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

dedica opzionale. Max a few lines.

Contents

Chapter 1

Introduzione

Structure of the Thesis

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

Chapter 2

Background

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, Message Queuing Telemetry Transpor (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

2.3 Message Queuing Telemetry Transport (MQTT)

Nel vasto panorama dell' Internet of Thing (IoT), dove milioni di dispositivi sono interconnessi per scambiare dati e informazioni in tempo reale, MQTT si distingue come uno dei protocolli di comunicazione più importanti e ampiamente adottati. La sua storia affonda le radici nella necessità di comunicare in modo efficiente tra dispositivi con risorse limitate, come sensori,ù attuatori e dispositivi embedded.

MQTT si presenta come una soluzione elegante per risolvere le sfide di comunicazione in un ambiente 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 tra i partecipanti alla rete.

Il modello publish-subscribe di MQTT è fondamentale per comprendere il suo funzionamento. In questo modello, 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.

Un aspetto cruciale di MQTT è la sua capacità di mantenere aperte le connessioni tra i dispositivi, riducendo al minimo la latenza e ottimizzando l'efficienza della comunicazione. Questo è particolarmente importante in contesti in cui è necessario uno scambio continuo di dati in tempo reale, come il monitoraggio ambientale, la gestione degli impianti industriali e la sorveglianza di sicurezza.

2.3.1 Quality of Service (QoS)

MQTT offre funzionalità avanzate per la gestione delle connessioni, la sicurezza e la qualità del servizio (Quality of Service (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 Quality of Service (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.

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.

2.4 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, semplificando 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.

Chapter 3

Requirements

3.1 Scalabilità

Il modulo di comunicazione sviluppato per il sistema di pulverizzazione e computazione distribuita PulvReAKt deve supportare la scalabilità orizzontale, permettendo l'aggiunta di nuovi nodi senza interruzioni del servizio. Questo implica che l'architettura deve essere in grado di adattarsi dinamicamente all'aumento delle richieste, distribuendo il carico tra i nuovi nodi in modo efficiente. Una soluzione di scalabilità orizzontale efficace dovrebbe prevedere meccanismi di auto-scaling, dove i nuovi nodi vengono aggiunti automaticamente in risposta a picchi di carico e ridotti durante i periodi di bassa domanda, ottimizzando l'uso delle risorse e minimizzando i costi operativi. Inoltre, il sistema deve essere in grado di gestire l'elasticità dell'infrastruttura, permettendo il provisioning e de-provisioning dei nodi in maniera automatizzata e senza interruzioni.

Per gestire la comunicazione diretta tra molteplici componenti, è stata scelta una struttura dei topic MQTT con un topic principale (PulvReAKt) all'interno del quale avvengono tutte le comunicazioni, secondo una struttura organizzata gerarchicamente. Seguendo lo schema PulvReAKt/mittente/destinatario, ad esempio, se un componente A volesse scrivere a un componente B, il topic del messaggio sarebbe PulvReAKt/A/B. Questo approccio consente una gestione ordinata e tracciabile della comunicazione, facilitando la manutenzione e l'analisi dei flussi di messaggi all'interno del sistema distribuito.

Inoltre, questa struttura permette di filtrare e gestire facilmente i messaggi in base alla sorgente e alla destinazione, migliorando l'efficienza della comunicazione e riducendo il rischio di collisioni o confusione tra i messaggi inviati dai diversi componenti. La scelta di una struttura dei topic chiara e ben definita è cruciale per assicurare che la comunicazione tra i componenti rimanga scalabile e manutenibile man mano che il sistema cresce e si evolve.

Ecco la versione estesa del testo:

3.2 Affidabilità

Un aspetto cruciale nella comunicazione tra i componenti è che deve essere robusta e affidabile per garantire il corretto funzionamento del sistema distribuito. I componenti devono poter inviare e ricevere messaggi utilizzando il protocollo di comunicazione MQTT. Questo richiede l'integrazione di librerie specifiche per MQTT, assicurando che i messaggi possano essere formattati, inviati e ricevuti correttamente. Deve essere garantita l'affidabilità nella consegna dei messaggi, in alcuni casi assicurando che ogni messaggio venga ricevuto dal destinatario previsto. Ciò implica la gestione di conferme di ricezione, meccanismi di ritrasmissione in caso di fallimento e gestione delle code di messaggi per evitare perdite, che sono tutti compresi all'interno del protocollo MQTT in caso si decida di usare una QoS di livello 1 o 2.

La gestione delle code di messaggi è 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.

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.

L'implementazione di meccanismi di failover e ripristino automatico è essenziale

per garantire la disponibilità continua del servizio. Ad esempio, l'uso di cluster di broker MQTT ridondanti può prevenire l'interruzione del servizio in caso di guasto di uno dei nodi. Inoltre, la configurazione di backup e ripristino regolari dei dati critici garantisce che il sistema possa essere rapidamente riportato allo stato operativo in caso di disastri.

L'affidabilità può essere ulteriormente migliorata attraverso l'uso di tecniche di testing rigorose, come i test di stress e i test di carico, per valutare le prestazioni del sistema sotto condizioni estreme. L'adozione di pratiche DevOps, con pipeline CI/CD (Continuous Integration/Continuous Deployment), assicura che le nuove versioni del software vengano distribuite in modo sicuro e senza interruzioni, mantenendo un elevato standard di qualità e affidabilità.

3.3 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.

3.3.1 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.

3.3.2 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.
- **Ecossistema 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, l'interazione con API REST, la manipolazione dei dati JSON e molto altro ancora, consentendo agli sviluppatori di sfruttare le funzionalità esistenti senza dover reinventare la ruota.
- **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.

- **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 wen.

3.3.3 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 caratteristcihe principale delle piattaforme native sono:

- **Esecuzione Diretta**: Compilazione in codice nativo che viene eseguito direttamente dall'hardware, senza layer di interpretazione. Che 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. Consentendo agli sviluppatori di creare applicazioni che interagiscono direttamente con l'hardware sottostante, sfruttando appieno le capacità del dispositivo target.

- **Assenza di Overhead**: Nessun overhead dovuto alla gestione della macchina virtuale o alla garbage collection, sebbene Kotlin/Native includa un garbage collector. Questo significa che le risorse del sistema possono essere utilizzate in modo più efficiente, riducendo il consumo di memoria e CPU.

- **Cross-platform**: Supporto per diverse piattaforme, inclusi iOS, Windows, Linux e macOS, facilitando lo sviluppo di applicazioni multi-piattaforma. Consentendo agli sviluppatori di scrivere un'unica base di codice Kotlin che può essere compilata per eseguire su una varietà di dispositivi e sistemi operativi, riducendo al minimo lo sforzo di sviluppo e la complessità del codice.

Chapter 4

Design

4.1 Struttura delle classi

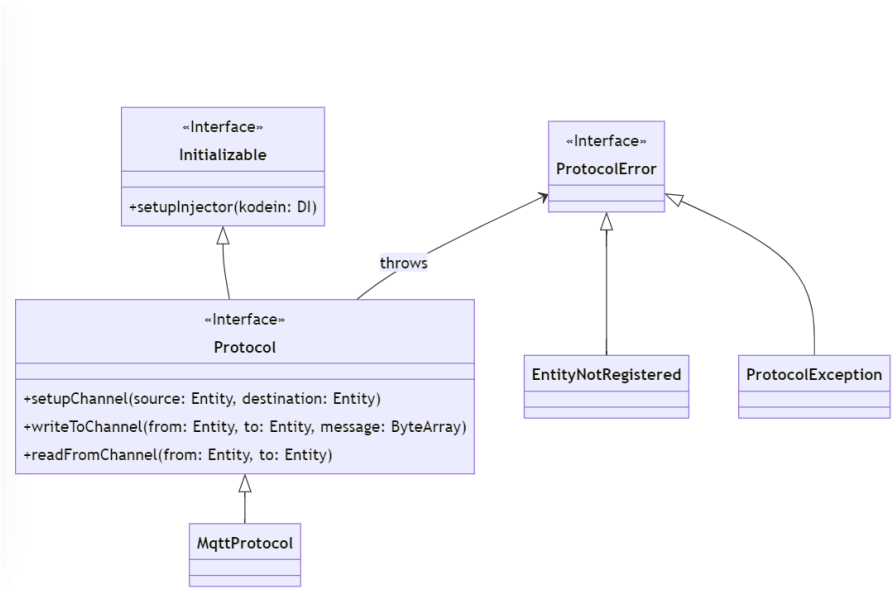

Figure 4.1: Diagramma delle classi

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

- `InjectAwareResource`: Questa è un'interfaccia che definisce il metodo utilizzato per configurare l'iniettore delle dipendenze.

- `ProtocolError`: 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 generale del protocollo.

- `Protocol`: Un'interfaccia che definisce i metodi essenziali per la gestione del protocollo di comunicazione. Questi metodi includono:

  - `setupChannel`: Metodo per configurare il canale di comunicazione tra due entità. Più chiamate a questo metodo con gli stessi `source` e `destination` dovrebbero essere idempotenti. Il runtime chiama questo metodo per configurare la comunicazione tra le entità; l'utente finale non dovrebbe interagire direttamente con questo metodo.
  - `writeToChannel`: Metodo per scrivere dati sul canale di comunicazione specificato. Questo metodo può riuscire o fallire con un `ProtocolError`. Fallisce se le entità `from` e `to` non sono registrate nel protocollo tramite il metodo `setupChannel`.
  - `readFromChannel`: Metodo per leggere dati dal canale di comunicazione specificato. Questo metodo restituisce un `Flow` caldo di messaggi che può essere consumato dal runtime. Può riuscire o fallire restituendo un `ProtocolError`. Fallisce se le entità `from` e `to` non sono registrate nel protocollo tramite il metodo `setupChannel`.

- `MqttProtocol`: Una classe concreta che implementa l'interfaccia `Protocol`. Questa classe fornisce l'implementazione specifica per il protocollo MQTT, 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.

4.2 struttura dei Package

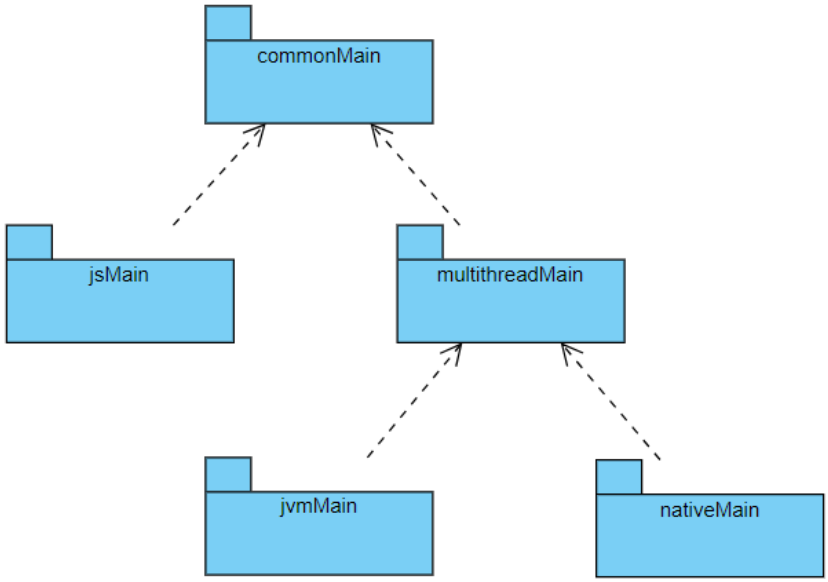

Figure 4.2: Diagramma dei package e dei moduli del progetto

Data la natura multiplatform del progetto, si è scelto di utilizzare una libreria comune per la JVM e native, mentre, data la natura singlethread di JS, si è optato per una libreria specifica per lo sviluppo con target NodeJS. 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.

Il modulo comune per la JVM e native è chiamato `multithreadedMain` e contiene l'implementazione comune ad entrambi i 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 `multithreadedMain` 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 `multithreadedMain` dipendono due moduli specifici per ogni target:

- `jvmMain`: Questo modulo estende `multithreadedMain` per includere ottimiz-

zazioni 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 `multithreadedMain` 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 correttamente sia individualmente che nel contesto dell'intero progetto.

4.3 Scenari Soggetti a Test

Il test valuta la robustezza di un protocollo di comunicazione MQTT attraverso una serie di scenari distinti. Innanzitutto, viene verificato che il protocollo possa essere inizializzato e finalizzato senza incorrere in errori. Questa fase è critica poiché assicura che tutte le operazioni di setup e di chiusura del protocollo siano eseguite correttamente, garantendo uno stato coerente prima e dopo il test.

Uno degli aspetti cruciali della valutazione consiste nell'esaminare come il protocollo gestisce gli errori in situazioni critiche. In particolare, si verifica come il

protocollo si comporta quando si tenta di scrivere su un canale non registrato e quando si tenta di leggere da un canale non registrato. È fondamentale che il protocollo sia in grado di identificare e segnalare tali errori in modo accurato, fornendo messaggi di errore comprensibili che facilitino la risoluzione dei problemi.

Infine, il test include scenari che simulano la comunicazione effettiva tra due entità registrate nel protocollo. Questi scenari permettono di verificare il flusso completo di comunicazione, dall'inizializzazione alla trasmissione e alla ricezione dei messaggi, fino alla finalizzazione del protocollo. Questa fase del test è cruciale poiché dimostra l'efficacia del protocollo nella gestione della comunicazione bidirezionale tra le entità coinvolte.

Questi scenari, inclusi nei test, mirano a 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.

Ecco una versione più estesa del capitolo:

4.4 Tecnologie Utilizzate

4.4.1 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) ne facilita l'adozione in progetti di qualsiasi dimensione e complessità.

4.4.2 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.

4.4.3 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.

4.4.4 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 sempliceda 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.

4.4.5 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.

4.4.6 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.

4.4.7 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.

4.4.8 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) 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.

Chapter 5

Implementazione e Validazione

5.1 Sviluppo

5.2 Validazione

5.3 Prestazione sulle diverse piattaforme

5.4 Integrazione in PulvReAKt

Chapter 6

Conclusioni

Bibliography

[Ak24] Arrow-kt. Arrow documentation, 2024.

[Dc24] Artur Dryomov and contributors. Detekt, 2024.

[Fou24] Eclipse Foundation. Mosquitto documentation, 2024.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern, 2004.

[Git24] Git. Git documentation, 2024.

[Giu24] Arnaud Giuliani. Koin documentation, 2024.

[Gra24] Gradle. Gradle documentation, 2024.

[Jet23] JetBrains. Kotlin distributed, 2023.

[Jet24a] JetBrains. Kmqtt repository, 2024.

[Jet24b] JetBrains. Kotlin documentation, 2024.

[Jet24c] JetBrains. Kotlin multiplatform, 2024.

[Kot24] Kotest. Kotest documentation, 2024.

[MQT19] MQTT.org. Mqtt version 5.0 specification, 2019.

[mqt24] mqttjs. Mqtt.js repository, 2024.

Acknowledgements

Optional. Max 1 page.