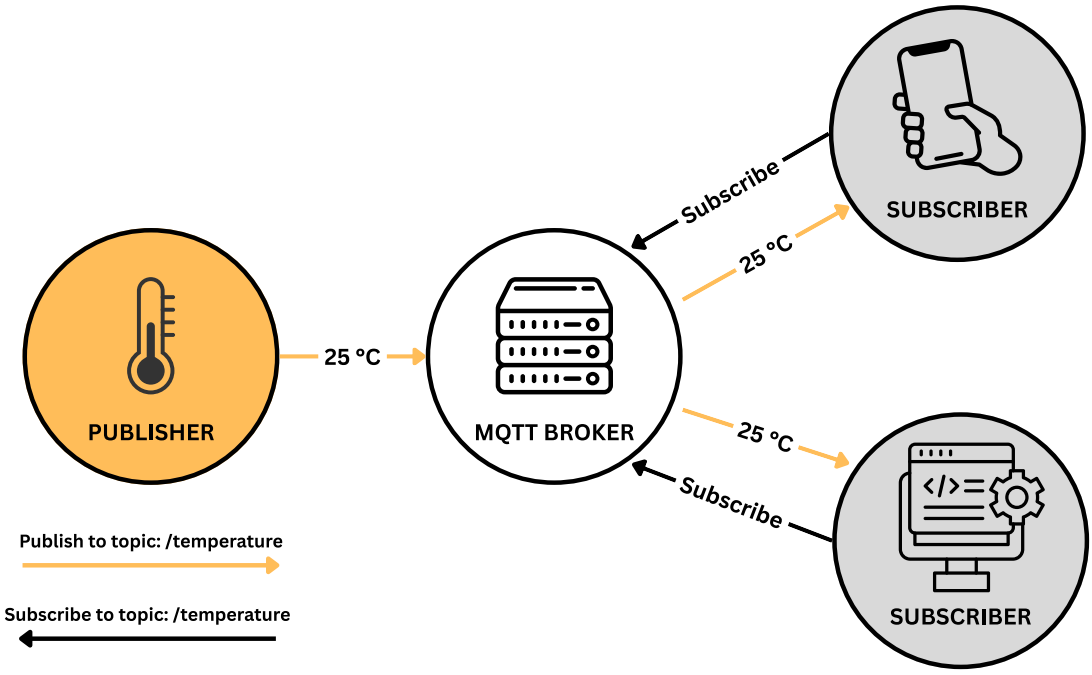

Figura 2: Esempio di architettura Publish/Subscribe

Rispetto ad un tradizionale modello *client-server* che utilizza l'approccio *"request-response"*, dove il client comunica direttamente con il *server endpoint* rischiando di creare un *bottleneck* che può rallentare le performance, il *pub/sub model* riduce l'accoppiamento tra i *publishers* e i *subscribers*, infatti essi non sono a conoscenza dell'esistenza degli uni e degli altri e viceversa. Inoltre è presente un *broker* che mantiene le connessioni tra loro.

Di seguito sono riportati alcuni benefici dell'architettura *pub/sub*:

- **Scalabilità migliorata**: l'architettura *pub/sub* è altamente scalabile, il che la rende adatta ad applicazioni che gestiscono numerosi client e messaggi. Il *broker* agisce come un *hub* centrale per tutti i messaggi, consentendo di gestire molti client senza compromettere le prestazioni.
- **Maggiore tolleranza ai guasti**: la natura disaccoppiata dell'architettura *pub/sub* garantisce anche una migliore tolleranza ai guasti. In un modello tradizionale *client-server*, tutti i client perdono la connessione se il *server* si arresta. Al contrario, nel modello *pub/sub*, il *broker* può memorizzare i messaggi fino a quando il client non si riconnette, assicurando che nessun messaggio venga perso.
- **Flessibilità**: l'architettura *pub/sub* è flessibile e può essere utilizzata in una vasta gamma di applicazioni, che spaziano da reti a bassa larghezza di banda e alta latenza fino a reti ad alta velocità e bassa latenza. Il protocollo *MQTT*, inoltre, supporta diversi livelli di *Quality of Service (QoS)*$_G$, offrendo la possibilità di scegliere il livello più adatto per la propria applicazione.

4.2.2 Client e Broker

Un **client *MQTT*** può essere qualsiasi dispositivo da un piccolo microcontrollore a grandi server. I client si riferiscono ai *publishers* e *subscribers*. Un *publisher* è un client che invia i messaggi mentre un *subscriber* è un client che riceve i messaggi. Tuttavia un client può essere allo stesso tempo sia un *publisher* che un *subscriber*.

Un **broker *MQTT*** è un *hub* centrale che riceve e smista i messaggi. Permette di connettere e gestire anche milioni di client concorrenti connessi. Il *broker* filtra i messaggi determinando quali client dovrebbero ricevere il messaggio. Autentica e autorizza i client in base alle credenziali immesse. In aggiunta alcuni *broker* supportano il *"clustering"* che permette a più istanze di un *broker* di lavorare insieme per gestire alti numeri di client e messaggi.

Le connessioni avvengono sempre tra client e *broker*, infatti i client non si connettono mai direttamente tra di loro.

4.2.3 Pacchetti MQTT

In *MQTT* i pacchetti sono le unità di dati scambiate tra i client e il *broker*. Ogni pacchetto ha un formato specifico, come illustrato in Figura 3, che include un header e un payload. L'header contiene informazioni come il tipo di pacchetto, la lunghezza del payload e altre opzioni. Il payload contiene i dati reali che vengono scambiati.

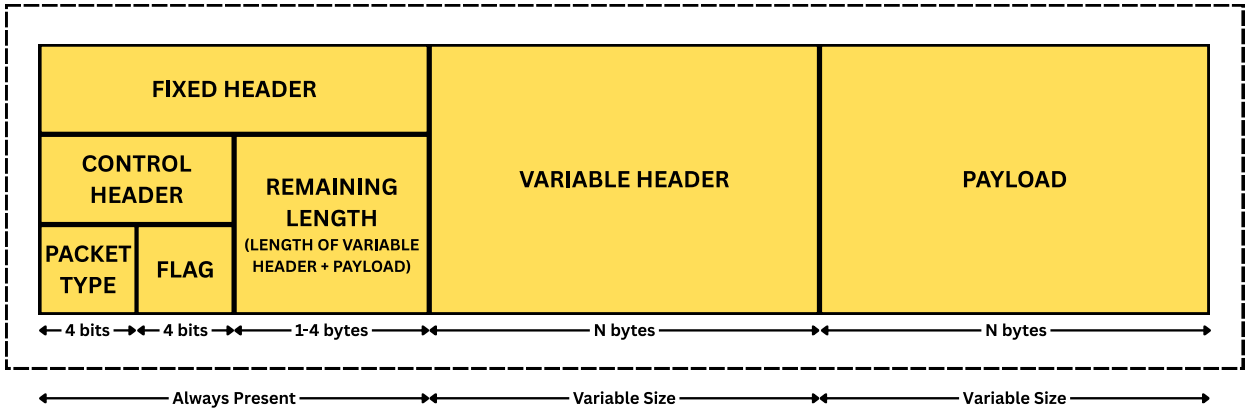

Figura 3: Struttura di un pacchetto MQTT

Esistono dunque diversi tipi di messaggi *MQTT* che possono essere scambiati tra client e *broker*. I pacchetti sono identificati da un tipo di messaggio specifico, che determina la loro funzione. Di seguito sono riportati i tipi di pacchetti più comuni in *MQTT*:

- **CONNECT**: messaggio inviato da un client al *broker* per stabilire una connessione.
- **CONNACK**: risposta del *broker* al pacchetto CONNECT, che indica se la connessione è stata accettata o meno.
- **PUBLISH**: pubblicazione di un messaggio da parte di un client su un determinato topic.
- **SUBSCRIBE**: iscrizione di un client a uno o più topic per ricevere i messaggi pubblicati su di essi.
- **SUBACK**: risposta del *broker* al pacchetto SUBSCRIBE, che conferma l'iscrizione a uno o più topic.
- **UNSUBSCRIBE**: annulla l'iscrizione di un client da uno o più topic.
- **UNSUBACK**: risposta del *broker* al pacchetto UNSUBSCRIBE, che conferma l'annullamento dell'iscrizione.

4.2.4 QoS

MQTT supporta tre livelli di qualità del servizio *QoS* per garantire la consegna dei messaggi:

- **QoS 0**: consegna del messaggio al massimo una volta, senza alcuna garanzia di consegna. Questo livello è il più veloce e consuma meno risorse, ma non garantisce che il messaggio venga ricevuto. Lo scambio di pacchetti con *QoS* 0 è illustrato in Figura 4.

Figura 4: Scambio di pacchetti con QoS 0

- **QoS 1**: consegna del messaggio almeno una volta, garantendo che il messaggio venga ricevuto dal destinatario. Questo livello può causare la consegna duplicata dei messaggi, ma garantisce che il messaggio venga ricevuto almeno una volta. Lo scambio di pacchetti con *QoS* 1 è illustrato in Figura 5.

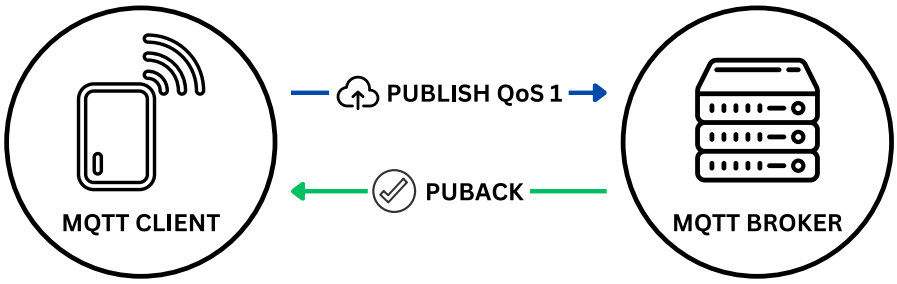

Figura 5: Scambio di pacchetti con QoS 1

- **QoS 2**: consegna del messaggio esattamente una volta, garantendo che il messaggio venga ricevuto senza duplicati. Questo livello è il più lento e consuma più risorse, ma garantisce la consegna esatta del messaggio. Lo scambio di pacchetti con *QoS* 2 è illustrato in Figura 6.

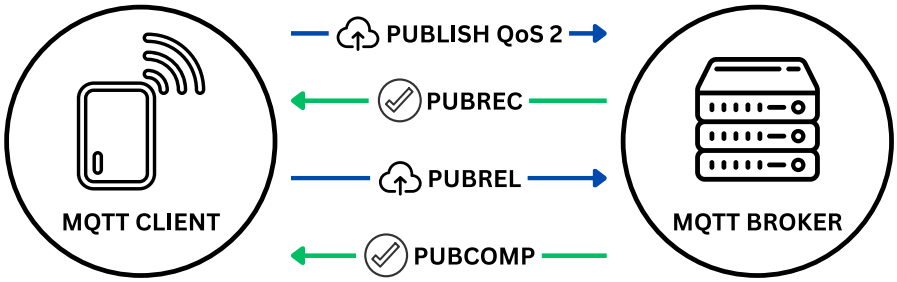

Figura 6: Scambio di pacchetti con QoS 2

4.2.5 Altre funzionalità

MQTT offre anche altre funzionalità utili per la comunicazione tra dispositivi *IoT*:

- **Last Will and Testament (LWT)**: consente a un client di specificare un messaggio che il *broker* invierà a un topic specifico se il client si disconnette in modo imprevisto. Questo è utile per notificare agli altri client della disconnessione di un dispositivo.
- **Retained Messages**: consente al *broker* di memorizzare l'ultimo messaggio pubblicato su un topic, in modo che i nuovi *subscribers* possano ricevere immediatamente l'ultimo stato del topic al momento della sottoscrizione.
- **Keep Alive**: è un meccanismo che consente ai client di mantenere attiva la connessione con il *broker* inviando periodicamente pacchetti di *ping*. Questo aiuta a rilevare le disconnessioni in modo tempestivo e a mantenere la connessione attiva.
- **Session Persistence**: consente ai client di mantenere le loro sottoscrizioni e lo stato della sessione anche dopo una disconnessione. Questo è utile per garantire che i messaggi non vengano persi durante le interruzioni temporanee della connessione.

4.3 CoAP

CoAP [3], [4] è un protocollo di comunicazione progettato specificamente per dispositivi con risorse limitate in ambienti *IoT*. Il protocollo è stato sviluppato per consentire a dispositivi a basso consumo energetico di partecipare efficacemente alle reti *IoT* mantenendo un *footprint* ridotto in termini di memoria e potenza di elaborazione.

4.3.1 Caratteristiche principali

CoAP opera sul protocollo di comunicazione *User Datagram Protocol (UDP)*$_G$ e implementa un modello di comunicazione *request-response* tra gli endpoint, facilitando l'interazione tra diverse tipologie di dispositivi. Le caratteristiche distintive del protocollo includono:

- **Architettura RESTful**: *CoAP* adotta un'architettura *RESTful* che garantisce elevata interoperabilità tra dispositivi eterogenei. Questa scelta progettuale semplifica lo sviluppo di applicazioni, poiché gli sviluppatori possono utilizzare i metodi standard già familiari da *HTTP* (GET, POST, PUT, DELETE) per interagire con le risorse.
- **Supporto multicast**: il protocollo offre capacità di comunicazione *multicast*, permettendo di inviare richieste a più dispositivi simultaneamente. Questa funzionalità è particolarmente vantaggiosa in scenari dove è necessario interrogare o controllare gruppi di dispositivi con una singola operazione.
- **Efficienza energetica**: *CoAP* è ottimizzato per ridurre il consumo energetico dei dispositivi, caratteristica fondamentale per nodi alimentati a batteria che devono operare per periodi prolungati senza manutenzione.
- **Leggerezza computazionale**: il protocollo richiede risorse computazionali minime, rendendolo adatto per microcontrollori con capacità di elaborazione e memoria limitata.

4.3.2 Formato dei messaggi

Il più piccolo messaggio *CoAP* è composto solamente da un header di 4 byte, che include informazioni essenziali come il tipo di messaggio, il codice di richiesta/risposta e l'identificatore del messaggio. Il token, le options e il payload sono opzionali.

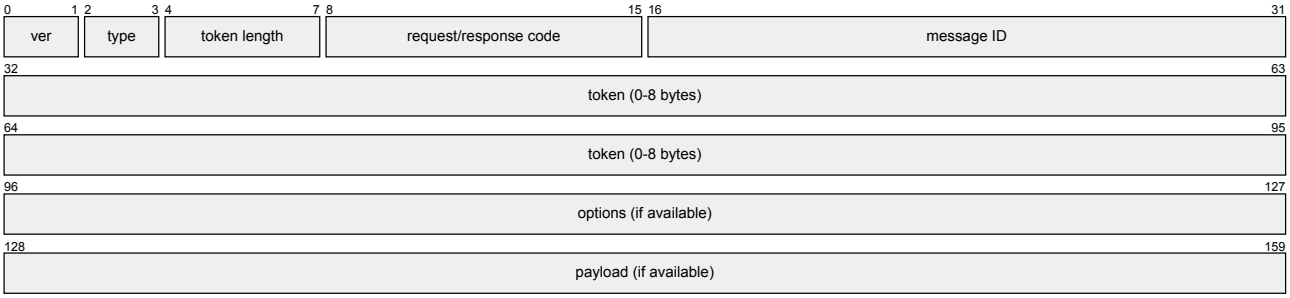

Figura 7: Struttura di un messaggio CoAP

Più dettagliatamente, come illustrato in Figura 7, un messaggio *CoAP* è composto da:

Header

- **Version** (2 bits): indica il numero di versione di CoAP.
- **Type** (2 bits): se si tratta di una request può essere (0 o 1) se *confirmable* o *non-confirmable* (ciò indica la volontà di ricevere un *acknowledgement message* o meno); se si tratta di una response può essere (2 o 3) nel caso di valore 2 si tratta di un *acknowledgement* (dunque di una risposta ad un *confirmable messagge*), mentre per il valore 3 si tratta di un *reset*, ciò significa che il messaggio è stato ricevuto ma non è stato possibile processarlo.
- **Token length** (4 bits): indica la lunghezza del campo token che può variare da 0 a 8 bytes.
- **Request/response code** (8 bits): i primi 3 bit sono analoghi ai codici di stato HTTP, mentre gli ultimi 5 comunicano ulteriori informazioni riguardo la richiesta o la risposta.
- **Message ID** (16 bits): utilizzato per trovare messaggi duplicati e per far corrispondere messaggi di tipo *acknowledgement/reset* a messaggi di tipo *confirmable/non-confirmable*.

Token

- Il campo **token** rappresenta un elemento opzionale di lunghezza variabile compresa tra 0 e 8 byte, utilizzato per stabilire una correlazione univoca tra richieste e relative risposte. Il valore del token viene generato dal client mittente e deve essere riportato inalterato dal server nella risposta corrispondente. Questo meccanismo consente di identificare in

modo inequivocabile l'associazione tra richieste e risposte, risultando particolarmente utile nella gestione di richieste multiple concorrenti.

Options

- Il campo **options** è un elemento opzionale che consente di specificare ulteriori parametri o metadati associati al messaggio. Le opzioni possono includere informazioni come il tipo di contenuto, le dimensioni del payload, le risorse richieste e altre configurazioni specifiche per la comunicazione. Le opzioni sono rappresentate da coppie chiave-valore e possono essere utilizzate per personalizzare il comportamento del protocollo in base alle esigenze specifiche dell'applicazione.

Payload

- Il campo **payload** rappresenta il contenuto effettivo del messaggio, che può essere di qualsiasi tipo e formato. Il payload è opzionale e può essere presente solo se necessario per la comunicazione. La dimensione del payload può variare notevolmente a seconda della natura dei dati scambiati, ma in generale, *CoAP* è progettato per gestire payload di dimensioni relativamente piccole, in linea con le limitazioni dei dispositivi *IoT*.

4.3.3 Altre funzionalità

CoAP offre diverse funzionalità avanzate che lo rendono adatto per applicazioni *IoT*:

- **Bult-In Discovery**: consente ai dispositivi di scoprire automaticamente le risorse disponibili su altri dispositivi nella rete. Questo facilita l'interazione tra dispositivi senza la necessità di configurazioni manuali complesse.
- **Asynchronous Message Exchanges**: *CoAP* supporta scambi di messaggi asincroni, consentendo ai dispositivi di inviare e ricevere messaggi in modo non bloccante. Questo è particolarmente utile in scenari in cui i dispositivi devono rispondere rapidamente a eventi esterni senza attendere una risposta immediata.
- **Optional Reliability with Confirmable Messages**: *CoAP* offre la possibilità di utilizzare messaggi *confirmable* per garantire la consegna affidabile dei messaggi. Questi messaggi richiedono una risposta di conferma

dal destinatario, assicurando che i dati siano stati ricevuti correttamente. Tuttavia, è possibile utilizzare anche messaggi *non-confirmable* per ridurre il sovraccarico in scenari in cui la consegna affidabile non è critica.

4.3.3.1 Aspetti positivi

- **Velocità**: il protocollo *CoAP* opera su *UDP*. *UDP* è un protocollo di trasmissione semplice che non richiede l'instaurazione di una connessione prima che i dati vengano inviati. Questo contrasta con *TCP* (Transmission Control Protocol), che richiede l'instaurazione di una connessione prima che i dati possano essere trasmessi. *UDP* è utile per i dispositivi *IoT*, che spesso devono inviare piccole quantità di dati in modo rapido ed efficiente.
- **Codifica efficiente**: *CoAP* utilizza uno schema di codifica binaria, che è più efficiente della codifica basata su testo utilizzata da *HTTP*. La codifica binaria riduce le dimensioni dei messaggi, risparmiando larghezza di banda e aumentando la velocità di comunicazione. Il protocollo *CoAP* supporta anche l'uso di *URI* compressi (Uniform Resource Identifiers), che riducono ulteriormente le dimensioni dei messaggi. Questo è particolarmente utile in ambienti vincolati, dove la larghezza di banda è spesso limitata.
- **Comunicazione stateless**: nella comunicazione *stateless*, ogni richiesta da un client a un server viene elaborata indipendentemente, senza alcuna conoscenza delle richieste precedenti. Questo rende il protocollo più robusto e resiliente, poiché non è influenzato dal fallimento di singole richieste. La comunicazione *stateless* semplifica anche l'implementazione del protocollo, poiché non richiede al server di mantenere uno stato per ogni client. Questo riduce i requisiti di risorse del server. La comunicazione *stateless* consente inoltre a *CoAP* di supportare la comunicazione asincrona, migliorando la flessibilità del protocollo e la sua idoneità per una varietà di applicazioni *IoT*.

4.4 HTTP

HTTP [5] è un protocollo di comunicazione basato sul modello *"request-response"*, ampiamente utilizzato per il trasferimento di dati su reti informatiche, in particolare per il web. È un protocollo semplice e flessibile che

consente la trasmissione di documenti ipertestuali e altri tipi di dati tra client e server.

4.4.1 Caratteristiche principali

- **Architettura Client-Server**: *HTTP* segue un'architettura client-server, in cui i client inviano richieste ai server e i server rispondono con i dati richiesti. Questo modello consente una comunicazione diretta tra client e server, facilitando l'interazione tra applicazioni web e servizi.
- **Stateless**: *HTTP* è un protocollo stateless, il che significa che ogni richiesta è indipendente dalle altre. Non mantiene lo stato tra le richieste, il che semplifica la gestione delle connessioni e riduce il carico sui server.
- **Supporto per diversi metodi di richiesta**: *HTTP* supporta diversi metodi di richiesta, tra cui GET, POST, PUT, DELETE e altri. Questi metodi consentono ai client di eseguire operazioni specifiche sui dati, come recuperare informazioni, inviare dati o eliminare risorse.

4.4.2 Versioni di HTTP

HTTP ha evoluto nel tempo attraverso diverse versioni:

- **HTTP/1.0**: la prima versione del protocollo, che introduce il concetto di richiesta e risposta tra client e server. Tuttavia, non supporta le connessioni persistenti, il che significa che ogni richiesta richiede una nuova connessione.
- **HTTP/1.1**: ha introdotto le connessioni *"keep-alive"* per consentire di sequenziare più scambi di richieste-risposte sulla stessa connessione *TCP*, quindi non richiede una nuova fase di creazione della connessione (handshake) per ogni richiesta.
- **HTTP/2**: ha introdotto un livello di *framing* e *multiplexing* binario per migliorare la latenza senza modificare il livello di trasporto.
- **HTTP/3**: si basa sul protocollo di trasporto *QUIC*, progettato specificamente per supportare connessioni multiplex senza dipendere da una singola connessione *TCP*.

4.5 Casi d'uso reali per l'utilizzo di MQTT, CoAP e HTTP

Di seguito sono riportati alcuni casi d'uso reali [6], [7] per l'utilizzo di *MQTT*, *CoAP* e *HTTP*:

- **MQTT**:
  - Monitoraggio di oleodotti e gasdotti: *MQTT* garantisce la consegna affidabile dei messaggi, aspetto cruciale per il monitoraggio continuo di oleodotti e gasdotti, al fine di rilevare tempestivamente perdite o danni.
  - Automazione industriale: nel settore manifatturiero, *MQTT* viene utilizzato per monitorare e controllare in tempo reale macchinari e impianti, migliorando l'efficienza operativa e riducendo i tempi di inattività.
  - Telematica veicolare: *MQTT* è impiegato per trasmettere dati sulle prestazioni e la diagnostica dei veicoli da dispositivi di bordo ai *server*, consentendo un monitoraggio in tempo reale della flotta.
  - Smart Home: in ambito domestico, *MQTT* è utilizzato per connettere e gestire dispositivi intelligenti come termostati, luci e sistemi di sicurezza, permettendo un controllo centralizzato e automazioni personalizzate.
- **CoAP**:
  - Sensori a basso consumo: grazie al suo basso overhead, CoAP è ideale per sensori IoT che operano su reti a bassa potenza e risorse limitate, prolungando la durata della batteria dei dispositivi.
  - Agricoltura: nell'ambito dell'agricoltura smart, CoAP può essere utilizzato per il monitoraggio dell'umidità del suolo, il controllo climatico nelle serre e il tracciamento del bestiame, migliorando la produttività e la sostenibilità.
  - Monitoraggio Ambientale: CoAP viene impiegato in dispositivi per il monitoraggio di condizioni ambientali come temperatura, umidità e qualità dell'aria, consentendo una raccolta dati efficiente anche in contesti remoti.
  - Smart metering: grazie al basso consumo energetico e alla capacità di funzionare su dispositivi limitati, CoAP offre un grande vantaggio

nella raccolta dati da contatori intelligenti per acqua, elettricità e gas, riducendo costi operativi e migliorando l'efficienza della rete.

- **HTTP**:
  - HTTP non è stato pensato ad-hoc per la messaggistica IoT tuttavia possiamo fare un esempio prendendo in considerazione un citofono con videocamera, quando una persona si avvicina, il dispositivo può attivarsi e iniziare una connessione *HTTP/HTTPS* per inviare un evento o una notifica iniziale. Ma per trasmettere il flusso video in tempo reale in modo efficiente e continuo, il dispositivo stabilisce una connessione *MQTT* per inviare i pacchetti dati, evitando il sovraccarico che comporterebbe l'uso di *HTTP* per ogni fotogramma.
  - Applicazioni Web: *HTTP* è il protocollo standard per la comunicazione tra browser e *server* web, consentendo la visualizzazione di pagine web, l'invio di moduli e l'interazione con applicazioni web.
  - API RESTful: *HTTP* è ampiamente utilizzato per implementare *API RESTful*, che consentono la comunicazione tra applicazioni e servizi web, facilitando l'integrazione tra sistemi diversi.

4.6 Confronto tra i protocolli

Nella Tabella 5 viene riportato un riassunto delle principali differenze tra *MQTT*, *CoAP* e *HTTP* [4], [5], [6]:

Caratteristica	MQTT	CoAP	HTTP
Stile di comunicazione	Pub/Sub con broker centrale	Request/ Response	Request/ Response
Protocollo di trasporto	TCP	UDP principalmente	TCP
Dimensione header	2 bytes	4 bytes	Variabile
Stato della connessione	Stateful	Stateless	Stateless
Supporto multicast	No	Sì	No

Efficienza energetica	Moderata	Alta	Bassa
Livelli di QoS	3 livelli (0, 1, 2)	Nessuno	Nessuno
Affidabilità	Alta (conferma di ricezione)	Moderata (conferma opzionale)	Bassa (stateless)
Consumo di banda	Medio	Basso	Alto
Scalabilità	Eccellente	Media	Alta
Tipi di messaggi	CONNECT, PUBLISH, SUBSCRIBE, UNSUBSCRIBE, etc.	GET, POST, PUT, DELETE	GET, POST, PUT, DELETE
Sicurezza	TLS/SSL	DTLS	TLS/SSL

Tabella 5: Tabella di confronto tra i protocolli MQTT, CoAP e HTTP

4.6.1 Performance dei protocolli

La performance dei protocolli può variare notevolmente a seconda delle specifiche implementazioni e degli scenari di utilizzo. Tuttavia, in generale, possiamo fare alcune considerazioni sulle performance relative di *MQTT*, *CoAP* e *HTTP* [5], [8].

Sebbene *HTTP* abbia un overhead inferiore per stabilire la connessione (dovuto al fatto che *MQTT* dopo aver stabilito la connessione *TCP* deve scambiarsi i pacchetti di *Connect*). *HTTP* consuma molti più byte per messaggio quindi *MQTT* è più efficiente con trasmissioni frequenti.

HTTP può essere più leggero per invii isolati, ma *MQTT* è chiaramente superiore in scenari a lungo termine e ad alto volume, grazie alla sua bassa latenza, connessioni persistenti e minore consumo di banda.

La riduzione del tempo di risposta per messaggio è attesa quando vengono inviati più messaggi durante una singola connessione *TCP*, dato che è possibile evitare l'overhead di stabilire la connessione per ogni singolo messaggio. *HTTP* non ha questa possibilità e questo rende il confronto più a favore del protocollo *MQTT*.

Considerando invece *CoAP*, gli overhead di *UDP* sono minori rispetto a quelli di *TCP*. Tuttavia per raggiungere le capacità di *TCP* (pacchetti più grandi, trasmissioni più affidabili) sarebbe necessario implementare dei meccanismi più complessi in *CoAP*. I tempi di risposta per messaggio sono simili a quelli di *MQTT* che utiliza *QoS* 0, se invece si utilizza *QoS* 1 o 2, i tempi di risposta aumentano notevolmente, rendendo il protocollo meno performante rispetto a *CoAP*.

4.7 Serializzazione dei dati

La serializzazione dei dati è il processo di conversione di strutture dati o oggetti in un formato che può essere facilmente memorizzato o trasmesso e successivamente ricostruito. Questo è particolarmente importante nei protocolli di comunicazione come *MQTT*, *CoAP* e *HTTP*. Se consideriamo uno scenario con dispositivi che hanno risorse limitate, la serializzazione dei dati diventa cruciale per garantire che le informazioni possano essere scambiate in modo efficiente e senza sprechi di banda.

Sono stati considerati diversi formati di serializzazione, tra cui:

- **JSON** [9]: è un formato di serializzazione dei dati basato su testo, ampiamente utilizzato per la sua leggibilità e facilità d'uso. Tuttavia, può essere meno efficiente in termini di spazio rispetto ad altri formati binari. Può vantare una buona compatibilità con diversi linguaggi di programmazione e piattaforme rendendolo una scelta popolare, tuttavia con ampi *dataset* può diventare pesante e meno performante.
- **Protocol Buffers** [10]: è un formato di serializzazione binario sviluppato da Google, noto per la sua efficienza e compattezza. È progettato per essere veloce e leggero, rendendolo adatto per applicazioni con risorse limitate. Tuttavia, richiede una definizione dello schema dei dati, il che può complicare la gestione delle modifiche ai dati nel tempo. Sebbene sia molto efficiente, la sua complessità e la necessità di definire uno schema possono renderlo meno flessibile dato che non ha capacità di auto-descrizione e lo schema che rappresenta i dati deve essere condiviso tra le parti comunicanti.
- **MessagePack** [11]: è un formato di serializzazione binario che mira a essere più compatto di JSON, mantenendo al contempo la semplicità d'uso.

È progettato per essere veloce e leggero, caratteristiche che lo rendono particolarmente adatto per applicazioni con risorse limitate. Tuttavia, MessagePack restituisce solo una struttura dati tipizzata dinamicamente e non fornisce controlli automatici sulla struttura dei dati.

- **CBOR** [12]: è un formato di serializzazione binario che si basa su *JSON* ma è progettato per essere più efficiente in termini di spazio. Supporta una vasta gamma di tipi di dati e può essere utilizzato in scenari con risorse limitate. È definito da uno standard, il che garantisce una certa interoperabilità tra implementazioni. Tuttavia, la sua complessità può renderlo meno intuitivo rispetto a formati più semplici come *JSON*.

4.8 Aspetti di sicurezza

Nella comunicazione di dati è fondamentale garantire la sicurezza e l'integrità delle informazioni scambiate. I protocolli *MQTT*, *CoAP* e *HTTP* offrono diverse opzioni per implementare la sicurezza. I protocolli possono essere protetti utilizzando tecniche di cifratura come *Transport Layer Security*$_G$ o *Datagram Transport Layer Security*$_G$ per garantire la riservatezza e l'integrità dei dati durante la trasmissione. Inoltre, è possibile implementare meccanismi di autenticazione per verificare l'identità dei dispositivi coinvolti nella comunicazione.

In particolare *TLS* è comunemente utilizzato con *TCP*, mentre *DTLS* è progettato per funzionare con *UDP*. Entrambi i protocolli offrono un livello di sicurezza elevato, ma DTLS può essere più complesso da implementare a causa della natura *stateless* di *UDP*. Questi metodi di cifratura sono essenziali per proteggere la comunicazione tra dispositivi.

È possibile poi utilizzare altri algoritmi di cifratura per cifrare il payload dei messaggi, garantendo che i dati siano protetti anche se intercettati durante la trasmissione. In questi casi è importante considerare che esistono anche degli algoritmi di cifratura leggera, ma ugualmente sicuri, che possono essere utilizzati per dispositivi con risorse limitate.

4.9 Scelte progettuali

Il progetto è stato sviluppato in linguaggio *Go*, che offre un'ampia gamma di librerie e strumenti per la gestione della comunicazione di rete, la serializzazione dei dati e la sicurezza. Inoltre, *Go* è noto per le sue prestazioni elevate e la facilità di scrittura di codice concorrente, rendendolo una scelta ideale per lo sviluppo di applicazioni *IoT*.

Come protocollo di comunicazione è stato utilizzato *CoAP* su *UDP*, in quanto i dispositivi *IoT* che si voleva simulare sono a bassa potenza e con risorse limitate. Inoltre si suppone che questi dispositivi non abbiano uno stack *"TCP/IP"* dunque la scelta di *CoAP* è stata fatta anche per questa ragione.

Volendo garantire la sicurezza della comunicazione, è stato scelto di utilizzare *DTLS* per cifrare i messaggi scambiati tra i dispositivi e il server.

Per la serializzazione dei dati, è stato scelto di utilizzare *CBOR* in quanto è un formato binario che consente di risparmiare banda e risorse, oltre al fatto che utilizza un algoritmo molto performante per compiere la serializzazione e deserializzazione dei dati.

Nei prossimi capitoli verranno presentate le implementazioni pratiche di queste scelte progettuali, specificando anche le librerie utilizzate e gli altri servizi che sono stati integrati nel progetto come il database *TimescaleDB* e le dashboard di *Grafana*.

Capitolo 5

Architettura del sistema

In questo capitolo viene presentata l'architettura del sistema, con una descrizione delle fasi di progettazione ad alto livello e di dettaglio accompagnate da una panoramica delle tecnologie e degli strumenti utilizzati, mentre le specifiche funzionalità di ogni componente verranno presentate nel prossimo capitolo.

5.1 Progettazione ad alto livello

Grazie alla progettazione ad alto livello del sistema sono stati individuati i principali componenti e le loro interazioni. I principali componenti del sistema, illustrati in Figura 8, sono:

- **Dispositivi IoT**: dispositivi simulati che raccolgono dati biomedici di una persona e inviano tali misurazioni con frequenza modificabile.
- **Server CoAP**: gestisce le richieste *CoAP* provenienti dai dispositivi *IoT*.
- **Database**: memorizza i dati raccolti dai dispositivi *IoT*.
- **API HTTP**: fornisce l'accesso ad alcuni dati rilevati dai dispositivi e permette l'invio di comandi ai dispositivi *IoT*.
- **Dashboard**: visualizza i dati raccolti da un particolare dispositivo *IoT* per monitorare i dati biomedici di una persona.

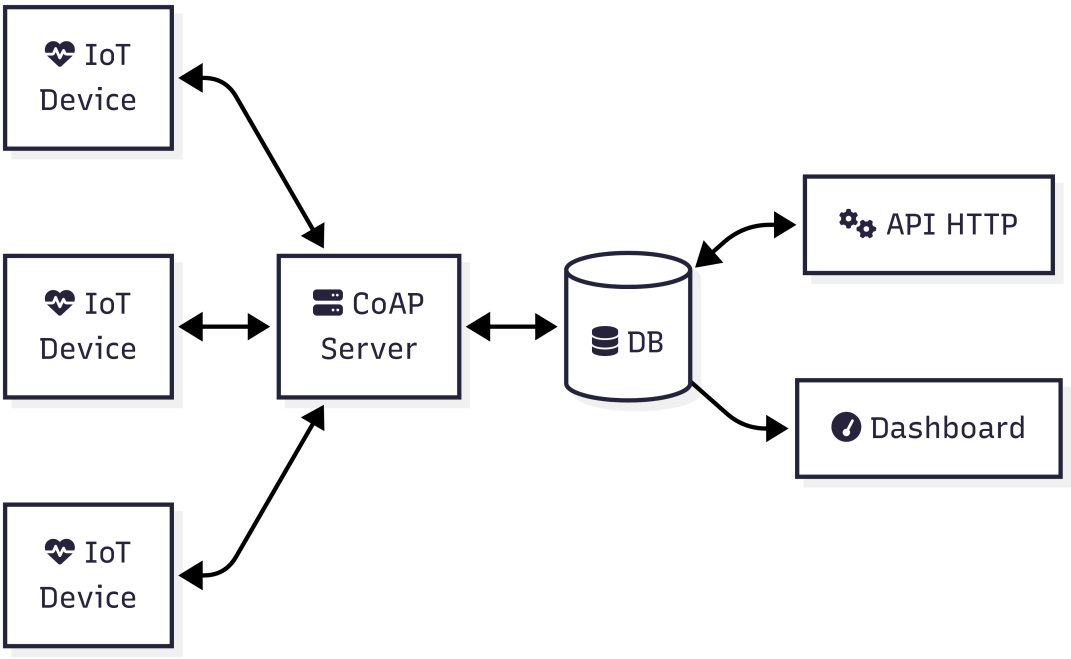

Figura 8: Architettura del sistema

Nelle scelte implementative si è deciso di utilizzare il protocollo *CoAP* per la comunicazione tra i dispositivi *IoT* e il server, in quanto è un protocollo leggero e adatto per dispositivi con risorse particolarmente limitate che operano in contesti *narrowband*.

I dispositivi *IoT* sono simulati e inviano dati attraverso messaggi di tipo *Confirmable* al server *CoAP*, quest'ultimo controlla che i dati arrivino da un dispositivo autorizzato, successivamente si occupa di validare i dati e di salvarli sul database. In seguito, il server *CoAP* controlla se ci siano comandi pendenti per quel dispositivo, in caso affermativo nella risposta che il server invia al client è contenuto anche il comando. L'*API HTTP* consente di inviare comandi ai dispositivi *IoT* tramite alcuni *endpoint*, questi comandi verranno poi consegnati dal server *CoAP* come accennato in precedenza. Infine è presente una dashboard che permette di visualizzare i dati raccolti da un particolare dispositivo *IoT*, la dashboard utilizza direttamente il *database* per recuperare i dati e visualizzarli in tempo reale.

5.2 Tecnologie e strumenti

Di seguito viene data una panoramica delle tecnologie e strumenti utilizzati.

5.2.1 Go

Go [13] (v.1.24.3) è un linguaggio di programmazione *open source* che consente di costruire agevolmente sistemi sicuri e scalabili. È stato scelto come linguaggio di programmazione principale per lo sviluppo del sistema per via della sua efficienza e semplicità nella gestione della concorrenza.

Uno dei principali punti di forza di *Go* è il suo modello di concorrenza basato sulle goroutine e sui channel, che permette di scrivere applicazioni concorrenti in modo intuitivo, riducendo la complessità tipica della programmazione *multithread*. Questo lo rende particolarmente adatto per sistemi distribuiti o applicazioni che richiedono elevate prestazioni in termini di parallelismo.

Inoltre, *Go* offre una compilazione rapida, un *garbage collector* efficiente e una libreria standard ricca che facilita lo sviluppo di applicazioni robuste. Il linguaggio è progettato per favorire la leggibilità e la manutenibilità del codice, grazie a una sintassi semplice e coerente, e a un sistema di tipi solido ma non eccessivamente complesso.

In *Go* la gestione delle dipendenze avviene tramite *packages*. Per questo progetto sono stati considerati i *packages* mostrati in Tabella 6.

Package	Versione	Descrizione
go-coap [14]	3.3.6	Implementa il protocollo *CoAP* per la comunicazione tra dispositivi *IoT*.
dtls [15]	3.0.2	Implementa il protocollo *DTLS* in *Go*, fornendo crittografia e autenticazione per le comunicazioni basate su *UDP*.
viper [16]	1.20.1	Consente di leggere e scrivere file di configurazione in vari formati.
cbor [17]	2.8.0	Implementa la codifica e decodifica di dati in formato *CBOR*, un formato binario efficiente per la serializzazione dei dati.

validator [18]	10.26.0	Utilizzato per la validazione dei dati, consente di definire regole di validazione per le strutture dati.
gin [19]	1.10.1	Framework per la creazione di applicazioni web, noto per la sua velocità e facilità d'uso, in particolare è stato utilizzato per lo sviluppo di una *API HTTP*.
pq [20]	1.10.9	*Driver* per il database *PostgreSQL*, che consente di interagire con un database *PostgreSQL* da un'applicazione *Go*.
testify [21]	1.10.0	*Framework* di testing che fornisce asserzioni e funzionalità di *mocking* per semplificare la scrittura dei test.
go-sqlmock [22]	1.5.2	Utilizzato per il testing delle interazioni con il *database*, consente di simulare un *database SQL* per testare le query senza dipendere da un *database* reale.

Tabella 6: Tabella dei packages di Go utilizzati

5.2.2 Docker

Docker [23] (v.28.0.4) è una piattaforma per la creazione, distribuzione e gestione di *container*, che consente di isolare le applicazioni e le loro dipendenze in ambienti leggeri e portabili. È stato scelto per facilitare la gestione dei servizi del sistema, in particolare per il database *TimescaleDB* e il servizio di visualizzazione dei dati *Grafana*.

L'utilizzo di *Docker* consente di semplificare la configurazione e l'esecuzione dei servizi, garantendo coerenza tra gli ambienti di sviluppo e produzione.

In particolare, è stato utilizzato *Docker Compose* per orchestrare i vari servizi del sistema, permettendo di avviare e gestire facilmente i container necessari attraverso un singolo file di configurazione. Questo approccio facilita la gestione delle dipendenze tra i servizi e semplifica il processo di *deployment* del sistema completo.

Nella tabella Tabella 7 sono riportate le immagini *Docker* utilizzate nel progetto, con le relative versioni e descrizioni.

Immagine	Versione	Descrizione
timescaledb [24]	2.20.3	Immagine *Docker* per *TimescaleDB*, un'estensione di *PostgreSQL* progettata per gestire dati temporali e serie storiche.
grafana [25]	12.0.1	Immagine *Docker* per *Grafana*, una piattaforma di visualizzazione e analisi dei dati che consente di creare dashboard interattive.

Tabella 7: Tabella delle immagini Docker utilizzate

5.2.3 Altri strumenti

In aggiunta agli strumenti principali, sono stati utilizzati altri strumenti e tecnologie per facilitare lo sviluppo e la gestione del progetto. Nella tabella Tabella 8 sono riportati alcuni di questi strumenti, con le relative versioni e descrizioni.

Tool	Versione	Descrizione
pgAdmin4 [26]	9.3	Strumento di amministrazione e gestione per *PostgreSQL*, utilizzato per la visualizzazione delle tabelle e la gestione del database.
hurl [27]	6.1.1	Strumento per testare le *API HTTP*, consente di inviare richieste e verificare le risposte in modo semplice e veloce.
vizb [28]	0.3.0	Strumento a riga di comando per visualizzare i risultati dei *benchmark Go* come grafici *HTML* interattivi, utile per confrontare le prestazioni di diverse implementazioni.

Tabella 8: Tabella degli strumenti utilizzati

5.3 Progettazione di dettaglio

La progettazione di dettaglio ha il compito di guidare la codifica, definendo unità architetturali che formano le varie componenti del sistema. Per ogni componente viene riportato come è stato strutturato il codice, in alcuni casi con il supporto di diagrammi delle classi che mostrano come una particolare funzionalità è stata implementa. Nel Capitolo 6 verranno poi presentate le specifiche funzionalità che sono state implementate per ogni componente.

5.3.1 Dispositivi IoT

I dispositivi *IoT* sono simulati ed implementati mediante un client *CoAP* che invia periodicamente dati al server. Ogni dispositivo simula il comportamento di un dispositivo indossabile con sensori, inviando misurazioni come frequenza cardiaca, temperatura corporea e altri parametri biomedici. Il client è configurabile tramite un file di configurazione che permette di modificare la frequenza di invio dei dati e altre impostazioni. Il codice del client è stato strutturato creando un *package Go* dedicato, che contiene le strutture dati e le funzioni necessarie per la simulazione del dispositivo *IoT*.

5.3.2 Server CoAP

Il server CoAP gestisce le richieste provenienti dai dispositivi *IoT* e si occupa di validare i dati ricevuti e di salvarli nel database. Il server ha un'architettura esagonale, che suddivide le responsabilità nei layer per le logiche di *application*, *business* e *persistence*. In particolare, il layer di *application* si occupa di gestire le richieste *CoAP* in ingresso, il layer di *business* contiene la logica di validazione e salvataggio dei dati, mentre il layer di *persistence* si occupa dell'interazione con il database. Il server è configurabile tramite un file di configurazione che permette di modificare le impostazioni del server e del database.

In seguito viene riportato un diagramma delle classi che mostra come viene strutturata ogni funzionalità del server *CoAP* seguendo l'architettura esagonale. In particolare, in Figura 9, sono riportate le classi principali del server *CoAP* per l'inserimento dei dati all'interno del *database*.

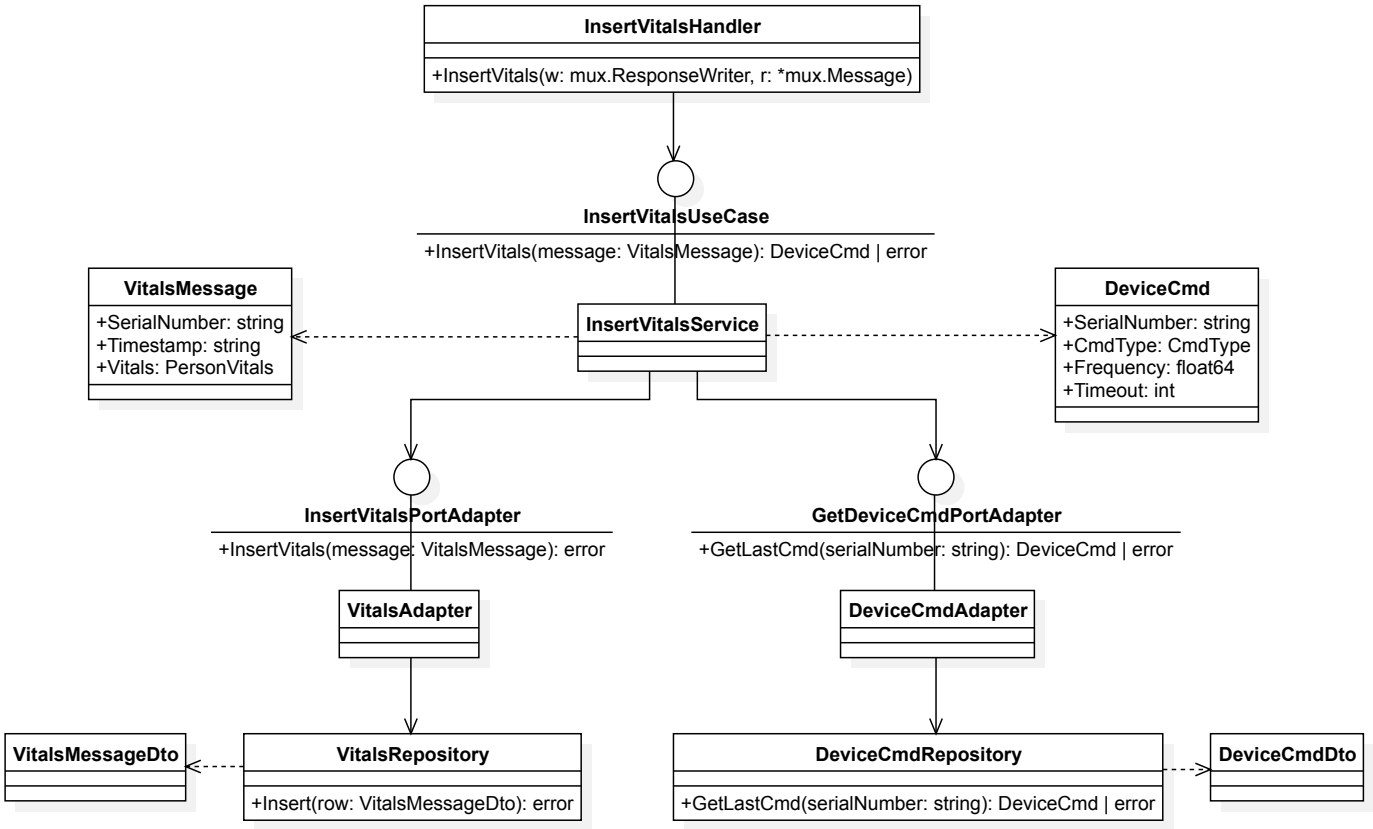

Figura 9: Diagramma delle classi del server CoAP

5.3.3 Database

Il *database* utilizzato è *TimescaleDB*, un'estensione di *PostgreSQL* progettata per gestire dati temporali e quelle che vengono definite *"time series"*. La scelta di questo *database* è motivata dalla natura dei dati raccolti dai dispositivi *IoT*, che sono caratterizzati da una forte componente temporale e necessitano di operazioni di inserimento e interrogazione efficienti su grandi volumi di dati. *TimescaleDB* offre funzionalità avanzate per l'ottimizzazione delle query su serie temporali, come ad esempio un'efficace aggregazione dei dati. Il *database* memorizza in formato tabellare i dati raccolti dai dispositivi *IoT*, inoltre, vengono anche memorizzati i comandi da inviare ai dispositivi. Di seguito, in Figura 10, viene inoltre riportato lo schema del *database*, che mostra le tabelle principali e le loro relazioni.

Le tabelle principali sono:

- **devices**: contiene le informazioni sui dispositivi *IoT* registrati nel sistema.

- **vitals**: memorizza le misurazioni raccolte dai dispositivi *IoT*.
- **cmds**: memorizza i comandi da inviare ai dispositivi *IoT*.

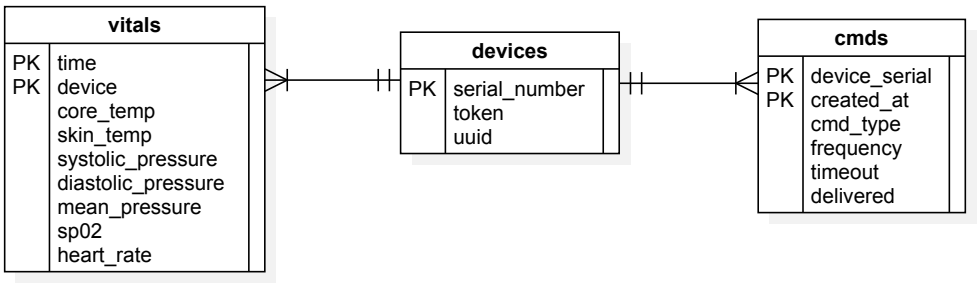

Figura 10: Schema del database

Il servizio di *TimescaleDB* è esposto tramite un'immagine *Docker*, rendendolo così disponibile per le operazioni del server *CoAP* e dell'*API HTTP*.

5.3.4 API HTTP

L'*API HTTP* fornisce alcuni *endpoint* per l'accesso ai dati raccolti dai dispositivi *IoT* e per l'invio di comandi ai dispositivi. L'*API* è implementata grazie ad un server *HTTP* per la gestione delle richieste e risposte. Così come il server *CoAP*, anche l'*API HTTP* è strutturata seguendo l'architettura esagonale, con i layer di *application*, *business* e *persistence*. Il layer di *application* gestisce le richieste *HTTP* in ingresso, il layer di *business* contiene le logiche per il recupero o il salvataggio dei dati, mentre il layer di *persistence* si occupa dell'interazione con il *database*. L'*API* è configurabile tramite un file di configurazione che permette di modificare le impostazioni del server e del *database*.

In Figura 11 viene riportato un esempio di diagramma delle classi che mostra come viene strutturata la funzionalità per l'invio del comando *"StartStreaming"* che permette di avviare lo streaming dei dati in maniera più frequente per un dispositivo *IoT*. Le altre funzionalità sono strutturate in modo simile per l'invio degli altri comandi e per il recupero dei dati.

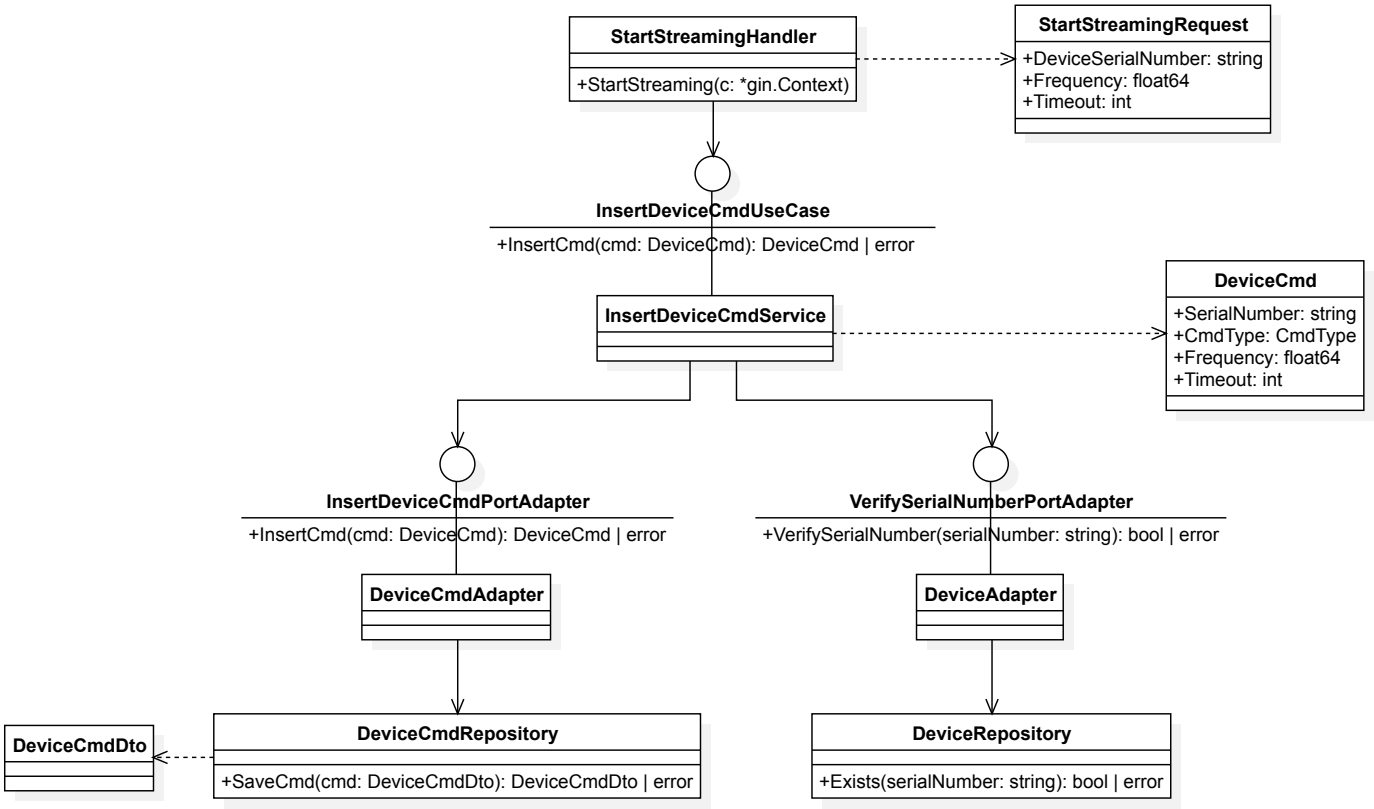

Figura 11: Diagramma delle classi dell'API HTTP

5.3.5 Dashboard

La dashboard è un'interfaccia utente che consente di visualizzare i dati raccolti dai dispositivi *IoT*. Utilizza *Grafana* per la visualizzazione dei dati e si connette al database per recuperare le informazioni necessarie. La dashboard è configurabile e permette di visualizzare i dati in tempo reale, offrendo una panoramica delle misurazioni dei dispositivi *IoT*.

Il servizio di *Grafana* viene esposto tramite un'immagine *Docker*, così da poter essere facilmente configurato tramite i file di configurazione da caricare su un apposito volume.

Capitolo 6

Sviluppo del prototipo

In questo capitolo viene descritto lo sviluppo del prototipo, con particolare attenzione riguardo le scelte tecnologiche iniziali, la codifica di ogni componente architetturale e ai test effettuati.

6.1 Fasi iniziali

Dopo aver definito i requisiti del progetto, è stato fondamentale eseguire alcuni *benchmark* comparativi per valutare le prestazioni delle tecnologie scelte. Questi test hanno permesso di identificare le soluzioni più adatte per il progetto, garantendo un'implementazione efficiente.

Data la natura dei dispositivi *IoT* che si è inteso simulare, è stato necessario considerare le limitazioni hardware, come la potenza di calcolo e la memoria disponibile, per garantire che il prototipo fosse in grado di funzionare in modo ottimale anche su dispositivi con risorse limitate. Inoltre nel contensto di reti *narrowband* sono stati considerati i vincoli di latenza e larghezza di banda, che sono cruciali per garantire una comunicazione efficace tra i dispositivi e il server.

Queste considerazioni hanno guidato le scelte tecnologiche e architetturali, assicurando che il prototipo fosse in grado di soddisfare i requisiti del progetto.

Al fine di confermare le scelte effettuate sono stati eseguiti *benchmark* di confronto, in particolare per alcuni formati di serializzazione dei dati e per alcuni algoritmi di compressione. Questi *benchmark* sono stati implementati utilizzando la suite di test fornita dalla libreria standard di *Go*.

6.1.1 Serializzazione dei dati

Nell'ambito della serializzazione dei dati, sono stati confrontati i seguenti algoritmi: *JSON* della libreria standard di *Go*, *JSON sonic* [29], due

implementazioni di *MsgPack* [11] e *CBOR* [12]. L'interesse è stato quello di valutare le prestazioni in termini di velocità e dimensione dei dati serializzati.

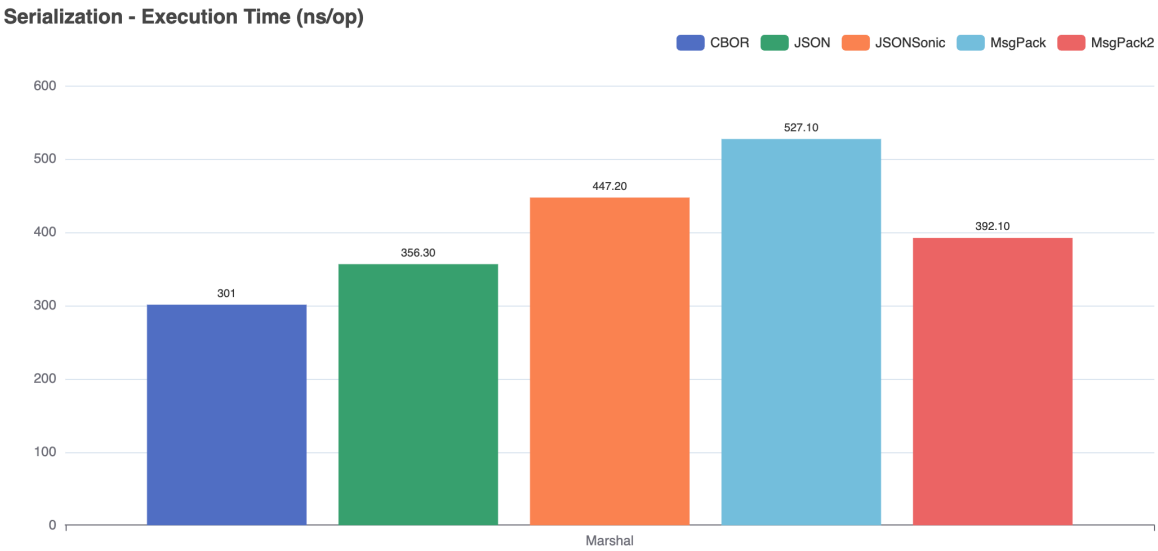

Figura 12: Tempo di serializzazione dei dati in diversi formati

Come si può osservare dal grafico in Figura 12, il formato *CBOR* è il più veloce in termini di tempo di serializzazione. Questo è un aspetto da tenere in considerazioni date le risorse limitate dei dispositivi *IoT*.

In termini di dimensione dei dati serializzati, il formato *CBOR* risulta essere il più efficiente, seguito da *MsgPack*, e infine *JSON*. Questo rende *CBOR* la scelta ideale per applicazioni che richiedono una trasmissione efficiente dei dati.

In Tabella 9 è riportata la dimensione dei dati serializzati in diversi formati, considerando il *payload* di un pacchetto contenente le misurazioni dei dati biomedici di una persona, che il dispositivo *IoT* manda periodicamente al server.

Formato	**Dimensione in bytes**
JSON	218
MsgPack	178
CBOR	102

Tabella 9: Tabella di confronto della dimensione dei dati serializzati in diversi formati

Infine, in Figura 13 viene riportato il tempo di deserializzazione dei dati, evidenziando la velocità di ciascun formato. Tuttavia, la deserializzazione non è cruciale come la serializzazione dato che è a carico del server, il quale non è un dispositivo estremamente limitato.

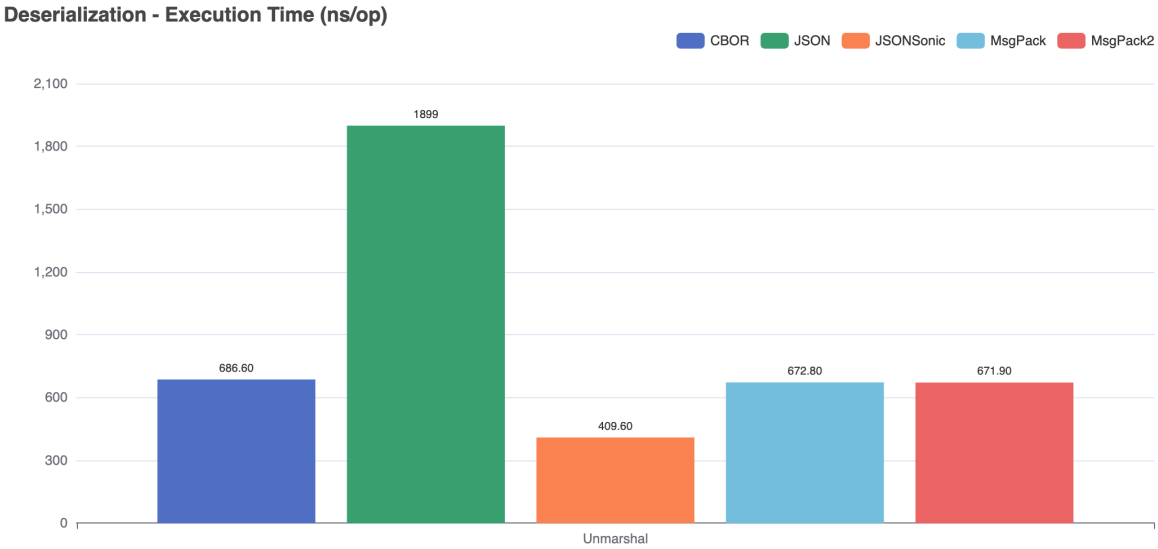

Figura 13: Tempo di deserializzazione dei dati in diversi formati

A seguito di queste considerazioni, si è deciso di utilizzare il formato *CBOR* per la serializzazione dei dati nel prototipo. Questo formato offre una ottima velocità di serializzazione e una dimensione dei dati ridotta, con una velocità di deserializzazione in linea con le altre opzioni.

6.1.2 Compressione dei dati

In alcuni scenari, il dispositivo *IoT* ha la necessità di inviare una mole di dati più elevata attraverso degli invii *batch*, questo accadde quando il dispositivo deve effettuare un processo di sincronizzazione con il server, ad esempio quando il dispositivo si riconnette alla rete dopo un periodo di inattività da parte del server.

A tale scopo, è stato necessario valutare le prestazioni di alcuni algoritmi di compressione dei dati, in particolare *gzip* [30] e *brotli* [31].

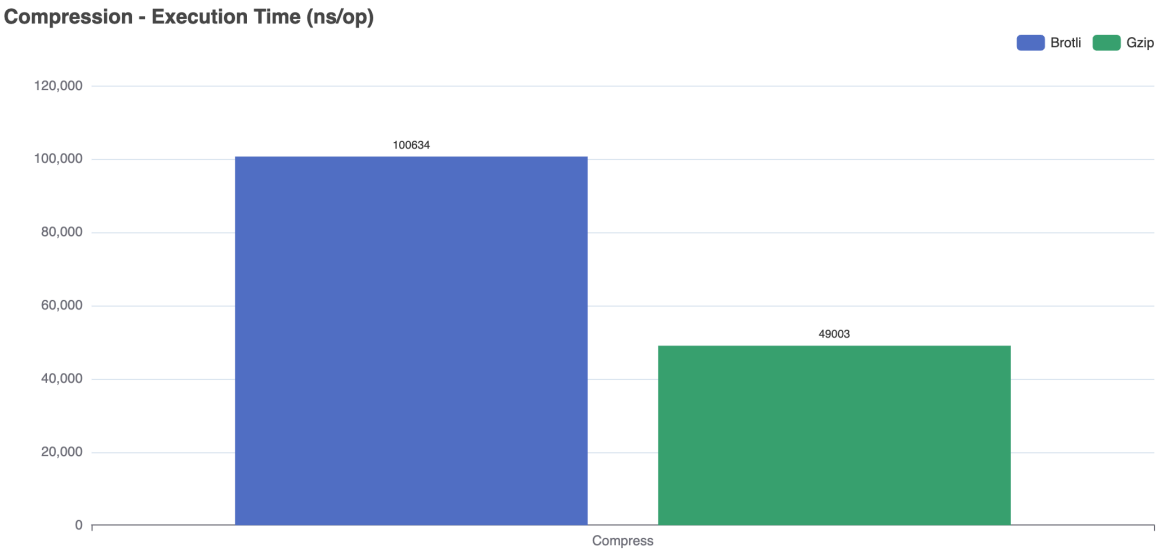

Figura 14: Tempo di compressione con diversi algoritmi

Come si può osservare dal grafico in Figura 14, l'algoritmo *gzip* risulta essere sensibilmente più veloce in termini di tempo di compressione rispetto a *brotli*.

Tuttavia, *brotli* offre una compressione più efficiente in termini di dimensione dei dati compressi.

In Tabella 10 è riportata la dimensione dei dati compressi con i due algoritmi, considerando il *payload* di un pacchetto contenente un *batch* di 10 misurazioni in momenti temporali differenti dei dati biomedici di una persona, che il dispositivo IoT manda al server durante un processo di sincronizzazione. Per il caso appena descritto la dimensione iniziale prima della compressione è di 1021 bytes.

Algoritmo	**Dimensione in bytes**
Gzip	350
Broli	278

Tabella 10: Tabella di confronto della dimensione dei dati compressi con diversi algoritmi

Inoltre, è stata considerata anche la velocità per la successiva decompressione dei dati che è a carico del server. In Figura 15 è riportato il tempo di decompressione dei dati con i due algoritmi.

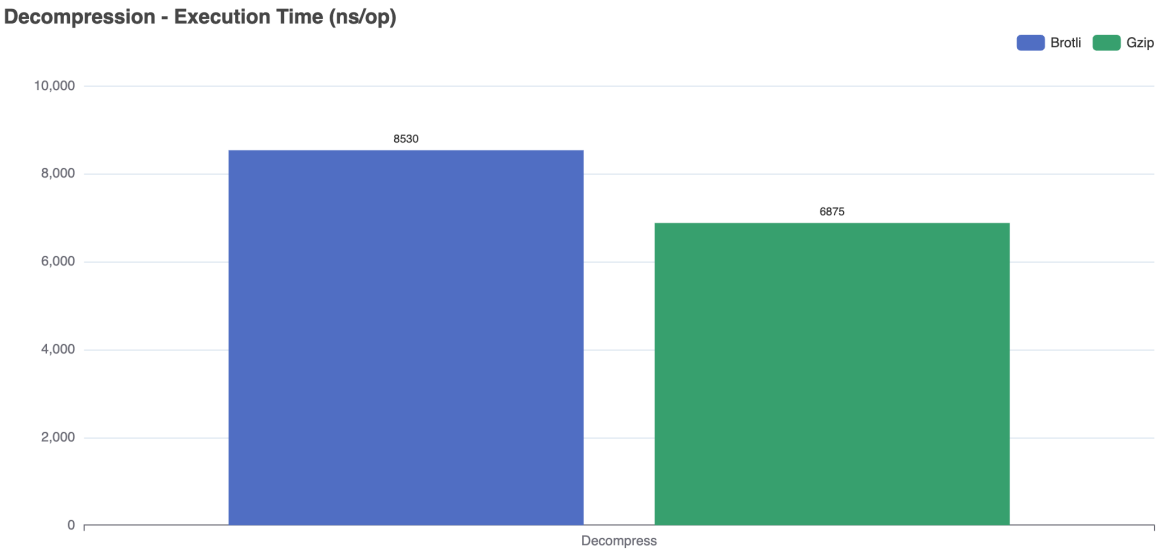

Figura 15: Tempo di decompressione con diversi algoritmi

A seguito di queste considerazioni, si è deciso di utilizzare l'algoritmo *gzip* per la compressione dei dati nel prototipo. Questo algoritmo offre una velocità di compressione significativamente superiore a *brotli* e fornisce comunque un rapporto di compressione adeguato, rendendolo più adatto allo scenario in cui si hanno dispositivi *IoT* con capacità di calcolo limitate.

6.2 Dati trasmessi

I dati che vengono trasmessi dal dispositivo *IoT* al server sono rappresentati da una struttura dati *"VitalsMessage"* che contiene il numero seriale del dispositivo, le misurazioni dei dati biomedici e il timestamp per tenere traccia del momento temporale in cui sono stati rilevati i dati. Questi dati vengono serializzati in formato *CBOR*. Come è possibile vedere in Codice 1, è possibile aggiungere dei metadati per identificare i campi della struttura dati, in modo da poterli serializzare correttamente in formato *CBOR*, anche se ciò non è strettamente necessario. Per risparmiare *bytes* nella trasmissione dei dati, si è deciso di utilizzare nomi brevi per questi metadati, che sono comunque significativi per il contesto.

```go
type Pulse struct {
  Value  int  `cbor:"p"`
```

```
3   }
4
5   type Saturation struct {
6     SpO2  int  `cbor:"s"`
7   }
8
9   type Pressure struct {
10    Systolic  int `cbor:"s"`
11    Diastolic int `cbor:"d"`
12    Mean      int `cbor:"m"`
13  }
14
15  type Temperature struct {
16    Skin float64 `cbor:"s"`
17    Core float64 `cbor:"c"`
18  }
19
20  type PersonVitals struct {
21    Pulse        Pulse       `cbor:"pu"`
22    Pressure     Pressure    `cbor:"pr"`
23    Temperature  Temperature `cbor:"t"`
24    Saturation   Saturation  `cbor:"s"`
25  }
26
27  type VitalsMessage struct {
28    Vitals       PersonVitals `cbor:"v"`
29    SerialNumber string       `cbor:"s"`
30    Timestamp    string       `cbor:"ts"`
31  }
```

Codice 1: Strutture dei dati biomedici trasmessi dal dispositivo IoT

6.3 Dispositivo IoT

La codifica del dispositivo *IoT* simulato è stata realizzata creando un *package Go* dedicato, che implementa le funzionalità richieste per la simulazione del dispositivo.

Il dispositivo viene gestito tramite un file di configurazione che contiene le informazioni necessarie per la connessione al server, come l'indirizzo e la porta, e il numero seriale del dispositivo stesso.

Quando il dispositivo si connette al server, attraverso uno *"StreamingManager"*, Codice 2, inizia ad inviare periodicamente i dati biomedici, serializzati in formato *CBOR*. Al fine di generare dei dati biomedici che non avessero variazioni irrealistiche, è stata utilizzata una distribuzione normale.

```go
package client

type StreamingManager struct {
  mu                 sync.Mutex
  DeviceSerialNumber string
  Token              string
  ServerAddr         string
  DefaultFrequency   float64
  StreamingFrequency float64
  Timeout            int
  Timer              *time.Timer
  RealTimeFlag       bool
  SyncNeeded         bool
}
```

Codice 2: Struttura dello StreamingManager del dispositivo IoT

Nel Codice 3 viene mostrato la funzione principale mediante cui lo *"StreamingManager"* gestisce lo streaming dei dati. Il dispositivo invia i dati con una frequenza predefinita impostata nel file di configurazione, ma tale frequenza può essere modificata dinamicamente tramite comandi inviati dal server. Infatti il dispositivo, per ogni richiesta effettuta al server, attende una risposta e controlla se ci sono comandi da eseguire.

Il dispositivo dunque è in grado di gestire due differenti modalità di streaming:

- Streaming di default, in cui i dati vengono inviati al server in maniera meno frequente, è la modalità predefinita appena il dispositivo viene attivato.
- Streaming più frequente, attivabile tramite un comando da poter inviare tramite l'*API HTTP*, in cui i dati vengono inviati al server con una frequenza più breve rispetto alla frequenza di default, per simulare un invio più simile al concetto di *real-time*.

Nel caso in cui il server non sia raggiungibile, il dispositivo salva i dati in un file locale, in modo da poterli inviare successivamente quando la connessione sarà ripristinata. Questo processo di sincronizzazione viene gestito da una *goroutine*, infatti, quando il dispositivo si riconnette al server, invia i dati salvati sfruttando un invio *batch* abbinato ad un processo di compressione dei dati, in modo da ridurre il numero di richieste al server e ottimizzare l'utilizzo della banda disponibile.

```go
package client

func (sm *StreamingManager) StartStreaming(path string, MakeRequest MakeRequestFunc) {
    personVitals := domain.GeneratePersonVitals()
    var vitalsMessage domain.VitalsMessage
    for {
        co := sm.SetConnection()
        ctxRequest, cancel := context.WithTimeout(context.Background(), 1*time.Second)
        vitalsMessage.GenerateMessage(sm.DeviceSerialNumber, personVitals)
        cborMessage, err := util.EncodeVitalsMessage(vitalsMessage)
        if err != nil {
            log.Fatal("Error encoding vitals message:", err)
        }
        response, err := MakeRequest(ctxRequest, co, path, cborMessage)
```

```
15          cancel()
16          if err != nil {
17              err := sm.saveDataToFile(vitalsMessage)
18              if err != nil {
19                  log.Fatalf("Error saving data to file: %v", err)
20              }
21          } else {
22              err = sm.handleResponse(response)
23              if err != nil {
24                  log.Fatalf("Error handling response: %v", err)
25              }
26              if sm.SyncNeeded {
27                  messages, err := retrieveLines(filenameToStoreData)
28                  if err != nil {
29                      log.Fatalf("Error retrieving lines: %v", err)
30                  }
31                  go sm.handleSync(messages)
32                  sm.setSyncNeeded(false)
33                  os.Truncate(filenameToStoreData, 0)
34              }
35          }
36          time.Sleep(time.Duration(sm.StreamingFrequency) * time.Second)
37          personVitals.GenerateNextRandomPersonVitals()
38      }
39  }
```

Codice 3: Processo di streaming del dispositivo IoT

6.4 Server CoAP

Il server è stato implementato utilizzando il package *go-coap* [14], che fornisce un'implementazione del protocollo *CoAP* in *Go*. Il server è in grado di gestire le richieste dei dispositivi *IoT* e di consegnare a loro dei comandi per modificare la frequenza di streaming dei dati.

Il server è stato strutturato seguendo un'architettura esagonale, in cui le dipendenze sono state isolate e le interazioni tra i componenti sono state gestite

tramite interfacce. In particolare, sono stati sfruttati i vantaggi di *inversion of control* e *dependency injection*, Codice 4, per gestire le dipendenze tra i componenti del server. Questo ha permesso di separare le responsabilità e di rendere il codice più manutenibile e testabile. Inoltre sono stati utilizzati anche vari *object adapters* per gestire la rappresentazione dei dati attraverso i vari layer dell'architettura.

```go
func main() {
    serverConfig, err := config.LoadServerConfig("../../config/")
    if err != nil {
        log.Fatalf("Error loading server configuration: %v", err)
    }

    db, err := repository.NewDB(serverConfig.DbConfig)
    if err != nil {
        log.Fatalf("Error connecting to the database: %v", err)
    }
    deviceCmdRepo := repository.NewDeviceCmdRepository(db)
    deviceCmdAdapter := repository.NewDeviceCmdAdapter(deviceCmdRepo)
    insertVitalsRepo := repository.NewVitalsRepository(db)
    vitalsAdapter := repository.NewVitalsAdapter(insertVitalsRepo)
    insertVitalsService := service.NewInsertVitalsService(vitalsAdapter, deviceCmdAdapter)
    insertVitalsHandler := handler.NewVitalsHandler(insertVitalsService)
    syncBackupService := service.NewSyncBackupService(vitalsAdapter)
    syncBackupHandler := handler.NewSyncBackupHandler(syncBackupService)

    router := mux.NewRouter()
    router.Handle("/vitals", mux.HandlerFunc(func(w mux.ResponseWriter, r *mux.Message) {
        insertVitalsHandler.InsertVitals(w, r)
    }))
```

```go
    router.Handle("/sync", mux.HandlerFunc(func(w mux.ResponseWriter, r *mux.Message) {
        syncBackupHandler.SyncBackup(w, r)
    }))

    dtlsConfig := auth.ServerDTLSConfig(db.GetDB())
    addr := fmt.Sprintf(":%d", serverConfig.Port)
    log.Fatal(coap.ListenAndServeDTLS("udp", addr, dtlsConfig, router))
}
```

Codice 4: Inizializzazione del server CoAP

6.4.1 Connessione DTLS

Il server e i clients comunicano attraverso il protocollo *CoAP* che utilizza il protocollo *UDP* come protocollo di trasporto sottostante. Per garantire la sicurezza della comunicazione tra il server e i dispositivi *IoT*, è stata implementata una connessione *DTLS* che il *package dtls* [15] mette a disposizione. Questo consente di stabilire una connessione sicura tra il server e i dispositivi, proteggendo i dati trasmessi da possibili attacchi. In particolare, come è possibile vedere nel Codice 5, il sever è in grado di autenticare i client tramite l'utilizzo di *Pre-Shared Key (PSK)*$_G$, che sono condivise tra il server e i dispositivi *IoT* prima della connessione.

```go
func ServerDTLSConfig(dbConn *sql.DB) *piondtls.Config {
    return &piondtls.Config{
        PSK: func(hint []byte) ([]byte, error) {
            deviceSerialNumber := string(hint)
            if psk, err := GetDevicePSK(dbConn, deviceSerialNumber); err == nil && psk != "" {
                return []byte(psk), nil
            }
            return nil, fmt.Errorf("client not authorized: %s", deviceSerialNumber)
        },
        PSKIdentityHint: []byte("server"),
        CipherSuites:    []piondtls.CipherSuiteID{piondtls.
```

```
        TLS_PSK_WITH_AES_128_CCM_8},
    }
}
```

Codice 5: Connessione DTLS tra il server e i dispositivi IoT

6.4.2 Validazione dei dati

Per garantire l'integrità e la validità dei dati ricevuti dai dispositivi *IoT*, il server implementa un processo di validazione dei dati. Questo processo verifica che i dati ricevuti siano conformi alle specifiche attese e che non contengano errori o anomalie. In particolare, il server verifica che i dati siano stati serializzati correttamente in formato *CBOR* e che i campi richiesti siano presenti e abbiano valori validi. In particolare, sono effettuati dei controlli sulle misurazioni dei dati biomedici, ed in caso di valore negativi oppure totalmente fuori scala, il server scarta i dati e non li salva nel database. Inoltre si suppone per convenzione che se il dato di una misurazione è pari a -1, significa che il dispositivo non è in grado di rilevare quel dato in quel determinato momento temporale. Nel Codice 6, viene mostrato un esempio di come viene effettuata la validazione dei dati ricevuti dal server grazie al *package validator* [18]. Questo *package* permette di definire delle regole di validazione per i campi della struttura dati. In questo caso, si verifica che il valore della saturazione dell'ossigeno nel sangue (*SpO2*) sia compreso tra 70 e 100, oppure che sia pari a -1, per indicare che il dispositivo non è in grado di rilevare il dato.

```go
type Saturation struct {
    SpO2    int    `cbor:"s" validate:"required,eq=-1|gte=70,lte=100"`
}
```

Codice 6: Metadati per la validazione dei dati biomedici

6.4.3 Business logic

Nella logica di business, Codice 7, dopo che i aver effettuato i controlli di validazione sulla serializzazione dei dati nell'*application logic*, il server valida il dato ricevuto con il *package validator* [18], e se il dato è valido, lo salva nel database. Inoltre, il server ricerca se ci sono comandi da consegnare al

dispositivo *IoT* che ha inviato i dati, in caso affermativo, il server risponde con un comando che il dispositivo dovrà eseguire.

```go
package coapservices

func (s InsertVitalsService) InsertVitals(message domain.VitalsMessage) (domain.DeviceCmd, error) {
  validate := validator.New(validator.WithRequiredStructEnabled())
  if err := validate.Struct(message); err != nil {
    return domain.DeviceCmd{}, coaperrors.ErrInvalidData
  }

  err := s.insertVitalsPortAdapter.InsertVitals(message)
  if err != nil {
    return domain.DeviceCmd{}, err
  }

  cmd, err := s.getDeviceCmdPortAdapter.GetLastCmd(message.SerialNumber)
  if err != nil {
    return domain.DeviceCmd{}, err
  }

  return cmd, nil
}
```

Codice 7: Validazione dei dati biomedici ricevuti dal server

In maniera simile, il server gestisce anche la sincronizzazione dei dati inviati dai dispositivi *IoT* quando si riconnettono al server dopo un periodo di inattività. In questo caso, il server riceve dei *batch* di dati e li salva all'interno del database.

6.5 API HTTP

Al fine di poter inviare dei comandi ai dispositivi *IoT*, è stata implementata un'*API HTTP*. L'*API* è stata implementata utilizzando il framework *Gin* [19],

che offre un'interfaccia semplice e veloce per la creazione di applicazioni web in *Go*.

Come per il server *CoAP*, anche l'*API HTTP* è stata strutturata seguendo un'architettura esagonale, in cui le dipendenze sono state isolate e le interazioni tra i componenti sono state gestite tramite interfacce. In particolare, sono stati sfruttati i vantaggi di *inversion of control* e *dependency injection*, per gestire le dipendenze tra i componenti dell'*API*.

6.5.1 Comandi

Sono presenti tre *endpoint* principali per l'invio dei comandi ai dispositivi *IoT*, ognuno dei quali è associato ad una richiesta *POST*:

- "/start_streaming": concepito per avviare uno streaming dei dati in maniera più frequente rispetto alla frequenza di default, per simulare un invio più simile al concetto di *real-time*. Nel Codice 8, viene mostrato un esempio di richiesta dove "frequency" è il numero di secondi che intercorrono tra i vari invii dei dati, mentre il "timeout" indica per quanto tempo il dispositivo deve continuare a inviare i dati con questa frequenza prima di fermarsi;
- "/stop_streaming": utilizzato per fermare uno streaming precendentemente avviato con il comando di "start_streaming" prima che scada il timeout, in questo caso il dispositivo smetterà di inviare i dati e tornerà alla frequenza di default;
- "/new_config": per impostare una nuova frequenza di default dello streaming dei dati.

```json
{
    "device_serial_number": "ABC-4G7T-K9V2",
    "frequency": 1,
    "timeout": 60
}
```

Codice 8: Esempio di richiesta per avviare lo streaming dei dati

6.5.2 Raccolta di dati aggregati

In aggiunta ai comandi, l'API *HTTP* fornisce anche due *endpoint* per la raccolta di dati aggregati sui dispositivi *IoT*. Questo consente di ottenere informazioni statistiche sui dati biomedici raccolti dai dispositivi.

In particolare, sono disponibili i seguenti *endpoint* che mettono a disposizione delle richieste *GET*:

- "/fever": per ottenere informazioni sui casi di febbre, ovvero quando la temperatura corporea supera una certa soglia. Infatti è possibile specificare un parametro "threshold" per definire la soglia di temperatura da considerare come febbre. Inoltre, è necessario specificare un parametro "device" per filtrare i risultati in base al numero seriale del dispositivo *IoT*, oltre al periodo in cui si vogliono ottenere i dati aggregati. In particolare, è possibile ottenere quante volte il dispositivo ha registrato una temperatura corporea superiore alla soglia definita, con i relativi massimi e minimi registrati oltre ad un valore medio.
- "/maxmin_history": consente di ottenere i valori minimi e massimi, per ogni intervallo specificato, delle misurazioni dei dati biomedici, come la frequenza cardiaca, la pressione sanguigna, la temperatura corporea e la saturazione dell'ossigeno nel sangue. Anche in questo caso è necessario specificare un parametro "device" per filtrare i risultati in base al numero seriale del dispositivo *IoT*, oltre al periodo in cui si vogliono ottenere i dati aggregati.

6.5.3 Business logic

La logica di business dell'*API HTTP* comprende l'inserimento dei comandi nel database e la gestione delle richieste per la raccolta di dati aggregati. In particolare, nel Codice 9 viene mostrato un esempio di come viene gestita la richiesta per l'inserimento di un comando. In questo caso, dopo aver validato il corpo della richiesta nell'*application logic*, il server verifica che il dispositivo sia autorizzato ed in caso positivo inserisce il comando nel database e restituisce il comando appena creato. Le richieste per la raccolta di dati aggregati sono gestite in maniera simile, dopo aver autorizzato il dispositivo, il *service* richiama il metodo della porta per recuperare i dati aggregati dal database.

```go
package httpservices

func (s *InsertDeviceCmdService) InsertCmd(cmd domain.DeviceCmd) (domain.DeviceCmd, error) {
  exists, err := s.verifySerialNumberPortAdapter.VerifySerialNumber(cmd.SerialNumber)
  if err != nil {
    return domain.DeviceCmd{}, httperrors.ErrDatabaseError
  }
  if !exists {
    return domain.DeviceCmd{}, httperrors.ErrInvalidSerialNumber
  }

  insertedCmd, err := s.insertDeviceCmdPortAdapter.InsertCmd(cmd)
  if err != nil {
    return domain.DeviceCmd{}, httperrors.ErrDatabaseError
  }

  return insertedCmd, nil
}
```

Codice 9: Gestione della richiesta per l'inserimento di un comando

6.6 Database

Il database utilizzato per il prototipo è *TimescaleDB* [24], un'estensione di *PostgreSQL* progettata per gestire dati temporali e serie storiche.

L'adozione di *TimescaleDB* ha permesso di avere una base di dati che si integrasse in maniera efficace con la dashboard per la visualizzazione dei dati raccolti dai dispositivi *IoT*.

Inoltre è stato utilizzato per favorire l'aggregazione dei dati e per eseguire query complesse sui dati biomedici raccolti dai dispositivi, come ad esempio quella per l'*endpoint* "maxmin_history" descritta alla Sezione 6.5.2.

In particolare, la query riportata in Codice 10 consente di raggruppare i dati biomedici per intervalli di tempo *(bucket)* e per dispositivo, calcolando il conteggio dei dati e i valori minimi e massimi delle misurazioni dei dati biomedici.

```sql
SELECT time_bucket($1, time) AS bucket, device, COUNT(*),
MIN(heart_rate) AS max_heartrate, MAX(heart_rate) AS min_heartrate,
MIN(mean_pressure) AS min_pressure, MAX(mean_pressure) AS max_pressure,
MIN(core_temp) AS min_temp, MAX(core_temp) AS max_temp,
MIN(sp02) AS min_sp02, MAX(sp02) AS max_sp02
FROM vitals
WHERE time > $2 AND time < $3 AND device = $4
GROUP BY bucket, device
ORDER BY bucket DESC;
```

Codice 10: Query per la raccolta dei dati aggregati per l'endpoint maxmin_history

6.7 Dashboard

Per la visualizzazione dei dati raccolti dai dispositivi *IoT*, è stato utilizzato *Grafana* [25], un potente strumento di visualizzazione dei dati che consente di creare dashboard interattive e personalizzate.

La dashboard è stata configurata per visualizzare i dati biomedici raccolti da un particolare dispositivo *IoT* configurabile in tempo reale.

In Figura 16 è mostrata un'anteprima della dashboard, che include grafici per le misurazioni dei dati biomedici, come la frequenza cardiaca, la pressione sanguigna, la temperatura corporea e la saturazione dell'ossigeno nel sangue.

Figura 16: Dashboard di Grafana per la visualizzazione dei dati

6.8 Testing

Il prototipo è stato sottosposto a una serie di test per verificare il corretto funzionamento di ogni componente architetturale e per garantire che le funzionalità sviluppate fossero conformi ai requisiti del progetto.

Il testing è stato effettuato utilizzando il *framework* di testing di *Go*, che consente di scrivere test di unità e di integrazione in modo semplice e veloce. Inoltre, sono stati utilizzati anche altri *package* per facilitare il processo di testing, come *sqlmock* [22] per simulare il database e *testify* [21] per le asserzioni nei test. Alla fine dello sviluppo del prototipo, è stato raggiunto un buon livello di copertura del codice, con oltre l'80% delle righe di codice coperte dai test.

In aggiunta, sono stati eseguiti anche altri test grazie al tool *hurl* [27], che ha consentito di testare l'*API HTTP* in modo semplice e veloce, nel Codice 11 viene riportato un esempio.

```
1   # Il dispositivo è autorizzato e richiesta valida      hurl
2   POST http://localhost:8080/start_streaming
```

```
{
    "device_serial_number": "ABC-4G7T-K9V2",
    "frequency": 1,
    "timeout": 30
}
HTTP 200
[Asserts]
jsonpath "$.data.device_serial_number" == "ABC-4G7T-K9V2"
jsonpath "$.data.frequency" == 1
jsonpath "$.data.timeout" == 30


# Test per il caso in cui il dispositivo non è autorizzato
POST http://localhost:8080/start_streaming
{
    "device_serial_number": "ABC-4G7T-K9V3",
    "frequency": 1,
    "timeout": 30
}
HTTP 401
```

Codice 11: Esempio di test dell'API HTTP con Hurl

6.9 Limiti e criticità della soluzione

Nonostante il prototipo sviluppato abbia raggiunto gli obiettivi prefissati, sono emersi alcuni limiti e criticità che potrebbero essere affrontati in futuri sviluppi. In particolare, sono stati riscontrati i seguenti aspetti:

- La gestione della sincronizzazione dei dati in caso di disconnessione del dispositivo *IoT* dal server potrebbe essere migliorata. Attualmente, il dispositivo salva i dati in un file locale e li invia successivamente quando la connessione viene ripristinata. Tuttavia, questo approccio potrebbe portare a una perdita di dati in caso di problemi durante il processo di sincronizzazione. Una possibile soluzione potrebbe essere quella di implementare un meccanismo di *retry* per garantire che i dati vengano inviati correttamente al server, anche in caso di errori di rete o di timeout.

- In alcuni casi, i server hanno dei *rate limit* per le richieste, che potrebbero causare problemi durante il processo di sincronizzazione dei dati. Attualmente, il dispositivo invia i dati in *batch* di 10 misurazioni, ma questo potrebbe non essere sufficiente in caso di un numero elevato di dispositivi IoT che si connettono contemporaneamente al server. Una possibile soluzione potrebbe essere quella di implementare un meccanismo di gestione dei *rate limit*, ad esempio utilizzando un sistema di code o di bilanciamento del carico, per garantire che i dati vengano inviati in modo efficiente e senza sovraccaricare il server.
- Durante lo sviluppo del prototipo, non è stata data troppa importanza alla gestione della scalabilità del sistema. La mancanza di fornitura di possibili soluzioni *cloud* ha reso complicato comprendere quanto effettivamente la soluzione adottata sia scalabile. In un contesto reale, il server potrebbe dover gestire un numero elevato di dispositivi IoT che inviano dati contemporaneamente.

Capitolo 7

Conclusioni

In quest'ultimo capitolo vengono riportate le considerazioni finali sul mio percorso di stage analizzando gli obiettivi raggiunti, alcuni margini di miglioramento e le conoscenze acquisite.

7.1 Raggiungimento degli obiettivi

I principali obiettivi del progetto sono stati raggiunti. L'efficace pianificazione mi ha permesso di completare tutti i requisiti obbligatori in maniera anticipata, così da permettermi di implementare anche la maggior parte dei requisiti desiderabili e facoltativi. Nelle tabelle Tabella 11 Tabella 12 Tabella 13 sono riportati i requisiti presentati al Capitolo 3 e il loro stato attuale:

Requisito	Stato
RFN-1	Soddisfatto
RFN-2	Soddisfatto
RFN-3	Soddisfatto
RFN-4	Soddisfatto
RFN-5	Soddisfatto
RFN-6	Soddisfatto
RFN-7	Soddisfatto
RFN-8	Soddisfatto
RFN-9	Soddisfatto
RFD-1	Soddisfatto
RFD-2	Soddisfatto
RFD-3	Soddisfatto
RFZ-1	Non soddisfatto
RFZ-2	Soddisfatto

Tabella 11: Tabella dello stato dei requisti funzionali

Requisito	Stato
RQN-1	Soddisfatto
RQD-1	Soddisfatto
RQD-2	Soddisfatto
RQD-3	Soddisfatto

Tabella 12: Tabella dello stato dei requisti qualitativi

Requisito	Stato
RVN-1	Soddisfatto
RVN-2	Soddisfatto
RVN-3	Soddisfatto
RVD-1	Soddisfatto
RVD-2	Soddisfatto
RVD-3	Soddisfatto

Tabella 13: Tabella dello stato dei requisti di vincolo

7.2 Margini di miglioramento

Nonostante il progetto sia stato completato con successo, ci sono alcuni margini di miglioramento da poter considerare:

- **Gestione della batteria**: implementazione di un sistema di gestione della batteria che permetta di ottimizzare l'uso dell'energia, implementando una modalità di risparmio energetico che riduca la frequenza di aggiornamento dei dati qualora la batteria sia sotto una certa soglia. Questo può essere utile per prolungare la durata della batteria in situazioni di emergenza.
- **Vincoli di scalabilità**: nel caso fosse stato previsto l'utilizzo di tecnologie come il cloud computing, si sarebbe potuto considerare vincoli di scalabilità per garantire che il sistema potesse gestire un numero crescente di dispositivi senza compromettere le prestazioni.
- **Gestione migliorata delle disconessioni**: l'implementazione di un sistema più robusto per gestire le disconnessioni della rete, che limiti al minimo la perdita di dati durante le interruzioni della connessione.

7.3 Conoscenze acquisite

Durante lo sviluppo del progetto, ho avuto modo di approfondire tecnologie che non avevo mai utilizzato prima, come ad esempio il linguaggio *Go*, le dashboard di *Grafana* e l'estensione di *TimescaleDB* per il database.

Inoltre, ho avuto la possibilità di studiare protocolli per la comunicazione in ambienti *IoT*, utilizzando il protocollo *CoAP* che è meno adottato rispetto ad altre alternative sul mercato, oltre a questo in generale ora ritengo di avere più familiarità nell'ambito delle reti.

Infine, ho avuto modo di migliorare le mie competenze nell'ambito dell'ingegneria del software, in particolare per la pianificazione delle attività e i processi di analisi dei requisiti, progettazione e codifica.

7.4 Valutazione personale

In generale sono soddisfatto del risultato finale del progetto, anche se alcuni aspetti sarebbero stati migliorabili, come per esempio l'utilizzo di alcune tecnologie *cloud*.

L'azienda si è rivelata un luogo molto accogliente e stimolante, con un team di lavoro molto competente e disponibile. Sono grato per l'opportunità che mi è stata data di lavorare su questo progetto e sono fiducioso che le competenze acquisite saranno utili per il mio futuro professionale.

Bibliografia

[1] M31, «La Nostra Storia». Consultato: 12 maggio 2025. [Online]. Disponibile su: https://www.m31.com/?page_id=2

[2] HiveMQ, «MQTT Essentials». Consultato: 27 maggio 2025. [Online]. Disponibile su: https://www.hivemq.com/mqtt/

[3] Wikipedia, «Constrained Application Protocol». Consultato: 2 giugno 2025. [Online]. Disponibile su: https://en.wikipedia.org/wiki/Constrained_Application_Protocol

[4] EQMX Team, «CoAP Protocol: Key Features, Use Cases, and Pros/Cons». Consultato: 2 giugno 2025. [Online]. Disponibile su: https://www.emqx.com/en/blog/coap-protocol

[5] Ian Craggs, «MQTT Vs. HTTP for IoT». Consultato: 7 giugno 2025. [Online]. Disponibile su: https://www.hivemq.com/blog/mqtt-vs-http-protocols-in-iot-iiot/

[6] EQMX Team, «MQTT vs CoAP: Comparing Protocols for IoT Connectivity». Consultato: 7 giugno 2025. [Online]. Disponibile su: https://www.emqx.com/en/blog/mqtt-vs-coap#mqtt-vs-coap-a-comparison

[7] John Keever, «HTTP vs. MQTT: Comparison for IoT». Consultato: 7 giugno 2025. [Online]. Disponibile su: https://www.telit.com/blog/http-vs-mqtt/

[8] Ian Craggs, «MQTT Vs. CoAP for IoT». Consultato: 7 giugno 2025. [Online]. Disponibile su: https://www.hivemq.com/blog/mqtt-vs-coap-for-iot/

[9] ECMA-404, «Introducing JSON». Consultato: 7 giugno 2025. [Online]. Disponibile su: https://www.json.org/json-en.html

[10] Protocol Buffers, «Protocol Buffers». Consultato: 7 giugno 2025. [Online]. Disponibile su: https://protobuf.dev/

[11] MessagePack, «MessagePack». Consultato: 7 giugno 2025. [Online]. Disponibile su: https://msgpack.org/

[12] CBOR, «CBOR». Consultato: 7 giugno 2025. [Online]. Disponibile su: https://cbor.io/

[13] «The Go Programming Language Specification». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://go.dev/ref/spec

[14] «Go-CoAP». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://github.com/plgd-dev/go-coap

[15] «Pion DTLS». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://github.com/pion/dtls

[16] «Viper». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://pkg.go.dev/github.com/spf13/viper

[17] «CBOR Codec». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://github.com/fxamacker/cbor

[18] «Package validator». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://pkg.go.dev/github.com/go-playground/validator/v10

[19] «Gin Web Framework». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://pkg.go.dev/github.com/gin-gonic/gin

[20] «pq - A pure Go postgres driver for Go's database/sql package». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://github.com/lib/pq

[21] «Testify - Thou Shalt Write Tests». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://github.com/stretchr/testify

[22] «Sql driver mock for Golang». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://github.com/DATA-DOG/go-sqlmock

[23] «Docker». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://www.docker.com/

[24] timescale, «timescale/timescaledb». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://hub.docker.com/r/timescale/timescaledb

[25] Grafana Labs, «grafana/grafana». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://hub.docker.com/r/grafana/grafana

[26] «pgAdmin4». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://www.pgadmin.org/

[27] «Hurl - HTTP Request Testing Tool». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://hurl.dev/

[28] «Vizb - Go Benchmark Visualization Tool». Consultato: 27 giugno 2025. [Online]. Disponibile su: https://github.com/goptics/vizb

[29] «Sonic - A blazingly fast JSON serializing & deserializing library». Consultato: 29 giugno 2025. [Online]. Disponibile su: https://github.com/bytedance/sonic

[30] «Gzip Compression». Consultato: 29 giugno 2025. [Online]. Disponibile su: https://pkg.go.dev/compress/gzip

[31] «Brotli Compression». Consultato: 29 giugno 2025. [Online]. Disponibile su: https://pkg.go.dev/github.com/andybalholm/brotli