UNIVERSITÀ TELEMATICA INTERNAZIONALE
UNINETTUNO

FACOLTÀ DI INGEGNERIA

Corso di Laurea Triennale in
Ingegneria Informatica

ELABORATO FINALE

in

Ingegneria del Software e Programmazione ad Oggetti

Progettazione e Sviluppo di un Software Predittivo per monitorare l'Insorgenza del Diabete: un approccio con Modello ad Attori e ML

RELATORE
Prof.ssa Patrizia Grifoni

CANDIDATO
Donadello Andrea

CORRELATORE
Prof. Mazzei Mauro

Anno Accademico 2023/24

A Elena ed alla mia famiglia

Sommario

Negli ultimi dieci anni, il panorama dell'informatica è stato notevolmente influenzato dalle discipline conosciute come *Big Data* e *Machine Learning*. L'ampia digitalizzazione dei servizi, la diffusa adozione di reti sociali, gli acquisti online, l'uso diffuso di dispositivi indossabili avanzati, e la crescente automazione nei settori industriali, come la produzione e la logistica, rappresentano solo alcune delle forze trainanti di un'enorme crescita dei dati a nostra disposizione. Queste enormi quantità di dati hanno messo in evidenza i limiti delle tradizionali metodologie di gestione dei dati, spingendo verso l'adozione di nuovi approcci più efficienti, che rientrano nell'ambito dei *Big Data*.
Questi nuovi approcci non solo sono in grado di gestire grandi quantità di informazioni, ma forniscono anche strumenti avanzati per scoprire modelli e relazioni nascoste, aprendo nuove prospettive di esplorazione dei dati. Inoltre, tali approcci si basano su architetture altamente scalabili che possono adattarsi alle esigenze specifiche e al volume di lavoro trattato.
Parallelamente, il settore dell'intelligenza artificiale noto come *Machine Learning* ha visto una notevole espansione al di fuori dei tradizionali contesti di ricerca. L'evoluzione dei veicoli autonomi e dei robot, la medicina personalizzata e la diagnosi predittiva, oltre all'automatizzazione avanzata dei processi decisionali aziendali, sono solo alcune delle applicazioni in cui il *Machine Learning* ha dimostrato il suo valore. L'introduzione del *Cloud Computing* ha ulteriormente rivoluzionato questo settore, consentendo l'utilizzo di risorse di calcolo virtualmente illimitate e delegando la complessità della gestione hardware e software ai fornitori di servizi cloud.
Questo progresso e l'interesse crescente per queste discipline hanno dato origine a una nuova figura professionale denominata *Data Scientist*, che rappresenta una combinazione di competenze in informatica, statistica, matematica e domini specifici dell'industria. Questi esperti sono essenziali per sfruttare appieno il potenziale dei dati e tradurli in informazioni significative per la strategia aziendale.

Indice

Elenco delle figure

Capitolo 1

Introduzione

L'evoluzione delle tecnologie legate al Big Data e all'apprendimento automatico ha innescato una vera e propria rivoluzione nei modi in cui i dati vengono gestiti, analizzati e sfruttati. La crescente complessità di queste metodologie è emersa come un elemento cruciale, consentendo non solo alle grandi imprese, ma anche a quelle di dimensioni più contenute, di personalizzare in modo avanzato le proprie strategie di marketing proattivo.
L'analisi del comportamento del cliente, basata su algoritmi predittivi sofisticati, ha raggiunto un livello di precisione tale da anticipare non solo le esigenze, ma anche i desideri del cliente. Ciò ha aperto la strada a un'offerta di prodotti e servizi altamente personalizzati. Settori come quello bancario e finanziario hanno subito profonde trasformazioni, abbandonando l'approccio tradizionale delle filiali fisiche in favore di servizi di home banking e digitalizzazione completa. Questa transizione ha migliorato non solo l'accessibilità e l'efficienza dei servizi, ma ha anche generato un ricco patrimonio informativo.
L'utilizzo di algoritmi di intelligenza artificiale assiste ora i clienti non solo nella scelta degli investimenti, ma anche nell'implementazione di soluzioni immediate attraverso l'impiego di chatbot avanzati.

Nel campo della ricerca, la bioinformatica ha vissuto una rivoluzione grazie alle nuove tecniche di sequenziamento del DNA. Questo progresso ha portato a un'abbondanza di dati genomici, riducendo drasticamente i costi e i tempi necessari per acquisirli. L'analisi di questo vasto patrimonio genetico attraverso algoritmi avanzati ha aperto nuove prospettive nella classificazione automatica e nella previsione di indicatori medici, consentendo una comprensione più approfondita della complessità biologica.
L'enorme quantità di dati disponibili ha spinto l'analisi avanzata verso l'adozione di tecniche di data mining sempre più sofisticate. Tuttavia, questa crescita esponenziale ha portato con sé sfide significative, come la diversità dei dati, il loro volume considerevole e l'esigenza di scalabilità per fronteggiare richieste dinamiche. Ciò ha stimolato l'innovazione in architetture e modelli di analisi dati, con una transizione da soluzioni tradizionali enterprise a approcci più agili e adattabili.
Il punto di partenza per ciò che oggi consideriamo come Big Data può essere fatto risalire al 2004 con l'introduzione del paradigma di programmazione MapReduce da parte dei ricercatori di Google, Jeffrey Dean e Sanjay Ghemawat. Questa innovazione, basata sul principio del Divide et Impera, ha rivoluzionato la capacità di elaborazione distribuita, aprendo la strada all'analisi su vasta scala e all'utilizzo efficiente delle risorse in ambienti cloud.

La necessità di elaborare grandi volumi di dati in tempo reale ha dato origine a un'architettura a pipeline. Questo modello, in cui ogni componente processa l'output del precedente, è stato plasmato per rispondere alle esigenze di contesti dinamici e mutevoli. Le fasi di ingestione, pre-processing, analisi, persistenza e visualizzazione definiscono un approccio strutturato che consente l'analisi continua e dinamica dei dati in tempo reale.
Nonostante il consolidato paradigma MapReduce, si è dimostrato insufficiente per affrontare l'elaborazione in tempo reale. L'introduzione di nuovi modelli di processamento streaming parallelo, come quelli proposti dalle architetture Lambda e lo stack SMACK, ha rappresentato una tappa fondamentale nell'ottimizzazione delle operazioni di analisi dei Big Data. Queste soluzioni offrono efficienza e tempi di risposta più rapidi, garantendo una gestione ottimale dei dati in tempo reale.

1.1 Architettura LAMBDA

L'architettura LAMBDA è frutto dell'ingegno di Nathan Marz, programmatore statunitense noto per aver ideato nel 2011 il framework *Apache Storm* per l'elaborazione in streaming, successivamente acquisito da Twitter. La concezione di questa architettura è stata presentata nel libro "*Big Data - Principles and best practices of scalable real-time data systems*" [5], in cui l'autore ha proposto un approccio architetturale innovativo, unendo al tradizionale metodo offline (*batch*) delle tecniche di elaborazione online (*streaming*) per il trattamento dei dati in tempo reale. La tipica architettura Lambda può essere suddivisa in tre strati distinti:

- **Batch Layer**: deputato all'elaborazione batch, tipicamente con paradigma MapReduce, svolgendo anche il compito di persistere il master dataset e garantirne l'allineamento e la consistenza.
- **Speed Layer**: l'elemento di innovazione, deputato al processamento dei dati in tempo reale attingendo dal flusso di input.
- **Serving Layer**: interfaccia verso l'esterno, offre servizi di interrogazione e fornisce in maniera trasparente all'utente un unico virtuale dataset costituito aggregando opportunamente i dati prodotti dalle componenti batch e real time.

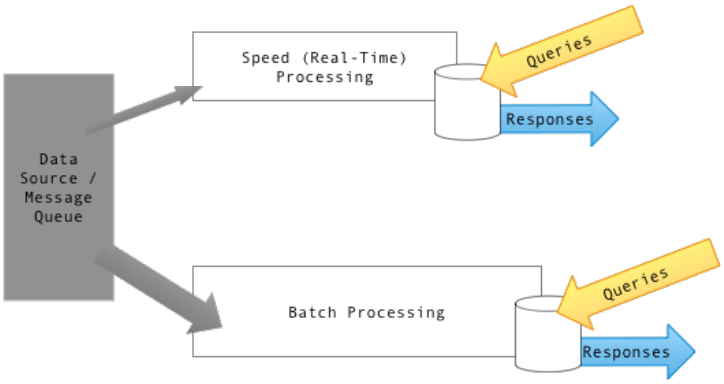

Figura 1.1: Diagramma di un'architettura LAMBDA [6]

Il flusso elaborativo è esemplificato dai seguenti passi:

- nella fase di *ingestion*, i dati raw vengono catturati in modo simultaneo dalla sorgente. Questi dati alimentano un master dataset che funge da deposito immutabile e aggiornabile solo in modo incrementale, contenendo una copia inalterabile dei dati nel formato originale. Contestualmente, le componenti di processamento in *streaming* vengono attivate per elaborare continuamente i dati in arrivo
- durante la fase di processamento, le componenti *batch* intraprendono un approccio di ricalcolo totale o incrementale sul master dataset. Queste operazioni coinvolgono un alto volume di dati, presentano una latenza significativa e hanno un *throughput* relativamente basso. Parallelamente, le componenti *speed* processano lo stesso set di dati acquisito dalle operazioni *batch*, fornendo risultati parziali attraverso tecniche di processamento real-time o pseudo real-time. Questo processo, eseguito a frequenze estremamente elevate e con bassa latenza, offre una visione istantanea e continua del blocco di dati trattato. I risultati ottenuti in questa fase vengono resi disponibili al *Serving Layer*

- il *Serving Layer*, infine, agisce come punto di convergenza, aggregando i risultati provenienti sia dalle operazioni *batch* che da quelle in *streaming*. Il suo ruolo principale è quello di fornire un'interfaccia unificata e coerente all'utente finale. Questo strato si occupa inoltre di sovrascrivere eventuali risultati parziali derivanti dall'elaborazione in *streaming* con quelli ottenuti dalle operazioni *batch*, garantendo così una visione complessiva e accurata dei dati trattati. Ad esempio, se nel processing in streaming vengono prodotti risultati parziali per un determinato set di dati, il *Serving Layer* li integra con i risultati ottenuti dalle operazioni *batch* per presentare all'utente una visione completa e aggiornata

Questo innovativo modello architetturale introduce notevoli elementi di cambiamento. Innanzitutto, la mutabilità gestita tradizionalmente all'interno dei database enterprise viene sostituita dall'immutabilità del master dataset, il quale preserva la storia inalterata nel tempo nella sua forma originale. Aggiunge continuamente nuovi flussi di dati senza mai rimuovere quelli precedenti. I due strati si compensano reciprocamente in modo trasparente per gli utenti finali, offrendo una visione completa dei dati trattati. Tale visione deriva da analisi approfondite sui dati storici e fornisce insight e statistiche parziali sui dati attuali. Il master dataset persiste tipicamente utilizzando file system distribuiti (come *Apache HDFS*) o database distribuiti *NoSQL* (come *MongoDB* orientato ai documenti, *Redis* basato su chiavi, *Neo4J* orientato ai grafi). La combinazione di calcolo e storage parallelo distribuito consente di massimizzare il principio di località del dato. Ciò implica il coordinamento e la distribuzione dell'elaborazione e del carico computazionale verso i nodi di lavoro che ospitano i blocchi di dati da elaborare. Nel contesto di questa architettura, i principali framework di riferimento includono *Apache Hadoop* [3] e *Apache Spark* [11] per le componenti *batch*, mentre per le componenti *streaming* sono menzionati *Apache Storm* e *Spark Streaming*, che fa parte dell'ecosistema di *Spark*.

1.2 Lo Stack SMACK

La stack *SMACK* è un insieme di tecnologie open source progettato per gestire e analizzare grandi volumi di dati in tempo reale. L'acronimo *SMACK* rappresenta le principali componenti dello stack: *Spark*, *Mesos*, *Akka*, *Cassandra* e *Kafka*. Ogni componente svolge un ruolo cruciale nel garantire una gestione efficiente, scalabile e resiliente dei dati.

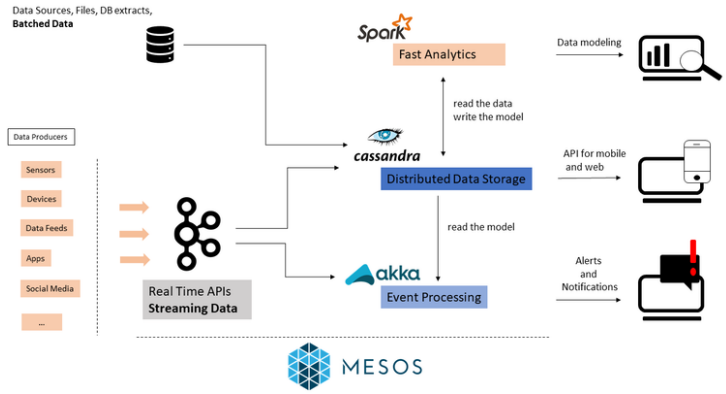

Figura 1.2: Architettura dello Stack SMACK [12]

Analizziamo ogni componente:

- **Apache Spark**: un framework di elaborazione distribuita che offre un modello di programmazione flessibile e veloce. Spark facilita l'analisi di dati in tempo reale e fornisce un'interfaccia per il processing batch, streaming, SQL e machine learning. La sua architettura a DAG (Directed Acyclic Graph) consente una pipeline di lavoro ottimizzata, promuovendo la velocità e la scalabilità.

- Modello di Programmazione Resiliente a Fallimenti (*RDD*): Spark utilizza un modello di programmazione chiamato Resilient Distributed Dataset (*RDD*). Gli *RDD* sono collezioni distribuite di oggetti immutabili che possono essere elaborati in parallelo. Questo modello fornisce una gestione automatica dei fallimenti, consentendo a Spark di tollerare la perdita di dati durante l'elaborazione.
- *Spark Core*: Questo è il nucleo del framework e fornisce le funzionalità di base di Spark, inclusa l'implementazione degli *RDD* e l'engine di esecuzione distribuita. Spark Core è responsabile della gestione delle risorse, della tolleranza ai guasti e dell'interazione con lo storage distribuito.
- *Spark SQL*: Questo modulo consente l'elaborazione di dati strutturati utilizzando SQL in Spark. Fornisce un'interfaccia per l'interrogazione dei dati attraverso comandi SQL standard, consentendo agli utenti di eseguire query su dati strutturati insieme alle elaborazioni batch e streaming.
- *Spark Streaming*: È un modulo di Spark che consente l'elaborazione di dati in tempo reale. Utilizzando intervalli di microbatching, Spark Streaming elabora i dati in piccoli lotti, offrendo una soluzione per l'analisi dei dati in streaming senza dover ricorrere a un sistema separato.
- *MLlib* (Machine Learning Library): Questa libreria integrata in Spark offre un'ampia gamma di algoritmi di machine learning scalabili. È progettata per semplificare lo sviluppo e l'implementazione di modelli di machine learning su grandi dataset distribuiti.
- *GraphX*: È una libreria per l'elaborazione di grafi e l'analisi di dati basati su grafi. Grazie a GraphX, Spark può gestire in modo efficiente operazioni su grafi su scala distribuita, rendendolo adatto per applicazioni di social network, analisi delle reti, e altro ancora.

- **Mesos** è un sistema di gestione delle risorse distribuite che fornisce un'astrazione delle risorse della macchina, consentendo l'esecuzione di applicazioni su cluster eterogenei. Mesos offre una gestione dinamica delle risorse, consentendo alle applicazioni di condividere efficientemente risorse come CPU e memoria, rendendo più flessibile l'utilizzo delle infrastrutture.

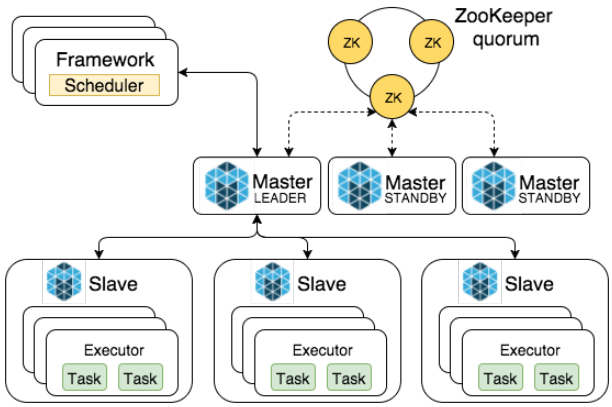

Figura 1.3: Diagramma dell'architettura di MESOS [1]

- **Akka** è un framework per la creazione di sistemi distribuiti e concorrenti basato sul modello di attori. Gli attori sono entità leggere che comunicano tra loro tramite messaggi, consentendo la costruzione di applicazioni altamente scalabili e resilienti. Akka facilita la gestione della concorrenza e fornisce strumenti per affrontare le sfide dei sistemi distribuiti.
- **Apache Cassandra** è un sistema di gestione del database distribuito altamente scalabile e altamente disponibile. Progettato per gestire grandi quantità di dati su cluster di macchine, Cassandra offre una distribuzione automatica dei dati e una replicazione multi-regionale, garantendo l'affidabilità e la resilienza del sistema anche in caso di guasti hardware o nodi inattivi.

- **Apache Kafka** è una piattaforma di streaming distribuito per la gestione di feed di dati in tempo reale. Kafka consente la pubblicazione e il consumo di eventi su larga scala, garantendo la robustezza e l'affidabilità delle pipeline di dati in tempo reale. La sua architettura a *topic* consente una facile scalabilità e gestione dei flussi di dati.

Insieme, queste tecnologie formano la stack SMACK, offrendo una soluzione completa per l'elaborazione e l'analisi di dati in tempo reale su scala distribuita. La combinazione sinergica di Spark, Mesos, Akka, Cassandra e Kafka fornisce un ambiente potente e flessibile per affrontare le sfide dell'analisi dei big data in tempo reale.

Capitolo 2

Attori: Scalabilità e Resilienza nel Modello di Programmazione

Il modello ad attori è un paradigma di programmazione concorrente che si basa sulla concettualizzazione di attori come unità di esecuzione indipendenti che comunicano tra loro attraverso lo scambio di messaggi. Questo approccio fornisce un modo efficace per gestire la concorrenza e costruire sistemi distribuiti altamente scalabili e resilienti.
L'uso del modello ad attori è motivato dalla necessità di semplificare lo sviluppo di applicazioni che richiedono un'elaborazione parallela efficiente. Invece di utilizzare tradizionali meccanismi di gestione della concorrenza come i thread, il modello ad attori permette agli sviluppatori di progettare applicazioni in cui attori indipendenti svolgono ruoli specifici e comunicano attraverso messaggi. Questa modularità favorisce una progettazione più chiara e una gestione più agevole della complessità.
Inoltre, il modello ad attori promuove la scalabilità orizzontale, consentendo la distribuzione di attori su più nodi di un sistema distribuito o di un cluster. Questa caratteristica è fondamentale per affrontare la crescente richiesta di elaborazione di grandi quantità di dati su ambienti distribuiti, fornendo una soluzione per costruire sistemi che possano crescere in modo flessibile in risposta ai carichi di lavoro.
La resilienza è un altro aspetto cruciale del modello ad attori. Gli attori possono essere supervisionati da altri attori genitori, e in caso di fallimenti, il sistema può intraprendere azioni correttive come il ripristino dell'attore o l'adottare misure di compensazione. Questo approccio contribuisce a costruire applicazioni robuste che possono gestire errori senza compromettere l'integrità dell'intero sistema.

2.1 La differenza tra concorrenza e parallelismo

Il parallelismo e la concorrenza sono concetti chiave nell'ambito dell'elaborazione e dell'esecuzione di task nei sistemi informatici. Il parallelismo si riferisce all'esecuzione simultanea di più attività indipendenti allo scopo di migliorare l'efficienza complessiva del sistema. In pratica, ciò implica la suddivisione di un problema in sotto-problemi indipendenti, che vengono eseguiti contemporaneamente da diverse unità di elaborazione.
D'altra parte, la concorrenza riguarda la gestione simultanea di più attività, ma non necessariamente in modo parallelo. Le attività concorrenti possono essere eseguite in momenti diversi e possono condividere risorse, richiedendo un meccanismo di coordinamento per evitare conflitti. La concorrenza è spesso associata a sistemi multitasking, dove le attività vengono commutate rapidamente in modo apparentemente simultaneo, anche se in realtà vengono eseguite in modo alternato.

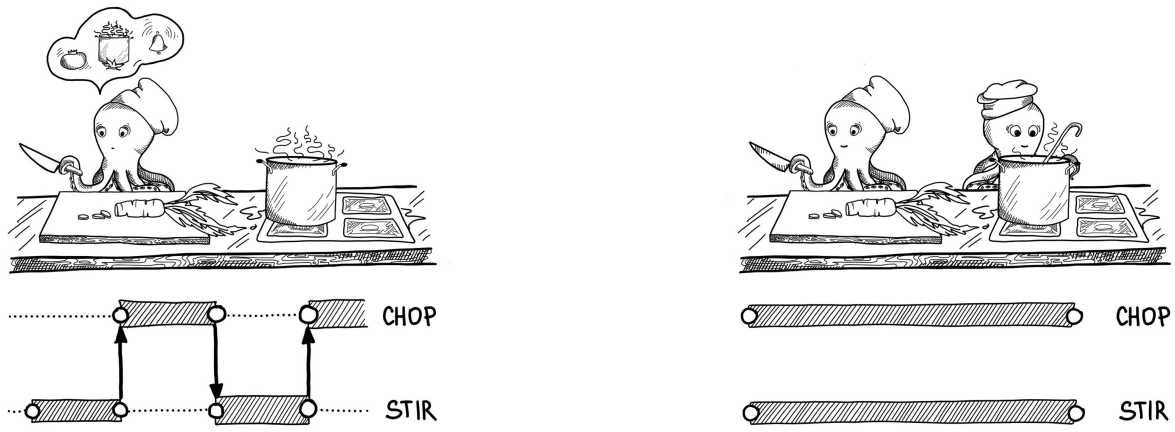

Figura 2.1: La differenza tra concorrenza e parallelismo [2]

2.2 Il modello ad attori

Il modello ad attori rappresenta un paradigma di programmazione che affonda le sue radici nelle teorie matematiche e nelle pratiche di elaborazione parallela. La sua storia si intreccia con il desiderio di affrontare le sfide emergenti dell'elaborazione parallela e distribuita. Un momento cruciale per la formulazione del modello ad attori fu l'opera "*A universal modular ACTOR formalism for artificial intelligence*" [4] scritta da Carl Hewitt, Peter Bishop, e Richard Steiger nel 1973. Questo lavoro pionieristico introdusse i fondamenti teorici del modello ad attori, proponendo una visione innovativa dell'elaborazione parallela.
Il modello ad attori si basa sull'idea che l'unità fondamentale di elaborazione sia l'"attore", un'entità computazionale indipendente che comunica con gli altri attraverso lo scambio di messaggi. Ogni attore è autonomo e possiede il proprio stato interno, eseguendo operazioni in risposta ai messaggi ricevuti. Questo approccio decentralizzato favorisce la scalabilità e la gestione dell'elaborazione parallela.
Gli attori operano in modo asincrono, il che significa che possono procedere indipendentemente senza dover aspettare che altri attori completino le loro operazioni. La comunicazione tra attori avviene esclusivamente tramite lo scambio di messaggi, e questo meccanismo di comunicazione definisce il flusso di controllo del programma.
Un aspetto distintivo del modello ad attori è la sua natura distribuita e la gestione della concorrenza. Gli attori possono esistere e operare su sistemi distribuiti, consentendo un'elaborazione parallela su nodi separati. La decentralizzazione offre una maggiore resistenza agli errori e una migliore tolleranza ai guasti, contribuendo a creare sistemi più robusti.
Negli anni successivi alla sua introduzione, il modello ad attori ha ispirato lo sviluppo di linguaggi di programmazione specifici, come Erlang, che implementano i principi fondamentali del modello ad attori. Questi linguaggi sono stati utilizzati per la creazione di sistemi distribuiti, sistemi di telecomunicazione e applicazioni orientate agli eventi.

Gli attori nel modello ad attori possiedono alcune caratteristiche chiave:

- **Autonomia**: Ogni attore ha un proprio stato interno e un comportamento indipendente. Gli attori possono eseguire operazioni in modo asincrono, senza dover aspettare il completamento di altre attività.
- **Comunicazione tramite messaggi**: Gli attori interagiscono solo scambiandosi messaggi. Un attore può inviare un messaggio ad un altro attore, e questo scambio di messaggi costituisce il principale meccanismo di comunicazione nel modello.
- **Località e indirizzamento**: Gli attori possono esistere su sistemi distribuiti e hanno un indirizzo unico che li identifica. Questo consente agli attori di comunicare tra loro anche su reti distanti.

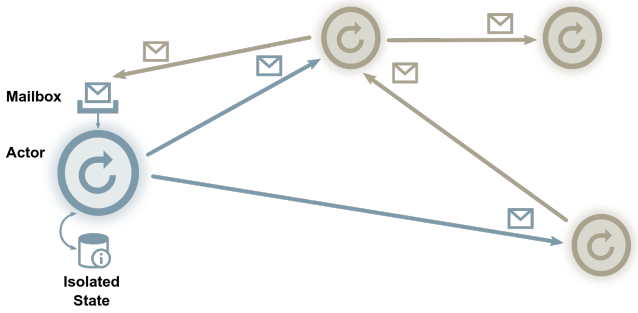

Figura 2.2: Modello ad attori [10]

- **Isolamento dello stato**: L'isolamento dello stato significa che uno stato interno di un attore è accessibile solo da quell'attore specifico. Gli attori possono condividere informazioni solo scambiandosi messaggi, mantenendo così un livello di indipendenza.
- **Elaborazione di messaggi**: Gli attori eseguono operazioni in risposta ai messaggi che ricevono. Queste operazioni possono includere la modifica del proprio stato interno, l'invio di messaggi ad altri attori o la creazione di nuovi attori.

Sotto il profilo tecnico, un attore può essere considerato essenzialmente come un *thread leggero*, ovvero un processo di esecuzione che è debolmente accoppiato con gli altri e si distingue per la sua capacità di inviare e ricevere messaggi. Questo concetto è fondamentale nel contesto del modello ad attori, in cui l'interazione tra attori avviene attraverso il paradigma del message-passing.
L'invio e la ricezione di messaggi avvengono in modo asincrono, il che significa che il mittente non rimane bloccato in attesa di una risposta da parte del destinatario. Questo approccio favorisce un'elaborazione parallela ed efficiente, consentendo agli attori di procedere con le proprie attività senza essere vincolati al completamento delle operazioni degli altri.
Ogni attore dispone di una coda di messaggi nota come *mailbox*, dove i messaggi in arrivo vengono accodati in attesa di essere elaborati. Il processo di elaborazione dei messaggi avviene secondo il principio *FIFO* (*First In First Out*), garantendo un ordine sequenziale nell'elaborazione. Un singolo attore, inoltre, è in grado di processare un solo messaggio alla volta, il che contribuisce a mantenere l'ordine e la coerenza nell'esecuzione delle operazioni.
Un aspetto interessante è la possibilità di creare mailbox con priorità differenti per lo stesso attore, consentendo così un'ottimizzazione del processamento dei messaggi in base alla loro importanza relativa. Questa caratteristica offre una maggiore flessibilità nell'implementazione di strategie di gestione dei messaggi, soprattutto quando si tratta di situazioni in cui alcuni messaggi potrebbero richiedere un'attenzione prioritaria rispetto ad altri.

Una regola fondamentale per garantire un efficace disaccoppiamento tra attori nel modello ad attori è il principio del *tell-don't-ask* [9]. Questo principio sottolinea l'importanza di utilizzare chiamate asincrone non bloccanti per lo scambio di messaggi.
Nel contesto del modello ad attori, due modalità principali di comunicazione possono essere distinte:

- **Fire-and-forget** (*tell*): Consiste nell'invio di un messaggio senza attendere una risposta immediata. Eventuali risposte vengono ricevute attraverso la successiva ricezione di un nuovo messaggio al termine dell'azione richiesta. Questo approccio favorisce la non dipendenza da risposte immediate, contribuendo così a mantenere un flusso di esecuzione asincrono.
- **Request-response** (*ask*): Questo approccio, più tradizionalmente adottato nelle applicazioni enterprise di tipo web, comporta l'invio di un messaggio sincrono seguito dall'attesa di una risposta. Tuttavia, è fortemente sconsigliato nel contesto delle applicazioni reattive poiché va in contrasto

con i principi fondamentali di reattività. Inoltre, richiede un numero maggiore di thread per gestire le chiamate di callback, introducendo potenziali inefficienze nell'esecuzione.

In fase di creazione degli attori nel modello ad attori, è possibile definire gerarchie e strutturare una relazione padre-figlio, configurando così una struttura ad albero. Questa organizzazione si rivela utile per automatizzare l'inoltro di nuovi messaggi verso i figli degli attori o per instradare messaggi in entrata. In alternativa, gli attori possono fungere da *guardiani*, supervisionando i loro figli e ricevendo notifiche in caso di malfunzionamenti. Questa capacità di definire relazioni gerarchiche aggiunge un livello di organizzazione e controllo, contribuendo a una gestione più efficiente del sistema ad attori.

2.3 Programmazione concorrente

La programmazione concorrente è un paradigma di programmazione che coinvolge l'esecuzione simultanea di più task o processi all'interno di un'applicazione. Questa approccio è particolarmente utile per sfruttare al massimo le risorse di un sistema e migliorare le prestazioni complessive di un'applicazione. Tuttavia, la programmazione concorrente presenta anche diverse criticità che devono essere gestite con attenzione. Uno dei principali problemi è rappresentato dalle condizioni di gara, situazioni in cui più processi cercano di accedere o modificare una risorsa condivisa contemporaneamente, portando a risultati imprevedibili o errori. La gestione della sincronizzazione diventa cruciale per evitare inconsistenze nei dati e problemi di accesso concorrente. Altri aspetti critici includono la gestione dei *deadlock*, situazioni in cui più processi sono bloccati perché ognuno attende che l'altro rilasci una risorsa, e la difficile individuazione e correzione degli errori dovuti alla concorrenza, che possono manifestarsi in modo sporadico e rendere complessa la fase di debugging. Inoltre, la progettazione di algoritmi e strutture dati per la concorrenza richiede un'approfondita comprensione delle problematiche legate a questa modalità di esecuzione, rendendo la programmazione concorrente più complessa rispetto a quella sequenziale.

Algorithm 1: Prelievo denaro da conto corrente

Input : Amount a
Output: a

```
1 if deposit > a then
2 |   deposit ← (deposit − a)
3 |   return a;
4 else
5 |   return 0;
6 end
```

Cosa accade se due utenti chiamano contemporaneamente la funzione prelievo? Il primo utente potrebbe superare la verifica di disponibilità dell'importo prelevato dal conto, ma prima di effettuare l'effettivo prelievo, un secondo utente potrebbe anch'esso richiedere un prelievo, riducendo il saldo disponibile sul conto al di sotto dell'importo richiesto dal primo utente. Il risultato sarebbe un saldo negativo nel conto. Per affrontare questa situazione, se desideriamo implementare tale funzionalità in linguaggio C, con supporto alla concorrenza in un contesto *multithreading* (dove ogni utente è un thread separato), sarebbe necessario utilizzare costrutti di sincronizzazione, come quelli forniti dalla libreria *pthread*.
Questa soluzione potrebbe essere implementata bloccando l'intera attività di verifica e prelievo nella *regione critica*, ma ciò comporterebbe un aumento della complessità nella codifica e un potenziale rischio di errori nell'implementazione. Lo stesso si applicherebbe se dovessimo affrontare questa sfida utilizzando Java e sfruttando i costrutti di sincronizzazione forniti dalle API della *JSE*.
Invece, mediante l'adozione del modello ad attori e l'utilizzo del toolkit Akka, l'intero processo verrà ricondotto a un'esecuzione atomica della funzione e allo scambio di messaggi per interagire con gli utenti. Il codice Scala per Akka appare notevolmente simile alla pseudo-codifica inizialmente presentata, con l'unica eccezione rappresentata dalla restituzione dell'output attraverso un messaggio.

Algorithm 2: Prelievo denaro da conto corrente con Scala e Akka

Input: Amount a

```
1 value = 0
2 if deposit > a then
3 |   deposit ← (deposit − a)
4 |   value = deposit;
5 end
6 sender ! gotCash(value)
```

In merito alle sfide legate alla concorrenza, è importante considerare tre *illusioni* comuni nel contesto delle soluzioni enterprise basate sul paradigma di programmazione orientato agli oggetti:

- L'**incapsulamento** suggerisce che lo stato interno di un oggetto sia privato e accessibile solo tramite appositi metodi, noti come accessori e modificatori per la lettura e la scrittura rispettivamente. Questo approccio è efficace in contesti single-thread, ma mostra i suoi limiti in ambienti multi-thread. In tali scenari, le chiamate possono intrecciarsi in modo non deterministico, compromettendo la possibilità di garantire una sequenza predefinita di esecuzione, a meno che non si ricorra nuovamente a meccanismi di sincronizzazione.
- La **gestione della memoria condivisa** in architetture multi-core presenta sfide significative. Le moderne architetture dei computer incorporano più unità di calcolo, ognuna con la propria memoria cache a livello CPU, non visibile alle altre. Questo può portare a inconsistenze tra i thread appartenenti a core diversi che manipolano lo stesso oggetto memorizzato nelle rispettive cache. L'unico rimedio è la sincronizzazione delle cache. Alcuni linguaggi offrono direttive per istruire il processore a recuperare un oggetto esclusivamente dalla memoria principale, evitando l'utilizzo delle cache delle CPU. Ad esempio, in Java, la keyword *volatile* può essere utilizzata per definire un attributo di una classe e renderlo *thread-safe*.
- L'esecuzione di **task in background** è comune in quasi tutti i linguaggi di programmazione, consentendo la specifica di un riferimento a cui richiamare una funzione al termine dell'esecuzione. Tuttavia, si presenta una problematica quando il thread incaricato di eseguire il task incontra un errore. In questo caso, il thread non dispone di mezzi per comunicare con il thread chiamante e notificare l'errore, poiché il call stack sarà inutilizzabile a seguito dell'eccezione. Questa situazione può rendere difficile la gestione degli errori nei task in background.

Il modello ad attori si propone di affrontare le sfide associate alla concorrenza e alla distribuzione di codice attraverso l'implementazione di poche e chiare regole:

- **Comunicazione Asincrona Esclusiva**: il modello ad attori adotta un approccio di scambio asincrono di messaggi come unico mezzo di comunicazione tra attori. Questa scelta impedisce situazioni di blocco incontrollato, consentendo l'esecuzione atomica e deterministica delle istruzioni contenute nel codice della funzione. Ciò garantisce la coerenza dello stato incapsulato nell'oggetto.
- **Elaborazione FIFO** (*First-In-First-Out*): gli attori gestiscono i messaggi in arrivo secondo il principio FIFO, assicurando che i messaggi vengano elaborati nell'ordine in cui sono stati ricevuti nella coda dell'attore.
- **Stato Locale e Non Condiviso**: lo stato degli attori è locale e non condiviso, comunicato esclusivamente attraverso lo scambio di messaggi. Questo approccio favorisce l'isolamento e la coerenza dei dati tra gli attori.
- **Gestione delle Eccezioni**: il modello ad attori gestisce le eccezioni in modo da notificare al chiamante eventuali fallimenti tramite un messaggio di risposta. Questa pratica contribuisce a una gestione robusta degli errori nel contesto della concorrenza distribuita.
- **Tecniche di Supervisione**: vengono adottate tecniche di supervisione per monitorare il comportamento degli attori e decidere l'azione appropriata in risposta a eventuali fallimenti di un attore. Questa capacità di supervisione migliora la resilienza del sistema distribuito.

In sintesi, l'applicazione di queste regole e l'adesione a pratiche consigliate consentono lo sviluppo di codice concorrente distribuito sicuro. Va notato che, sebbene il modello ad attori non possa garantire lo stesso livello di parallelizzazione delle applicazioni native multithread, fornisce comunque strumenti per rendere il sistema scalabile. L'adozione di queste regole semplifica la gestione della concorrenza e della distribuzione, contribuendo a una progettazione più affidabile e resiliente del software.

2.4 La cassetta degli attrezzi

Esaminiamo adesso l'approccio di Akka come framework per realizzare il paradigma degli attori all'interno delle nostre soluzioni, avvalendoci del linguaggio di programmazione Scala. Questo linguaggio, per la sua intrinseca natura funzionale, si dimostra particolarmente idoneo in situazioni in cui la scalabilità è un requisito essenziale.

2.4.1 Il linguaggio di programmazione SCALA

Il termine *Scala* è una contrazione di SCAlable LAnguage ed è stato coniato nel 2001 presso la Scuola Politecnica Federale di Losanna, in Svizzera, da Martin Odersky, l'attuale fondatore di Lightbend. Quest'azienda si occupa del monitoraggio degli sviluppi del linguaggio, della formazione e della consulenza professionale.
Scala si posiziona all'incrocio della programmazione funzionale, della programmazione ad oggetti e dei linguaggi di scripting. Questa peculiarità deriva dalla sua capacità di offrire i costrutti tipici della programmazione funzionale, i principi e le regole della programmazione ad oggetti, consentendo nel contempo un prototipaggio rapido grazie al linguaggio conciso che lo assimila a linguaggi di scripting tradizionali.
Il linguaggio è particolarmente adatto per la scrittura di codice scalabile, basandosi su concetti chiave come l'immutabilità dei dati, il passaggio di funzioni e la valutazione pigra delle istruzioni.
Durante la compilazione, il codice Scala genera bytecode Java, che viene interpretato ed eseguito dalla *Java Virtual Machine* (JVM). Ciò rende Scala completamente interoperabile con il codice scritto in Java o altri linguaggi che generano bytecode Java, offrendo un evidente vantaggio nel riutilizzo di librerie già esistenti.

Figura 2.3: Il logo del linguaggio SCALA [8]

Le caratteristiche distintive del linguaggio Scala comprendono:

- **Funzioni di Prima Classe**: Questa caratteristica, ereditata dai linguaggi funzionali, consente alle funzioni di essere trattate come variabili. Possono essere assegnate a variabili e passate come argomenti a altre funzioni. Questo paradigma offre una flessibilità notevole nel design del software, facilitando la creazione di codice modulare e riutilizzabile.
- **Inferenza dei Tipi**: Scala presenta un sistema di inferenza dei tipi che elimina la necessità di dichiarare esplicitamente il tipo di una variabile. Il tipo può essere dedotto automaticamente durante la valutazione dell'espressione. Questo aspetto semplifica il processo di scrittura del codice, rendendo Scala particolarmente adatto per attività di prototipazione rapida, riducendo il costo di progettazione del software e consentendo una maggiore operatività immediata.

- **Funzioni Anonime**: Trattando le funzioni come oggetti, Scala permette la definizione di funzioni anonime in-line. Questo significa che è possibile implementare una funzione specificando solo il corpo, l'input e l'output dell'implementazione senza dover dichiarare staticamente la funzione. Questa flessibilità è utile per semplificare la scrittura del codice e migliorare la leggibilità.
- **Valutazione Pigra**: Scala adotta la valutazione pigra, che ritarda l'esecuzione delle espressioni fino al momento in cui i valori ottenuti sono effettivamente necessari. Questo approccio contribuisce all'efficienza del programma, evitando la valutazione di espressioni superflue e ottimizzando l'utilizzo delle risorse.
- **Pattern Matching**: Oltre a esaminare il valore in sé, Scala consente il pattern matching, che comporta la verifica di modelli di scomposizione del valore in diverse parti. Questa caratteristica si rivela particolarmente potente nell'implementazione di logiche complesse e nella gestione di strutture dati intricate.
- **Immutabilità**: Per promuovere la scalabilità e la sicurezza in contesti concorrenti, Scala incoraggia l'uso di valori immutabili. Questo significa che, una volta definito un valore, non può essere modificato. Tale pratica riduce il rischio di inconsistenze causate da variabili mutevoli, contribuendo a una progettazione del software robusta e thread-safe.
- **Utilizzo di Tuple**: Scala offre il supporto per le tuple, che sono strutture dati immutabili contenenti un numero fissato di elementi, ognuno con un proprio tipo distinto. Le tuple sono utili per rappresentare collezioni eterogenee di dati in modo conciso ed efficiente.
- **Orientamento Completo agli Oggetti**: A differenza di alcuni linguaggi come Java, Scala è completamente orientato agli oggetti. Ogni elemento, che siano funzioni, letterali stringa o numeri, è definito e trattato come un oggetto. Questa coerenza nella progettazione del linguaggio semplifica la comprensione e la gestione del codice.

2.4.2 Ambiente AKKA

Come precedentemente enunciato, Akka rappresenta un toolkit ideato da Jonas Bonér all'interno della società Typesafe, ora denominata Lightbend, di cui Bonér è cofondatore insieme al creatore di Scala, Martin Odersky. Questa piattaforma consente la progettazione di programmi che operano in modo parallelo e concorrente, aderendo al modello ad attori. Ciò viene realizzato senza richiedere al programmatore la conoscenza dettagliata e l'impegno richiesto nella risoluzione delle comuni problematiche che emergono nei sistemi concorrenti. Tali sfide vanno dalla gestione low-level di processi e thread fino alla sincronizzazione, affrontando fenomeni come deadlock, starvation e race condition. L'obiettivo chiave di Akka è di assistere il programmatore nell'implementazione di soluzioni parallele, asincrone, altamente scalabili e resilienti. Merita rilevare che Akka è interamente implementato utilizzando il linguaggio Scala.
Akka sposa la filosofia di Carl Hewitt e del suo modello ad attori, concependo l'attore come l'unità primitiva e universale di elaborazione per la programmazione concorrente. Hewitt, con grande lungimiranza, progettò questo modello con l'anticipazione di un ambiente in cui l'elevata parallelizzazione fosse resa possibile dall'uso su larga scala di microprocessori indipendenti e geograficamente distribuiti, capaci di comunicare a basse latenze. Questa visione di Hewitt è diventata ora una realtà con l'avvento del Cloud Computing, e Akka è stato appositamente concepito per adottare e implementare tali principi.
Adesso, procederemo all'analisi approfondita delle caratteristiche principali di Akka. Esploreremo il livello di astrazione fornito per la programmazione concorrente, il meccanismo di comunicazione basato sul passaggio di messaggi, l'utilizzo di processi leggeri, il ciclo di vita degli attori, la capacità di auto-ripristino (self-healing), la trasparenza della localizzazione, il routing dei messaggi e l'implementazione del clustering per soddisfare i requisiti di scalabilità.

Livello di atrazione per la programazione concorrente

L'entità attoriale, ritenuta l'elemento cardine per il computo, la conservazione e la trasmissione di informazioni, si distingue per la sua prudenza nell'evitare di esporre il proprio stato interno. La comunicazione tra attori avviene mediante lo scambio di messaggi, che sono *immutabili*, *asincroni* e *non vincolanti*.

Figura 2.4: Il logo del toolkit AKKA [7]

Questa strategia consente una flessibilità e una scalabilità notevoli nel contesto di sistemi concorrenti. L'attore, come unità primitiva, attua la sua logica di business in risposta agli stimoli esterni che gli vengono trasmessi tramite messaggi. Tuttavia, è importante sottolineare che le operazioni consentite da un attore sono limitate al modello ad attori, il che implica la possibilità di inviare messaggi, creare nuovi attori e definire comportamenti interni in risposta a specifici messaggi.
L'impiego di messaggi immutabili è una scelta progettuale intrinseca, concepita per mitigare possibili problematiche di concorrenza e inconsistenza di stato. Questa caratteristica assicura che attori distinti, nel medesimo istante temporale, visualizzino versioni consistenti e uniformi dello stato di un attore. Ciò contribuisce in modo significativo all'integrità e alla coerenza del sistema, mitigando gli effetti indesiderati che potrebbero emergere in scenari concorrenti complessi.
Esaminiamo ora il processo di definizione di un attore, con particolare riferimento agli esempi che seguiranno, tutti redatti in linguaggio Scala. A tale scopo, delineeremo l'evoluzione di un programma *Hello World*, un esercizio storico spesso adoperato per illustrare i concetti basilari di sintassi e funzionamento. Questa scelta ci consentirà di analizzare in dettaglio come il linguaggio Scala si integri e si adatti al paradigma ad attori in questo specifico contesto.

```
//actor object
object HelloWorldActor{

    //istanziazione attore tramite factory method props
    def props(name:String):Props=Props(new HelloWorldActor())

    //definizione messaggio
    case class Greet(msg: String)

}

// actor class
class HelloWorldActor()extends Actor {

    def receive = {

        //ricezione messaggio
        case Greet(name) =>

            println("Hello␣" + name)

    }

}
```

All'interno del caso pratico illustrato, l'entità denominata *Greeter* assume il compito di accogliere un nome specifico e di renderlo visibile attraverso la stampa su dispositivo video. Questo processo di trasmissione di informazioni è mediato da un messaggio immutabile, precisamente delineato dalla struttura definita nella classe Greet.
Per approfondire questo concetto, è rilevante esaminare in dettaglio la dinamica sottostante. In questo contesto, la classe *Greeter* non è solo un semplice oggetto che riceve e stampa un nome; essa incarna un attore all'interno del modello ad attori. L'adozione del paradigma ad attori consente una gestione asincrona ed efficiente degli stimoli esterni, in questo caso, rappresentati dalla ricezione di messaggi immutabili.

La scelta di utilizzare un messaggio immutabile, come definito dalla classe Greet, è una decisione progettuale strategica. Tale struttura immutabile, per sua natura, contribuisce a prevenire problematiche legate alla concorrenza e alla consistenza dello stato. Inoltre, essa riflette il principio fondamentale dell'immutabilità nell'ambito del modello ad attori, dove ogni messaggio rappresenta un'istanza unica e indipendente di dati.
In questo scenario, la comunicazione tra attori, manifestata attraverso il passaggio di messaggi, sottolinea l'approccio decentralizzato e distribuito del modello ad attori. Questo design non solo favorisce la modularità del sistema, ma anche la creazione di un ambiente altamente scalabile e reattivo.
Per cogliere appieno la complessità e la potenza di questo approccio, è utile analizzare ulteriori aspetti relativi alla gestione degli attori, alla definizione di comportamenti in risposta a messaggi specifici e alla gestione del ciclo di vita dell'attore stesso. In questo modo, possiamo apprezzare l'efficacia e la versatilità del paradigma ad attori nell'implementazione di soluzioni software resilienti, parallele e altamente scalabili.

```
object App {

    def main(args: Array[String]) {

        //creazione Akka system
        val system = ActorSystem("hello-world")

        //creazione attore HelloWorld
        val greeter = system.actorOf(HelloWorldActor.props(), "greeter")

        //invio messaggio fire-and-forget
        greeter ! Greet("Elon")

        Await.ready(system.whenTerminated, Duration.Inf)

    }
}
```

Affinché l'intero processo venga avviato, è necessario inizializzare l'ambiente creando il sistema, procedendo all'istanziazione del nuovo attore attraverso l'utilizzo del metodo di fabbrica (*factory method*) denominato *props* e, successivamente, inoltrare un messaggio asincrono e non bloccante alla classe chiamante.

Per esplorare più approfonditamente questa sequenza di operazioni, è vantaggioso considerare i dettagli operativi di ciascuna fase. In primo luogo, la creazione del sistema costituisce un passaggio cruciale. In questo contesto, *system* rappresenta il quadro globale entro il quale gli attori interagiscono, e la sua istanziazione stabilisce l'ambiente di esecuzione in cui gli attori agiscono. Successivamente, l'azione di istanziare un nuovo attore mediante il metodo *props* introduce la flessibilità e la configurabilità nell'ambito della creazione degli attori. Questo metodo, funzionando come una sorta di fabbrica, accetta i parametri necessari per definire il comportamento e le caratteristiche dell'attore in questione. Tale approccio favorisce la modularità e la personalizzazione nell'implementazione degli attori, rendendo possibile la creazione di diverse istanze con comportamenti specifici.
L'invio di un messaggio asincrono e non bloccante alla classe chiamante costituisce l'ulteriore passo in questa sequenza operativa. Questo processo di comunicazione asincrona permette alla classe chiamante di proseguire la sua esecuzione senza dover attendere una risposta immediata dall'attore appena creato. Tale aspetto è cruciale in un contesto di programmazione concorrente, in quanto consente alle attività di procedere in modo parallelo, migliorando l'efficienza e la reattività del sistema nel suo complesso.
Inoltre, è interessante notare come la scelta di un messaggio non bloccante contribuisca alla fluidità e alla dinamica del flusso di esecuzione complessivo. Questo approccio consente di evitare situazioni in cui la classe chiamante si bloccherebbe in attesa di una risposta sincrona, mantenendo invece la continuità delle operazioni in modo agile e responsivo.

Passaggio di messaggi

Akka concepisce la definizione di un attore in modo simile a una struttura dell'object-oriented, ma si distingue notevolmente in quanto non espone alcun metodo o attributo direttamente. L'interazione con un attore avviene esclusivamente attraverso l'invio di messaggi, un paradigma di comunicazione event-driven implementato mediante il *message passing*. Ogni attore è dotato di un proprio buffer denominato *mail-box*, una sorta di casella di posta, dove i messaggi vengono accodati e successivamente estratti in modo sincrono e bloccante, seguendo l'ordine di ricezione (*First-In-First-Out*). Ogni messaggio è associato a un thread specifico per l'elaborazione. Va notato che non esiste una garanzia assoluta di recapito corretto dei messaggi. Nel caso in cui la consegna fallisca, il messaggio viene instradato verso una mail-box di sistema denominata *dead letters*.
In un contesto asincrono, se, ad esempio, gli attori A e B inviano messaggi all'attore C in successione, non c'è certezza che i messaggi saranno ricevuti nel medesimo ordine di invio. La sola garanzia fornita dal sistema è che, all'interno di una coppia mittente/destinatario specifica, i messaggi saranno elaborati dal destinatario nell'ordine di invio.
Gli attori possono comunicare tra loro mediante azioni di *tell* o *ask*, rispettivamente utilizzate per comunicazioni di tipo *fire-and-forget* o *request-response*. Tuttavia, al fine di mantenere i principi di disaccoppiamento, è fortemente scoraggiato lo scambio di messaggi bloccanti di tipo *ask*. Tale pratica è considerata una violazione dei principi fondamentali del modello ad attori, poiché compromette la scalabilità, favorendo invece l'utilizzo del tipo *tell*.
Questa configurazione specifica del modello ad attori enfatizza la natura asincrona delle interazioni, incoraggiando un design che favorisca la modularità, la reattività e la scalabilità all'interno dei sistemi che adottano Akka. La comprensione approfondita di queste dinamiche consente di sfruttare al massimo i benefici offerti dal paradigma ad attori nella costruzione di applicazioni robuste e altamente performanti.

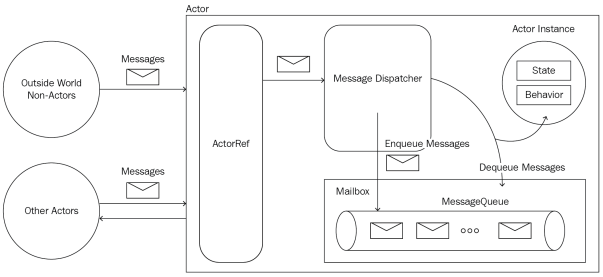

Figura 2.5: Diagramma che mostra i componenti interni di un Attore e come interagiscono tra di loro

Esempio di un invio di messaggio asincrono *fire-and-forget*:

```
greeter ! Greet("Elon");
```

Invio messaggio sincrono *request/response*, con gestione attesa valore di ritorno mediante costrutto *Future*:

```
//sync actor class
class HelloWorldSyncActor() extends Actor {

  def receive = {

  //ricezione messaggio
  case Greet(name) =>

    sender ! Greet("Hello " + name + ", Hello world!")
}
}

//attivazione attore sincrono
val syncGreeter  = system.actorOf(HelloWorldSyncActor.props(), "syncGreeter")
```

```
implicit val timeout = Timeout(1 second)

//invio messaggio request/response
val future = syncGreeter ? Greet("Elon2")
val result = Await.result(future, timeout.duration)
```

L'attore, in questa configurazione specifica, abbandona la pratica di stampare il messaggio a video, invece opta per restituirlo al chiamante. Nel corso di questa operazione, il chiamante si troverà in uno stato di blocco in attesa della risposta. Questo comportamento è dettato dall'impiego dell'operatore *?* (*ask*) anziché *!* (*tell*) per l'invio del messaggio.
È di rilievo esplorare più a fondo il concetto di *ask pattern*. Questa scelta determina una variante nel paradigma di recapito del messaggio noto come *at-most-once*. Tale paradigma offre la garanzia che la consegna del messaggio avvenga al massimo una sola volta, ma non fornisce alcuna assicurazione sull'effettiva avvenuta consegna.
È cruciale comprendere le differenze con il paradigma *at-least-once*, il quale garantisce l'effettiva consegna del messaggio ma non impedisce che il messaggio venga recapitato più volte. Un altro paradigma, denominato *exactly-once*, offre la certezza che la consegna si verifichi una e una sola volta.
Akka, coerentemente con i principi fondamentali del modello ad attori, adotta il paradigma *at-most-once*. Questa scelta strategica è orientata a evitare l'introduzione di complessità nella codifica e di overhead prestazionali. Il sistema sfrutta la capacità di supervisione degli attori per gestire eventuali fallimenti, una caratteristica intrinseca del modello ad attori. In contrasto, un approccio *at-least-once* richiederebbe l'implementazione di un protocollo personalizzato per la gestione di *ACK/NACK*, al fine di ottenere feedback sulla consegna. Analogamente, un approccio *exactly-once* implicherebbe l'implementazione di logiche complesse per garantire la consegna e scartare eventuali messaggi duplicati. Questa riflessione sottolinea l'acume di Akka nel bilanciare la garanzia di consegna dei messaggi con l'obiettivo di mantenere un approccio snello e performante, integrando al contempo la capacità di gestire dinamicamente eventuali fallimenti all'interno dell'ecosistema ad attori.

Ciclo di vita degli attori

Il percorso evolutivo di un attore è caratterizzato da tre differenti categorie di eventi, ognuna delle quali assume un ruolo significativo nella gestione dinamica dell'entità attoriale:

- **Creazione dell'Attore** (Stato: *Started*): la fase iniziale è rappresentata dalla creazione dell'attore, la quale lo colloca nello stato di *started*. La generazione di un attore avviene mediante l'applicazione del metodo *actorOf* sul *actorSystem*. Questo processo inaugura la vita dell'attore, preparandolo per le interazioni e le responsabilità che andrà ad assumere nel corso del suo ciclo di vita.
- **Riavvio dell'Attore** (*Strategia di Supervisione*): un evento cruciale si verifica quando un attore affronta un fallimento e si attiva la strategia di supervisione. In questo contesto, il riavvio dell'attore è innescato su richiesta del supervisore, il quale, con saggezza, decide di riportare in vita l'attore. Tale processo di riavvio implica l'esecuzione di hook specifici, come *preRestart* e *postRestart*, permettendo l'iniezione di eventuali segmenti di codice aggiuntivo durante queste fasi critiche. Questo approccio flessibile consente di gestire dinamicamente i fallimenti e di reintegrare l'attore nella sua operatività con le correzioni necessarie.
- **Terminazione dell'Attore** (Stato: *Stopped*): la fase di stop segna la conclusione dell'operatività dell'attore, trasferendolo nello stato di *stopped*. Questa terminazione può essere avviata sia dal supervisore che dall'attore stesso. Una volta in questo stato, i successivi messaggi indirizzati all'attore verranno instradati verso la mail-box di sistema denominata *dead letters*. È rilevante notare che nello scenario di default, lo stop di un attore comporta la cessazione delle attività anche per tutti i suoi attori figli.

Esaminando attentamente ciascuna di queste fasi, emergono le dinamiche complesse e le considerazioni strategiche che permeano il ciclo di vita di un attore all'interno di un sistema basato su Akka. Questa prospettiva dettagliata fornisce una panoramica esaustiva delle diverse transizioni e delle opportunità per

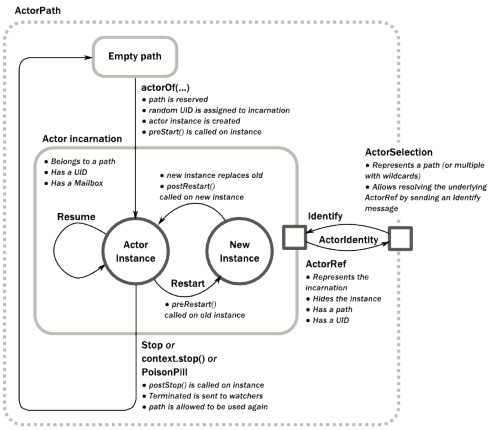

Figura 2.6: Il ciclo di vita di un attore nell'ecosistema AKKA

gestire eventuali problematiche, sottolineando l'elasticità e la robustezza insite nell'approccio ad attori implementato da Akka.

Processo leggero

Nel contesto di Akka, ciascun attore si manifesta come un processo leggero, denominato *light*, il che consente una notevole agilità esecutiva, con un'approssimativa capacità di gestire fino a 2.7 milioni di attori per ogni gigabyte di memoria RAM. L'orchestrazione di questo scenario avviene quando un attore, ricevendo un messaggio nella propria mail-box, si impegna nel processo di elaborazione grazie a un thread assegnatogli dal componente *Dispatcher*. La gestione dell'aspetto multi-threaded è strategicamente attuata all'interno di questo componente, evidenziando la robustezza e la scalabilità del toolkit.

Akka offre nativamente diverse opzioni di dispatcher, ciascuna progettata per adattarsi a specifiche esigenze di gestione dei thread. Tra queste opzioni, troviamo:

- *SimpleDispatcher*: frutta i thread-pool per condividere l'uso di thread tra attori diversi. Questa strategia si rivela particolarmente utile nel contesto di un sistema in cui la condivisione di risorse è una priorità, consentendo un'efficace gestione dell'allocazione di thread tra attori distinti.
- *PinnedDispatcher*: assegna un thread dedicato a ciascun attore, promuovendo un'approccio più isolato e specifico per la gestione di ogni entità attoriale. Questa configurazione è vantaggiosa in situazioni in cui è necessaria una separazione marcata tra le attività degli attori.
- *CallingThreadDispatcher*: esegue più invocazioni riutilizzando lo stesso thread, una scelta solitamente riservata a scopi di debug. Questa opzione permette di semplificare il tracciamento e la comprensione delle chiamate, facilitando le operazioni di debugging.

L'implementazione di queste strategie di dispatcher consente di ottimizzare l'allocazione e l'utilizzo dei thread in modo flessibile, consentendo a ogni attore di sfruttare efficientemente le risorse disponibili. Questo approccio, in cui il numero di attori allocabili per unità di memoria supera ampiamente il numero di thread istanziabili, sottolinea l'efficienza e la scalabilità della progettazione di Akka, contribuendo così a costruire sistemi altamente reattivi e performanti.

Fault tolerance

Esaminiamo con attenzione l'approccio di Akka alla gestione della *fault tolerance*, un aspetto cruciale per garantire la robustezza e l'affidabilità dei sistemi attoriali. Durante il processo di creazione degli attori, emerge la possibilità di definire una gerarchia, consentendo di stabilire se l'attore appena creato instaurerà una relazione padre-figlio o discenderà direttamente dalla radice del sistema. Questa scelta innesca la formazione di una relazione di supervisione, la cui dinamica può essere plasmata attraverso l'adozione di specifiche strategie, sia personalizzate che fornite nativamente dal toolkit.
All'inizio del ciclo vitale del sistema attoriale, si delineano tre attori radice: il *root guardian*, che rappresenta la radice effettiva dell'albero gerarchico; lo *user guardian*, padre di tutti gli attori dell'applicazione; e il *system guardian*, responsabile della supervisione degli attori di sistema.
Ad esempio, in caso di insuccesso di un attore figlio, il supervisore può intraprendere diverse azioni, tra cui il riavvio, la terminazione o la delega del problema al proprio supervisore. Questo paradigma, noto come *let it crash*, si distingue per la scelta di consentire il fallimento dell'attore supervisionato anziché implementare complesse logiche di gestione delle eccezioni. In questo contesto, la gestione centralizzata del fallimento avviene nel supervisore, che applica un'azione in base alla strategia previamente definita.

Le strategie di gestione dei fallimenti possono essere personalizzate per rispondere alle specifiche esigenze del sistema o selezionate tra quelle fornite dal toolkit:

- **Restart**: l'attore viene ricreato attraverso il proprio factory method, riprendendo immediatamente l'attività. Gli altri attori continuano a referenziare l'attore utilizzando un riferimento logico costante esternamente.
- **Resume**: l'attore prosegue con l'elaborazione dei messaggi, ignorando il fallimento.
- **Stop**: l'attore viene terminato, impedendogli di continuare a elaborare messaggi.
- **Escalate**: il supervisore trasferisce la problematica al proprio supervisore.

Adottare l'approccio *let it crash* offre notevoli vantaggi. Consente al supervisore di gestire gli errori sollevati dagli attori supervisionati senza compromettere le relazioni con gli altri attori. Questi ultimi, quando possibile, continuano a scambiare messaggi con una nuova istanza dell'attore fallito piuttosto che con lo stesso attore ripristinato, assicurando un recupero senza soluzione di continuità.

Per illustrare concretamente questa metodologia, consideriamo il seguente frammento di codice. Qui definiamo un attore supervisore, *HelloWorldSupervisorActor*, che stabilisce una strategia comune per gli attori figli supervisionati, come ad esempio "*HelloWorldActor.*" Questa strategia è selezionata in base all'eccezione lanciata dal figlio, evidenziando la flessibilità e l'efficacia nella gestione delle situazioni di errore all'interno del contesto attoriale. Questo approccio riflette la sofisticata architettura di Akka, che permette una gestione dinamica e scalabile degli errori in scenari reali e complessi.

```
  class HelloWorldSupervisorActor extends Actor {

  // creazione attore figlio
  val child = context.actorOf(Props(new HelloWorldActor), "greeter-actor")

  // avvio supervisione
  context.watch(child)

  // definizione strategia di supervisione
  override def supervisorStrategy =
    AllForOneStrategy() {
      case StopException => Stop
      case RestartException => Restart
      case ResumeException => Resume
      case _: Exception => Escalate
    }

  // inoltro al figlio messaggio ricevuto
  override def receive: Receive = {
    case msg => child forward msg
  }
}
```

Location transparency

Il meccanismo di localizzazione degli attori in Akka si basa su un servizio di naming distribuito, un pilastro cruciale per l'architettura di sistemi attoriali che mirano alla scalabilità e alla distribuzione. Ogni singolo attore è distintamente identificato all'interno di uno spazio utente condiviso attraverso un riferimento logico noto come *actorRef*, il quale è successivamente mappato da un percorso che segue la struttura gerarchica degli attori fino a raggiungere la radice del sistema.
Parallelamente, è presente un riferimento fisico noto al sistema, denominato *actorPath*, che consente il riconoscimento univoco di un attore. Questo riferimento è rappresentato da una stringa strutturata nel seguente formato:

```
protocol://actorSystem@hostname:port/actorPath‘
```

Per esempio, consideriamo l'*actorPath*:

```
akka.tcp://mySystem@myHost:9001/user/myActor‘
```

che identifica l'attore denominato *myActor* nel contesto del sistema Akka chiamato *mySystem*. Quest'ultimo è in esecuzione sulla macchina denominata *myHost*, ascoltando sulla porta 9001 attraverso il protocollo *tcp*.

Quando il programmatore necessita di localizzare un attore, utilizza esclusivamente il riferimento logico, trascurando la necessità di conoscere la reale locazione dell'attore, che potrebbe essere sia locale che remota. La responsabilità di risolvere il riferimento logico in un riferimento fisico è affidata al sistema, garantendo così una gestione trasparente e efficiente della distribuzione degli attori.
Il riferimento logico, oltre a essere utilizzato per operazioni di lookup degli attori da parte del sistema, viene implicitamente incorporato come mittente nei messaggi, semplificando le interazioni tra attori. Inoltre, attraverso l'utilizzo del componente *ActorSelection*, è possibile recuperare il riferimento logico insieme ad altri riferimenti. Questo componente consente l'applicazione di *wildcard*, facilitando la localizzazione su larga scala di più attori. L'architettura di naming distribuito implementata in Akka costituisce un elemento chiave che offre un alto livello di astrazione. Ciò consente ai programmatori di concentrarsi sulla logica applicativa, mentre il sistema gestisce in modo efficiente e trasparente la complessità della distribuzione degli attori. Tale approccio fornisce una flessibilità e una scalabilità superiori nell'implementazione di sistemi complessi e distribuiti.

Routing, scalabilità e clustering

Per assicurare la sostenibilità delle prestazioni di fronte alla crescente domanda, è essenziale adottare strategie di scalabilità nel sistema di attori. Questo può avvenire attraverso gli approcci tradizionali di *scale up* e *scale out*.
Nel caso dello *scale up*, si potenzia il sistema verticalmente mediante l'aggiunta di attori locali al nodo esistente. In alternativa, con lo "*scale out*", si introduce un nuovo nodo remoto nel cluster di sistema, includendo nuovi attori.

La gestione efficiente delle richieste è ottimizzata attraverso l'impiego di attori specializzati noti come *Router*. Questi attori indirizzano le richieste attraverso azioni definite per i Routee, creando e gestendo diverse istanze dello stesso tipo di attore mediante un *actor-pool*. Un'ulteriore ragione per l'adozione del routing potrebbe derivare dalla necessità di selezionare il *Routee* in base al messaggio trasmesso o allo stato interno del router. Akka offre una gamma di strategie di routing predefinite:

- *Round Robin*: selezione ciclica dei Routee senza priorità.
- *Random*: selezione casuale dei Routee.
- *Smallest Mailbox*: instradamento basato sulla mail-box con meno messaggi in coda.
- *Balancing Pool*: condivisione della stessa mail-box tra tutti i Routee, con il router che distribuisce i messaggi ai Routee liberi.

- *Broadcast*: distribuzione del medesimo messaggio a tutti i Routee.
- *Scatter-Gather First Completed*: simile al broadcast, ma con il router che attende la prima risposta tra i Routee entro un intervallo di tempo specifico.
- *Tail Chopping*: il router invia il messaggio a un Routee selezionato casualmente, applicando questa logica a intervalli regolari fino alla ricezione della prima risposta.
- *Consistent Hashing*: instradamento basato sull'hashing del messaggio.

Ricordando la flessibilità di Akka nell'implementazione di applicazioni distribuite, ciò è realizzato attraverso la definizione di tecniche di clustering. Questo consente l'espansione orizzontale mediante l'aggiunta di nodi al cluster, in cui ciascun nodo appartiene a una rete peer-to-peer e mantiene la sincronizzazione attraverso il protocollo di gossip con gli altri nodi. Ciò agevola il lookup degli attori in modo indipendente dalla loro localizzazione fisica, promuovendo la *location transparency*, e permette il rilevamento delle cadute di altri nodi. L'obiettivo primario del cluster è garantire la fault tolerance in caso di fallimento di un nodo e il load balancing per distribuire in modo efficiente i messaggi ai vari nodi. Inoltre, un *actor system* può essere suddiviso in partizioni dislocate su nodi diversi per soddisfare esigenze prestazionali specifiche.

Bibliografia

[1] *Architettura MESOS*. URL: https://github.com/OddExtension5/SMACK-Resources/blob/master/assets/Mesos-Overview.png.

[2] *Concurrency vs Parallelism*. URL: https://freecontent.manning.com/concurrency-vs-parallelism/.

[3] *Hadoop*. URL: https://hadoop.apache.org/.

[4] Steiger R. Hewitt C. Bishop P. «A universal modular ACTOR formalism for artificial intelligence». In: (1973).

[5] Nathan Marz James Warren. *Big Data: Principles and best practices of scalable realtime data systems*. A cura di Manning Pubblications Co. 2015.

[6] *Lambda Architecture*. URL: https://en.wikipedia.org/wiki/Lambda_architecture.

[7] *Logo AKKA*. URL: https://akka.io/resources/images/akka_full_color.svg.

[8] *Logo SCALA*. URL: https://www.lightbend.com/assets/images/brand/scala/scala-logos/svg/scala-full-color.svg.

[9] Flower M. *Tell don't ask*. URL: https://martinfowler.com/bliki/TellDontAsk.html.

[10] *Modello ad attori*. URL: https://berb.github.io/diploma-thesis/original/resources/actors.svg.

[11] *Spark*. URL: https://spark.apache.org/.

[12] *Stack SMACK*. URL: https://www.cologne-intelligence.de/fileadmin/_processed_/4/2/csm_Blog-SMACK-Architektur_53383bcafd.png.