ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Analisi e sviluppo di architetture a microservizi ad eventi per il design di Digital Twins

Tesi di laurea in:
Pervasive Computing

Relatore
Prof. Alessandro Ricci

Correlatori
Prof. Marco Picone
Dott. Samuele Burattini

Candidato
Andrea Acampora

IV Sessione di Laurea
Anno Accademico 2022-2023

Sommario

La continua evoluzione delle architetture software ha portato a una crescente adozione di paradigmi agili e flessibili, tra i quali sono emersi i microservizi. Questi si sono affermati come un pattern architetturale di riferimento per i sistemi distribuiti, offrendo una serie di vantaggi chiave come modularità, scalabilità e manutenibilità. Inoltre, l'integrazione con le moderne architetture ad eventi consente ai microservizi di acquisire ulteriori caratteristiche distintive.

Il presente progetto di tesi si propone di esplorare l'integrazione di tre concetti chiave nell'ambito delle moderne architetture software: microservizi, eventi e Digital Twin. In particolare, verrà mostrato come l'utilizzo di queste tipologie di architetture possa agevolare il design e lo sviluppo di Digital Twin o, nello specifico, lo sviluppo di piattaforme che supportano l'esecuzione di ecosistemi di Digital Twin.

Il primo capitolo della tesi ha l'obbiettivo di fornire un'introduzione ai principali pattern architetturali, delineando le caratteristiche delle architetture monolitiche, orientate ai servizi e ad eventi. In seguito, viene effettuata una panoramica sui Digital Twin, definendone le caratteristiche, i contesti e gli ambiti di applicazione e fornendo così un quadro completo di questa innovativa tecnologia. Il focus si sposta poi sul tema centrale della tesi, ossia i microservizi ad eventi, esplorando un nuovo paradigma di comunicazione. Il terzo capitolo introduce il concetto di "Web of Digital Twins", una proposta presente in letteratura che estende il paradigma con lo scopo di virtualizzare complesse realtà di asset interconnessi, appartenenti anche a domini e organizzazioni differenti. In questo contesto, viene introdotto e analizzato un framework a supporto di questa visione in modo da fornire il contesto adeguato per la comprensione del caso di studio svolto. Il quarto capitolo presenta un caso di studio concreto: verrà dimostrato come le architetture orientate agli eventi possano essere utilizzate nel design di piattaforme che supportano l'esecuzione di ecosistemi di Digital Twin. Attraverso esempi dettagliati, si evidenziano scelte architetturali, tecnologie impiegate e le sfide affrontate durante lo sviluppo. Nel quinto capitolo sono riportati alcuni dettagli implementativi dei diversi componenti software realizzati nel corso del progetto. Infine, l'ultimo capitolo contiene le conclusioni ed i risultati ottenuti.

Indice

Sommario **iii**

1 Introduzione **1**
1.1 Introduzione ai pattern architetturali 2
1.1.1 Architetture monolitiche 2
1.1.2 Architetture orientate ai servizi 5
1.1.3 Architetture ad eventi 17
1.2 Panoramica sui Digital Twins 20
1.2.1 Definizioni e caratterstiche 20
1.2.2 Contesto e ambiti di applicazione 30
1.2.3 Strumenti e tecnologie 33

2 Microservizi ad eventi **39**
2.1 Un nuovo modo di comunicare 39
2.2 Design e contratto degli eventi 43
2.3 Canali di comunicazione 49
2.4 Pattern e metodologie 52
2.5 Limiti e problematiche 56

3 Web of Digital Twins **59**
3.1 Panoramica 59
3.2 Modello dei Digital Twin 61
3.3 White Label Digital Twin Framework 65
3.3.1 Meta-modello 66
3.3.2 Componenti principali 67
3.3.3 Limiti e problematiche 70

4 Progettazione di un'architettura event-driven per il design di Digital Twins **73**
4.1 Requisiti dell'architettura 73
4.2 Core & Plugin 76

4.3 Design architetturale 80
4.4 Esempio d'uso 83

5 Implementazione **89**
5.1 Processo di sviluppo 89
5.2 Implementazione Event Bus interno 90
5.3 Implementazione Event Bus remoto 93
5.4 Testing 96

6 Conclusioni **99**

Bibliografia **101**

Ringraziamenti **105**

Elenco delle figure

1.1 Schema funzionamento architettura monolitica. 3
1.2 Interoperabilità di un Enterprise Service Bus. 9
1.3 Diffrenze tra architettura monolitica e microservizi. 12
1.4 Schema funzionamento architettura ed eventi. 18
1.5 Schema del funzionamento di AWS IoT Twin Maker. 34
1.6 Schema del funzionamento di Eclipse Ditto. 35
1.7 Grafo di esempio di Azuere Digital Twins. 38

2.1 Topologia di esempio di un microservizio ad eventi. 42
2.2 Topologia di business di un'architettura a microservizi ad eventi. . . 43
2.3 Schema funzionamento di un Event Broker. 50
2.4 Schema funzionamento di un Message Broker. 51
2.5 Schema di funzionamento del pattern Event Sourcing. 53
2.6 Workflow di esempio del pattern Orchestration. 54
2.7 Workflow di esempio del pattern Coreography. 55

3.1 Layer presenti nella visione Web of Digital Twins. 60
3.2 Diagramma degli stati del ciclo di vita di un Digital Twin in WoDT. 64
3.3 Schema meta modello libreria White Label Digital Twins. 66
3.4 Componenti principali libreria White Label Digital Twins. 68

4.1 Componenti presenti nell'architettura proposta. 77
4.2 Schema della comunicazione tra Core e Plugin mediante l'Event Bus. 77
4.3 Diagramma degli stati del componente Core. 78
4.4 Diagramma degli stati del componente Plugin. 79
4.5 Diagramma delle classi dell'architettura realizzata. 81
4.6 Schema dei componenti nel progetto di esempio Smart Home. . . . 85
4.7 Esempio di un Digital Twin di una Smart Home. 87

5.1 Implementazione Event Bus interno nella libreria WLDT. 90
5.2 Esempio Reactive Stream con operatore filter. 91
5.3 Schema di funzionamento della tecnologia Redis Pub/Sub. 94

Elenco dei listati

2.1 Esempio di evento in formato Avro. 48
2.2 Esempio di evento in formato Protobuf. 48
2.3 Esempio di evento in formato Json. 49
5.1 Implementazione reattiva dell'Event Bus interno. 92
5.2 Implementazione del client di Redis tramite la liberia Lettuce. . . . 95
5.3 Esempio di Unit Tests per la comunicazione tra Core e Plugin. . . . 97

Capitolo 1

Introduzione

Le architetture a microservizi ad eventi e il paradigma dei Digital Twin stanno emergendo come tecnologie chiave per la progettazione di sistemi software complessi. Infatti, le architetture a microservizi suddividono un'applicazione in piccoli servizi indipendenti e autonomi e questi servizi comunicano tra loro in modo asincrono tramite eventi, favorendo la scalabilità, la resilienza e la flessibilità. D'altro canto, il paradigma dei Digital Twin, che consiste nel clonare un asset fisico nella sua controparte digitale, ha trovato applicazione in molteplici ambiti: da quello industriale fino alle smart city e all'healthcare.

Questa tesi si propone di esplorare l'integrazione di queste due tecnologie, con particolare attenzione ai vantaggi ed alle sfide che ne derivano. In particolare, verrà mostrato come l'utilizzo di queste tipologie di architetture possa agevolare il design e lo sviluppo di Digital Twin.

In questo primo capitolo viene fornita inizialmente un'introduzione ai principali pattern architetturali ossia le architetture monolitiche, orientate ai servizi e ad eventi. Successivamente, viene fornita una panoramica sul paradigma dei Digital Twin spiegandone le caratteristiche principali, gli ambiti di applicazione e gli strumenti a supporto. Entrambi i concetti risulteranno indispensabili per la comprensione del caso di studio che verrà spiegato nei capitoli successivi.

1.1 Introduzione ai pattern architetturali

L'evoluzione della tecnologia e la crescente complessità delle applicazioni hanno portato alla ricerca di modelli architetturali che rispondano alle esigenze di flessibilità, scalabilità e manutenibilità dei sistemi software moderni. In questo contesto, diversi pattern architetturali sono emersi come risposte strategiche alle diverse sfide progettuali. Tra i principali, troviamo le architetture monolitiche, orientate ai servizi, ad eventi e infine i microservizi. Questa sezione fornirà una panoramica introduttiva su questi pattern, evidenziandone le caratteristiche principali e le situazioni in cui risultano più adatti.

1.1.1 Architetture monolitiche

Il tradizionale approccio alla creazione delle applicazioni è di tipo monolitico e prevede che tutte le funzionalità e i servizi di un'applicazione siano bloccati insieme e funzionino come una singola unità. Quando l'applicazione viene aggiunta o migliorata in qualsiasi modo, la complessità dell'architettura aumenta, insieme alla difficoltà di ottimizzare ogni singola funzione dell'applicazione senza suddividerla in più parti [SA22]. Inoltre, se è necessario dimensionare uno dei processi che compongono l'applicazione, sarà necessario dimensionarla nella sua interezza. Come si può notare in Figura 1.1 presente nell'articolo al seguente link, nelle architetture monolitiche tutti i processi sono strettamente collegati tra loro e vengono eseguiti come un singolo servizio. Ciò significa che se un processo dell'applicazione sperimenta un picco nella richiesta, è necessario ridimensionare l'intera architettura. Aggiungere o migliorare una funzionalità dell'applicazione monolitica diventa più complesso, in quanto sarà necessario aumentare la base di codice. Tale complessità limita la sperimentazione e rende più difficile implementare nuove idee. Le architetture monolitiche rappresentano un ulteriore rischio per la disponibilità dell'applicazione, poiché la presenza di numerosi processi dipendenti e strettamente collegati aumenta l'impatto di un errore in un singolo processo.

Sebbene le architetture monolitiche godano di una pessima reputazione, ci sono dei vantaggi nel mantenere un'applicazione con un'unica base di codice. Nel

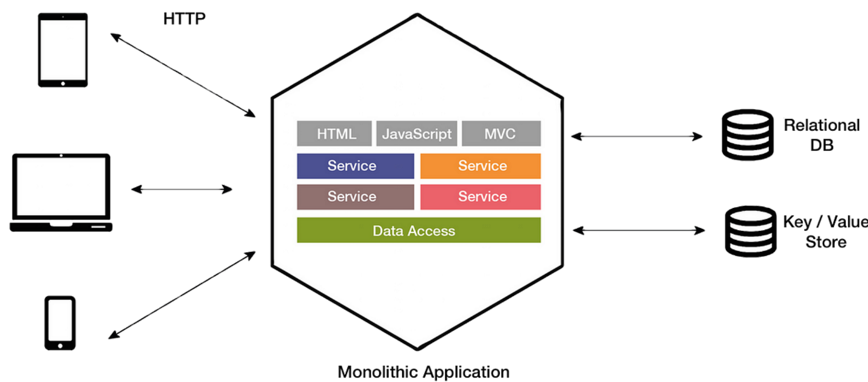

Figura 1.1: Schema funzionamento architettura monolitica.

seguente elenco verranno illustrati i principali vantaggi delle architetture monolitiche:

- *Chiarezza dei processi di business*: con una singola base di codice, è possibile ispezionare rapidamente il flusso end-to-end. Infatti, è possibile senza sforzo trovare qualsiasi funzionalità si desidera poiché esse sono in un unico repository. È anche più facile vedere gli impatti dei nuovi sviluppi poiché è possibile vedere le dipendenze di quel flusso. Il flusso aziendale può diventare difficile da comprendere con le architetture basate sugli eventi a causa dell'interazione asincrona tra più servizi. Per capire i processi di business , è necessario comprendere il flusso di eventi tra ciascun servizio. Risulta quindi sicuramente più complicato comprendere il quadro generale e quest'ultimo è molto importante soprattutto nell'aggiunta di nuove feature, sia per tenere conto dei possibili impatti sia per capire se lo sviluppo è quello corretto per la funzionalità. In un monolite, invece, è possibile capire fin da subito l'impatto di nuove features nel flusso aziendale complessivo.

- *Nessun sovraccarico di rete e dipendenze esterne limitate*: tutti i diversi moduli dell'architettura comunicano tra loro direttamente all'interno dell'applicazione. Non ci sono chiamate tramite API esterne o broker di eventi. Questa caratteristica può godere di un aumento delle prestazioni poiché non

esiste sovraccarico della rete. Inoltre, non è necessario occuparsi della gestione delle API o del controllo delle versioni degli eventi. In aggiunta, se c'è la necessità di fare una modifica sostanziale, possiamo farlo in un'unica release in quanto tutte le dipendenze sono all'interno dell'applicazione. Questa caratteristica semplifica lo sviluppo e il rilascio di funzionalità che necessitano di cambiamenti più importanti.

- *Validazione locale*: in un'architettura monolitica è possibile eseguire localmente l'intero ambiente. Spesso si può aggiungere e convalidare una nuova feature eseguendo la singola applicazione. Tuttavia, su un'architettura basata sugli eventi spesso è difficile eseguire l'intero flusso di lavoro localmente a causa della complessità dell'elevato numero di servizi, ciascuno con le proprie configurazioni specifiche e dipendenze. La maggior parte dei servizi tipicamente non si conoscono nemmeno poiché appartengono ad altri team.

- *Riuso del codice*: poiché tutto il codice è all'interno dell'applicazione, è facile riutilizzarlo e sviluppare codice a partire dalle funzionalità esistenti. In un'architettura basata sugli eventi, ogni servizio è isolato con la propria base di codice. Infatti, molti servizi avranno bisogno di funzionalità simili e, ovviamente, non è possibile condividere il codice.

- *Monitoraggio*: il monitoraggio di un architettura monolitica è semplice poiché si tratta di una sola applicazione e di conseguenza anche l'individuazione e risoluzione di un problema risulta più semplice. In un'architettura guidata dagli eventi si richiede una strategia completamente diversa per il monitoraggio. C'è infatti bisogno di un'infrastruttura preesistente per gestire tutto il traffico dei microservizi.

Dopo aver descritto i principali vantaggi, in questa sottosezione si discuteranno i principali problemi che vengono associati alle architetture monolitiche e come possono limitare la crescita del business. Esse possono rappresentare una soluzione adeguata in diversi contesti, tuttavia, man mano che le aziende crescono, i loro handicap diventano una preoccupazione crescente. Il problema principale è che, col tempo, questi software diventano fortemente complessi e accoppiati. Poiché

tutte le funzionalità si trovano all'interno di un'unica applicazione, è relativamente semplice compromettere i confini di ciascun dominio. Una singola modifica potrebbe influenzare diverse parti del sistema e provocare impatti imprevedibili [GZ20]. Inoltre, man mano che il team di sviluppo cresce, diventa sempre più difficile lavorare su una singola applicazione. Anche con moduli chiaramente definiti, lo sviluppo richiede ancora comunicazione e allineamento e l'implementazione dell'applicazione necessita di coordinamento tra i diversi team. I branch delle nuove funzionalità spesso portano a grandi fusioni con un elevato numero di conflitti. Sebbene lo scaling sia possibile, come accennato nella sezione precedente, è limitato alla singola applicazione. Ciò implica che l'applicazione abbia i meccanismi per gestire la concorrenza con diverse istanze. Quando questo non succede, vengono richiesti cambiamenti importanti per essere in grado di gestire richieste concorrenti. Quando scaliamo un'applicazione monolitica, viene scalata l'intera applicazione. Tipicamente, solo uno o parte dei moduli necessitano di scalare ma poiché ogni modulo è legato insieme, l'unica opzione è ridimensionare l'intera applicazione. Inoltre, nella maggior parte dei casi queste applicazioni durano diversi anni e, nel frattempo, alcune delle tecnologie potrebbero essere state interrotte. Infatti, se l'applicazione utilizza una tecnologia che non è più supportata dall'azienda che l'ha realizzata, è necessario cambiare l'intero stack tecnologico.

1.1.2 Architetture orientate ai servizi

L'evoluzione delle esigenze aziendali e delle applicazioni software ha portato a una necessità crescente di superare le limitazioni intrinseche delle architetture monolitiche. Esse, sebbene siano state una soluzione conveniente inizialmente, possono diventare complesse e difficili da gestire man mano che le applicazioni crescono in dimensioni e complessità. La transizione verso architetture più modulari e flessibili è stata dettata dalla necessità di affrontare questi vincoli e di consentire una maggiore scalabilità, riusabilità e manutenibilità. In questo contesto, le architetture orientate ai servizi, chiamate anche *Service oriented Architectures (SOA)*, si sono affermate come un modello che offre una soluzione alle sfide presentate dalle architetture monolitiche. L'architettura orientata ai servizi è un metodo di sviluppo del software che utilizza componenti chiamati servizi per creare applicazioni

aziendali [ES09]. Ogni servizio fornisce una funzionalità aziendale e i servizi possono comunicare tra loro attraverso piattaforme e lingue diverse. Gli sviluppatori utilizzano le SOA per riutilizzare i servizi in sistemi diversi o per combinare più servizi indipendenti ed eseguire compiti complessi. Ad esempio, diversi processi aziendali in un'organizzazione richiedono la funzionalità di autenticazione degli utenti. Invece di riscrivere il codice di autenticazione per tutti i processi aziendali, è possibile creare un unico servizio di autenticazione e riutilizzarlo per tutte le applicazioni. Allo stesso modo, la maggior parte dei sistemi di un'organizzazione sanitaria, come i sistemi di gestione dei pazienti e i sistemi di cartelle cliniche elettroniche (EHR), devono registrare i pazienti. Questi sistemi possono ricorrere a un unico servizio comune per eseguire la registrazione del paziente.

L'architettura orientata ai servizi presenta numerosi vantaggi rispetto alle tradizionali architetture monolitiche in cui tutti i processi vengono eseguiti come una singola unità. Alcuni dei principali vantaggi delle SOA sono:

- *Time-to-market ridotto*: gli sviluppatori riutilizzano i servizi in diversi processi aziendali per risparmiare tempo e costi. Possono assemblare applicazioni molto più velocemente con le SOA che scrivendo codici ed eseguendo integrazioni da zero.

- *Manutenzione efficiente*: è più facile creare, aggiornare ed eseguire il debug di piccoli servizi rispetto ai blocchi di codice di grandi dimensioni nelle applicazioni monolitiche. La modifica di qualsiasi servizio in una SOA non influisce sulla funzionalità complessiva del processo aziendale.

- *Indipendenza dei servizi*: ogni servizio all'interno di un'architettura orientata ai servizi è autonomo, indipendente dagli altri. Ciò consente una maggiore flessibilità nell'evoluzione, l'aggiornamento o la sostituzione di singoli servizi senza influire sull'intero sistema.

- *Maggiore adattabilità*: la SOA è più adattabile ai progressi tecnologici. È possibile modernizzare le applicazioni in modo efficiente ed economico. Ad esempio, le organizzazioni sanitarie possono utilizzare la funzionalità dei vecchi sistemi di cartelle cliniche elettroniche nelle nuove applicazioni basate su cloud.

Non esistono linee guida standard ben definite per l'implementazione dell'architettura orientata ai servizi. Tuttavia, alcuni principi base sono comuni a tutte le implementazioni di queste architetture:

- *Interoperabilità*: ciascun servizio include documenti descrittivi che specificano la funzionalità del servizio e i relativi termini e condizioni. Qualsiasi client può eseguire un servizio, indipendentemente dalla piattaforma o dal linguaggio di programmazione sottostante. Ad esempio, i processi aziendali possono utilizzare servizi scritti sia in Java che in Python. Poiché non esistono interazioni dirette, le modifiche in un servizio non influiscono sugli altri componenti che utilizzano il servizio.

- *Accoppiamento debole*: i servizi dovrebbero essere accoppiati debolmente, con la minor dipendenza possibile da risorse esterne come modelli di dati o sistemi informativi. Dovrebbero anche essere indipendenti senza mantenere alcuna informazione da sessioni o transazioni passate. In questo modo, se si modifica un servizio, non si avrà un impatto significativo sulle applicazioni client e sugli altri servizi che utilizzano il servizio.

- *Astrazione*: i client o gli utenti del servizio non devono conoscere la logica del codice del servizio o i dettagli di implementazione. Per loro, i servizi dovrebbero apparire come una scatola nera. I client ottengono le informazioni richieste su ciò che fa il servizio e su come utilizzarlo tramite contratti di servizio e altri documenti di descrizione del servizio.

- *Granularità*: i servizi dovrebbero avere una dimensione e un ambito appropriati, idealmente racchiudendo una discreta funzione aziendale per servizio. Gli sviluppatori possono quindi utilizzare più servizi per creare un servizio composito per l'esecuzione di operazioni complesse.

Protocolli di comunicazione

Nell'architettura orientata ai servizi, essi funzionano in modo indipendente e forniscono funzionalità o scambi di dati ai propri utenti. L'utente richiede informazioni e invia i dati di input al servizio. Il servizio elabora i dati, esegue l'attività e invia

una risposta. Ad esempio, se un'applicazione utilizza un servizio di autorizzazione, fornisce al servizio il nome utente e la password. Il servizio verifica il nome utente e la password e restituisce una risposta appropriata. Come riportato in [PH07], l'utilizzo di protocolli standardizzati facilita l'interazione e la cooperazione tra i servizi. Alcuni dei principali protocolli utilizzati nelle SOA sono:

1. *SOAP*: protocollo basato su XML utilizzato per la comunicazione tra applicazioni. Definisce un formato di messaggio standard e regole per la sua struttura, consentendo la trasmissione di informazioni strutturate tra servizi. Spesso utilizzato in scenari in cui è necessaria una comunicazione più formale e complessa, ad esempio in applicazioni enterprise.

2. *REST*: modello architetturale basato su principi, utilizza il protocollo HTTP per la comunicazione. Si basa su operazioni standard HTTP come GET, POST, PUT e DELETE per consentire l'accesso e la manipolazione delle risorse. È noto per la sua semplicità e leggibilità, ed è spesso utilizzato in scenari in cui la leggerezza e la scalabilità sono fondamentali.

3. *JMS (Java Message Service)*: standard di messaggistica per la comunicazione asincrona tra componenti software scritti in linguaggio Java. È spesso utilizzato in questi contesti per la gestione di code di messaggi e la comunicazione tra servizi distribuiti.

4. *AMQP (Advanced Message Queuing Protocol)*: protocollo di messaggistica avanzato, progettato per la trasmissione efficiente di messaggi tra applicazioni. Supporta la comunicazione asincrona e la gestione di code di messaggi, facilitando l'integrazione tra sistemi distribuiti.

Enterprise Service Bus

Quando le aziende hanno iniziato ad aggiungere più servizi ai loro sistemi, hanno avuto bisogno di eseguire integrazioni point-to-point per collegare le applicazioni in modo che potessero lavorare insieme in modo efficace. Le integrazioni manuali point-to-point sono impossibili da scalare, si rompono facilmente e richiedono tempo e risorse per la manutenzione, il che rallenta l'azienda e ne limita la velocità di

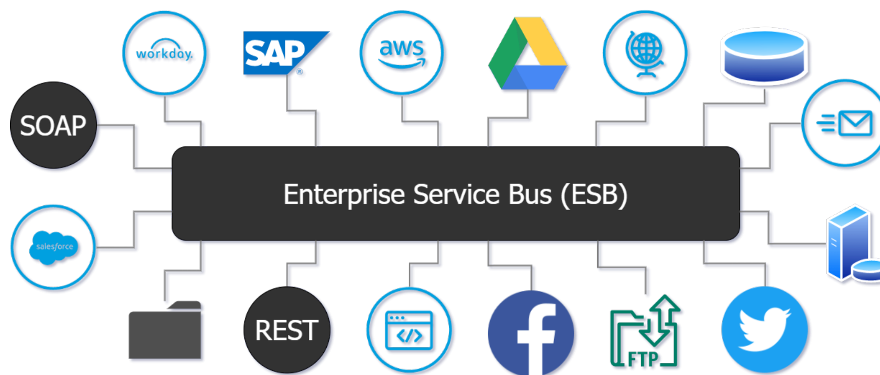

Figura 1.2: Interoperabilità di un Enterprise Service Bus.

commercializzazione. Per risolvere questo problema, come descritto in [AFA+20] le aziende hanno iniziato a utilizzare uno strumento di middleware chiamato *bus di servizi aziendali (ESB)* per fornire un modo centralizzato di gestire le integrazioni e la scalabilità. Un bus di servizio aziendale (ESB) è un software che è possibile utilizzare durante la comunicazione con un sistema che dispone di più servizi. Esso stabilisce la comunicazione tra servizi e utenti di servizi, indipendentemente dalla tecnologia. È possibile pensare a un ESB come a un servizio centralizzato che instrada le richieste di servizi al servizio appropriato. Inoltre, trasforma la richiesta in un formato accettabile per la piattaforma sottostante e il linguaggio di programmazione del servizio. Come si può notare in Figura 1.2, presente al seguente link, la maggior parte delle aziende ha un mix di sistemi on-premises, legacy e applicazioni cloud nel proprio ambiente ibrido. Un ESB crea le connessioni utilizzando un adattatore o un connettore, oppure l'API della nuova applicazione software. L'ESB gestisce vari formati di dati, esegue la trasformazione e l'instradamento dei dati, consentendo l'integrazione delle applicazioni e facilitando la scalabilità con l'aggiunta di altre applicazioni. Inoltre, esso elimina la necessità di integrazioni point-to-point e i relativi costi, tempi e rischi. L'utilizzo di un ESB per l'integrazione aziendale offre diversi vantaggi, tra cui i seguenti:

- *Governance e gestione centralizzate*: un ESB fornisce un punto di controllo e gestione centrale per l'integrazione aziendale. Ciò consente di gestire e applicare in modo coerente le politiche, le regole e i processi di integrazione

in tutta l'azienda e facilita il monitoraggio e la gestione dell'ambiente di integrazione.

- *Riusabilità ed estensibilità*: un ESB consente di sviluppare e riutilizzare componenti di integrazione, come connettori, adattatori e trasformazioni, in diverse applicazioni e servizi. Ciò può migliorare l'efficienza e la produttività dello sviluppo dell'integrazione e facilitare l'estensione delle capacità di integrazione del sistema.

- *Accoppiamento libero e interoperabilità*: un ESB promuove l'accoppiamento libero tra applicazioni e servizi, che possono essere sviluppati, distribuiti e gestiti in modo indipendente. Ciò migliora l'interoperabilità e la flessibilità del sistema e facilita l'aggiunta, la rimozione o la sostituzione di componenti senza che ciò influisca sull'intero sistema.

- *Miglioramento delle prestazioni e dell'affidabilità*: un ESB può migliorare le prestazioni e l'affidabilità dell'integrazione aziendale fornendo funzioni quali il buffering dei messaggi, il routing e la trasformazione, che possono ottimizzare il flusso di dati tra applicazioni e servizi. Ciò può ridurre l'impatto di guasti o colli di bottiglia e migliorare la disponibilità e l'affidabilità complessiva del sistema.

Gli ESB forniscono interoperabilità e supportano l'integrazione delle applicazioni e dei dati. Gli sviluppatori dedicano meno tempo all'integrazione e si concentrano sull'innovazione. Tuttavia, le aziende di oggi spesso scoprono che gli ESB non forniscono ancora la velocità e la stabilità necessarie in un ambiente sempre attivo. La modifica delle integrazioni in un ESB può destabilizzare le altre e gli aggiornamenti del middleware ESB devono essere testati per garantire che non abbiano un impatto sulle integrazioni esistenti. Gli ESB sono gestiti a livello centrale, il che significa che l'IT deve comunque accettare le richieste di integrazione e questo può comportare lunghi tempi di attesa prima che le integrazioni vengano effettuate e i flussi di lavoro migliorati. È inoltre costoso implementare il disaster recovery e l'alta disponibilità per i server ESB. Molte aziende hanno scoperto che, come soluzione di integrazione, gli ESB non supportano l'automazione, la scalabilità e la velocità di cui hanno bisogno per competere nell'era digitale. Oggi,

l'uso degli enterprise service bus è limitato principalmente ai sistemi legacy che richiedono integrazioni complesse. Il modello architetturale ESB è stato sostituito dall'architettura a microservizi e altre tecnologie.

Da servizi a microservizi

L'architettura a microservizi è un'evoluzione delle SOA. Sebbene ogni servizio nelle SOA rappresenti una funzionalità aziendale completa, ogni microservizio è un componente software molto più piccolo specializzato in una sola attività. I microservizi affrontano le carenze delle SOA per rendere il software più compatibile con i moderni ambienti aziendali basati su cloud.

Sebbene l'architettura orientata ai servizi possa funzionare bene per la creazione di applicazioni aziendali di grandi dimensioni, è necessaria una maggiore flessibilità per scalare applicazioni aziendali più piccole. Infatti, alcune limitazioni delle SOA sono:

- L'Enterprise Service Bus collega più servizi tra loro, il che lo rende un singolo punto di errore.
- Tutti i servizi condividono un archivio dati comune. Ciò rende i servizi difficili da gestire individualmente.
- Ogni servizio ha un ampio campo di applicazione. Pertanto, se uno dei servizi fallisce, l'intero flusso di lavoro aziendale ne risentirà.

Il modello a microservizi, illustrato in Figura 1.3 la quale si può trovare in questa presentazione, divide un servizio tipico delle SOA in servizi più piccoli. Ogni microservizio opera all'interno del suo *bounded context* e viene eseguito indipendentemente dagli altri servizi [TG19]. In breve, l'architettura a microservizi presenta interdipendenze limitate o assenti tra i singoli servizi e riduce il rischio di guasti a livello di sistema. Le architetture a microservizi rappresentano quindi un paradigma innovativo nello sviluppo software, caratterizzato dalla suddivisione di un'applicazione monolitica in componenti autonomi e scalabili che comunicano tramite API. In questo modo, ogni singolo servizio può essere distribuito e ridimensionato in maniera indipendente. Questo approccio consente di distribuire in

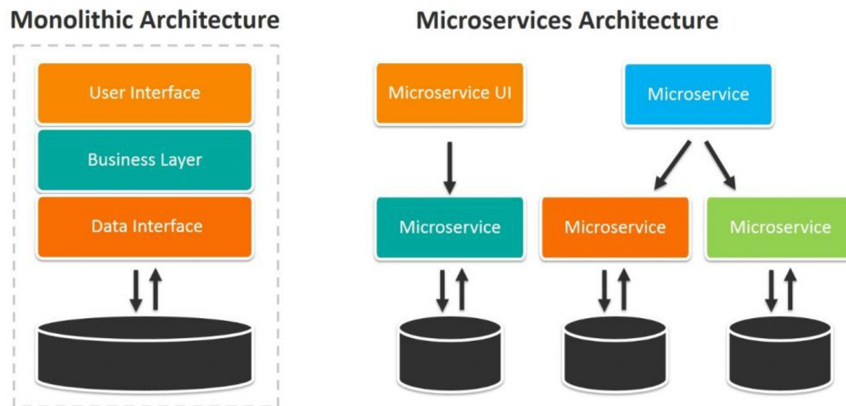

Figura 1.3: Diffrenze tra architettura monolitica e microservizi.

modo rapido e frequente le applicazioni grandi e complesse. Alcune caratteristiche di queste architetture sono:

- *Servizi con componenti multipli*: i microservizi sono costituiti da servizi con componenti debolmente accoppiati che possono essere sviluppati, distribuiti, utilizzati, modificati e ridistribuiti senza compromettere il funzionamento degli altri servizi o l'integrità dell'applicazione. Ciò consente di distribuire in modo rapido e semplice le singole funzioni di un'applicazione.

- *Elevate capacità di manutenzione e test*: i microservizi consentono ai team di sperimentare nuove funzioni ed eseguire il rollback se qualcosa va storto. Ciò semplifica l'aggiornamento del codice e accelera il time-to-market delle nuove funzioni. Inoltre, semplifica il processo di isolamento e correzione degli errori e dei bug nei singoli servizi.

- *Di proprietà di piccoli team*: in genere, i team piccoli e indipendenti creano un servizio all'interno dei microservizi, per questo motivo sono incoraggiati ad adottare le pratiche Agile e DevOps. I team hanno la possibilità di lavorare in modo indipendente e più rapidamente, riducendo i tempi del ciclo di sviluppo.

- *Organizzazione in base alle funzionalità aziendali*: con i microservizi, è possibile organizzare i servizi in base alle funzionalità aziendali. I team sono

interfunzionali e dispongono di tutte le competenze necessarie per sviluppare e soddisfare le singole funzionalità.

- *Infrastruttura automatizzata*: I team che si occupano della creazione e della gestione dei microservizi in genere utilizzano pratiche di automazione dell'infrastruttura come Continuous Integration (CI), Continuous Delivery (CD) e Continuous Deployment (CD). Grazie a queste pratiche, i team possono creare e distribuire ciascun servizio in modo indipendente senza influire sugli altri team. Possono inoltre distribuire la nuova versione di un servizio fianco a fianco a quella precedente.

Vantaggi e svantaggi

In questa sezione si analizzano con dettaglio i vantaggi e gli svantaggi associati alle architetture a microservizi, delineando come essi abbiano rivoluzionato l'approccio allo sviluppo e al deployment delle applicazioni. Per quanto riguarda i vantaggi, è possibile trovare:

- *Modularità*: uno dei principali vantaggi delle architetture a microservizi risiede nella loro natura modulare. Ogni servizio, o microservizio, rappresenta un modulo indipendente con specifiche funzionalità. Questo favorisce la suddivisione del sistema in unità gestibili separatamente, semplificando lo sviluppo, il testing e la manutenzione. La modularità consente anche una maggiore agilità, consentendo lo sviluppo parallelo di diversi microservizi da parte di team dedicati.

- *Scalabilità*: la scalabilità è un'altra caratteristica fondamentale. Poiché ogni microservizio può essere scalato in modo indipendente in risposta a carichi di lavoro specifici, le architetture a microservizi si adattano dinamicamente alle esigenze del sistema. Ciò contrasta con le tradizionali architetture monolitiche, dove la scalabilità può essere più complessa e comporta spesso la duplicazione dell'intera applicazione.

- *Migliore gestione della complessità*: le applicazioni monolitiche tendono a diventare complesse man mano che crescono in dimensioni e complessità

funzionale. Le architetture a microservizi affrontano questa sfida frammentando il sistema in singoli microservizi, ognuno dei quali si concentra su un aspetto specifico del dominio aziendale. Questa divisione favorisce una migliore gestione delle complessità, in quanto ogni microservizio è più gestibile, comprensibile e può essere sviluppato indipendentemente dagli altri. La separazione delle responsabilità attraverso i microservizi semplifica anche la manutenzione e l'aggiornamento del sistema nel tempo, consentendo l'introduzione di nuove funzionalità senza impattare l'intero sistema.

- *Velocità di sviluppo e rilascio*: la suddivisione di un'applicazione complessa in microservizi agevola lo sviluppo continuo e l'implementazione di metodologie DevOps. I team possono lavorare in modo indipendente su singoli microservizi, accelerando la velocità di sviluppo. La modularità consente anche il rilascio continuo di singoli microservizi senza interrompere l'intera applicazione, consentendo un deployment più rapido e sicuro. La capacità di rispondere prontamente ai cambiamenti dei requisiti del business e di rilasciare nuove funzionalità in modo agile è un elemento chiave che le architetture a microservizi portano al contesto dello sviluppo software.

- *Resilienza e isolamento degli errori*: la progettazione a microservizi favorisce la resilienza dell'applicazione. Se uno specifico microservizio fallisce, gli altri possono continuare a funzionare senza intoppi, riducendo l'impatto degli errori sul sistema nel suo complesso. Questo livello di isolamento degli errori è essenziale per garantire un'esperienza utente continua e ridurre il rischio di failure a cascata.

- *Libertà tecnologica*: le architetture basate su microservizi non applicano un unico approccio all'intera applicazione. I team hanno la libertà di scegliere gli strumenti migliori per risolvere i loro problemi specifici. Di conseguenza, i team che costruiscono i microservizi possono scegliere il miglior strumento per ciascun lavoro.

Sebbene le architetture a microservizi offrano una serie di vantaggi, è cruciale esaminare anche gli svantaggi e le problematiche associate a questo approccio. In particolare, i principali svantaggi possono essere riassunti nel seguente elenco:

- *Complessità di gestione*: mentre la modularità è un vantaggio, può anche portare a una complessità di gestione. Coordinare e monitorare un gran numero di microservizi richiede strumenti sofisticati e una pianificazione attenta. La gestione della distribuzione, il controllo delle versioni e la sincronizzazione tra i microservizi possono diventare complessi, richiedendo un'infrastruttura robusta e pratiche operative ben definite.

- *Overhead delle richieste di rete*: le architetture a microservizi comportano la comunicazione tra diversi servizi, spesso attraverso chiamate di rete. Questo può generare un overhead significativo, specialmente in applicazioni distribuite su larga scala. La latenza delle chiamate di rete può influire sulle prestazioni complessive del sistema, richiedendo strategie di gestione attente per mitigare questo impatto.

- *Coerenza dei dati*: mentre ogni microservizio ha il proprio database, garantire la coerenza dei dati attraverso l'intero sistema può essere una sfida. La gestione di transazioni distribuite e la sincronizzazione dei dati tra microservizi richiedono una progettazione accurata per evitare inconsistenze o errori nei dati.

- *Complessità di testing e debugging*: la natura distribuita degli ambienti a microservizi rende il testing e il debugging più complessi rispetto alle applicazioni monolitiche. Identificare e risolvere errori che coinvolgono più microservizi richiede strumenti e metodologie specifiche, aumentando la complessità del processo di sviluppo e manutenzione.

- *Overhead di sicurezza*: la sicurezza è un aspetto critico in qualsiasi sistema software, e le architetture a microservizi possono introdurre nuovi sfidanti aspetti di sicurezza. La gestione delle autorizzazioni, la protezione delle comunicazioni tra microservizi e la sicurezza dei dati richiedono attenzione particolare per evitare vulnerabilità.

Il bilancio tra vantaggi e svantaggi dipende fortemente dalle esigenze specifiche del progetto e dell'organizzazione. L'agilità e la flessibilità offerte da queste architetture possono giustificare l'adozione di questo approccio, ma devono essere

attentamente ponderate rispetto alla complessità e ai costi operativi aggiuntivi che possono emergere. Risulta evidente quindi che per progetti di dimensioni contenute o con requisiti di scalabilità limitati, un'architettura monolitica potrebbe risultare più semplice da gestire. D'altra parte, in scenari in cui la scalabilità dinamica, la distribuzione di team, e il deployment continuo sono fondamentali, le architetture a microservizi possono rappresentare la scelta più adatta.

Le architetture a microservizi richiedono un insieme di strumenti e tecnologie specializzati per gestire la complessità della progettazione, lo sviluppo, il deployment e la gestione operativa dei singoli microservizi. Uno degli elementi chiave in questo contesto è *Docker*, una tecnologia di containerizzazione che consente di confezionare applicazioni e le loro dipendenze in contenitori isolati, garantendo coerenza e portabilità tra ambienti diversi. Un partner essenziale di Docker è *Kubernetes*, una piattaforma di orchestrazione open-source. Kubernetes semplifica il deployment, la scalabilità e la gestione delle applicazioni basate su container, coordinando le interazioni complesse tra i diversi microservizi. Per la gestione delle comunicazioni tra i microservizi, si fanno spesso ricorso a protocolli come *gRPC* e *RESTful* API. gRPC offre una comunicazione efficiente basata sul protocollo HTTP e supporta la definizione di interfaccia di servizio tramite Protocol Buffers, mentre le API RESTful sono scelte per la loro semplicità e compatibilità con numerosi linguaggi di programmazione. Inoltre, per ottimizzare la gestione del traffico, la sicurezza e il monitoraggio avanzato, viene impiegato *Istio*. Questa piattaforma open-source semplifica il controllo delle versioni, il bilanciamento del carico e il tracciamento delle richieste tra i diversi microservizi. Affrontando il tema del monitoraggio delle prestazioni, strumenti come *Prometheus* e *Jaeger* svolgono un ruolo chiave. Prometheus è uno strumento flessibile per la raccolta di dati di monitoraggio, mentre Jaeger facilita il tracciamento delle richieste attraverso i vari componenti del sistema. Infine, nel processo di sviluppo dei microservizi, l'utilizzo di framework come *Spring Boot* (Java) e *Express.js* (JavaScript/Node.js) semplifica la gestione della logica di business, la persistenza dei dati e le operazioni CRUD.

1.1.3 Architetture ad eventi

Questa sezione si propone di esplorare in dettaglio le *Event-Driven Architectures (EDA)*, analizzando i principi fondamentali, le caratteristiche distintive e le applicazioni pratiche di questa metodologia. Verrà dimostrato infatti come le architetture ad eventi consentono di gestire dinamicamente le informazioni e rispondere in modo tempestivo alle nuove esigenze. Verranno esaminate anche le sfide e le best practices associate all'implementazione di queste architetture ad eventi, fornendo una panoramica completa delle considerazioni cruciali per gli sviluppatori e gli architetti interessati ad utilizzare questa tipologia di architettura nei loro progetti. Negli ultimi anni, l'evoluzione delle esigenze aziendali e la crescente complessità delle applicazioni software hanno portato all'emergere di nuove metodologie architetturali. Tra queste, un ruolo di rilievo è stato assunto dalle architetture ad eventi. Le architetture ad eventi, come quella in Figura 1.4 che si può trovare al seguente link, rappresentano un paradigma architetturale che pone l'accento sulla gestione dinamica degli eventi all'interno di un sistema software. Gli eventi, intesi come cambiamenti di stato o segnali significativi, diventano il motore che guida il flusso di controllo dell'applicazione. Infatti, l'essenza delle EDA risiede nella gestione degli eventi come elementi chiave per l'interazione tra i diversi componenti di un sistema distribuito. Gli eventi possono essere scatenati da varie fonti, come l'interazione dell'utente, cambiamenti di stato nei dati o segnali provenienti da altri sistemi [LF23]. L'aspetto cruciale è che, una volta generati, gli eventi fungono da messaggi asincroni che possono essere consumati da altri componenti interessati senza richiedere una comunicazione diretta.

Alcuni esempi di eventi possono essere: un articolo inserito in un carrello, un file caricato in un sistema di archiviazione o un ordine pronto per la spedizione. Gli eventi possono contenere lo stato (come il nome dell'articolo, il prezzo o la quantità di un ordine) o semplicemente gli identificatori (ad esempio, "l'ordine n.8942 è stato spedito") necessari per cercare le informazioni correlate. Ricapitolando, le EDA includono tre componenti principali:

- *Produttori di eventi*: sono responsabili della generazione e dell'invio degli eventi nel sistema. Possono essere applicazioni, servizi o sistemi esterni che generano eventi significativi.

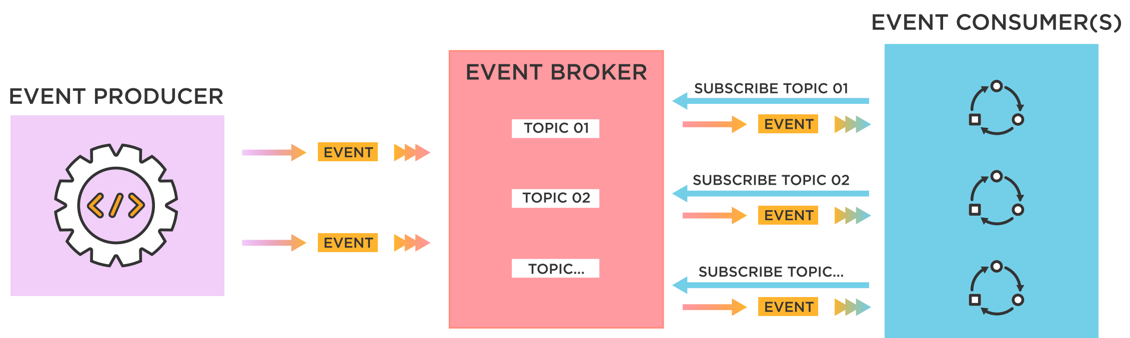

Figura 1.4: Schema funzionamento architettura ed eventi.

- *Event Bus o Event Queue*: rappresenta il canale attraverso il quale gli eventi vengono trasmessi dai produttori ai consumatori. Fornisce un'interfaccia di messaggistica asincrona.

- *Consumatori di eventi*: sono i componenti che reagiscono agli eventi. Possono essere applicazioni, servizi o moduli specifici all'interno del sistema che elaborano gli eventi ricevuti dalla coda.

Nel seguente elenco vengono riportati alcuni tra i principali vantaggi offerti dalle architetture ad eventi.

- *Reattività e Agilità*: grazie alla gestione degli eventi, le EDA consentono una risposta immediata ai cambiamenti dell'ambiente. Ciò rende il sistema più reattivo e adatto a scenari dinamici.

- *Disaccoppiamento*: gli eventi fungono da messaggi asincroni, riducendo le dipendenze dirette tra i componenti. Questo favorisce una maggiore flessibilità e manutenibilità del sistema.

- *Scalabilità*: le EDA permettono una distribuzione scalabile dei componenti, in quanto gli eventi possono essere gestiti in modo distribuito senza la necessità di comunicazioni dirette.

- *Integrazione efficiente*: l'architettura basata su eventi è anche altamente espandibile. Altri team possono ampliare le caratteristiche e aggiungere funzionalità senza compromettere i servizi esistenti. Attraverso la pubblicazione

di eventi, un'applicazione può integrarsi con i sistemi esistenti e le future applicazioni potranno integrarsi facilmente come utenti di eventi senza modificare la soluzione esistente. I produttori di eventi non hanno conoscenze in merito agli utenti finali, perciò l'ampliamento del sistema ha una frizione minore e le nuove caratteristiche o integrazioni non aggiungono dipendenze che rischiano di rallentare lo sviluppo futuro.

- *Agilità nello sviluppo*: grazie all'architettura basata su eventi e agli strumenti di instradamento, gli sviluppatori non sono più tenuti a scrivere il codice personalizzato per eseguire il polling, il filtraggio e l'instradamento degli eventi. Uno strumento di instradamento, infatti, filtra automaticamente gli eventi e li invia in modalità push ai consumatori. Lo strumento di instradamento elimina anche la necessità di un eccessivo coordinamento tra i servizi produttori e consumatori, per cui l'agilità degli sviluppatori aumenta.

- *Riduzione dei costi*: le EDA utilizzano tipicamente un approccio push, quindi tutto accade on demand quando un evento si presenta nello strumento di instradamento. Per questo motivo, non è necessario pagare per eseguire il polling continuo alla ricerca di un evento. Ciò si traduce in minore consumo di larghezza di banda della rete, minore utilizzo della CPU.

Le architetture basate su eventi sono ideali per migliorare l'agilità e la rapidità. In genere si trovano nelle applicazioni moderne che utilizzano microservizi o in qualsiasi applicazione che disponga di componenti disaccoppiati. Se si dispone infatti di sistemi in esecuzione su diversi stack, è possibile utilizzare un'architettura event-driven per condividere informazioni tra di essi senza accoppiarli. Inoltre, invece di controllare costantemente le risorse, è possibile usufruire di un'EDA per monitorare e ricevere avvisi su anomalie, modifiche e aggiornamenti. Queste risorse includono bucket di archiviazione, tabelle di database, funzioni serverless, nodi di calcolo e altro ancora. A differenza delle applicazioni monolitiche, che possono elaborare tutto all'interno dello stesso spazio di memoria su un singolo dispositivo, le applicazioni basate su eventi comunicano attraverso reti. Questa configurazione introduce una latenza variabile. Nonostante le applicazioni monolitiche possano avere una latenza minore o meno variabile, ciò di solito va a discapito di scalabilità e disponibilità.

1.2 Panoramica sui Digital Twins

In questa sezione verranno descritte le definizioni e le caratteristiche fondamentali del paradigma dei *Digital Twins*. Successivamente, si analizzerà il contesto in cui i Digital Twins trovano applicazione, esplorando gli ambiti chiave in cui si manifestano le loro caratteristiche. Inoltre, verrà spiegato come i Digital Twins influenzino settori specifici, contribuendo a innovare processi e pratiche. Nel corso della sezione, verranno esaminati anche gli strumenti e le tecnologie essenziali che supportano l'implementazione pratica dei Digital Twins. Questa sezione mette in evidenza le infrastrutture e gli strumenti cruciali per la realizzazione e la gestione efficace di essi. Infine, si affronteranno le sfide, i limiti tecnologici, le questioni etiche e le criticità connesse alla sicurezza e alla privacy dei dati, fornendo una panoramica sintetica su aspetti critici dei Digital Twins.

1.2.1 Definizioni e caratterstiche

Il concetto di Digital Twin viene introdotto per la prima volta nel 2002, quando *Michael Grieves* presentò un "ideale concettuale" per il Product Lifecycle Management (PLM) durante una sessione formativa per un centro di PLM [GV17]. Questo modello concettuale fu successivamente ripreso nel 2003 da Grieves durante un corso di PLM presso l'Università del Michigan, dove lo definì come "Mirrored Space Model". Pur non utilizzando esplicitamente il termine "Digital Twin", il concetto delineato incorpora le tre caratteristiche fondamentali del paradigma: un oggetto fisico nello spazio, la sua rappresentazione virtuale e una connessione tra i due spazi per facilitare il flusso di dati [Gri15]. Fin dall'inizio, Michael Grieves ha riconosciuto l'applicabilità dei Digital Twins nel settore manifatturiero come un concetto e un insieme di tecnologie in grado di affiancare i sistemi di produzione per l'intero ciclo di vita di un prodotto. Inizialmente, nel suo lavoro [Gri15], Grieves ha concettualizzato l'idea di un Digital Twin, oggi denominato "descrittivo", che è capace di rappresentare lo stato corrente di un oggetto fisico. Questa concezione offriva la possibilità di condurre analisi e osservazioni di prodotti o di asset fisici direttamente dalla loro replica digitale. Successivamente, grazie agli sviluppi proposti da Grieves, si sono delineate visioni sempre più complete e ricche di concetti

dove il Digital Twin è stato impiegato anche per condurre test e forme iniziali di simulazione. Successivamente, il rapido sviluppo delle tecnologie di comunicazione, la diffusione dell'*Internet of Thing* (IoT), e dunque dei Big Data, e l'evoluzione delle tecnologie di simulazione e dell'Intelligenza Artificiale hanno contribuito all'esplosione dei Digital Twin fornendo un supporto maturo e pratico alla loro implementazione [TZLN19]. Nello stesso momento, altri ricercatori, iniziavano a dare i primi contributi a questo concetto. In particolare, Minerva, Lee e Crespi individuarono i settori e le tecnologie che influenzarono maggiormente il concetto e sopratutto un insieme di caratteristiche che, ad oggi, vengono considerate la base del concetto di Digital Twin [MLC20].

La prima definizione data da Michael Grieves sui Digital Twins fu:

> "The Digital Twin is a set of virtual information constructs that fully describes a potential or actual physical manufactured product from the micro atomic level to the macro geometrical level. At itsoptimum, any information that could be obtained from inspecting a physical manufactured product can be obtained from its Digital Twin."

La definizione fornita non si limita a descrivere l'oggetto fisico virtualizzato come un'entità generica del mondo reale, ma piuttosto come un prodotto industriale. Questa specificità è dovuta al fatto che Grieves ha concepito il concetto di Digital Twin proprio in relazione al settore manifatturiero. Un aspetto rilevante che emerge da questa definizione è la capacità di utilizzare il Digital Twin come strumento per ottenere informazioni dirette sullo stato dell'oggetto fisico. L'adozione del paradigma in contesti al di fuori dell'industria ha portato a una trasformazione della definizione. Da riferirsi esclusivamente ad artefatti industriali, essa si è estesa a includere qualsiasi oggetto fisico, potenzialmente anche intangibile, come processi e attività. Nel survey di Minerva [MLC20], infatti , viene presenta una definizione più ampia e generale:

> "A Digital Twin is a comprehensive software representation of an individual physical object. It includes the properties, conditions, and behavior(s) of the real-time object through models and data. A Digital Twin is a set of realistic models that can simulate an object's behavior in the deployment environment. The Digital Twin represent

and reflects its physical twin and remains its virtual counterpart across the object's entire lifecycle."

In questa ultima descrizione possiamo notare alcune nozioni che rappresentano l'essenza del concetto:

- Un Digital Twin si riferisce a un oggetto fisico modellandone proprietà e comportamento.
- Un Digital Twin permette di modellare un oggetto fisico per tutto il suo ciclo di vita, evolvendosi insieme a esso. In questo modo, il DT permette anche di descrivere la storia dell'asset fisico.
- Un Digital Twin contiene tutte le informazioni necessarie per predire e simulare il comportamento del relativo oggetto fisico.

La capacità di un Digital Twin di modellare e descrivere in modo accurato gli aspetti chiave dell'asset fisico è fondamentale per rappresentarne le proprietà e il comportamento di esso. Questa modellazione deve essere precisa senza includere dettagli superflui, consentendo al contempo la simulazione, la previsione e l'ottimizzazione del comportamento dell'oggetto fisico. La modellazione del Digital Twin deve selezionare le proprietà della componente fisica a diversi livelli, comprendendo sia aspetti macroscopici come forma e dimensione, che caratteristiche microscopiche come la ruvidezza del materiale della sua superficie. L'idea che il Digital Twin non sia una mera rappresentazione statica dell'entità fisica corrispondente, bensì un modello dinamico che evolve parallelamente al suo oggetto fisico per l'intero ciclo di vita, è intrinseca fin dalla prima formulazione concettuale di Grieves nel 2002 [GV17]. La continua evoluzione della copia virtuale è resa possibile grazie all'incessante interazione, comunicazione e sincronizzazione con l'oggetto fisico, realizzate attraverso la relazione bidirezionale che esiste tra i due.

Nel whitepaper di formalizzazione dei Digital Twin, Grieves elenca 3 principali proprietà:

- **Contestualizzazione**: l'essere umano possiede una capacità distintiva di visualizzare globalmente una situazione, comprendere il contesto e concettualizzare un problema. Nell'affrontare una problematica, la tendenza tipica

è quella di utilizzare la vista per tradurre il problema in numeri, simboli e lettere, per poi doverlo contestualizzare nuovamente attraverso la stessa percezione visiva. Questo doppio passaggio comporta una perdita di informazione ed è inefficace. Contrariamente, i Digital Twin possiedono la capacità di fornire in tempo reale una visualizzazione completa della situazione da analizzare, consentendo simultaneamente l'indicazione dei parametri desiderati, compresi quelli futuri in caso di simulazione. Questa caratteristica consente di contestualizzare e analizzare la problematica in modo significativamente più rapido, eliminando la perdita di informazioni e migliorando l'efficienza del processo decisionale.

- **Comparazione**: gli esseri umani utilizzano tipicamente il confronto come strumento per valutare una situazione. Consapevolmente o anche inconsciamente, tendiamo a confrontare i risultati ottenuti con quelli desiderati al fine di individuare le differenze e capire come eliminarle o ridurle. I Digital Twin consentono di visualizzare contemporaneamente lo stato attuale dell'oggetto fisico e gli intervalli di valori entro cui ogni suo parametro dovrebbe rientrare. Questo approccio accelera il confronto tra la situazione attuale e quella desiderata, permettendo di attuare tempestivamente eventuali attività di recupero necessarie.

- **Collaborazione**: la collaborazione rappresenta uno degli strumenti più potenti nelle mani degli esseri umani, consentendo di unire diversi punti di vista e di elaborare soluzioni più intelligenti e innovative per i problemi. Tuttavia, questa potente risorsa si scontra con il limite umano, poiché la concettualizzazione o contestualizzazione del problema avviene solitamente a livello individuale. I Digital Twin superano questa limitazione consentendo la condivisione della contestualizzazione di un problema. Essi abilitano un numero illimitato di persone, anche dislocate in luoghi diversi, a visualizzare in modo uniforme una stessa situazione. Ciò favorisce la collaborazione efficace, permettendo a un gruppo di individui di partecipare attivamente alla risoluzione di problemi senza le restrizioni fisiche della presenza nello stesso luogo.

Sempre nel contesto delle proprietà dei Digital Twins, nel survery di Minerva [MLC20], vengono presentate invece un insieme di caratteristiche che contrad-

distinguono i Digital Twin focalizzandosi però sulla loro relazione con l'Internet of Things (IoT). Quest'ultimo, infatti, se da un lato beneficia direttamente del paradigma dei Digital Twin, dall'altro ne costituisce sicuramente un fattore abilitante. Di seguito vengono riportate nel dettaglio le caratteristiche individuate da Minerva necessarie affinché un Digital Twin possa essere definito tale.

Identità

Alla luce della definizione di Digital Twin, che lo considera "una rappresentazione software completa di un oggetto fisico individuale", emerge la necessità di un'identificazione altrettanto complessa, se non superiore, per le repliche digitali rispetto alle loro controparti fisiche. Per poter essere individuate all'interno dello spazio software, le repliche digitali devono possedere un identificatore univoco che tenga conto sia dello spazio che del tempo. Considerando queste premesse, diventa possibile utilizzare la replica digitale per rappresentare la controparte fisica nello spazio e nel tempo. Ad esempio, nel caso di un Digital Twin di un paziente, l'identificazione del paziente consente di riconoscere la persona, mentre l'identificazione nello spazio e nel tempo stabilisce lo stato delle informazioni e del modello, rappresentando il paziente nel contesto necessario per l'analisi. È importante notare che più repliche possono fare riferimento allo stesso asset fisico, ma la replica digitale è comunque individuata attraverso l'identificatore del Digital Twin e il riferimento all'identificatore dell'asset fisico.

Rappresentatività e contestualizzazione

È essenziale che la copia virtuale in un Digital Twin sia il più verosimile possibile all'entità fisica di riferimento. Tuttavia, rappresentare l'oggetto fisico in tutte le sue sfaccettature può essere estremamente complesso e, talvolta, inutile. Alcune caratteristiche dell'oggetto possono infatti essere non funzionali al raggiungimento dell'obiettivo per cui viene implementato il Digital Twin. Il modello del Digital Twin deve essere progettato e implementato con chiarezza rispetto al contesto in cui opera. Deve rappresentare esclusivamente le proprietà, le caratteristiche e i comportamenti dell'asset fisico che sono necessari e sufficienti per raggiungere gli obiettivi prefissati. In altre parole, il modello deve essere mirato, includen-

do solo quegli aspetti dell'oggetto fisico che sono rilevanti per classificarlo come rappresentativo sotto tutti gli aspetti. La rappresentatività di un Digital Twin dipende quindi dalla similarità rispetto all'asset fisico a cui vogliamo arrivare e dalla contestualizzazione. Contestualizzare un Digital Twin significa che tutte le caratteristiche e i dati del modello sono necessari e sufficienti per rappresentare l'asset fisico nello specifico contesto in considerazione.

Riflessione

Un Digital Twin deve essere in grado di riflettere in modo accurato lo stato del corrispondente oggetto fisico. Ogni cambiamento di stato o evento generato dall'entità fisica deve essere fedelmente registrato dal Digital Twin [MC21]. Lo stato dell'oggetto fisico, costituito dall'insieme dei valori degli attributi, degli eventi e delle azioni, evolve nel tempo. Pertanto, possiamo affermare che è adeguatamente descritto attraverso il Digital Twin solo se ogni valore è tempestivamente mappato nel suo equivalente nello stato gemello virtuale. La proprietà di riflessione si riferisce alla capacità del modello del Digital Twin di rappresentare univocamente, anche se non necessariamente in modo uno-a-uno, ciascuna caratteristica rilevante dell'oggetto fisico. Ogni valore rilevante della controparte fisica deve essere univocamente rappresentato nel Digital Twin. Ad ogni valore può essere applicata una funzione di trasformazione *f(x)* con la quale mappare i dati. La funzione *f(x)*, dove x sono i dati dell'asset fisico e f è la trasformazione dei dati all'interno della replica, può essere semplicemente una funzione identità oppure potrebbe essere una vera e propria funzione che consente anche di ottenere dati derivati.

Replicazione

Le proprietà precedentemente menzionate implicano che il modello utilizzato per un Digital Twin, ovvero l'insieme delle caratteristiche dell'entità fisica selezionato per la rappresentazione virtuale, deve essere ridotto al minimo necessario per rendere efficace ed efficiente l'uso della copia virtuale nel contesto applicativo specifico. Questa necessità suggerisce che ogni concetto reale e, in modo ricorsivo, ogni copia software, possano essere virtualizzati e replicati in diverse istanze virtuali. Ciascun clone sarà modellato su misura per soddisfare i requisiti applicativi

specifici del contesto in cui deve operare. È possibile individuare diversi pattern di virtualizzazione di un asset fisico in un Digital Twin. Il più semplice È quello che vede una mappatura 1-1 tra l'asset e la replica. In questo modo il Digital Twin È come se fosse la copia esatta dell'asset all'interno del mondo virtuale. Altri pattern comuni invece mirano a sfruttare le potenzialità computazionali dei servizi cloud i quali fanno largo uso della virtualizzazione. Ogni replica comunque mantiene lo stato dell'asset fisico e a livello di framework si deve assicurare che lo stato tra le varie repliche sia sempre consistente al fine di erogare il servizio nella maniera corretta con la stessa visione condivisa della realtà.

Entanglement

Fin dalla concezione del paradigma, tre elementi fondamentali costituiscono il nucleo del Digital Twin: i) un oggetto fisico, ii) una copia digitale, e iii) una connessione tra i due. Il termine "entanglement", utilizzato da Minerva et al., si riferisce specificamente a questa terza componente, ossia al legame tra il livello fisico e quello virtuale. Attraverso questo collegamento, le informazioni che descrivono il primo componente scorrono in tempo reale, consentendo al clone software di elaborarle e renderle disponibili alle applicazioni. Le proprietà dell'entanglement caratterizzano il legame tra l'oggetto fisico e virtuale, in particolare come avviene lo scambio di informazioni tra i due. Solitamente si considerano tre caratteristiche fondamentali:

- Connetività: La comunicazione efficace tra l'asset fisico e la sua copia digitale è fondamentale per garantire una rappresentazione accurata e in tempo reale nel Digital Twin. Questa comunicazione può avvenire direttamente tra l'oggetto reale e il Digital Twin o indirettamente attraverso un mediatore esterno. Nel caso della comunicazione diretta, l'oggetto fisico deve essere dotato di un mezzo di trasmissione delle informazioni. Questo può avvenire attraverso sensori, dispositivi di telemetria o altri strumenti integrati nell'asset. Ad esempio, sensori di temperatura, accelerometri o fotocamere possono raccogliere dati sullo stato dell'oggetto fisico e trasmetterli direttamente al Digital Twin. Nel caso della comunicazione indiretta, un mediatore esterno funge da intermediario tra l'oggetto reale e il Digital Twin. Questa terza en-

tità è responsabile di raccogliere, elaborare e trasmettere le informazioni tra i due. Il mediatore può essere un dispositivo hardware o un sistema software che agisce come ponte tra il mondo fisico e quello digitale.

- Tempestività: lo scambio tempestivo di informazioni tra l'entità fisica e quella digitale è essenziale per mantenere una rappresentazione accurata nel Digital Twin. Il tempo impiegato per trasmettere le informazioni tra le due componenti dovrebbe essere trascurabile rispetto al periodo tra due cambiamenti successivi di stato. Il limite massimo accettabile per il tempo di trasmissione delle informazioni tra le due entità è indicato dal tempo medio che intercorre tra due cambiamenti successivi di stato di una delle due parti. Questo approccio assicura un allineamento efficace tra l'oggetto fisico e il Digital Twin, consentendo una riflessione in tempo reale delle dinamiche dell'oggetto e facilitando risposte immediate a cambiamenti critici. L'efficienza nel tempo di trasmissione delle informazioni è cruciale per garantire che il Digital Twin rimanga una rappresentazione fedele e aggiornata dell'oggetto reale, supportando analisi, simulazioni e monitoraggio in tempo reale delle variazioni di stato.

- Associazione: la relazione tra l'entità logica e quella fisica può essere unidirezionale o bidirezionale. Nel primo caso, il flusso informativo si muove dal piano fisico a quello virtuale o viceversa. Ad esempio, nel caso di sensori, il flusso informativo può andare dal piano fisico al piano digitale, mentre nel caso di attuatori, il flusso può avvenire dal piano digitale a quello fisico attraverso l'invio di comandi. Nel secondo caso, la relazione prevede uno scambio bidirezionale di informazioni. Ciò significa che il flusso informativo avviene in entrambe le direzioni. Questo approccio può portare a un notevole valore aggiunto per l'oggetto fisico, poiché può ricevere sia aggiornamenti che comandi mirati a migliorare il suo funzionamento. La bidirezionalità nella comunicazione consente una sincronizzazione più completa tra l'entità fisica e quella digitale, permettendo una gestione più efficace e reattiva delle dinamiche dell'oggetto nel mondo reale.

Persistenza

Questa proprietà si riferisce alla capacità del Digital Twin di persistere nel tempo, superando i limiti fisici a cui può essere sottoposto l'oggetto fisico nel mondo reale. Deve garantire una continua disponibilità alle applicazioni che interagiscono con esso. Inoltre, in situazioni di malfunzionamento dell'oggetto fisico, la copia digitale deve fungere da fonte di informazioni per ripristinarne lo stato.

Memorizzazione

Un Digital Twin deve avere la capacità di memorizzare e rappresentare tutte le informazioni rilevanti, sia attuali che passate. Considerando il concetto di rappresentatività e la necessità di fornire contestualizzazione, la quantità di informazioni grezze o elaborate da gestire potrebbe essere significativa. In particolare, questi grandi dataset servono a due scopi principali: da un lato, consentono di comprendere e analizzare lo stato e il comportamento dell'oggetto fisico, mentre dall'altro, abilitano il Digital Twin a predire il futuro stato dell'oggetto fisico, anche in eventuali condizioni ambientali diverse.

Compositività

Gli oggetti reali spesso risultano essere l'aggregazione di più entità individuali. In relazione a ciò, un Digital Twin deve possedere la capacità di monitorare simultaneamente sia il comportamento dell'oggetto composito che quello delle singole componenti. In un sistema di dimensioni considerevoli, ogni componente o sottosistema dovrebbe essere rappresentato da un apposito clone digitale. Questo insieme di copie software deve tuttavia essere in grado di collaborare e interagire come se fosse un'unica grande entità per soddisfare le esigenze delle applicazioni. D'altra parte, questa proprietà si riferisce alla possibilità, attraverso il Digital Twin, di astrarsi dalla complessità di un sistema di grandi dimensioni. Ciò consente di concentrarsi sugli aspetti rilevanti senza la necessità di considerare ogni singola componente.

Responsabilità

Ogni Digital Twin deve essere gestito in modo completo e accurato, anche in caso di malfunzionamenti. Dopo aver avviato una procedura di recovery, deve essere in grado di rispondere a interrogazioni sullo stato della controparte fisica, fornendo le informazioni più recenti e disponibili. Come già menzionato, un Digital Twin deve rappresentare l'entità fisica per l'intero ciclo di vita e oltre. Questa proprietà implica che la copia virtuale debba persistere ed essere operativa anche oltre il periodo di "vita" dell'entità fisica. La possibilità di applicare il paradigma in diversi contesti, come ad esempio nel settore sanitario, richiede inoltre la presenza di diversi livelli di gestione e responsabilità per lo stesso Digital Twin.

Augmentation

A differenza di ciò che avviene nel mondo fisico, le funzionalità e i servizi offerti da un Digital Twin possono essere modificati e potenziati nel corso del tempo, sia attraverso l'implementazione di nuove soluzioni software che grazie all'analisi dei dataset costituiti dai dati inviati dall'oggetto fisico. Il concetto di "augmentation" consente al Digital Twin di fornire funzionalità aggiuntive rispetto a quelle intrinseche all'entità fisica.

Ownership

: Questa caratteristica si riferisce al concetto di possesso del Digital Twin (DT), che deve essere regolato su due fronti distinti. Da un lato, è necessario stabilire chi sia il proprietario della vasta mole di dati generati e utilizzati dal DT. Dall'altro lato, è fondamentale determinare il possessore del DT, o più precisamente, del software che implementa la componente digitale del paradigma. Nel contesto degli oggetti fisici, è generalmente semplice determinare chi ne sia il proprietario. Tuttavia, per le corrispondenti copie digitali, la situazione potrebbe non essere altrettanto chiara, poiché i due proprietari non coincidono spesso. Il concetto di ownership risulta quindi complesso, ma date l'ampia diffusione dei Digital Twin, deve essere attentamente analizzato, gestito e regolamentato.

1.2.2 Contesto e ambiti di applicazione

Dal momento della sua concezione, il concetto di Digital Twin ha trovato applicazioni naturali in due settori principali: quello manifatturiero e quello dell'aviazione. Tuttavia, in tempi più recenti, questo paradigma ha visto una diffusione sempre più ampia in vari altri ambiti, tra cui spiccano l'healthcare e le Smart City.

Settore manifatturiero

Il settore manifatturiero è stato il precursore nell'adozione dei Digital Twins, contribuendo significativamente allo sviluppo iniziale di questo concetto. Infatti, Grieves colloca le radici del termine nel contesto manifatturiero grazie ai sistemi PLM (Product Lifecycle Management), i quali gestiscono il ciclo di vita di un prodotto. Questo settore non solo ha influenzato la definizione di molteplici aspetti e caratteristiche dei Digital Twins, ma ha anche rappresentato uno dei primi campi in cui sperimentare, verificare e testare l'applicabilità reale e i vantaggi dei Digital Twins su larga scala e in contesti ad elevata complessità. Il settore manifatturiero ha introdotto diverse approcci che successivamente hanno influito su altre industrie e hanno guidato la progettazione e lo sviluppo di soluzioni basate sui Digital Twins, anche al di fuori di questo specifico settore. Prima dell'avvento dei sistemi IoT (Internet of Things), l'unico modo per ottenere informazioni sullo stato di un oggetto fisico, di una macchina o di un'azienda era essere fisicamente nelle sue vicinanze per condurre analisi. Di conseguenza, le informazioni relative all'oggetto fisico erano strettamente legate a esso. Solo grazie alle tecnologie recenti è diventato possibile estrarre e rendere disponibili esternamente le informazioni di un asset fisico, dando vita a quello che oggi conosciamo come Digital Twin. Le informazioni, indipendentemente dalla loro natura, sono di cruciale importanza, come sottolineato da Grieves nei "Sistemi Complessi" [GV17]. Questi sistemi sono caratterizzati da una vasta rete di componenti con comunicazioni molti-a-molti e un'elaborazione delle informazioni complessa, rendendo la predizione dello stato di tali sistemi estremamente complicata. L'utilizzo dei Digital Twins nella progettazione di un prodotto consente ai progettisti di effettuare valutazioni sulla qualità fin dalle prime fasi, facilitando un design più efficiente e informato. Nel contesto del processo di produzione, i Digital Twins rendono tale processo più affidabile, flessi-

bile e prevedibile. Grazie allo scambio di dati tra il livello fisico e quello digitale, i Digital Twins consentono il monitoraggio di complesse linee di produzione, agevolando attività tempestive di risoluzione dei problemi e ottimizzazione del processo. Nel supporto all'intera fabbrica, i Digital Twins sono comunemente impiegati in tre casi d'uso principali: ottimizzazione dei processi, manutenzione dei macchinari e progettazione della loro collocazione ideale [BCF19]. La creazione dei Digital Twins per i diversi macchinari all'interno della fabbrica e la loro interconnessione permettono di ottimizzare l'intero processo produttivo. Contemporaneamente, i diversi Digital Twins consentono il monitoraggio dello stato dei singoli macchinari e la raccolta di dati storici per l'impiego di modelli di Machine Learning e AI al fine di prevedere eventuali malfunzionamenti. La progettazione della disposizione dei macchinari all'interno della fabbrica può essere ottimizzata grazie all'uso dei Digital Twins, rendendo modulare e parametrizzata l'architettura. Simulando la disposizione delle diverse componenti attraverso i Digital Twins, è possibile testare, ottimizzare e validare diverse soluzioni progettuali. Questo processo consente di accelerare la produzione e, al contempo, ridurre i costi complessivi.

Healthcare

Poter rappresentare un paziente, un ospedale o sistemi anche più complessi in modo digitale può infatti fornire un enorme valore, soprattutto se pensiamo a casi come quello della medicina personalizzata o dei test clinici. In tali ambiti, i Digital Twin possono essere validamente utilizzati per simulare terapie individuali e progressione della malattia, visualizzandone in tempo reale i potenziali risultati e prevedendone gli effetti a lungo termine. Secondo Schwartz et al. (2020) [SWBB20] e Voigt et al. (2021) [VID+21], un'automobile è equipaggiata con più di cinquanta sensori e mini-computer che monitorano in modo continuo il suo funzionamento. Questo permette di creare modelli virtuali e fare simulazioni considerando un'ampia gamma di fonti di dati che rappresentano in modo fedele l'ambiente circostante. Allo stesso modo, per creare una fedele rappresentazione digitale di una persona o un ospedale, è necessario dotare il paziente o la struttura di svariati sensori che monitorano in tempo reale la situazione. Da ciò possiamo intuire la complessità della transizione ad un modello basato sul digital twin in Sanità. Occorrono,

infatti, sensori intelligenti di ogni tipo, da quelli cardiaci a quelli di movimento (per esempio, per rilevare possibili cadute) e molti altri ancora che permettano di raccogliere dati dei più disparati per poi aggregarli e fornire una visione d'insieme del soggetto. E ciò, ovviamente, pur rappresentando un percorso sempre più esplorato e battuto, non è ancora lo stato dell'arte dell'Healthcare per come lo conosciamo oggi. In relazione all'idea di generare pazienti digitali, molti degli studi in questo ambito sono volti a generare i DT del corpo umano o di alcuni dei suoi organi [CGMR20]. Un esempio tangibile dell'applicazione dei Digital Twins è rappresentato da *Siemens Healthineer*, che ha sviluppato un Digital Twin del cuore. Questo Digital Twin simula i processi fisiologici sottostanti al fine di anticipare le risposte al trattamento prima dell'intervento concreto. Un approccio innovativo è rappresentato dall'idea di creare un ecosistema di Digital Twins interconnessi per realizzare un sistema dedicato al Trauma Management [RCM22a]. In questo scenario, differenti Digital Twins, possibilmente provenienti da diversi fornitori, vengono utilizzati sinergicamente per supportare i medici nella gestione delle attività di soccorso. L'approccio si basa sulla creazione di un Digital Twin per ciascun asset fisico strategico all'interno dell'organizzazione sanitaria. Questo permette di riflettere e potenziare le funzionalità di ciascun asset a livello digitale, contribuendo così a ottimizzare le operazioni di gestione del trauma.

Smart City

L'impiego e il potenziale dei Digital Twins nel contesto delle Smart City stanno crescendo di anno in anno, alimentati dal sempre maggiore utilizzo dei dispositivi IoT (Internet of Things). Tra i molteplici possibili impieghi di questo paradigma nell'ambito delle Smart City, risultano di particolare rilevanza quelli legati alla gestione del traffico, al consumo energetico e agli edifici. Attraverso l'utilizzo combinato dei Digital Twins e dei modelli di Machine Learning, è possibile sviluppare sistemi per monitorare e prevenire la congestione del traffico, un problema sempre più pressante a causa dell'incremento significativo del numero di veicoli in circolazione. Questo risultato è ottenuto mediante la creazione dei Digital Twins dei mezzi in circolazione, dell'infrastruttura stradale e dei pedoni. Questi Digital Twins, combinati con l'uso dei sistemi di videosorveglianza, consentono di mo-

dellare digitalmente il sistema del traffico. Dotare le città di numerosi sensori e tecnologie smart abilita la creazione dei rispettivi Digital Twins. Questi ultimi permettono, ad esempio, la realizzazione di simulazioni per rispondere a domande del tipo "cosa succederebbe se" e forniscono agli analisti la capacità di comprendere come le città si comporterebbero in diverse condizioni economiche, ambientali e sociali. Inoltre, consentono di identificare i possibili fattori di rottura [MT17].

1.2.3 Strumenti e tecnologie

La diffusione crescente del paradigma dei Digital Twins negli ultimi anni ha favorito lo sviluppo di numerose tecnologie che consentono di definire Digital Twins in modi più o meno strutturati e completi. Tra le principali soluzioni software che facilitano la creazione di Digital Twins, spiccano Azure Digital Twins di Microsoft, AWS IoT TwinMaker di Amazon Web Services ed Eclipse Ditto.

AWS IoT TwinMaker

Il servizio cloud offerto da Amazon Web Services, chiamato AWS IoT TwinMaker, è progettato per la definizione dei Digital Twins, principalmente nel contesto industriale. L'architettura utilizzata per modellare il sistema fisico si basa principalmente sui concetti di "Entità" e "Componente". Le entità rappresentano le versioni digitali degli elementi distinti di un Digital Twin, ciascuna catturando una specifica capacità dell'elemento modellato. Questi elementi possono essere dispositivi fisici, concetti astratti o processi. Ogni entità è associata a componenti che descrivono i dati e il contesto relativi a quell'entità. I componenti possono fornire sia dati statici che flussi di dati provenienti da ulteriori servizi AWS.

Come si può notare in Figura 1.5 presente nella documentazione al seguente link, una caratteristica distintiva di questa soluzione è l'enfasi posta sull'aspetto di visualizzazione del Digital Twin. Infatti, il servizio fornisce strumenti per associare un modello tridimensionale al Digital Twin definito. Questo approccio consente una rappresentazione visiva e interattiva del Digital Twin, facilitando la comprensione e la gestione del sistema fisico rappresentato.

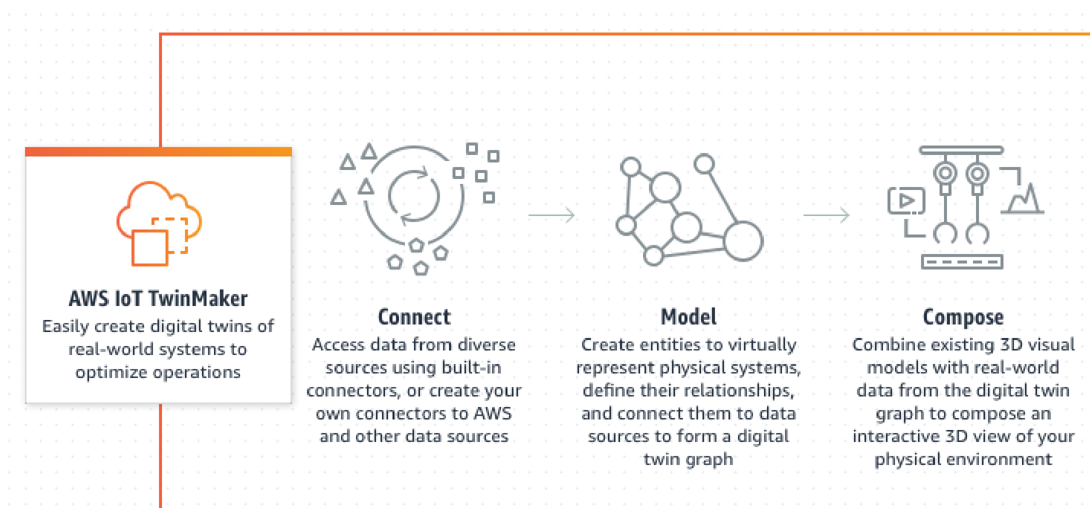

Figura 1.5: Schema del funzionamento di AWS IoT Twin Maker.

Eclipse Ditto

Eclipse Ditto è un framework open source progettato per implementare Digital Twins. Il suo obiettivo principale è creare una rappresentazione virtuale di oggetti esistenti nella realtà, come macchine intelligenti, stazioni di ricarica, e così via. Queste rappresentazioni sono denominate "Twins" e consentono agli utenti di accedere all'oggetto fisico come se fosse un servizio web convenzionale. Eclipse Ditto offre inoltre la possibilità di utilizzare protocolli di comunicazione già esistenti per interagire con i dispositivi. È importante notare che Eclipse Ditto non mira a essere una piattaforma completa per l'Internet of Things (IoT). Nessuna parte di Eclipse Ditto esegue su hardware IoT, e non è definito alcun protocollo per la comunicazione diretta con i dispositivi stessi. Invece, si concentra sulla creazione di una rappresentazione digitale dei dispositivi esistenti, consentendo agli utenti di interagire e gestire tali dispositivi attraverso un'interfaccia basata su servizi web.

Come si può notare in Figura 1.6, presente nella documentazione al seguente link, Eclipse Ditto funge da middleware IoT, agendo come un livello di astrazione per gli oggetti fisici nelle soluzioni IoT tramite l'implementazione di Digital Twins. Questo framework offre la flessibilità di interfacciarsi con dispositivi utilizzando diversi tipi di protocolli attraverso l'uso di connessioni. Ciò consente alle soluzioni

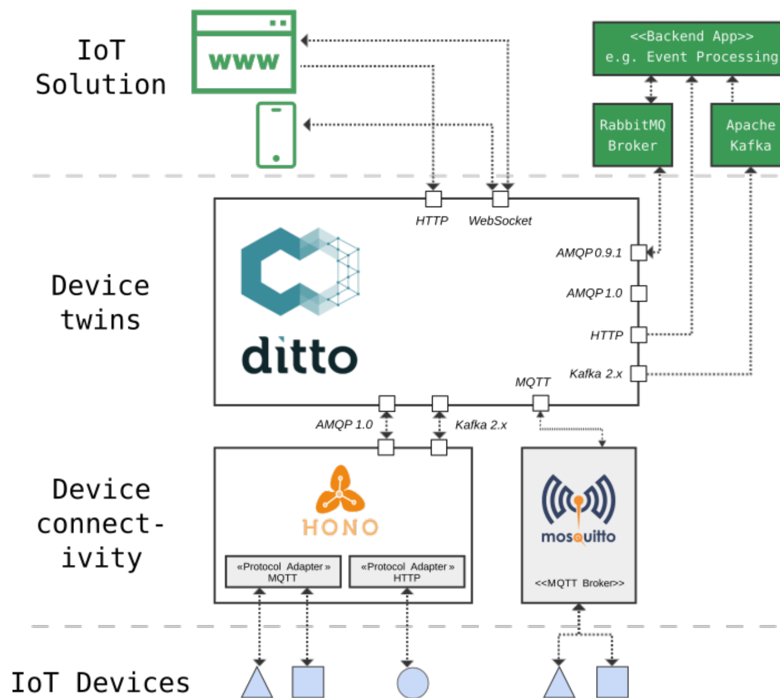

Figura 1.6: Schema del funzionamento di Eclipse Ditto.

IoT di interagire con gli oggetti fisici in modi diversi. La connessione attraverso Eclipse Ditto deve essere indirizzata verso uno strato di connettività che funge da intermediario tra Ditto e il dispositivo fisico. Alcuni esempi di questi strati di connettività sono Eclipse Hono ed Eclipse Mosquitto. Per quanto riguarda la connettività tra Ditto e le soluzioni IoT, Ditto offre connessioni tramite HTTP e WebSocket. Questa versatilità nelle connessioni consente una vasta gamma di interazioni tra i Digital Twins e gli oggetti fisici nell'ecosistema IoT. Le entità principali del modello sono:

- *Thing*: i "Thing" rappresentano gli elementi modellati come Digital Twins, ovvero qualsiasi tipo di asset fisico o dispositivo virtuale. Ogni Thing è identificato da un proprio ThingID e fa riferimento a una policy che ne definisce i permessi.Un Thing può includere una "Definition", che descrive le capacità e le caratteristiche del Digital Twin. Questa Definition è spesso modellata attraverso Eclipse Vorto. Successivamente, è possibile aggiungere attributi di qualsiasi tipo per descrivere un Thing. Gli attributi sono generalmente

utilizzati per definire tipi di dati statici, ovvero quelli con una bassa (quasi nulla) frequenza di aggiornamento. D'altro canto, se si desidera descrivere dati che cambiano frequentemente, si utilizzano le "features". Un Thing può contenere un numero arbitrario di features. Inoltre, sia gli attributi che le features possono essere descritti attraverso l'aggiunta di metadati, fornendo ulteriori informazioni e contestualizzazione alle informazioni associate al Digital Twin.

- *Feature*: le "Feature" in Eclipse Ditto sono anch'esse identificate da un identificativo simile ai "Thing" e sono soggette a regole ben precise riguardanti i caratteri permessi e quelli vietati, garantendo coerenza nelle denominazioni. Le Feature sono utilizzate per modellare le parti dinamiche di un Digital Twin e ognuna contiene una lista di proprietà. Queste proprietà possono essere categorizzate, e l'insieme di esse rappresenta un oggetto JSON; ciascuna può essere un valore semplice o un oggetto complesso, e sono permessi tutti i tipi supportati da JSON. È possibile specificare le "Desired Properties" per le Feature, che rappresentano uno stato desiderato delle proprietà. Queste possono essere utilizzate quando si è interessati al raggiungimento da parte di una proprietà di uno specifico stato. Tuttavia, attualmente Ditto non fornisce un'implementazione specifica per le Desired Properties, lasciando questa responsabilità al programmatore. Analogamente ai "Thing", le "Feature" possono avere una definizione che ne specifica meglio le capacità e i comportamenti. Gli standard utilizzati per definire queste caratteristiche sono Eclipse Vorto e Web of Things. Tuttavia, non viene garantito che i tipi delle proprietà delle Feature seguano quelli forniti dalla definizione, rendendo il sistema più flessibile ma richiedendo attenzione nella gestione dei tipi di dati.

- *Policy*: le Policy vengono utilizzate all'interno di Ditto per regolare l'accesso ai Thing e alle altre entità, a diversi livelli di granularità. Più precisamente, esse concedono l'accesso ad un Subject, specificando un accesso di READ/WRITE per una certa risorsa.

Azure Digital Twins

Azure Digital Twins di Microsoft rappresenta una delle soluzioni più versatili e complete nel contesto dei Digital Twins. Questo servizio cloud, appartenente alla categoria PaaS (Platform as a Service), consente la creazione di grafi di conoscenza basati su modelli digitali di interi ambienti. Azure Digital Twins offre una piattaforma robusta che permette la modellazione digitale di ambienti complessi. Attraverso questo servizio, è possibile creare rappresentazioni virtuali di asset fisici, spazi e interazioni all'interno di un ecosistema. Il supporto per la creazione di modelli digitali avanzati consente agli utenti di ottenere una visione dettagliata e interconnessa di sistemi complessi. Questa soluzione di Microsoft si distingue per la sua versatilità e completezza, fornendo strumenti avanzati per la gestione, l'analisi e l'ottimizzazione di ambienti fisici attraverso la rappresentazione digitale. In Azure Digital Twins, le entità logiche, che rappresentano i Digital Twins, vengono descritte utilizzando modelli definiti in un linguaggio JSON-like chiamato Digital Twins Definition Language (DTDL). Questi modelli delineano i Digital Twins in termini delle loro proprietà, dati telemetrici, eventi, componenti e relazioni. I modelli possono anche definire relazioni tra le varie entità, permettendo di collegare i diversi Digital Twins e creare un grafo di conoscenza che riflette le interazioni nell'ambiente. I modelli dei Digital Twins, una volta istanziati (analogamente a un'istanza di una classe in programmazione), diventano rappresentazioni dinamiche e aggiornate del mondo reale. Sfruttando le relazioni descritte nei modelli DTDL, è possibile connettere i Digital Twins, creando un grafo che riflette la complessità delle interazioni nell'ambiente. In Figura 1.7 è possibile notare un grafo di esempio di Azure Digital Twins di un progetto di esempio il quale si può trovare al seguente link.

Azure Digital Twins fornisce strumenti, come Azure Digital Twins Explorer, che consentono di visualizzare il grafo in modo intuitivo. Per estrarre dati dal grafo di conoscenza, il servizio offre un potente sistema di query tramite API, permettendo di ottenere proprietà, relazioni, informazioni sul modello e altro ancora. Inoltre, Azure Digital Twins integra un ricco sistema di eventi, consentendo di gestire il data processing e la logica di business. È possibile connettere risorse computazionali esterne, come le Azure Functions, per elaborare dati in modo

Figura 1.7: Grafo di esempio di Azuere Digital Twins.

scalabile e personalizzato, sfruttando appieno le potenzialità del servizio.

Capitolo 2

Microservizi ad eventi

L'evoluzione delle architetture software ha portato a una crescente adozione di paradigmi agili e distribuiti, tra i quali sono presenti i microservizi ad eventi. Quest'ultimi si basano sul concetto di comunicazione asincrona attraverso la generazione, la pubblicazione e la sottoscrizione di eventi. In questo contesto, gli eventi diventano il collante che collega le diverse parti del sistema, permettendo una comunicazione efficace e distribuita tra i diversi microservizi. Questo capitolo ha l'obiettivo di esplorare questo nuovo pattern architetturale, delineando i motivi alla base della loro adozione e presentando i concetti fondamentali che li caratterizzano. Inoltre, verrà descritto il design ed il contratto degli eventi così come i principali canali di comunicazione utilizzati ed alcuni pattern coinvolti. Infine, nell'ultima sezione si discuterà dei principali limiti e problemi legati all'utilizzo di queste architetture.

2.1 Un nuovo modo di comunicare

I microservizi e le architetture a microservizi esistono da molti anni, in forme diverse e spesso sotto nomi diversi. Le architetture orientate ai servizi (SOA) sono spesso composte da più microservizi che comunicano in modo sincrono direttamente tra loro [Bel20]. Le architetture a scambio di messaggi utilizzano eventi per comunicare in modo asincrono tra loro. La comunicazione basata sugli eventi non è certamente è una novità, ma la necessità di gestire grandi quantità di dati, su

larga scala e in tempo reale, è nuova e necessita di un cambiamento rispetto ai vecchi stili architetturali. In una moderna architettura a microservizi basata sugli eventi, i sistemi comunicano emettendo e consumando eventi. Questi eventi non vengono distrutti al momento del consumo come in sistemi a scambio di messaggi, ma rimangono invece prontamente disponibili per altri consumatori da leggere nel momento in cui lo richiedano. I servizi stessi sono piccoli e costruiti appositamente per soddisfare le necessità e gli obiettivi specifici dell'organizzazione. Questi servizi consumano eventi da flussi di dati di input, applicano la loro business logic ed in seguito possono emettere i propri eventi di output. Inoltre, possono fornire dati tramite API o eseguire altre azioni richieste.

I team, i sistemi e le persone di un'organizzazione devono tutti comunicare tra loro per raggiungere i loro obiettivi. Queste comunicazioni formano una topologia interconnessa di dipendenze chiamate strutture comunicative. Ci sono diverse tipologie di strutture comunicative e ciascuna di essi influisce sul modo in cui operano le imprese. Possono essere infatti a livello di business, a livello di implementazione ed a livello dei dati. Le strutture comunicative a livello dei dati comprendono i processi attraverso i quali vengono trasmessi e comunicati i dati in tutta l'azienda. Le strutture comunicative di un'organizzazione influenzano notevolmente il modo in cui vengono progettati i programmi. Questo vale anche a livello di team: le comunicazioni tra i team influenzano le soluzioni realizzate per i vari requisiti di business.

L'approccio event-driven offre un'alternativa alle strutture comunicative tradizionali. Le comunicazioni basate sugli eventi non sostituiscono le comunicazioni a richiesta-risposta, piuttosto costituiscono un modo completamente diverso di comunicare tra i servizi. Una struttura basata su stream di eventi disaccoppia la produzione e la proprietà dei dati dall'accesso ad essi. I servizi non sono più accoppiati tramite le API di richiesta-risposta, ma invece attraverso i dati definiti all'interno dei flussi di eventi.

In questo approccio, descritto in [Roc22] tutti i dati vengono pubblicati in una serie di flussi di eventi, formando un flusso continuo e una narrazione unica che descrive in dettaglio tutto ciò che è accaduto nell'organizzazione. Di conseguenza questo diventa il canale principale attraverso il quale i sistemi comunicano tra loro. Gli eventi non sono semplici segnali che indicano che i dati sono pronti da qualche

parte oppure solamante un mezzo per il trasferimento diretto dei dati. Piuttosto, fungono sia da archivio di dati che da mezzo di comunicazione asincrona tra i servizi.

Il flusso degli eventi all'interno di un'organizzazione diventa di conseguenza l'unica fonte di verità.

Le applicazioni possono ora accedere a dati che altrimenti sarebbero stati laboriosi da ottenere tramite collegamenti punto-punto. I nuovi servizi quindi possono semplicemente acquisire tutti i dati necessari dai flussi di eventi, successivamente creare i propri modelli ed il proprio stato ed eseguire i propri servizi senza dipendere dalla connessione diretta con un altro servizio.

I microservizi ad eventi consentono le operazioni sulla business logic necessarie per soddisfare i requisiti del proprio bouned context. Queste applicazioni sono incaricate di soddisfare questi requisiti e di emettere gli eventi necessari ad altri consumatori. I vantaggi principali nell'utlizzo di microservizi ad eventi sono:

- **Granularità**: i servizi si adattano perfettamente ai bounded context e possono essere facilmente riscritti quando cambiano i requisiti aziendali.

- **Scalabilità**: i servizi possono scalare in modo semplice a seconda della necessità.

- **Flessibilità tecnologica**: i servizi utilizzano i linguaggi e le tecnologie più appropriati. Questo permette di conseguenza una facile prototipazione utilizzando le ultime tecnologie.

- **Basso accoppiamento**: i microservizi ad eventi sono accoppiati sui dati del dominio ma non sull'implementazione delle API. Gli schemi degli eventi possono essere usati per migliorare la gestione del cambiamento di essi.

- **Testing**: i microservizi tendono ad avere molte meno dipendenze rispetto ai monoliti e di conseguenza risulta più facile individuare i test da eseguire.

Una topologia di un microservizio è la topologia basata sugli eventi interna a un singolo microservizio [Bel20]. Ciò definisce le operazioni da eseguire sugli eventi in entrata, come ad esempio la trasformazione, l'archiviazione o l'emissione. La

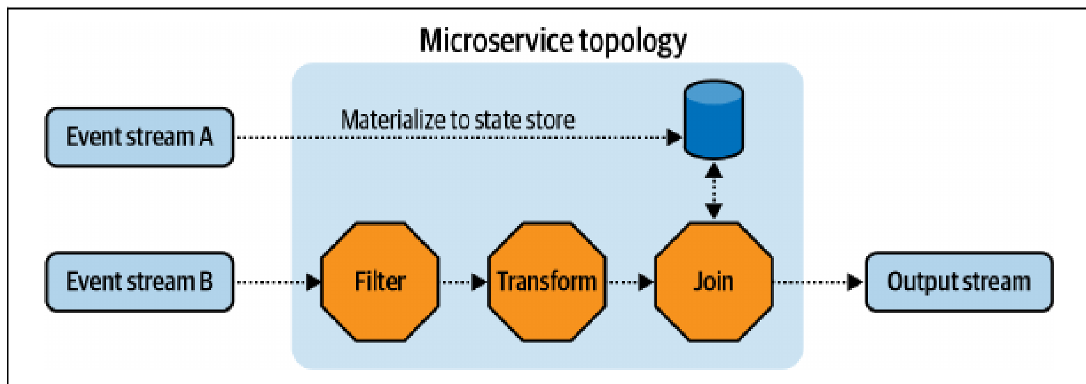

Figura 2.1: Topologia di esempio di un microservizio ad eventi.

topologia del microservizio in Figura 2.1, presente in [Bel20], consuma gli eventi dal flusso di eventi A e li materializza in un archivio dati. Nel frattempo, il flusso di eventi B viene consumato e alcuni eventi vengono filtrati, trasformati e quindi contribuiscono al cambiamento dello stato interno. I risultati successivamente vengono propagati su un nuovo flusso di eventi. Il consumo, l'elaborazione e l'output del microservizio fanno parte della topologia di esso.

D'altro canto, una topologia di business è l'insieme dei microservizi, stream di eventi e API necessari per determinate aree di business. Ricapitolando, la topologia del microservizio descrive il funzionamento interno di esso mentre lo topologia di business descrive le relazioni e le interazioni presenti tra i vari servizi.

La Figura 2.2, presente in [Bel20], mostra una topologia di business contenente tre microservizi indipendenti ed uno stream di eventi. Si può infatti notare come questa topologia evidenzia il comportamento complessivo e non il funzionamento interno dei microservizi.

Per quanto riguarda la comunicazione tra diversi microservizi, uno dei concetti più utilizzati è sicuramente il *Single Writer Principle*. Questo principio consiste infatti nell'utilizzo di un unico microservizio produttore per ogni flusso di eventi. Questo approccio chiarisce chiaramente quale microservizio è responsabile per la generazione di ciascun tipo di evento riducendo così il rischio di conflitti o inconsistenze all'interno del sistema. Ogni componente, infatti, si occupa solo degli eventi legati al proprio dominio, riducendo la complessità e migliorando la modularità del

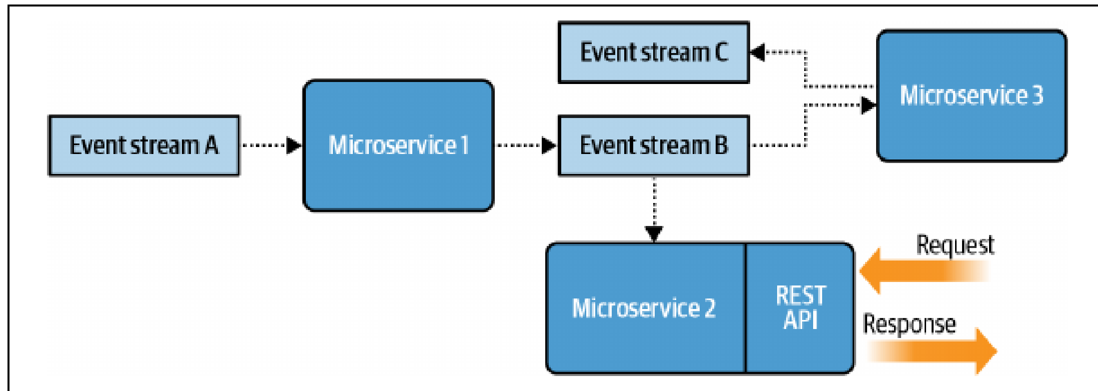

Figura 2.2: Topologia di business di un'architettura a microservizi ad eventi.

sistema.

2.2 Design e contratto degli eventi

Un evento può essere qualsiasi cosa accaduta nell'ambito di un'organizzazione aziendale. Una volta che questi eventi iniziano ad essere consumati, è possibile creare sistemi per sfruttarli e utilizzarli all'interno dell'organizzazione [Bel20]. Un evento è una registrazione di ciò che è accaduto, proprio come le informazioni e i registri degli errori di un'applicazione che registrano ciò che accade. A differenza di questi registri, tuttavia, gli eventi fungono anche come l'unica fonte di verità. In quanto tali, devono contenere tutte le informazioni necessarie per descrivere accuratamente quanto accaduto.

Esistono numerose best practice da seguire durante il design e la creazione di eventi [Bel20], oltre a diversi anti-pattern da evitare:

- *Dire tutta la verità*: una buona definizione di evento non è semplicemente un messaggio che indica che qualcosa è successo, ma piuttosto la descrizione completa di tutto ciò che è accaduto durante quell'evento. In termini aziendali, si tratta dei dati risultanti prodotti quando i dati di input vengono acquisiti e viene applicata la logica aziendale. Questo evento di output deve essere trattato come singola fonte di verità e deve essere registrato come

un fatto immutabile per il consumo da parte dei consumatori. Esso ha la piena e totale autorità su ciò che è realmente accaduto e i consumatori non dovrebbero aver bisogno di consultare altre fonti di dati per sapere che un simile evento ha avuto luogo.

- *Utilizzare un singolo schema per ciascuno Stream*: Uno stream di eventi dovrebbe contenere eventi che rappresentano un singolo evento logico. Non è consigliabile mescolare diversi tipi di eventi all'interno dello stesso stream, perché così facendo può confondere le definizioni di cosa sia l'evento e cosa rappresenti lo stream.
- *Minimizzare la grandezza degli eventi*: gli eventi funzionano bene quando sono piccoli, ben definiti e facilmente elaborabili. Tuttavia, i grandi eventi possono esistere in determinati contesti. Generalmente questi eventi più grandi rappresentano molte informazioni relative a un dato evento che sono necessarie per esprimere quanto accaduto.
- *Evitare di utilizzare eventi come semafori*: gli eventi non devono essere utilizzati per implementare meccanismi di sincronizzazione in quanto in quel caso l'evento non funge da unica fonte di verità ma per comprenderne il significato è necessario consumare anche tutti gli altri eventi dello stream.

Gli eventi vengono generalmente rappresentati utilizzando un formato chiave/-valore. Il valore memorizza i dettagli completi dell'evento, mentre la chiave viene utilizzata per scopi di identificazione, instradamento e operazioni di aggregazione su eventi con la stessa chiave. Tipicamente la chiave non è un campo obbligatorio. Ci sono tre tipologie principali di eventi:

1. **Eventi senza chiave**: gli eventi che non contengono una chiave sono usati per descrivere un evento come una dichiarazione di un fatto avvenuto. Un esempio potrebbe essere un evento che indica il fatto che un cliente ha interagito con un prodotto.
2. **Eventi entità**: con entità si intende un oggetto dotato di un identificatore. Gli eventi entità sono utilizzati quindi per descrivere le proprietà e lo stato di un oggetto appartenente al dominio del servizio. La chiave dell'evento

di conseguenza è la stessa chiave dell'entità. Questi eventi sono utilizzati per fornire la storia dello stato dell'entità e di conseguenza lo stato corrente dell'entità è rappresentato dall'ultimo evento ricevuto.

3. **Eventi con chiave**: sono tutti quegli eventi che contengono una chiave ma non rappresentano un'entità. Sono utilizzati tipicamente per partizionare gli stream ed eseguire operazioni di aggregazione sulla chiave.

Il produttore e il consumatore devono avere una comprensione comune del messaggio. In caso contrario, potrebbe essere interpretato erroneamente e la comunicazione risulterà incompleta. Il formato dei dati da comunicare e la logica con cui vengono creati formano il contratto dell'evento. Questo contratto è seguito sia dal produttore che dal consumatore dell'evento e fornisce quindi la forma ed il significato ad esso a seconda del contesto in cui si trova. Esistono due componenti di un contratto: il primo riguarda la definizione dei dati o cosa verrà prodotto (ovvero i campi, i tipi e le varie strutture dati), il secondo è la logica scatenante, o il motivo per cui viene prodotto (l'attività specifica che ha innescato la creazione dell'evento). È possibile apportare modifiche sia alla definizione dei dati che alla logica di attivazione man mano che i requisiti aziendali evolvono.

È necessario prestare attenzione quando si modifica il contratto dell'evento, in modo da non eliminare o alterare i campi utilizzati dai consumatori. Il modo migliore per applicare i contratti e garantire coerenza è definire uno schema per ogni evento. Il produttore definisce uno schema esplicito che descrive in dettaglio la definizione dei dati e la logica di attivazione, con tutti gli eventi dello stesso tipo che aderiscono a questo formato. In tal modo, il produttore fornisce un meccanismo a cui comunicare il formato dell'evento a tutti i potenziali consumatori. I consumatori, a loro volta, possono costruire con sicurezza la propria logica di business sulla base dei dati schematizzati. Un consumatore deve essere in grado di estrarre i dati necessari per i propri processi aziendali, e non può farlo senza avere una serie di aspettative su quali dati dovrebbero essere disponibili. Inoltre, esiste un rischio sostanziale nel non utilizzare schemi espliciti e richiedere quindi a ciascun consumatore di interpretare i dati in modo indipendente: un consumatore potrebbe interpretarli in modo diverso rispetto ad altri, il che porta a visioni incoerenti dell'unica fonte di verità.

Evoluzione del contratto dei dati

Lo schema deve necessariamente supportare una gamma completa di regole di evoluzione. L'evoluzione dello schema consente ai produttori di aggiornare il formato di output dei propri eventi consentendo al tempo stesso ai consumatori di continuare a consumare gli eventi senza interruzioni. Le modifiche potrebbero essere l'aggiunta di nuovi campi, la deprecazione dei vecchi campi o la modifica dello scope di un campo. Un framework per la gestione dell'evoluzione dello schema garantisce che questi cambiamenti possano avvenire in modo sicuro e che produttori e consumatori possano essere aggiornati indipendentemente l'uno dall'altro. Gli aggiornamenti ai servizi diventano proibitivi senza il supporto all'evoluzione dello schema. Produttori e consumatori sono costretti a coordinarsi continuamente e i vecchi dati precedentemente compatibili potrebbero non essere più compatibili con i sistemi attuali. È irragionevole aspettarsi che i consumatori aggiornino i propri servizi ogni volta che un produttore modifica lo schema dei dati. Infatti, un principio fondamentale dei microservizi è che dovrebbero essere indipendenti dai cicli di rilascio di altri servizi, tranne in casi eccezionali. Un insieme di regole di evoluzione dello schema contribuisce notevolmente a consentire sia ai consumatori che ai produttori di aggiornare le proprie applicazioni con i propri tempi. Queste regole sono note come tipi di compatibilità:

- *Forward Compatibility*: consente di leggere i dati prodotti con uno schema più recente come se fossero prodotti con uno schema precedente. Questo è un requisito evolutivo particolarmente utile in un'architettura guidata dagli eventi, poiché il modello più comune di modifica del sistema inizia con il produttore che aggiorna la propria definizione di dati e produce dati con lo schema più recente. Il consumatore è tenuto solo ad aggiornare la sua copia dello schema e del codice nel caso abbia bisogno di accedere ai nuovi campi.

- *Backward Compatibility*: consente di leggere i dati prodotti con uno schema precedente come se fossero prodotti con uno schema più recente. Ciò consente a un consumatore di dati di utilizzare uno schema più recente per leggere i dati più vecchi.

- *Full Compatibility*: l'unione di forward e backward compatibility, questa è la garanzia più forte e quella che si dovrebbe utilizzare quando possibile. È sempre possibile allentare i requisiti di compatibilità in un secondo momento, ma spesso è molto più difficile renderli più restrittivi.

Formato degli eventi

Sebbene siano disponibili molte opzioni per la formattazione e la serializzazione, i contratti degli eventi vengono soddisfatti al meglio con formati come *Avro*, *Json* o *Protobuf*. Alcuni dei maggiori event broker supportano la serializzazione e la deserializzazione di eventi codificati con questi formati. Ad esempio, sia Apache Kafka che Apache Pulsar supportano i formati JSON, Protobuf e Avro.

Avro è un formato binario efficiente per la serializzazione di dati. Offre uno schema flessibile che consente di evolvere i dati nel tempo ed è ampiamente utilizzato in ambito Big Data e da tecnologie come Apache Kafka. Nel listato 2.1 viene riportato un evento di esempio in formato *Avro*.

Protobuf è un formato binario compatto e veloce per la serializzazione di dati. Richiede uno schema rigido che definisce in modo preciso i dati, ma è molto efficiente in termini di spazio di archiviazione e velocità di elaborazione. Nel listato 2.2 viene riportato un evento di esempio in formato *Protobuf*.

JSON è un formato di testo leggibile e facilmente interpretabile dagli esseri umani. Ha uno schema flessibile simile ad Avro ed è ampiamente utilizzato per la comunicazione tra applicazioni web e API. Nel listato 2.3 viene riportato un evento di esempio in formato *Json*.

L'alternativa all'utilizzo di questi formati può essere l'utilizzo di testo semplice utilizzando coppie chiave/valore, che offre comunque una certa struttura ma non fornisce schemi espliciti o gestione dell'evoluzione dello schema.

Riassumendo, le architetture asincrone basate su eventi fanno molto affidamento sulla qualità degli eventi. Gli eventi di alta qualità sono definiti esplicitamente con uno schema evolutivo, hanno una logica di attivazione ben definita e includono definizioni complete dello schema con commenti e documentazione. Gli schemi impliciti, sebbene più facili da implementare e mantenere per il produttore, scaricano gran parte del lavoro di interpretazione sul consumatore. Sono inoltre più soggetti

Listing 2.1: Esempio di evento in formato Avro.

```
{
  "type": "record",
  "name": "Acquisto",
  "namespace": "esempio",
  "fields": [
    {
      "name": "id",
      "type": "long"
    },
    {
      "name": "timestamp",
      "type": "long"
    },
    {
      "name": "cliente_id",
      "type": "long"
    },
    {
      "name": "prodotto_id",
      "type": "long"
    },
    {
      "name": "prezzo",
      "type": "double"
    },
    {
      "name": "quantita",
      "type": "int"
    }
  ]
}
```

Listing 2.2: Esempio di evento in formato Protobuf.

```
message Acquisto {
  int64 id = 1;
  int64 timestamp = 2;
  int64 cliente_id = 3;
  int64 prodotto_id = 4;
  double prezzo = 5;
  int32 quantita = 6;
}
```

Listing 2.3: Esempio di evento in formato Json.

```
1 {
2   "id": 12345,
3   "timestamp": 1644988400000,
4   "cliente_id": 10,
5   "prodotto_id": 20,
6   "prezzo": 10.99,
7   "quantita": 1
8 }
```

a guasti imprevisti dovuti alla mancanza di dati sugli eventi e a modifiche involontarie. Gli schemi espliciti sono una componente essenziale per un'adozione diffusa di architetture guidate dagli eventi, in particolare quando un'organizzazione cresce e diventa impossibile trasmettere la conoscenza a livello organizzativo. Un evento dovrebbe rappresentare un fatto specifico e contenere i campi appropriati per registrare ciò che è accaduto. Questi eventi costituiscono la narrazione ufficiale delle operazioni aziendali e possono essere utilizzati da altri microservizi per le proprie esigenze. L'evoluzione dello schema è un aspetto molto importante degli schemi espliciti, poiché consente un meccanismo di modifica controllato per il modello del dominio degli eventi. È normale che un modello di dominio si evolva, in particolare quando emergono nuovi requisiti aziendali e l'organizzazione si espande. L'evoluzione dello schema consente infatti a produttori e consumatori di isolarsi dai cambiamenti che non sono essenziali per le loro operazioni, permettendo invece ad altri servizi che sono interessati delle modifiche di aggiornarsi di conseguenza. In alcuni casi l'evoluzione dello schema non è possibile ed è necessario che si verifichi un cambiamento sostanziale. I produttori e i consumatori devono comunicare le ragioni alla base dei cambiamenti radicali e riunirsi per ridefinire il modello di dominio.

2.3 Canali di comunicazione

L'efficace funzionamento dei microservizi ad eventi dipende fortemente da un'infrastruttura di comunicazione robusta, capace di facilitare lo scambio affidabile,

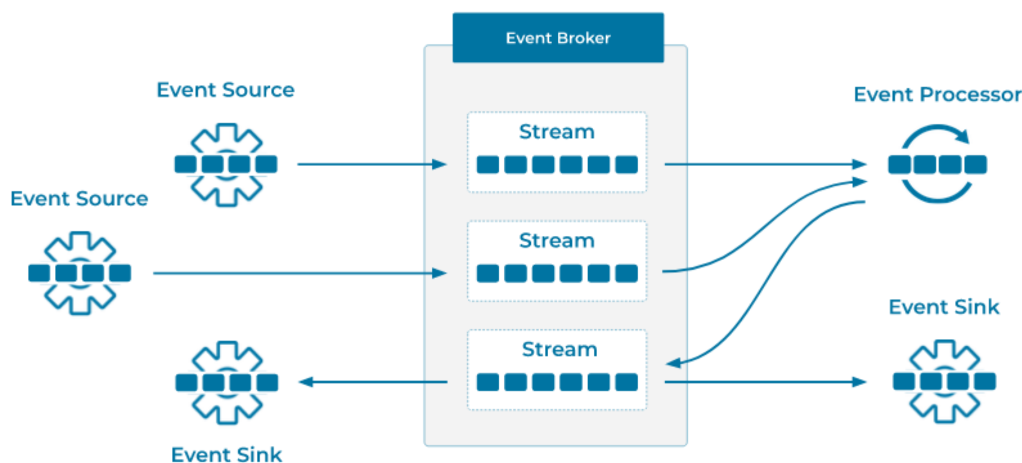

Figura 2.3: Schema funzionamento di un Event Broker.

scalabile e distribuito degli eventi tra i vari servizi. Questa sezione si propone di esaminare da vicino i concetti chiave relativi ai canali di comunicazione , con un focus particolare sugli event broker e message broker.

Event Broker

Un *Event Broker* è un componente che si concentra specificamente sulla gestione e distribuzione degli eventi all'interno di un sistema basato su microservizi. Come si può notare in Figura 2.3, presente nell'articolo al seguente link, la sua funzione principale è quella di ricevere, instradare e distribuire gli eventi ai microservizi interessati. Gli event broker sono progettati per supportare il paradigma delle architetture ad eventi, offrendo caratteristiche come la pubblicazione e la sottoscrizione degli eventi. Alcune caratteristiche di un Event Broker sono:

- *Orientato agli eventi*: l'Event Broker è strettamente orientato alla gestione degli eventi. Si occupa della ricezione e della distribuzione di eventi all'interno del sistema.

- *Pubblicazione e Sottoscrizione*: fornisce un modello di comunicazione basato sulla pubblicazione e sottoscrizione. I microservizi possono pubblicare eventi

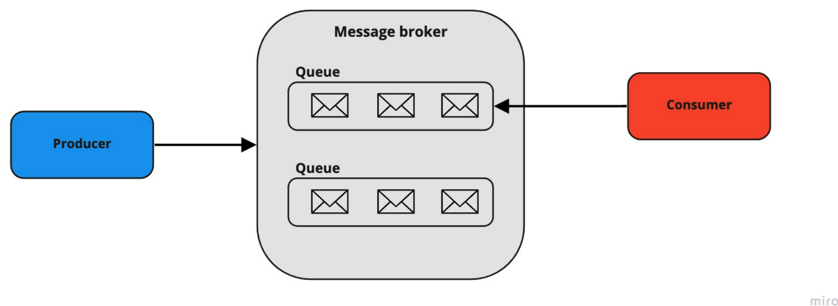

Figura 2.4: Schema funzionamento di un Message Broker.

a cui altri microservizi possono sottoscriversi per ricevere notifiche quando quegli eventi si verificano.

- *Decoupling*: aiuta a ridurre le dipendenze dirette tra i microservizi. I microservizi non hanno bisogno di conoscere direttamente gli altri servizi, ma possono interagire attraverso l'Event Broker.

- *Comunicazione asincrona*: supporta la comunicazione asincrona, consentendo ai microservizi di operare in modo indipendente e migliorando la resilienza del sistema.

- *Gestione del Flusso di Eventi*: gestisce il flusso di eventi, assicurando che vengano consegnati in modo affidabile ai servizi interessati.

Message Broker

Un *Message Broker* è un componente più generico che gestisce la comunicazione asincrona tra i vari servizi di un sistema distribuito. Mentre può gestire una varietà di messaggi, inclusi gli eventi, non è specificamente focalizzato solo sugli eventi. Come si può notare in Figura 2.4, presente nell'articolo al seguente link, un Message Broker utilizza tipicamente delle code per orchestrare l'invio e la ricezione di messaggi tra i vari componenti del sistema. Alcune caratteristiche chiave di un Message Broker sono:

- *Comunicazione asincrona*: supporta la comunicazione asincrona tra i microservizi. I microservizi possono inviare e ricevere messaggi in modo indipendente.
- *Instradamento dei messaggi*: offre funzionalità avanzate di instradamento e smistamento dei messaggi. I messaggi possono essere indirizzati a destinatari specifici in base a criteri definiti.
- *Supporto a vari tipi di messaggi*: gestisce una varietà di messaggi, inclusi quelli che non sono strettamente eventi. Può supportare messaggi di comando, query e altri tipi di comunicazione.
- *Scenari di utilizzo ampi*: èuò essere utilizzato in una vasta gamma di scenari di comunicazione, non limitandosi alla sola gestione degli eventi.

La scelta tra Event Broker e Message Broker dipende dalle esigenze specifiche del sistema e dai requisiti funzionali. In molti casi, potrebbe essere vantaggioso utilizzare una combinazione di entrambi, sfruttando le loro rispettive forze per garantire una comunicazione efficiente e affidabile.

2.4 Pattern e metodologie

L'evoluzione delle architetture a microservizi ha portato all'adozione di nuovi approcci che affrontano sfide specifiche nell'implementazione di sistemi distribuiti. In questa sezione, si esploreranno alcuni concetti chiave, pattern e metodologie per la gestione dello stato e per la comunicazione e interazione tra diversi servizi.

Event Sourcing

Il paradigma *Event Sourcing* propone un approccio innovativo alla gestione dello stato delle applicazioni. Al contrario del tradizionale modello basato sullo stato corrente, l'Event Sourcing adotta un approccio innovativo registrando ogni cambiamento di stato come un evento immutabile [MP23]. Questo approccio non solo preserva lo storico degli eventi, ma offre anche la possibilità di ricostruire lo stato dell'applicazione in qualsiasi momento. Ogni evento, come illustrato in Figura 2.5

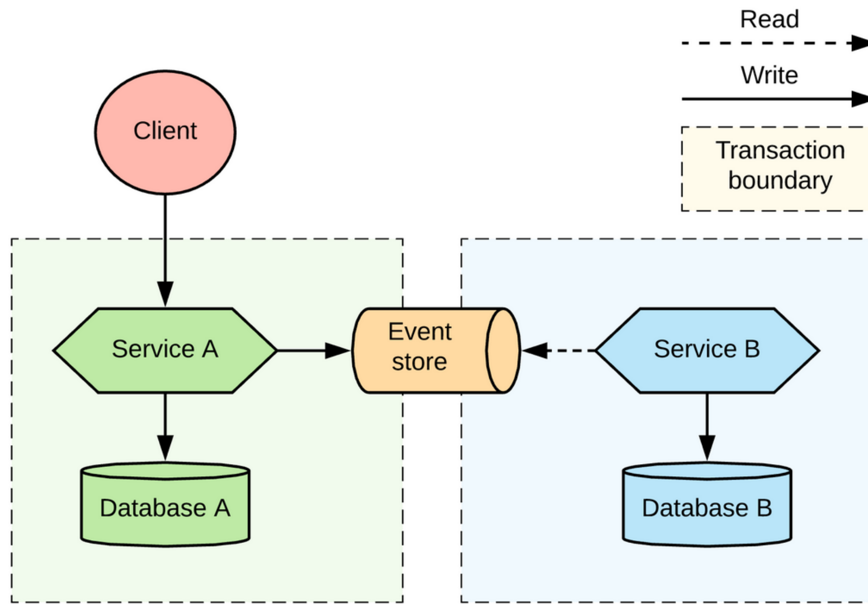

Figura 2.5: Schema di funzionamento del pattern Event Sourcing.

presente in questo articolo, viene salvato all'interno di un *Event Store* ed una volta registrato, diventa immutabile. Questo aspetto facilita la tracciabilità e la riproducibilità del sistema in qualsiasi punto nel tempo.

Coreography vs Orchestration

I microservizi, per definizione, operano solo su una piccola parte del workflow complessivo di un'organizzazione. Un workflow è un insieme di azioni che compongono un processo aziendale, comprese eventuali diramazioni logiche e azioni compensative. I flussi di lavoro richiedono comunemente più microservizi, ciascuno con il proprio buonded context, che esegue i propri compiti ed emette nuovi eventi per i consumatori. In questo contesto si distinguono due diversi approcci per la comunicazione e l'interazione tra diversi microservizi: *coreography* e *orchestration*.

Il pattern orchestration, illustrato in Figura 2.6 in [Bel20], è basato sull'idea che un sistema centrale controlla tutte le interazioni tra i vari elementi del sistema, cioè i microservizi. In questo approccio la parola chiave è "supervisione": il controllo del rispetto delle singole istruzioni. Nell'orchestration, infatti, esiste un servizio che fa da controller e che gestisce le comunicazioni tra i singoli microser-

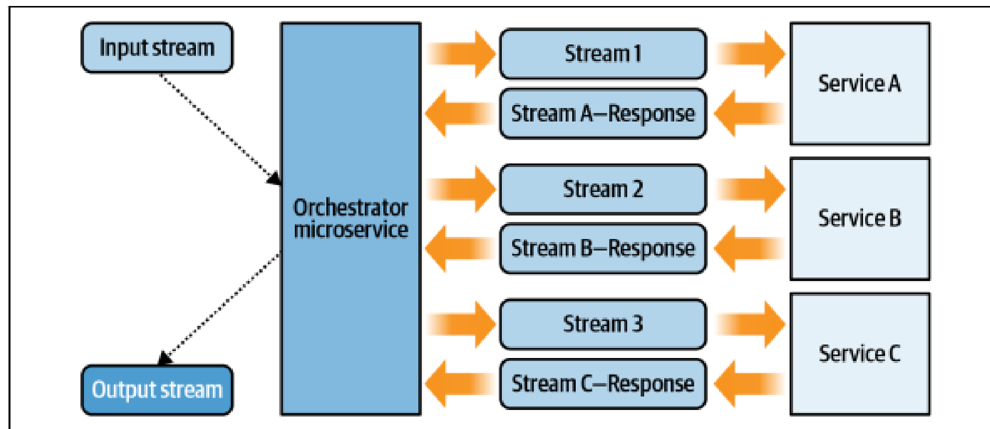

Figura 2.6: Workflow di esempio del pattern Orchestration.

vizi. Così facendo garantisce che ciascun servizio esegua la parte che gli è stata assegnata. Il controller è un vero e proprio middleware, cioè un software che fornisce alle applicazioni delle funzionalità e dei servizi comuni. Nel caso particolare di questo pattern, la componente di middleware ha la funzione di supervisore che controlla le interazioni tra i vari microservizi. L'orchestration può essere utilizzata per una serie ampia di compiti. Per esempio, *Kubernetes* consente di gestire in maniera dichiarativa le risorse orchestrate astraendo il livello fisico sottostante, in modo da essere gestito come un unico pool di risorse computazionali. In sintesi, l'approccio "orchestrato" si basa sull'idea di creare un sistema di gestione dei processi di business centralizzato e ben organizzato che sia in grado di dare stabilità all'applicazione e renderne lo stato facilmente misurabile e modificabile. Nell'orchestration il controller deve comunicare direttamente con ciascun servizio per potergli indicare cosa deve fare e deve quindi attendere che la comunicazione sia stabilita e il servizio risponda. Quando l'architettura è costituita da centinaia o migliaia di microservizi, si possono creare problemi di disponibilità del servizio o latenze eccessive. Esiste un limite oltre il quale un controller centrale non riesce più a gestire in maniera efficiente questo tipo di architettura. In questo, caso è come se si fosse creata un'applicazione monolitica all'interno di un ambiente distribuito.

D'altro canto, il pattern coreography, illustrato in Figura 2.7 in [Bel20], utilizza un approccio completamente diverso e fondamentalmente decentralizzato nel quale nessun elemento esegue un ordine esplicito, bensì ciascuno conosce una se-

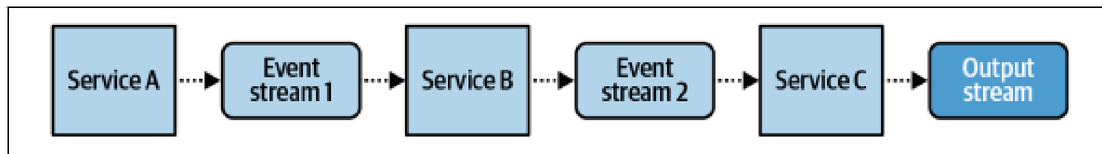

Figura 2.7: Workflow di esempio del pattern Coreography.

rie di azioni che deve compiere in relazione agli altri elementi con cui interagisce. L'approccio coreografico si basa sull'idea che un elemento di controllo centrale, l'orchestratore, sia ridondante. I singoli componenti, cioè i microservizi, devono essere capaci di autogestirsi senza intervento esterno e accoppiati in maniera debole, cioè in modo largo e non troppo serrato. Questo vuol dire che i microservizi devono essere in grado di fornire da soli il maggior valore possibile per il business, senza avere un impatto diretto sugli altri componenti. In questa maniera la complessità si riduce perché non c'è un controller da programmare e gestire. Inoltre, come intuibile, non c'è un unico nodo, cioè un elemento potenzialmente critico per tutta l'infrastruttura in caso di problemi. La coreografia avviene secondo una sequenza di step. Ogni microservizio che ha svolto un'azione spedisce un evento su determinati topic. Gli altri microservizi, iscritti a quello specifico topic, ricevono il messaggio. A quel punto sanno da soli cosa devono fare, perché sono progettati per rispondere in automatico a determinati eventi. L'utilizzo di questo pattern consente di cambiare i microservizi (aggiungendone o togliendone a seconda del bisogno) senza che la logica sottostante venga resa inservibile e debba essere riscritta.

Alla luce dei benefici e delle limitazioni descritte, la decisione di quale pattern adottare deve essere presa sulla base di diversi fattori. Nella decisione peseranno il tipo di progetto che si vuole realizzare, i bisogni di business, gli obiettivi che devono essere raggiunti, il team e le risorse a disposizione. In linea di massima, se vogliamo favorire la scalabilità del progetto e siamo preparati a gestire una complessità elevata possiamo considerare l'approccio Choreography. Viceversa, se la nostra priorità è ottenere controllo per dare stabilità all'applicazione e renderla più facilmente misurabile e modificabile, possiamo valutare l'Orchestration o l'approccio ibrido.

2.5 Limiti e problematiche

L'adozione di microservizi ad eventi risulta sicuramente una strategia chiave nella progettazione di architetture flessibili e scalabili. Tuttavia, come con qualsiasi innovazione tecnologica, questa transizione presenta alcune sfide cruciali che richiedono un'attenzione particolare. In questo contesto, una delle questioni più rilevanti da affrontare è di certo la perdita di eventi. Nel contesto di un ambiente distribuito, la perdita di eventi rappresenta un rischio sostanziale per la coerenza e l'affidabilità del sistema. Questa perdita può derivare da una serie di fattori, tra cui problemi di rete come pacchetti persi o ritardi prolungati. Inoltre, la mancanza di un adeguato meccanismo di persistenza degli eventi può amplificare questo problema, specialmente quando si verificano guasti del sistema o dei microservizi. La dinamicità di queste architetture, con l'introduzione e la rimozione di nuovi microservizi, può anch'essa contribuire alla perdita di eventi e all'inevitabile incoerenza dello stato del sistema. Oltre alla perdita, un altro aspetto cruciale riguarda sicuramente l'ordinamento degli eventi. La natura asincrona degli eventi rende complesso stabilire un ordine temporale preciso tra di essi. Gli eventi inviati simultaneamente possono essere ricevuti in ordine diverso, dando luogo a risultati imprevisti e a uno stato del sistema incoerente. Per affrontare queste sfide, è necessario mettere in campo prima di tutto dei meccanismi per la persistenza degli eventi. Infatti, la persistenza assicura che gli eventi siano conservati anche in situazioni di guasto o interruzione temporanea preservando così la coerenza dello stato. Inoltre, un sistema efficace di monitoraggio e logging gioca un ruolo fondamentale nell'individuare rapidamente eventuali perdite di eventi o problemi di ordinamento, semplificando l'identificazione e la risoluzione delle problematiche. In questo contesto, la scelta di un Event Broker adeguato svolge un ruolo fondamentale. Esso, infatti, può migliorare significativamente la robustezza e l'affidabilità di un sistema basato su microservizi ad eventi. In particolare, è importante comprendere quali meccanismi di garanzia di consegna sono utilizzati nell'Event Broker scelto.

Un altro svantaggio di questo pattern architetturale riguarda l'overhead di latenza: la gestione degli eventi introduce una latenza aggiuntiva, specialmente quando è necessario garantire l'ordinamento preciso o quando si richiede una persistenza elevata degli eventi. Infatti, per garantire l'ordinamento degli even-

ti, potrebbe essere necessario introdurre meccanismi aggiuntivi che ne rallentano l'elaborazione e la consegna, aumentando così la latenza complessiva. Inoltre, la distribuzione degli eventi su un ambiente distribuito introduce ulteriori ritardi, specialmente quando i microservizi sono implementati su nodi geograficamente distanti. Infine, la latenza può aumentare anche a causa di problemi di rete, come la congestione o la perdita di pacchetti.

Capitolo 3

Web of Digital Twins

In questo capitolo, si introdurrà il concetto di *Web of Digital Twin (WoDT)*, una proposta presente in letteratura e presentata in [RCM+22b]. Inizialmente, si offre una visione d'insieme del contesto in cui questa prospettiva si colloca, insieme alle motivazioni che hanno condotto alla sua formulazione. Successivamente, viene presentato il modello di Digital Twin adottato in WoDT, con una definizione delle sue caratteristiche principali, del ciclo di vita e del suo approccio ad eventi. Infine, verrà introdotto e descritto un framework a supporto di questa visione. Quest'utltimo sarà necessario per la comprensione del caso di studio svolto.

3.1 Panoramica

Uno dei problemi principali nel contesto dei Digital Twin riguarda l'isolamento delle attuali soluzioni o visioni. L'utilizzo di interfacce e piattaforme proprietarie e closed-source rende estremamente difficile la creazione di ecosistemi di Digital Twins in grado di collaborare e interoperare efficacemente per massimizzare il valore delle informazioni offerte.

In risposta a questa sfida, è stato proposto il concetto di Web of Digital Twins in [RCM+22b]. Questa proposta mira a sfidare la visione dominante nella ricerca e nell'ambito aziendale, che considera i Digital Twins come virtualizzazioni di asset fisici all'interno di ambienti o piattaforme closed-source e difficilmente interoperabili. L'obiettivo è quello di creare un ecosistema di Digital Twins che possano

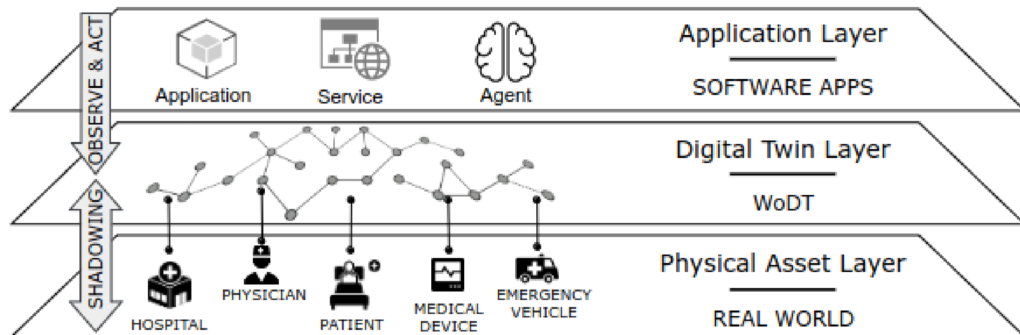

Figura 3.1: Layer presenti nella visione Web of Digital Twins.

comunicare in modo efficiente tra diverse organizzazioni, garantendo un elevato grado di interoperabilità.

Nel contesto del Web of Digital Twins, il concetto di Digital Twin non è più strettamente legato a una singola organizzazione, ma è utilizzato come un servizio grazie alla stratificazione dell'architettura proposta. Come si può notare in Figura 3.1, presente in [RCM+22b], il Web of Digital Twins definisce quindi un layer che facilita il collegamento tra il mondo reale e quello digitale, su cui vengono sviluppate applicazioni che interagiscono con gli asset fisici.

In parole più semplici, il Web of Digital Twins consente di riflettere il mondo reale in un ambiente digitale, rappresentando ogni asset del mondo reale come un Digital Twin nello spazio virtuale. Ciò assicura l'esistenza di una copia sempre aggiornata, utilizzata per prendere decisioni e assistere durante le attività svolte nel mondo reale.

In letteratura, l'idea più vicina a questa visione pervasiva dei Digital Twins è definita dal concetto di *Mirror Worlds*, presentato da D. Gelernter in [Gel93].

Il Web of Digital Twins può essere considerato un approccio pratico per progettare e realizzare Mirror Worlds sfruttando il paradigma dei Digital Twins. Pertanto, esso si configura come un ecosistema aperto e dinamico, rappresentato tramite un grafo in cui i nodi corrispondono ai Digital Twins stessi e gli archi indicano le relazioni che essi stabiliscono per modellare le connessioni presenti nella realtà. In questa prospettiva, i Digital Twins sono concepiti come entità software attive,

dotate di un proprio ciclo di vita. Possono riflettere lo stato della corrispondente entità fisica e offrire funzionalità aggiuntive. Al fine di garantire l'interoperabilità a livello applicativo e rendere i Digital Twins trasversali e non legati a un settore specifico, essi espongono interfacce di programmazione delle applicazioni (API).

3.2 Modello dei Digital Twin

In Web of Digital Twins (WoDT), ogni Digital Twin si basa su un modello M corrispondente ad un Physical Asset (PA), che corrisponde a qualsiasi entità che ha una manifestazione o rilevanza nel mondo fisico e un tempo di vita ben definito. Il modello definisce come il PA sia rappresentato nel livello digitale. Questa rappresentazione è definita in termini di:

- *Proprietà*: rappresentano gli attributi osservabili dell'asset fisico, sotto forma di valori (variabili) che possono cambiare dinamicamente in base all'evoluzione dello stato.
- *Eventi*: rappresentano gli eventi, a livello di dominio, che possono essere osservati nel asset fisico.
- *Relazioni*: rappresentano le relazioni dell'asset fisico con altri asset, attraverso collegamenti ad altri Digital Twins. Come le proprietà, anche le relazioni possono essere osservabili, create dinamicamente e cambiano nel tempo. A differenza delle proprietà, queste non riguardano esclusivamente lo stato locale dell'asset fisico, ma consentono di fare riferimento ad altri asset, rappresentati dai rispettivi Digital Twins.
- *Azioni*: rappresentano le azioni invocabili sul PA tramite l'interazione con il DT.

Defínto il modello M, lo stato dinamico S_{DT} di un Digital Twin è definito dalla tupla:

$$S_{DT} =< P, E, R, A, t >$$

dove:

- P: rappresenta l'insieme delle proprietà e dei relativi valori
- E: rappresenta la sequenza di eventi generati al momento t
- R: rappresenta l'insieme delle relazioni del DT
- E: rappresenta l'insieme delle azioni disponibili
- t: rappresenta il timestamp ossia il tempo corrente dell'asset fisico.

Grazie a questa modellazione, ogni cambio di stato o ogni evento dell'asset fisico rilevante per il modello viene rappresentato come un evento interno al Digital Twin che ne causerà il cambio di stato. Di conseguenza, risulta necessario l'adozione di un approccio event-driven anche per il processo di shadowing. Riassumendo, il processo di aggiornamento del DT (shadowing) è composta da 3 step principali:

1. Ciascun cambiamento rilevante dell'asset fisico viene modellato in un evento e_{PA}.
2. L'evento viene propagato al Digital Twin
3. Una volta ricevuto l'evento, lo stato del Digital Twin viene aggiornato a seconda della funzione di shadowing definita, che dipende di conseguenza dal modello M.

Inoltre, il processo di shadowing deve essere in grado di considerare molteplici sorgenti di eventi e informazioni. Infatti, considerando la proprietà di composizione dei DT, le sorgenti di eventi che influiscono sul processo di aggiornamento del DT possono essere a loro volta dei DT. Questo concetto inerente alla composizione di Digital Twins risulta di fondamentale importanza e verrà spiegato meglio in seguito in quanto è oggetto del caso di studio realizzato. Il processo di shadowing, inoltre, consente al DT di riflettere e invocare anche le possibili azioni dell'asset fisico. Prendendo come esempio il DT di una lampadina, questo dovrà fornire le azioni di accensione e spegnimento.

Per quanto riguarda il modello di interazione, ossia le primitive con le quali interagire con il Digital Twin, sono presenti diverse funzionalità:

- *Azioni digitali*: le applicazioni possono comunicare con i gemelli digitali (DT) utilizzando le azioni previste dal loro modello. Per fare questo, si utilizza il sistema ad eventi descritto in precedenza:

  1. L'applicazione invia una richiesta di esecuzione di un'azione, che genera un evento *eDT*.
  2. La funzione di shadowing intercetta l'evento eDT e lo trasforma in un evento ePA.
  3. L'evento ePA viene inviato all'asset fisico, che esegue l'azione richiesta.
  4. L'esecuzione dell'azione può modificare lo stato dell'asset fisico.
  5. La modifica dello stato dell'asset fisico può a sua volta modificare lo stato del DT

  Questo sistema di interazione consente di mantenere una rappresentazione coerente dello stato del DT e dell'asset fisico e permette di controllare gli asset fisici in modo remoto e automatizzato.

- *Osservazione di eventi*: le applicazioni esterne possono monitorare le modifiche dello stato di un DT attraverso un sistema di notifiche basato su eventi.

  1. Le applicazioni si sottoscrivono agli eventi di interesse specificando quali proprietà, eventi o relazioni del modello del DT desiderano monitorare.
  2. Quando il valore di una proprietà monitorata cambia, o si verifica un evento o una modifica in una relazione a cui l'applicativo è sottoscritto, viene generata una notifica.
  3. L'applicativo riceve la notifica e può quindi aggiornare la sua rappresentazione dello stato del DT.

  Questo approccio permette alle applicazioni di rimanere aggiornate sullo stato del DT in tempo reale e consente di realizzare sistemi di monitoraggio e controllo efficienti.

- *Query*: oltre al tracciamento delle modifiche, le applicazioni possono interrogare lo stato di un DT e del grafo di cui fa parte in due modi:

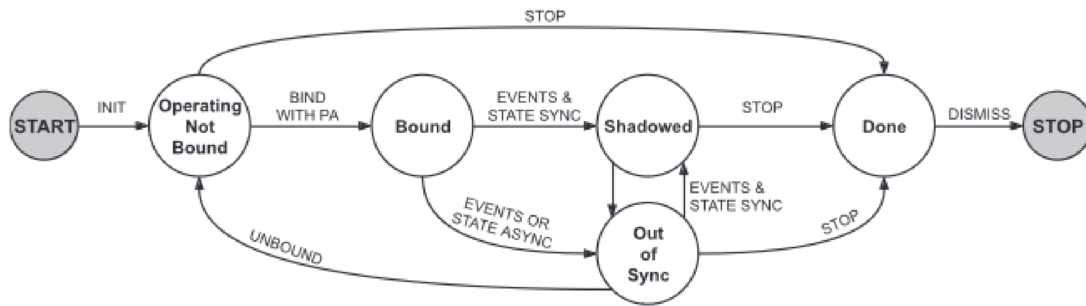

Figura 3.2: Diagramma degli stati del ciclo di vita di un Digital Twin in WoDT.

- Richieste semplici: l'applicazione può recuperare il valore di una specifica proprietà del DT. La richiesta può essere formulata in modo semplice e diretto.
- Richieste complesse: l'applicazione può utilizzare linguaggi di query come *SPARQL* per interrogare il DT e il grafo in modo più complesso. Le query possono essere utilizzate per recuperare informazioni complesse e per eseguire interrogazioni semantiche sullo stato del DT.

Affinché il modello WoDT sia implementato in modo efficace e operativo, necessita di una definizione astratta del ciclo di vita dei Digital Twin e di un'architettura software adeguata.

Come si può notare in Figura 3.2, illustrata in [RCM+22b], un Digital Twin può transitare in uno dei seguenti stati:

- *Operating & Not Bound*: il DT viene avviato e si sposta nello stato "Operating & Not Bound", dove tutti i moduli interni sono attivi ma non è ancora collegato con l'entità fisica (PA).

- *Bound*: avviene la procedura di associazione tra il DT ed il PA secondo i requisiti specifici del dominio.

- *Shadowed*: il DT è correttamente collegato alla sua controparte fisica e può quindi gestire eventi bidirezionali, interagire con l'asset fisico e avviare il processo di shadowing per sincronizzarsi in termini di eventi e stato.

- *Out of Sync*: qualsiasi errore durante la sincronizzazione porta il DT in uno stato nuovo, denominato "Out of sync", in cui non è in grado di gestire eventi, allineare il suo stato o interagire con il mondo esterno. Solo una volta ripristinata correttamente la sincronizzazione, il DT torna allo stato "Shadowed".
- *Done*: durante il suo ciclo di vita, il DT può essere arrestato e spostato nello stato "Done". In questo stato è ancora attivo e accessibile da applicazioni e utenti esterni (mantenendo la memoria e il registro degli eventi), ma non è più collegato o sincronizzato con il PA.
- *Stop*: alla fine del suo ciclo di vita, il DT può essere definitivamente chiuso e associato allo stato "Stop".

3.3 White Label Digital Twin Framework

In questa sezione verrà introdotto e descritto un framework che supporta ed implementa la visione *Web of Digital Twin*. In particolare, ne verrà descritto il meta-modello in quanto risulta necessario per la comprensione del caso di studio descritto in seguito.

La libreria *WLDT (White Label Digital Twin)* si pone l'obiettivo di supportare la progettazione, lo sviluppo e la distribuzione di Digital Twin all'interno di ecosistemi Internet of Things (IoT). La sua creazione è basata sulle più recenti definizioni di DT provenienti sia dal settore industriale che scientifico, considerando i Digital Twin come componenti software attivi, flessibili e scalabili.

In altre parole, WLDT offre uno strumento completo per:

- Ideare: progettare Digital Twin che rispondano alle esigenze specifiche del proprio caso d'uso.
- Sviluppare: crere facilmente i modelli e le logiche dei gemelli digitali.
- Distribuire: implementare i Digital Twin in ambienti IoT reali.

La libreria è stata sviluppata tenendo conto delle ultime definizioni di Digital Twin presenti in letteratura [RCM+22b], considerandoli non solo come semplici

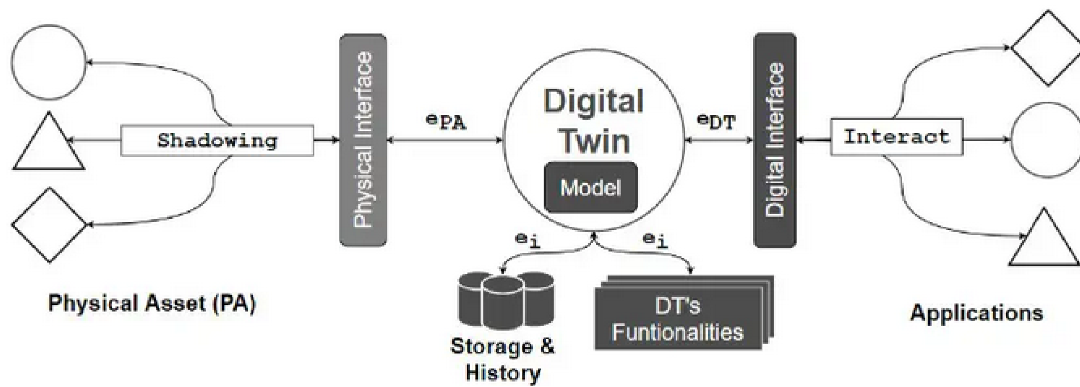

Figura 3.3: Schema meta modello libreria White Label Digital Twins.

rappresentazioni digitali di oggetti fisici, ma come componenti software attivi, flessibili e scalabili. Un'istanza di WLDT può essere paragonata ad un software che implementa tutte le caratteristiche e le funzionalità di un Digital Twin, il quale può essere eseguito in Cloud come in Edge. I Digital Twin generati attraverso la libreria, infatti, sono a tutti gli effetti dei microservizi eseguibili attraverso container Docker.

Questo significa che i Digital Twin sviluppati con WLDT possono:

- Reagire in modo dinamico e adattarsi alle modifiche dell'ambiente e in base alle nuove informazioni raccolte.

- Interagire con altri sistemi inviando comandi ai dispositivi fisici e ricevendo dati in tempo reale.

- Evolvere nel tempo ossia essere facilmente aggiornati e migliorati in base alle necessità.

3.3.1 Meta-modello

Come anticipato, l'intera libreria si basa sulla visione di WoDT presente in [RCM+22b] e descritta nella sezione precedente.

Come si può notare in Figura 3.3, presente nella documentazione della libreria al seguente link, il meta-modello di un Digital Twin è formato dai seguenti componenti:

- *Physical Interface*: è responsabile della digitalizzazione iniziale (shadowing) e del continuo mantenimento della sincronizzazione tra il DT e l'oggetto fisico (PA) durante tutto il suo ciclo di vita. La *Physical Interface* può utilizzare diversi adattatori per interagire con il PA, inoltre si occupa di rilevare e digitalizzare eventi fisici in base alla loro natura e ai protocolli e formati supportati (es. HTTP e JSON).

- *Digital Interface*: è complementare alla *Physical Interface* e si occupa di gestire le variazioni interne del DT e gli eventi verso entità e consumatori esterni. Inoltre, utilizza diversi *Digital Adapter* per gestire interazioni e eventi e garantire l'interoperabilità del DT con applicazioni esterne.

- *Modello del DT*: definisce il comportamento del DT e le sue funzionalità. Inoltre, permette l'esecuzione di vari moduli e funzionalità configurabili e riutilizzabili, gestendo sia eventi fisici che digitali in base al comportamento implementato. Il modello del Digital Twin si occupa di catturare e rappresentare l'asset fisico ad un livello di astrazione appropriato. In parole semplici, questo significa che il modello evita i dettagli non importanti ossia non si sofferma su aspetti dell'oggetto che non sono rilevanti per il suo scopo specifico ed inoltre modella solo le informazioni importanti a livello di dominio, tralasciando quelle più tecniche.

3.3.2 Componenti principali

In Figura 3.4 sono presenti i componenti principali che costituiscono l'architettura della libreria e, dunque, attraverso i quali si implementa il singolo Digital Twin. Nello specifico, nell'immagine si possono notare i tre livelli su cui si sviluppa l'architettura: quello relativo al *core* della libreria, quello che modella il DT e, infine, quello degli *adapter*.

In Figura 3.4, presente nella documentazione della libreria al seguente al seguente link, è possibile individuare i seguenti componenti:

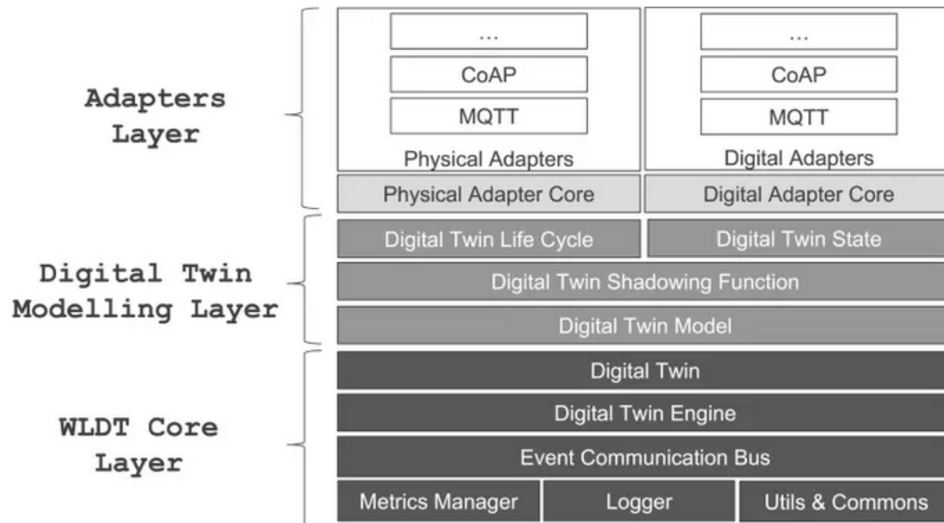

Figura 3.4: Componenti principali libreria White Label Digital Twins.

- *Metrics Manager*: fornisce funzionalità per monitorare e tracciare metriche interne e personalizzate all'interno dei Digital Twin, con un approccio flessibile e adattabile.

- *Logger*: consente di registrare in modo efficiente e personalizzabile gli eventi all'interno dei twin implementati, con livelli di log configurabili e opzioni di output versatili.

- *Utils & Commons*: racchiude una raccolta di classi comuni pronte all'uso in tutte le implementazioni dei Digital Twin, dalla gestione di strutture dati comuni a strumenti utili per la manipolazione di stringhe.

- *Event Communication Bus*: rappresenta il bus interno degli eventi, progettato per supportare la comunicazione tra i diversi componenti dell'istanza del Digital Twin. Permette di definire eventi personalizzati per modellare input e output sia fisici che digitali. Ogni componente di WLDT può pubblicare sul bus di eventi condiviso e definire un filtro per specificare quali tipi di eventi è interessato a gestire, associando una callback per elaborare i diversi eventi.

- *Digital Twin Engine*: definisce il motore multi-thread della libreria, consentendo l'esecuzione e il monitoraggio simultaneo di più gemelli digitali (e dei loro componenti principali). È quindi responsabile anche dell'orchestrazione dei diversi moduli interni dell'architettura, tenendo traccia di ciascuno di essi, e può essere considerato il cuore della piattaforma stessa, consentendo l'esecuzione e il controllo dei gemelli digitali distribuiti. Attualmente supporta l'esecuzione di twin sullo stesso processo Java, tuttavia la stessa astrazione del motore potrebbe essere utilizzata per estendere il framework e supportare l'esecuzione distribuita, ad esempio attraverso diversi processi o microservizi.

- *Digital Twin*: modella una struttura modulare del gemello digitale costruita attraverso la combinazione del Core insieme alle funzionalità di Digital e Physical Adapters. Questo livello include lo stato, definendo quindi l'elenco di proprietà, relazioni, eventi e azioni. Le diverse istanze possono corrispondere direttamente a elementi dell'asset fisico o derivare dalla loro combinazione; in ogni caso, è la funzione di shadowing (SF) che definisce il mapping, seguendo il modello definito in fase di progettazione. Questo componente espone anche un insieme di metodi per consentire la manipolazione della SF. Ogni volta che lo stato del gemello digitale viene modificato, viene generato l'evento corrispondente per notificare tutti i componenti della variazione.

- *Shadowing Function*: è responsabile di definire il comportamento del Digital Twin interagendo con lo stato. In particolare, implementa il processo di shadowing che consente di mantenere sincronizzato il DT rispetto al suo asset fisico. La Shadowing Function costituisce il componente fondamentale che deve essere esteso dal progettista del DT per concretizzarne il modello. La funzione di shadowing osserva il ciclo di vita del Digital Twin al fine di essere notificata dei diversi cambiamenti di stato. Ad esempio, viene informata quando il DT passa allo stato Bound, cioè quando i suoi Physical Adapter hanno terminato la procedura di binding con l'asset fisico. Questo componente, inoltre, consente di definire il comportamento del DT nel caso in cui venga modificata una proprietà, scatenato un evento o invocata un'azione.

- *Physical Adapter*: definisce le funzionalità essenziali che devono essere implementate dalle singole estensioni relative a specifici protocolli. Come previsto dalla definizione del DT, un DT può essere dotato di più Physical Adapter per gestire la comunicazione con la corrispondente entità fisica. Ciascuno di essi produrrà una *Physical Asset Description* (PAD), ovvero una descrizione delle proprietà, eventi, azioni e relazioni che l'asset fisico espone attraverso il protocollo specifico. Il DT passa dallo stato Unbound allo stato Bound quando tutti i suoi Physical Adapters hanno prodotto i rispettivi PAD. La funzione di shadowing, seguendo il modello DT, seleziona i componenti dei vari PAD che è interessata a gestire.

- *Digital Adapter*: fornisce l'insieme delle callback che ciascuna implementazione specifica può utilizzare per essere notificata dei cambiamenti di stato del DT. Simmetricamente a quanto avviene per i Physical Adapter, un Digital Twin può definire molteplici Digital Adapter per esporre il proprio stato e proprietà attraverso diversi protocolli.

3.3.3 Limiti e problematiche

I componenti descritti nella sezione precedente corrispondono a dei moduli interni della libreria. In questo caso, a livello architetturale si può affermare che la libreria è implementata come un monolite. Se da un lato questo consente il rapido avvio di istanze di Digital Twin in quanto sono subito disponibili tutti i componenti necessari, dall'altro provoca una serie di svantaggi, tra i quali:

- mancanza di scalabilità: la rigidità dell'architettura rende difficile l'aggiunta di nuovi moduli o la modifica di quelli esistenti senza dover ridistribuire l'intera libreria.

- dipendenza dall'event bus interno: l'utilizzo di un event bus interno limita la flessibilità e la portabilità della libreria. Inoltre, la comunicazione tra i moduli è vincolata all'implementazione specifica dell'event bus interno.

- obbligo di utilizzare stesso linguaggio di programmazione: i vari componenti appartengono allo stesso software e di conseguenza devono essere scritti nello stesso linguaggio di programmazione.

Attualmente, la comunicazione tra il modulo `Digital Twin Engine` ed i vari physical e digital adapter è basata su eventi e avviene mediante un altro componente che funge da Event Bus interno al processo. Infatti, come anticipato in precedenza, ciascun componente può pubblicare sul bus di eventi condiviso e definire un filtro per specificare quali tipi di eventi è interessato a gestire, associando una callback per elaborare i diversi eventi. Di conseguenza l'implementazione attuale, oltre ad aggiungere una dipendenza tra le interfacce dei diversi componenti, provoca una gestione centralizzata dei vari adapter. Infatti, essi devono essere necessariamente registrati nello stesso momento dell'engine, ossia prima di mandare in esecuzione il software. Questo è sicuramente in contrasto con i principi descritti nella visione "Web of Digital Twin" [RCM+22b] in quanto ciascun componente ha un proprio stato che verrà aggiornato a seconda degli eventi ricevuti. Ad esempio, il Core del Digital Twin potrebbe trovarsi nello stato *Operating & Not Bound*, per un determinato arco di tempo, nel quale aspetta di essere connesso con il rispettivo Physical Asset. Solo in un secondo momento, alla ricezione dell'evento di binding potrà transitare nello stato *Bound*.

Capitolo 4

Progettazione di un'architettura event-driven per il design di Digital Twins

In questo capitolo, partendo dalle limitazioni e problematiche descritte nella sezione precedente, si presenterà un'architettura event-driven ed asincrona realizzata per migliorare il design e la comunicazione tra i componenti della libreria *White Label Digital Twin*. L'obbiettivo del caso di studio è appunto quello di proporre un modello di Digital Twin basato su eventi utilizzando i pattern e le metodologie descritte nei capitoli iniziali della tesi.

4.1 Requisiti dell'architettura

Come anticipato, l'obiettivo del progetto è quello di dimostrare i vantaggi ed i benefici ottenuti nell'utilizzo di un'architettura ad eventi nel design di una libreria per l'esecuzione di ecosistemi di Digital Twin.

L'obiettivo di questa sezione è quello di delineare i requisiti ad alto livello che dovranno essere soddisfatti nella progettazione dell'architettura proposta.

Gestione della comunicazione asincrona tra i componenti

A fronte dell'analisi svolta sui pattern e sulle best practice nel contesto delle moderne architetture ad eventi, è emerso che la comunicazione tra diversi microservizi, in questo caso i componenti della libreria, deve avvenire univocamente tramite consumo e pubblicazione di eventi su determinati stream. Infatti, questo requisito esprime la necessità di utilizzare un approccio event-driven per tutto ciò che riguarda la comunicazione tra i componenti, eliminando in questo modo accoppiamento e dipendenze inutili.

Gestione ciclo di vita

Per quanto riguarda il ciclo di vita dei componenti della libreria risulta necessario implementare la business logic necessaria per transitare da uno stato all'altro a seconda dell'evento ricevuto. In questo modo, i componenti dovranno solo occuparsi di consumare e pubblicare eventi sull'event bus ed ognuno di essi sarà poi responsabile della propria business logic.

Miglioramento Event Bus interno

L'attuale implementazione dell'Event Bus interno prevede l'utilizzo di una *LinkedBlockingQueue* come buffer di eventi. L'utilizzo della coda come Event Bus obbliga lo sviluppatore a dover mettere in campo meccanismi per la gestione di concorrenza e sincronizzazione per evitare corse critiche nell'aggiunta o nel consumo di eventi dalla coda. Di conseguenza questo requisito consiste nel migliorare la soluzione attuale utilizzando i pattern e le best practice per la comunicazione intra-processo ed event-driven.

Aggiunta di un Event Bus remoto

Uno dei requisiti principali dell'architettura è sicuramente quello di disaccoppiare il componente principale, ossia l'*engine* della libreira, dai vari *adapters*. Nella soluzione attuale, infatti, essi appartengono allo stesso software ed allo stesso flusso di esecuzione e questo provoca una serie di problemi tra cui:

- Obbligo di mandare in esecuzione il *WLDT Engine* e gli *adapters* nello stesso momento.
- Obbligo di utilizzare lo stesso linguaggio di programmazione per l'implementazione dei diversi componenti
- Obbligo di utilizzare stesse strutture dati per l'invio e ricezione di eventi

Tutte queste problematiche possono essere risolte tramite l'aggiunta di un Event Bus remoto. In questo modo, il componente "core" potrà andare in esecuzione in qualsiasi momento e dovrà occuparsi solamente di connettersi ad un istanza di un Event Bus remoto e successivamnte attendere la registrazione di uno o più *adapter*. Allo stesso modo, i vari *adapter* potranno andare in esecuzione in un secondo momento e connettersi quindi alla stessa istanza dell'Event Bus in cui è presente il core. In aggiunta, in questo caso il core e gli adapter potranno essere implementati con linguaggi di programmazione differenti in quanto sarà necessario solamente condividere lo schema degli eventi scambiati. In questo contesto, è possibile utilizzare uno dei formati descritti nella sezione 2.2.

Interoperabilità nei confronti del canale di comunicazione utilizzato

Questo requisito è necessario per fornire completa interoperabilità nell'utilizzo di un Event Bus interno o remoto. L'aggiunta della possibilità di utilizzare un Event Bus remoto per la comunicazione tra i componenti non deve infatti sostituire in modo completo la soluzione attuale, che prevede invece l'utilizzo di un Event Bus interno al processo. In particolare, si vuole garantire completa trasparenza e interoperabilità verso l'utilizzo di un determinato canale di comunicazione. A fronte di ciò, vengono previsti due scenari principali di utilizzo dell'architettura:

- Utilizzo di un Event Bus interno al processo: in questo caso i diversi componenti appartengono allo stesso programma e comunicano scambiandosi eventi tramite un Event Bus interno al processo in esecuzione.
- Utilizzo di un Event Bus remoto: in questo caso i componenti comunicano mediante un'istanza remota di un Event Bus. Essi quindi possono

sia appartenere allo stesso programma che andare in esecuzione in modo indipendente.

4.2 Core & Plugin

Al fine di evitare di utilizzare direttamente i moduli della libreria, è stato creato un progetto contenente i seguenti componenti:

- **Core**: rappresenta la parte centrale dell'architettura. Ha il compito di gestire la registrazione di diversi *Plugins* e di interagire con essi in quanto contribuiranno al suo cambiamento di stato. Il Core è un astrazione del modulo *WLDT Core Layer* della libreria WLDT.

- **Plugin**: rappresenta un componente che implementa specifiche funzionalità dell'architettura. Ha il compito di registrarsi al modulo *Core* ed attendere comandi per eseguire il proprio comportamento. In questo caso, i Plugin sono un astrazione dei moduli Physical Adapter o Digital Adapter della libreria WLDT.

- **Event Bus**: consente la comunicazione asincrona tra ciascun componente che pubblica eventi (Publisher) o che consuma eventi da un determinato topic (Subscriber).

L'obbiettivo è quello di realizzare un'architettura asincrona che consenta la comunicazione tra il modulo Core ed una serie di Plugin utilizzando un approccio event-driven. In questo scenario, illustrato in Figura 4.2, Core e Plugin comunicano attraverso un Event Bus e di conseguenza entrambi fungono da Event Bus client integrando sia le funzionalità di Publisher che di Subscriber.

Core

Come anticipato, il Core è il componente principale dell'architettura. Analogamente al core della libreria WLDT, quest'ultimo ha il compito di:

- gestire gli eventi di sottoscrizione provenienti da uno o più plugin

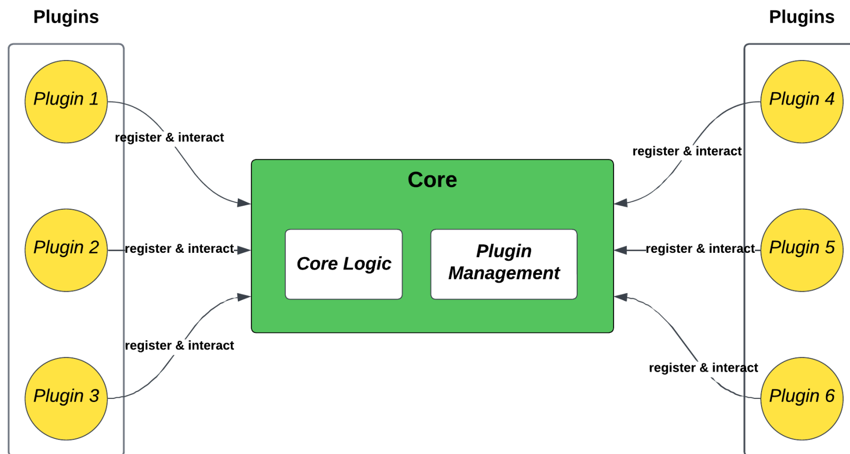

Figura 4.1: Componenti presenti nell'architettura proposta.

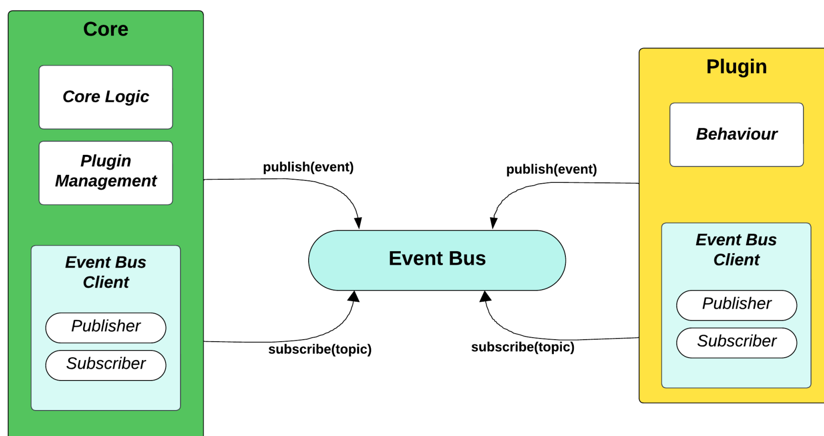

Figura 4.2: Schema della comunicazione tra Core e Plugin mediante l'Event Bus.

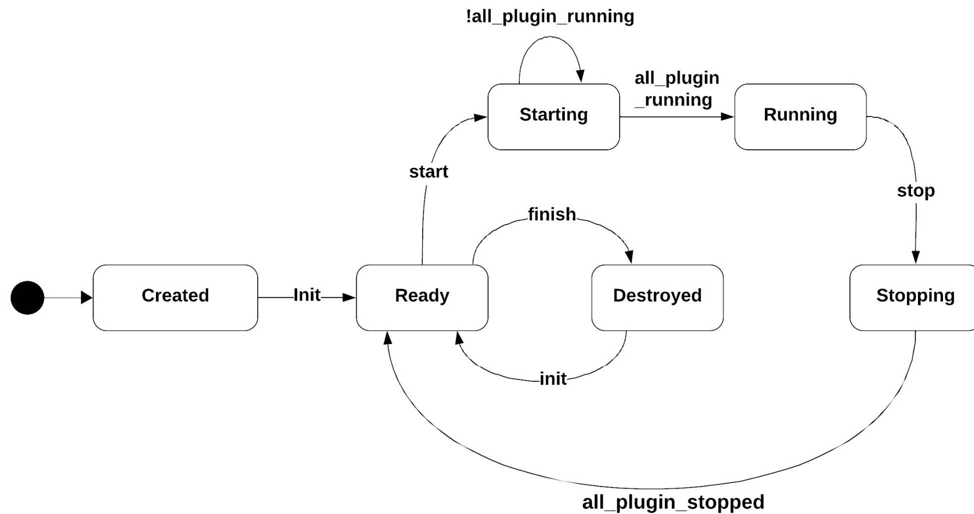

Figura 4.3: Diagramma degli stati del componente Core.

- mandare in esecuzione i plugin una volta avvenuta la registrazione
- gestire il proprio stato interno a seconda della logica dei plugin
- mandare eventi di stop ai plugin
- terminare correttamente la propria esecuzione

Da questo si evince che il Core può essere implementato a tutti gli effetti come un microservizio ad eventi contenente un proprio stato interno che viene aggiornato a seconda degli eventi ricevuti.

In Figura 4.3 è possibile notare il diagramma degli stati del componente Core contenente i seguenti metodi di transizione:

- *Init*: necessario per inizializzare il Core e connettersi all'Event Bus, fa transitare lo stato da *Created* a *Ready*.
- *Start*: manda in esecuzione il Core e, se non si è registrato nessun Plugin allora passerà direttamente allo stato *Running*, altrimenti manderà degli eventi di start a tutti i plugin registrati e passerà allo stato *Starting*.

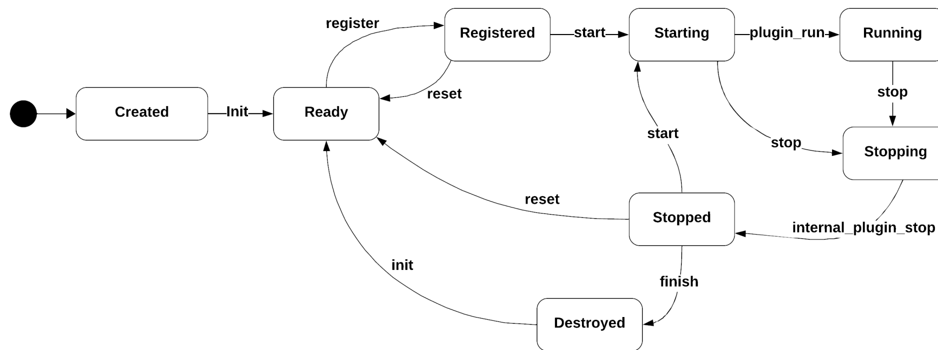

Figura 4.4: Diagramma degli stati del componente Plugin.

- *Stop*: termina il Core facendolo passsare nello stato *Stopping*. In questo stato manderà un evento di stop a tutti i plugin registrati e aspetterà la loro terminazione. Se nessun plugin è presente passerà subito allo stato Ready.

- *Finish*: questo metodo può essere chiamato solo dallo stato Ready ed è necessario per disconnettersi dall'event bus e passare allo stato *Destroyed*.

Plugin

I Plugin sono moduli ad hoc che implementano dei task specifici per l'architettura. Essi si registrano al Core e si mettono in ascolto di eventi per iniziare ad eseguire il proprio task.

In Figura 4.4 è possibile notare il diagramma degli stati del componente Plugin contenente i seguenti metodi di transizione:

- *Init*: inizializza il plugin, si occupa della connesione all'Event Bus e fa transitare lo stato da *Created* a *Ready*. Questo passaggio non viene comunicato al Core in quanto si tratta di uno stato interno e non è rilevante. Al contrario, il Core necessita di sapere quando il Plugin si trova nello stato *Registered*.

- *Register*: si occupa di registrare il plugin al core e mandare un ack di avvenuta registrazione. Quando l'ack viene ricevuto allora lo stato transiterà da *Ready* a *Registered*.
- *Start*: manda in esecuzione il task del plugin e di conseguenza fa transitare lo stato in *Starting*. Se l'avvio va a buon fine allora il plugin passerà nello stato *Running*, altrimenti passerà allo stato *Stopped* restituendo un errore.
- *Stop*: termina l'esecuzione del plugin cambiando lo stato in *Stopped*.
- *Reset*: si occupa di resettare il plugin, cancellare la registrazione dal Core e cambiare stato in *Ready*.
- *Finish*: questo metodo può essere invocato solo se il plugin si trova nello stato *Stopped* e si occupa di terminare l'esecuzione del plugin cancellando la registrazione e disconnettendosi dall'Event Bus.

4.3 Design architetturale

Come si può notare in Figura 4.5, l'architettura in questione ha previsto lo sviluppo delle seguenti classi:

- `CoreInstance`: rappresenta un'istanza di un componente Core.
- `PluginInstance`: rappresenta un'istanza di un componente Plugin.
- `EventBusClient`: interfaccia generica che fornisce un contratto per la connessione, l'invio ed il consumo di messaggi da un Event Bus.
- `ReactiveInternalEventBusClient`: rappresenta un client di un Event Bus interno al processo.
- `RedisRemoteEventBusClient`: rappresenta un client di un Event Bus remoto.

Uno degli aspetti più importanti è sicuramente la presenza dell'interfaccia `EventBusClient`, la quale fornisce appunto un contratto comune per utilizzare

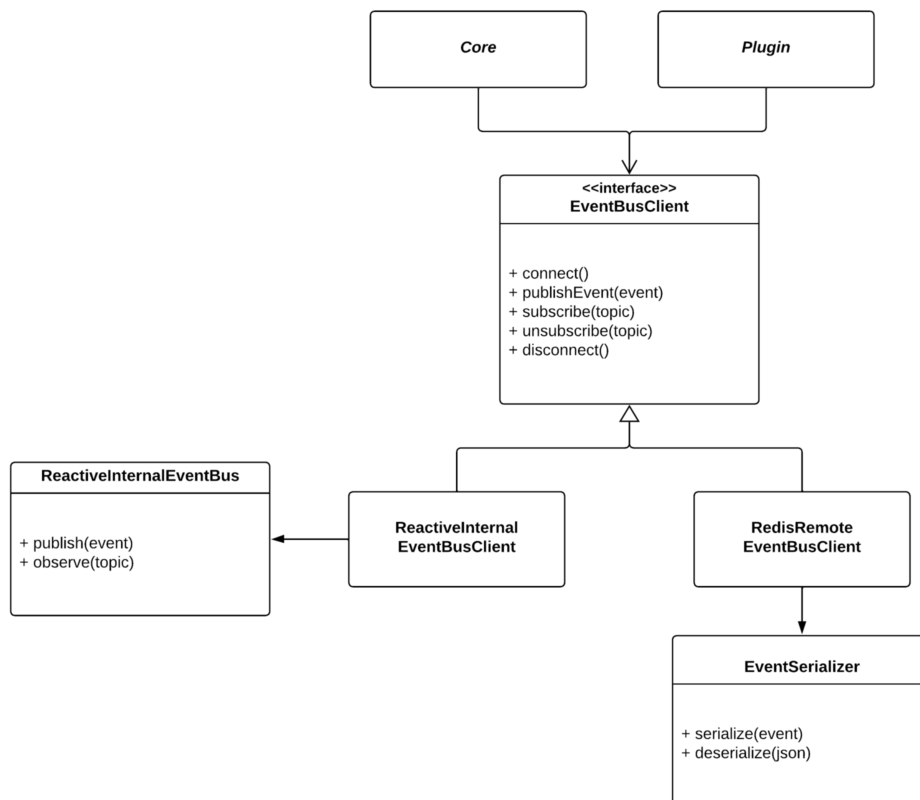

Figura 4.5: Diagramma delle classi dell'architettura realizzata.

un generico Event Bus. In questo modo, i componenti possono utilizzare il client dell'Event Bus in modo completamente trasparente e non sono a conoscenza della presenza di un Event Bus interno o remoto. Questo design consente quindi di soddisfare il requisito relativo all'interoperabilità nei confronti del canale di comunicazione utilizzato. In particolare, i due scenari di utilizzo previsti sono entrambi supportati in quanto con la stessa implementazione dei componenti è sufficiente cambiare il client dell'Event Bus interno con uno remoto.

Inoltre, l'aggiunta di un client remoto consente il disaccoppiamento tra i vari componenti in quanto è possibile mandarli in esecuzione in momenti differenti e su processi diversi. Tornando all'analogia con i Digital Twin, questo diventa un aspetto fondamentale in quanto il Core, che rappresenta quindi il componente principale del Digital Twin, può essere eseguito in qualsiasi momento e non è necessario attendere che i Plugin siano disponibili. Solo in un secondo momento, i Plugin, che possono rappresentare dei *Physical Asset*, si registreranno al Digital Twin ed inizieranno ad inviare eventi di aggiornamento. In questo modo, ciacun componente risulta a tutti gli effetti un microservizio event-driven in quanto ha un proprio stato interno e consuma una serie di eventi da un Event Bus esterno. In questo contesto, vengono previste due differenti tipologie di eventi:

- Eventi di sincronizzazione: questi eventi riguardano ad esempio la registrazione dei Plugin al Core, l'avvio, lo stop o in generale tutti gli eventi legati al ciclo di vita dei vari componenti.

- Domain events: una volta in esecuzione i componenti iniziano a scambiare Domain events, ossia eventi inerenti alla business logic di ciascun componente.

Sebbene nel contesto dei microservizi event-driven si consiglia la presenza di soli *Domain events*, i quali esprimono dei fatti accaduti legati al dominio, in questo caso è necessario prevedere anche eventi di sincronizzazione in quanto si tratta di un aspetto fondamentale in questa tipologia di architettura.

Un'ulteriore caratteristica del design proposto riguarda la possibilità di creare una sorta di composizione tra diversi componenti. Come anticipato nel capitolo introduttivo sui Digital Twin, una tra le principali proprietà elencate in [MLC20]

è appunto la compositività. Infatti, un Digital Twin deve possedere la capacità di monitorare simultaneamente sia il comportamento dell'oggetto composito che quello delle singole componenti. In un sistema di dimensioni considerevoli, ogni componente o sotto-sistema dovrebbe essere rappresentato da un apposito clone digitale. Questo insieme di copie software deve tuttavia essere in grado di collaborare e interagire come se fosse un'unica grande entità per soddisfare le esigenze delle applicazioni. D'altra parte, questa proprietà si riferisce alla possibilità, attraverso il Digital Twin, di astrarsi dalla complessità di un sistema di grandi dimensioni. Ciò consente di concentrarsi sugli aspetti rilevanti senza la necessità di considerare ogni singola componente. In questo design, utilizzando un adeguato livello di astrazione, è possibile soddisfare questa proprietà in quanto il componente Core può a sua volta fungere da Plugin verso un altro componente Core. In questo modo, è possibile creare delle "gerarchie" logiche nelle quali alcuni componenti sono in realtà il frutto di una composizione di altri componenti.

4.4 Esempio d'uso

Al fine di validare il design realizzato e testarne il funzionamento in uno scenario reale è stato creato un progetto di esempio in cui si fa uso dell'architettura in questione. Il progetto si colloca nel contesto dell' *Internet of Things* ed in particolare viene trattata una *Smart Home*. La *Smart Home* è dotata di una serie di sensori di temperatura presenti nelle varie stanze ed inoltre viene prevista la presenza di un *SmartTemperatureDT*, il quale ha il compito di ricevere i dati dai sensori, aggregarli e calcolare ad esempio la temperatura media della casa.

La comunicazione tra i sensori di temperatura ed il *SmartTemperatureDT* è completamente asincrona in quanto ciascuno di essi potrà rilevare il proprio valore in qualsiasi momento e successivamente spedire un evento ad esso contenente il dato.

Per quanto riguarda i sensori di temperatura, essi utilizzano il protocollo *CoAP* per inviare periodicamente la temperatura rilevata ad un gateway (CoAP Server). CoAP (Constrained Application Protocol) è un protocollo di rete progettato per dispositivi con risorse limitate, come quelli utilizzati appunto nell'Internet of Things (IoT).

In questo caso, tornando all'analogia con l'architettura realizzata, risulta fin da subito evidente la presenza di due componenti distinti:

- `SmartTemperatureDT`, componente principale che deve raccogliere i dati dai sensori ed elaborarli.
- i sensori di temperatura, i quali devono misurare la temperatura ed inviare i dati.

Di conseguenza, in questo esempio il *SmartTemperatureDT* rappresenta il componente *Core*, mentre i vari sensori sono a tutti gli effetti dei *Plugin* i quali si registrano al Core ed iniziano a mandare eventi contenenti i valori misurati.

In Figura 4.6 è possibile notare l'interazione tra i vari componenti del sistema. In particolare, si può notare come i vari sensori di temperatura interrogano il server CoAP ed estendono il componente *Plugin* il quale consente di registrarsi al *Core* e di iniziare a produrre eventi tramite l'Event Bus di Redis. In questo contesto, l'architettura realizzata ha permesso lo sviluppo di entrambi i componenti in modo agile ed ha fornito tutto il necessario per la comunicazione asincrona tra di essi. Inoltre, essa ha consentito di ottenere i seguenti vantaggi:

- Il *SmartTemperatureDT* può andare in esecuzione in modo indipendente in qualsiasi momento. Infatti, è sufficiente connettersi all'Event Bus ed attendere la registrazione dei sensori di temperatura.
- La natura asincrona ed event-driven della comunicazione tra i sensori ed il core consente di non dover attendere tutti i valori prima di comunicarli e di evitare in questo modo di creare delle dipendenze tra i vari sensori.
- L'utilizzo di un Plugin per ciascun sensore di temperatura consente di essere indipendenti verso la tecnologia specifica del sensore. Infatti alcuni sensori potrebbero utilizzare un altro protocollo, come ad esempio *MQTT*, e questo non inciderebbe in alcun modo sul comportamento del sistema.
- La presenza di un'istanza remota dell'Event Bus apre la possibilità di utilizzare il componente Core in Cloud.

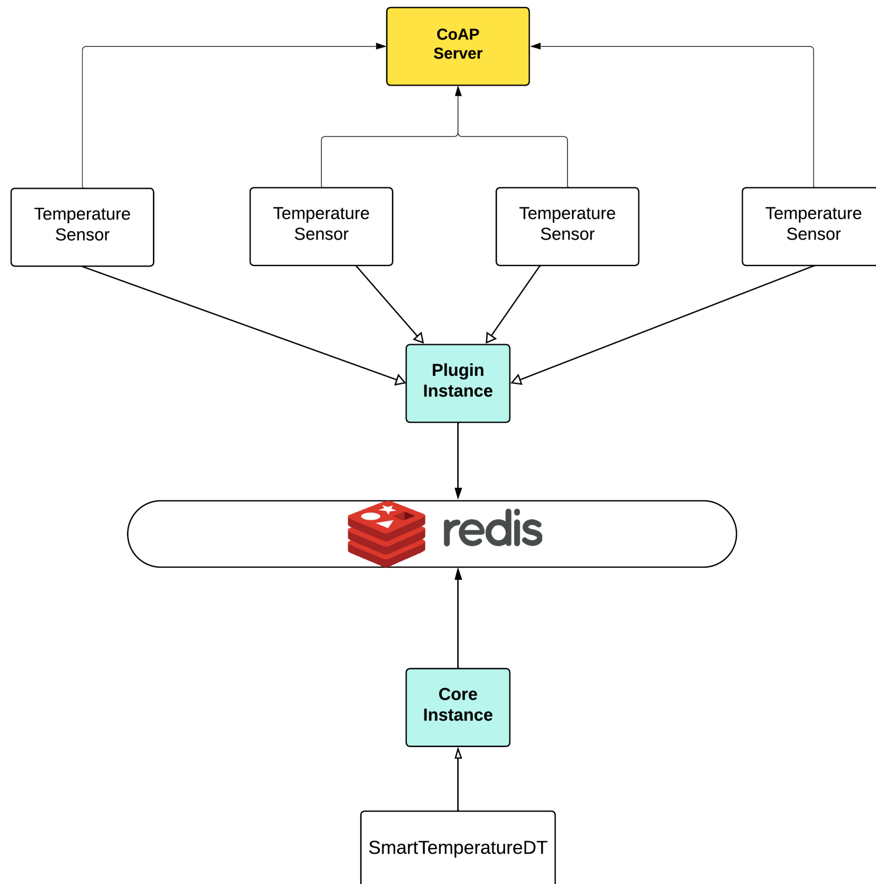

Figura 4.6: Schema dei componenti nel progetto di esempio Smart Home.

Il progetto realizzato potrebbe essere ulteriormente esteso con l'ausilio di questa architettura introducendo ad esempio un Digital Twin della casa. Infatti, si potrebbe prevedere la presenza dei seguenti componenti:

- `SmartTemperatureDT`: raccoglie ed elabora dati dai sensori di temperatura presenti nella casa.
- `SmartLuminosityDT`: raccoglie ed elabora dati dai sensori di luminosità presenti nella casa.
- `SmartHeatingDT`: gestisce gli attuatori relativi al riscaldamento nella casa.

Tutti questi componenti, che inizialmente corrispondono a dei *Core* nei confronti dei rispettivi sensori, possono rappresentare in questo caso dei *Physical Asset* di un Digital Twin della casa creando così una sorta di composizione tra Digital Twin.

Come si può notare in Figura 4.7 il Digital Twin della casa, che corrisponde quindi al componente Core, avrà come proprietà temperatura, luminosità e riscaldamento e sarà alimentato dagli ulteriori Digital Twin presenti.

In conclusione, l'architettura realizzata consente il rapido sviluppo di sistemi che necessitano di comunicare in modo asincrono ed event-driven. Essa fornisce infatti un livello di astrazione tale da poter essere utilizzata sia per il design di Digital Twin che per altri software che condividono gli stessi requisiti.

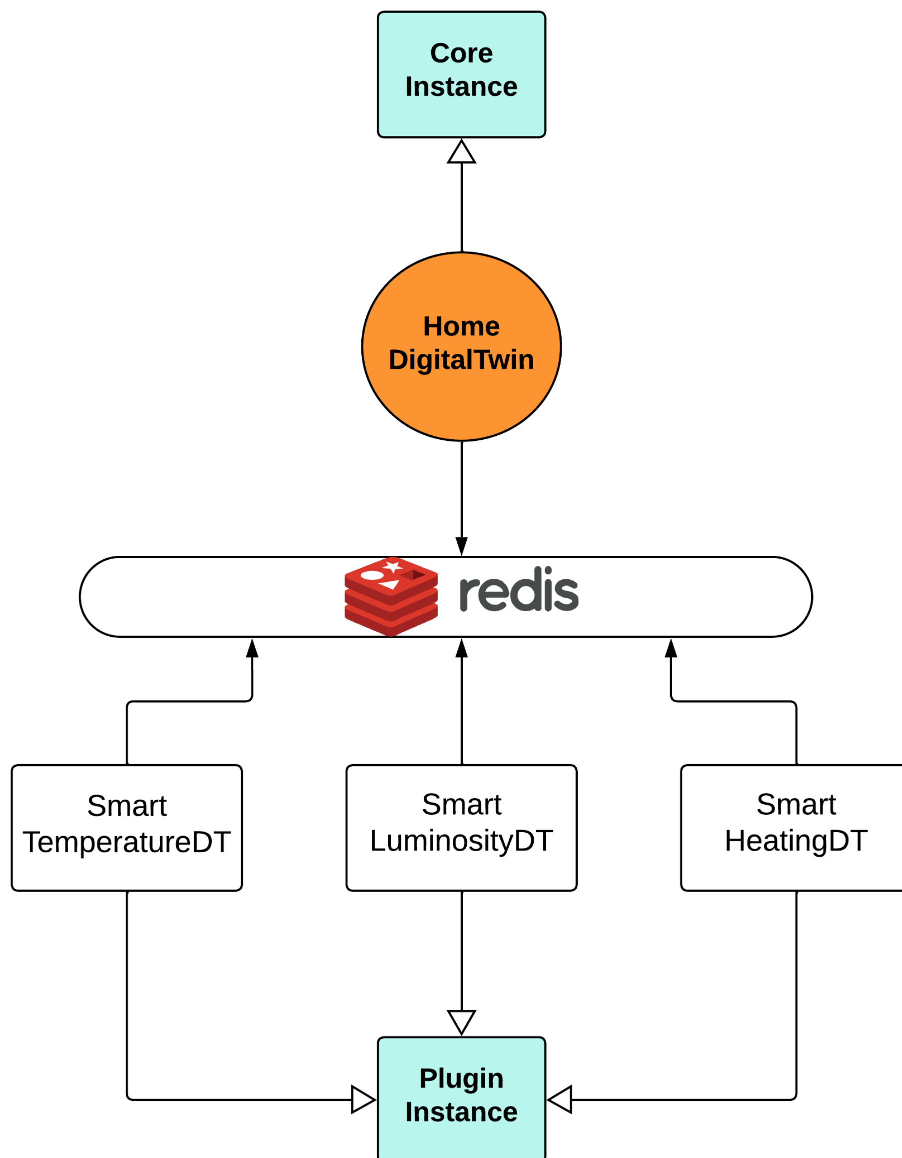

Figura 4.7: Esempio di un Digital Twin di una Smart Home.

Capitolo 5

Implementazione

In questo capitolo, viene descritta la fase di implementazione del progetto in questione. In particolare, inizialmente vengono forniti alcuni dettagli tecnici riguardanti l'intero processo di sviluppo. Successivamente, verranno spiegate le tecnologie e metodologie utilizzate per l'implementazione dell'Event Bus interno e remoto utilizzato per la comunicazione asincrona ed event-driven tra i componenti. Infine, verrà descritta la fase di testing svolta, la quale si è resa necessaria per assicurare sia il corretto funzionamento dei componenti che dell'event bus realizzato.

5.1 Processo di sviluppo

Al fine di mantenere allineato il progetto realizzato con la libreria *White Label Digital Twin Framework*, la quale è stata oggetto del caso di studio, si è scelto di utilizzare come linguaggio di programmazione *Java*. In particolare, è stato creato un progetto *Gradle*, nel quale sono presenti due ulteriori sotto-progetti:

- Il primo comprende l'effettiva implementazione dell'architettura descritta.
- Il secondo progetto consiste in un esempio di utilizzo dell'architettura, realizzato per validare i risultati ottenuti.

Inoltre, è stato creato un workflow di *Continuous Integration* il quale si occupa di mandare in esecuzione tutti i test ad ogni nuova modifica nel codice sorgente

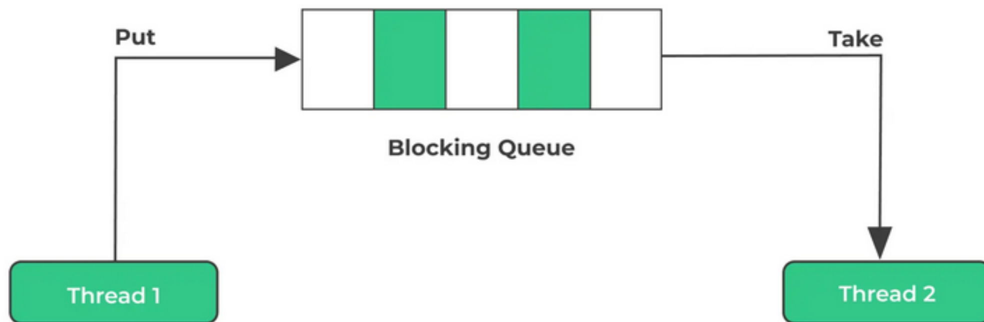

Figura 5.1: Implementazione Event Bus interno nella libreria WLDT.

così da evitare errori di regressione. Il workflow è stato realizzato tramite l'utilizzo delle *GitHub Actions*.

5.2 Implementazione Event Bus interno

Per quanto riguarda l'Event Bus interno al processo Java, prima di procedere con l'implementazione è stata effettuata un'analisi della soluzione attuale presente nella libreria. In particolare, l'attuale implementazione prevede l'utilizzo di una `LinkedBlockingQueue` condivisa tra i vari componenti.

Come si può notare in Figura 5.1, l'utilizzo della coda come Event Bus obbliga lo sviluppatore a dover mettere in campo meccanismi per la gestione di concorrenza e sincronizzazione per evitare corse critiche nell'aggiunta o nel consumo di eventi dalla coda. Dopo aver cercato alcune librerie Java che consentissero di utilizzare un Event Bus per lo scambio di eventi si è scelto di implementarne una versione ad hoc utilizzando il paradigma della programmazione reattiva. La programmazione reattiva è un paradigma di programmazione costruito sulla nozione di propagazione del cambiamento. Essa facilita lo sviluppo di applicazioni event-driven permettendo allo sviluppatore di esprimere programmi in termini di che cosa devono fare e consentendo così al linguaggio di gestire automaticamente quando farlo. In questo paradigma, i cambiamenti di stato sono efficientemente

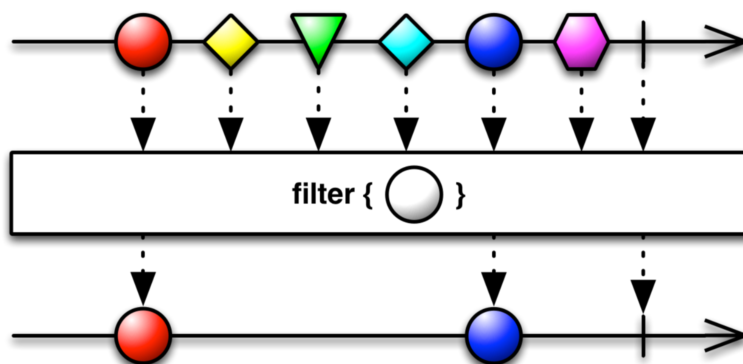

Figura 5.2: Esempio Reactive Stream con operatore filter.

e automaticamente propagati attraverso la rete delle dipendenze computazionali dal sottostante modello di esecuzione. Riassumendo, la programmazione reattiva permette di gestire facilmente stream asincroni di dati ed eventi. Questi eventi vengono catturati in modo asincrono definendo delle funzioni che verranno eseguite rispettivamente quando un nuovo valore viene emesso, quando avviene un errore o quando lo stream termina. In questo caso, viene definito *Observer* colui che rimane in ascolto di nuovi valori ed *Observable* lo stream di dati o eventi.

Per quanto riguarda il progetto in questione, è stata utilizzata la libreria *RxJava*, la quale consiste in un'implementazione del framework *ReactiveX* nel linguaggio Java. In particolare l'Event Bus è stato implementato mediante un `PublishSubject`, ossia una struttura dati reattiva che consente ai nuovi Observer di ricevere tutti gli elementi successivi alla loro sottoscrizione. Un PublishSubject si comporta in modo simile a un Observable, in quanto emette una sequenza di elementi agli Observer, la differenza è che esso può essere utilizzato appunto come "buffer" di elementi.

Come si può notare nel codice riportato nel listato 5.1, la pubblicazione di un evento consiste semplicemente nel invocare il metodo `onNext()` sulla struttura dati mentre la sottoscrizione ad un topic restituisce direttamente un *Observable* nel quale viene applicato l'operatore *filter*, come si può notare in Figura 5.2 la quale è presente nella documentazione della libreria al seguente link.

La soluzione proposta, consente quindi un approccio moderno e reattivo per la comunicazione tra diversi componenti appartenenti allo stesso software. Inoltre, in

Listing 5.1: Implementazione reattiva dell'Event Bus interno.

```
public class ReactiveInternalEventBus {

    private final PublishSubject<Event> eventStream;
    private static ReactiveInternalEventBus instance = null;

    public ReactiveInternalEventBus() {
        logger.info("ReactiveInternalEventBus Created !");
        this.eventStream = PublishSubject.create();
    }

    public synchronized static ReactiveInternalEventBus getInstance() {
        if(instance == null)
            instance = new ReactiveInternalEventBus();
        return instance;
    }

    public void publishEvent(final Event event) throws EventBusException {
        this.eventStream.onNext(event);
    }

    public Observable<Event> observe(final String topic) throws EventBusException
        {
        return this.eventStream.filter(event -> Objects.equals(event.getTopic(),
            topic));
    }

    public Observable<Event> observeType(final String topic, final String type) {
       return this.observe(topic).filter(event -> event.getType().equals(type));
    }

    public Observable<Event> observeTypes(final String topic, final List<String>
        types) {
        return this.observe(topic).filter(event -> types.contains(event.getType())
            );
    }
}
```

questo caso non è necessario mandare in esecuzione thread ad-hoc per il consumo e l'invio di eventi ma viene incapsulato all'interno delle strutture dati reattive.

5.3 Implementazione Event Bus remoto

Uno dei principali requisiti dell'architettura in questione riguarda sicuramente l'aggiunta di un Event Bus remoto per la comunicazione asincrona tra i componenti Core e Plugin. A fronte di ciò, è stata effettuata un'indagine con lo scopo di identificare la tecnologia migliore per il progetto in questione. Nello specifico, si è scelto di utilizzare *Redis*, il quale nasce come un database in-memory NoSQL ed open source ed è noto per la sua velocità e flessibilità. Infatti, a partire da Redis 5 sono stati introdotti i *Redis Streams*, un modello di dati simili alle code di messaggi ma che offrono funzionalità avanzate come la riproduzione, la compattazione e la ricerca. I Redis Streams permettono di gestire flussi di eventi o messaggi temporali e sono quindi particolarmente utili per la costruzione di sistemi di messaggistica in tempo reale, registri di eventi e gestione delle code. I flussi sono quindi costituiti da una sequenza ordinata di eventi identificati da una chiave. Inoltre, un altro meccanismo fornito da Redis per lo scambio di messaggi è *Redis Pub/Sub*. Quest'ultimo è stato realizzato per consentire la comunicazione asincrona tramite il pattern *Publish/Subscribe* tra diversi client Redis. Permette quindi sia delle connessioni *end-to-end* che *molti-a-molti*, nelle quali uno o più produtti pubblicano messaggi su dei canali ed uno o più consumatori ricevono i messaggi.

In Figura 5.3, presente nella documentazione di Redis al seguente link, si può notare il funzionamento di questa tecnologia.

La scelta dell'utilizzo di Redis è stata motivato dal fatto che il contesto di utilizzo di questa architettura riguarda la presenza di Digital Twin ed ecosistemi di Digital Twin. A fronte di ciò, l'alta velocità ed una bassa latenza sono requisiti fondamentali per questa architettura e Redis, grazie alla sua implementazione in-memory ed alle garanzie di real-timeness, risulta la tecnologia più adatta per questo caso d'uso.

Nello specifico, per l'implementazione del client dell'Event Bus di Redis, è stata utilizzata *Lettuce*, una libreria client Java che fornisce un'implementazione reattiva basata sui *Reactive Streams*. Questa libreria è progettata per integrarsi

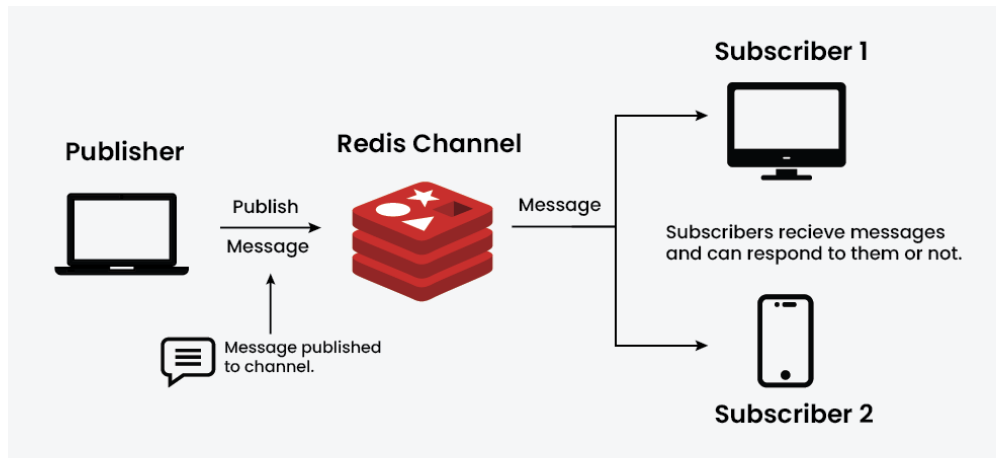

Figura 5.3: Schema di funzionamento della tecnologia Redis Pub/Sub.

facilmente con applicazioni Java che richiedono una gestione asincrona ed efficiente delle operazioni Redis. Lettuce, essendo una libreria reattiva basata su Reactive Streams, offre un'API reattiva per interagire con Redis in modo asincrono. In particolare utilizza il *Project Reactor*, un framework reattivo per Java, per fornire operazioni asincrone e reattive.

Nel listato in sezione 5.2 è possibile notare i metodi necessari per pubblicare e consumare messaggi da un'istanza Redis.

Serializzazione

Un aspetto critico nella progettazione di un sistema distribuito con un approccio event-driven è sicuramente la serializzazione/deserializzazione degli eventi. Come anticipato nei capitoli precedenti, in primo luogo è necessario che i produttori e i consumatori siano entrambi a conoscenza del modello dei dati degli eventi. Inoltre, è necessario convertire gli eventi in un formato che possa essere trasmesso ed interpretato dai vari componenti del sistema. La scelta del formato di serializzazione dovrebbe essere basata su requisiti specifici del sistema. In questo progetto , infatti, si è scelto di serializzare gli eventi in formato *Json* per la sua semplicità e facilità di parsing. Nello specifico, è stata utilizzata la libreria *Jackson* per la conversione da oggetti Java a Json e viceversa.

Listing 5.2: Implementazione del client di Redis tramite la liberia Lettuce.

```
public class RedisRemoteEventBusClient implements IEventBusClient {

    @Override
    public void publishEvent(Event event) throws EventBusClientException {
            this.producerCommands.publish(event.getTopic(), jsonEvent)
                    .doOnError(error -> System.out.println("[Redis Client ] Error
                        sending event!"))
                    .blockOptional();

    }

    @Override
    public void subscribe(String topic) throws EventBusClientException {
        if (!this.subscriptions.containsKey(topic)) {
            this.consumerCommands.subscribe(topic).subscribe();
            Disposable subscription = this.consumerCommands.observeChannels(
                FluxSink.OverflowStrategy.BUFFER)
                    .map(channelMessage -> eventSerializer.deserialize(
                        channelMessage.getMessage()))
                    .filter(event -> event.getTopic().equals(topic))
                    .subscribe(event -> this.eventListener.onEvent(event));
            this.subscriptions.put(topic, subscription);
            this.eventListener.onEventSubscribed(topic, Event.
                ALL_EVENT_TYPES_WILD_CARD);
        }
    }

    @Override
    public void subscribe(String topic, String eventType) throws
        EventBusClientException {
        if (!this.subscriptions.containsKey(topic)) {
            this.consumerCommands.subscribe(topic).subscribe();
            Disposable subscription = this.consumerCommands.observeChannels(
                FluxSink.OverflowStrategy.BUFFER)
                    .map(channelMessage -> eventSerializer.deserialize(
                        channelMessage.getMessage()))
                    .filter(event -> event.getTopic().equals(topic))
                    .filter(event -> event.getType().equals(eventType))
                    .subscribe(event -> this.eventListener.onEvent(event));
            this.subscriptions.put(topic, subscription);
            this.eventListener.onEventSubscribed(topic, eventType);
        }
    }

    @Override
    public void unSubscribe(String topic) {
        if (this.subscriptions.containsKey(topic)) {
            this.subscriptions.remove(topic).dispose();
        }
    }

}
```

5.4 Testing

L'intero processo di sviluppo ha previsto la creazione di Unit Test per assicurare il corretto funzionamento di tutti i componenti del sistema. In particolare, sono stati svolti i seguenti test:

- Unit test sul ciclo di vita del componente Core
- Unit test sul ciclo di vita del componente Plugin
- Unit test sull'implementazione dell'Event Bus interno reattivo
- Unit test sull'implementazione del client dell'Event Bus interno reattivo
- Unit test sull'implementazione del client dell'Event Bus di Redis remoto

Inoltre è stata testata anche la comunicazione asincrona tra il componente Core ed uno o più Plugin. In particolare è stata creata una classe astratta contenente alcuni scenari d'uso come la registrazione dei Plugin al Core, l'effettiva esecuzione e lo stop di alcuni Plugin. La classe astratta delega la creazione dei componenti alla superclasse, in modo da rendere la logica della comunicazione indipendente e trasparente dalla tipologia di Event Bus utilizzato. In questo caso, sono state create due classi:

1. La prima manda in esecuzione i componenti che utilizzano il client dell'Event Bus interno.
2. La seconda manda in esecuzione i componenti che utilizzano il client di un Event Bus di Redis remoto.

Per quanto riguarda la creazione di tutti gli Unit Test è stata utilizzata la libreria *Junit 5*, mentre per testare la comunicazione asincrona dei componenti mediante l'Event Bus remoto di Redis è stata utilizzata la libreria *TestContainer* la quale consente di mandare in esecuzione un container Docker contenente l'istanza di Redis direttamente all'interno del processo Java.

Di seguito vengono riportati alcuni esempi di test effettuati.

Listing 5.3: Esempio di Unit Tests per la comunicazione tra Core e Plugin.

```
public abstract class AbstractAsyncCommunicationTests {

    @Test
    public void testPluginRegistration() throws InterruptedException {
        pluginInstance1.register();
        assertEquals(PluginState.REGISTERED, pluginInstance1.getPluginState());
    }

    @Test
    public void testCorePluginRegistrationList() throws InterruptedException {
        pluginInstance1.register();
        assertEquals(1, coreInstance.getPluginRegistrationList().size());
    }

    @Test
    public void testAddingAnotherPlugin() throws InterruptedException {
        pluginInstance1.register();
        pluginInstance2.register();
        assertEquals(2, coreInstance.getPluginRegistrationList().size());
    }

    @Test
    public void testStopAndDestroyPlugin() throws InterruptedException {
        pluginInstance1.register();
        coreInstance.stopPlugin(pluginInstance1.getPluginId());
        assertEquals(PluginState.STOPPED, pluginInstance1.getPluginState());
    }

    @Test
    public void testCoreStop() throws InterruptedException {
        pluginInstance1.register();
        coreInstance.stop();
        assertEquals(CoreState.STOPPING, coreInstance.getCoreState());
    }
}
```

Capitolo 6

Conclusioni

L'architettura sviluppata in questo progetto prende spunto dalla combinazione di diversi approcci e tecnologie, ossia i microservizi ad eventi ed i Digital Twins. L'obiettivo della tesi era appunto quello di dimostrare come l'utilizzo di moderni pattern architetturali, come i microservizi guidati da eventi, possa agevolare il design e lo sviluppo di Digital Twin.

Il paradigma dei Digital Twin è molto ampio e comprende numerose proposte presenti in letteratura. La visione adottata in questa tesi è nell'ottica del *Web of Digital Twins* [RCM+22b] la quale ha l'obiettivo di creare un sistema distribuito, event-driven, decentralizzato ed interoperabile che consente il collegamento e l'utilizzo di Digital Twins appartenenti ad organizzazioni diverse.

Nel contesto di questa tesi, il caso di studio svolto ha inizio con lo studio di una piattaforma per l'esecuzione di ecosistemi di Digital Twin, ossia *White Label Digital Twin Framework (WLDT)*. Nello specifico, ne viene analizzato il meta-modello, i componenti principali ed i limiti e le problematiche presenti. Attualmente, la comunicazione tra i diversi moduli della libreria è basata su eventi e avviene mediante un altro componente che funge da event bus interno al processo. Questo approccio comporta la presenza di alcune dipendenze tra le interfacce ed una gestione centralizzata degli *adapter*. Infatti, essi devono essere necessariamente registrati nello stesso momento dell'engine, ossia prima di mandare in esecuzione il software. Inoltre, l'utilizzo di un event bus interno limita la flessibilità e la portabilità della libreria e la comunicazione tra i moduli è vincolata all'implementazione specifica

di esso. A fronte di ciò, si è scelto di migrare verso un approccio più modulare e distribuito, utilizzando le nozioni ed i pattern descritti nei capitoli iniziali. In particolare, la libreria *WLDT* è stata estesa realizzando un'architettura distribuita ed event-driven con l'obiettivo di migliorare la comunicazione tra i diversi componenti della libreria ed utilizzando un event bus remoto. Inoltre, al fine di validare i risultati ottenuti, è stato creato un progetto di esempio in cui l'architettura creata viene utilizzata per la creazione di una *Smart Home* nella quale dei sensori di temperatura rappresentano dei Physical Asset ed inviano dei dati ad un Digital Twin. Tramite la piattaforma sviluppata i sensori ed il Digital Twin comunicano in modo asincrono scambiandosi eventi mediante un event bus remoto. Questo consente di mandare in esecuzione i software in modo autonomo ed indipendente come dei veri e propri microservizi event-driven.

Concludendo, alla luce dei risultati ottenuti, è possibile affermare che la combinazione tra microservizi, eventi e Digital Twin è sicuramente una combinazione vincente ed efficace, in quanto l'adozione di queste tipologie di architetture basate su eventi si è rilevata una strategia ottima per la realizzazione del caso di studio svolto.

Bibliografia

[AFA+20] Omer Aziz, Shoaib Farooq, Adnan Abid, Rubab Saher, and Naeem Aslam. Research trends in enterprise service bus (esb) applications: A systematic mapping study. *IEEE Access*, 8:1–1, 02 2020.

[BCF19] Barbara Rita Barricelli, Elena Casiraghi, and Daniela Fogli. A survey on digital twin: Definitions, characteristics, applications, and design implications. *IEEE Access*, 7:167653–167671, 2019.

[Bel20] A. Bellemare. *Building Event-driven Microservices: Leveraging Organizational Data at Scale*. O'Reilly Media, 2020.

[CGMR20] Angelo Croatti, Matteo Gabellini, Sara Montagna, and Alessandro Ricci. On the integration of agents and digital twins in healthcare. *Journal of Medical Systems*, 44:161, 08 2020.

[ES09] John Erickson and Keng Siau. Service oriented architecture: A research review from the software and applications perspective. *Innovations in Information Systems Modeling: Methods and Best Practices*, pages 190–203, 01 2009.

[Gel93] D. Gelernter. *Mirror Worlds: or the Day Software Puts the Universe in a Shoebox...How It Will Happen and What It Will Mean*. Oxford University Press, 1993.

[Gri15] Michael Grieves. Digital twin: Manufacturing excellence through virtual factory replication. 03 2015.

[GV17] Michael Grieves and John Vickers. *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems*, pages 85–113. 08 2017.

[GZ20] Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. pages 150–153, 04 2020.

[LF23] Luan Lazzari and Kleinner Farias. Uncovering the hidden potential of event-driven architecture: A research agenda. 08 2023.

[MC21] Roberto Minerva and Noël Crespi. Digital twins: Properties, software frameworks, and application scenarios. *IT Professional*, 23(1):51–55, 2021.

[MLC20] Roberto Minerva, Gyu Myoung Lee, and Noël Crespi. Digital twin in the iot context: A survey on technical features, scenarios, and architectural models. *Proceedings of the IEEE*, 108(10):1785–1824, 2020.

[MP23] Roman Malyi and Serdyuk Pavlo. Identifying suitable applications for event sourcing: Criteria, assessment, and implementation guidelines. 06 2023.

[MT17] Neda Mohammadi and John Taylor. Smart city digital twins. pages 1–5, 11 2017.

[PH07] Mike Papazoglou and Willem-Jan Heuvel. Service oriented architectures: Approaches, technologies and research issues. *The VLDB Journal*, 16:389–415, 07 2007.

[RCM22a] A. Ricci, A. Croatti, and S. Montagna. Pervasive and connected digital twins—a vision for digital health. *IEEE Internet Computing*, 26(05):26–32, sep 2022.

[RCM+22b] Alessandro Ricci, Angelo Croatti, Stefano Mariani, Sara Montagna, and Marco Picone. Web of digital twins. *ACM Trans. Internet Technol.*, 22(4), nov 2022.

[Roc22] Hugo Rocha. *Practical Event-Driven Microservices Architecture: Building Sustainable and Highly Scalable Event-Driven Microservices*. 01 2022.

[SA22] Nada Salaheddin and Nuredin Ahmed. Microservices vs. monolithic architectures [the differential structure between two architectures]. *MINAR International Journal of Applied Sciences and Technology*, 4:484–490, 10 2022.

[SWBB20] Steven M. Schwartz, Kevin Wildenhaus, Amy Bucher, and Brigid Byrd. Digital twins and the emerging science of self: Implications for digital health experience design and "small" data. *Frontiers in Computer Science*, 2, 2020.

[TG19] Nupura Torvekar and Pravin Game. Microservices and its applications an overview. *International Journal of Computer Sciences and Engineering*, 7:803–809, 04 2019.

[TZLN19] Fei Tao, He Zhang, Ang Liu, and A. Y. C. Nee. Digital twin in industry: State-of-the-art. *IEEE Transactions on Industrial Informatics*, 15(4):2405–2415, 2019.

[VID+21] Isabel Voigt, Hernan Inojosa, Anja Dillenseger, Rocco Haase, Katja Akgün, and Tjalf Ziemssen. Digital twins for multiple sclerosis. *Frontiers in Immunology*, 12, 2021.

Ringraziamenti

Un sincero ringraziamento va a tutti coloro che mi hanno aiutato in vario modo a raggiungere questo traguardo.
Ringrazio tutti i professori del corso di studi in Ingegneria e Scienze Informatiche, che mi hanno trasmesso grande passione per le loro materie.
In particolar modo, ringrazio il professor Alessandro Ricci, il professor Marco Picone ed il Dottore Samuele Burattini per l'opportunità che mi hanno concesso e per le numerose nozioni apprese durante il progetto.
Ringrazio i miei genitori ed i miei amici, che mi hanno supportato incondizionatamente durante il percorso.
Ringrazio infine i miei compagni di studio, in particolare Giacomo ed Andrea, con i quali ho condiviso cinque bellissimi anni e che sono stati una spalla su cui contare dall'inizio alla fine della mia carriera universitaria.