ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Kotlin Multiplatform e PulvReAKt: analisi e prototipazione di applicazioni IoT eterogenee

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Prof. Viroli Mirko

Candidato
Coli Alessandro

Correlatore
Dott. Farabegoli Nicolas

I Sessione di Laurea
Anno Accademico 2023-2024

A te che ci sei sempre stato
e sempre sei con me anche se te ne sei andato.
A te che tanto mi hai insegnato
e sempre mi hai supportato.

A te che più di chiunque altro avrei voluto qua.
A mio nonno Gianni.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Structure of the Thesis

Coli Alessandro: At the end, describe the structure of the paper

Capitolo 2

Contesto

2.1 Sviluppo Multiplatform

Lo sviluppo multiplatform è diventato un approccio sempre più diffuso per la creazione di applicazioni che devono essere eseguite su diverse piattaforme senza dover scrivere codice specifico per ciascuna di esse. Questo approccio consente agli sviluppatori di massimizzare la condivisione del codice, riducendo i costi di sviluppo e manutenzione e garantendo una maggiore coerenza e compatibilità tra le diverse versioni dell'applicazione.

Esistono diversi approcci per lo sviluppo multiplatform, ciascuno con le proprie caratteristiche e vantaggi:

- **Codice Condiviso**: In questo approccio il codice condiviso viene scritto una volta e viene compilato per essere eseguito su diverse piattaforme. Questo può essere realizzato utilizzando linguaggi di programmazione che supportano la compilazione multiplatform, come Kotlin, o utilizzando framework e librerie che consentono la condivisione del codice tra piattaforme diverse.

- **Interfacce di Programmazione (API)**: Un altro approccio consiste nel definire un'API comune che fornisce un'interfaccia uniforme per l'accesso alle funzionalità dell'applicazione su diverse piattaforme. Questo consente di implementare la logica dell'applicazione specifica per ciascuna piattaforma, mantenendo al contempo una coerenza nell'interfaccia e nei dati condivisi.

- **Virtualizzazione e Contenitori**: Utilizzando tecnologie di virtualizzazione e contenitori, è possibile creare un ambiente uniforme per l'esecuzione dell'applicazione su diverse piattaforme. Questo approccio permette di isolare l'applicazione dall'infrastruttura sottostante, garantendo una maggiore portabilità e flessibilità.

Lo sviluppo multiplatform offre una serie di vantaggi significativi:

- **Massimizzazione della Condivisione del Codice**: Condividendo il codice tra le diverse piattaforme è possibile ridurre il lavoro ripetitivo e aumentare l'efficienza dello sviluppo.

- **Riduzione dei Costi di Sviluppo**: Riducendo il numero di linee di codice da scrivere e gestire è possibile ridurre i costi complessivi di sviluppo e manutenzione dell'applicazione.

- **Coerenza e Compatibilità**: Mantenendo una sola base di codice per tutte le versioni dell'applicazione è possibile garantire una maggiore coerenza e compatibilità tra le diverse versioni.

- **Flessibilità**: Utilizzando approcci multiplatform gli sviluppatori possono adottare una varietà di tecnologie e linguaggi di programmazione per soddisfare le esigenze specifiche del progetto.

2.2 Programmazione Distribuita

Nel contesto dell'informatica moderna, la programmazione distribuita su piattaforme eterogenee emerge come un approccio fondamentale per affrontare le sfide della interconnessione in un ambiente eterogeneo di dispositivi e sistemi. Questo paradigma si basa sulla distribuzione delle responsabilità e delle risorse su una varietà di dispositivi e piattaforme, che possono differire per architettura hardware, sistema operativo e linguaggio di programmazione.

La programmazione distribuita su piattaforme eterogenee è guidata dalla necessità di integrare e coordinare dispositivi e sistemi diversi per creare soluzioni

innovative e scalabili. Questo approccio richiede la collaborazione e la comunicazione efficace tra nodi distribuiti, che possono trovarsi in ambienti eterogenei come cloud, dispositivi embedded, server on-premise e dispositivi mobili.

Alcune tecnologie e approcci chiave utilizzati nella programmazione distribuita su piattaforme eterogenee includono:

- **Protocolli di Comunicazione Universali**: Questi protocolli consentono la comunicazione tra dispositivi e sistemi eterogenei, fornendo un'interfaccia standardizzata per lo scambio di dati e messaggi. Esempi di protocolli universali includono HTTP[1], Message Queuing Telemetry Transport (MQTT) e gRPC, che consentono la comunicazione su diverse piattaforme e architetture.

- **Middleware Distribuito**: Il middleware distribuito fornisce un livello di astrazione tra le applicazioni e l'infrastruttura sottostante, consentendo la trasparenza della distribuzione su piattaforme eterogenee. Questo può includere servizi di messaggistica, servizi di gestione delle transazioni e sistemi di caching distribuiti.

- **Containerizzazione**: Le tecnologie di containerizzazione, come Docker e Kubernetes, consentono di confezionare, distribuire e gestire applicazioni su piattaforme eterogenee in modo uniforme. I container forniscono un'unità di distribuzione leggera e isolata, garantendo la portabilità e la scalabilità delle applicazioni su diverse infrastrutture.

- **Orchestrazione Multi-Cloud**: L'orchestrazione multi-cloud permette di distribuire carichi di lavoro su più fornitori di servizi cloud, garantendo la ridondanza, la resilienza e la flessibilità delle applicazioni su piattaforme eterogenee

[1]HTTP (HyperText Transfer Protocol) è un protocollo di comunicazione utilizzato per il trasferimento di ipertesti, come pagine web, su reti informatiche, principalmente il World Wide Web.

2.3 Protocollo MQTT

Nel vasto panorama dell' Internet of Thing (IoT), dove milioni di dispositivi sono interconnessi per scambiare dati e informazioni in tempo reale, Message Queuing Telemetry Transport (MQTT) si distingue come uno dei protocolli di comunicazione più importanti e ampiamente adottati, in quanto si presenta come una soluzione elegante per risolvere le sfide di comunicazione proposte in ambito IoT. Le sue caratteristiche distintive includono il basso utilizzo di banda e CPU, che lo rendono particolarmente adatto per i dispositivi con risorse limitate, e il modello di comunicazione publish-subscribe, che consente la trasmissione efficiente dei messaggi anche grazie al protocollo Transmission Control Protocol (TCP) a cui si appoggia per la trasmissione dei messaggi.

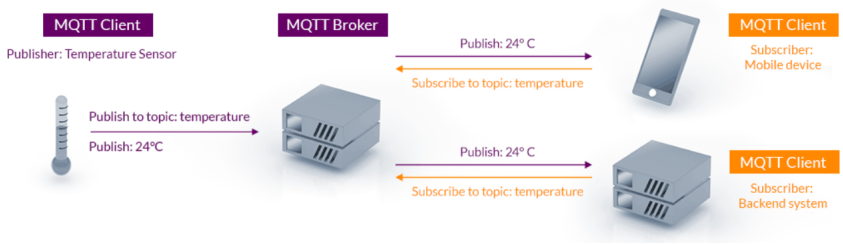

Figura 2.1: Esempio di comunicazione tramite MQTT

Nel modello publish-subscribe di MQTT i dispositivi si dividono in due ruoli principali: i publisher, che inviano i messaggi su specifici topic, e i subscriber, che si iscrivono a questi topic per ricevere i messaggi pertinenti. Questo approccio consente una comunicazione flessibile e scalabile, in cui i dispositivi possono interagire in modo dinamico senza la necessità di conoscere direttamente gli indirizzi dei destinatari.

Il protocollo MQTT è composto da molteplici entità di due tipologie: un singolo broker di messaggi e un numero variabile di client. Il broker MQTT è un server con la funzione di ricevere i messaggi da tutti i client e di instradarli verso i destinatari interessati. Un client MQTT è un qualsiasi dispositivo che esegue una libreria MQTT e si connette a un broker MQTT e può svolgere sia il ruolo di publisher che di subscriber. Quando un publisher ha nuovi dati da trasmettere, invia un messaggio con i dati al broker a cui è connesso. Il broker, quindi, distribuisce le

informazioni a tutti i client che si sono iscritti a quell'argomento. Non è necessario che il publisher conosca il numero o la posizione degli iscritti e questi ultimi, a loro volta, non necessitano di alcuna informazione sui publisher.

Un aspetto cruciale del protocollo MQTT è la sua capacità di mantenere aperte le connessioni tra i dispositivi per lunghi periodi, riducendo al minimo la latenza e ottimizzando l'efficienza della comunicazione. Questo risulta particolarmente importante in contesti dove è essenziale uno scambio continuo e tempestivo di dati in tempo reale, permettendo un monitoraggio costante e affidabile. Tale caratteristica rende MQTT ideale per applicazioni che richiedono una trasmissione dati ininterrotta e reattiva, garantendo prestazioni ottimali anche in ambienti con risorse limitate e connettività intermittente.

I principali tipi di messaggi MQTT sono quattro:

- **Connect**: il client attende che venga stabilita una connessione con il broker e crea un collegamento tra i nodi;
- **Disconnect**: attende se il client MQTT deve terminare delle operazioni e poi chiude la sessione TCP/IP;
- **Publish**: il client pubblica un messaggio nella classe specificato;
- **Subscribe**: iscrive il client all'argomento specificato;

2.3.1 Formato pacchetti MQTT

Un pacchetto MQTT è composto da diverse parti strutturate in modo specifico, può essere suddiviso nelle seguenti componenti principali:

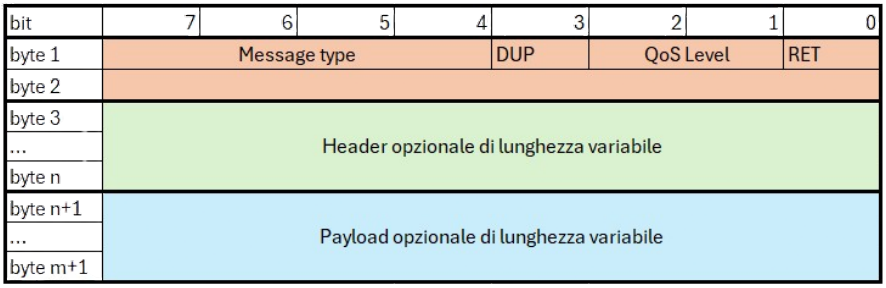

Figura 2.2: formato di un pacchetto MQTT

- **Fixed Header (Intestazione Fissa):** Ogni pacchetto MQTT contiene un'intestazione fissa che è obbligatoria e ha una lunghezza di 2 byte. L'intestazione fissa include il tipo di messaggio, il livello di Quality of Service (QoS), la lunghezza rimanente del pacchetto e informazioni sui flag di controllo tra cui DUP (duplicate) che indica se è il primo tentativo di invio di tale pacchetto e RET (retain) che indica se tale pacchetto deve essere salvato nel server. I tipi di messaggio possono includere CONNECT, PUBLISH, SUBSCRIBE, UNSUBSCRIBE, DISCONNECT.

- **Variable Header (Intestazione Variabile):** L'intestazione variabile è opzionale e viene utilizzata in base al tipo di pacchetto. Questa sezione può includere diversi campi come l'identificatore del pacchetto, i codici di ritorno e altre informazioni specifiche del tipo di messaggio. Ad esempio, nei pacchetti di tipo PUBLISH, l'intestazione variabile include il topic del messaggio.

- **Payload (Corpo del Messaggio):** Il corpo del messaggio o payload è anch'esso opzionale e contiene i dati effettivi da trasmettere. La lunghezza del payload può variare e nei pacchetti PUBLISH, esso contiene il contenuto del messaggio che viene inviato agli abbonati al topic specifico. La presenza e il contenuto del payload dipendono dal tipo di messaggio specifico.

2.3.2 Quality of Service (QoS)

MQTT offre funzionalità avanzate per la gestione delle connessioni, la sicurezza e la qualità del servizio (QoS), che consentono di personalizzare e ottimizzare la comunicazione in base alle esigenze specifiche dell'applicazione. Queste caratteristiche lo rendono un protocollo estremamente flessibile e adattabile, in grado di soddisfare una vasta gamma di requisiti e casi d'uso nell'ambito dell'IoT. Ogni livello di QoS in MQTT offre un trade-off tra affidabilità e overhead di rete, permettendo agli sviluppatori di scegliere il livello più appropriato in base alle esigenze specifiche dell'applicazione, semplicemente modificando il valore dei due bit che indicano il livello QoS nei pachetti, tramite le API fornite dalle librerie.

Figura 2.3: funzionamento dei diversi livelli di QoS

Il livello di QoS 0, noto anche come 'al massimo una volta' (at most once), garantisce che i messaggi vengano inviati senza conferma di ricezione. I messaggi vengono consegnati in modo best effort, il che significa che non ci sono garanzie di consegna. Questo livello di QoS è utilizzato quando la perdita di messaggi occasionali è accettabile, come nelle applicazioni dove la latenza è più importante dell'affidabilità, ad esempio per il monitoraggio di dati in tempo reale che vengono aggiornati frequentemente.

Il livello di QoS 1, noto anche come 'almeno una volta' (at least once), garantisce che i messaggi vengano consegnati almeno una volta al destinatario. Ciò implica che il mittente continua a inviare il messaggio finché non riceve una conferma di ricezione (acknowledgement) dal destinatario. Tuttavia, questo può portare alla possibilità di ricevere duplicati del messaggio. Il livello QoS 1 è utilizzato quando è importante che i messaggi vengano ricevuti, anche se ciò significa ricevere duplicati, ad esempio per le transazioni di dati critici che devono essere garantite.

Il livello di QoS 2, noto anche come 'esattamente una volta' (exactly once), garantisce che ogni messaggio venga consegnato esattamente una volta al destinatario. Questo livello di QoS utilizza un meccanismo di handshake a quattro fasi per assicurare che i messaggi non vengano duplicati e non vadano persi, garantendo la massima affidabilità. Questo livello di QoS è utilizzato in scenari dove è fondamentale evitare duplicati e garantire la consegna esatta di ogni messaggio, come nelle applicazioni finanziarie o nei sistemi di controllo industriale.

2.3.3 MQTT over WebSocket

MQTT over WebSocket rappresenta una combinazione potente che estende le capacità del protocollo MQTT nel contesto delle applicazioni web. WebSocket è un protocollo di comunicazione che fornisce un canale full-duplex su una singola connessione TCP, consentendo una comunicazione bidirezionale in tempo reale tra il client e il server.

L'integrazione di MQTT con WebSocket permette ai client basati su web di comunicare direttamente con un broker MQTT, eliminando la necessità di intermediari o di soluzioni di polling inefficaci. Questo è particolarmente utile per le applicazioni IoT che richiedono aggiornamenti in tempo reale e un'interazione continua tra dispositivi e interfacce utente basate su web. Le principali caratteristiche di MQTT over WebSocket includono:

- **Compatibilità Browser**: Consente ai browser web di stabilire connessioni MQTT direttamente, facilitando lo sviluppo di applicazioni web interattive e in tempo reale.
- **Efficienza della Connessione**: Utilizza una singola connessione TCP per gestire sia il canale di comunicazione MQTT che la trasmissione dei dati, riducendo il sovraccarico di rete.
- **Sicurezza**: Supporta la crittografia SSL/TLS per garantire che la comunicazione tra il client e il server sia sicura, proteggendo i dati trasmessi da intercettazioni e attacchi.
- **Flessibilità**: Permette una facile integrazione con le infrastrutture web esistenti, utilizzando lo stesso stack di tecnologia web e riducendo la complessità del sistema.

MQTT over WebSocket è ideale per le applicazioni che necessitano di aggiornamenti frequenti e rapidi, come dashboard di monitoraggio, sistemi di allarme in tempo reale, e applicazioni di controllo remoto. La combinazione di MQTT e WebSocket offre una soluzione robusta e scalabile per la comunicazione in tempo reale nel mondo dell'IoT.

2.4 Dependency Injection

Nel contesto dell'ingegneria del software, la *Dependency Injection* rappresenta un pattern di progettazione fondamentale per la gestione delle dipendenze tra i vari componenti di un sistema. Questo approccio promuove un'architettura modulare e facilita la manutenzione, il testing e l'estensibilità del software.

La Dependency Injection è una tecnica attraverso la quale un oggetto (il *client*) riceve le proprie dipendenze da un oggetto esterno (l' *injector*), anziché crearle autonomamente. Le dipendenze sono componenti di cui il client ha bisogno per eseguire le proprie funzionalità. In altre parole, la Dependency Injection inverte il controllo della creazione delle dipendenze, trasferendo questa responsabilità a un container o a un framework.

Esistono tre principali varianti di Dependency Injection:

- **Constructor Injection**: Le dipendenze vengono fornite tramite il costruttore dell'oggetto. Questo metodo garantisce che le dipendenze siano disponibili al momento della creazione dell'oggetto.
- **Setter Injection**: Le dipendenze vengono fornite tramite metodi setter. Questo approccio consente una maggiore flessibilità nella configurazione delle dipendenze dopo la creazione dell'oggetto.
- **Interface Injection**: L'oggetto client espone un'interfaccia che permette al container di fornire le dipendenze. Questo metodo è meno comune ma può essere utile in contesti specifici.

L'adozione della Dependency Injection offre numerosi vantaggi, tra cui:

- **Maggiore modularità**: I componenti del sistema possono essere sviluppati e testati in isolamento, migliorando la coesione e riducendo l'accoppiamento.
- **Facilità di testing**: Le dipendenze possono essere sostituite con mock o stub, semplificando il processo di unit testing.
- **Manutenzione e estensibilità migliorate**: La separazione delle preoccupazioni facilita la modifica e l'estensione delle funzionalità del sistema senza impattare sul resto del codice.

- **Inversione del controllo**: La Dependency Injection implementa il principio di inversione del controllo, permettendo al framework di gestire il ciclo di vita degli oggetti e le loro dipendenze.

Diversi framework e container sono stati sviluppati per supportare la Dependency Injection in vari linguaggi di programmazione. Un esempio prominente, in particolare per l'ecosistema Kotlin, è **Koin**: un framework di Dependency Injection per Kotlin che si distingue per la sua semplicità e leggerezza. Koin utilizza una DSL (Domain Specific Language) per definire le dipendenze in modo chiaro e conciso, rendendo l'integrazione e la configurazione estremamente intuitive. Koin è particolarmente apprezzato nell'ambito dello sviluppo di applicazioni Android.

La Dependency Injection è un pattern di progettazione che offre una soluzione elegante alla gestione delle dipendenze in sistemi software complessi. La sua adozione consente di sviluppare applicazioni modulari, testabili e facilmente manutenibili, contribuendo a migliorare la qualità complessiva del software. Nelle sezioni successive, esploreremo più in dettaglio i vari aspetti della Dependency Injection, i suoi benefici e come implementarla efficacemente utilizzando diversi framework, con un focus particolare su Koin per applicazioni Kotlin.

2.5 PulvReAKt

PulvReAKt è un framework leggero e multipiattaforma per Kotlin, progettato per facilitare la pulverizzazione dei sistemi, ovvero la suddivisione di applicazioni complesse in muyltemplici componenti modulari che possono essere distribuiti su diverse piattaforme. Il framework supporta Java Virtual Machine (JVM), JavaScript (JS) e alcune piattaforme native, come Linux, macOS e iOS. PulvReAKt consente di definire, configurare e distribuire componenti software utilizzando protocolli di comunicazione come MQTT e RabbitMQ, rendendolo particolarmente adatto per lo sviluppo di applicazioni distribuite e sistemi IoT. Consente di suddividere l'applicazione in componenti indipendenti, facilitando lo sviluppo e la manutenzione. Supporta JVM, JS e varie piattaforme native, permettendo l'esecuzione dei componenti in diversi ambienti.

2.6 Kotlin

La scelta del linguaggio utilizzato per questo progetto è stata relativamente semplice, poiché il progetto PulvReAKt, nel quale è contestualizzato, è scritto interamente in Kotlin con target Multiplatform. La maggior parte dello sviluppo è stata eseguita in Kotlin, con l'unica eccezione dell'utilizzo di una libreria JavaScript per lo sviluppo della parte JS, integrata comunque all'interno di Kotlin grazie alla sua interoperabilità.

Kotlin è un linguaggio di programmazione moderno sviluppato da JetBrains, noto per il suo approccio pragmatico e la sua forte interoperabilità con Java. Dal suo lancio nel 2011, Kotlin ha guadagnato rapidamente popolarità, diventando nel 2017 il linguaggio ufficiale per lo sviluppo di applicazioni Android.

Kotlin è stato progettato per essere conciso, riducendo la quantità di codice boilerplate necessario. Grazie alle funzioni lambda e alle estensioni di funzioni, il codice scritto in Kotlin risulta più leggibile e mantenibile rispetto a Java. Inoltre, Kotlin si propone di minimizzare gli errori a runtime, spostando la rilevazione degli errori al tempo di compilazione.

Uno dei punti di forza di Kotlin è la sua totale interoperabilità con Java. Questo significa che il codice Kotlin può chiamare ed essere chiamato da codice Java senza difficoltà. Questo facilita la migrazione graduale da Java a Kotlin nei progetti esistenti, permettendo agli sviluppatori di integrare Kotlin in maniera incrementale.

Kotlin introduce le coroutines, una potente astrazione per la programmazione asincrona e concorrente. Le coroutines offrono un modo semplice ed efficiente per gestire operazioni che altrimenti richiederebbero la gestione complessa dei thread. Esse permettono di scrivere codice asincrono in uno stile sequenziale, migliorando la leggibilità e riducendo il rischio di errori.

Le coroutines in Kotlin sono supportate a livello di linguaggio, con una libreria standard che fornisce molte funzionalità pronte all'uso. Ecco alcuni dei principali vantaggi delle coroutines:

- **Semplicità del codice asincrono:** Le coroutines permettono di scrivere codice che sembra sincrono ma che viene eseguito in modo asincrono, sem-

plificando notevolmente la gestione delle operazioni I/O e altre operazioni bloccanti.

- **Efficienza delle risorse:** Le coroutines sono molto leggere rispetto ai thread tradizionali. Possono essere sospese e riprese senza un costo significativo, consentendo una gestione efficiente delle risorse.

- **Facilità di cancellazione:** Le coroutines forniscono meccanismi per gestire la cancellazione delle operazioni in corso, riducendo il rischio di risorse bloccate o perdite di memoria.

- **Composizione semplice:** Le coroutines permettono di comporre in modo semplice operazioni asincrone, rendendo il codice più modulare e facile da mantenere.

L'utilizzo delle coroutines in Kotlin rappresenta un approccio moderno alla programmazione concorrente, semplificando notevolmente il processo di scrittura e gestione del codice asincrono.

La sintassi concisa e le caratteristiche avanzate di Kotlin contribuiscono ad aumentare la produttività degli sviluppatori. Funzionalità come le funzioni di estensione, la gestione avanzata delle nullità e le espressioni lambda riducono il codice boilerplate e semplificano lo sviluppo, permettendo agli sviluppatori di concentrarsi sulla logica applicativa piuttosto che sui dettagli implementativi.

La forte tipizzazione e le funzionalità di sicurezza integrata di Kotlin contribuiscono a migliorare la qualità del codice. Il compilatore di Kotlin è in grado di rilevare molti tipi di errori a tempo di compilazione, riducendo il numero di bug presenti nel software finito. Inoltre, la gestione avanzata delle nullità e le funzioni di estensione aiutano a scrivere codice più robusto e meno incline agli errori.

Uno degli aspetti più potenti di Kotlin è la sua capacità di supportare lo sviluppo multiplatform. Grazie al quale gli sviluppatori possono scrivere un unico set di codice Kotlin che può essere condiviso e utilizzato su diverse piattaforme, come la JVM, JavaScript e la compilazione nativa. Questo approccio consente agli sviluppatori di massimizzare la condivisione del codice, riducendo al minimo la duplicazione e semplificando la manutenzione dell'applicazione su più piattaforme.

Capitolo 3

Requisiti

3.1 Requisiti funzionali

Il progetto si propone di offrire un gestore delle comunicazioni versatile, capace di adattarsi a diversi ambiti applicativi. In particolare, è stato progettato pensando alle funzionalità richieste da un sistema distribuito ed eterogeneo come PulvReAKt, nel quale è stato successivamente integrato. PulvReAKt è caratterizzato dalla possibilità di avere componenti dislocate in posizioni geografiche diverse, con capacità e risorse variabili. Questo sistema deve essere quindi in grado di operare efficacemente in contesti molto diversificati, incluso l'ambito IoT, dove le macchine coinvolte dispongono spesso di risorse hardware limitate. Inoltre, il sistema deve essere progettato per funzionare correttamente anche in ambienti con connettività di rete instabile o variabile, garantendo prestazioni adeguate indipendentemente dalle condizioni di rete.

Uno degli obiettivi principali del progetto è assicurare che le comunicazioni avvengano in modo efficiente e affidabile. Particolare attenzione è stata posta nel garantire che i messaggi vengano recapitati esclusivamente ai destinatari interessati. Questo richiede l'implementazione di meccanismi sofisticati di gestione dei messaggi, che consentano un indirizzamento preciso e prevengano lo spreco di risorse. La capacità di indirizzare correttamente i messaggi è fondamentale per evitare sovraccarichi di rete e per mantenere l'integrità e la sicurezza delle comunicazioni.

La gestione dei canali di comunicazione deve essere trasparente per l'utente

finale. Tuttavia, è responsabilità dell'utente definire quali entità debbano comunicare tra loro, definire un formato per i messaggi e conseguentemente formattarli correttamente, poiché il sistema è progettato per ricevere il messaggio in input, insieme alle entità sorgente e destinazione, e per inviarlo al destinatario corretto mantenendo l'integrità del mittente e del contenuto del messaggio. La formattazione non sarà unificata, se non per quanto riguarda l'incapsulamento necessario per garantire l'invio e la ricezione corretti dei messaggi, utilizzando MQTT.

Un ulteriore aspetto critico del progetto è la necessità di sviluppare test esaustivi. La scrittura di test approfonditi è fondamentale per garantire la robustezza e l'affidabilità del sistema. I test devono coprire un'ampia gamma di scenari, inclusi quelli più comuni e quelli meno frequenti ma potenzialmente problematici. È essenziale che vengano testati tutti i possibili casi d'uso, compresi quelli che coinvolgono condizioni di rete instabili, errori di comunicazione e gestione delle risorse limitate. I test devono inoltre verificare che il sistema risponda correttamente a input non validi e che mantenga l'integrità dei dati durante l'intero processo di comunicazione. Solo attraverso una rigorosa attività di testing è possibile assicurare che il sistema soddisfi i requisiti funzionali e operi come atteso in tutte le casistiche di utilizzo.

3.2 Scalabilità

Il modulo di comunicazione progettato per utilizzare MQTT, deve essere in grado di scalare efficacemente per gestire un numero variabile di entità comunicanti, sia che esse siano poche o molte, sia che queste comunichino spesso o raramente.

Il sistema deve essere modulare per facilitare l'aggiunta o la rimozione di componenti senza compromettere il funzionamento generale. Utilizzando MQTT come middleware per la comunicazione asincrona, deve garantire la capacità di scalare orizzontalmente. È essenziale che il protocollo di comunicazione sia leggero e ad alte prestazioni, permettendo di gestire un elevato numero di connessioni simultanee con un overhead minimo. Per ottimizzare l'uso delle risorse di rete, deve essere supportato il multiplexing, che consente di gestire più connessioni su un singolo canale di comunicazione.

Per quanto riguarda la gestione delle connessioni, il sistema deve mantenere connessioni persistenti per ridurre il costo di stabilire connessioni multiple. Inoltre, deve utilizzare un pool di connessioni per riutilizzare connessioni esistenti, migliorando così l'efficienza complessiva del sistema. La gestione efficiente delle connessioni è cruciale per assicurare che il sistema possa gestire un numero crescente di entità comunicanti senza degradare le prestazioni.

Un altro aspetto fondamentale per la scalabilità è la gestione della coda di messaggi. Il sistema deve utilizzare code di messaggi distribuite per gestire efficacemente l'accodamento e la distribuzione dei messaggi tra le varie entità. Questo approccio garantisce che il sistema possa scalare in modo lineare con l'aumento del numero di messaggi e delle entità coinvolte, mantenendo alta la performance e l'affidabilità.

Questi requisiti sono necessari per assicurare che il modulo di comunicazione sia in grado di operare in modo efficiente e affidabile, indipendentemente dal numero di entità coinvolte e dalle condizioni di rete. Mantenendo alte prestazioni e stabilità del sistema, si garantisce che le comunicazioni avvengano in modo fluido e continuo, soddisfacendo le esigenze di un ambiente di rete variabile e potenzialmente instabile.

3.3 Affidabilità

Un aspetto cruciale nella comunicazione tra i componenti di un sistema distribuito è la robustezza e l'affidabilità, elementi fondamentali per garantire il corretto funzionamento complessivo del sistema. I componenti devono essere in grado di inviare e ricevere messaggi utilizzando il protocollo MQTT. Questo richiede l'integrazione di librerie specifiche per MQTT, assicurando che i messaggi possano essere formattati, inviati e ricevuti correttamente. È imperativo garantire l'affidabilità nella consegna dei messaggi, assicurando in alcuni casi che ogni messaggio venga ricevuto dal destinatario previsto. Per raggiungere questo obiettivo, è necessario implementare la gestione delle conferme di ricezione, meccanismi di ritrasmissione in caso di fallimento e gestione delle code di messaggi per evitare perdite.

Il progetto offre questa possibilità utilizzando una QoS di livello 1 o 2, permettendo allo sviluppatore di scegliere di utilizzare tali garanzie di ricezione e non

duplicazione.Il progetto offre anche la possibilità di usare QoS 0 e quindi non garantire la consegna, ma consente avere prestazioni più elevate, in particolare su dispositivi IoT dotati di scarsa poternza hardware.

La gestione delle code di messaggi rappresenta un'altra componente critica. Un sistema di coda ben progettato può aiutare a prevenire la perdita di messaggi in situazioni di traffico elevato o in caso di disconnessioni temporanee. Implementare meccanismi di bufferizzazione e di ritrasmissione può migliorare significativamente l'affidabilità complessiva del sistema. In particolare, le code devono essere in grado di gestire i picchi di traffico senza perdere dati e devono poter recuperare rapidamente in caso di disconnessione, assicurando che i messaggi vengano consegnati non appena la connessione viene ristabilita.

Il modulo di comunicazione deve includere funzionalità di logging e monitoraggio per tracciare l'attività e le performance del sistema. Questi strumenti permettono di identificare e risolvere rapidamente eventuali problemi, migliorando la reattività del sistema alle condizioni di errore. Il logging dettagliato degli eventi di comunicazione aiuta anche nella diagnosi e nella prevenzione di guasti futuri, fornendo dati preziosi per l'analisi delle cause principali dei problemi. Un sistema di monitoraggio efficiente può allertare gli operatori in caso di anomalie, permettendo interventi tempestivi e mirati per minimizzare l'impatto sugli utenti finali.

L'affidabilità può essere ulteriormente migliorata attraverso l'adozione di pratiche DevOps, con pipeline Continuous Integration and Continuous Deployment (CI/CD), che assicurano che le nuove versioni del software vengano distribuite in modo sicuro e senza interruzioni, mantenendo un elevato standard di qualità e affidabilità. La continua integrazione e distribuzione permette di rilevare e risolvere rapidamente i bug, riducendo al minimo il rischio di errori introdotti da nuove funzionalità o aggiornamenti. Queste pratiche, combinate con tecniche di testing rigorose come i test di stress e i test di carico, valutano le prestazioni del sistema sotto condizioni estreme, identificando i punti deboli del sistema e apportando le necessarie migliorie prima che i problemi si manifestino in produzione.

3.4 Interoperabilità

L'interoperabilità si riferisce alla capacità dei diversi sistemi e componenti software di lavorare insieme senza problemi, scambiando dati e utilizzando le informazioni scambiate in modo efficace. Nel contesto di PulvReAKt, questo significa che i componenti sviluppati per JVM devono poter interagire con quelli sviluppati in JavaScript o con quelli che girano su piattaforme native. Questo richiede che il protocollo di comunicazione, in questo caso MQTT, sia implementato in modo standard e conforme su tutte le piattaforme.

3.4.1 Java Virtual Machine

La JVM è una macchina virtuale che esegue bytecode Java, ed è la piattaforma di riferimento per lo sviluppo in Java e Kotlin. Essa offre una serie di caratteristiche che la rendono una scelta popolare per l'implementazione di applicazioni complesse e ad alte prestazioni. Le caratteristiche principali della JVM sono:

- **Interoperabilità**: La JVM permette l'interoperabilità tra Kotlin e Java, consentendo l'uso di librerie e strumenti Java consolidati, permettendo agli sviluppatori di sfruttare l'ampio ecosistema Java esistente.
- **Portabilità**: Il bytecode generato può essere eseguito su qualsiasi dispositivo dotato di una JVM, garantendo portabilità cross-platform. Questo è particolarmente utile per le applicazioni distribuite su una vasta gamma di dispositivi e sistemi operativi.
- **Ottimizzazioni di Runtime**: La JVM include un Just-In-Time (JIT) compiler che ottimizza il codice durante l'esecuzione, migliorando le performance e consentendo un'esecuzione più veloce delle applicazioni Kotlin rispetto a linguaggi interpretati o compilati staticamente.
- **Garbage Collection**: Gestione automatica della memoria tramite garbage collection, riducendo il rischio di memory leaks. Allevia gli sviluppatori dalla necessità di gestire manualmente l'allocazione e la deallocazione della memoria, semplificando lo sviluppo e riducendo il rischio di errori.

3.4.2 JavaScript e Node.js

JavaScript è il linguaggio di scripting dominante per lo sviluppo web client-side, eseguito all'interno dei browser. Kotlin/JS permette di compilare codice Kotlin in JavaScript, sfruttando le capacità di questa piattaforma. Le caratteristiche principali di JS sono:

- **Compatibilità con il Web**: JavaScript è nativamente supportato dai browser, permettendo l'esecuzione di applicazioni Kotlin/JS direttamente nel contesto web e consente agli sviluppatori di creare applicazioni web interattive e dinamiche utilizzando Kotlin come linguaggio di sviluppo.

- **Ecosistema di Librerie**: Ampia disponibilità di librerie e framework per lo sviluppo web, che possono essere utilizzati anche con Kotlin/JS. Include librerie per la gestione del DOM[1], l'interazione con API REST[2], la manipolazione dei dati JSON[3] e molto altro ancora, consentendo quindi agli sviluppatori di sfruttare le funzionalità esistenti senza doverle implementare nuovamente.

- **Asincronia**: Supporto nativo per le operazioni asincrone tramite callback, Promises e async/await che vanno a sopperire alla mancanza di capacità multithread essendo progettato per lavorare anche in contesto browser, quindi con un singolo thread. Questa particolarita è estremamente utile per le applicazioni web che devono gestire operazioni di rete, come il caricamento di dati da un server remoto.

- **Interoperabilità**: Kotlin/JS permette l'interoperabilità con le API JavaScript esistenti, facilitando l'integrazione con altre tecnologie web, consentendo agli sviluppatori di utilizzare librerie e framework JavaScript esistenti all'interno del codice Kotlin, con una maggiore flessibilità e libertà di scelta nella progettazione delle applicazioni.

[1]DOM (Document Object Model): Rappresentazione ad albero di un documento HTML/XML che permette di accedere e modificare dinamicamente la struttura e il contenuto delle pagine web.

[2]API REST (Representational State Transfer): Insieme di convenzioni per creare servizi web che permettono la comunicazione tra client e server utilizzando operazioni HTTP standard.

[3]JSON (JavaScript Object Notation): Formato di testo leggero per lo scambio di dati, facilmente leggibile dagli umani e parsabile dai computer.

- **Node.js** rappresenta una potente piattaforma per lo sviluppo backend, estendendo le capacità di JavaScript oltre il browser e offrendo un ambiente efficiente e scalabile per la creazione di applicazioni server-side moderne. Utilizzando Node.js, gli sviluppatori possono beneficiare di un ecosistema ricco di strumenti avanzati per costruire applicazioni robuste e performanti.

- **Event-driven e Non-blocking I/O**: Node.js utilizza un modello event-driven e non-blocking per le operazioni di I/O, consentendo di gestire un numero elevato di connessioni concorrenti con un singolo thread grazie al meccanismo delle promise in JS.

- **Ecosistema di Moduli** : Node.js ha un vasto ecosistema di moduli e pacchetti disponibili tramite npm (Node Package Manager). Questi moduli coprono una vasta gamma di funzionalità, come la gestione del database, la manipolazione dei file e la creazione di server web.

3.4.3 Native

Kotlin/Native permette di compilare codice Kotlin direttamente in codice macchina eseguibile su dispositivi target, senza la necessità di una macchina virtuale o un interprete. Le caratteristiche principali delle diverse piattaforme native sono:

- **Esecuzione Diretta**: Compilazione in codice nativo che viene eseguito direttamente dall'hardware, senza layer di interpretazione. Si traduce in prestazioni superiori rispetto alle applicazioni eseguite su una macchina virtuale o interprete, poiché non vi è alcun overhead dovuto alla traduzione del bytecode in istruzioni di macchina.

- **Performance**: Le applicazioni native tendono ad avere performance superiori rispetto a quelle eseguite su una macchina virtuale o interprete, grazie alla compilazione diretta. Questo è particolarmente importante per le applicazioni ad alte prestazioni che richiedono un tempo di risposta rapido e una bassa latenza.

- **Accesso a Funzionalità di Sistema**: Possibilità di accedere direttamente alle API di sistema e alle risorse hardware, offrendo maggior controllo

e ottimizzazione. Consente agli sviluppatori di creare applicazioni che interagiscono direttamente con l'hardware sottostante, sfruttando appieno le capacità del dispositivo target.

- **Assenza di Overhead**: Sebbene Kotlin/Native includa un garbage collector, quando viene eseguito su piattaforme native non vi è nessun overhead dovuto alla gestione della macchina virtuale. Questo significa che le risorse del sistema possono essere utilizzate in modo più efficiente, riducendo il consumo di memoria e CPU.

- **Cross-platform**: La capacità di supportare diverse piattaforme, inclusi iOS, Windows, Linux e macOS, facilita lo sviluppo di applicazioni multipiattaforma e consente agli sviluppatori di scrivere un'unica base di codice Kotlin che può essere compilato ed eseguito su una varietà di dispositivi e sistemi operativi, riducendo al minimo lo sforzo di sviluppo e la complessità del codice.

Capitolo 4

Design

4.1 Struttura di MQTT

La discrepanza tra il modello publish-subscribe caratteristico di MQTT e la necessità di comunicazione diretta tra le entità, come previsto dal progetto, richiede una definizione accurata della struttura dei topic di MQTT. Per affrontare questa sfida, si è deciso di strutturare i topic nella forma `nomeProgetto/mittente/destinatario`. In questa struttura, 'nomeProgetto' rappresenta un topic unico all'interno del progetto, che ospita tutti i topic necessari per la comunicazione tra le singole entità. Ad esempio, nel progetto PulvReAKt, il nome del progetto è 'PulvReAKt', ma può essere personalizzato tramite il costruttore del gestore delle comunicazioni.

I componenti 'mittente' e 'destinatario' rappresentano rispettivamente il nome dell'entità che invia il messaggio e quello dell'entità destinataria. Questi nomi sono definiti esternamente al modulo comunicativo, rendendo l'utente responsabile della loro unicità all'interno del sistema per evitare ambiguità. In alternativa, l'utente può utilizzare nomi comuni per le entità che devono ricevere gli stessi messaggi. Questa struttura permette una gestione chiara e organizzata dei messaggi, facilitando il monitoraggio e la manutenzione del sistema di comunicazione.

Si è scelto di utilizzare MQTT over WebSocket, dove il pacchetto MQTT viene inserito all'interno di un messaggio WebSocket. Questa scelta è motivata principalmente dalla maggiore efficienza nell'utilizzo della rete, poiché WebSocket fornisce un canale full-duplex su una singola connessione TCP. Questa configurazione per-

mette di gestire sia il canale di comunicazione MQTT sia la trasmissione dei dati tramite una sola connessione TCP, riducendo il sovraccarico di rete e migliorando le prestazioni complessive del sistema, garantendo una comunicazione più fluida ed efficiente. La riduzione del sovraccarico è particolarmente importante in ambienti con risorse di rete limitate, dove ogni connessione aggiuntiva può influire negativamente sulle prestazioni complessive.

Un altro motivo importante per la scelta di WebSocket è la compatibilità con le infrastrutture HTTP esistenti. WebSocket permette alle comunicazioni MQTT di utilizzare strutture HTTP già presenti, come la porta HTTP 80, firewall e proxy, che non sarebbero utilizzabili se MQTT fosse impiegato direttamente su TCP. Questa compatibilità offre vantaggi in termini di sicurezza e gestione della rete, facilitando l'integrazione del protocollo MQTT in ambienti con restrizioni di rete più rigide.

Per quanto riguarda le caratteristiche dei messaggi inviati e ricevuti, è stato deciso di utilizzare come impostazione predefinita il livello di QoS 2 (exactly once), poiché è il livello più affidabile, anche se meno efficiente. Tuttavia, si offre all'utente la possibilità di specificare il livello di QoS desiderato, il valore del ServerKeepAlive (tempo in secondi che il server mantiene la connessione viva dopo l'ultima interazione con il client, impostato di default a 10 secondi) e il flag retain del messaggio (che indica se i messaggi inviati vadano salvati in memoria nel broker fino all'arrivo del messaggio successivo, impostato di default a True).

Il payload dei messaggi è gestito con particolare attenzione: viene inserito nel pacchetto MQTT e in quello WebSocket esattamente come ricevuto dall'utente, viene poi inviato al broker e successivamente rimosso dal pacchetto MQTT per essere restituito all'entità destinataria. Questa gestione assicura che il contenuto del messaggio mantenga la sua integrità e che la comunicazione avvenga in modo trasparente ed efficace, rispettando le esigenze di affidabilità e performance del sistema. Questo approccio permette di criptare i messaggi prima dell'invio e decriptarli dopo la ricezione, senza che l'integrità del messaggio venga intaccata dal protocollo o dall'utilizzo del progetto.

4.2 Struttura delle classi

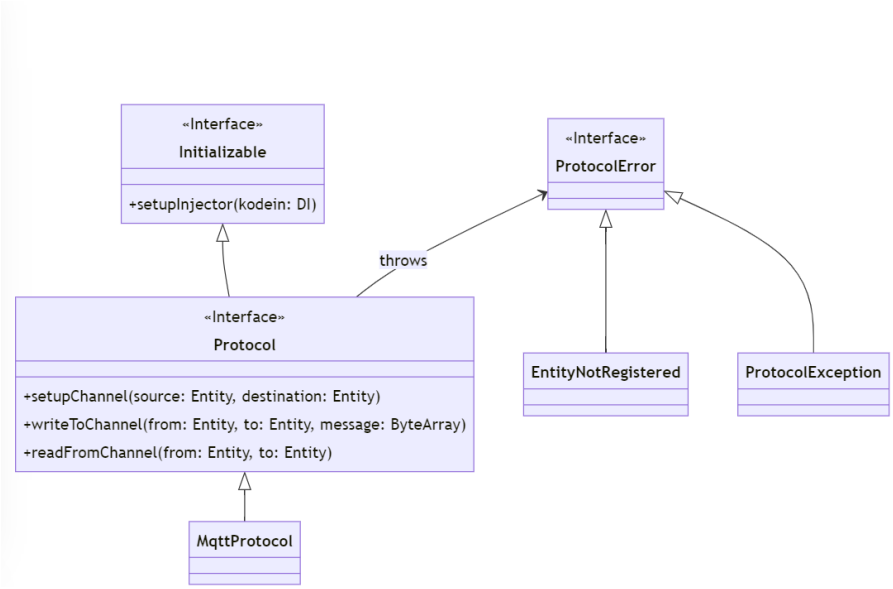

Figura 4.1: Diagramma delle classi

Il diagramma delle classi presentato in Figura 4.1 illustra l'architettura e le relazioni tra le interfacce e le classi implementate nel progetto. Di seguito, viene fornita una descrizione dettagliata di ciascun componente e delle loro interazioni:

- `InjectAwareResource`: è un'interfaccia che definisce il metodo utilizzato per configurare l'iniettore delle dipendenze all'interno di PulvReAKt .
- `ProtocolError`: è un'interfaccia che rappresenta gli errori relativi al protocollo. Viene implementata da due classi concrete:
  - `EntityNotRegistered`: rappresenta un errore che si verifica quando un'entità non è registrata.
  - `ProtocolException`: rappresenta un'eccezione generata direttamente dal protocollo MQTT, riportata all' interno di Kotlin in modo tale da poterla gestire correttamente e minimizzare gli errori durante l'esecuzione.

- `Protocol`: è un'interfaccia che definisce i metodi essenziali per l'utilizzo di un generico protocollo di comunicazione. Fornisce infatti un metodo per ognuna delle tre operazioni fondamentali:

  - `setupChannel`: metodo per configurare i canali di comunicazione tra due entità `source` e `destination`. Nello specifico verrà aperto un canale `/source/destination` e un canale `/destination/source`, in quanto ci si aspetta la comunicazione tra le due entità sia bidirezionale nella maggior parte delle casistiche di utilizzo, evitando in questo modo di rendere necessarie due chiamate al metodo, una per direzione di comunicazione nella coppia. Una volta chiamato il setup sarà valido per tutto il tempo di esecuzione, sarà quindi sufficiente chiamarlo una volta per ogni coppia `source` e `destination`.

  - `writeToChannel`: Metodo per scrivere un messaggio al destinatario specificato `to` rendendo noto il mittente `from`. Questo metodo può riuscire o fallire con un `ProtocolError`. Fallisce se il canale di comunicazione tra le entità `from` e `to` non è stato creato correttamente tramite il metodo `setupChannel` prima di tentare di mandare un messaggio.

  - `readFromChannel`: Metodo per leggere dati dal canale di comunicazione specificato. Questo metodo restituisce un `Flow` di messaggi che può essere consumato a runtime. Può riuscire o fallire restituendo un `ProtocolError`. Fallisce se il canale di comunicazione tra le entità `from` e `to` non è stato creato correttamente tramite il metodo `setupChannel` prima di tentare di leggere quanto scritto sul canale.

- `MqttProtocol`: Una classe concreta che implementa l'interfaccia `Protocol`. Questa classe fornisce l'implementazione specifica per il protocollo MQTT per le differenti piattaforme, includendo i metodi definiti in `Protocol`. L'implementazione in `MqttProtocol` sfrutta le funzionalità di MQTT per gestire le comunicazioni tra i nodi nel sistema distribuito.

4.3 Struttura dei Package

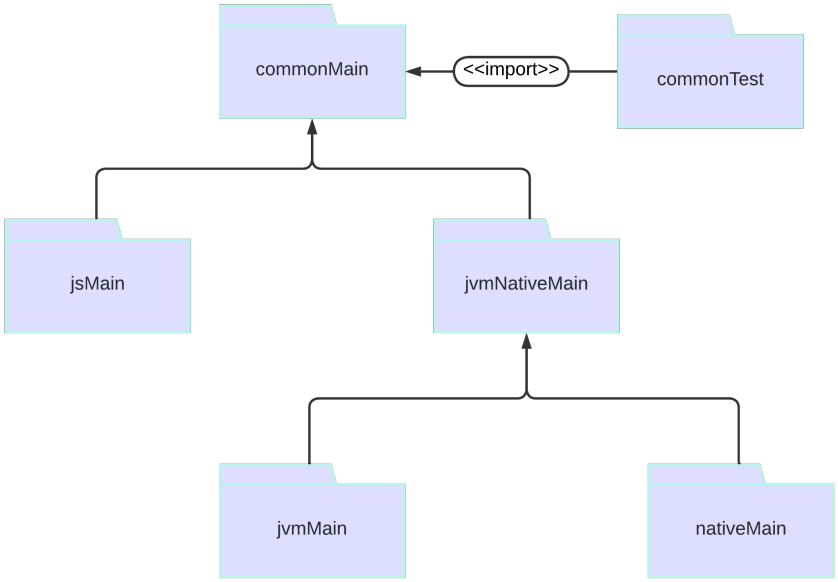

Figura 4.2: Diagramma dei package

Dovendo il progetto funzionare su diverse piattaforme, si è cercato di unificare quanti più target possibili, in modo tale da potere scrivere e revisionare il codice una sola volta. Si è riusciti ad utilizzare una libreria comune e di conseguenza un modulo unificato per lo sviluppo di JVM e native mentre, data la natura singlethread di JS è stato necessario separare il target dagli altri due. Si è quindi optato per una libreria specifica per lo sviluppo con target NodeJS, in modo che questa fosse ottimizzata per l'ambiente specifico.

Per questo motivo il progetto è stato strutturato in modo tale da avere moduli distinti per ciascun ambiente, con un modulo specifico per JS e un modulo comune per JVM e native. Entrambi questi moduli implementano una API fornita nel modulo comune (`commonMain`) grazie alla struttura degli `expected` e degli `actual` fornita da Kotlin specificatamente per queste casistiche, in cui le stesse funzionalità e gli stessi metodi devono essere comuni alle diverse piattaforme ma richiedono implementazioni o librerie sottostanti differenti.

Il modulo comune per la JVM e native è chiamato `jvmNativeMain` e contiene un'implementazione comune ai due target. Questo modulo include tutte le funzionalità che possono essere eseguite in un ambiente multithread, sfruttando le capacità di concorrenza della JVM e delle piattaforme native. In `jvmNativeMain` sono implementati i componenti core del sistema, garantendo che la logica di base sia condivisa e riutilizzabile tra le diverse piattaforme. Questo approccio riduce la duplicazione del codice e facilita la manutenzione del progetto.

Dalla struttura di `jvmNativeMain` dipendono due moduli specifici per ogni target:

- `jvmMain`: Questo modulo estende `jvmNativeMain` per includere ottimizzazioni e integrazioni necessarie per funzionare efficacemente in un ambiente JVM. Qui vengono implementate le funzionalità specifiche che sfruttano le caratteristiche avanzate della JVM, come la gestione avanzata dei thread e l'ottimizzazione delle prestazioni.

- `nativeMain`: Questo modulo estende `jvmNativeMain` per le piattaforme native, includendo le implementazioni necessarie per funzionare su tali ambienti. Viene adattato per sfruttare le peculiarità delle diverse piattaforme native, garantendo efficienza e compatibilità con il codice nativo.

Per quanto riguarda lo sviluppo con JS, data la natura singlethread dell'ambiente, si è sviluppato un modulo specifico chiamato `jsMain`. Questo modulo contiene le implementazioni necessarie per funzionare con NodeJS, adattandosi alle caratteristiche dell'ambiente singlethread. `jsMain` è stato progettato per garantire che tutte le operazioni asincrone e le chiamate di rete siano gestite in modo efficiente, sfruttando il modello event-driven di NodeJS.

In aggiunta ai moduli funzionali, è stato creato un modulo comune per i test chiamato `commonTest`. Questo modulo contiene tutti i test necessari per verificare l'API comune tra i diversi target. Utilizzando `commonTest`, è possibile garantire che tutte le funzionalità comuni siano testate in modo uniforme su tutte le piattaforme, assicurando la coerenza e l'affidabilità del progetto. I test inclusi in `commonTest` coprono una vasta gamma di scenari, dalle unit test alle integration test, assicurando che ogni parte del sistema funzioni in modo corretto sia individualmente che nel contesto dell'intero progetto.

4.4 Struttura dei Test

Il modulo contenente i test valuta in maniera approfondita la robustezza del progetto attraverso una serie di scenari distinti. Tra questi scenari, viene verificato che il gestore delle comunicazioni possa essere inizializzato e finalizzato correttamente, senza incorrere in errori. Questa verifica è cruciale poiché assicura che tutte le operazioni di setup e di chiusura del protocollo siano eseguite correttamente, garantendo uno stato coerente del sistema sia prima che dopo l'esecuzione.

Un altro aspetto fondamentale della valutazione consiste nell'esaminare come il progetto gestisca correttamente gli errori in situazioni critiche. In particolare, vengono analizzati i comportamenti del protocollo quando si tenta di scrivere su un canale non registrato e quando si tenta di leggere da un canale non registrato. È essenziale che il protocollo sia in grado di identificare e segnalare tali errori con precisione, fornendo messaggi di errore comprensibili che facilitino la risoluzione dei problemi.

Inoltre, i test includono scenari che simulano la comunicazione effettiva tra due entità registrate correttamente. Questi scenari permettono di verificare il flusso completo della comunicazione, dall'inizializzazione alla trasmissione e ricezione dei messaggi, fino alla finalizzazione del protocollo. Questa verifica è particolarmente importante poiché dimostra l'efficacia del protocollo nella gestione della comunicazione tra le entità coinvolte e permette di controllare che il progetto funzioni correttamente senza generare errori inattesi.

Questi scenari, inclusi nei test, sono progettati per garantire che il protocollo MQTT implementato risponda in modo affidabile e robusto in una varietà di situazioni, sia in condizioni ideali di funzionamento che in presenza di errori o situazioni impreviste. L'obiettivo è assicurare che il protocollo mantenga prestazioni elevate e affidabili, indipendentemente dalle circostanze operative. Il risultato atteso è che il sistema risponda adeguatamente alle esigenze di comunicazione e gestione degli errori, offrendo una soluzione solida e stabile per l'uso previsto.

Per garantire la completezza e l'affidabilità dei test, tutti i test sono stati eseguiti su dispositivi diversi e in condizioni di rete variabili. Questa diversificazione è stata necessaria per valutare come il sistema si comporta in ambienti eterogenei, riflettendo le diverse configurazioni hardware e le diverse condizioni di rete che

potrebbero essere incontrate nel mondo reale. I test hanno incluso scenari con alta latenza, perdita di pacchetti e variabilità della larghezza di banda, permettendo di verificare che il sistema mantenga prestazioni adeguate anche in situazioni di rete non ottimali. Questo ha assicurato che il protocollo MQTT possa essere utilizzato con successo in moltelplici di situazioni operative funzionando come atteso.

4.5 Tecnologie Utilizzate

4.5.1 Kotest

Kotest offre un framework di testing multipiattaforma specificamente progettato per Kotlin. Kotest oltre a semplificare il processo di scrittura e di esecuzione dei test, si distingue per la sua flessibilità e le funzionalità avanzate. Grazie al supporto per test di proprietà, test parametrizzati e altre caratteristiche, Kotest garantisce prestazioni eccellenti e affidabilità nei risultati, fornendo un solido fondamento per la verifica e la convalida del codice Kotlin.

Kotest permette anche la creazione di test personalizzati, che possono essere configurati in modo dettagliato per adattarsi alle specifiche esigenze del progetto. La possibilità di eseguire test in parallelo contribuisce a ridurre significativamente i tempi di esecuzione, migliorando l'efficienza complessiva del ciclo di sviluppo. Inoltre, la compatibilità di Kotest con diversi ambienti di integrazione continua (CI/CD) ne facilita l'adozione in progetti di qualsiasi dimensione e complessità.

4.5.2 Arrow

Arrow è una libreria potente che porta la programmazione funzionale idiomatica in Kotlin, arricchendo il linguaggio con concetti come i tipi di dati immutabili e le monadi. L'integrazione di Arrow nel progetto migliora significativamente la qualità del codice, facilitandone la manutenzione e rendendolo più leggibile e testabile. Questa libreria contribuisce in modo tangibile alla costruzione di applicazioni Kotlin più robuste e manutenibili.

Arrow offre una vasta gamma di funzionalità, tra cui i data types per rappresentare strutture di dati comuni in modo sicuro e idiomatico, e le typeclasses

che consentono di estendere le funzionalità del linguaggio in maniera modulare. La presenza di effetti gestiti (Managed Effects) permette di scrivere codice che gestisce side effects in modo controllato e prevedibile, riducendo i rischi associati a operazioni come l'accesso al network o al filesystem. L'utilizzo di Arrow facilita inoltre l'adozione di pattern di programmazione funzionale come le funzioni pure e le composizioni, migliorando la coesione e la riusabilità del codice.

4.5.3 Kmqtt

Kmqtt riveste un ruolo cruciale nell'implementazione della comunicazione tramite MQTT nel progetto. Questa libreria, progettata per Kotlin Multiplatform, offre un'implementazione unificata di MQTT su tutte le piattaforme di interesse al progetto, In particolare per JVM e native. La sua facilità d'uso e la flessibilità consentono di integrare la comunicazione MQTT in modo uniforme e affidabile su tutte le piattaforme supportate, fornendo una solida base per lo sviluppo delle funzionalità di comunicazione nell'applicazione.

4.5.4 MQTT.js

MQTT.js è una libreria per lo sviluppo di applicazioni web che richiedono comunicazione MQTT in ambienti JavaScript. MQTT.js è progettata per essere leggera, altamente performante e semplice da usare, consentendo di gestire un elevato numero di messaggi con una latenza minima. La libreria supporta tutti i principali meccanismi di sicurezza, inclusa la crittografia SSL/TLS e l'autenticazione basata su username e password, garantendo una comunicazione sicura anche in contesti sensibili.

4.5.5 Mosquitto

Mosquitto rappresenta il cuore dell'infrastruttura MQTT del progetto, offrendo un broker MQTT open source che supporta sia l'hosting locale che il servizio pubblico. La leggerezza e la configurabilità di Mosquitto lo rendono una scelta ideale per creare un ambiente di sviluppo e testing robusto per le applicazioni basate su

MQTT. Le sue funzionalità avanzate, come l'autenticazione e la sicurezza delle connessioni, garantiscono un'esperienza di sviluppo sicura e affidabile.

4.5.6 Git

Git svolge un ruolo fondamentale nel controllo di versione del codice sorgente del progetto. Questo sistema di controllo di versione distribuito consente agli sviluppatori di tracciare le modifiche al codice, collaborare in modo efficiente con altri membri del team e gestire diverse versioni del progetto. La sua flessibilità e la vasta gamma di funzionalità supportate lo rendono una scelta versatile per il controllo di versione in progetti di qualsiasi dimensione e complessità.

Git offre strumenti potenti per la gestione delle branch, permettendo agli sviluppatori di lavorare su nuove funzionalità o bug fix in isolamento, prima di integrare le modifiche nel branch principale. Il sistema di merging di Git è sofisticato e permette di risolvere conflitti in maniera efficace. La presenza di piattaforme come GitHub ha ulteriormente esteso le capacità di Git, fornendo strumenti per la revisione del codice, l'integrazione continua e la gestione dei progetti, facilitando una collaborazione più strutturata e produttiva.

4.5.7 Gradle

Gradle rappresenta il cuore del processo di build e automazione del progetto. Questo sistema di automazione della compilazione, con la sua configurabilità ed estendibilità, semplifica la gestione delle dipendenze, la compilazione del codice e l'esecuzione dei test. Grazie alla sua integrazione con Kotlin, Gradle contribuisce in modo significativo a rendere il processo di sviluppo più efficiente e organizzato, offrendo un ambiente di sviluppo ottimizzato e scalabile.

4.5.8 Detekt

Detekt è uno strumento di analisi statica del codice progettato specificamente per Kotlin, che aiuta a identificare problemi di qualità del codice e potenziali bug. Integrando Detekt nel progetto, è possibile eseguire una scansione automatica del codice sorgente per rilevare problemi di stile, errori comuni e complessità del

codice. Detekt offre una vasta gamma di regole predefinite che coprono vari aspetti della qualità del codice, come la complessità ciclomatica, le convenzioni di codice, i potenziali bug e le vulnerabilità di sicurezza. Inoltre, è possibile configurare Detekt per adattarsi alle esigenze specifiche del progetto, personalizzando le regole esistenti o creando nuove regole personalizzate. L'uso di Detekt contribuisce a mantenere il codice pulito e manutenibile, facilitando la conformità agli standard di codifica e riducendo il rischio di errori. Detekt può essere integrato nei processi di integrazione continua (CI/CD) per garantire che il codice nuovo o modificato venga analizzato automaticamente, fornendo feedback immediato agli sviluppatori e mantenendo alta la qualità del codice durante tutto il ciclo di sviluppo.

Capitolo 5

Implementazione e Validazione

5.1 Sviluppo

Conclusa la prima fase di documentazione e design descritta nei capitoli precedenti, è stata affrontata la fase di sviluppo e alla scrittura del codice effettivo. All'inizio di questa fase è stato deciso di focallizare l'attenzione sul modulo `jvmNativeMain` per i target Native e JVM, lasciando momentaneamente in secondo piano JavaScript. Questa decisione è stata presa in modo da poter affrontare le complessità delle piattaforme Native e JVM prima di occuparsi delle peculiarità di JavaScript.

Per lo sviluppo di questo modulo, ci si è avvalsi della libreria KMQTT, la quale fornisce un'implementazione completa di un client MQTT in Kotlin. KMQTT è stata scelta per la sua robustezza e la sua aderenza agli standard del protocollo MQTT, che garantiscono un'ottima interoperabilità con vari broker MQTT e una gestione efficiente delle comunicazioni. In particolare, questa libreria offre metodi per le principali operazioni del protocollo MQTT, facilitando così l'implementazione delle funzionalità necessarie nel progetto:

- `connect`: Questo metodo consente di connettere il client al broker, permettendo di specificare tutte le opzioni della connessione, come *cleanStart* e le opzioni relative all'utilizzo di MQTT over WebSocket. Richiede inoltre come parametro la funzione da eseguire ogni volta che il client riceve un messaggio su uno qualsiasi dei topic a cui è iscritto. La connessione può poi essere chiusa tramite il metodo `disconnect`.

- `publish`: Questo metodo permette di pubblicare un messaggio sul topic specificato con i parametri desiderati, tra cui il livello di QoS del messaggio inviato e la flag *retain* del messaggio all'interno del broker.

- `subscribe`: Questo metodo consente al client di iscriversi a uno o più topic, specificando per ciascuno le opzioni di iscrizione relative al singolo topic, come il livello di QoS per la ricezione dei messaggi.

Poiché si è scelto di utilizzare la libreria KMQTT, che per funzionare su piattaforme native dipende a sua volta dall'utilizzo di OpenSSL, è stato necessario incorporare le implementazioni specifiche di OpenSSL per ogni piattaforma all'interno del progetto in una struttura apposita (`src/nativeLibs`). Questa operazione ha comportato l'aggiunta delle dipendenze ai file specifici per i singoli target nativi all'interno dei sourceset nel file di configurazione Gradle. In questo modo, si è potuta garantire la compatibilità del progetto con le diverse piattaforme target, rendendo possibile l'utilizzo delle funzionalità di KMQTT sia su piattaforme JVM che Native, superando le limitazioni imposte dalle dipendenze di basso livello come OpenSSL. Questo approccio strutturato ha facilitato l'integrazione delle librerie necessarie, assicurando un ambiente di sviluppo stabile e coerente.

Il modulo sviluppato per i target JVM e Native sfrutta le coroutines di Kotlin per permettere un'esecuzione parallela del client MQTT e del gestore delle comunicazioni tra le varie entità. Le coroutines di Kotlin offrono un modo efficace per gestire operazioni asincrone senza bloccare il thread principale, un elemento cruciale per le applicazioni che devono mantenere alta reattività. Questo modulo riceve, tramite il builder, il parametro `CoroutineDispatcher`, necessario per la corretta creazione ed esecuzione del `CoroutineScope`. Tale scope ha la funzione di ospitare ed eseguire il client per l'invio e la ricezione effettivi dei pacchetti MQTT. L'utilizzo delle coroutines consente di gestire in modo efficiente le operazioni asincrone, garantendo che il client MQTT possa operare in parallelo con altre componenti del sistema senza bloccare il thread principale. Questo approccio migliora le prestazioni complessive dell'applicazione e facilita la gestione delle comunicazioni MQTT, rendendole più reattive e scalabili. L'architettura basata su coroutines permette di ottenere un'applicazione più robusta e capace di gesti-

re un elevato numero di messaggi contemporaneamente, adattandosi alle esigenze dinamiche del sistema.

L'ultimo target della fase di sviluppo è stato JavaScript, nello specifico NodeJS, poiché non è stato possibile trovare una libreria JS che implementasse il protocollo MQTT per il browser e che fosse compatibile con lo sviluppo in Kotlin. La scelta di concentrarsi su questa piattaforma ha comportato le sfide più significative dell'intero progetto. La natura single-threaded di JavaScript, che lo contraddistingue dai precedenti target capaci di operare su più thread, permettendo un vero parallelismo e l'uso di coroutine su thread separati, ha rappresentato una sfida considerevole. A differenza delle piattaforme multi-threaded, JavaScript gestisce tutte le operazioni su un singolo thread, il che complica la gestione di operazioni concorrenti e asincrone. Questa peculiarità di JavaScript ha richiesto un adattamento delle strategie di programmazione rispetto a quelle adottate per gli altri target e ha generato diversi ostacoli, alcuni dei quali sono stati risolti in modo relativamente facile, mentre altri hanno richiesto strategie più intricate.

Ad esempio, la gestione degli eventi asincroni è stata una delle principali difficoltà incontrate. È stato necessario garantire che le operazioni di rete non bloccassero il thread principale e che l'applicazione rimanesse sempre reattiva agli input dell'utente. La gestione asincrona in JavaScript, pur essendo molto potente, può diventare complessa da gestire quando si tratta di garantire che tutte le operazioni avvengano in modo coordinato e senza interferenze. Per affrontare questa sfida, sono state adottate tecniche avanzate di gestione degli eventi per mantenere il codice leggibile e manutenibile, pur garantendo prestazioni ottimali. L'integrazione di queste tecniche ha permesso di gestire in modo efficiente le operazioni asincrone, assicurando che il client MQTT per NodeJS fosse reattivo e affidabile.

Nello sviluppo di questo modulo, è stata sfruttata la libreria MQTT.JS, che offre un'implementazione di un client MQTT in JavaScript. Questa libreria è stata scelta per la sua facilità d'uso e la sua ampia adozione nella comunità, che garantisce una buona documentazione e un supporto continuo. L'integrazione di MQTT.JS in Kotlin ha richiesto l'utilizzo delle funzionalità di interoperabilità JS disponibili in Kotlin, che permettono di interagire facilmente con le librerie JavaScript. Questa interoperabilità è stata fondamentale per permettere a Kotlin di chiamare funzioni JavaScript e viceversa senza problemi, mantenendo così una

coerenza tra le diverse parti del progetto. Inoltre, grazie a Kotlin/JS, è stato possibile sfruttare appieno le potenzialità della libreria MQTT.JS, mantenendo allo stesso tempo la coerenza con il resto del progetto.

L'interoperabilità tra Kotlin e JavaScript è stata cruciale per il successo di questa integrazione. Grazie a Kotlin/JS, è stato possibile definire classi e metodi esterni che mappano direttamente sulle controparti JavaScript, permettendo di chiamare funzioni JavaScript come se fossero parte del codice Kotlin. Questo ha reso il processo di sviluppo più fluido e ha ridotto significativamente la complessità legata alla gestione delle differenze tra i due linguaggi. In particolare, è stata definita una classe con il modificatore `external` per rispecchiare in Kotlin la classe JS che espone i seguenti metodi del client MQTT:

- `connect`: Questo metodo consente di connettere il client al broker, permettendo di specificare tutte le opzioni della connessione, tra cui *cleanStart* e le opzioni relative all'utilizzo di MQTT over WebSocket. La connessione può poi essere chiusa tramite il metodo `end`.

- `subscribe`: Questo metodo permette al client di iscriversi a un topic del broker, specificando per ciascuno le opzioni di iscrizione relative al singolo topic, come il livello di QoS per la ricezione dei messaggi.

- `publish`: Questo metodo consente di pubblicare un messaggio sul topic specificato con i parametri desiderati, tra cui la QoS del messaggio inviato e la flag *retain* del messaggio nel broker.

- `on`: Questo metodo permette di specificare la funzione da eseguire come conseguenza dell'evento specificato, come *connect* ed *error*. Di particolare importanza è la chiamata con *message* come parametro, in quanto specifica cosa fare in risposta alla ricezione di un messaggio.

L'approccio utilizzato ha facilitato l'integrazione del client MQTT con le altre componenti del sistema, assicurando una gestione efficiente delle comunicazioni e un'esperienza utente fluida.

5.2 Validazione

Il processo di validazione ha coinvolto diverse fasi di test su più piattaforme, ciascuna con le proprie peculiarità e sfide. La validazione è iniziata con la piattaforma JVM, considerata il target iniziale, e si è estesa successivamente alle piattaforme native, includendo Linux, Windows e macOS, per poi concentrarsi su JavaScript, in particolare NodeJS. Ogni piattaforma ha richiesto strategie specifiche per garantire la compatibilità e le prestazioni ottimali del sistema, considerando le diverse caratteristiche e limitazioni di ciascuna.

Dopo aver superato una fase iniziale di test abbastanza agevole avente come target la JVM, durante la quale lo sviluppo è proceduto senza intoppi, il focus è stato spostato sulle piattaforme native, con particolare riferimento a Linux e Windows. In questa fase, mentre Linux non ha causato particolari problemi, è stato riscontrato un problema significativo legato ai socket durante l'esecuzione nativa su sistema operativo Windows. Questo problema ha richiesto un'analisi approfondita per essere risolto e ha messo in luce le differenze nelle implementazioni specifiche dei socket di rete tra i vari sistemi operativi.

In particolare, l'implementazione interna della libreria KMQTT causava un'eccezione `Socket.IOException` quando tentava di stabilire una connessione al socket per una qualsiasi operazione, segnalando l'errore `Error 10047: Address family not supported by protocol family`. Questa situazione ha richiesto un'attenta analisi per riuscire ad individuare la causa del problema. Dopo una revisione approfondita del codice, è stata identificata l'origine del problema nella libreria KMQTT, che non gestiva correttamente alcuni aspetti relativi ai socket di comunicazione su Windows e di conseguenza generava l'errore di tipologia Winsock Errors. Questo difetto ha reso evidente la necessità di focallizare una specifica attenzione nei test per ogni singola piattaforma.

Per risolvere questa problematica, è stato necessario apportare alcune modifiche al codice che utilizzava la libreria e segnalare il problema allo sviluppatore di KMQTT, fornendo dettagli specifici sull'errore riscontrato. Fortunatamente, l'attenzione immediata del team di sviluppo ha portato a una soluzione rapida ed efficace. Il problema è stato risolto e incorporato nella successiva release della libreria, assicurando una migliore compatibilità con le piattaforme Windows.

Rappresentando un genrale miglioramento alla libreria KMQTT in termini di funzionalità ed affidabilità e garantendone il corretto funzionamento all'interno del progetto. Questo intervento ha permesso di superare un ostacolo significativo e ha migliorato l'affidabilità e la robustezza del sistema.

Dopo aver risolto le problematiche con i target nativi, l'attenzione si è spostata su JavaScript, la cui natura single-threaded ha generato diversi ostacoli anche durante la fase di testing. Per garantire che la logica dei test non portasse a bloccare il thread principale a prescindere dal funzionamento del codice soggetto a validazione e che l'applicazione rimanesse reattiva, è stato necessario revisionare il codice dei test. Durante la progettazione e l'esecuzione dei test è stata prestata particolare attenzione alla gestione degli eventi asincroni, ed è stato fondamentale garantire che il sistema fosse in grado di gestire correttamente le connessioni al broker MQTT. La fase di testing ha permesso di identificare e risolvere diversi problemi legati alla natura asincrona di JavaScript. Ad esempio, è stato necessario assicurarsi che tutte le operazioni di rete fossero gestite correttamente senza bloccare il thread principale, garantendo al contempo che l'applicazione rimanesse reattiva e rispondesse prontamente agli input dell'utente. Questo approccio ha consentito di testare in modo efficace il modulo MQTT in ambiente NodeJS, assicurando che il sistema funzionasse correttamente e fosse in grado di gestire le comunicazioni MQTT in modo affidabile ed efficiente.

In conclusione, le diverse fasi di sviluppo e test hanno messo in luce le peculiarità e le sfide specifiche di ciascuna piattaforma. L'uso combinato delle librerie KMQTT e MQTT.JS, insieme all'adozione di tecniche avanzate per la gestione delle operazioni asincrone, ha permesso di sviluppare un sistema robusto e versatile, capace di operare efficacemente su piattaforme JVM, native e JavaScript. Le soluzioni implementate hanno garantito la compatibilità e la reattività del sistema, migliorando l'esperienza utente e assicurando un'ottima gestione delle comunicazioni MQTT. Questo progetto ha dimostrato l'importanza della flessibilità e dell'adattamento nell'affrontare le sfide poste dalle diverse piattaforme di destinazione, assicurando il successo del sistema finale.

5.3 Prestazione sulle diverse piattaforme

Durante la fase di sviluppo, ma soprattutto durante quella di validazione, è stata notata una notevole differenza di tempo di esecuzione tra le diverse piattaforme. Per comprendere meglio queste differenze, sono stati conseguentemente svolti dei semplici test con lo scopo di evidenziare tali differenze e confrontare le prestazioni del progetto sulle differenti piattaforme aventi la stessa architettura sottostante. Il parametro scelto per svolgere tali valutazioni è stato la media dei tempi di completamento dei test del progetto, un indicatore utile e significativo per comprendere le prestazioni relative. Tali test sono stati eseguiti sistematicamente tramite il comando `gradlew allTest --parallel --profile` in differenti condizioni operative, garantendo un confronto equo e accurato tra le diverse configurazioni.

Il computer utilizzato per i test ha le seguenti caratteristiche hardware: Processore 11th Gen Intel(R) Core(TM) i5-1135G7 con una frequenza base di 2.40GHz e una frequenza massima di 2.42GHz, RAM SODIMM a 3200 MHz da 8.00 GB. Il sistema operativo è un ambiente a 64-bit, con un processore basato su architettura x64. Questa configurazione hardware è stata scelta per garantire una base solida e coerente per l'esecuzione dei test, riducendo al minimo le variabili dovute a differenze di hardware.

I risultati dei test nativi su Linux e Windows sono stati ottenuti utilizzando lo stesso computer, configurato in due partizioni separate, e quindi con lo stesso hardware sottostante. Questa configurazione ha permesso di isolare le variabili legate all'hardware e focalizzarsi sulle differenze di performance dovute esclusivamente ai sistemi operativi. I test per le piattaforme JVM e JS sono stati eseguiti sia in ambiente Linux che in ambiente Windows, sempre utilizzando lo stesso dispositivo già impiegato per i test nativi. Ciò ha assicurato che qualsiasi differenza osservata fosse attribuibile esclusivamente alle piattaforme software e alle differenze tra le due librerie utilizzate.

I test sono stati divisi in due tipologie principali, differenziate in base al server utilizzato per ospitare il broker MQTT. Nel primo caso, il broker è stato ospitato localmente tramite Mosquitto, configurato per richiedere autenticazione. Questa configurazione ha offerto un contesto controllato e stabile per le misurazioni, consentendo di osservare le prestazioni del sistema in un ambiente ideale. Nel secondo

caso, è stato utilizzato il broker remoto `test.mosquitto.org`, che non richiede autenticazione. Questo setup ha simulato un ambiente più realistico e variabile, permettendo di valutare le prestazioni del sistema in condizioni di utilizzo reale.

Questa doppia configurazione di test ha permesso di ottenere una visione completa e approfondita delle capacità e dei limiti del progetto nelle diverse condizioni operative. L'utilizzo di un broker locale ha fornito un punto di riferimento per le migliori prestazioni possibili, mentre il broker remoto ha evidenziato come il sistema si comporta in situazioni di rete meno prevedibili. Questo approccio ha garantito che le valutazioni fossero complete e rappresentative delle varie situazioni in cui il sistema potrebbe essere utilizzato, fornendo dati preziosi per ulteriori ottimizzazioni e miglioramenti. L'analisi dettagliata dei risultati ha permesso di identificare chiaramente i punti di forza e le aree in cui sono possibili miglioramenti, assicurando che il progetto possa raggiungere livelli ottimali di efficienza.

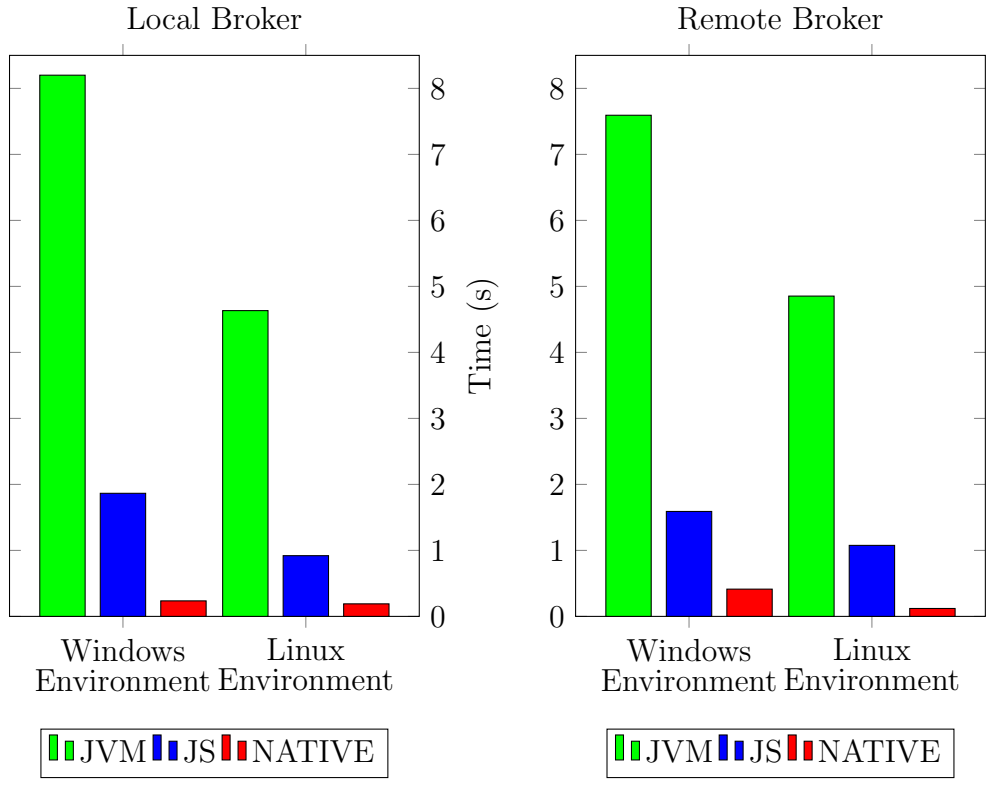

Figura 5.1: tempi di esecuzione medi nelle diverse condizioni

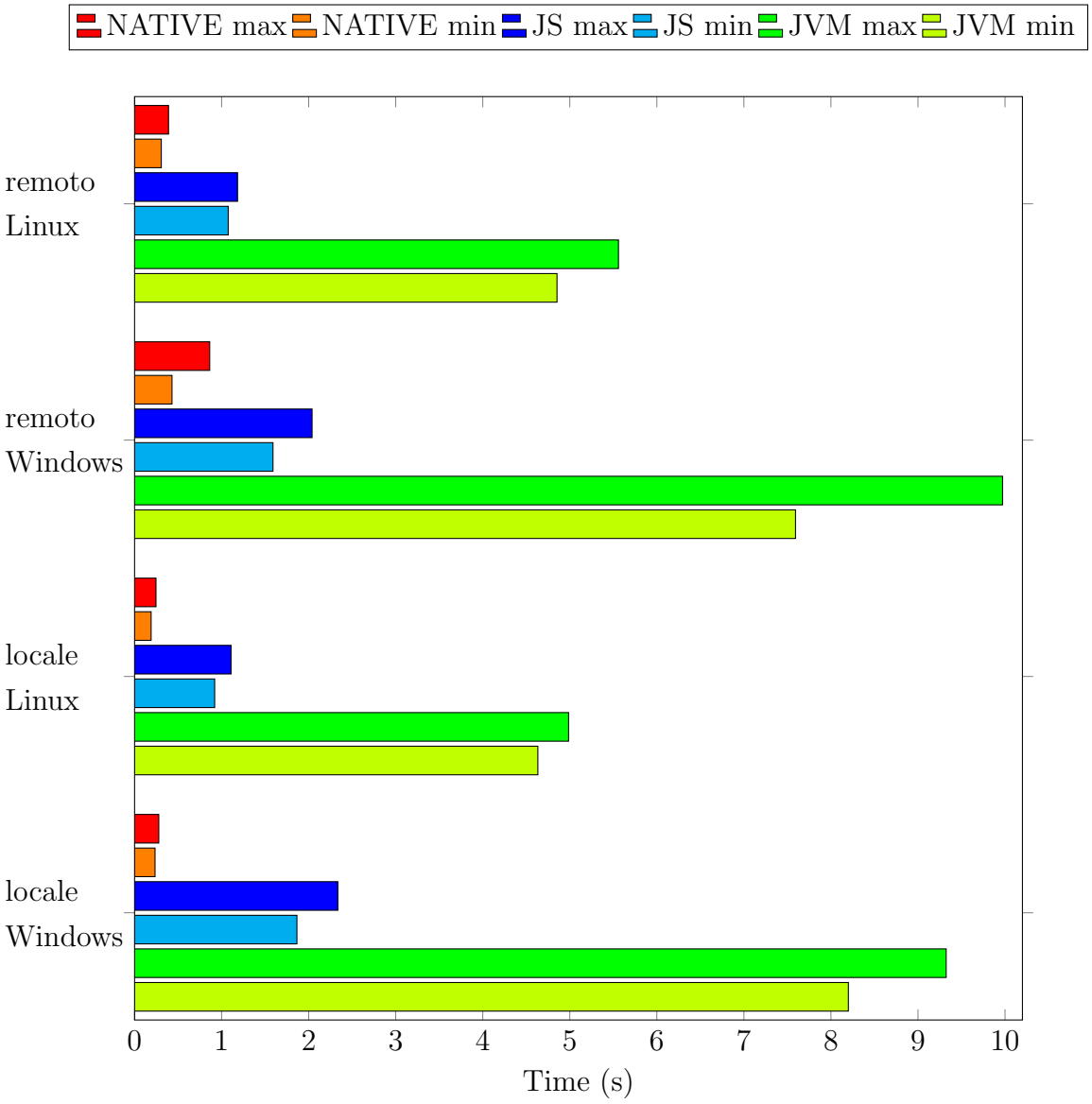

Figura 5.2: range dei tempi di esecuzione nelle diverse condizioni

I risultati delle diverse esecuzioni dei test nelle diverse condizioni, sono rappresentati nei grafici soprastanti che ne evidenziano i differenti tempi d'esecuzione medi, minimi e massimi. Da questi dati si possono trarre diverse conclusioni, interessanti da tenere in considerazione qualora si voglia utilizzare il modulo di comunicazione all'interno di altri progetti:

- Il sistema operativo influenza le prestazioni qualora il codice non venga eseguito nativamente ma appoggiandosi alla JVM o a JS. Nel caso ci sia appoggi alla JVM vi è una differenza di 3.153 secondi, nel caso di JS la discrepanza è di 730 millisecondi mentre in caso di esecuzione nativa di solamente 75ms.

- Il luogo in cui viene ospitato il broker dei messaggi (locale piuttosto che remoto) ha anch'esso una scarsa influenza sul risultato finale del test. Le differenze nei tempi di esecuzione sono sempre al di sotto dei 300 millisecondi. Si è verificata un'unica e specifica eccezione, durante l'esecuzione su JVM ospitata in un sistema Window, il cui tempo di esecuzione in caso di broker ospitato localmente è stata di 8.200s rispetto a 7593 utilizzando un broker remoto. In questo caso la differenza è stata di 607ms, che risulta più importante rispetto alle altre ma comunque trascurabile. Questo suggerisce che, a meno di condizioni estremamente specifiche, l'ubicazione del broker MQTT non rappresenta un fattore critico nelle prestazioni complessive.

- L'elemento che influenza maggiormente le prestazioni è rappresentato dalla piattaforma su cui viene eseguito il codice. Questo che emerge chiaramente quando si confrontano le tre piattaforme nello stesso ambiente (Windows o Linux) e con la medesima ubicazione del broker (locale o remota). Il tempo di esecuzione medio su target Nativo è 290 millisecondi, su JS è 1 secondo e 362 millisecondi mentre su JVM è 6.320 secondi. Questi valori indicano in maniera estremamente chiara che quando il codice veiene eseguito tramite la JVM le prestazioni sono notevolmente peggiori rispetto agli altri due target, essendo più lento di 6s rispetto a native e di 5s rispetto a JS. Vale comunque la pena evidenziare che anche la differenza di circa 1 secondo osservata tra le due piattaforma più veloci rappresenta uno scostamento non indifferente se si considerano i tempi in questione

- Qualora il codice venga eseguito sulla JVM i tempi di esecuzione sono notevolmente meno stabili con una differenza di 2 secondi tra il tempo di esecuzione minore e quello maggiore, mentre in caso di esecuzione su JS e native la differenza rispettivamente è di 425 e 211 millisecondi.

5.4 Integrazione in PulvReAKt

Una volta concluso lo sviluppo e la validazione del progetto singolo, Gli sforzi sono stati concentrati nell'integrazione del sistema sviluppato all'interno del più ampio e preesistente progetto `PulvReAKt`. Questo passaggio ha comportato una serie di adattamenti e modifiche al codice mirate a garantire la piena compatibilità tra i due progetti e la completa funzionalità di MQTT all'interno dell'ecosistema già stabilito di PulvReAKt. Questo ha ovviamente compreso anche una fase di validazione prima del completamento dell'integrazione del modulo di comunicazione sviluppato all'interno di PulvReAKt.

Prima di tutto è stato necessario modificare lievemente le interfacce del progetto originale per conformarle a quelle presenti all'interno di PulvReAKt. Questo processo ha richiesto di porre particolare attenzione alla Dependency Injection, che sebbene non fosse necessaria nel progetto autonomo, è fondamentale all'interno di PulvReAKt ed è stato quindi necessario avilupparla anche all'interno del modulo MQTT per una corretta integrazione. In PulvReAKt, la Dependency Injection è gestita tramite il framework Kodein[1]. Di conseguenza, è stato necessario implementare la funzione `setupInjector(kodein: DI)` per assicurare che le classi del modulo sviluppato fossero correttamente fruibilili all'interno dell' ecosistema di PulvReAKt.

Oltre alla Dependency Injection, sono state necessarie ulteriori modifiche minori per conformare completamente il modulo agli standard di qualità del codice e stile di PulvReAKt. Questo ha incluso una ridefinizione dettagliata della documentazione delle interfacce e dei test, al fine di renderle più chiare e congruenti con le linee guida stilistiche di PulvReAKt. Tali modifiche hanno comportato una revisione accurata della struttura e del contenuto della documentazione, migliorando notevolmente la leggibilità del codice. Inoltre, queste revisioni hanno facilitato la manutenzione e l'estensibilità del progetto nel lungo termine, rendendo il sistema più robusto e flessibile per future implementazioni.

Un'altra area che ha richiesto particolare attenzione durante l'integrazione è stata la struttura di Gradle. In particolare, è stato necessario aggiornare il fi-

[1]Kodein è un framework di Dependency Injection per Kotlin che facilita l'inversione del controllo. Consente di iniettare le dipendenze in modo flessibile ed efficiente, semplificando la gestione delle dipendenze e migliorando la modularità e la testabilità del codice.

le `build.gradle` per trasformarlo da file di configurazione adatto a un progetto autonomo a uno adatto ad un modulo all'interno di un progetto più ampio come PulvReAKt. Questo aggiornamento ha richiesto un'analisi approfondita della gestione di Gradle in PulvReAKt e un attento confronto con le necessità del progetto sviluppato, in modo tale da evitate duplicazioni di codice e garantendo al contempo tutte le funzionalità necessarie per il corretto funzionamento.

Inoltre, il sistema di CI/CD di PulvReAKt ha evidenziato diversi piccoli problemi all'interno del modulo sviluppato, in particolare l'assenza del target `linuxArm64` tra i target nativi del progetto Gradle. Questa mancanza è stata prontamente colmata aggiungendo correttamente il supporto al target specifico all'interno del file `build.gradle`, senza riscontrare problemi di compatibilità.

Capitolo 6

Conclusioni

Bibliografia

[Ak] Arrow-kt. Arrow documentation.

[Dry] Artur Dryomov. Detekt.

[Fou] Eclipse Foundation. Mosquitto documentation.

[Fow] Martin Fowler. Inversion of control containers and the dependency injection pattern.

[Git] Git. Git documentation.

[Giu] Arnaud Giuliani. Koin documentation.

[Gra] Gradle. Gradle documentation.

[Jeta] JetBrains. Kmqtt repository.

[Jetb] JetBrains. Kotlin distributed.

[Jetc] JetBrains. Kotlin documentation.

[Jetd] JetBrains. Kotlin multiplatform.

[Kod] KodeinKoders. Kodein documentation.

[Kot] Kotest. Kotest documentation.

[Mic] Microsoft. Windows documentation.

[mqta] mqttjs. Mqtt.js repository.

[MQTb] MQTT.org. Mqtt official documentation.

[MQTc] MQTT.org. Mqtt version 5.0 specification.

Acknowledgements

Optional. Max 1 page.