Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Adozione del framework Arrowhead per l'automazione di scenari IoT: Discovery, Orchestrazione e Coreografia.

Relatore:
Dott.
FEDERICO MONTORI

Presentata da:
RODOLFI MATILDE

II Sessione - secondo appello
Anno Accademico 2021/2022

Introduzione

L'IoT, acronimo per *Internet of Things*, è uno dei paradigmi tecnologici più utilizzato degli ultimi anni. Ma cos'è esattamente?

Per IoT si intende un sistema dinamico di oggetti intelligenti che possono collegarsi alla rete, identificarsi in modo univoco, raccogliere, scambiare ed/o elaborare dati, condividere il proprio stato e modificare il proprio comportamento in base ai dati e all'ambiente circostante in totale autonomia grazie a standard e protocolli di comunicazione specifici. Il potenziale applicativo è sconfinato: domotica, trasporti, medicina, agricoltura, allevamento, pubblica amministrazione e tutela ambientale sono solo alcuni dei molteplici campi di impiego. Il suo utilizzo fino ad ora ha aiutato a migliorare la qualità della vita e la competitività aziendale.

Gli oggetti su cui si basa principalmente l'IoT, a differenza di quanto si possa comunemente pensare, non sono i classici dispositivi tecnologici come computer, tablet o telefoni, ma bensì oggetti fisici di uso quotidiano come lampadine, automobili, elettrodomestici, impianti produttivi, dispositivi medici ed esseri viventi di vari tipi come animali selvatici, bestiame o piante. L'IoT permette di creare nella rete un equivalente dell'oggetto fisico in modo che esso possa raccogliere dati o modificare il proprio stato o quello di altri oggetti in base all'ambiente circostante in tempo reale e senza l'intervento umano, permettendo ad ogni oggetto, anche quelli più semplici, di acquistare un ruolo attivo.

Il primo a parlare chiaramente e a dare una definizione di Internet of Things è stato Kevin Ashton nel 1999, anche se i primi esperimenti di collegamento di oggetti a sensori accessibili da terze parti risalgono ai primi anni del decennio precedente, nel '82 vi fu, infatti, il primo dispositivo IoT: un distributore automatico di Coca-Cola modificato alla

Carnegie Mellon University per riportare il proprio inventario e segnalare la temperatura delle bevande al suo interno.

Le tecnologie utilizzate per collegarsi alla rete sono molteplici, ma il più delle volte a corto raggio. I tag RFID (sistema formato da dispositivi passivi a corto raggio, leggibili tramite radiofrequenza) sono stati uno dei primi metodi di comunicazione dei dati utilizzati, poi con l'avvento di nuove tecnologie e di nuovi standard di comunicazione si sono potuti sfruttare collegamenti alla rete Intenet tramite reti cellulari, Ethernet o WiFi. È comune anche l'uso di PLC, NFC e Bluetooth per la comunicazione tra oggetti che non necessitano di una connessione a Internet.

Per identificare un dispositivo IoT esso necessita di un codice univoco di riferimento, che può essere principalmente di 3 tipi: Electronic Product Code (o EPC, un codice di 64 o 98 bit registrato elettronicamente su un RFID che memorizza oltre al numero seriale, anche alcune specifiche dell'oggetto in questione), indirizzi IP (di tipo IPv4 o IPv6, sono codici che identificano un dispositivo all'interno di una rete) o barcode (rappresentazione grafica tramite rettangoli e spazi di varie dimensioni di un codice alfanumerico, pensati per essere letti facilmente dalle macchine tramite scanner laser o videocamere)

Un altro fattore determinante per la diffusione di massa dell'IoT è stata la creazione di sensori per la raccolta dati affidabili, di hardware miniaturizzati e allo stesso tempo molto potenti e di batterie efficienti e leggere, il tutto accompagnato da una diminuzione del loro prezzo di mercato.

Non bisogna poi dimenticare che per la creazione di un sistema IoT robusto e sicuro è necessaria l'integrazione con le altre tecnologie di punta nel campo informatico, come Big Data, per la raccolta di dati sull'utilizzo e il funzionamento dei dispositivi IoT, Artificial Intelligence, per l'analisi intelligente dei dati forniti, Blockchain, per la certificazione dell'integrità e della provenienza dei dati forniti, ed Edge Computing, per l'aggregazione ed elaborazione in real time dei dati.

Per avere un'implementazione di successo dell'IoT è importante creare un sistema dinamico e interoperabile, che garantisce l'affidabilità e la privacy dei dati e degli utenti con consumi energetici efficienti. Queste rappresentano tutt'oggi le sfide del settore IoT e quindi i maggiori campi di ricerca e di investimento.

Lo sviluppo in campo IoT è principalmente sponsorizzato dalle grandi aziende che

ricavano grandi vantaggi dal suo utilizzo: maggiore efficienza, accelerazione dei processi, riduzione degli errori, prevenzione dei furti e creazione di sistemi complessi ma flessibili sono solo alcuni dei vantaggi ottenuti dall'utilizzo dell'IoT.

Il mercato dell'IoT si è sviluppato del molto negli ultimi anni, gli studi calcolano il suo valore nel 2021 in circa 150 miliardi di dollari con un incremento del 22.4% rispetto all'anno precedente [4] e stimano che nel 2025 i dispositivi IoT connessi saranno circa 27 miliardi [5]. Nello specifico solo in Italia gli ultimi studi calcolano circa 110 milioni di dispositivi IoT attivi, cioè circa 1,8 per persona [6] e, si stima, aumenteranno notevolmente nei prossimi anni, anche grazie ai fondi stanziati dal PNRR per lo sviluppo Smart.

L'IoT è, come abbiamo visto, un campo di ricerca attivo e ad alto interesse a livello internazionale, con finanziamenti sia pubblici che privati. È, quindi, facile immaginare quanti progetti e ricerche sono ad ora all'opera per risolvere e innovare questo settore. Uno di questi progetti si chiama Arrowhead Framework e questa tesi studierà il suo funzionamento e proporrà una demo di utilizzo con le sue componenti principali.

Indice

Elenco delle figure

Capitolo 1

Abstract

In questa tesi approfondiamo l'utilizzo del framework Arrowhead per creare una demo di un sistema per l'automazione IoT industriale.

Il framework fornisce dei core specifici per il controllo di un sistema industriale nell'ottica dell'industria 4.0. Nello specifico prendiamo in analisi i tre core principali di Arrowhead, più un quarto, il Choreographer, interessante perchè fornisce servizi non comuni, ma molto utili, per la gestione di sistemi industriali.

La demo è stata suddivisa in livelli per permettere un'analisi dettagliata dei vantaggi e delle scelte implementative che l'integrazione dei vari core apporta alla demo. Si è scelto, inoltre, di analizzare separatamente le scelte teoriche e pratiche che sono state effettuate nello sviluppo della demo.

In questa tesi, quindi, analizziamo l'integrazione del framework Arrowhead all'interno di un sistema di IoT industriale e porteremo alla luce i vantaggi del suo utilizzo e per quale motivo non risulta essere competitivo nel suo mercato di appartenenza.

Capitolo 2

L'industria 4.0 e Arrowhead

Negli ultimi 300 anni l'industria è stata modificata radicalmente grazie all'introduzione di invenzioni innovative: la meccanizzazione e la forza vapore, il lavoro in catena di montaggio e la conseguente produzione di massa, le macchine automatiche e l'Iot, non hanno solo cambiato il modo di lavorare e di produrre, ma anche tutta la società, diminuendo i costi di produzione e aumentando il benessere dei lavoratori.

Le capacità richieste ai lavoranti si sono anch'esse evolute: inizialmente si richiedevano lavoratori con molta forza fisica e con competenze specifiche per produrre un determinato oggetto dall'inizio alla fine, ora invece i lavoratori effettuano un lavoro di supervisione dei macchinari intervenendo ove necessario, necessitando quindi di sole conoscenze generiche del funzionamento dei macchinari e delle interazioni tra essi e riducendo il lavoro umano necessario solo in fase di installazione e manutenzione dei macchinari.

È facilmente intuibile quanto sia importante creare un sistema aziendale efficiente, sicuro e aperto a modifiche, così da diminuire e semplificare il più possibile l'intervento umano necessario durante l'intero ciclo di vita del sistema. È, quindi, fondamentale integrare, in fase di creazione o ristrutturazione, strumenti informatici che permettano ciò.

Il problema principale dell'industria 4.0 è che, al momento, coesistono livelli di integrazione differenti: un misto di sistemi basati su Bus con alcune integrazioni di Ethernet (con più latenza, ma maggiore affidabilità) e soluzioni Wireless (più pratico, ma meno affidabile), che utilizzano, per lo più, protocolli non standardizzati o non aggiornati.

Il più delle volte in questi sistemi sono presenti, inoltre, sistemi embedded[1] o dispositivi obsoleti ma non rimpiazzabili, rendendo maggiormente complicata l'integrazione e l'interconnessione tra i dispositivi del sistema.

Una grande tendenza degli ultimi anni da parte delle aziende è di prediligere applicazioni basate su cloud, così da non dover gestire in prima persona il sistema informatico, con una maggiore diminuzione e prevedibilità dei costi IT da un lato e dall'altro così da creare modelli unificati per una più facile gestione da parte dei creatori di software. È, inoltre, cambiato le modalità di accesso ai dati industriali: ora sono dispositivi mobili come smartphone o tablet i punti finali d'accesso, il quale deve essere sempre possibile, indipendentemente da dove essi si trovino; i sistemi informatici industriali devono, quindi, garantire velocità e sicurezza di comunicazione anche se i dispositivi finali si trovano fisicamente all'esterno del sistema stesso.

Ogni struttura aziendale ha requisiti differenti che vanno privilegiati, i più comuni sono connessione in tempo reale, mobilità, sicurezza, protezione da attacchi esterni e disponibilità. Il sistema, quindi, indipendentemente dall'eterogeneità dei dispositivi e dei metodi di comunicazioni presenti al suo interno, deve essere ottimizzato per soddisfare questi requisiti, compito non sempre semplice.

La creazione di reti di comunicazione, però, non è sufficiente, sono altrettanto importanti metodi per la modellizzazione e la descrizione delle informazioni e di come accedervi. Anche in questo caso vi sono molteplici standard non compatibili tra loro e per lo più poco completi e non esaustivi.

L'assenza di tecnologie efficienti, sicure, veloci e universali ha permesso lo svilupparsi di numerosi progetti che si propongono di raggiungere, almeno in parte, questi obbiettivi. Alcuni di questi hanno come obbiettivo la creazione di toolchain[2] che permettano uno sviluppo strutturato e ben pianificato di sistemi industriali che consentono l'integrazione con sistemi già presenti e che interagiscono, tramite uno schema di orchestrazione, in maniera automatica, senza bisogno di intervento umano.

[1] sistemi a microprocessore specificamente progettato, quindi non riprogrammabili

[2] insieme di programmi utilizzati per lo sviluppo di un programma o di un sistema informatico

2.1 Arrowhead

L'obbiettivo di Arrowhead Tools è di permettere la creazione o la modernizzazione di sistemi industriali nell'ottica dell'industria 4.0, affrontando i problemi di interoperabilità e integrazione presenti ad oggi e creando al contempo sistemi dinamici, robusti e sicuri; inoltre fornisce strumenti per migliore le performance di un sistema embedded e soluzioni per l'integrazione di *legacy systems* (sistemi obsoleti, ma non rimpiazzabili)

Il framework Arrowhead[3] si basa sul concetto di Architettura Orientata ai Servizi, consentendo a tutti i dispositivi nel sistema di lavorare in un approccio comune e unificato, così da permettere l'interoperabilità tra quasi tutti gli elementi IoT. Per far ciò crea Cloud Locali (LC) autonomi, che permettono, rispetto al Cloud globale, una gestione in tempo reale dei dati, una maggiore sicurezza di essi e del sistema stesso e una maggiore scalabilità. Arrowhead fornisce, inoltre, strumenti per la creazione di SoS, acronimo per *System of Systems*, orientati ai servizi.

2.2 Core Systems Arrowhead

Un LC è la più piccola unità di governance all'interno dell'Arrowhead Framework, in esso sono presenti, oltre ai sistemi IoT, vari Core Sistems, sistemi che forniscono differenti servizi. Non tutti i core sono obbligatori, ad accezione di quelli che garantiscono le funzionalità minime necessarie per consentire la collaborazione e lo scambio sicuro di informazioni tra i dispositivi.

I Core Systems fondamentali sono:

- *Service Registry*, un database contenente le informazioni riguardanti i servizi che sono offerti attualmente nel Cloud locale
- *Authorization*, una lista di permessi per utilizzare i servizi, sia all'interno del Cloud locale che all'esterno
- *Orchestrator*, un sistema che, una volta che un sistema applicativo si connette ad esso, indica la lista dei servizi con cui esso può interagire

[3]framework e documentazione completa su github.com/eclipse-arrowhead/core-java-spring

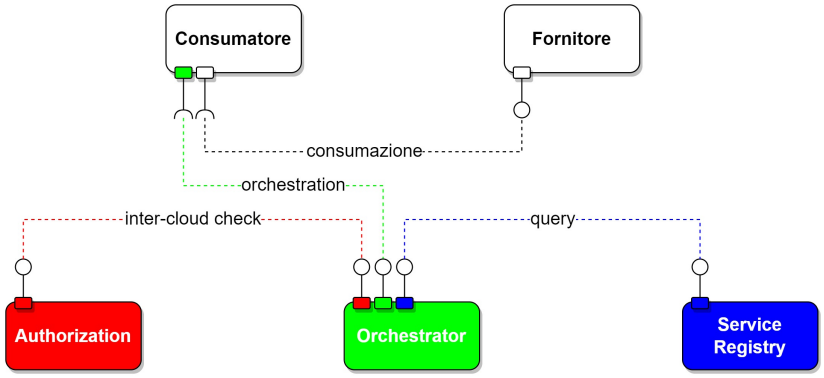

Figura 2.1: Comunicazione tra i Core Systems principali e i sistemi applicativi

Quelli facoltativi sono:

- *Choreographer*, un sistema di supporto che consente di eseguire flussi di lavoro predefiniti tramite l'orchestrazione e il consumo di servizi
- *Plant Description*, una vista astratta sui sistemi contenuti nell'impianto e su come essi sono collegati come consumatori e produttori, viene utilizzata per popolare l'Orchestrator tramite una descrizione astratta.
- *Event Handler*, ha lo scopo di fornire un sistema di messaggistica di pubblicazione e sottoscrizione autorizzato all'Arrowhead Framework
- *Gatekeeper*, un servizio con lo scopo di fornire funzionalità di manutenzione inter-Cloud per la ricerca e la negoziazione con altri Cloud locali
- *Gateway*, un sistema di supporto con lo scopo di stabilire un percorso dati protetto, se necessario, tra un consumatore e un provider situato in diversi Cloud
- *Data Manager*, un database per immagazzinare i dati provenienti dai sensori
- *Time Manager*, un sistema che aiuta a fornire servizi basati sul tempo e sulla posizione
- *Certificate Authority*, un sistema per l'emissione di certificati firmati da utilizzare nel Cloud locale

2.3 Orchestrazione e Coreografia

Nel mondo dell'architettura orientata ai servizi vi sono sempre stati i concetti di orchestrazione e coreografia: due differenti metodi per la gestione della cooperazione dei servizi. Prima di utilizzare i core che implementano questi concetti analizzeremo in questa sezione le loro caratteristiche e differenze.

L'orchestrazione si basa su un sistema centrale che gestisce quali elementi possano utilizzare determinati servizi. Vi è, quindi, un elemento centrale che supervisiona ogni interazione, controllando che ogni elementi esegua solo la parte che gli è stata assegnata ed evitando interazioni non autorizzate. Sono, quindi, gli elementi stessi a decidere quando consumare un determinato servizio, per poi richiedere l'autorizzazione all'orchestratore.

La coreografia, invece, si basa sulla creazione di un piano di lavoro che determina quali operazioni devono essere eseguite e in quale ordine. Gli elementi, quindi, non decidono più quando eseguire una determinata azione, ma aspettano che l'operazione precedente nel piano di lavoro sia conclusa. La verifica dell'avvenuto termine dell'operazione precedente può avvenire in due differenti modi: attesa attiva, l'elemento campiona l'operazione precedente fin quando essa non risulta conclusa, o attesa passiva, l'elemento attende di essere richiamato all'avvio dall'operazione conclusa. Il Framework Arrowhead integra il secondo metodo di attesa e permette di avere un core centrale che gestisce l'avvio delle operazioni nel giusto ordine e a cui notificare la fine dell'esecuzione delle operazioni, così da diminuire ulteriormente le conoscenze di cui necessita ogni elemento.

L'orchestrazione è il metodo attualmente più utilizzato in quanto sfrutta una struttura logica già impiegata anche nei sistemi più vecchi, risultando, quindi, facilmente integrabile anche in essi. La coreografia, invece, necessita di una modifica dell'architettura del sistema.

La coreografia, d'altro canto, permette un'ordine ferreo delle operazioni e una più semplice modifica in caso si vogliano cambiare le interazioni tra gli utenti.

Capitolo 3

Architettura

Questa tesi si pone l'obbiettivo di creare una demo di utilizzo del framework Arrowhead Eclipse.

La demo ripropone un classico esempio di sistema industriale, riproponendo al suo interno i 3 classici elementi universalmente presenti in esso:

- un sensore, in questo caso utilizzato per monitorare la batteria di un computer
- un database, per conservare i dati raccolti
- un'interfaccia utente, per la visualizzazione dei dati e la modifica delle informazioni del sistema

La demo è stata divisa in 4 livelli applicativi che permettono di analizzare passo a passo l'integrazione del framework Arrowhead in un sistema già funzionante, sia da parte del progettista del sistema sia dell'utilizzatore finali.

In questo esempio specifico l'utilizzo del framework risulta più macchinoso di quanto non sia progettare il sistema senza di esso, ma permette, anche in un sistema così piccolo, di modificare alcune variabili delle interazioni del sistema senza dover modificare ogni volta il codice degli elementi interpellati, permettendo una maggiore scalabilità e versatilità in caso di future modifiche. È questo il principale punto di forza del framework che si vuole enfatizzare con questa demo.

Si è scelto di utilizzare solo alcuni dei core messi a disposizione da Arrowhead, nello specifico i 3 principali e il Choreographer, questo perchè sono questi i core più caratte-

rizzanti di questo framework, in quanto gli altri tipi di core sono servizi comunemente offerti anche da framework concorrenti a differenza di quelli analizzati, specialmente il Choreographer.

Di seguire si illustrerà le scelte architetturali effettuate, analizzando nello specifico i core utilizzati e descrivendo le interfacce e i tipi di comunicazioni che avvengono tra i vari elementi del sistema, tutto suddiviso per livelli implementativi.

3.1 Demo base - Livello 0

In questo livello analizzeremo l'architettura del sistema senza alcuna implementazione dei core Arrowhead, così da concentrarci meglio su cosa deve fare ogni elemento della nostra demo, come essi devono interagire tra loro e, quindi, che servizi essi devono offrire.

3.1.1 Sensore

Il nostro sensore deve essere rappresentare un classico sensore IoT, il suo compito principale, quindi, deve essere quello di rilevare dati ogni tot secondi, memorizzandoli in una memoria provvisoria, e renderli accessibili da elementi esterni. Il sensore, quindi, interagisce, con gli altri elementi del sistema in modo passivo, fornendo servizi ma non consumandone.

Si è scelto di permettere all'utilizzatore di decidere il tempo di attesa tra un campionamento e il successivo e la grandezza della memoria in cui essi vengono salvati; questa scelta non è sempre possibile in quanto in alcuni sensori il timer di campionamento è implementato fisicamente e/o la memoria a disposizione è fissa; questo però non riduce la generalità di questo esempio in quanto, in presenza di questi casi specifici, basta omettere la parte corrispondente alla gestione del timer e/o della memoria.

Si deduce, quindi, che il sensore debba fornire i seguenti servizi:

- Fornire i dati: deve permettere all'utilizzatore di accedere ai dati registrati fino a quel momento, cancellandoli dalla propria memoria interna; i dati dovranno riportare ognuno la percentuale di batteria registrata (*percent*), se il dispositivo è in carica (*plugged*) e la data e l'ora precisa del campionamento (*time*).

- Modificare la configurazione interna: deve permettere all'utilizzatore di modificare il tempo di attesa tra un campionamento e il successivo (*sampling time*) e la dimensione della memoria in cui i dati vengono salvati (*buffer length*)

3.1.2 Database

Il nostro database ha il funzionamento di un classico database industriale: legge i dati campionati dal sensore, li immagazzina a lungo termine e li rende accessibili ove si necessiti di accedervi. Il database, quindi, interagisce con gli altri elementi del sistema sia in modo attivo sia in modo passivo, sia fornendo servizi che consumandone.

Si è scelto di permettere all'utilizzatore di scegliere ogni quanto il database debba leggere i dati campionati dal sensore e di eliminare tutti i dati all'interno del database. Anche in questo caso, come successo con il sensore queste scelte non influiscono sulla generalità di utilizzo del database in quanto in caso queste due funzionalità non dovessero essere utili basta rimuovere le relative parti.

Il database deve, quindi, implementare i seguenti servizi:

- Fornire i dati: deve permettere all'utilizzatore di accedere ai dati registrati fino a quel momento; i dati dovranno riportare ognuno la percentuale di batteria registrata (*percent*), se il dispositivo è in carica (*plugged*) e la data e l'ora precisa del campionamento (*time*).

- Eliminare i dati: deve permettere all'utente di eliminare tutti i dati registrati fino a quel momento;

- Modificare la configurazione interna: deve permettere all'utilizzatore di modificare il tempo di attesa tra una lettura dei dati campionati dal sensore e la successiva (*interval*)

3.1.3 Interfaccia utente

L'interfaccia utente deve permettere all'utente finale di modificare le configurazioni interne del database e del sensore, visualizzare i dati presenti nel database e, even-

tualmente, eliminarli. L'interfaccia utente, quindi, interagisce con gli altri elementi del sistema solo in modo attivo, consumando servizi senza fornirne.

Per definizione di interfaccia utente le azioni effettuate dall'utente non hanno tempistiche specifiche, bisogna quindi far sì che i servizi richiesti dall'utente siano sempre accessibili.

3.1.4 Interazioni

In figura 3.1 uno schema del sistema fino ad ora progettato, con i collegamenti di consumatore-fornitore di servizi

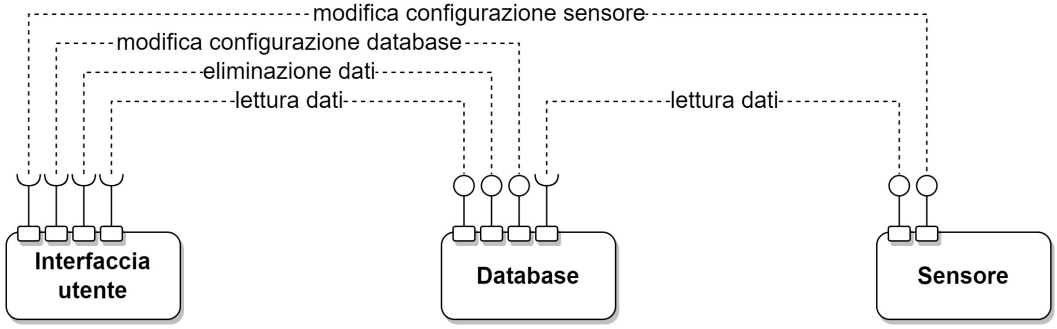

Figura 3.1: Schema Sistema - Livello 0

In figura 3.2 uno schema delle iterazioni tra gli elementi del sistema, come si può facilmente comprendere gli elementi devono conoscere a priori come mettersi in contatto tra loro.

In questa fase le iterazioni sono molto semplici e lineari, ma vedremo che man mano che aumentiamo di livello lo schema si complicherà molto.

3.2 Livello 1

A questo livello di progettazione vogliamo risolvere il problema per cui ogni elemento del sistema deve conoscere tutte le variabili per mettersi in connessione con il servizio di cui necessita. In un sistema di pochi elementi come quello di questa demo questa sembra un'inutile preoccupazione, ma nei sistemi industriali composti da molteplici elementi che

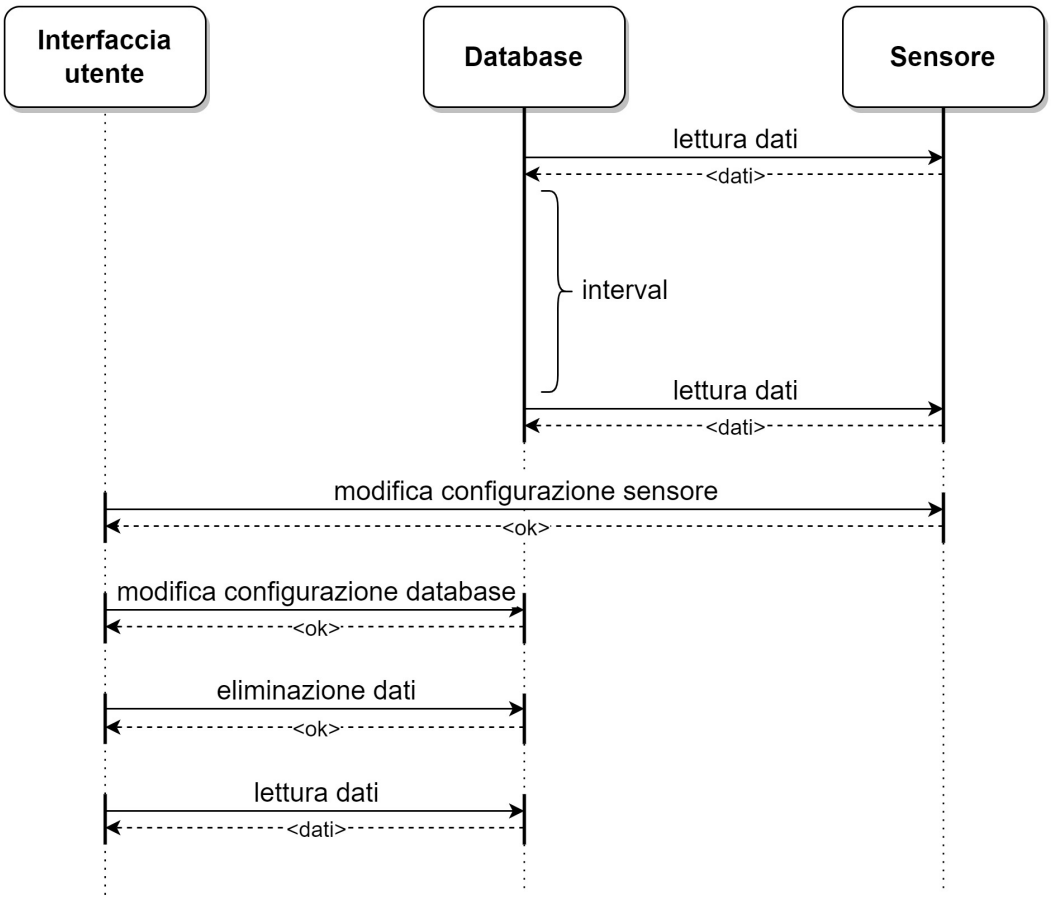

Figura 3.2: Schema Interazioni - Livello 0

possono essere spostati, rimpiazzati o eliminati anche il solo cambiamento dell'indirizzo IP di un servizio potrebbe causare un effetto a catena che causerebbe la modifica di molteplici linee di codice, cosa insostenibile nel lungo periodo.

Per risolvere questo spiacevole inconveniente Arrowhead fornisce un core per l'individuazione dei servizi offerti attualmente nel sistema.

Di seguito analizzeremo il funzionamento di questo core e come variano, di conseguenza, l'architettura e le interazioni tra gli elementi del nostro sistema demo.

3.2.1 Service Registry

Il Service Registry fornisce un database contenente tutte le informazioni relative ai servizi attivi offerti in questo momento all'interno del LC, questo permette ad ogni elemento del sistema che fornisce un servizio di registrare, aggiornare ed eliminare i dati relativi al servizio offerto e agli elementi del sistema che consumano un servizio di accedere ai dati relativi al servizio richiesto.

I servizi offerti da questo core sono molteplici, ma solo 3 di essi sono pubblici:

- Register, servizio per la registrazione o la modifica dei dati e dei metadati relativi al servizio offerto dall'utilizzatore
- Unregister, servizio per l'eliminazione dei dati e dei metadati relativi al servizio offerto dall'utilizzatore
- Query, servizio per trovare e tradurre il nome simbolico di un servizio in un endpoint fisico (fornisce le informazioni necessarie per accedere al servizio)

Gli altri servizi offerti sono servizi utili a verificare lo stato interno e ad accedere a informazioni specifiche, servizi pensati principalmente per essere utilizzati dal Cloud Administrator.

3.2.2 Sensore

Il sensore, fornendo solo servizi ma non consumandone, sfrutta solo i primi due servizi offerti dal Service Registry.

Appena il sensore va online deve registrare i servizi che sta offrendo e appena prima di andare offline li deve eliminare per evitare che nel database del Service Registry siano presenti servizi non più offerti

3.2.3 Database

Il database sia fornendo servizi che consumandone utilizza tutti i servizi visti sopra del Service Registry.

Appena il database va online deve registrare i servizi che sta offrendo e appena prima di andare offline li deve eliminare.

Quando deve interagire con il sensore per leggere i dati campionati il database deve utilizzare il servizio *query* così da ottenere i dati necessari per l'interazione.

3.2.4 Interfaccia utente

L'interfaccia utente utilizzerà solo il servizio *query* del Service Registry, in quanto consuma servizi, ma non ne offre.

Anche in questo caso prima di poter interagire con gli altri elementi del sistema, bisogna richiedere al Service Registry le informazioni necessarie per l'iterazione.

3.2.5 Interazioni

In figura 3.3 uno schema del sistema con le nuove iterazioni

In figura 3.4 uno schema delle iterazioni tra gli elementi del sistema con l'aggiunta del Service Registry.

Possiamo notare, fin da subito, l'utilizzo ripetuto del servizio *query* offerto dal SR. Questo, nonostante sembri un inutile ripetizione, è stato inserito di proposito. I servizi che forniscono i dati di accesso ai servizi, in questo livello *query*, ma nel prossimo *orchestration*, sono servizi che vanno richiamati ogni volta che si necessita di accedere ad un servizio non fornito dai core, anche se vi si è già avuto accesso, questo per garantire che i dati siano sempre aggiornati. È, quindi, un uso scorretto utilizzare tale tipo di servizi solo la prima volta che si usa un servizio, per poi utilizzare i dati ottenuti anche nelle chiamate successive o addirittura utilizzarli per ottenere i dati di accesso ad un

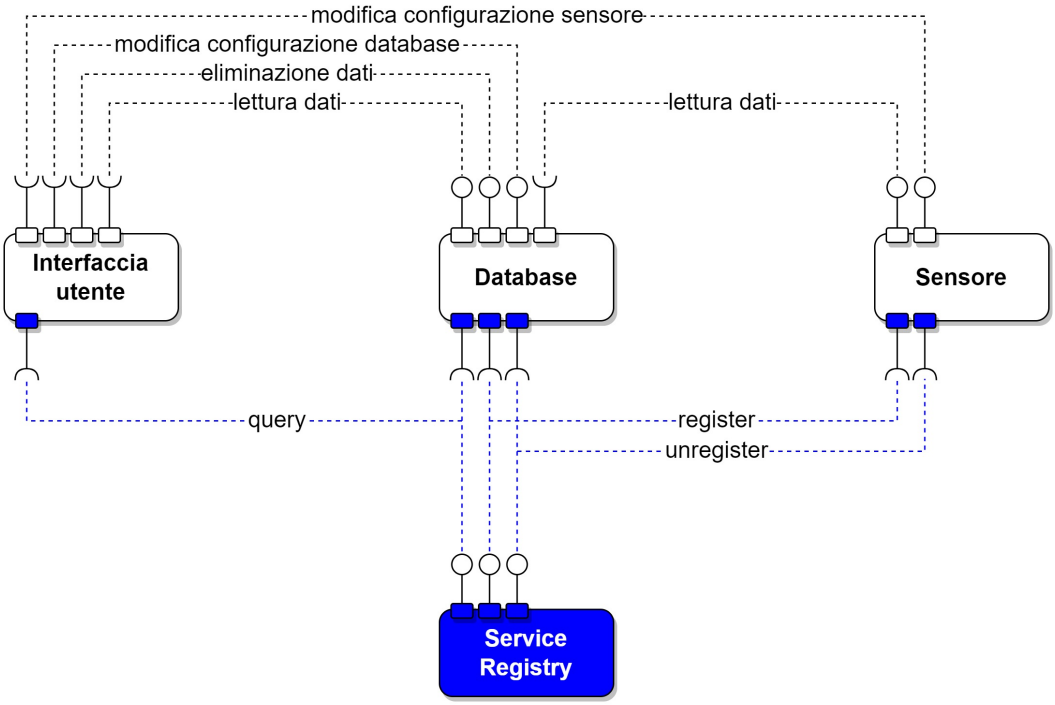

Figura 3.3: Schema Sistema - Livello 1

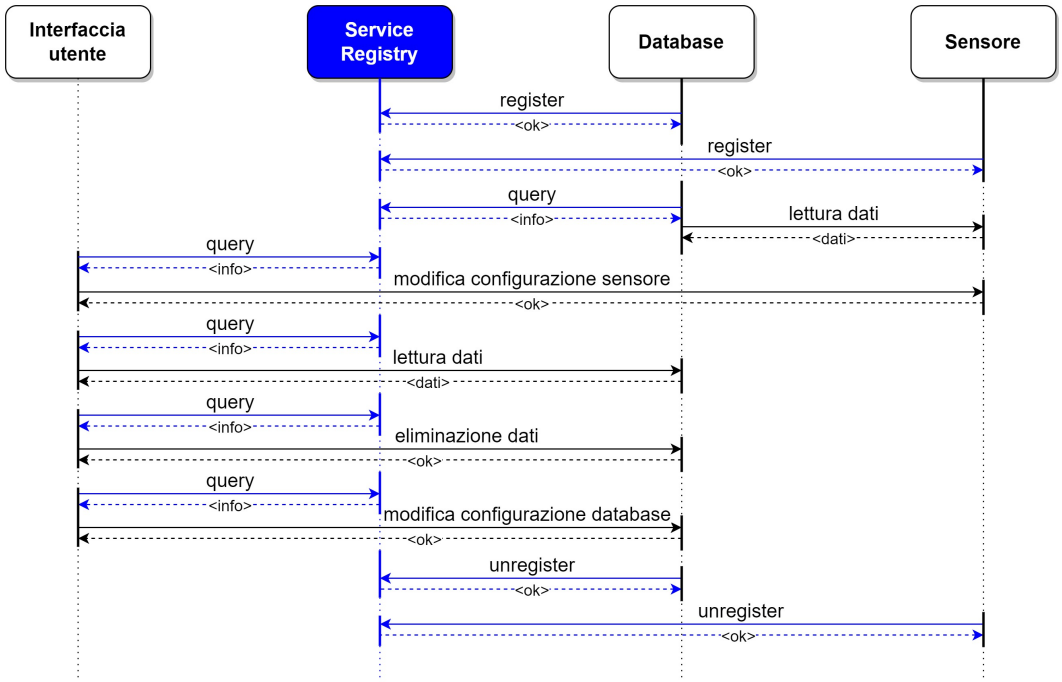

Figura 3.4: Schema Interazioni - Livello 1

determinato elemento del sistema ed utilizzare i dati così ottenuti per accedere anche a tutti gli altri servizi offerti da tale elemento. Questi utilizzi, nonostante ben tollerati in sistemi piccoli e poco modificati nel tempo, in sistemi complessi potrebbero causare grossi problemi, in quanto la struttura del sistema potrebbe cambiare tra una consumazione di servizi e la successiva.

Si può ben vedere, inoltre, che le iterazioni sono diventate più complesse rispetto al grafico precedente, migliorando però la generalità del sistema che ora supporta spostamenti, rimpiazzamenti o eliminazioni di servizi senza alcun intervento da parte dell'amministratore.

Arrowhead, però, sconsiglia questo tipo di utilizzo in quanto potrebbe creare problemi di sicurezza: non vi sono filtri che impediscono l'accesso ai servizi da parte di terzi non autorizzati.

3.3 Livello 2

In questo livello del progetto vogliamo introdurre il sistema base di core consigliato da Arrowhead, aggiungendo, quindi, il core *Authorization* e *Orchestrator*.

Questo livello permette una maggiore sicurezza nelle interazioni, poichè prima che ad un elemento del sistema vengano fornite tutte le informazioni su un determinato servizio il core *Orchestrator* controlla che esso sia abilitato ad accedervi e che sia nella lista, fornita dal Cloud Administrator, dei sistemi con cui può interagire.

In questo livello risulta, quindi, necessario la gestione del sistema da parte di un Cloud Administrator che decide le interazioni autorizzate, apportando modifiche quando necessario. In questa demo questo tipo di operazioni (definiti operazioni di management) saranno gestite dall'interfaccia utente, ma qualora lo si tenesse necessario si potrebbe estrapolare la parte per la gestione delle operazioni creando così un'unità apposita nel sistema.

La gestione manuale delle operazioni di management è consigliabile solo se il sistema è di piccole dimensioni e non vi sono molte variazioni, nella maggior parte dei casi un controllo tramite interfaccia permette una gestione più veloce e semplice delle operazioni di management, facendo sì che possano essere effettuate anche da personale non formato sul funzionamento di Arrowhead.

La gestione manuale delle operazioni può avvenire in due metodi differenti: o si utilizzano i servizi offerti dai core per le operazioni di management, scelta lunga, macchinosa e facilmente soggetta ad errori, o si utilizza il Management Tool offerto da Arrowhead[1], un programma per pc per la gestione dei core principali tramite interfaccia grafica.

Di seguito analizzeremo il funzionamento dei due nuovi core e come variano, di conseguenza, l'architettura e le interazioni tra gli elementi del nostro sistema già esaminati.

3.3.1 Authorization

L'Authorization fornisce un database contenente le autorizzazioni per l'interazione tra elementi all'interno del sistema, inoltre memorizza anche le autorizzazioni per le richieste di interazioni provenienti da altri LC.

[1] scaricabile da www.aitia.ai/downloads/ah-mgmt-tool/

Le autorizzazioni vengono, quindi, divise in due categorie:

- Intra-Cloud, regole che descrivono quale elemento del sistema può consumare un dato servizio fornito da un determinato elemento del sistema
- Inter-Cloud, regole che descrivono quali LC (oltre a sè stesso) sono autorizzati a utilizzare un determinato servizio fornito da questo cloud

In questa demo utilizzeremo solo il primo tipo di autorizzazioni.

I servizi offerti da questo core, anche qui, sono molteplici, ma solo 2 saranno analizzati e utilizzati attivamente all'interno di questa demo:

- Intra-cloud check, è un servizio utilizzabile solo da altri core Arrowhead per verificare se un dato elemento del sistema è autorizzato a consumare un determinato servizio
- Intra-cloud set, servizio di management che crea una regola per autorizzare un dato elemento del sistema a consumare un determinato servizio

Gli altri servizi offerti servono per la gestione Inter-Cloud e per verificare lo stato interno o ad accedere ad informazioni specifiche del core, non utili in questa demo.

3.3.2 Orchestrator

L'Orchestrator fornisce un servizio di associazione tra gli elementi del sistema: su richiesta fornisce al richiedente la lista completa dei servizi che è autorizzato e che dovrebbe consumare e tutti i dati necessari per accedervi.

Per permettere questa orchestrazione il core ha un database interno contenente una lista completa dei servizi che ogni elemento del sistema può consumare e interagisce con l'Authorization per verificare che il collegamento sia permesso e con il Service Registry per ottenere tutte le informazioni inerenti ad un determinato servizio.

I servizi offerti, anche in questo caso sono molteplici, ma in questa demo ne utilizzeremo solo 2:

- Orchestration, è il servizio utilizzato per effettuare la richiesta di orchestrazione

- Store, servizio di management utilizzato per creare una regola di consumazione, definendo che un elemento del sistema può consumare un determinato servizio fornito di un dato elemento del sistema

Una volta che viene effettuata una richiesta di orchestrazione da un elemento del sistema, l'Orchestrator estrae la lista dei servizi che il richiedente deve consumare, per ogni servizio controlla con l'Authorization che sia presente l'autorizzazione di consumo da parte del richiedente, se presente richiede al Service Registry tutte le informazioni del servizio e poi restituisce al richiedente la lista così ottenuta.

È, quindi, necessario un metodo di identificazione univoca non solo dei servizi offerti, ma anche degli elementi del sistema e, nello specifico in Arrowhead, viene usato l'id con cui vengono identificati al momento della registrazione con il Service Registry. Dobbiamo, quindi, usare un ulteriore servizio del Service Registry per la registrazione degli elementi del sistema, anche qualora non forniscano un servizio.

3.3.3 Service Registry

Il suo ruolo e funzionamento rimane invariato, cambiano solo gli utilizzatori del servizio query che non sono più gli elementi della demo, ma l'Orchestrator.

Inoltre, in questo livello della demo utilizzeremo un servizio del Service Registry non illustrato precedentemente:

- Register system, servizio utilizzato per la registrazione di un elemento del sistema, indipendentemente dal fatto che essi forniscano o meno un servizio

- Unregister system, servizio utilizzato per eliminare un sistema da quelli presenti nella lista del SR

3.3.4 Sensore

Il funzionamento del sensore rimane invariato

3.3.5 Database

Il funzionamento del database rimane invariato, se non che al posto della richiesta al Service Registry per informazioni di comunicazione con il servizio di lettura dati, effettua una richiesta di orchestrazione all'Orchestrator.

3.3.6 Interfaccia utente

L'interfaccia utente è l'elemento base della demo che cambia maggiormente in questo livello: essa, infatti, passa da una semplice interfaccia di visualizzazione dati e di modifica dei valori interni degli elementi a interfaccia per la gestione delle operazioni di management del sistema attraverso Arrowhead.

L'interfaccia utente, oltre ai servizi già utilizzati in precedenza, consumerà anche tutti i servizi di management sopra illustrati così da permettere al Cloud Administrator di effettuare tutte le operazioni di management tramite un'interfaccia grafica facile e veloce, riducendo i tempi necessari e i possibili errori.

L'interfacci, inoltre, utilizzerà il servizio Service Registry appena esposto per registrarsi senza necessariamente registrare anche un servizio offerto, così da poter essere identificato tramite id durante le operazioni di management.

3.3.7 Interazioni interne orchestrazione

In figura 3.5 uno schema delle interazioni durante un'orchestrazione. Si utilizzeranno due Application System generici definiti *consumatore* e *fornitore* così da concentrare l'attenzione sul funzionamento generale durante l'orchestrazione e non su un caso d'uso specifico.

In figura 3.6 uno schema delle iterazioni tra i core durante un'orchestrazione generica.

La parte di schema identificata come *check loop* viene eseguita per ogni servizio con cui il consumatore deve interagire secondo la lista interna dell'Orchestrator, una volta verificata tutta la lista, viene restituita al consumatore con anche tutti i dati ottenuti dal Service Registry. Come si può ben vedere le iterazioni necessarie per un'orchestrazione non sono molte, ma permettono una maggiore sicurezza nel sistema.

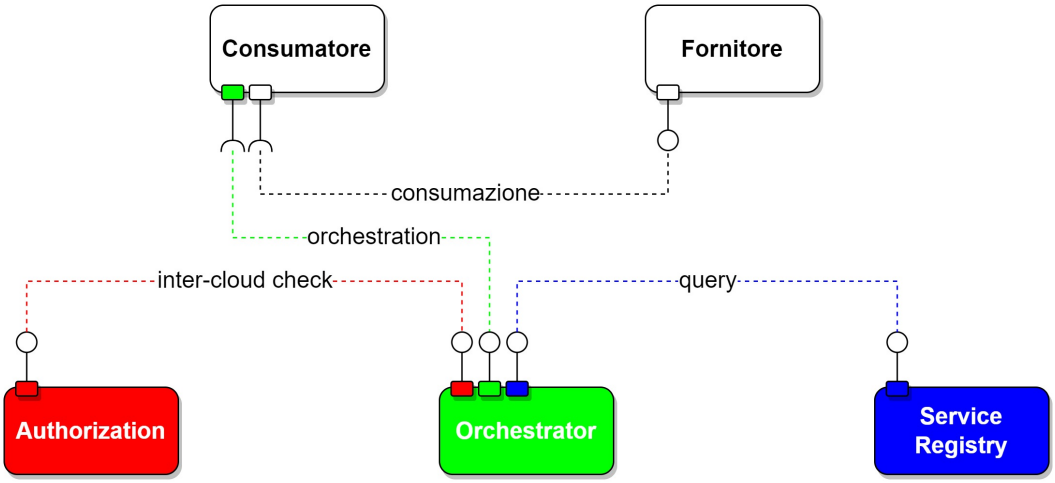

Figura 3.5: Schema Sistema - orchestrazione

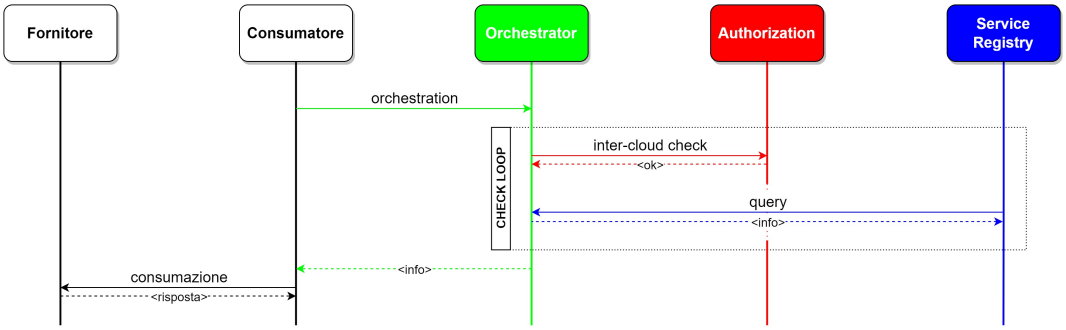

Figura 3.6: Schema Interazioni - orchestrazione

3.3.8 Interazioni

In figura 3.7 uno schema del sistema con le nuove iterazioni.

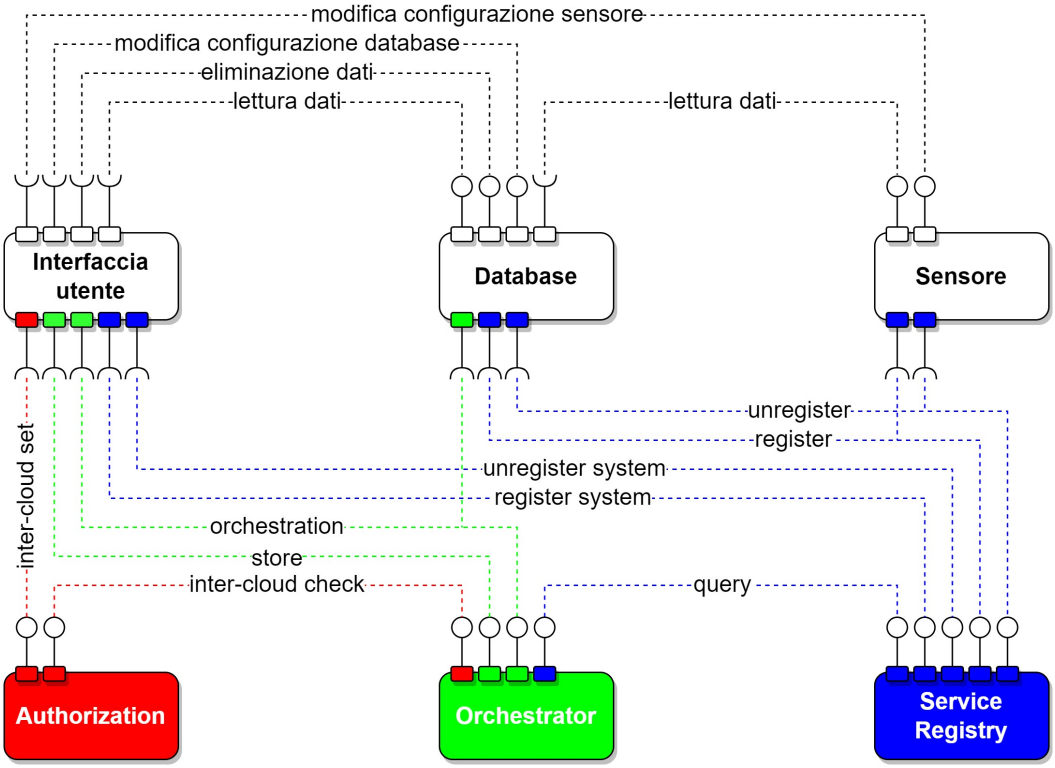

Figura 3.7: Schema Sistema - Livello 2

In figura 3.8 uno schema delle iterazioni tra gli elementi del sistema con l'aggiunta dell'Orchestrator. Le interazioni interne ad un'orchestrazione verranno omesse per una più semplice lettura dello schema, in quanto lo schema generico sopra esposto vale per ogni caso specifico. La parte di schema identificata come *management loop* viene eseguita 5 volte: per settare le regole di management tra database-sensore, interfaccia utente-sensore e interfaccia utente-database (x3)

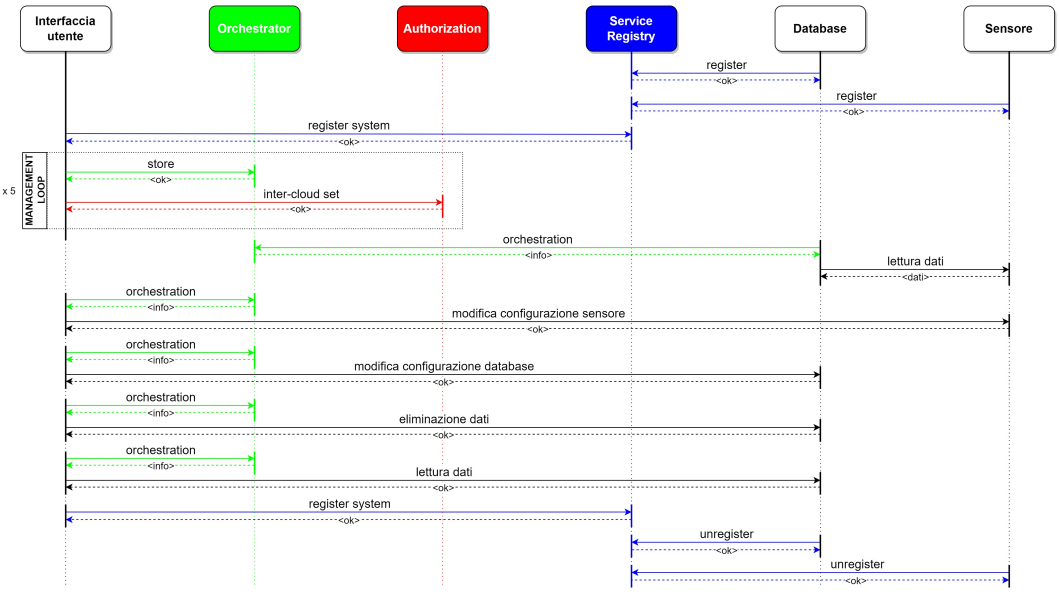

Figura 3.8: Schema Interazioni - Livello 2

3.4 Livello 3

In questo livello della demo toglieremo la libertà degli elementi di scegliere quando agire, sarà il Cloud Administrator a decidere quando ogni elemento dovrà compiere una specifica azione.

Come per i livelli precedenti, anche in questo caso, sembra un inutile complicazione l'aggiunta di questo nuovo core nella presente demo, ma come nei casi precedenti il suo utilizzo è utile per comprendere appieno il suo funzionamento, per poi poter essere applicato a realtà più complesse.

Questa scelta permette di ridurre considerevolmente la gestione separata degli elementi per evitare concorrenza e interazioni errate o scoordinate. Si occupa il Choreographer di gestire le operazioni degli elementi, eseguendo le azioni in ordine preciso e eseguendole solo se quelle precedenti sono state già eseguite con successo. Questo metodo di lavoro è molto utile perchè permette di avere un solo punto per la gestione del workflows, piuttosto che dover programmare ogni singolo elemento del sistema per rispettare il flusso di lavoro predefinito; questo permettere una più veloce e sicura modifica del workflow, soprattutto in realtà industriali dove esso cambia spesso.

Per far ciò verrà utilizzato il core Choreographer e gli elementi base della demo verranno ulteriormente modificati per aspettare un ordine del Choreographer prima di effettuare una determinata operazione.

3.4.1 Choreographer

Il Choreographer è un core Arrowhead che permette di eseguire flussi di lavoro predefiniti.

Il suo funzionamento è, a livello teorico, abbastanza semplice: il Cloud Administrator definisce un workflow e il Choreographer consuma, nell'ordine prestabilito e dopo aver ottenuto i dati necessari dall'Orchestrator, i servizi indicati; prima di eseguire un servizio aspetta, però, che i servizi precedenti abbiano notificato l'avvenuta conclusione.

Il workflow è suddiviso in *actions*, i quali a loro volta sono suddivisi in *steps*, che rappresentano un servizio da consumare. Così facendo si crea una sequenzializzazione di gruppi di servizi coerenti, aumentando la trasparenza del workflow.

Anche con questo core i servizi offerti sono molteplici, ma noi ne utilizzeremo solo 3:

- Notify step done, servizio usato dai servizi consumati per notificare al Choreographer che hanno finito di eseguire il loro compito
- Add plan, servizio di management utilizzato per creare un nuovo workflow
- Start, servizio di management utilizzato per eseguire un workflow identificato tramite ID

L'utilizzo di questo core causa una ristrutturazione di parte del funzionamento di ogni elemento del sistema. Analizziamo quindi i cambiamenti dei nostri elementi base del sistema.

3.4.2 Sensore

Il sensore deve perdere il suo potere decisionale, nel suo caso, non usufruendo di servizi forniti da altri, l'unica azione che effettua che ha un effettivo risultato sui dati che registra è la scelta di quando iniziare e finire di campionare (ogni quanto è già

deciso dall'interfaccia utente), fin d'ora iniziava a campionare appena si avvia e smetteva quando si spegne.

Per rendere il sensore logicamente inserito in un sistema gestito dal Choreographer bisogna quindi rimuovergli tale potere decisionale: all'avvio non inizia più a registrare i dati e ad immagazzinarli, si mette, invece, ad aspettare la richiesta del Choreographer per iniziare a leggere dati e continua a registrarli fin quando il Choreographer non gli dice di smettere.

Il sensore fornisce, quindi, 2 nuovi servizi:

- Avvio, servizio usato dal Choreographer per avviare la registrazione dei dati
- Termine, servizio usato dal Choreographer per terminare la registrazione dei dati

Come precedentemente spiegato nel Choreographer i servizi una volta terminate tutte le azioni che devono compiere devono notificare al Choreographer la fine; il sensore, quindi, una volta avviata/terminata la registrazione dei dati lo notificherà tramite il servizio Notify step done.

Per il resto il funzionamento rimane invariato.

3.4.3 Database

Il database deve perdere il suo potere decisionale, nel suo caso, l'unica azione attiva che compie è la scelta di quando campionare i dati dal sensore, fino ad ora iniziava a leggere i dati del sensore ogni *interval* da quando si avvia a quando si spegne.

Per rendere il database logicamente inserito in un sistema gestito dal Choreographer bisogna quindi rimuovergli tale potere decisionale: all'avvio non inizia il ciclo di lettura-attesa, ma si mette in attesa dell'autorizzazione del Choreographer per effettuare un singolo ciclo di lettura-attesa.

Il database fornisce, quindi, un nuovo servizio:

- Riempimento, servizio usato dal Choreographer per avviare la lettura da parte del database dei dati registrati dal sensore

Come per tutti i servizi avviati dal Choreographer anche in questo caso una volta terminato il singolo ciclo di lettura-attesa il database notificherà l'avventa fine del servizio tramite il Notify step done.

Il servizio di riempimento, a differenza dei nuovi servizi dal sensore, può essere utilizzato logicamente più volte all'interno di uno stesso piano.

Il resto del funzionamento rimane invariato.

3.4.4 Interfaccia utente

L'interfaccia utente è l'unica, dei 3 elementi base della demo, che non perde il suo potere decisionale, questo perchè non è l'interfaccia utente stessa a prendere le decisioni, ma l'utilizzatore; risulterebbe, quindi, controproducente rimuovere il potere dell'utente di decidere quando visualizzare i dati, eliminarli o modificare le configurazioni di sensore e database.

L'unica modifica che viene effettuata all'interfaccia utente in questo livello è l'aggiunta delle regole di management per la gestione corretta del Choreographer e l'inserimento di un'interfaccia per la configurazione e l'avvio del piano del Choreographer.

Il funzionamento rimane quindi invariato, se non per il consumo di due nuovi servizi: add plan e start forniti dal Choreographer.

3.4.5 Interazioni

In figura 3.9 uno schema del sistema con le nuove iterazioni.

In figura 3.10 uno schema delle iterazioni tra gli elementi del sistema con l'aggiunta dell'Orchestrator. Le interazioni interne ad un'orchestrazione verranno omesse per una più semplice lettura dello schema. La parte di schema identificata come *management loop* viene eseguita 5 volte: per settare le regole di management tra database-sensore, interfaccia utente-sensore e interfaccia utente-database (x3), mentre la parte di schema identificata come *fill loop* verrà ripetuta il numero di volte scelto dall'utente.

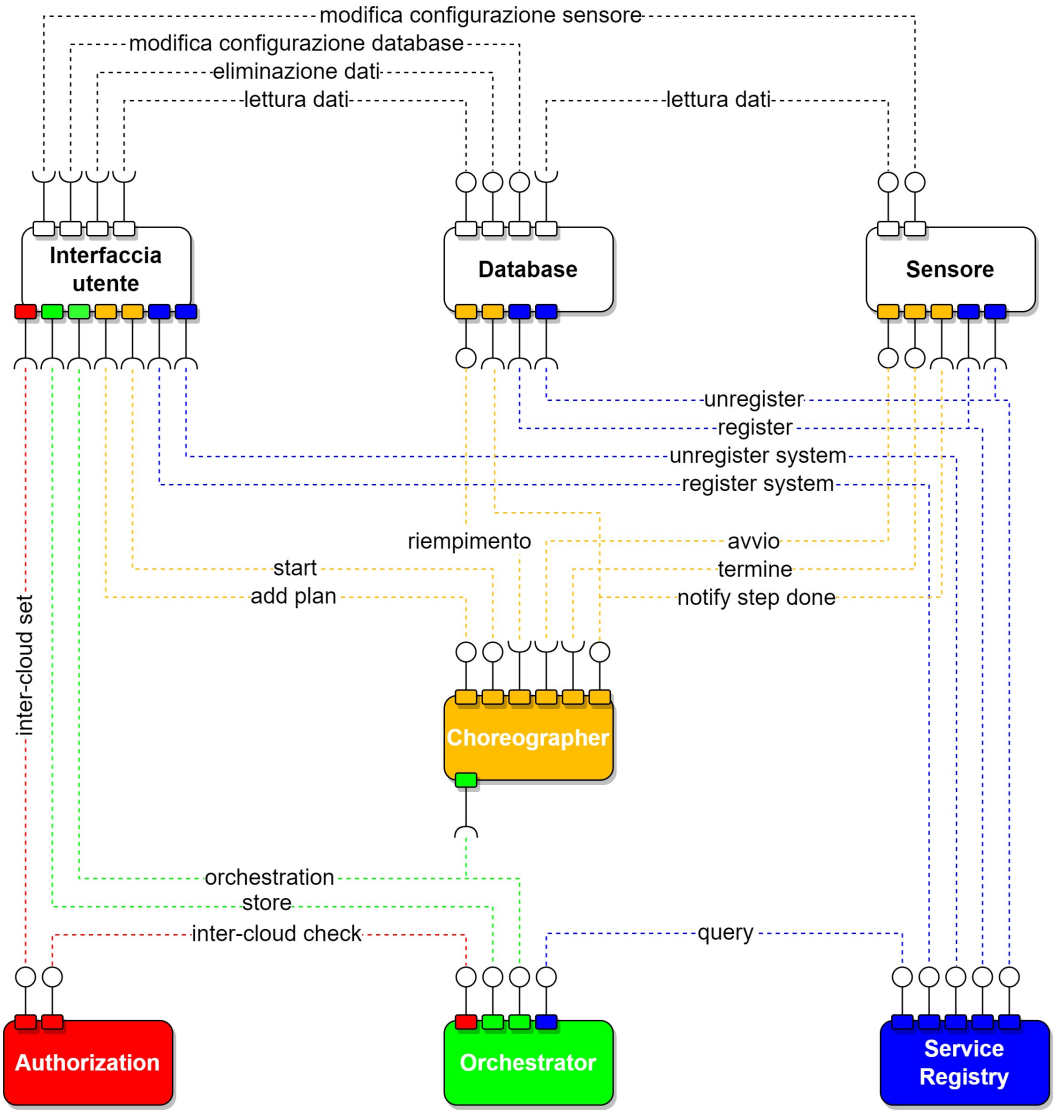

Figura 3.9: Schema Sistema - Livello 3

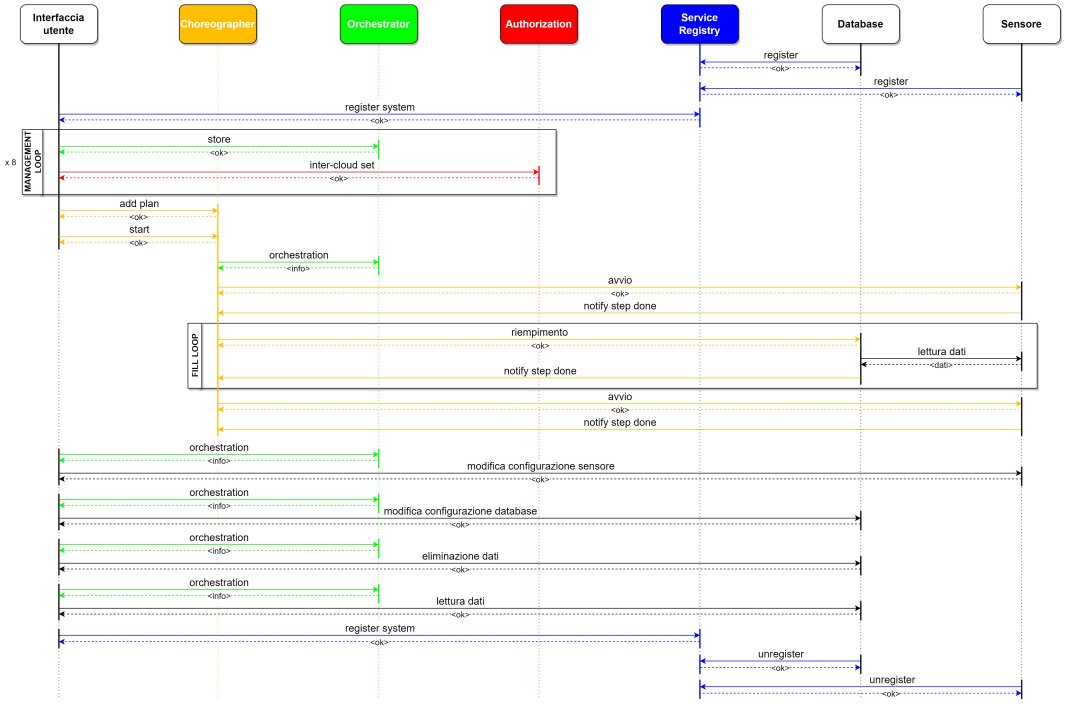

Figura 3.10: Schema Interazioni - Livello 3

Capitolo 4

Sviluppo

In questo capitolo analizzeremo le scelte implementative, vedremo alcuni esempi di codice e come l'architettura, precedentemente spiegata solo in linea teorica, è stata sviluppata nella realtà.

La prima scelta implementativa necessaria per lo sviluppo della demo è decidere il protocollo da utilizzare per la comunicazione tra gli elementi. La scelta risulta essere abbastanza semplice: il protocollo HTTP, esso è il protocollo più comune per le comunicazioni di rete ed è, infatti, anche quello utilizzato da Arrowhead, risulta quindi inutile utilizzare un altro protocollo, complicando ulteriormente le comunicazioni.

La seconda scelta implementativa necessaria è la scelta del formato per lo scambio dei dati, anche in questo caso la scelta risulta essere molto semplice: JSON, oltre ad essere uno dei formati standard più utilizzati nelle comunicazioni tramite HTTP, è quello utilizzato da Arrowhead per lo scambio di informazioni, permettendo così una maggior uniformità degli elementi all'interno del sistema.

Ora analizziamo di seguito tutte le scelte suddivise per livello in cui esse vengono prese.

4.1 Demo base - Livello 0

In questo livello dovremo implementare da 0 gli elementi base del sistema.

La prima cosa da scegliere è il linguaggio di programmazione per implementare i vari elementi del sistema.

Per il sensore e il database si è scelto di utilizzare Python, un linguaggio ad alto livello orientato agli oggetti con una sintassi molto semplice, simile ad uno pseudo codice, e famoso per il ricco assortimento di funzionalità base e moduli che permettono una facile lettura del codice anche da parte di principianti. Rispetto ad altri linguaggi, in Python la lettura di elementi JSON e la comunicazione tramite protocollo HTTP è molto più semplice e flessibile, così da poterci concentrare sull'implementazione degli elementi della demo e non nella gestione delle comunicazioni. Inoltre, in questo linguaggio è presente un micro-framework per la creazione semplificata di servizi accessibili tramite HTTP su specifici indirizzi e porte.

Per l'interfaccia utente, invece, si è scelto di creare un'applicazione per dispositivi mobili, questo perchè permette una maggior flessibilità di movimento da parte dell'utilizzatore; inoltre, i dispositivi mobili, come anche evidenziato nel primo capitolo, sono comunemente utilizzati per creare interfacce utente anche per la gestione di reti industriali, ci risultava quindi interessante fornirne un esempio. Tra le varie possibilità per creare un'applicazione mobile abbiamo deciso di crearne una per dispositivi Android utilizzando come linguaggio principale Java, questo per due motivi principali: il sistema Android è il sistema mobile attualmente più utilizzato e perchè Java è un linguaggio vastamente utilizzato e quindi facilmente capibile e modificabile anche da terzi.

Un'altra importante decisione da prendere per sviluppare la demo è scegliere il tipo di database in cui immagazzinare le informazioni raccolte. In questo caso si è scelto di utilizzare un database con MySQL server, questo perchè l'interazione risulta essere semplice grazie all'integrazione con Python.

Si è scelto, inoltre, di creare file esterni al codice con tutte le costanti necessarie per il collegamento di rete, così da essere facilmente modificabili senza cercare in tutto il codice.

Ora analizziamo più nel dettaglio le scelte applicative per ogni elemento base della demo.

4.1.1 Sensore

Il sensore deve eseguire due tipologie differenti di operazioni: campionare ogni tot i dati salvandoli in memoria e controllare se vi sono richieste per i servizi offerti. Questo, ovviamente, implica che il sensore debba avere due thread che lavorano in contemporanea.

La gestione a più thread potrebbe risultare difficoltosa, fortunatamente Python fornisce moduli appositi per semplificare la loro gestione: utilizzeremo nello specifico il modulo *threading* per la gestione del thread per il campionamento e il modulo *flask* per fornire i servizi tramite protocollo HTTP.

Per il campionamento si è deciso di creare una classe per raggruppare tutte le funzionalità necessarie per il corretto funzionamento del thread e renderlo maggiormente modificabile da terzi. Nella classe, nello specifico, sono state create funzioni per campionare i dati e salvali in una coda, ottenere le misurazioni eliminandole dalla coda, cambiare configurazione e gestire il thread. La configurazione interna standard è campionamento ogni 2 secondi con salvataggio in una coda di massimo 100 unità.

```
def measurement_thread(self):
  while(True):
    if self.stop_threads:
      break
    if((datetime.now() - self.last_measurement_time).total_seconds() >self.
        config['sampling_time']):
      logging.debug("measuring " + str(datetime.now()))
      while(len(self.buffer.queue) >= self.config["buffer_length"]):
        self.buffer.get()
      self.buffer.put(self.generate_measurement())

def start_measurements(self):
  self.x = threading.Thread(target=self.measurement_thread, daemon=True)
  self.stop_threads=False
  self.x.start()

def stop_measurements(self):
```

```
17  self.stop_threads=True
```

Listing 4.1: codice gestione thread

La parte più interessante in questa classe è la gestione del thread, in 4.1 il codice inerente. Come possiamo vedere viene eseguita in un differente thread la funzione measurement_thread; la durata del thread dipende, quindi, dalla durata di questa funzione: per fare terminare il ciclo infinito di cui è composta, usiamo semplicemente un controllo su un termine di uscita, che quando vero, farà terminare il ciclo.

Per il thread dei servizi offerti, invece, si è scelto di inserire le funzioni per la gestione di questi direttamente nel flusso principale del codice; i servizi vengono offerti tramite protocollo HTTP sull'indirizzo IP del dispositivo e sulla porta 5000 e vengono offerti con la seguente interfaccia:

- Fornire i dati - URL: /batterySensor-getData, metodo: GET - input: null, output: lista JSON dei dati raccolti, composti dai campi "plugged", "percent" e "time"
- Modificare la configurazione interna - URL: /batterySensor-setConfig, metodo: POST - input: JSON con campi "sampling_time" e "buffer_length", output: stringa di ok o d'errore

```
1  app = Flask(__name__)
2
3  @app.route('/'+SENSOR_GETDATA, methods=['GET'])
4  def temperature_measurement():
5    measurements = jsonify(SENSOR.get_measurements())
6    return measurements
7
8  atexit.register(shutdown)
9  app.run(host=SENSOR_IP_ADDRESS, port=SENSOR_PORT)
```

Listing 4.2: codice fornitore servizi

Nel riquadro 4.2 è stato inserito il codice utilizzato per fornire servizi. Per prima cosa bisogna inizializzare la variabile inerente alla gestione di flask, poi nella riga 3 vediamo come settare flask per offrire un servizio: dobbiamo indicare in ordine l'URL e il metodo

del nostro servizio e poi definire la funzione da eseguire quando viene richiamato; una volta definiti tutti i nostri servizi possiamo lanciare il tutto con il comando *run* indicando rispettivamente indirizzo IP e porta.

Oltre alle funzioni per la gestione dei servizi offerti si è creata anche la funzione *shutdown* per fermare il thread per il campionamento quando si chiude il thread per l'offerta dei servizi. Nella riga 8 il codice per far eseguire tale funzione in fase di chiusura.

4.1.2 Database

Il database deve anch'esso eseguire due differenti tipologie di operazioni: leggere i dati campionati dal sensore, salvandoli nel database MySQL, e controllare se vi sono richieste per i servizi offerti. Anche in questo caso utilizzeremo due thread, nello specifico utilizzeremo il modulo *threading* per la gestione del thread per la lettura dei dati e il modulo *flask* per fornire i servizi tramite protocollo HTTP.

A differenza del sensore qui i thread sono gestiti tutti all'interno del flusso principale del codice, questo per evidenziare un'altra possibile implementazione dei due flussi di lavoro, questa scelta tende ad essere più confusionaria della precedente in quanto il codice dei due thread tende a mescolarsi, non essendo più fortemente suddiviso in blocchi obbligatoriamente separati, ma non compromette l'efficacia del codice stesso.

Il funzionamento dei moduli *threading* e *flask* è già stato illustrato tramite gli esempi di codice nella sezione precedente, analizzeremo, quindi, solo alcune parti del nuovo codice; nello specifico il 4.3 illustra nella riga 2 la consumazione di un servizio tramite HTTP e dalla 3 in poi illustra il funzionamento del modulo per interagire con un server MySQL, in questo caso specifico per immagazzinare delle informazioni. Come si può facilmente notare la gestione della comunicazione con il server MySQL risulta più lunga rispetto al resto del codice esaminato fino ad ora, questo perchè i due moduli usati nel codice precedente sono moduli che fornisco funzioni molto "forti" che fanno molte cose nascondendole al programmatore che le usa. Per comunicare con il server MySQL bisogna per prima cosa aprire la connessione fornendo l'indirizzo IP, le credenziali di accesso e il nome del database a cui si vuole accedere, poi creare il codice SQL per inserire i dati, inviarlo e, se tutto è andato come previsto, chiudere la connessione.

```
1 def saveData(serviceURL):
```

```
response = requests.get(serviceURL)

try:
  connection = MySQL.connector.connect(
    host="localhost",
    user="root",
    password="database",
    database="tesi"
  )

  cursor = connection.cursor()

  sql = "INSERT INTO sensordata (time, plugged, percent) VALUES (%s, %s, %s)"
  val = []

  for datapoint in json.loads(response.text):
    plugged = bool(datapoint['plugged'])
    percent = str(datapoint['percent'])
    time = str(datapoint['time'])

    val.append((time, plugged, percent))

  cursor.executemany(sql, val)
  connection.commit()
  print("Record inserted successfully")

except MySQL.connector.Error as error:
  print("Failed to insert into MySQL table {}".format(error))

finally:
  if connection.is_connected():
    cursor.close()
    connection.close()
```

```
35      print("MySQL connection is closed")
```

Listing 4.3: codice salvataggio dati MySQL

I servizi vengono offerti tramite protocollo HTTP sull'indirizzo IP del dispositivo e sulla porta 5001 e vengono offerti con la seguente interfaccia:

- Fornire i dati - URL: /database-getData, metodo: GET - input: null, output: lista JSON dei dati raccolti, composti dai campi "plugged", "percent" e "time"
- Eliminare i dati - URL: /database-deleteData, metodo: POST - input: null, output: risultato modifica MySQL
- Modificare configurazione interna - URL: /database-setConfig, metodo: POST - input: JSON con campi "sampling_time" e "buffer_length", output: stringa di ok o d'errore

4.1.3 Interfaccia utente

L'interfaccia utente è l'elemento della demo logicamente più complesso da implementare, questo perchè in fase di sviluppo bisogna tenere in considerazione le caratteristiche che una buona applicazione Android deve avere per essere considerata tale.

Per prima cosa bisogna individuare le funzionalità principali che la nostra applicazione deve avere: una parte deve permettere la visualizzazione dei dati ed, eventualmente l'eliminazione, e poi deve essere presente una seconda sezione in cui si possono modificare le configurazioni del database e del sensore. Queste due attività essendo logicamente differenti verranno suddivise in due differenti Activity[1] : la parte di visualizzazione ed eliminazione dati sarà l'Activity principale (Main), mentre la parte di modifica delle configurazioni sarà gestita da un'Activity secondaria chiamata Settings.

Definiti i ruoli delle varie Activity bisogna ora decidere l'interfaccia grafica da adottare, che deve essere al contempo semplice, ma completa. In 4.1 le interfacce grafiche scelte.

[1]finestra che contiene un'interfaccia utente di un'applicazione con lo scopo di permettere un'interazione con l'utente

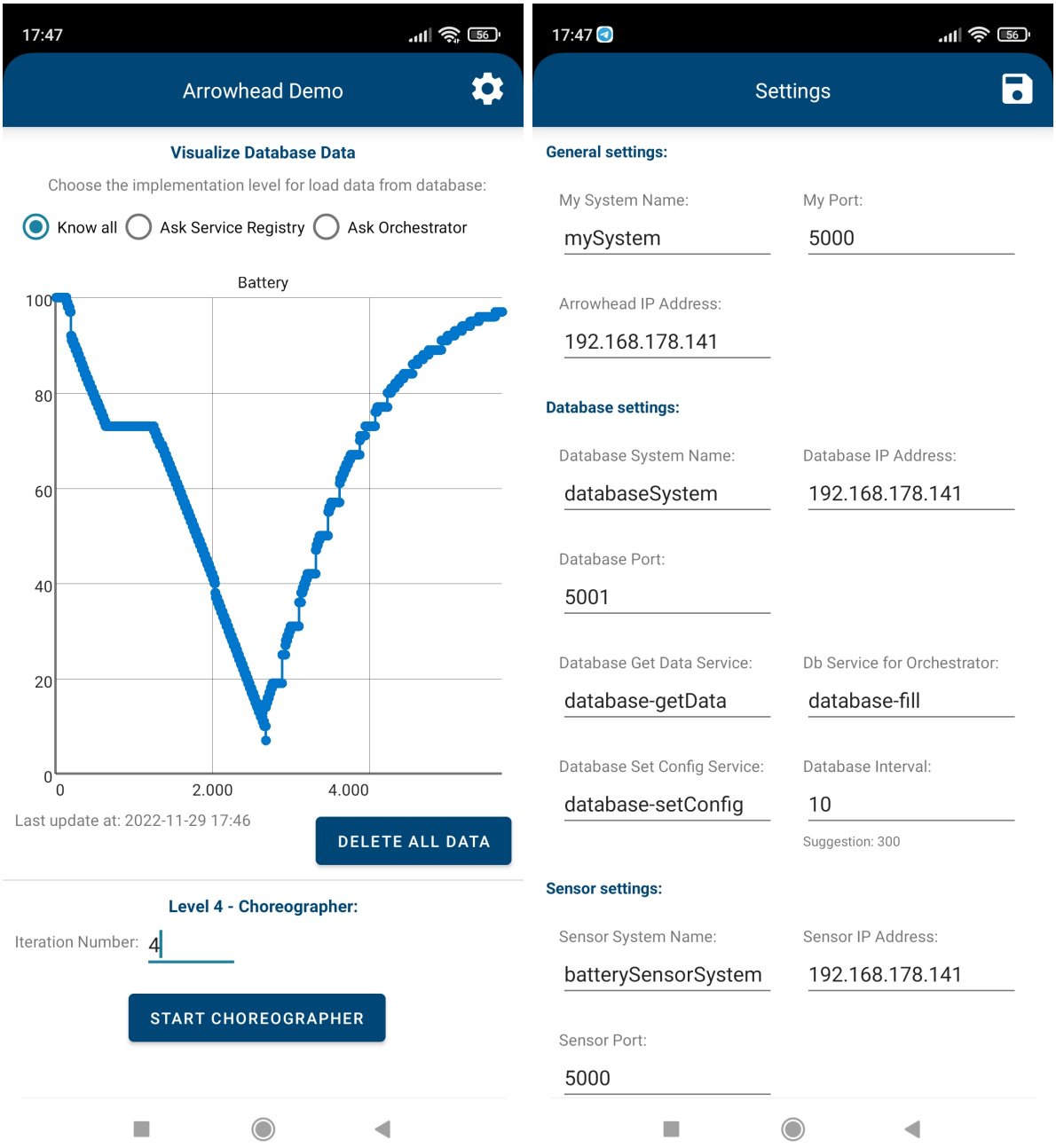

Figura 4.1: Interfacce grafiche

Come si può ben vedere vi sono già presenti elementi per la gestione anche dei livelli successivi, questo perchè la modifica dell'interfaccia grafica ad ogni livello avrebbe aggiunto una macchinosità inutile; il nostro obbiettivo è concentrarci sul funzionamento del sistema e non sulla veste grafica, ne facciamo qui un piccolo excursus per completezza. L'interfaccia è stata sviluppata seguendo il più possibile le regole di Material Design, come colore si è scelto il blu tipico di Arrowhead, come logo il logo di Arrowhead e per le icone per il passaggio tra un'Activity quelle standard fornite da Google Fonts; inoltre, i dati vengono visualizzati in un grafico per una loro maggiore comprensione da parte dell'utilizzatore.

All'interno della Settings, oltre alla modifica dei valori per la configurazione interna del database e del sensore, è possibile modificare tutte le informazioni necessarie per la comunicazione con gli altri elementi della rete: indirizzi IP, porte e nomi dei servizi offerti, così da permettere all'utente di modificarli direttamente dall'interfaccia grafica, qual ora vi fosse necessità, senza dover modificare il codice dell'interfaccia utente. Per una maggiore praticità e per salvare anche tra una sessione e l'altra le impostazioni scelte si è deciso di salvare tutte le informazioni all'interno delle SharedPreferences[2].

Al caricamento della MainActivity l'interfaccia utente consuma subito il servizio del database per la visualizzazione dei dati tramite grafico, così da rendere visibile i dati fin da subito all'utente. L'aggiornamento del grafico non avverrà in automatico, ma solo quando deciso dall'utente. Per maggiore praticità sotto al grafico è presente un riquadro che indica l'ultima volta che si è aggiornato il grafico e il tasto per l'eliminazione dei dati.

L'implementazione della comunicazione tramite protocollo HTTP in Java risulta più macchinosa rispetto a quella in Python (per lo stesso risultato avremmo impiegato solo un paio di righe di codice).

In 4.4 un esempio, nello specifico l'aggiornamento della configurazione interna del database. Come si può notare bisogna gestire la richiesta, che risulta essere un task asincrono, con un'apposita classe per la gestione del task al di fuori del flusso normale dell'Activity; anche l'aggiunta del body della richiesta HTTP risulta essere più complessa,

[2]oggetto che ha lo scopo di memorizzare informazioni utili per il funzionamento delle applicazioni sottoforma di coppie nome-valore

dovendosi appoggiare ad apposite classi per la bufferizzazione del flusso in output.

```
1  private class updateIntervalDatabase extends AsyncTask<String, Void, String> {
2
3    @Override
4    protected String doInBackground(String... params) {
5      String urlWebService = params[0];
6      HttpURLConnection urlConnection = null;
7      try {
8        int interval = pref.getInt(DATABASE_INTERVAL, 10);
9        String query = "{\"interval\": "+interval+"}";
10
11       URL url = new URL("http://"+urlWebService);
12
13       urlConnection = (HttpURLConnection) url.openConnection();
14       urlConnection.setRequestMethod("POST");
15
16       urlConnection.setRequestProperty("Content-Type", "application/json");
17       urlConnection.setRequestProperty("Accept", "application/json") ;
18       urlConnection.setDoInput(true);
19       urlConnection.setDoOutput(true);
20
21       BufferedOutputStream out = new BufferedOutputStream(urlConnection.
             getOutputStream());
22       BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(out,
             StandardCharsets.UTF_8));
23       writer.write(query);
24       writer.flush();
25       writer.close();
26       out.close();
27
28       urlConnection.connect();
29
30       if (urlConnection.getResponseCode() == 200){
```

```
31          InputStream inputStream = urlConnection.getInputStream();
32          InputStreamReader inputStreamReader = new InputStreamReader(inputStream)
                ;
33          BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
34          String response = bufferedReader.readLine();
35
36          if (response.contains("Config Update")) {
37            return "OK";
38          }
39      ...
40      } finally {
41        if (urlConnection!=null){
42          urlConnection.disconnect();
43        }
44      }
45    }
46
47    @Override
48    protected void onPostExecute(String s) {
49      super.onPostExecute(s);
50      if (Objects.equals(s, "error") || s==null){
51        new AlertDialog.Builder(MainActivity.this)
52            .setTitle("ERROR")
53      }
54      else{
55        Toast.makeText(MainActivity.this, "Database Interval update with success",
              Toast.LENGTH_LONG).show();
56      }
57    }
58  }
```

Listing 4.4: codice aggiornamento configurazione database

4.2 Livello 1

In questo livello le scelte implementative non sono molte. La principale modifica è l'aggiunta di nuove funzioni per aggiungere le nuove iterazioni del livello.

Tra le poche scelte implementative a questo livello c'è la decisione di permettere all'utente di selezionare il livello a cui far girare la demo, si è quindi inserito un campo all'avvio del programma del database e del sensore e una scelta multipla nell'interfaccia utente.

Un'altra importante scelta implementativa di questo livello è la creazione di una classe esterna per la gestione delle funzioni per comunicare con i core Arrowhead, se ne è creata una per Pyhton e una per Java con le funzioni di cui gli elementi sviluppati in quel linguaggio necessitavano. Questa scelta permette una maggiore semplicità di lettura del codice e una sua più facile modifica in caso di cambio dei metodi di comunicazione con i core.

4.2.1 Arrowhead

L'utilizzo di Arrowhead risulta essere, per ovvi motivi, una scelta implementativa scontata. Faremo qui solo un piccolo inciso sulla versione utilizzata nella demo, sull'istallazione e sulle risorse di cui necessita per completezza.

In questa demo si userà la versione 4.5.0 uscita a fine giugno 2022. Essa non risulta essere attualmente l'ultima versione, in quanto durante la redazione di questa tesi è stata pubblicata quella che, ad oggi, risulta essere l'ultima versione. Le due versioni risultano essere per lo più compatibili, rendendo quanto detto in questa tesi corretto a livello teorico anche per la nuova versione. La differenza principiale tra le due versioni è l'aggiunta di nuove funzionalità al Choreographer che permettono un maggior controllo sul core; potrebbe essere necessario, quindi, modificare alcune parti di codice inerenti a questo core per sfruttare al meglio i nuovi servizi offerti.

L'installazione della versione 4.5.0 di Arrowhead risulta essere difficoltosa, non perchè non vi sono spiegazioni su come effettuarla, ma bensì per la presenza di codice "sporco" rimasto dalle versioni precedenti e che causa errori in fase di esecuzione: per il corretto funzionamento bisogna rimuovere le righe 6 e 40 in *scripts/gams_privileges.sql*.

Una volta modificato il codice dobbiamo verificare di avere i programmi corretti: Java JDK 11 (unica versione supportata, nel mio caso ho dovuto fare il downgrade con il conseguente problema di compatibilità con programmi che richiedevano JDK con versione minima maggiore), Maven 3.5 o superiore e MySQL server 5.7 o superiore.

Una volta compilato tutto il codice seguendo le indicazioni fornite da Arrowhead il framework risulterà funzionante su HTTPS (versione sicura con autenticazione tramite certificati di HTTP), bisogna, quindi, modificare il valore della variabile *server.ssl.enabled* in *false* in ogni file application.properties per ogni singolo core.

L'avvio dei core risulta essere la parte più delicata del processo, non in quanto difficoltosa, ma perchè richiede un mucchio di risorse: in caso si avviino tutti i core utilizza in modo intensivo la CPU in fase di avvio, il quale richiede svariati minuti, e utilizza costantemente circa 4,5GB di RAM. La necessità di tutte queste risorse, purtroppo, limita molto i dispositivi su cui Arrowhead può funzionare correttamente.

4.2.2 Service Registry

I servizi vengono offerti sull'indirizzo IP del dispositivo su cui è stato istallato e sulla porta 8443:

- Query - URL: /serviceregistry/query, metodo: POST - input: JSON con il campo "serviceDefinitionRequirement" con il nome del servizio richiesto, output: lista JSON degli n servizi corrispondenti nello schema 4.5

```
{
  "serviceQueryData": [
    {
      "id": 0,
      "serviceDefinition": {
        "id": 0,
        "serviceDefinition": "string",
        "createdAt": "string",
        "updatedAt": "string"
      },
      "provider": {
```

```
      "id": 0,
      "systemName": "string",
      "address": "string",
      "port": 0,
      "createdAt": "string",
      "updatedAt": "string"
    },
    "serviceUri": "string",
    "secure": "NOT_SECURE",
    "version": 0,
    "interfaces": [
      {
        "id": 2,
        "interfaceName": "HTTP-INSECURE-JSON",
        "createdAt": "string",
        "updatedAt": "string"
      }
    ],
    "createdAt": "string",
    "updatedAt": "string"
  },
  ...
 ],
 "unfilteredHits": n
}
```

Listing 4.5: codice JSON degli n servizi

- Register - URL: /serviceregistry/register, metodo: POST - input: JSON nello schema 4.6, output: JSON del servizio registrato (stesso schema ma singolo e non inserito in un array di 4.5)

```
{
 "interfaces": [
  "HTTP-INSECURE-JSON"
```

```
  ],
  "providerSystem": {
    "address": "string",
    "port": 0,
    "systemName": "string"
  },
  "serviceDefinition": "string",
  "serviceUri": "string"
}
```

Listing 4.6: codice JSON registrazione servizio

- Unregister - URL: /serviceregistry/unregister, metodo: DELETE - input: JSON nello schema 4.7, output: codice di risposta

```
{
  "address": "string",
  "port": 0,
  "service_definition": "string",
  "service_uri": "string",
  "system_name": "string"
}
```

Listing 4.7: codice JSON eliminazione servizio

4.2.3 Sensore

Nessuna scelta implementativa specifica oltre a quelle già elencate.

4.2.4 Database

Nessuna scelta implementativa specifica oltre a quelle già elencate.

4.2.5 Interfaccia utente

In questo livello implementativo l'unica scelta è quella inerente all'esecuzione di una determinata azione in caso di risposta positiva da parte di un servizio utilizzato. Come abbiamo già visto nel livello precedente la comunicazione HTTP in Java avviene in maniera asincrona, ciò implica che, una volta ottenuto il risultato della comunicazione, bisogna trovare un metodo per far eseguire una determinata azione al flusso normale dell'Activity. Nello specifico utilizzeremo gli intent broadcast per passare i dati ottenuti dalla fine dell'uso del servizio o per segnalare la loro conclusione.

```
IntentFilter filter = new IntentFilter();
filter.addAction(intentType);
this.registerReceiver(new Receiver(), filter);
```

Listing 4.8: codice fornitore servizi

Nel riquadro 4.8 il codice per filtrare gli intent broadcast ricevuti e definire un'azione da eseguire quando si riceve un determinato tipo di intent.

4.3 Livello 2

In questo livello non vi è alcuna scelta implementativa generale da portare all'attenzione. Questo livello risulta essere molto simile, a livello di codice e tecniche risolutive utilizzate, al precedente.

4.3.1 Authorization

I servizi vengono offerti sull'indirizzo IP del dispositivo su cui è stato istallato e sulla porta 8445:

- Intra-cloud check - URL: /authorization/mgmt/intracloud, metodo: GET - input: null, output: lista JSON delle n regole intra-cloud 4.9

```
{
  "data": [
    {
```

```
    "id": 0,
    "consumerSystem": {
      "id": 0,
      "systemName": "string",
      "address": "string",
      "port": 0,
      "createdAt": "string",
      "updatedAt": "string"
    },
    "providerSystem": {
      "id": 0,
      "systemName": "string",
      "address": "string",
      "port": 0,
      "createdAt": "string",
      "updatedAt": "string"
    },
    "serviceDefinition": {
      "id": 0,
      "serviceDefinition": "string",
      "createdAt": "string",
      "updatedAt": "string"
    },
    "interfaces": [
      {
        "id": 2,
        "interfaceName": "HTTP-INSECURE-JSON",
        "createdAt": "string",
        "updatedAt": "string"
      }
    ],
    "createdAt": "string",
    "updatedAt": "string"
```

```
    },
    ...
  ],
  "count": n
}
```

Listing 4.9: codice JSON regole intra-cloud

- Intra-cloud set - URL: /authorization/mgmt/intracloud, metodo: POST - input: JSON nello schema 4.10, output: codice di risposta

```
{
  "consumerId": "string",
  "interfaceIds": [2],
  "providerIds": ["string"],
  "serviceDefinitionIds": ["string"]
}
```

Listing 4.10: codice JSON creazione regola intra-cloud

4.3.2 Orchestrator

I servizi vengono offerti sull'indirizzo IP del dispositivo su cui è stato istallato e sulla porta 8441:

- Orchestration - URL: /orchestrator/orchestration, metodo: POST - input: JSON nello schema 4.11, output: codice di risposta

```
{
  "requesterSystem": {
    "address": "string",
    "metadata":{},
    "port": 0,
    "systemName": "string"
  }
}
```

Listing 4.11: codice JSON richiesta orchestrazione

- Store - URL: /orchestrator/mgmt/store, metodo: POST - input: JSON nello schema 4.12, output: codice di risposta

```
[
  {
    "attribute": {},
    "cloud": null,
    "consumerSystemId": 0,
    "priority": 1,
    "providerSystem": {
      "systemName": "string",
      "address": "string",
      "port": 0
    },
    "serviceDefinitionName": "string",
    "serviceInterfaceName": "HTTP-INSECURE-JSON"
  }
]
```

Listing 4.12: codice JSON creazione regola orchestrazione

4.3.3 Service Registry

Oltre ai servizi del SR sopra esposti utilizzeremo, in questo livello, anche questi servizi:

- Register system - URL: /serviceregistry/register-system, metodo: POST - input: JSON con indirizzo IP, porta e nome del sistema, output: JSON nello schema 4.13

```
{
  "address": "string",
  "authenticationInfo": "string",
  "createdAt": "string",
  "id": 0,
  "metadata": {},
  "port": 0,
```

```
  "systemName": "string",
  "updatedAt": "string"
}
```

Listing 4.13: codice JSON ritorno registrazione sistema

- Unregister system - URL: /serviceregistry/unregister-system, metodo: DELETE - input: JSON con indirizzo IP, porta e nome del sistema, output: codice di risposta

4.3.4 Sensore

Nessuna scelta implementativa specifica.

4.3.5 Database

Nessuna scelta implementativa specifica.

4.3.6 Interfaccia utente

In questo livello non vi sono grandi scelte implementative, l'unica aggiunta importante di codice è l'inserimento del consumo dei servizi di management; di questa l'unica scelta implementativa interessante è l'utilizzo del servizio *query* del SR per trovare gli ID di ogni sistema e servizio di cui bisogna definire le regole di management.

4.4 Livello 3

In questo livello non vi sono scelte implementative generali degne di nota. L'implementazione rispecchia e sfrutta le scelte effettuate ai livelli precedenti.

4.4.1 Choreographer

I servizi vengono offerti sull'indirizzo IP del dispositivo su cui è stato istallato e sulla porta 8457:

- Notify step done - URL: /choreographer/notifyStepDone, metodo: POST - input: JSON con "runningStepId" e "sessionId", output: codice di risposta
- Add plan - URL: /choreographer/mgmt/plan, metodo: POST - input: JSON nello schema 4.14, output: codice di risposta

```
{
  "actions": [
    {
      "firstStepNames": ["string"],
      "name": "string",
      "nextActionName": "string",
      "steps": [
        {
          "metadata": null,
          "name": "string",
          "nextStepNames": ["string"],
          "parameters": null,
          "quantity": 0,
          "serviceName": "string"
        },
    ...
      ]
    },
    ...
  ],
  "firstActionName": "string",
  "name": "string"
}
```

Listing 4.14: codice JSON creazione piano

- Start - URL: /choreographer/mgmt/session/start, metodo: POST - input: JSON con ID del piano da avviare, output: codice di risposta

Oltre a quelli elencati sopra, utilizzeremo anche il seguente servizio:

- Get plans - URL: /choreographer/mgmt/plan, metodo: GET, restituisce la lista di tutti i piani presenti all'interno del Choreographer - input: null, output: JSON nello schema 4.15

```
{
  "actions": [
    {
      "firstStepNames": ["string"],
      "name": "string",
      "nextActionName": "string",
      "steps": [
        {
          "metadata": null,
          "name": "string",
          "nextStepNames": ["string"],
          "parameters": null,
          "quantity": 0,
          "serviceName": "string"
        },
    ...
      ]
    },
    ...
  ],
  "firstActionName": "string",
  "name": "string"
}
```

Listing 4.15: codice JSON piani Choreographer

4.4.2 Sensore

Il sensore offre a questo livello, sempre all'indirizzo IP del dispositivo sulla porta 5000, anche i seguenti servizi con interfaccia:

- Avvio - URL: /batterySensor-startData, metodo: POST - input: JSON con "sessionId" e "runningStepId", output: null
- Termine - URL: /batterySensor-stopData, metodo: POST - input: JSON con "sessionId" e "runningStepId", output: null

4.4.3 Database

Il Database offre a questo livello, sempre all'indirizzo IP del dispositivo sulla porta 5001, anche il seguente servizio con interfaccia:

- Riempimento - URL: /database-fill, metodo: POST - input: JSON con "sessionId" e "runningStepId", output: null

4.4.4 Interfaccia utente

In questo livello l'interfaccia utente ha integrato la gestione dell'avvio del Choreographer. L'unico problema incontrato nella sua implementazione è che il Choreographer, una volta registrato il piano, non restituisce il suo ID; per avviare il piano, quindi, bisogna utilizzare il servizio Get plans e cercare nella lista restituita il piano appena registrato e il suo ID.

Capitolo 5

Conclusioni

Arrowhead, come abbiamo visto, fornisce un framework per la creazione di un sistema industriale sicuro e integrabile nell'ottica dell'industria 4.0.

Esso è composto da core che forniscono servizi pensati appositamente per il controllo sicuro e interoperabile della rete industriale. I core presi in analisi in questa Demo sono i tre principali: Service Registry, Authorization e Orchestrator, più un core facoltativo, il Choreographer, interessante per le funzionalità non comuni da lui offerte.

La demo evidenzia le potenzialità del framework nella creazione di sistemi industriali sicuri e flessibili. I core Arrowhead hanno permesso alla nostra demo di passare dalla versione base, in cui ogni elemento deve essere a conoscenza di tutto il sistema che lo circonda, con grandi difficoltà in caso di modifica di uno o più elementi del sistema, a un sistema composto da elementi passivi, che si affidano al framework per ogni cosa, compreso la scelta di quando effettuare determinate azioni, creando così un sistema flessibile e aperto a futuri cambiamenti, senza necessitare di grandi modifiche da parte del Cloud Administrator.

Il framework Arrowhead ha una grande flessibilità e fornisce la maggior parte delle funzionalità richieste per un framework per la gestione di sistemi industriali, anche qualora si necessitasse di integrare sistemi embedded o legacy systems. Oltre a quanto usato in questa demo sono presenti alcuni tools per l'installazione del framework e la creazione di certificati per il sistema Arrowhead, che rendono maggiormente completo e semplice il suo utilizzo e la sua integrazione in sistemi già esistenti.

Il framework però, oltre ad avere molte buone qualità e potenzialità, ha alcuni difetti che, purtroppo, rendono difficoltoso il suo utilizzo:

- alcuni core tendono ad avere malfunzionamenti o a non avviarsi proprio, questo non capita molto spesso, ma capita e in caso di sistemi industriali complessi questo potrebbe causare problemi; l'ultima versione, però, promette la risoluzione dei problemi della versione precedente; quindi, questo problema può essere stato potenzialmente risolto;

- il framework consuma molte risorse rendendolo difficilmente utilizzabile se il dispositivo su cui lo si vuole attivare non ha grandi prestazioni; purtroppo, il codice non ha un uso ottimizzato, come ci si aspetterebbe, delle risorse che usa, limitandone molto le possibilità di utilizzo;

- il codice è difficilmente trovabile se si conosce solo il suo nome: il framework è scaricabile solo da github, ma se ricercato tramite google il link ad esso o non viene visualizzato o è difficilmente trovabile. Anche nel sito web ufficiale Arrowhead il link per il download è difficilmente individuabile;

- l'installazione è difficoltosa, non perchè richieda competenze specifiche, ma perchè vi sono alcune righe di codice rimaste dalle versioni precedenti e che causano errori in fase di compilazione; inoltre, il framework gira solo ed esclusivamente con Java 11, il che potrebbe creare problemi di compatibilità se sul dispositivo sono presenti altri programmi che necessitano di una versione più recente;

- la documentazione del framework è poca, per alcuni core insufficiente o incompleta e, ad eccezione di quella presente su github, difficilmente trovabile anche sul sito web ufficiale Arrowhead; inoltre, la documentazione purtroppo non è aggiornata all'ultima versione, rendendo, quindi, difficile l'utilizzo e l'upgrade alle nuove versioni;

- i tools aggiuntivi sono introvabili, anch'io sono riuscita ad accedevi solo dopo che uno degli sviluppatori Arrowhead mi ha passato il link;

- non vi è un vero e proprio forum per lo scambio di informazioni tra gli utilizzatori; vi è un canale su Slack per domande e chiarimenti, ma non è citato da nessuna parte.

Il framework Arrowhead risulta, quindi, avere molto potenziale, ma possiede alcuni difetti che, ora come ora, non gli permettono di essere effettivamente competitivo nella sua fetta di mercato. La maggior parte dei problemi sono facilmente risolvibili con una maggiore attenzione ai dettagli, se risolti, quindi, potrebbero permettere ad Arrowhead di diventare, in un breve futuro, un valido strumento per la gestione di sistemi industriali.

Bibliografia

[1] Madakam S., Ramaswamy R. e Tripathi S., Internet of Things (IoT): A Literature Review, Scientific Research Publishing - Maggio 2015

[2] Pradyumna G., Bhat O. e Bhat S., Introduction to IOT, Research Gate - Gennaio 2018

[3] Trautman L. J. e Hussein M. T., The Internet of Things (IoT) in a Post-Pandemic World, SSRN - Giugno 2022

[4] Wegner P., Global IoT market size grew 22% in 2021, iot-analytics - Marzo 2022

[5] Hasen M., State of IoT 2022: Number of Connected IoT Devices Growing 18% to 14.4 billion Globally, iot-analytics - Maggio 2022

[6] Balabio B. e Orlando P., Cresce il mercato Internet of Things in Italia (+22%), superando i livelli pre-pandemia, osservatorio Politecnico Milano - Aprile 2022

[7] Wollschlaeger M., Sauter T. e J. Jasperneite, The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0, IEEE Industrial Electronics Magazine - Marzo 2017

[8] Aldalur I., Illarramendi M., Larrinaga F., Perez T., SÃ¡enz F., Unamuno G. e Lazkanoiturburu I., Advantages of Arrowhead Framework for the Machine Tooling Industry, IECON 2020 - Ottobre 2020

[9] Varga P., Blomstedt F., Ferreira L. L., Eliasson J., Johansson M., Delsing J. e Martìnez de Soria I., Making system of systems interoperable - The core components of

the arrowhead framework, Journal of Network and Computer Applications - Agosto 2016

[10] Kulcsàr G., Tatara M. S. e Montori F., Toolchain Modeling: Comprehensive Engineering Plans for Industry 4.0, IECON 2020 - Ottobre 2020

[11] Kulcsàr G., Varga P., Tatara M. S., Montori F., Inigo M. A. and Urgese G. and Azzoni P., Modeling an Industrial Revolution: How to Manage Large-Scale, Complex IoT Ecosystems?, 2021 IFIP/IEEE International Symposium on Integrated Network Management (IM) - Maggio 2021

[12] Kozma D., Varga P. e Szabò K., Achieving Flexible Digital Production with the Arrowhead Workflow Choreographer, IECON 2020 - Ottobre 2020

[13] A. Stutz, A. Fay, M. Barth e M. Maurmaier, Orchestration vs. Choreography Functional Association for Future Automation Systems, IFAC-PapersOnLine - Dicembre 2020

Ringraziamenti

In questi ultimi anni ho visto cambiare radicalmente la realtà costruita, ma ho anche imparato cosa accomuna tutti i vincenti: la voglia di continuare a sognare e la costanza per conseguire i propri obbiettivi, nonostante le difficoltà, nonostante gli ostacoli, nonostante tutti e tutto.

Un grazie speciale a tutti quelli che ci sono sempre stati e a quelli che hanno creduto in me anche quando anch'io ne ho dubitato, ma anche a tutti coloro che ci sono stati solo quando era comodo a loro, a quelli che non ci sono mai stati e a quelli che non ci sono potuti essere, grazie ad ognuno di voi tutto questo è stato possibile.

Con queste parole si chiude la mia tesi e un capitolo della mia vita.