UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

DIPARTIMENTO DI INGEGNERIA "ENZO FERRARI"

Laurea Triennale in Ingegneria Informatica

Utilizzo di OSGi per lo sviluppo di Digital Twin

Relatore:

Prof: Nicola Bicocchi

Prof: Marco Picone

Candidato:

Luciano Imbimbo

Anno Accademico 2022/2023

Indice

Elenco delle figure

Capitolo 1

Digital Twins

1.1 Cosa sono i Digital Twins?

I DTs [9] possono essere definiti come delle rappresentazioni digitali complete di prodotti, processi, persone, luoghi, infrastrutture, sistemi e dispositivi; forniscono un alter ego software di un oggetto fisico (PO) riflettendone le proprietà, i comportamenti e le relazioni in base al contesto in cui operano. Data la definizione, risulta facile comprendere che i DTs vengono utilizzati in tutti quegli ambienti industriali complessi che devono essere gestiti in modo flessibile e dinamico e che devono evolvere velocemente, alimentati dalle innovazioni digitali e tecnologiche. I corrispettivi fisici e software collaborano e coevolvono per consentire funzionalità come il controllo dei dispositivi, la simulazione, l'analisi e la capacità di migliorare le funzionalità della controparte fisica. L'adozione dei DT favorisce un'interazione semplificata e flessibile tra i moduli software e hardware, consentendo di nascondere l'eterogeneità delle macchine ma anche di facilitare il loro monitoraggio e la loro riconfigurazione. In particolar modo, i DT aumentano l'adattabilità degli ambienti industriali interagendo tra loro e fornendo astrazioni di alto livello per i servizi esterni e le applicazioni.

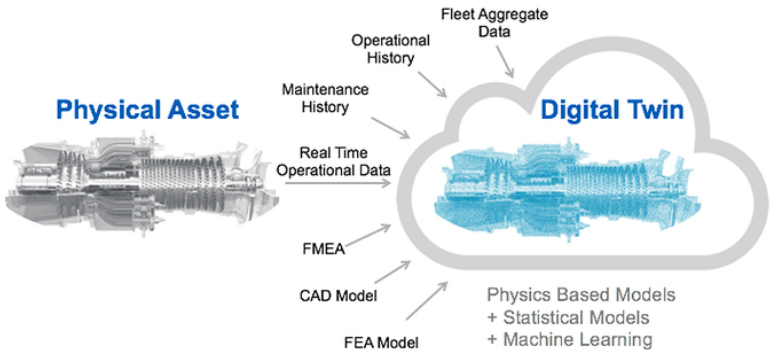


Figura 1.1: Digital Twin ed asset fisici

Per rendere questa tecnologia utile nella maggior parte degli ambienti industriali, i DTs [3] devono evolversi verso entità software attive, cioé capaci di estendere le capacità degli oggetti fisici, di percepire l'ambiente circostante, di comunicare tra di loro e di prendere decisioni in modo autonomo per raggiungere obiettivi produttivi ed operativi. Per far ció, i DT devono godere delle seguenti proprietá:

- **Reflection**
  Questa propritá descrive un DT come un'entità che rispecchia il comportamento e lo stato dell'oggetto fisico; i cambiamenti che avvengono nel DT devono essere riprodotti nel PO e viceversa.

- **Persistency**
  Definisce un DT come un'entità che è sempre disponibile, cioé supera l'esistenza effettiva del PO.

- **Memorization**
  Definisce un DT come un'entità che memorizza tutti i cambiamenti di stato e gli eventi che avvengono sul PO.

- **Augmentation**
  Definisce un DT come un'entità che può estendere le funzioni del PO e offrirle tramite API. L'ampliamento può aggiungere nuove funzionalità non supportate dal PO o fornire l'accesso ai dati in formati particolari.

- **Replication**
  Definisce un DT come un'entità che può essere replicata per soddisfare le esigenze di diverse applicazioni. Le repliche dello stesso PO devono comportarsi in modo coerente, cioè non possono avere uno stato diverso e non possono esibire comportamenti diversi.

Grazie alle caratteristiche appena elencate i DTs consentono di ottimizzare le prestazioni delle apparecchiature, di prevedere le prestazioni future dell'asset fisico e di sperimentare miglioramenti senza doverli testare sul prodotto (simulazione), grazie all'utilizzo di modelli predittivi elaborati da algoritmi di Intelligenza Artificiale. Inoltre offrono una visione visiva e digitale completa dell'impianto di produzione, consentendo il monitoraggio ed il controllo a distanza. Per questo motivo i "gemelli digitali" si stanno affermando, oltre che nell'ambito industriale, anche nel campo sanitario, nel retail e in tutti quei campi fortemente dinamici.

1.2 Framework per lo sviluppo di DT

Per sviluppare oggetti software "complessi" come i Digital Twin, sono stati sviluppati e distribuiti appositi framework, i quali forniscono un insieme di strumenti, tecnologie e metodologie software che facilitano la creazione, la gestione e l'utilizzo dei "gemelli digitali". Esistono diverse piattaforme software che permettono lo

sviluppo di DT e che possono essere personalizzate in base alle proprie esigenze e requisiti. All'interno di questo capitolo vedremo alcuni esempi: Microsoft Azure Digital Twin e Eclipse Ditto.

1.2.1 Microsoft Azure Digital Twins

Azure Digital Twins [11] è una piattaforma software che consente la creazione di modelli digitali di interi ambienti, che possono essere edifici, reti energetiche, stadi ed intere città.

La definizione delle entitá digitali avviene attraverso l'utilizzo di modelli, cioé oggetti software, definiti attraverso l'utilizzo di un linguaggio simile a JSON chiamato Digital Twins Definition Language (DTDL), che descrivono i tipi di entità in base alle loro proprietà di stato, agli eventi di telemetria, ai comandi, ai componenti e alle relazioni. Una volta definiti i modelli di dati, è possibile utilizzarli per definire gemelli digitali che rappresentino ogni entità specifica dell' ambiente da digitalizzare. Difatti un gemello digitale è un'istanza di uno dei modelli definiti in modo personalizzato che può essere connesso ad altri gemelli digitali tramite relazioni per formare un grafico gemello, cioé una rappresentazione dell'intero ambiente. Per esporre i dati dai DTs, Azure Digital Twins viene comunemente utilizzato in combinazione con altri servizi come parte di una soluzione IoT più ampia. Difatti una possibile architettura centralizzata di Azure Digital Twins può contenere i seguenti componenti:

- Istanza del servizio Azure Digital Twins, il quale memorizza i modelli di gemelli e il grafo dei gemelli con il relativo stato e orchestra l'elaborazione degli eventi.
- Una o più applicazioni client che gestiscono l'istanza di Azure Digital Twins
- Un hub IoT per fornire funzionalità di gestione dei dispositivi e flussi di dati IoT
- Una o più risorse di calcolo esterne per elaborare gli eventi generati da Azure Digital Twins
- Servizi a valle per fornire elementi quali l'integrazione dei flussi di lavoro o l'analisi

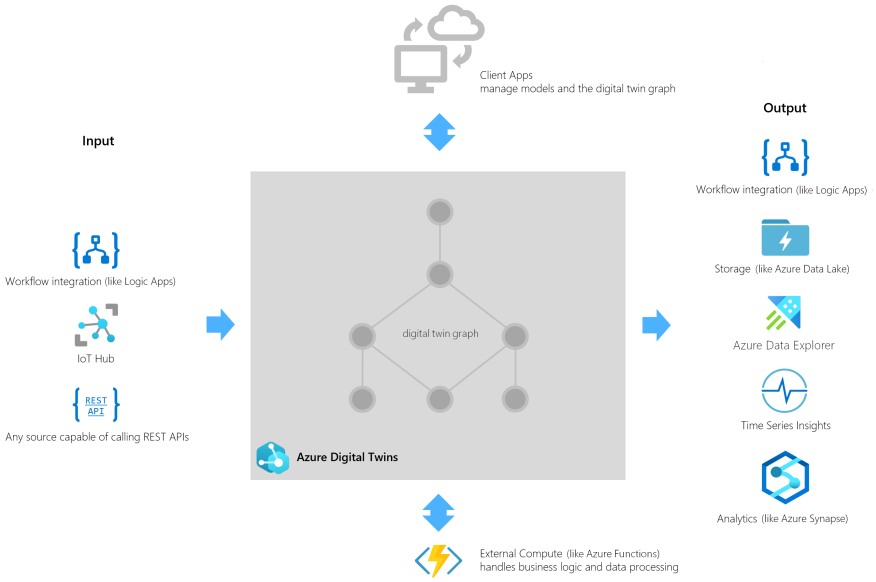


Figura 1.2: Azure Digital Twins

1.2.2 Eclipse Ditto

Eclipse Ditto [10] è un framework free ed open source che permette la definizione di gemelli digitali per la maggior parte dei dispositivi fisici. Assume il ruolo di middleware IoT che astrae l'accesso ai dispositivi IoT tramite API; difatti Eclipse Ditto non mira ad integrare direttamente i dispositivi ma si concentra sul fornire API per applicazioni web, applicazioni mobili o altri servizi backend.

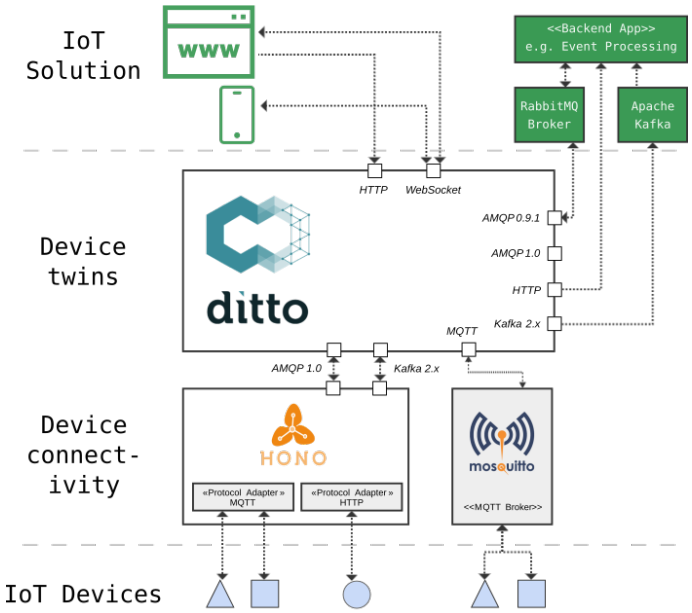


Figura 1.3: Eclipse Ditto

Eclipse Ditto definisce i Digital Twin come la combinazione di 3 principali elementi:

- Thing: rappresentazione digitale del dispositivo fisico.
- Feature: utilizzata per gestire tutti i dati e le funzionalità di una Thing.
- Policy: permette di configurare facilmente un controllo di accesso.

La particolaritá di Ditto é che la comunicazione con le applicazione esterne avviene principalmente in due modi:

- **Twin**
  La prospettiva gemella di Eclipse Ditto è una struttura dati persistente a cui i dispositivi si collegano e in cui memorizzano il loro ultimo stato segnalato. Le applicazioni possono leggere l'ultimo stato segnalato ed essere avvisate dei cambiamenti. Questo può essere fatto attraverso il rispettivo endpoint del protocollo. I dispositivi utilizzano questi endpoint anche per aggiornare il loro stato nel database o per ricevere notifiche.
- **Live**
  In questo caso, Ditto funziona come un router. Un'applicazione che vuole accedere alla temperatura live di un sensore viene instradata da Ditto direttamente al dispositivo. In questo caso, Ditto non elabora i comandi e non emette gli eventi ma si limita a trasmettere le informazioni.

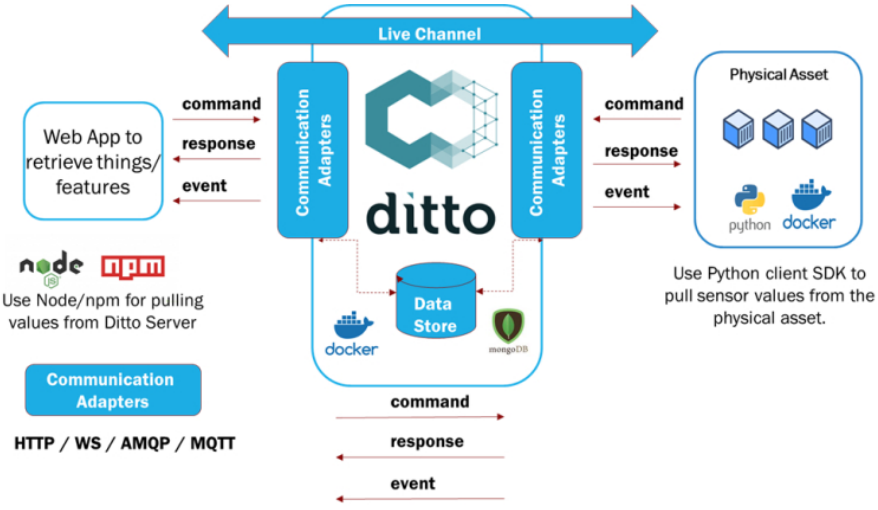


Figura 1.4: Un esempio di Digital Twin costruito con Eclipse Ditto.

1.3 WhiteLabel Digital Twin Framework

WhiteLabel Digital Twin (WLDT) [6] é una libreria Java sviluppata da Marco Picone, ricercatore e docente presso l'Università degli Studi di Modena e Reggio Emilia,

con l'intento di sviluppare facilmente ed in modo efficace Digital Twins per applicazioni IoT. I principali concetti della libreria, che verrano descritti brevemente in questo capitolo, sono stati utilizzati come base del lavoro di integrazione tra DTs e programmazione modulare.

1.3.1 Astrazione del DT

La libreria è stata progettata identificando i DTs come componenti software attivi basati sull'astrazione schematizzata in figura, la quale illustra la struttura e definisce i componenti principali di un Digital Twin e ne chiarisce la responsabilità di tramite tra il mondo digitale e il mondo fisico.

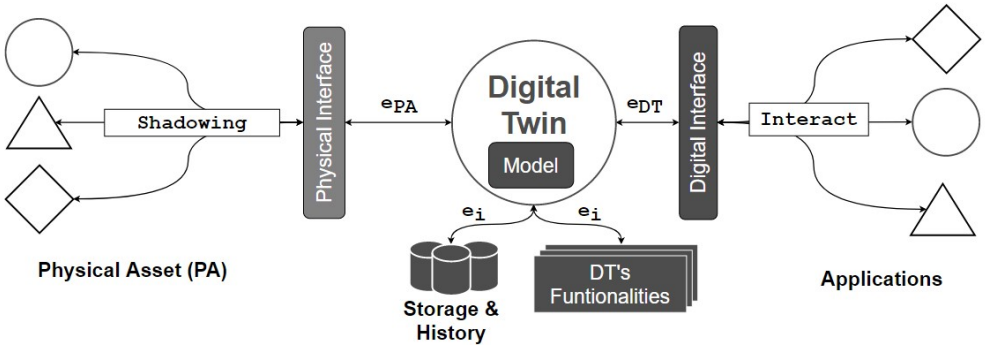


Figura 1.5: Digital Twin: astrazione e modello

I componenti principali di un Digital Twin sono i seguenti:

- **Interfaccia fisica**: é l'entità responsabile sia del processo iniziale di digitalizzazione che della responsabilità continua di mantenere il DT e il PO sincronizzati durante il loro ciclo di vita.
- **Interfaccia digitale**: é il componente complementare dell'interfaccia fisica e responsabile della gestione delle variazioni ed eventi interni del DT verso entità digitali esterne ed applicazioni.
- **Modello del DT**: é il modulo che definisce il comportamento e le funzionalità del Digital Twin.

A differenza dei framework descritti in precedenza, nei quali il DT é un elemento centralizzato a cui poter accedere attraverso API, WLDT intende massimizzare la modularità e la riutilizzabilità definendo un DT come una composizione di moduli indipendenti tra loro che cooperano per svolgere determinate operazioni. Difatti WLDT scompone il Digital Twin in moduli, altamente coesi e debolmente accoppiati, con l'intento di avere una certa flessibilitá e dinamicitá tra i componenti. Il livello di modularitá che mette a disposizione la libreria é la motivazione principale per la quale é stata utilizzata come base di partenza di questo lavoro.

La problematica principale del framework riguarda la gestione, del tutto assente, dei vari moduli a tempo di esecuzione. É in questa ottica che entra in gioco la programmazione a bundle offerta da OSGi che permette la gestione dinamica del ciclo di vita di ogni modulo e, di conseguenza, il ciclo di vita dell'intero DT. Quindi grazie al framework OSGi gli utilizzatori potranno installare, avviare, fermare e disinstallare, a tempo di esecuzione, le varie componenti del DT.

Inoltre all'interno della libreria, per mantenere le informazioni e lo stato del PO, é stato definito un componente chiamato Digital Twin State, il quale é rappresentato dai seguenti termini:

1. Proprietà: rappresentano gli attributi osservabili del corrispondente PO come dati etichettati, i cui valori possono cambiare dinamicamente nel tempo.

2. Eventi: rappresentano gli eventi che avvengono nel PO.

3. Relazioni: rappresentano i collegamenti che esistono tra il PO modellato e altri asset fisici delle organizzazioni.

4. Azioni: rappresentano le azioni che possono essere invocate sul PO tramite interazione con il DT.

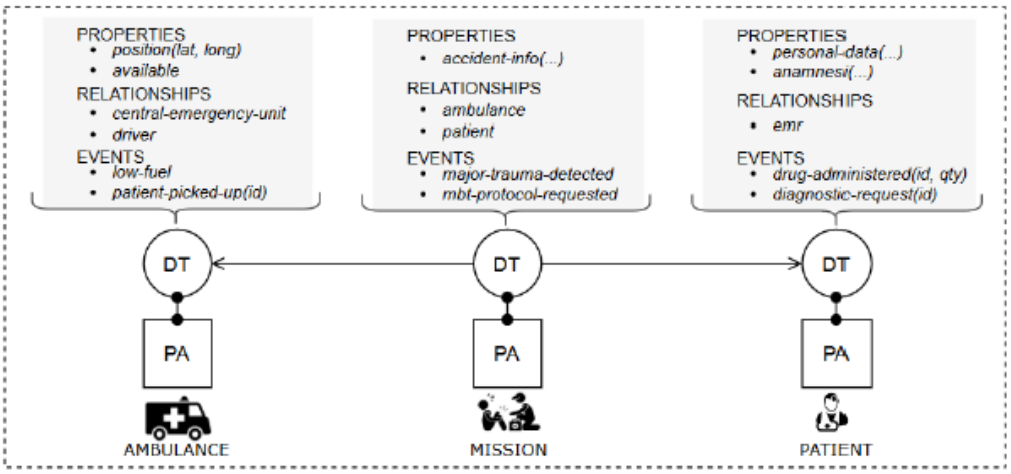


Figura 1.6: Digital Twin State

1.3.2 Processo di shadowing

Come descritto in precedenza, l'interfaccia fisica é il componente responsabile di mantenere la sincronizzazione tra DT e PO; questa funzionalita é fornita grazie all'utilizzo di un processo di shadowing. Nello specifico, ogni aggiornamento rilevante dello stato del PO non si traduce direttamente in una modifica dello stato del DT ma é tradotto in una sequenza di 3 fasi:

- Ogni varazione é modellata come un evento fisico (e_{PA}).

- L'evento viene propagato al DT.
- Viene applicato una funzione di shadowing, la quale contiene le condizioni per un cambio di stato.

Inoltre il processo di shadowing permette al DT di riflettere e invocare le possibili azioni "digitali" sul PO: il DT riceve una richiesta di azione dalla sua interfaccia digitale, applica la funzione di shadowing per convalidarla e poi propaga la richiesta attraverso la sua interfaccia fisica.

1.3.3 Architettura ad evento

Viste le particolaritá e le caratteristiche della libreria, é stato utilizzato, un modello architetturale software, comunemente utilizzato nei sistemi IoT, basato su eventi. Brevemente il concetto chiave del modello è che i componenti di un sistema comunicano attraverso la generazione, la trasmissione e il consumo di eventi, che rappresentano, avvenimenti o variazioni all'interno del sistema. All'interno di questo modello architetturale sono 2 i principali attori che operano sugli eventi:

- I produttori di eventi, responsabili della generazione ed emissione di eventi quando si verificano condizioni o azioni specifiche.
- I consumatori di eventi, cioé i componenti che si sottoscrivono agli eventi e intraprendono azioni quando li ricevono.

Per sviluppare un comportamento dinamico basato su eventi, sono stati utilizzati i principali concetti della gestione eventi in Java, cioé Listener e callback.

Capitolo 2

OSGi

2.1 Che cos'é OSGi?

Come giá detto, all'interno degli ambienti industriali, la complessitá é la sfida piú ardua da superare. L'approccio ingegneristico piú efficace ed utile é il famoso "Dividi et Impera", cioé suddividere problemi complessi in sottoproblemi piú piccoli e comprensibili. Come vedremo nel prossimo paragrafo Java da solo non supporta la modularità ed é in questi casi che entra in gioco OSGi, il quale non è né più né meno che un modo per creare applicazioni modulari in Java. In particolar modo OSGi (Open Service Gateway Initiative) [1] è un framework Java che fornisce un ambiente runtime modulare e dinamico per la creazione e la distribuzione di applicazioni; difatti consente agli sviluppatori di creare applicazioni altamente modulari composte da componenti liberamente accoppiate e riutilizzabili chiamati bundle.

2.2 Perché Java non é sufficente?

Java, come unitá standard di distribuzione, mette a disposizione il file JAR, il quale é un'aggregazione di tanti piccoli file. Possiamo utilizzare i file JAR come se fossero dei moduli? Prima di rispondere a questa domanda, dobbiamo chiarire il concetto di modulo. Un modulo é un oggetto software che gode delle seguenti proprietá:

- **Autonomia**
  Un modulo è un tutt'uno logico, può essere spostato, installato e disinstallato come una singola unità ma non è un elemento atomico.
- **Altamente coeso**
  Un modulo deve attenersi ad uno scopo logico e realizzarlo bene.
- **Debolmente accoppiato**
  Un modulo non deve occuparsi dell'implementazione interna degli altri moduli con cui interagisce.

Chiarito il concetto di modulo e date le caratteristiche dei file JAR, possiamo dichiarare che i file JAR non possono essere utilizzati come moduli per le seguenti motivazioni:

- **Non esiste un concetto di runtime**
  I JAR sono significativi solo in fase di compilazione e di distribuzione.
- **Non contengono metadati standard per indicare le loro dipendenze.**
- **Non sono classificati in base alla versione**
  Supponiamo di trovarci in una situazione descritta dalla figura sottostante, nella quale la nostra applicazione richiede due librerie: A e B. Entrambe le librerie dipendono a loro volta da una terza libreria C, ma la libreria A richiede la versione 1.2 di C, mentre la libreria B richiede la versione 1.1 di C e non funzionerà con la versione 1.2. Questo tipo di problema non può essere risolto senza riscrivere A o B in modo che entrambe funzionino con la stessa versione di C. La motivazione a questa problematica é, come sempre, legata al classpath piatto e globale; infatti se proviamo a caricare entrambe le versioni di C nel classpath, verrà utilizzata solo la versione che verrá trovata e risolta per prima e quindi sia A che B dovranno usare quella specifica versione.

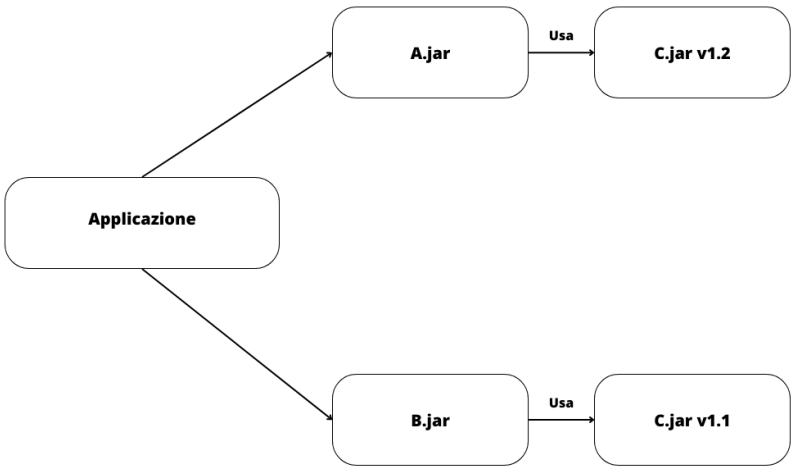


Figura 2.1: Versioni contrastanti

Quindi abbiamo bisogno di specificare un intervallo di versioni perché dipendere da una singola versione di una determinata libreria renderebbe il nostro sistema fragile e poco flessibile.

- **Non esiste un meccanismo di information hiding**
  La maggior parte dei file JAR contengono più di un package, i quali generalmente contengono classi che devono accedere alle classi di altri package dello

stesso JAR. Per far funzionare il tutto dobbiamo rendere pubbliche la maggior parte delle nostre classi, perché public è l'unico modificatore di accesso che rende le classi visibili oltre i confini del package. Di conseguenza, tutte le classi dichiarate pubbliche sono accessibili anche ai client al di fuori del JAR e quindi l'intero JAR è di fatto un'API pubblica che non permette l'incapsulamento delle informazioni.

Inoltre il Java Development Kit (JDK) fornisce solo strumenti e tecnologie molto rudimentali per la gestione e la composizione dei JAR.

2.3 Perché OSGi é la soluzione?

La principale fonte dei tanti problemi Java descritti in precedenza, è il classpath globale e piatto. Per questo motivo, OSGi adotta un approccio completamente differente: ogni modulo ha il proprio classpath, separato dal classpath di tutti gli altri moduli; questo elimina immediatamente quasi la totalitá dei problemi discussi in precedenza. Grazie a questo approccio la relazione di dipendenza tra due moduli non è gerarchica: non c'è un "genitore" o un "figlio", ma solo una rete di fornitori e utenti.

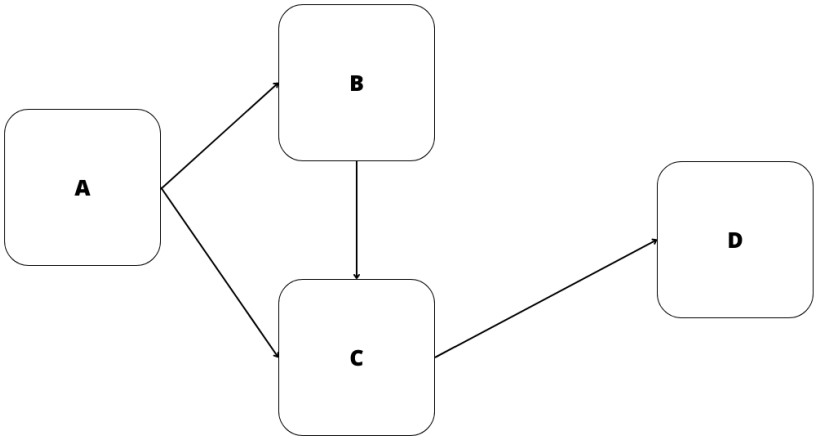


Figura 2.2: Grafo dipendenze moduli

Alla base del framework c'é il concetto di bundle, che di fatto rappresenta il modulo di una applicazione OSGi; questo bundle é semplicemente un file JAR, al quale dobbiamo semplicemente aggiungere dei metadati standard:

- **Il nome del bundle**.
- **La versione del bundle**.

- **L'elenco delle importazioni e delle esportazioni**.

Questi metadati sono inseriti all'interno del file JAR in un file speciale chiamato MANIFEST.MF, che fa parte di tutti i file JAR e serve proprio a questo scopo: è un luogo standard per aggiungere metadati opzionali.

```
1 Manifest-Version: 1.0
2 Created-By: 1.4.2_06-b03 (Sun Microsystems Inc.)
3 Bundle-ManifestVersion: 2
4 Bundle-Name: My First OSGi Bundle
5 Bundle-SymbolicName: org.osgi.example1
6 Bundle-Version: 1.0.0
7 Bundle-RequiredExecutionEnvironment: J2SE-1.5
8 Import-Package: javax.swing
```

Figura 2.3: File MANIFEST.MF

Grazie all'utilizzo dei metadati all'interno dei file MANIFEST.MF possiamo risolvere i problemi di conflitto e di versione descritti in precedenza. Come sviluppatori di bundle, dovremo semplicemente scrivere alcune dichiarazioni per definire gli export dei pacchetti, con tanto di attributo di versione, e gli import specificando un range di versione. Infatti sará il framework OSGi ad assumersi la responsabilità di far corrispondere l'importazione con l'esportazione corrispondente. Questo meccanismo é noto come processo di risoluzione, il quale è piuttosto complesso ma é implementato direttamente dal framework OSGi e non dai bundle stessi. Una volta che un'importazione viene abbinata ad un'esportazione, i bundle coinvolti vengono "cablati" (wired), ciò significa che, riferendoci alla figura 3.2, quando si verifica una richiesta di caricamento di una classe nel bundle A la richiesta sarà immediatamente delegata al ClassLoader del bundle B.
Infine l'utilizzo del file MANIFEST.MF ci permette di incapsulare le informazioni all'interno dei bundle; infatti all'interno del framework, solo i pacchetti che sono esplicitamente esportati da un bundle possono essere importati e utilizzati in un altro bundle (tutti i pacchetti sono "privati" per impostazione predefinita). Questo metodo permette di nascondere facilmente ai client i dettagli dell'implementazione interna dei nostri bundle.

2.4 Gestione dinamica dei moduli

In veritá, OSGi non é semplicemente un sistema per la programmazione modulare in Java, ma puó essere definito come un sistema di gestione di moduli dinamici. Infatti il framework é formato da un livello funzionale che permette la gestione completa del ciclo di vita di ogni singolo bundle, insieme ad un modello di programmazione che consente di pubblicare dinamicamente i cosidetti "servizi". Quest'ultimi sono il mezzo principale di comunicazione e collaborazione tra i bundle in OSGi; difatti un servizio è un oggetto Java che implementa un'interfaccia specifica e fornisce

funzionalità ad altri bundle. I bundle possono registrare i propri servizi ed utilizzare i servizi messi a disposizione da altri bundle.
Il ciclo di vita di un bundle é schematizzato dalla figura 3.4.

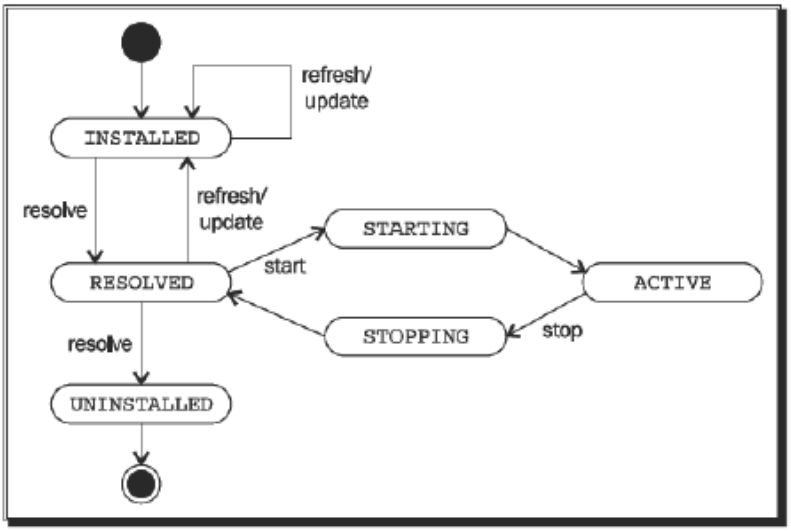


Figura 2.4: Stati di un bundle OSGi

Quindi all'interno del framework, i bundle hanno un significato a runtime e possono essere installati, aggiornati e disinstallati senza stoppare l'intera applicazione.

2.5 Apache Felix

OSGi è uno standard definito da un'alleanza di circa quaranta aziende e le sue specifiche sono free ed open source. Per questo motivo, esistono diversi framework OSGi implementati in modo indipendente; i piú famosi ed utilizzati sono Apache Felix [2] ed Eclipse Equinox. In particolar modo, in questo capitolo, verrá posto il focus sull'implementazione presentata da Apache perché banalmente, per la sua facilitá d'uso, é stata utilizzata in questo lavoro di integrazione tra OSGi e Digital Twin. Apache Felix, oltre a dare una propria implementazione del framework OSGi, fornisce alcuni servizi utili che migliorano notevolemente l'esperienza degli sviluppatori e semplificano le attività di amministrazione del framework, come ad esempio:

- Gestore delle dipendenze
- File Install: servizio che fornisce una semplice gestione del deployement dei bundle basata su directory
- Gogo: shell avanzata per interagire con il framework OSGi.
- Maven Bundle Plugin: add-on che migliora l'esperienza di sviluppo dei bundle le fornendo l'automazione del processo di creazione dei metadati e del file MANIFEST.MF.

Capitolo 3

OSGi Digital Twins

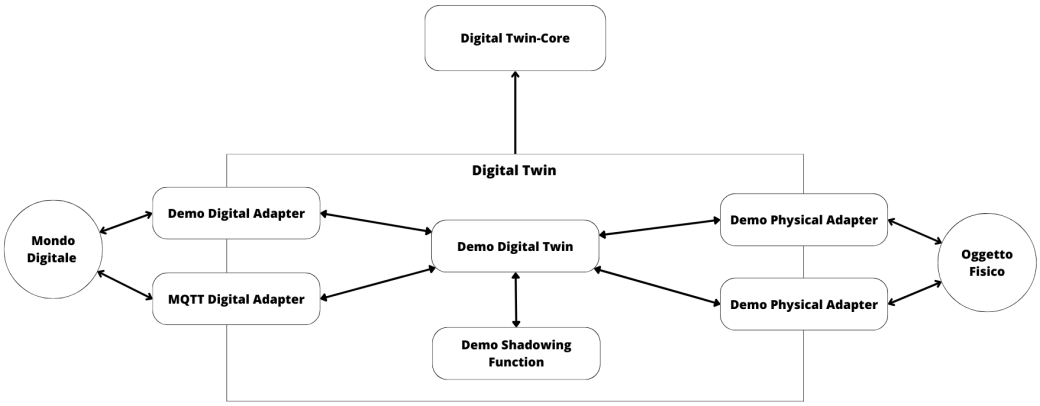
All'interno di questo capitolo verrá descritto, piú nel dettaglio, l'applicativo, sviluppato durante l'attivitá progettuale universitaria, che integra le principali funzionalitá della programmazione a bundle OSGi all'interno di una versione minimale della libreria WLDT, con l'obiettivo di migliorare la modularità, la flessibilità e la dinamicità nello sviluppo di Digital Twin.

3.1 Struttura del progetto

L'applicazione, seguendo l'astrazione del DT proposta nel Capitolo 2, é composta da 5 principali bundle:

- DigitalTwin-Core: bundle che rappresenta una versione minimale della libreria WLDT.
- Demo Physical Adapter: bundle per la gestione dell'interfaccia fisica e che monitora una determinata proprietá del PO.
- Demo Digital Adapter: bundle per la gestione dell'interfaccia digitale
- Demo Shadowing Fuction: bundle per la gestione del processo di shadowing
- Demo Digital Twin: bundle per la gestione del modello del DT.

La struttura del progetto, schematizzata dalla figura sottostante, descrive tutte le dipendenze e le interazioni tra i vari bundle: tutti i bundle che compongono il DT dipendono da DigitalTwin-Core, cioé il bundle che mette a disposizione le principali funzionalitá (classi ed interfacce) per lo sviluppo dei DTs. All'interno del DT, le componenti possono essere avviate ed fermate in modo autonomo; in particolar modo, come vedremo in seguito, interfacce fisiche e digitali possono essere collegate e scollegate in modo autonomo ed indipendente dal Digital Twin, senza intaccarne lo stato (Demo Digital Twin). L'unico bundle o componente strettamente legato allo stato del DT è, per motivi architetturali, il Demo Shadowing Function.



I componenti "interni" al DT comunicano tra loro con l'utilizzo delle principali funzionalità messe a disposizione dal framework OSGi: Service e ServiceTracker.

3.1.1 Service e ServiceTracker

Un servizio è un semplice oggetto Java, il quale viene pubblicato nel registro dei servizi OSGi con il nome di una o più interfacce Java ed i consumatori che desiderano utilizzarlo possono cercarlo usando uno qualsiasi di questi nomi. La particolarità del servizio è che non crea una dipendenza statica tra se stesso ed il consumatore; difatti i servizi possono essere registrati o eliminati dinamicamente in qualsiasi momento, in modo da formare solo associazioni temporanee. I consumatori per ascoltare e ricevere notifiche sullo stato del servizio, possono utilizzare le classi ed interfacce, messe a disposizione dal framework, come ServiceTracker e ServiceListener.

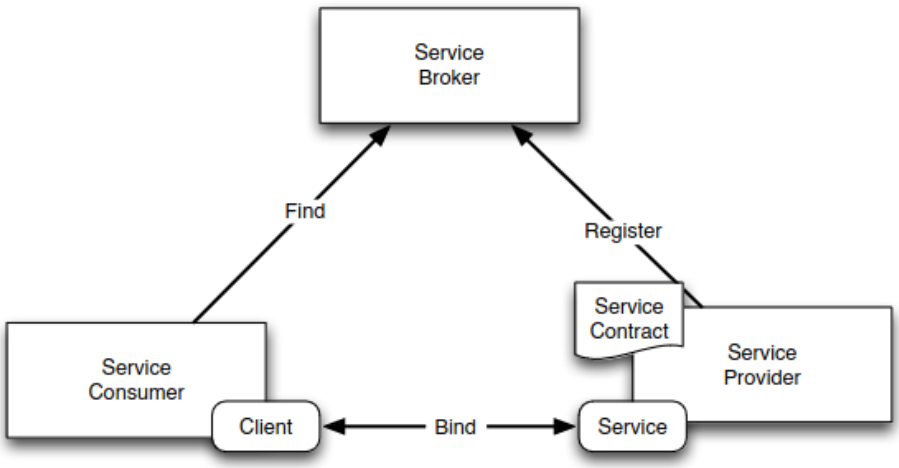


Figura 3.1: Service Pattern OSGi

3.2 Definizione dei bundle

Prima di entrare nel dettaglio di tutte le componenti, dobbiamo descrivere come definire un bundle e come rendere il bundle eseguibile all'interno del framework. Innanzitutto, per definire un bundle, come già detto, dobbiamo inserire dei metadati all'interno del file MANIFEST.MF. Ma con l'utilizzo del Maven Bundle Plugin, messo a disposizione da Apache Felix, possiamo dimenticarci del file MANIFEST.MF e possiamo semplicemente aggiungere alcuni tag all'interno del file pom.xml, presente all'interno di ogni progetto Maven.

I file pom.xml dei bundle dell'applicativo sono, tranne per qualche piccola differenza, del tutto speculari; prendiamo quindi, come esempio, il file pom.xml del bundle DigitalTwin-Core.

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
    <groupId>org.digitaltwin</groupId>
    <artifactId>digital-twin-core</artifactId>
    <version>2.0</version>
    <packaging>bundle</packaging>
    <name>digitalTwinCore</name>

    <dependencies>
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.core</artifactId>
            <version>6.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>osgi.cmpn</artifactId>
            <version>7.0.0</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>5.1.9</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Export-Package>org.digitaltwin.*</Export-Package>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Per prima cosa possiamo notare che per utilizzare il Maven Bundle Plugin dovremmo "dichiararlo" all'interno del tag build, specificando la versione, l'utilizzo di estensioni e la configurazione. All'interno della configurazione del plugin dovremmo inserire i tag Export-Package e Import-Package per definire le dipendenze del

bundle. In secondo luogo dovremmo aggiungere il tag packaging con valore bundle per segnalare al plugin che dovrà generare come artifatto, un file che può essere indirizzato usando delle coordinate e che Maven scarica, installa e distribuisce, un bundle e non un semplice JAR. Infine dovremmo aggiungere le dipendenze statiche all'interno del tag dependencies.

Con l'utilizzo del framework OSGi, dovremmo fare molta attenzione con la dichiarazione delle dipendenze statiche, specificate all'interno dei tag Import-Package e dependency, perchè un import dovrá essere sempre soddisfatto da un export di un altro bundle installato all'interno del framework. Nel caso del pom.xml descritto in precedenza, non ci sono particolari problemi perchè i package org.osgi.core e osgi.cmpn sono interni al framework e quindi non devono essere installati. Invece nel caso dovessimo utilizzare librerie esterne al JRE ed al framework OSGi, dovremmo installare all'interno dell'ambiente OSGi i bundle che esportano quei determinati package utilizzati. Per esempio per utilizzare le classi fornite dalla libreria Paho, utile per lo sviluppo del Digital Adapter MQTT, dovremmo installare all'interno del framework il bundle che esporta i package della libreria Paho.

Per rendere il bundle "eseguibile", cioè esporre le funzionaltà del bundle all'interno del framework, dovremmo definire una classe che funga da punto di accesso al bundle; difatti il framework utilizzerà questo entry-point per gestire il ciclo di vita del bundle. Per farlo, dobbiamo sviluppare un cosidetto bundle activator, il quale si tratta semplicemente di una classe che implementa l'interfaccia BundleActivator, una delle interfacce più importanti di OSGi. L'interfaccia BundleActivator contiene solamente due metodi, start e stop, che vengono richiamati dal framework quando il bundle viene avviato e arrestato.

```
1  import org.osgi.framework.BundleActivator;
2  import org.osgi.framework.BundleContext;
3
4  public class ActivatorBundle implements BundleActivator {
5      @Override
6      public void start(BundleContext context) throws Exception { }
7
8      @Override
9      public void stop(BundleContext context) throws Exception { }
10 }
```

Una volta implementata l'interfaccia dovremmo segnalare al framework la classe che funge da BundleActivator inserirendo il nome della classe nel tag Bundle-Activator, tag interno alla configurazione del Maven Build Plugin.

```
1 <configuration>
2     <instructions>
3         <Bundle-Activator>package.ActivatorBundle</Bundle-Activator>
4     </instructions>
5 </configuration>
```

Alla fine di questo processo, il nostro bundle sarà formato da un file MANIFEST.MF, generato attraverso il Maven Build Plugin, che ne descrive le proprietà e da un BundleActivator che funge da entry-point.

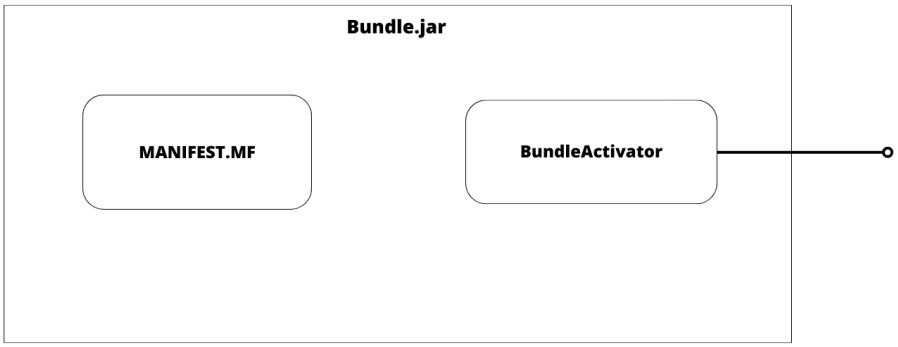


Figura 3.2: Struttura interna di un bundle

3.3 Demo Digital Twin

Adesso entriamo nel dettaglio delle varie componenti dell'applicativo. Il componente centrale, nell'astrazione del DT proposta in questo lavoro, è il bundle DemoDigitalTwin che rappresenta il modello del Digital Twin e tiene traccia dello stato utilizzando un oggetto fornito dalla libreria WLDT. Il bundle è composto da una sola classe denominata DigitalTwinActivator che funge da BundleActivator.

3.3.1 DigitalTwin

Prima di approfondire nel dettaglio l'implementazione della classe DigitalTwinActivator, dobbiamo descrivere la classe, fornito dalla libreria, che rappresenta il cuore del DT. La classe DigitalTwin tiene traccia e gestisce tutte le componenti di un DT:

- Modello
- Stato
- Lista dei PhysicalAdapter
- Lista dei DigitalAdapter
- Aggiornamenti ricevuti dai physical adapter
- Stato dei physical adapter
- Shadowing function
- Listener per la gestione degli eventi

La particolarità di questa classe è che è strettamente legata alla classe ShadowingModelFunction ed il codice sottostante ne descrive il motivo.

```
public class DigitalTwin implements PhysicalAdapterListener {

    private String digitalTwinId;
    private ShadowingModelFunction shadowingModelFunction = null;
    private IDigitalTwinState digitalTwinState = null;
    private Map<String, PhysicalAssetDescription>
    physicalAdaptersPhysicalAssetDescriptionMap;
    private List<PhysicalAdapter> physicalAdapterList;
    private List<DigitalAdapter> digitalAdapterList;
    private List<LifeCycleListener> lifeCycleListenerList;
    private Map<String, Boolean> physicalAdaptersBoundStatusMap;
    private ModelEngine modelEngine = null;

    public DigitalTwin(String ID, ShadowingModelFunction
    shadowingModelFunction) {
        init(ID, shadowingModelFunction);
    }

    private void init(String ID, ShadowingModelFunction
    shadowingModelFunction){

        this.shadowingModelFunction = shadowingModelFunction;
        this.modelEngine = new ModelEngine(this.digitalTwinState, this.
    shadowingModelFunction);
        lifeCycleListenerList.add(this.modelEngine);
}
```

Come possiamo vedere all'interno del metodo init, l'istanzazione dell'oggetto modelEngine, il quale rappresenta il modello, dipende da un oggetto di tipo ShadowingModelFunction istanziato in precedenza. È questa la principale motivazione per la quale, come vedremo nel prossimo capitolo, l'oggetto DigitalTwin presente all'interno della classe DigitalTwinActivator verrà instanziato solamente quando un servizio di tipo ShadowingModelFunction verrà registrato.

Inoltre il Digital Twin implementa l'interfaccia PhysicalAdapterListener, la quale definisce i metodi per la gestione degli eventi occorsi sulle interfacce fisiche connesse. Questo permetterà al DT di ricevere notifiche di eventuali aggiornamenti dei valori monitorati dalle interfacce fisiche.

3.3.2 DigitalTwinActivator

La classe DigitalTwinActivator, come si può notare dal codice sottostante, contiene alcuni attributi che permettono la gestione dinamica delle componenti di un DT:

- context: una variabile che salva il contesto del bundle
- digitalTwin: oggetto Java che rappresenta il DT.
- Una serie di ServiceTracker che permettono la connessione e disconnessione automatica ed indipendente dei bundle rappresentanti le componenti del DT.

```
public class DigitalTwinActivator implements BundleActivator{

    private BundleContext context;
    private DigitalTwin digitalTwin = null;

    private ServiceTracker physicalAdapterServiceTracker;

    private ServiceTracker digitalAdapterServiceTracker;

    private ServiceTracker shadowingModelFunctionServiceTracker;
```

All'interno della classe, essendo un'implementazione dell'interfaccia BundleActivator, troviamo l'override dei metodi start e stop.

Il metodo start salva il contesto del bundle nella variabile context ed istanzia ed apre, cioè mette in ascolto ed in attesa di notifiche, i vari ServiceTracker dichiarati in precedenza. In questo modo il bundle rimarrà in attesa di notifiche dai vari componenti finchè non verrà stoppato; in particolar modo non appena i servizi di ShadowingFunction, Physical Adapter e Digital Adapter verranno registrati nel registro dei servizi del framework, il DT, in attesa, verrà notificato della registrazione ed agirà di conseguenza.

```
public void start(BundleContext context) throws Exception {
    this.context = context;

    physicalAdapterServiceTracker = new ServiceTracker<>(context,
    PhysicalAdapter.class.getName(), new ServiceTrackerCustomizer<>() { /*
    ... */ });

    digitalAdapterServiceTracker = new ServiceTracker<DigitalAdapter,
    DigitalAdapter>(context, DigitalAdapter.class.getName(), new
    ServiceTrackerCustomizer<DigitalAdapter, DigitalAdapter>() { /* ... */
    });

    shadowingModelFunctionServiceTracker = new ServiceTracker<>(context,
    ShadowingModelFunction.class.getName(), new ServiceTrackerCustomizer
    <>() { /* ... */ });

    physicalAdapterServiceTracker.open();
    digitalAdapterServiceTracker.open();
    shadowingModelFunctionServiceTracker.open();
}
```

Come possiamo notare dal codice soprastante, durante la definizione dei serviceTracker abbiamo istanziato l'interfaccia ServiceTrackerCustomizer che ci permette di indicare quali azioni eseguire in caso di aggiunta, rimozione e modifica del servizio tracciato:

- In caso di registrazione di un servizio di tipo ShadowingModelFunction, il bundle reagirà, per le motivazione descritte in precedenza, istanziando un oggetto di tipo DigitalTwin e registrando l'oggetto come servizio, rendendosi visibile alle interfacce fisiche e digitali. In caso di rimozione, invece, il bundle reagirà rimuovendo il servizio di tipo DigitalTwin, eliminando le interfacce precedentemente connesse e non permettendo la connessione di nuovi adapter.

```
@Override
public Object addingService(ServiceReference<Object> reference) {
    if(digitalTwin != null){
        ShadowingModelFunction shadowingModelFunction = (
    ShadowingModelFunction context.getService(reference);
        digitalTwin = new DigitalTwin(String.valueOf(new Random().
    nextInt()), shadowingModelFunction);
        return context.registerService(DigitalTwin.class.getName(),
    digitalTwin, null);
    }
        return null;
}
@Override
public void removedService(ServiceReference<Object> reference, Object
    service) {
    ServiceRegistration<?> registration = (ServiceRegistration)
    service;
    registration.unregister();
    context.ungetService(reference);
}
```

- In caso di aggiunta o rimozione dei servizi PhysicalAdapter e DigitalAdapter, il bundle reagirà aggiungendo e rimuovendo gli adapter, forniti dai servizi, dal DT.

```
@Override
public Object addingService(ServiceReference<Object> reference) {
    PhysicalAdapter newPhysicalAdapter = (PhysicalAdapter) context.
    getService(reference);
    if(digitalTwin != null){
        digitalTwin.addPhysicalAdapter(newPhysicalAdapter);
    }
    return newPhysicalAdapter;
    }
@Override
public void removedService(ServiceReference<Object> reference, Object
    service) {
    digitalTwin.removePhysicalAdapter((PhysicalAdapter) service);
}
```

Di conseguenza l'oggetto DigitalTwin svolgerà le proprie funzionalità solo in presenza di un bundle che fornisca l'implementazione della ShadowingModelFunction; questo crea, inevitabilmente, una forte dipendenza tra il bundle DemoDigitalTwin ed il bundle DemoShadowingFunction.

Infine all'interno del metodo stop troviamo le istruzione per chiudere e fermare l'ascolto dei vari ServiceTracker. Da notare che non è necessario aggiungere istruzioni per eliminare i servizi registrati perché sarà il framework OSGi a disattivare automaticamente tutti i servizi precedentemente registrati dal nostro bundle.

```
@Override
public void stop(BundleContext context) throws Exception {
    physicalAdapterServiceTracker.close();
    digitalAdapterServiceTracker.close();
    shadowingModelFunctionServiceTracker.close();
}
```

3.4 Shadowing Function

Il bundle DemoShadowingFunction, necessario al funzionamento del Digital Twin, è composto da un singolo package contenente 2 semplici classi: DemoShadowingFunctionActivator e DemoShadowingFunction.

DemoShadowingFunctionActivator è la classe che svolge la funzione di bundle activator e che quindi implementa l'interfaccia BundleActivator ed i metodi start e stop.

```
public class DemoShadowingFunctionActivator implements BundleActivator{

    @Override
    public void start(BundleContext context) throws Exception {
        DemoShadowingFunction demoShadowingFunction = new
    DemoShadowingFunction("1", context);
        context.registerService(ShadowingModelFunction.class.getName(),
    demoShadowingFunction, null);
    }

    @Override
    public void stop(BundleContext context) throws Exception { }
}
```

Dal codice della classe notiamo che il bundle non dovrà ascoltare per eventuali servizi o rimanere in attesa di determinati componenti per svolgere le proprie funzionalità ma semplicemente, al suo avvio, istanzierà un oggetto di tipo DemoShadowingFunction ed attraverso la propria variabile di contesto registrerà l'oggetto come servizio. Questo permetterà all'eventuale DemoDigitalTwin in ascolto di utilizzare l'oggetto esposto come servizio ed istanziare un oggetto di tipo DigitalTwin.

L'altra classe del bundle è DemoShadowingFunction, la quale non è altro che un'estensione della classe ShadowingModelFunction, fornita dalla libreria WLDT e che definisce gli attributi, i metodi e le principali responsabilità del processo di shadowing. Il codice sottostante chiarisce le funzionalità della shadowing function e mostra i principali aggiornamenti a cui deve reagire:

- onDigitalTwinBound: questo evento si verifica quando tutte le interfacce fisiche connesse al DT hanno comunicato la lista di proprietà che monitorano e sono pronte ad inviare aggiornamenti. Al verificarsi dell'evento la shadowing function crea, all'interno dello stato del DT, una proprietà in comune accordo con le proprietà osservate dalle interfacce fisiche.

```
@Override
public void onDigitalTwinBound(Map<String, PhysicalAssetDescription>
    adaptersPhysicalAssetDescriptionMap) {
    adaptersPhysicalAssetDescriptionMap.values().forEach(pad -> {
        pad.getProperties().forEach(p -> {
            try {
                DigitalTwinStateProperty<?> StateProperty = new
    DigitalTwinStateProperty<>(p.getKey(), p.getInitialValue());
                valueFromPhysicalAdapter.add(StateProperty);
                this.digitalTwinState.createProperty(StateProperty);
                notifyPropertyCreated(StateProperty);
                DigitalTwinStateProperty<?> averageProperty = new
    DigitalTwinStateProperty<>("Average " + p.getKey(),0);
                this.digitalTwinState.createProperty(averageProperty);
                notifyPropertyCreated(averageProperty);
                } catch (Exception ignored) {}
        });
    });
}
```

- onDigitalTwinUnBound: è l'evento complementare a onDigitalTwinBound e può verificarsi a causa di errori su una o più interfacce fisiche.
- onPhysicalAdapterUpdate: è l'evento che si verifica in caso di aggiornamento di una o più proprietà da parte di una o più interfacce fisiche. Al verificarsi dell'evento la shadowing function aggiornerà la corrispondente proprietà con il nuovo valore.

```
protected void onPhysicalAdapterUpdate(String adapterId,
    PhysicalAssetDescription adapterPhysicalAssetDescription) {
    adapterPhysicalAssetDescription.getProperties().forEach(p -> {
        try {
            DigitalTwinStateProperty<?> StateProperty = new
    DigitalTwinStateProperty<>(p.getKey(), p.getInitialValue());
            valueFromPhysicalAdapter.add(StateProperty);
            this.digitalTwinState.updateProperty(StateProperty);
            notifyPropertyUpdate(StateProperty);

            if(valueFromPhysicalAdapter.size() % 5 == 0){
                DigitalTwinStateProperty<?> averageProperty = new
    DigitalTwinStateProperty<>("Average " + p.getKey(),average());
                this.digitalTwinState.updateProperty(averageProperty);
                notifyPropertyUpdate(averageProperty);
                valueFromPhysicalAdapter.clear();
            }
        } catch (Exception ignored){ }
```

All'interno della classe ShadowingModelFunction sono inoltre dichiarate i metodi onCreate(), onStart(), onStop() che permettono una gestione completa del ciclo di vita del processo di shadowing.

```
public class DemoShadowingFunction extends ShadowingModelFunction {

    @Override
    public void onCreate() { }
```

```
5
6       @Override
7       public void onStart() { }
8
9       @Override
10      public void onStop() { }
11
12  }
```

Il codice fornito è un chiaro esempio di come il DT è un'entità che può estendere le funzioni del PO e offrirle tramite API. Difatti, nell'esempio proposto, la shadowing fucntion non si limita a salvare le proprietà osservate dalle interfacce fisiche ma definisce anche delle nuove proprietà (averageProperty) che permettono al DT di salvare informazioni riguardanti la media dei valori ricevuti.

3.5 Physical Adapter

Per mostrare la dinamicità e la flessibilità fornita dal framework OSGi, all'interno dell'applicativo sono stati sviluppati 2 bundle, del tutto speculari, che implementano le funzionalità e gestiscono il ciclo di vita di un PhysicalAdapter; l'unica differenza è nella proprietà monitorata dall'interfaccia fisica. Per questo motivo, all'interno di questo capitolo, l'attenzione della nostra analisi sarà posta su uno solo dei 2 bundle: il DemoPhysicalAdapter che monitora i valori di temperatura da un dispositivo IoT fittizio.
Il bundle è composto, come quasi tutti i bundle del progetto, da 2 principali classi:

- DemoPhysicalAdpater: classe che estende Physical Adapter, classe della libreria WLDT che definisce i metodi e gli attributi di un'interfaccia fisica.
- DemoPhysicalAdapterActivator: classe che funge da bundle activator.

Incominciamo a parlare del bundle activator, mostrandone prima di tutto il codice.

```
1  public class DemoPhysicalAdapterActivator implements BundleActivator {
2      @Override
3      public void start(BundleContext context) throws Exception {
4
5          IotDevice iotDevice = new IotDevice("Temperature", 20.0);
6          physicalAdapter = new DemoPhysicalAdapter("1", iotDevice);
7
8          serviceTracker = new ServiceTracker(context, DigitalTwin.class.
       getName(), new ServiceTrackerCustomizer() {
9              @Override
10             public Object addingService(ServiceReference reference) {
11                 ServiceRegistration<?> serviceRegistration = context.
       registerService(PhysicalAdapter.class.getName(), physicalAdapter,null);
12                 digitalTwin = (DigitalTwin) context.getService(reference);
13                 physicalAdapter.onAdapterCreate();
14                 return serviceRegistration;
15             }
16             @Override
17             public void removedService(ServiceReference reference, Object
       service) {
```

```
18                  physicalAdapter.onAdapterStop();
19                  ServiceRegistration<?> registration = (ServiceRegistration)
     service;
20                  registration.unregister();
21                  context.ungetService(reference);
22              }
23          });
24
25          serviceTracker.open();
26      }
27  }
```

All'avvio del bundle verrá istanziato un oggetto di tipo IotDevice, il quale sarà il dispositivo fittizio dal quale monitorare i valori di temperatura (generati randomicamente). Successivamente verrá istanziato un oggetto di tipo DemoPhysicalAdapter che rappresenta il componente, interno al bundle, che gestisce le operazioni e le funzionalità svolte dalla interfaccia fisica. Infine verrá istanziato ed aperto un ServiceTracker per tracciare un servizio di tipo DigitalTwin. Come possiamo notare dalle funzioni all'interno del ServiceTracker, il PhysicalAdapter verrà avviato e registrato come servizio, quindi reso visibile a tutti gli altri bundle, solo quando un servizio di tipo DigitalTwin sarà registrato all'interno del registro dei servizi del framework OSGi. Inoltre, possiamo vedere dal metodo removedService, che in caso di rimozione di un servizio di tipo DigitalTwin, il PhysicalAdapter verrà stoppato ed il servizio verrà rimosso dal registro dei servizi. Questo processo crea un forte legame tra i bundle delle interfacce fisiche ed bundle del DT; in particolar modo l'interfaccia fisica sarà in funzione e sarà connessa al DT fino a quando sará disponibile un servizio di tipo DigitalTwin.

L'altra classe interna al bundle é, come già detto, DemoPhysicalAdapter, la quale è un'estensione della classe PhysicalAdapter fornita dalla libreria WLDT. Quest'ultima é la classe responsabile della gestione delle operazioni e del ciclo di vita dell'interfaccia fisica; in particolar modo, come possiamo vedere dal codice sottostante, la classe definisce tutta una serie di metodi per notificare gli eventi occorsi sull'interfaccia (stato Bound ed eventuali aggiornamenti dei valori monitorati) e per gestire l'intero ciclo di vita dell'interfaccia fisica. Per ricevere aggiornamenti dei valori monitorati dal nostro dispositivo IoT abbiamo utilizzo l'Observer pattern, con il PhysicalAdapter che interpreta il ruolo di observer e con l'IotDevice che interpreta il ruolo di subject e che quindi invia, in caso di cambio di stato, notifiche a tutti gli observer registrati.

```
1  public abstract class PhysicalAdapter implements Observer{
2
3      public PhysicalAdapter(String ID, IotDevice iotDevice) {
4          this.ID = ID;
5          this.iotDevice = iotDevice;
```

```
        this.iotDevice.registerObserver(this);
    }

    @Override
    public void update(String property, Double value, Boolean bound) {
        PhysicalAssetDescription pad = new PhysicalAssetDescription();
        pad.getProperties().add(new PhysicalAssetProperty<>(property, value
    ));
        setPhysicalAssetDescription(pad);
        if (!bound){
            notifyPhysicalAdapterUpdate(getPhysicalAssetDescription());
        }else {
            notifyPhysicalAdapterBound(getPhysicalAssetDescription());
        }
    }

    protected void notifyPhysicalAdapterBound(PhysicalAssetDescription
    physicalAssetDescription) {
        if(getPhysicalAdapterListener() != null)
            getPhysicalAdapterListener().onPhysicalAdapterBound(getID(),
    this.physicalAssetDescription);
    }

    protected void notifyPhysicalAdapterUpdate(PhysicalAssetDescription
    physicalAssetDescription) {
        if(getPhysicalAdapterListener() != null)
            getPhysicalAdapterListener().onPhysicalAdapterUpdate(getID(),
    physicalAssetDescription);
    }
}
```

La maggior parte del processo per l'invio di aggiornamenti al DigitalTwin è svolto dalla classe appena descritta, ed é per questo motivo che all'interno della classe DemoPhysicalAdapter dovremmo semplicemente definire le funzioni per la gestione del ciclo di vita (onAdapterCreate(),onAdapterStart(), onAdapterStop()). Per simulare il ciclo di vita del Physical Adapter abbiamo utilizzato un thread, che se avviato aggiorna, ogni 5 secondi, il valore monitorato dal dispositivo IoT.

```
public class DemoPhysicalAdapter extends PhysicalAdapter {

    private final Thread thread;

    public DemoPhysicalAdapter(String ID, IotDevice iotDevice) {
        thread = new Thread(() -> {
            try {
                while (!Thread.currentThread().isInterrupted()){
                    iotDevice.setValue(iotDevice.getValue() + new Random().
    nextDouble(-1,1));
                    Thread.sleep(5000);
                }
            } catch (Exception ignored){ }
        });
    }

    @Override
    public void onAdapterStart() {
        thread.start();
    }

    @Override
```

```
22      public void onAdapterStop() {
23          if(thread.isAlive()){
24              thread.interrupt();
25          }
26      }
27
28      @Override
29      public void onAdapterCreate() {
30          getIotDevice().generatePad();
31          onAdapterStart();
32      }
33  }
```

3.6 Digital Adapter

Allo stesso modo dei PhysicalAdapter, per testare la dinamicità e la flessibilità fornita dal framework OSGi sono stati sviluppati due bundle DemoDigitalAdapter. A differenza, però, delle interfacce fisiche i bundle rappresentati le interfacce digitali sono sostanzialmente diversi e per questo motivo verranno trattati separatamente. L'unico punto in comune tra i due bundle è rappresentato dal bundle activator, uguale in entrambi i casi. Il funzionamento di quest'ultimo, esposto nel codice sottostante, é simile a quello del bundle activator delle interfacce fisiche. Infatti, anche in questo caso, viene utilizzato un oggetto di tipo DigitalAdapter, classe che definisce le operazioni e che gestisce il ciclo di vita delle interfacce digitali, ed un ServiceTracker per tracciare un servizio di tipo DigitalTwin. Essendo l'interfaccia digitale il componente complementare all'interfaccia fisica, il comportamento sará del tutto speculare; difatti verrà avviato e verrà registrato un servizio di tipo DigitalAdapter solo quando sarà disponibile un servizio di tipo DigitalTwin ed invece sarà stoppato e rimosso il servizio DigitalAdapter quando non sarà più disponibile il servizio DigitalTwin.

```
1  public void start(BundleContext context) throws Exception {
2      this.context = context;
3      digitalAdapter = new DemoDigitalAdapter("1");
4
5      serviceTracker = new ServiceTracker<>(this.context, DigitalTwin.class.
       getName(), new ServiceTrackerCustomizer<>() {
6          @Override
7          public Object addingService(ServiceReference<Object> reference) {
8              ServiceRegistration<?> serviceRegistration = context.
       registerService(DigitalAdapter.class.getName(),digitalAdapter, null);
9              digitalTwin = (DigitalTwin) context.getService(reference);
10             digitalAdapter.onAdapterStart();
11             return serviceRegistration;
12         }
13
14         @Override
15         public void removedService(ServiceReference<Object> reference,
       Object service) {
16             digitalAdapter.onAdapterStop();
17             ServiceRegistration<?> registration = (ServiceRegistration)
       service;
```

```
            registration.unregister();
            context.ungetService(reference);
        }
    });
    serviceTracker.open();
}
```

Come descritto in precedenza, le interfacce digitali ricevono aggiornamenti sullo stato del digital twin solo in seguito all'esecuzione del processo di shadowing. Per questo motivo all'interno della classe DigitalAdapter, messa a disposizione dalla libreria WLDT, troviamo una serie di metodi e attributi per la gestione del ciclo di vita e per ricevere notifiche sul cambio di stato del DT.

Il primo dei due bundle, come descritto dal codice sottostante, implementa una semplicissima interfaccia digitale che alla notifica di un determinato evento (creazione, modifica o eliminazione di una proprietá) ne stampa il contenuto.

```
public class DemoDigitalAdapter extends DigitalAdapter {

    @Override
    public void onStateChangePropertyCreated(DigitalTwinStateProperty<?>
    digitalTwinStateProperty) {
        System.out.println("(Digital Adapter stop " + getID() + "):
    Property created { name :" + digitalTwinStateProperty.getKey() + ";
    value: " + digitalTwinStateProperty.getValue() + " }");
    }

    @Override
    public void onStateChangePropertyUpdate(DigitalTwinStateProperty<?>
    digitalTwinStateProperty) {
        System.out.println("(Digital Adapter " + getID() + "): Property
    update { name :" + digitalTwinStateProperty.getKey() + "; value: " +
    digitalTwinStateProperty.getValue() + " }");
    }

    @Override
    public void onStateChangePropertyRemove(DigitalTwinStateProperty<?>
    digitalTwinStateProperty) {
        System.out.println("(Digital Adapter " + getID() + "): Property
    removed { name :" + digitalTwinStateProperty.getKey() + "; value: " +
    digitalTwinStateProperty.getValue() + " }");
    }
}
```

3.6.1 MQTT Digital Adapter

Il secondo bundle é, invece, un esempio di Digital Adapter complesso, cioè di un Adapter che comunica con le applicazioni esterne utilizzando un determinato protocollo applicativo. In particolar modo, all'interno del progetto, è stata sviluppata un'interfaccia digitale che fa uso del protocollo MQTT (Message Queuing Telemetry Transport). Quest'ultimo è un protocollo applicativo di messaggistica leggero di tipo publish-subscribe, molto utilizzato all'interno del mondo IoT per via del basso

impatto energetico e per l'utilizzo minimale di banda. Il pattern publish-subscribe richiede un broker di messaggistica, responsabile della distribuzione dei messaggi ai client destinatari.

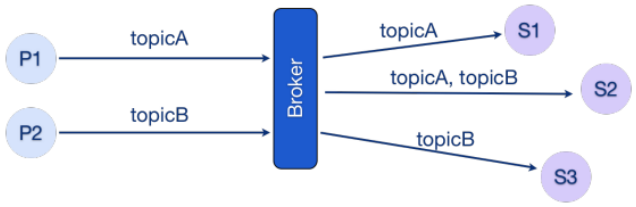


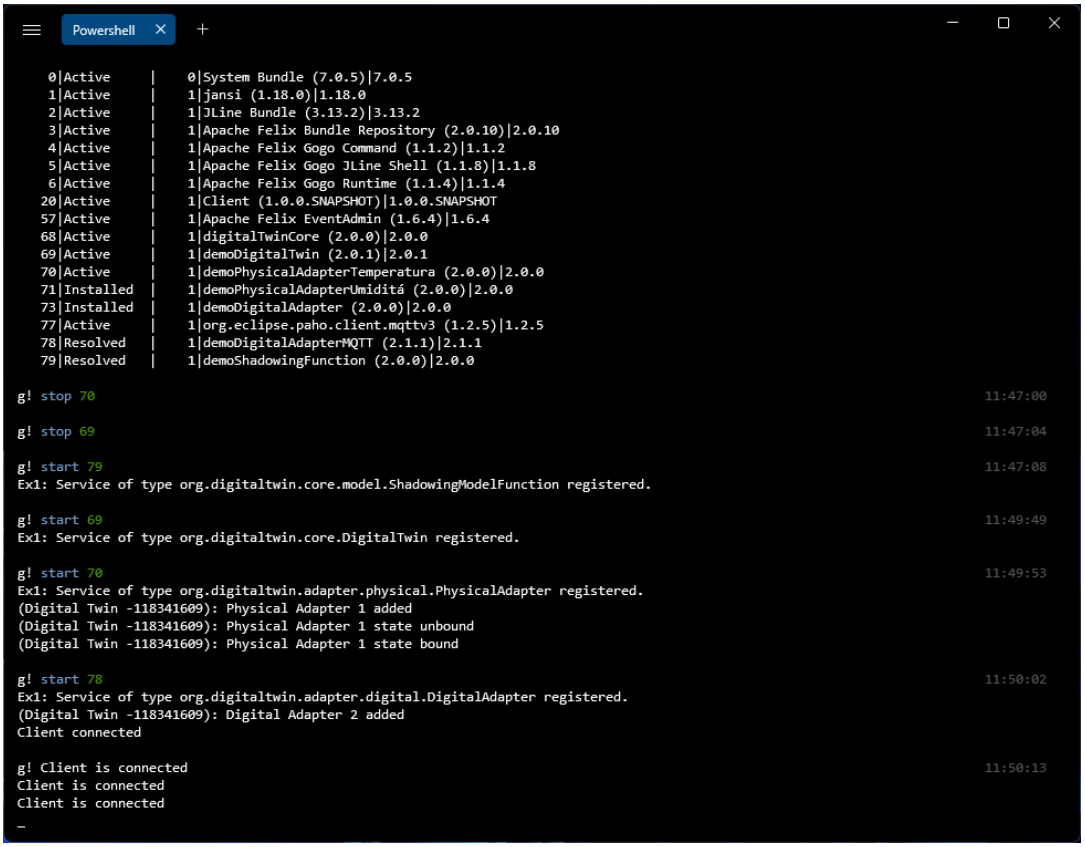
Figura 3.3: Protocollo MQTT

All'interno del bundle, come riportato dal codice sottostante, è stato utilizzato un broker di messaggi free ed open source chiamato Mosquitto.
Nel nostro caso, il bundle svolgerà il ruolo di publisher, cercando, al suo avvio, di connettersi al broker e pubblicando dei topic rappresentanti le proprietà interne allo stato del DT. Infatti le funzioni di OnStateChangePropertyCreated e OnStateChangePropertyUpdate permetteranno all'interfaccia digitale, alla creazione o modifica di una proprietà all'interno dello stato del DT, di pubblicare o aggiornare un topic presente nel broker rappresentante la proprietà ed il suo valore.

```
1  public class MQTTDigitalAdapter extends DigitalAdapter {
2
3      private IMqttClient publisher;
4      private static final String TOPIC = "digitaltwin/properties/";
5
6      @Override
7      public void onAdapterStart() {
8          String publisherId = UUID.randomUUID().toString();
9          publisher = new MqttClient("tcp://test.mosquitto.org:1883",
       publisherId);
10         try {
11             publisher.connect();
12         } catch (MqttException e) {
13             throw new RuntimeException(e);
14         }
15     }
16
17     @Override
18     public void onStateChangePropertyCreated(DigitalTwinStateProperty<?>
       digitalTwinStateProperty) {
19         if (publisher.isConnected()){
20             byte[] payload = String.format("T:%04.2f",
       digitalTwinStateProperty.getValue()).getBytes();
21             MqttMessage msg = new MqttMessage(payload);
22             try {
23                 publisher.publish(TOPIC + digitalTwinStateProperty.getKey()
       ,msg);
24             } catch (MqttException e) {
25                 throw new RuntimeException(e);
26             }
27         }
28     }
29
```

```
30      @Override
31      public void onStateChangePropertyUpdate(DigitalTwinStateProperty<?>
        digitalTwinStateProperty) {
32        // uguale a onStateChangePropertyCreated
33      }
34  }
```

Per testare l'effettivo funzionamento dell'interfaccia digitale MQTT, abbiamo utilizzato un applicativo chiamato MQTTExplorer che permette di visualizzare tutti i topic pubblicati all'interno di un broker. Come mostrato dalla figura 4.4, all'avvio di un DT, al quale sono connessi un'interfaccia fisica che monitora un valore di temperatura e un'interfaccia digitale che implementa il protocollo MQTT, potremmo visualizzare all'interno del broker con topic digitaltwin/properties/temperature i valori di temperatura letti dal PhysicalAdapter, inviati al DigitalTwin ed esposti dal DigitalAdapter.



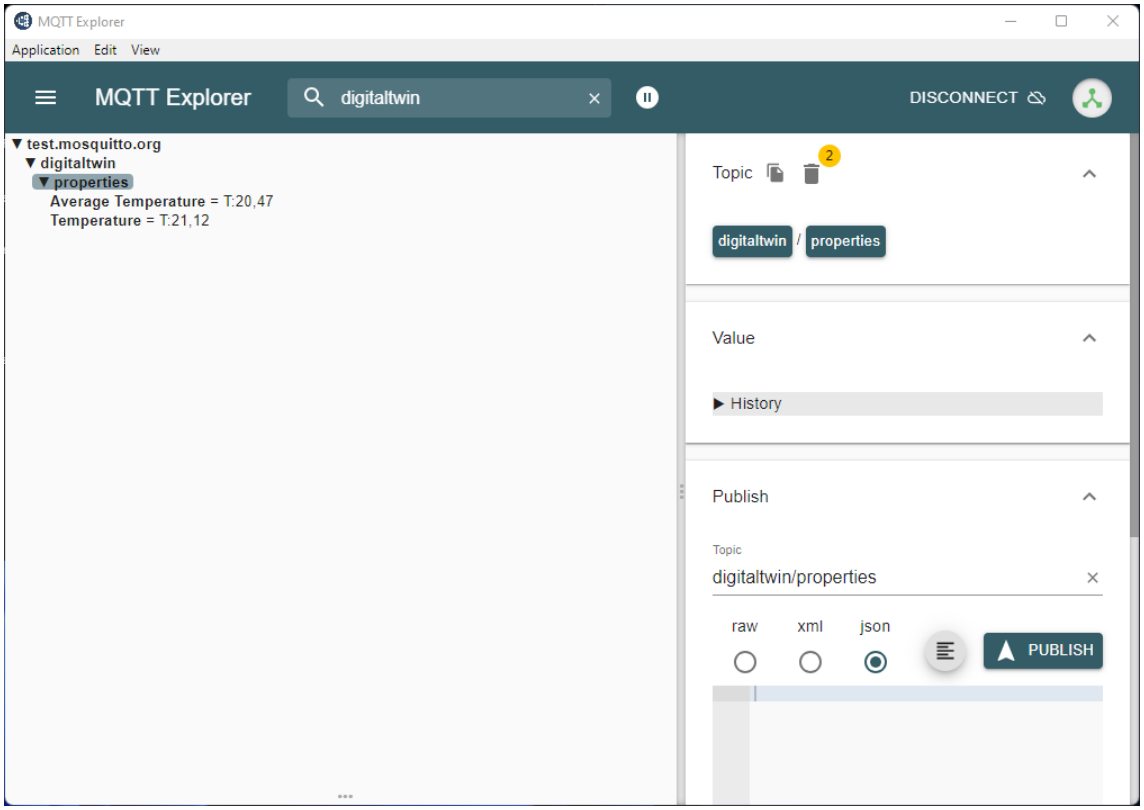


Figura 3.4: Testing DigitalAdapterMQTT

Capitolo 4

Conclusioni

L'applicazione, descritta all'interno di questa elaborato, ovviamente non è ultimata ma è solamente un punto di partenza per l'introduzione della programmazione modulare all'interno della libreria WLDT. Difatti il progetto può essere, sicuramente, migliorato ed ampliato sviluppando diverse tipologie di DigitalAdapter e PhysicalAdapter "complessi" (HTTPDigitalAdapter e MQTTPhysicalAdapter). Inoltre la flessibilità e la dinamicità tra le varie componenti del Digital Twin può essere, ulteriormente, migliorata riducendo la dipendenza tra il bundle rappresentante il modello del DT ed il bundle della ShadowingFunction, permettendo al Digital Twin di utilizzare più ShadowingFunction senza resettare lo stato interno. Infine potrebbe essere sviluppato anche un ulteriore bundle, che attraverso l'utilizzo di un event bus, gestisca la comunicazione tra i vari componenti del Digital Twin.

Bibliografia

[1] Neil Bartlett. *OSGi In Practice*. 2009.

[2] Walid Gédéon. *OSGi and Apache Felix 3.0*. PACKT publishing, 2010.

[3] Paolo Bellavista, Nicola Bicocchi, Mattia Fogli, Carlo Giannelli, Marco Mamei, and Marco Picone. *Requirements and Design Patterns for Adaptive, Autonomous, and Context-aware Digital Twins in Industry 4.0 Digital Factories*. 2023.

[4] Roberto Minerva, Gyu Myoung Lee, and Noël Crespi. *Digital Twin in the IoT Context: A Survey on Technical Features, Scenarios, and Architectural Models*. 2020.

[5] Alessandro Ricci, Angelo Croatti, Stefano Mariani, Sara Montagna, and Marco Picone. *Web of Digital Twins*. 2022.

[6] Marco Picone. *WLDT: A general purpose library to build IoT digital twins*. 2021.

[7] Marco Picone. *IoT Pub/Sub Protocols, MQTT e AMQP*. uniMORE, 2022.

[8] Emanuele Barrano. *OSGi, Java e la programmazione a bundle*. 2010.

[9] Laura Zanotti. *Digital Twin: cos'è e come funziona il modello del gemello digitale*. 2023.

[10] Eclipse. *Documentazione Eclipse Ditto*. 2023.

[11] Microsoft. *Documentazione Microsoft Azure Digital Twins*. 2023.

[12] Bosch Digital Blog. *Eclipse Ditto: How digital twins boost development in the IoT*. 2012.