Alma Mater Studiorum · Università di Bologna
Campus di Cesena

SCUOLA DI SCIENZE

Corso di Laurea Triennale in Ingegneria e Scienze Informatiche

DESIGN E PROTOTIPAZIONE DI UN MIDDLEWARE PER APPLICAZIONI AGGREGATE LOCATION-BASED

Tesi in

Programmazione ad Oggetti

Relatore:
Prof. MIRKO VIROLI

Presentata da:
LINDA VITALI

Correlatore:
Dott. ROBERTO CASADEI

Sessione: IV
Anno Accademico 2018/2019

With a little help from my Fap

Abstract

Negli ultimi anni stiamo assistendo ad un notevole incremento di dispositivi intelligenti capaci di interagire tra loro, dando vita a ciò che viene chiamato "ubiquitous computing", ovvero computazione onnipresente. Le tecnologie attuali, infatti, permettono ad oggetti del quotidiano di diventare supporti computazionali, in grado di comunicare sulla rete internet e potenzialmente scambiarsi informazioni. Il paradigma dell'Aggregate Computing si inserisce in tale contesto con lo scopo di introdurre un nuovo modello di programmazione che faciliti lo sviluppo di sistemi pervasivi, spostando l'attenzione dal comportamento del singolo device, a quello di un gruppo di dispositivi che collaborano per un obiettivo comune. Le caratteristiche di questi ultimi, come protocolli di comunicazione, sistemi operativi e architetture, sono però molto diversificate e si rende quindi necessario un livello software, chiamato middleware, che aumenti il livello di astrazione nella realizzazione di applicazioni aggregate. Il lavoro presentato in questa tesi tratta l'analisi generale di un middleware a supporto di applicazioni di questo tipo e mostra il processo di progettazione, nonché di realizzazione, di una sua versione "peer to peer", in cui ogni device è localizzato tramite coordinate Gps. Il software è stato realizzato applicando il modello ad attori, sfruttandone il supporto predisposto per applicazioni concorrenti e distribuite.

Indice

Introduzione

Il contesto tecnologico in cui ci troviamo oggigiorno, pone sfide sempre più complesse per quanto riguarda lo sviluppo di applicazioni distribuite e pervasive a causa del grande numero e della grande varietà di dispositivi che vi partecipano.

L'Aggregate computing è un paradigma di programmazione che si pone come obiettivo quello di sostenere sistemi di tale portata, spostando il focus dalla computazione tradizionale su singolo dispositivo, ad una più ampia visione, che guarda alla collaborazione di tanti dispositivi atta al raggiungimento di un obiettivo comune. Nonostante questo però, è necessario offrire un ulteriore supporto che permetta allo sviluppatore di concentrarsi sullo sviluppo applicativo e di focalizzarsi il meno possibile sugli aspetti architetturali e fisici, di non considerare i vari protocolli di comunicazione che possono coesistere in un sistema così diversificato e di non dover sottostare ai diversi servizi offerti dai sistemi operativi che i dispositivi utilizzano.

Software che offrono la possibilità di risolvere questi problemi sono i middleware. Infatti, un programma di questo tipo, si pone come intermediario tra applicazioni e hardware mettendo a disposizione servizi per sistemi distribuiti complessi.

L'obiettivo della tesi, è quello di realizzare un middleware che renda più facile la realizzazione di applicazioni aggregate. Nelle successive pagine viene analizzato il problema in modo approfondito e vengono illustrate le possibili modalità di implementazione: "peer to peer" e "client/server". Vengono poi proposte progettazione e sviluppo del metodo "peer to peer" evidenziando il funzionamento del sistema con un esempio applicativo.

Nel primo capitolo vengono affrontati tutti gli elementi che hanno permesso di capire quale fosse lo stato dell'arte sia nell'Aggregate computing che nel mondo middleware.

Inoltre vengono mostrate in breve le tecnologie che sono state utilizzate all'interno del progetto. Nei capitoli successivi invece, viene descritto il processo di sviluppo vero e proprio seguendo un approccio *top-down*.

Infatti, nel secondo capitolo, viene effettuata l'analisi e vengono esposti i requisiti che il sistema deve avere, facendo riferimento alle necessità dell'Aggregate Computing. Nel terzo capitolo viene mostrato il design del sistema a due livelli di dettaglio: in primo luogo se ne illustra l'architettura ad attori, successivamente si approfondisce l'aspetto di progettazione illustrando i pattern utilizzati. Si scende ad un ulteriore livello di dettaglio nel capitolo quattro, in cui si trovano alcuni aspetti dell'effettiva implementazione dei concetti descritti nei capitoli precedenti mostrando alcuni estratti di codice.

Nell'ultimo capitolo si trova la descrizione delle tecnologie di testing, quali *ActorTestKit* e *ScalaTest*. In seguito viene descritto un caso di studio basato sull'architettura precedentemente illustrata, che utilizza il programma aggregato di Scafi chiamato Gradiente, per determinare le distanze dei nodi da un nodo sorgente.

Infine, si trovano conclusioni sul lavoro svolto e considerazioni su sviluppi futuri, elencati come possibili implementazioni o test da effettuare sul middleware.

Capitolo 1

Background

1.1 Aggregate computing

L'aggregate computing [1] è un paradigma di programmazione nato per sopperire al problema di coordinamento tra device in sistemi embedded pervasivi. Si pone come obiettivo quello di superare i limiti degli approcci di programmazione tradizionali, che al crescere della complessità comportano problemi di design, scalabilità, riusabilità, ecc., convogliando l'attenzione dalla visione del singolo dispositivo come parte di un insieme, verso lo studio del comportamento globale generato da una moltitudine di elementi "aggregati" aumentando così il livello di astrazione. In questo modo gli aspetti del singolo device vengono messi in secondo piano. L'obiettivo è quello di semplificare progettazione, creazione e manutenzione di sistemi software complessi.

L'intero modello di programmazione è basato sul *field-calculus* e può essere descritto tramite i *computational field*. Questi ultimi sono funzioni che mappano ogni device, localizzato in un determinato punto dello spazio-tempo, ad un valore computazionale. Il field calculus invece, è un modello teorico che permette di descrivere un insieme di primitive che vengono poi tradotte in costrutti di base atti a manipolare i campi computazionali.

Fondamentale nell' Aggregate computing e nei sistemi pervasivi in generale, è il concetto di *context* (contesto), elemento su cui si basa ogni computazione, che permette di ottenere informazioni quali:

- L'identificativo del device che sta eseguendo la computazione.
- Il set dei vicini con il risultato della loro ultima computazione (chiamato *export*).

- Lo stato dei sensori locali.
- Informazioni sui sensori ambientali, che forniscono dati, utili su piano computazionale, sullo stato dei vicini del dispositivo.

L'esecuzione di un programma aggregato consiste nell'eseguire su ogni device, periodicamente e in modo asincrono, un *round*, al quale viene passato il contesto e, una volta terminata la computazione, ne restituisce l' *export* relativo. A questo vengono alternati periodi di inattività.

In particolare, ogni dispositivo che partecipa ad un sistema aggregato, durante un round esegue i seguenti passaggi:

1. Creazione del contesto.
2. Esecuzione del programma aggregato basato sul contesto appena creato. Questa azione genera un export(descrittore del round) e un output (uno o più valori).
3. Propagazione dell'export ai vicini.
4. Messa in esecuzione degli attuatori contro l'output prodotto in precedenza.

In un sistema aggregato, il comportamento di ogni nodo che interagisce con gli altri, tutti immersi in uno stesso ambiente, è conseguente a quello di tutti gli altri dispositivi, ed è il risultato quindi, di un calcolo globale. I device di un sistema aggregato, eseguono e interagiscono ripetutamente tra loro secondo il modello di esecuzione sopra descritto.

1.2 Middleware

Da un punto di vista computazionale, un middleware è un livello software che risiede tra il sistema operativo e le applicazioni. Più in generale può essere definito come entità che permette allo sviluppatore di astrarre da aspetti di sistema, hardware o di rete, in costrutti di programmazione intuitivi ed accessibili facilitando alcuni aspetti dello sviluppo software come: accesso a servizi, gestione delle risorse, funzionalità di alto livello, comunicazione, sicurezza, coordinazione ecc.

1.2.1 Middleware in sistemi pervasivi e Iot

La necessità di utilizzare una parte software come un middleware nei sistemi pervasivi, nasce dall' ampia eterogeneità di dispositivi che compongono un sistema di questo tipo (o uno di tipo IoT) (Figura 1.1) ognuno dei quali utilizza diversi modi di comunicazione, diversi sistemi operativi, gestisce diversamente le proprie risorse, si distingue per capacità computazionale - che può essere anche molto bassa, come ad esempio i tag RFID-, portata di memoria o anche capacità della batteria. Queste caratteristiche vengono nascoste dal livello middleware il cui compito è, quindi, quello di garantire interoperabilità tra tutti i componenti e di fornire i servizi necessari in modo che tutti i dispositivi possano partecipare al sistema.

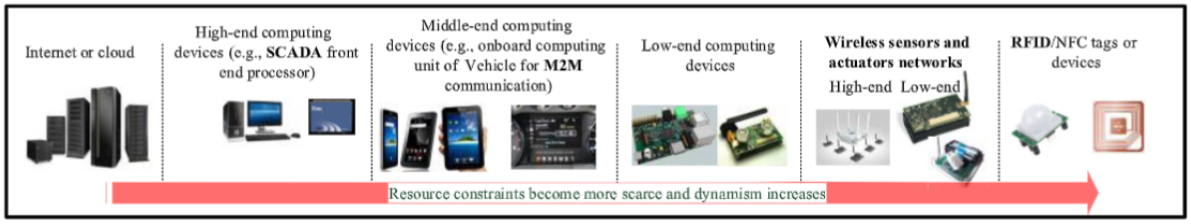


Figura 1.1: La grande eterogeneità di device che possono comporre sistemi Iot o pervasivi [2].

I sistemi pervasivi mirano a sviluppare applicazioni per l'utente in modo che quest'ultimo abbia un accesso continuo ai servizi. Vi sono numerosi domini applicativi che spaziano dall'ambito healthcare ad ambienti smart, industria, trasporto e logistica. Non esiste , un singolo middleware che possa sopperire a tutte le problematiche di un sistema pervasivo e si deve quindi tener conto delle esigenze date dalla situazione. Tuttavia si possono individuare temi comuni di cui uno sviluppatore di applicazioni pervasive deve tipicamente tener conto in modo da adattare i servizi in modo adeguato:

- Gestione e coordinamento del grande numero di nodi collegati tra loro e dei servizi forniti.
- Raccolta, memorizzazione ed elaborazione di enormi quantità di dati.
- Capacità di risposta a guasti hardware.
- Gestione dei casi di corruzione dei dati o del contesto (falsi positivi, falsi negativi).

- Gestione delle risorse (capacità di calcolo e di energia).
- Dinamicità della topologia di rete .
- Affidabilità, sicurezza e privacy.

I requisiti di un middleware di questo tipo sono solitamente l'alta flessibilità, la riusabilità, la scalabilità, l'adattamento e la consapevolezza del contesto [3]. Si possono individuare tre aspetti principali per la progettazione di un middleware:

1. *Astrazioni di programmazione*

   Definiscono interfacce di programmazione ad alto livello, grazie alle quali il programmatore può dimenticarsi dell'ifrastruttura sottostante e dedicarsi allo sviluppo. Nel creare questi elementi, si tiene conto di: livello di astrazione, che si riferisce al modo di concepire l'ambiente pervasive, se come insieme di nodi distribuiti o come un singolo sistema virtuale in cui gira un singolo programma centralizzato; paradigma di programmazione, che può essere context-based, component-based o decentralized interaction-based; tipo di interfaccia, che si riferisce allo stile di API da fornire. Gli aspetti di astrazione sopraelencati, possono variare in base alle specifiche di una applicazone e in base all'infrastruttura. Le applicazioni pervasive possono adottare diversi modelli di programmazione: basato sul contesto, sui componenti o su una interazione decentralizzata.

2. *Servizi di sistema e supporto a run-time*

   I servizi sono il meccanismo centrale di un middleware pervasive e vengono utilizzati dall'applicazione tramite le suddette interfacce. Possono essere di sistema, quindi occuparsi di raccogliere i dati ed elaborarli, nonché della gestione del contesto creato contro gli stessi, di gestione della fruibilità dei servizi, di affidabilità e sicurezza, oppure possono essere servizi a run-time che si occupano di gestire funzioni di pianificazione delle attività, elaborazione locale, comunicazione interprocesso o con altri dispositivi, controllo della memoria e controllo dell'energia. Forniscono inoltre supporto per il deployment, l'esecuzione e la gestione dei dispositivi e della rete. Il concetto che sta alla base della computazione di un sistema pervasivo è la *context-awareness*. Nel contesto sono racchiuse e combinate insieme tutte le informazioni rilevanti arrivate al sistema derivate da fattori fisici, come rumore, luce,

temperatura; informatici, condizioni di rete, stato di memoria e CPU; o relativi all'utente, come la posizione, l'attività, le relazioni sociali ecc. In un sistema distribuito non basta ragionare sui singoli dati, è quindi necessario che dal contesto vengano estrapolate informazioni di più alto livello per capire in quale stato si trovi il sistema.

3. *Architettura del sistema*

   Forniscono le implementazioni delle astrazioni. Solitamente, le architetture middleware per sistemi pervasivi adottano scelte di progettazione top-down o bottom-up. Le modalità di controllo del sistema possono essere centralizzate, dove vi è un componente centrale che controlla il resto dei dispositivi e prende decisioni, o decentralizzate, in cui nessun dispositivo prende decisioni definitive ma collabora con altri per raggiungere obiettivi comuni. Tuttavia, per applicazione pervasive è sconsigiabile adottare una soluzione centralizzata dove è presente un solo *point of failure*, e si preferisce adottare soluzioni distribuite. L'interazione tra componenti di un sistema in ambito pervasive può avvenire tramite primitive di comunicazione quali: scambio di messaggi, spazio di tuple, modello publish/subscribe.

In figura 1.2 viene mostrato un framework di riferimento che identifica i servizi di base di un middleware per pervasive.

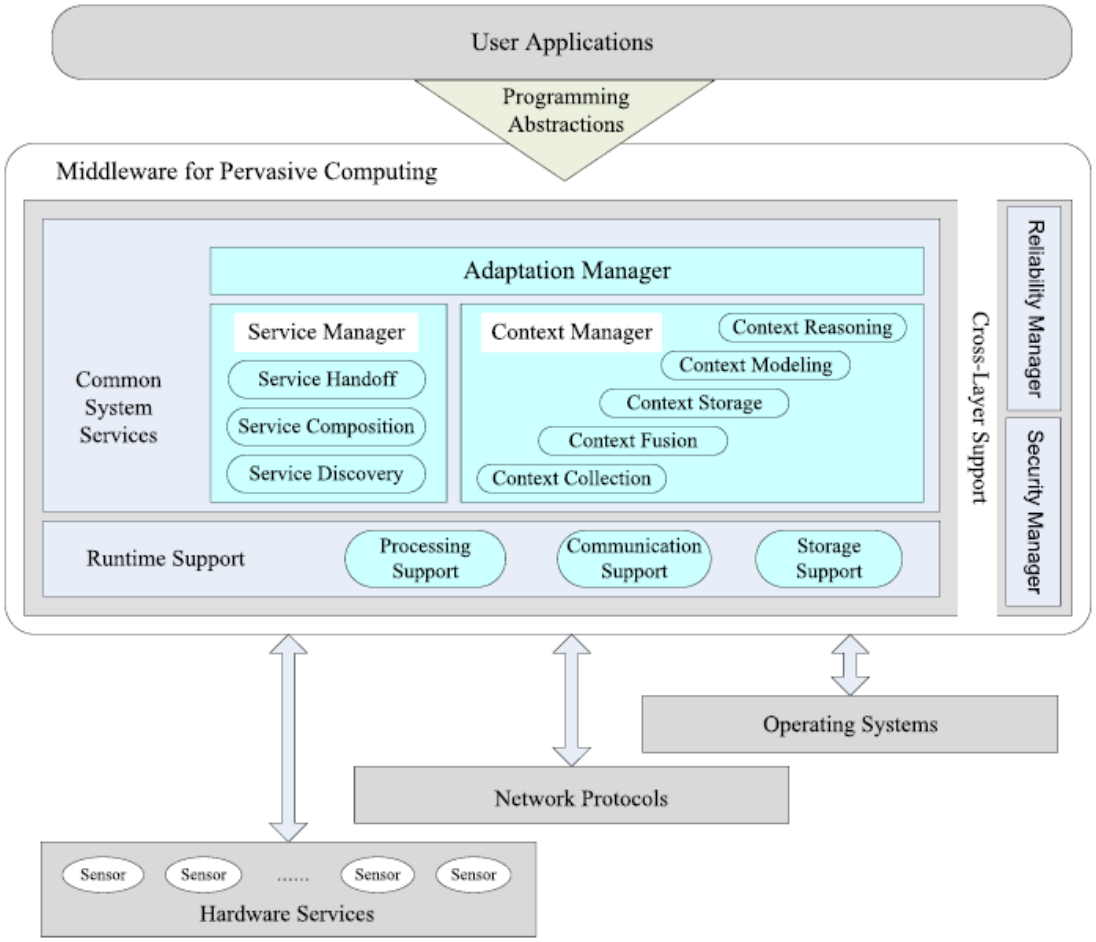


Figura 1.2: Modello di riferimento di un middleware per Pervasive Computing [3].

Il compito del middleware è quindi quello di astrarre aspetti architetturali sia fisici che logici dando la possibilità di comunicare con qualsiasi entità. Anche del caso di applicazioni aggregate quindi, è indispensabile tenere conto della necessità di fornire un supporto che permetta questo scambio.

1.3 Scala

Scala è stato scelto come linguaggio di riferimento per questa tesi rivelandosi molto utile per per comprendere appieno alcuni elementi dell' Aggregate computing tramite il framework Scafi [4].

La sua caratteristica principale è la grande scalabilità (da cui prende il nome, SCAlable LAnguage), ovvero la capacità di crescere in base al dominio applicativo che - come abbiamo visto - è un elemento fondamentale anche per un middleware.

Scala è un linguaggio di programmazione ad oggetti puro e anche i tipi primitivi come interi, double, long, ecc. diventano oggetti. Inoltre, combina questo paradigma con quello funzionale introducendone i costrutti. Di seguito ne vengono descritti alcuni tra i principali:

- Collezioni immutabili: oltre alle collezioni mutabili che troviamo nei classici linguaggi di programmazione, Scala introduce le *immutable collections*, ovvero costrutti impossibili da modificare. Infatti, ogni volta che si effettua un'operazione su una di esse il contenuto non viene cambiato ma viene creata una nuova collezione. Questo comporta un vantaggio importante per la concorrenza.

- Funzioni parziali: sono un tipo di espressioni in cui la funzione riceve argomenti solo per alcuni parametri, i quali potranno essere forniti in seguito.

- Pattern Matching: è un modo per valutare corrispondenze tra un valore e uno o più pattern. Anche oggetti o classi dichiarati utilizzando la parola chiave *case* possono essere utilizzati nel pattern matching.

- Currying: offre la possibilità di disassemblare una funzione, che prende in ingresso più argomenti, in una serie di funzioni che accettano un solo argomento.

- Impliciti: permettono di creare un parametro con valore predefinito in modo da non doverlo esplicitare più volte, oppure di effettuare conversioni di tipo in modo automatico.

L'integrazione delle funzionalità dei paradigmi di programmazione ad oggetti e funzionale, permette di trarre i vantaggi di entrambi. Inoltre si ha la possibilità, secondo esigenza, di creare domain specific languages (DSLs). Il codice Scala è anche completamente integrabile con quello Java e, conseguentemente, con tutte le librerie sviluppate per esso negli anni.

1.4 Scafi

Scafi è un framework scritto totalmente in Scala che fornisce un DSL per il field calculus e una macchina virtuale per l'esecuzione dei programmi scritti in questo linguaggio.

Vengono messi a disposizione del programmatore costrutti sintattici, una specifica semantica e il programma può essere combinato con il codice scritto in Scala.

La nascita di Scafi è dovuta all'esigenza di supportare il modello computazionale dell'Aggregate computing e alla necessità di eseguire programmi aggregati.

Scafi è inoltre fornito di un simulatore, utile per comprendere il funzionamento di un programma aggregato e visualizzare le interazioni tra device in un contesto di aggregate computing. Il limite del simulatore è che centralizza l'esecuzione di un sistema, che è altrimenti distribuita.

1.5 Modello ad Attori

Quello ad attori, è un modello definito *event-driven* composto da un insieme di attori che collaborano.

Un attore è un'entità computazionale che incapsula il proprio stato e il proprio comportamento senza comunicarli mai ad altri attori o ad altri componenti, se non tramite scambio volontario di messaggi che avviene in modo diretto e asincrono.

Alla ricezione di messaggi, che vengono inviati da altri attori, essi possono rispondere a loro volta. Inoltre possono inviare messaggi ad attori diversi, creare nuovi attori o ridefinire il comportamento che adotteranno all'arrivo di un nuovo messaggio. Questi oggetti infatti, sono detti *reattivi*, ovvero reagiscono all'arrivo di messaggi secondo un comportamento definito in precedenza talvolta modificando il loro stato interno.

Ogni attore possiede una coda (chiamata in questo contesto *mailbox*) in cui vengono inseriti i messaggi man mano che arrivano, i quali vengono processati uno alla volta su un thread separato quando disponibile.

Uno scheduler fa eseguire a turno ogni attore del sistema ed essi, appena prendono la parola elaborano il messaggio che si trova in testa alla coda (Figura 1.3).

Non condividendo mai il proprio stato mutevole o le proprie risorse interne, non hanno mai bisogno di eseguire azioni bloccanti prima di reagire ad un messaggio o mentre aspettano il verificarsi di un evento esterno.

Il modello ad attori predispone un ambiente dinamico in cui gli attori possono entrare o uscire in ogni momento cambiando il loro comportamento anche secondo necessità di dominio. Sistemi reattivi di questo tipo, non sono solo concorrenti e distribuiti, ma anche elastici, dinamici, reattivi e resilienti[5], caratteristiche che - come abbiamo visto nei capitoli precedenti - sono molto importanti per lo sviluppo di middleware per sistemi aggregati.

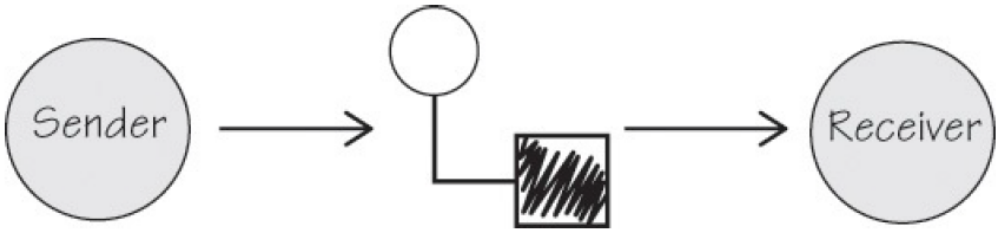


Figura 1.3: Sender manda un messaggio a Receiver che lo elabora su un thread libero [5].

1.5.1 Attori e limiti della programmazione object-oriented

Il modello ad attori sopperisce ad alcune limitazioni del tradizionale modello object-oriented. La prima risiede nel concetto di *incapsulamento* che consiste nel fatto che lo stato interno di ogni oggetto non possa essere modificato direttamente dall'esterno ma debba essere correttamente protetto rendendo pubblici solo i metodi strettamente necessari. Questo è esattamente quello che serve in un sistema distribuito.

Tuttavia, questo concetto non assicura nessun risultato prevedibile nel momento in cui due o più thread tentano di accedere alla stessa funzione. Solitamente in questo caso, si fa utilizzo di *lock*[1], ma questo comporterebbe altre complicazioni [6]:

- Aumento dei costi di esecuzione.
- Il thread è bloccato e non può svolgere altre azioni (non reattività) e l'utilizzo di altri thread che le svolgano è costoso a sua volta.

[1]Procedimento con il quale si impedisce che più thread concorrenti accedano in contemporanea alla stessa porzione di memoria.

- Si introduce la possibilità di incorrere in *deadlock*[2].
- *Lock* distribuiti incrementano la latenza e il sistema risulta poco scalabile.

Gli attori invece di invocare chiamate di metodo, si scambiano messaggi e li elaborano, senza causare il trasferimento dell'esecuzione e senza bloccare il thread che può continuare a computare o ricevere altri messaggi dall'esterno. A differenza dei metodi, i messaggi non hanno un valore di ritorno e il risultato della computazine viene, eventualmente, inviato attraverso un messaggio di risposta. Ogni attore elabora i messaggi sequenzialmente uno alla volta (Figura 1.4) mentre gli altri attori lavorano concorrentemente in modo che un sistema di attori possa elaborare contemporaneamente tanti messaggi quanti processori sono disponibili sulla macchina.

Non è quindi necessario utilizzare alcun *lock* poiché la modifica dello stato di un attore è possibile solo tramite lo scambio di messaggi e non è condiviso.

Gli attori all'interno di un unico sistema, interagiscono tra loro esattamente come accade nella comunicazione remota tra dispositivi diversi, in cui i messaggi possono essere paragonati ai pacchetti che viaggiano attraverso la rete mentre lo stato originale è mantenuto sulla macchina che può essere paragonata al singolo attore.

Un altro problema della programmazione tradizionale, incorre nel momento in cui si cerca di rendere un sistema reattivo: in alcuni casi, i thread possono "delegare" ad altri azioni da eseguire in background in modo da non rimanere bloccati. Questo però, comporta il fatto che il thread delegante non possa sapere quando l'esecuzione del task sia finita, ne tanto meno se si è verificata un'eccezione che può causare il malfunzionamento del sistema. Ancora una volta nel modello ad attori viene risolto questo aspetto e un evento di questo tipo viene segnalato con un messaggio al thread principale che può così decidere di riavviare l'esecuzione del task fallito.

[2]situazione in cui due o più processi o azioni si bloccano a vicenda, aspettando che uno esegua una certa azione (es. rilasciare il controllo su una risorsa come un file, una porta input/output ecc.) che serve all'altro e viceversa. `https://it.wikipedia.org/wiki/Deadlock`

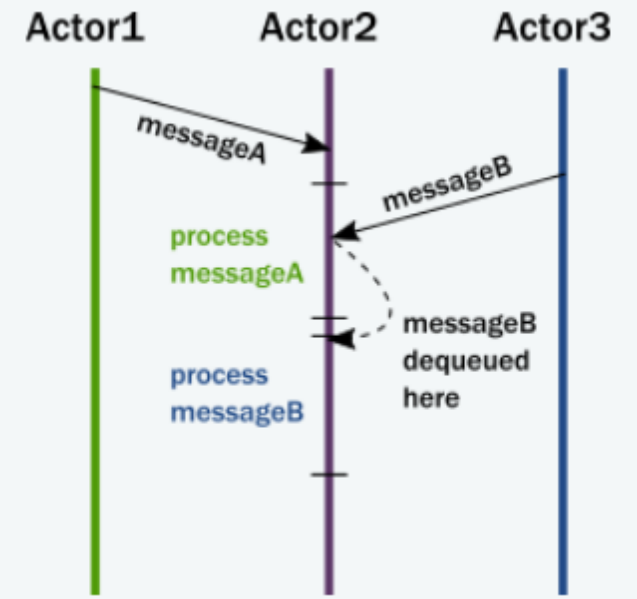


Figura 1.4: Gestione mailbox di un attore [6].

1.5.2 Estensione di Akka al modello ad attori

Akka[3] è un toolkit[4] sviluppato da Lightbend con licenza Open Source, per costruire applicazioni concorrenti e distribuite ad alto livello senza dover interagire direttamente con i thread. Oltre alle caratteristiche base di un sistema ad attori descritte in precedenza, come il fatto che ogni elemento computazionale abbia un comportamento definito che descrive la reazione ai messaggi, possegga una mailbox e agisca in un ambiente di esecuzione che aziona gli attori ciclicamente, Akka integra altri tre concetti fondamentali:

- *Location transparency:* Gli attori vengono creati dal sistema con un metodo factory che restituisce il riferimento all'istanza (ActorRef 1.5). Quest'ultima rappresenta l' indirizzo di un attore che viene poi trasmesso agli altri dal sistema sottostante che si occupa anche di consegnare i messaggi.

[3] `https://akka.io/`

[4] Insieme di classi correlate e riusabili, che compongono una o più librerie, progettate per fornire funzionalità di interesse generale per applicazioni ad attori [7]

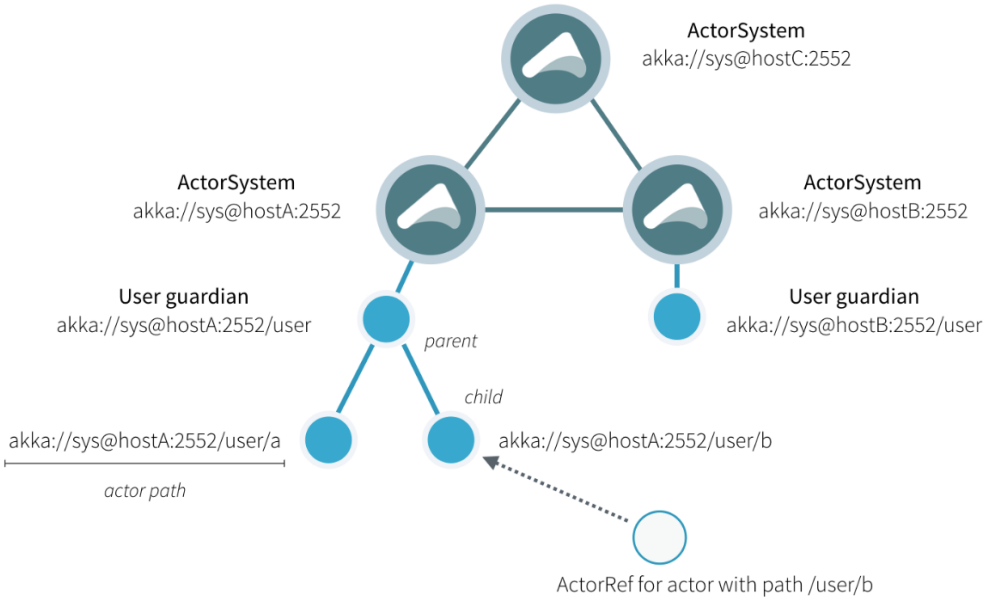


Figura 1.5: Relazione tra entità del sistema con rispettivi ActorRef [6].

Grazie a questo meccanismo, un attore non ha bisogno di sapere se l'attore a cui, o da cui, sta arrivando un messaggio si trovi sul sistema localmente, sia su un dispositivo remoto o si trovi su una JVM differente, tanto che il suo riferimento remoto risulta identico nella forma a quello di un attore locale e lo identifica univocamente all'interno dell'intera rete 1.1.

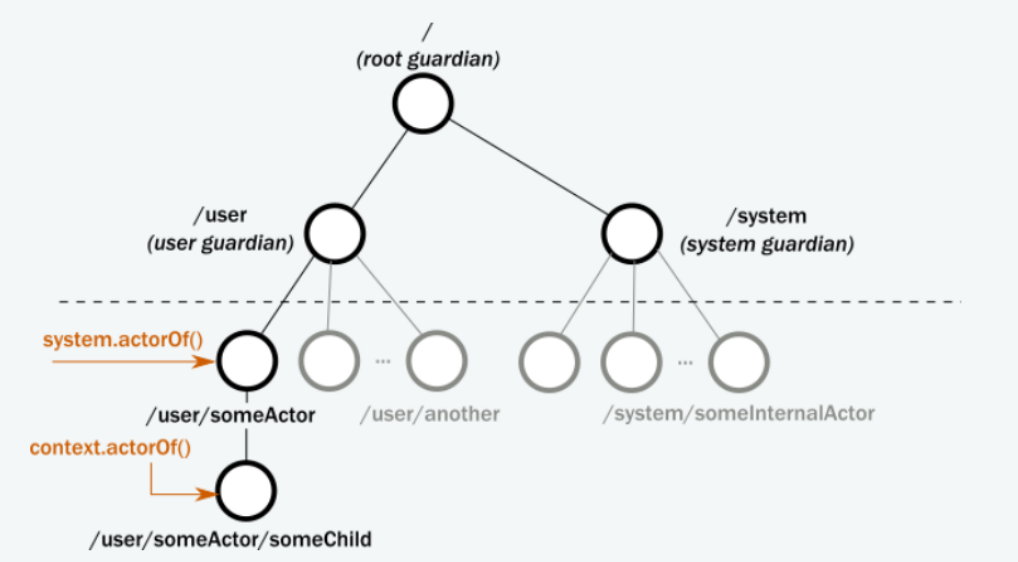
```
//riferimento di un attore locale
val local = context.actorSelection(
                    "akka://actorSystemName/user/localActorName")

//riferimento di un attore remoto
val remote = context.actorSelection(
                    "akka://actorSystemName@127.0.0.1:25520/user
                    /remoteActorName")
```

Listing 1.1: Stessa modalità per ottenere riferimenti di attori locali o remoti

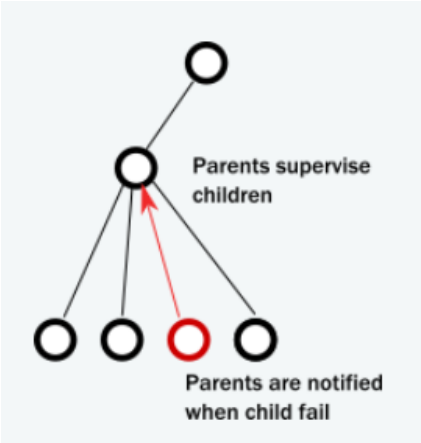
- *Serializzazione:* I messaggi mandati sulla stessa JVM vengono passati per riferimento, ma tra JVM diverse, devono essere serializzabili.[5] Akka usa la Java serialization di default ma altri tipi possono essere configurati.

- *Supervisione:* Akka fornisce l'intera infrastruttura per un sistema ad attori e gli elementi neccessari per controllarne il comportamento di base. Per fare questo, ogni volta che viene creato un attore, Akka lo inserisce all'interno di una gerarchia in cui sono già stati creati tre attori built-in: *root guardian*, *user guardian* e *system guardian* (Figura 1.6a). Il primo è il padre di tutti gli attori che vengono creati nel sistema ed è l'ultimo che si ferma quando il sistema viene terminato. L'attore *user* è padre di tutti gli attori creati dall'utente mentre *system* è l'attore in cima alla gerarchia definita dall'utente. In questo modo gli attori formano una relazione di dipendenza, un attore padre sorveglia i propri figli e può decidere se riavviarli in caso di errori oppure anche di farli terminare (Figura 1.6b).

[5] Gli oggetti devono essere convertiti in e da array di byte.



(a)



(b)

Figura 1.6: Sistema gerarchico di attori in Akka

Un aspetto importante di akka, è che, a differenza di quanto permesso dai thread, un sistema potrebbe presumibilmente ospitare milioni di attori, in quanto ogni istanza è costituita da poche centinaia di byte.

Akka mette a disposizione le sue funzionalità a tutti i linguaggi sulla JVM, uno dei quali, Scala, permette di usufruire di alcuni costrutti molto utili: `case object` e `case class` che, essendo costrutti immutabili, sono perfetti per la comunicazione tra thread

diversi e possono essere inviati sotto forma di messaggi. Inoltre, supportano il *pattern matching*, ampiamente utilizzato nella libreria ad attori.

I messaggi sono definibili la «API pubblica» [6] di un attore ed è consigliato dichiararli nel *companion object* di un classe con un nome legato al significato semantico specifico per il dominio che si sta considerando, in modo da permettter l'immediata comprensione del tipo di comportamento che ci si aspetta da quell'attore. Nel *companion object* si inserisce anche un metodo `props` con il quale si descrive la costruzione dell'attore. Il codice Scala in 1.2 riporta l'implementazione di un semplice attore `Ponger` che ogni qualvolta riceve un messaggio `Ping` stampa una stringa e risponde al mittente con un `Pong`.

Akka garantisce un sistema affidabile, alte prestazioni e tolleranza ai guasti (*fault tolerance*) oltre che facilitare l'implementazione di sistemi concorrenti, paralleli e distribuiti.

```
object Ponger{
  def props(id: String): Props = Props(new Ponger(id))
  final case object Ping
  final case object Pong
}

class Ponger(id: String) extends Actor{
  override def receive: Receive = {
    case Ping =>
      println("Hi")
      sender() ! Pong
  }
}

//creazione e avvio dell'attore
object Table extends App{
val system = ActorSystem("PingPongSystem")
val ponger = system.actorOf(Ponger.props(id = "p1"), name = "Pong")
ponger ! Ping
}
```

Listing 1.2: Esempio di un attore Akka scritto in Scala.

Capitolo 2

Requisiti e Analisi

L'obiettivo della seguente analisi, è quello di individuare gli elementi fondamentali che vadano a costituire un middleware per applicazioni aggregate.

Quest'ultimo deve permettere alle entità che compongono un sistema aggregato di accedere facilmente ai servizi e alla comunicazione.

I requisiti riportati sono frutto sia dell'analisi delle applicazioni aggregate, sono quindi basati sul modello logico dell'Aggregate Computing, che delle caratteristiche dei sistemi pervasivi in generale. Inoltre, è stato effettuato uno studio per quanto riguarda il deployment dell'applicazione, che ha permesso di far emergere alcune problematiche riguardanti la già citata "eterogeneità dei dispositivi".

2.1 Requisiti

Di seguito vengono categorizzati i requisiti del middleware, in modo da chiarire le principali proprietà che dovranno essere affrontate al momento dell'implementazione.

2.1.1 Requisiti funzionali

- *Collegamento iniziale*: il middleware deve essere in grado di rendere disponibili le diverse modalità di connessione: diretta, in cui ogni device comunica esplicitamente con gli altri, o indiretta, ovvero mediata da ambiente o da altri dispositivi fisici.

- *Gestione vicinato*: devono essere configurabili le relazioni di vicinato che governano il sistema aggregato. Il middleware deve sapere quali sono i criteri secondo i quali un nodo è vicino di un altro o meno.

- *Gestione topologia*: un altro aspetto configurabile deve essere il criterio secondo cui è possibile considerare un nodo ancora parte del sistema o meno.

- *Tipo di comunicazione*: ogni entità del sistema può comunicare "peer to peer" o tramite una architettura "client/server". Questo aspetto può essere statico o dinamico.

- *Inizio scambio di messaggi*: si deve individuare il momento opportuno nel quale si avvia la comunicazione.

- *Oggetto dei messaggi*: il middleware deve poter inviare tutto l'export o solo una parte, come ad esempio il solo output.

- *Salvataggio*: è necessario fare in modo che ogni dispositivo conservi solo l'ultimo o gli ultimi export arrivati dai vicini. Questo deve essere un aspetto configurabile.

- *Modalità di acquisizione informazioni*: ogni dispositivo può ricevere export dai vicini in modalità "push" o "pull". Nel primo caso il dispositivo chiede i dati, nel secondo gli vengono mandati ciclicamente.

- *Esecuzione su supporto*: il middleware deve gestire il fatto che l'applicazione aggregata possa essere eseguita sul device o sul server, per far fronte al problema dei dispositivi che hanno poca capacità di calcolo o, semplicemente, diventano ad un certo punto impossibilitati ad operare.

- *Riconoscimento del programma*: è necessario capire quale programma aggregato sia correntemente in esecuzione, quando metterlo in esecuzione, chi lo esegue (server o dispositivo) e sapere se è lo stesso che stanno eseguendo i vicini. Se un device chiede di diventare vicino ma non sta eseguendo il mio stesso programma aggregato, il middleware deve respingere la richiesta. Inoltre devo poterlo interrompere, metterlo in pausa o metterne in esecuzione un altro.

- *Esecuzione del programma*: Un device può eseguire più programmi aggregrati e si rende quindi necessario identificare questi ultimi con un "Id" per permettere di capire se tutti i partecipanti al sistema aggregato stiano eseguendo lo stesso programma e quale esso sia.

- *Frequenza di esecuzione*: il middleware deve essere configurabile anche per quanto riguarda la frequenza di esecuzione del programma aggregato. Essa può dipendere dall'accadimento di eventi quali il cambiamento dei vicini o dei valori dei sensori. La frequenza può essere modificata anche a run-time nel caso in cui il middleware si accorgesse che le dinamiche tra i componenti del programma sono cambiate. Si prevede dunque l'utilizzo di uno scheduler, interno o esterno al sistema.

2.1.2 Requisiti non funzionali

- Caratteristica fondamentale di un middleware è la scalabilità, per far fronte al potenziale incremento di dispositivi all'interno del sistema.

- Il middleware deve permettere la semplice implementazione di sistemi aggregati quindi deve essere facilmente utilizzabile.

- Sicurezza per quanto riguarda export e programma aggregato: non dovrebbero essere manipolabili.

2.1.3 Requisiti tecnologici

- **Akka**: l'intera struttura del progetto si basa sull'utilizzo del toolkit Akka per le carateristiche illustrate nella sezione 1.5.2 quali: comunicazione asincrona, totale incapsulamento del proprio stato interno, *location-transparency* e poca occupazione in memoria delle istanze.

- **Scala**: linguaggio usato per l'implementazione del middleware ad attori nella sola parte ad oggetti, eccetto per quanto riguarda alcuni costrutti fondamentali e utili di natura funzionale, come ad esempio il *pattern matching*, le *lambda expressions* e gli *stream*.

- **Gradle**[6]: sistema scelto per la *build automation* del progetto.

2.2 Analisi e modello del dominio

Il middleware in analisi, deve consentire la creazioni di applicazioni aggregate mettendo a disposizione quattro servizi fondamentali:

1. Configurazione della comunicazione.
2. Connessione e topologia.
3. Acquisizione ed elaborazione dei dati.
4. Esecuzione del programma aggregato.

Queste quattro entità possono essere implementate con una architettura che prevede relazioni *dirette* o *indirette*, ed è possibile scegliere in base alle caratteristiche del sistema che si intende programmare.

Le implementazioni possono differire all'interno dello stesso sistema aggregato ed il middleware è in grado di gestire dinamicamente entrambe le modalità, che possono variare in real-time. Il middleware deve, quindi, interporsi tra applicazioni e hardware raccogliendo i dati utili (*export* e informazioni dei sensori) dagli altri dispositivi (i vicini) in modo da poter eseguire il programma aggregato e comunicare a chi lo richiede il risultato della computazione.

Un' astrazione del modello viene mostrata in figura 2.1 e mostra generalmente come si comporta il middleware in un contesto aggregato.

[6] https://gradle.org/

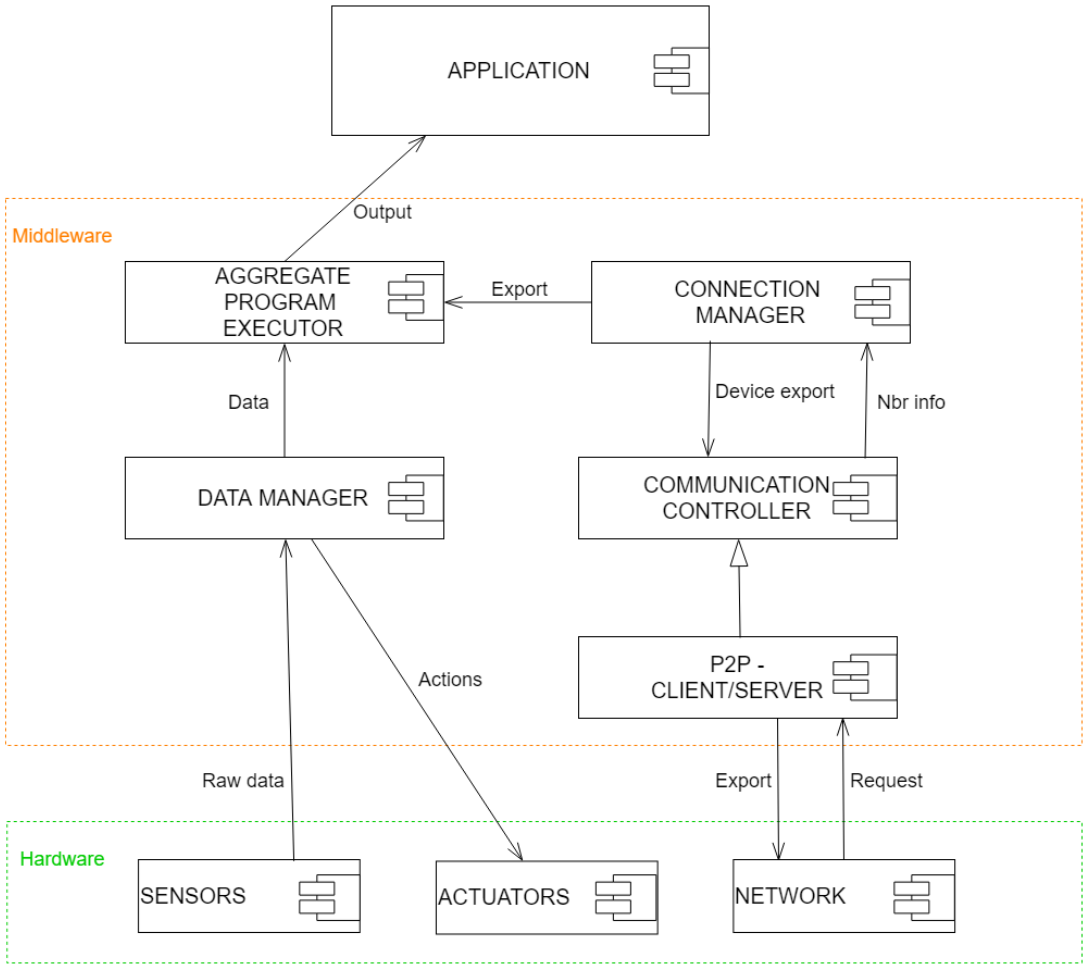


Figura 2.1: Il ruolo del middleware per la costruzione di sistemi di Aggregate computing

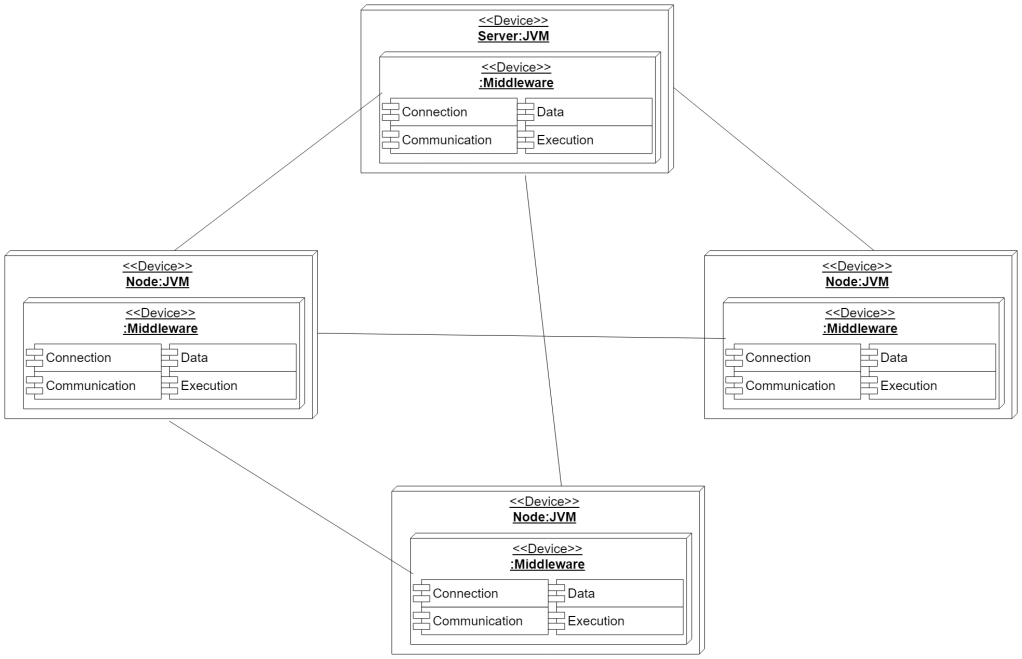
Di seguito vengono descritti, in modo approfondito, i servizi sopraelencati, fornendo una esplicazione delle caratteristiche di ognuno.

2.2.1 Configurazione della comunicazione

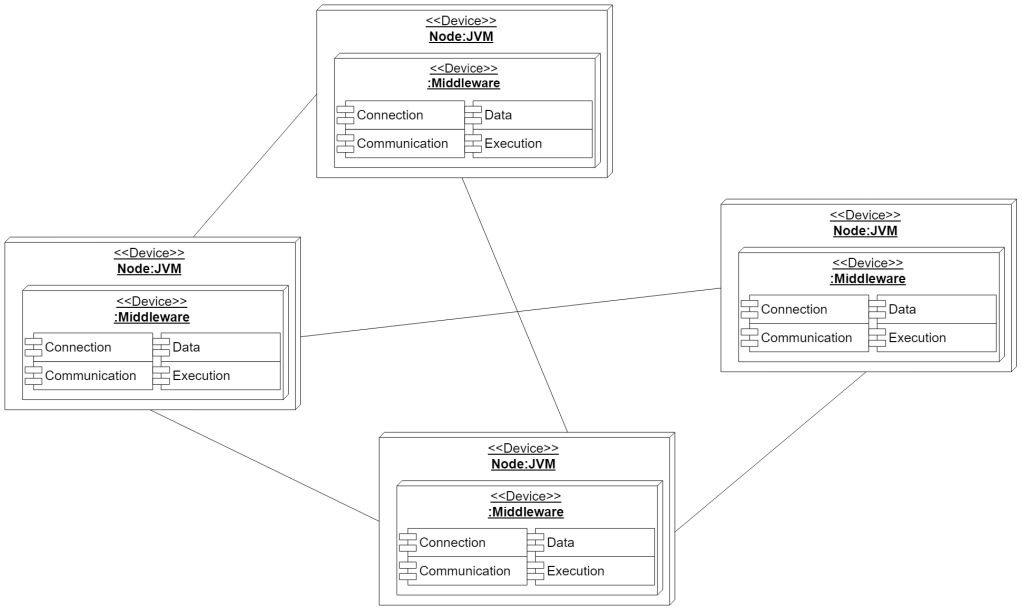
Vista la necessità di rendere la partecipazione ad un sistema aggregato accessibile a qualsiasi dispositivo, il middleware deve predisporre due tipi di comunicazione: diretta e indiretta. Per quanto riguarda la comunicazione diretta, il middleware è presente su ogni dispositivo completo di tutti i servizi e permette di comunicare con gli altri nodi senza la mediazione di altri supporti. È, quindi, il dispositivo a mettere in atto la comunicazione con tutti i vicini.

Differente è invece la comunicazione indiretta, in cui alcuni servizi possono trovarsi su un dispositivo diverso da quello principale e comunicare per vie traverse. Ad esempio il dispositivo invierà i dati ad un server che si occuperà di elaborarli e instradare le informazioni ai vicini conosciuti.

Ogni dispositivo specifica il tipo di comunicazione che vuole attuare in modo tale da permettere al middleware di effettuare le opportune configurazioni. Le due modalità di comunicazione sono rappresentate in figura 2.2.



(a) Modalità indiretta



(b) Modalità diretta

Figura 2.2: Due tipi di comunicazione tra dispositivi

2.2.2 Connessione e topologia

In un sistema aggregato è necessario gestire la mappatura dei device, in modo da permettere ad ogni dispositivo di entrare a far parte del sistema (o uscirne) in ogni momento e tenere traccia di quali e quanti ne fanno parte in qualsiasi momento, oltre che poterli escludere quando necessario.

Il middleware deve determinare i vicini di ogni nodo a seconda delle relazioni di vicinato che possono essere sia precedentemente specificate, che cambiare in corso di esecuzione del programma. Un dispositivo può essere ritenuto "vicino" di un altro secondo qualsiasi configurazione logica (ad esempio in base ad una relazione *location-based*) e le modalità di conservazione dell'informazione variano in base al tipo di architettura, diretta o indiretta: nel primo caso il device vede solo i propri vicini, non esiste una topologia completa, quindi decide autonomamente se un altro device possa essere ritenuto suo vicino o meno. Nel secondo caso, invece, il server possiede tutta la mappatura del sistema ed è quest'ultimo a decidere e ad assegnare i vicini di ogni dispositivo. Inoltre, è necessario capire in che modo un dispositivo entra a far parte di un sistema, evitando possibilmente di centralizzare la registrazione dei device ad un'unica entità.

2.2.3 Acquisizione ed elaborazione dei dati

Ogni device può possedere sensori logici e/o fisici. Sensori logici importanti sono le mappe che contengono le relazioni di vicinato (i sensori ambientali). Sensori fisici invece (sensori locali), possono essere segnali GPS, sensori di movimento, temperatura ecc. Questi dispositivi, sono fondamentali per costruire il contesto in cui opera un programma aggregato. L'entità che gestisce i dati, in base a ciò che riceve, si occupa anche di azionare gli attuatori. Le azioni che conseguono sono contenute nell'output insieme ai messaggi per i vicini.

Anche gli attuatori possono essere fisici o logici. I primi agiscono sull'ambiente, ad esempio l'azionamento di un motorino o l'accensione di una luce, i secondi invece, comprendono azioni da eseguire sul device, come la modifica su una interfaccia grafica o interventi a livello di middleware: restrizione del raggio di azione, aumento o diminuzione della frequenza con cui invio messaggi o eseguo il programma aggregato perché i dati cambiano più o meno velocemente, o modifiche sulla modalità di comunicazione

2.2.4 Esecuzione del programma aggregato

Il servizio si occupa di eseguire il programma aggregato in base al contesto che gli viene fornito oltre che capire quale programma è in esecuzione e se è lo stesso che stanno eseguendo gli altri.

Nel modello a comunicazione indiretta, posso avere un server a cui un dispositivo manda la richiesta di esecuzione del programma contro i suoi dati e il suddetto si occupa di accertarsi che sia lo stesso in esecuzione nel sistema. Nel modello diretto, ogni volta che un vicino tenta di connettersi, il device controlla che stia eseguendo lo stesso programma, altrimenti viene rifiutato.

Ogni nodo può effettuare un controllo per capire su quale supporto far computare il programma (se sul device stesso o sul server) agendo opportunisticamente in base alle esigenze. La scelta viene effettuata dinamicamente (in real-time) oppure staticamente (so già che un dispositivo non computerà mai). Il middleware può operare questa funzione anche via attuatore.

Capitolo 3

Design

Nell'ambito della tesi è stata realizzata la parte "peer to peer" del middleware, con comunicazione diretta tra i nodi.

Le strategie utilizzate per realizzare il progetto fanno riferimento al modello ad attori e posseggono le caratteristiche di un middleware event-based, in cui gli eventi sono rappresentati da messaggi.

Ogni servizio evidenziato in fase di analisi, viene concretizzato in uno o più attori che collaborano per il funzionamento del middleware a supporto del sistema aggregato.

3.1 Architettura a comunicazione diretta

Nell' architettura ad attori a comunicazione diretta (modello "peer to peer"), progettata per il sistema, ogni nodo possiede tutti i servizi del middleware e tutta l'esecuzione avviene sul dispositivo stesso. Gli attori in questo caso si trovano tutti su un unico Actor System e su un'unica JVM quindi la comunicazione è resa possibile grazie al solo passaggio per riferimento. Ogni attore definisce il tipo di messaggi che può ricevere e alla ricezione di uno di essi agisce conseguentemente.

L'architettura del sistema è rappresentata in figura 3.2 e si può far riferimento a figura 2.2b per quanto riguarda il deployment che verrebbe effettuato con questo tipo di architettura.

È possibile mostrare la composizione del progetto anche da un punto di vista dei

processi[7], in modo da mostrare un esempio di esecuzione asincrona del sistema (Figura 3.1).

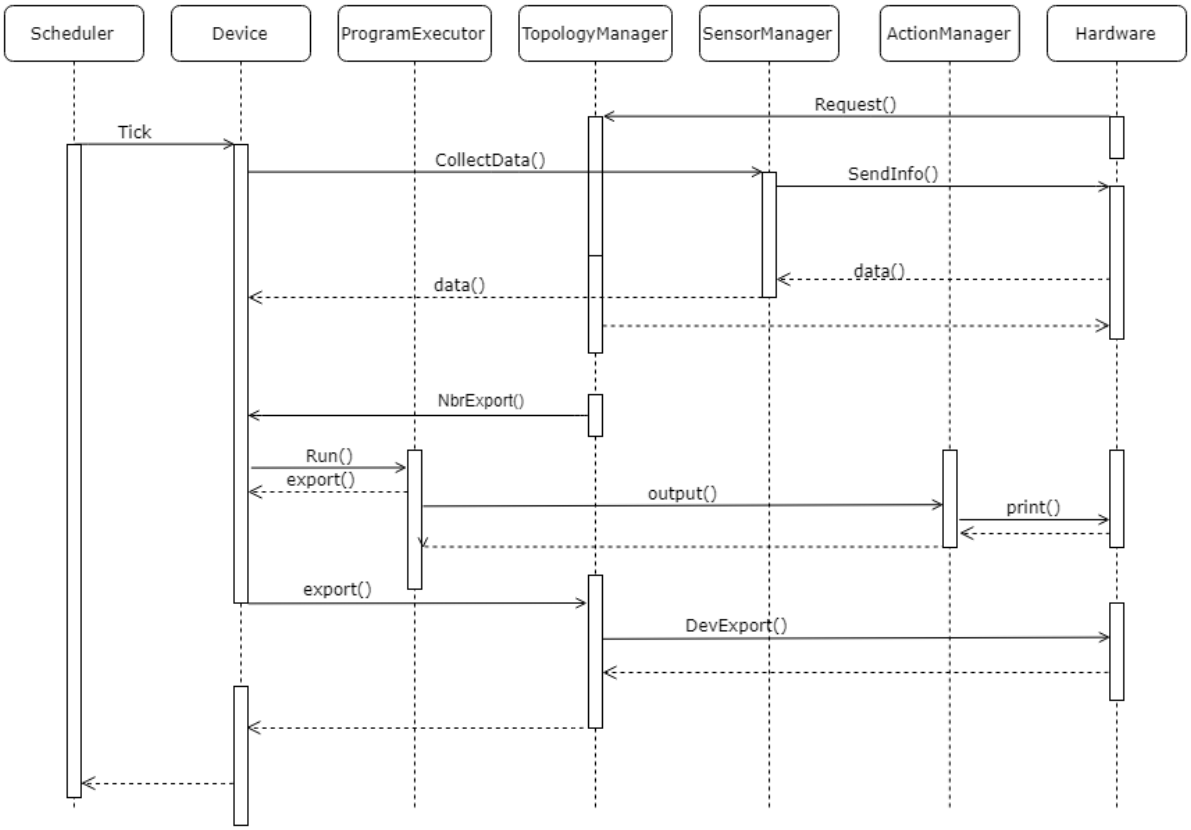


Figura 3.1: Diagramma di sequenza che mostra le principali interazioni tra attori

3.1.1 Attori del sistema

All'avvio dell'applicazione, per ogni dispositivo che intende partecipare al sistema aggregato, vengono creati 8 attori più all'occorrenza uno o più attori Sensor. Di seguito viene descritto come i componenti del middleware interagiscono tra loro:

- `Scheduler:` ha come unico compito quello di inviare periodicamente (ogni 500 millisecondi di default) il messaggio `Tick` all'attore `Device Computation`. Può ricevere un messaggio da `Actuator` con la nuova quantità di tempo con cui modificare la frequenza di invio del messaggio.

[7] `https://en.wikipedia.org/wiki/4%2B1_architectural_view_model`

- `Logics Controller`: viene configurato in base al tipo di logica di vicinato da utilizzare e alle altre caratteristiche che il sistema deve avere. Invia poi queste informazioni agli attori `Device Computation`, `Topology Manager` e `Action Manager` che operano in base al tipo di dato ricevuto. Può ricevere un messaggio da `Actuator` contenente una nuova logica che poi deve nuovamente propagare.

- `Device Computation`: attore che rappresenta il dispositivo. Si registra al `Topology Manager` e ne elabora i messaggi contenenti gli *export* dei vicini. Inoltre, raccogliere i dati relativi ai sensori che vengono mandati dal `Sensor Manager`.

  Una volta ottenute tutte le informazioni necessarie crea il contesto che viene inviato all' `Aggregate Program Engine`. Quando possiede l'export relativo al *round* appena eseguito, lo invia al `Topology Manager`.

- `Aggregate Program Engine`: entità che esegue il programma aggregato secondo configurazione preventiva. Invia l'export al `Device Computation` e l'output (contenuto nell'export) all'`Action Manager`.

- `Action Manager`: in base al tipo di logica fornita ancora una volta dal `Logics Controller`, l'`Action Manager` stabilisce l'azione da compiere con l'output del programma, lo elabora secondo necessità e informa `Actuator`.

- `Actuator`: in base al tipo di azione contenuta nel messaggio ricevuto mette in pratica azioni logiche e/o fisiche come ad esempio informare lo `Scheduler` che deve modificare la frequenza con cui manda il tick, avvisare il `Logics Controller` che deve cambiare le logiche con cui vengono determinati i vicini o segnalare a componenti hardware o software di eseguire certe azioni come accendere una luce nel primo caso e stampare a video una stringa nel secondo.

- `Sensors Manager`: questo attore ha il compito di raccogliere tutti i dati dei sensori del dispositivo, sia quelli locali, che derivano da sensori fisici o logici, che quelli ambientali. Un messaggio contenente queste informazioni viene mandato al `Device Computation`.

- `Sensor`: per ogni tipologia di sensori viene istanziato un attore `Sensor`, che si occupa di immagazzinare ed elaborare i dati ricevuti per poi instradarli al `Sensor`

`Manager`. Può ricevere messaggi da dispositivi fisici e logici, locali o ambientali. Un particolare riferimento di `Sensor`, chiamato *NbrRange* e di tipo ambientale, viene creato da `Topology Manager` ed è utile per l'esecuzione di alcuni programmi aggregati. A questo attore vengono inviati, sempre da `Topology Manager`, i valori delle distanze corrispondenti ad ogni vicino.

- `Topology Manager:` il `Topology Manager` si occupa di ricevere ed elaborare messaggi in arrivo e per, altri dispositivi. Essi sono:

  1. Richieste di ottenere lo status di vicino. In base alla logica di vicinato corrente, se l'esito è positivo il riferimento al dispositivo esterno viene aggiunto nell'elenco dei vicini.
  2. Una volta registrato, il device esterno può mandare i suoi *export*. A sua volta, il `Topology Manager` manda a tutti i vicini gli *export* del device locale. Dai vicini ottiene anche il valore da attribuire al *NbrRange*.

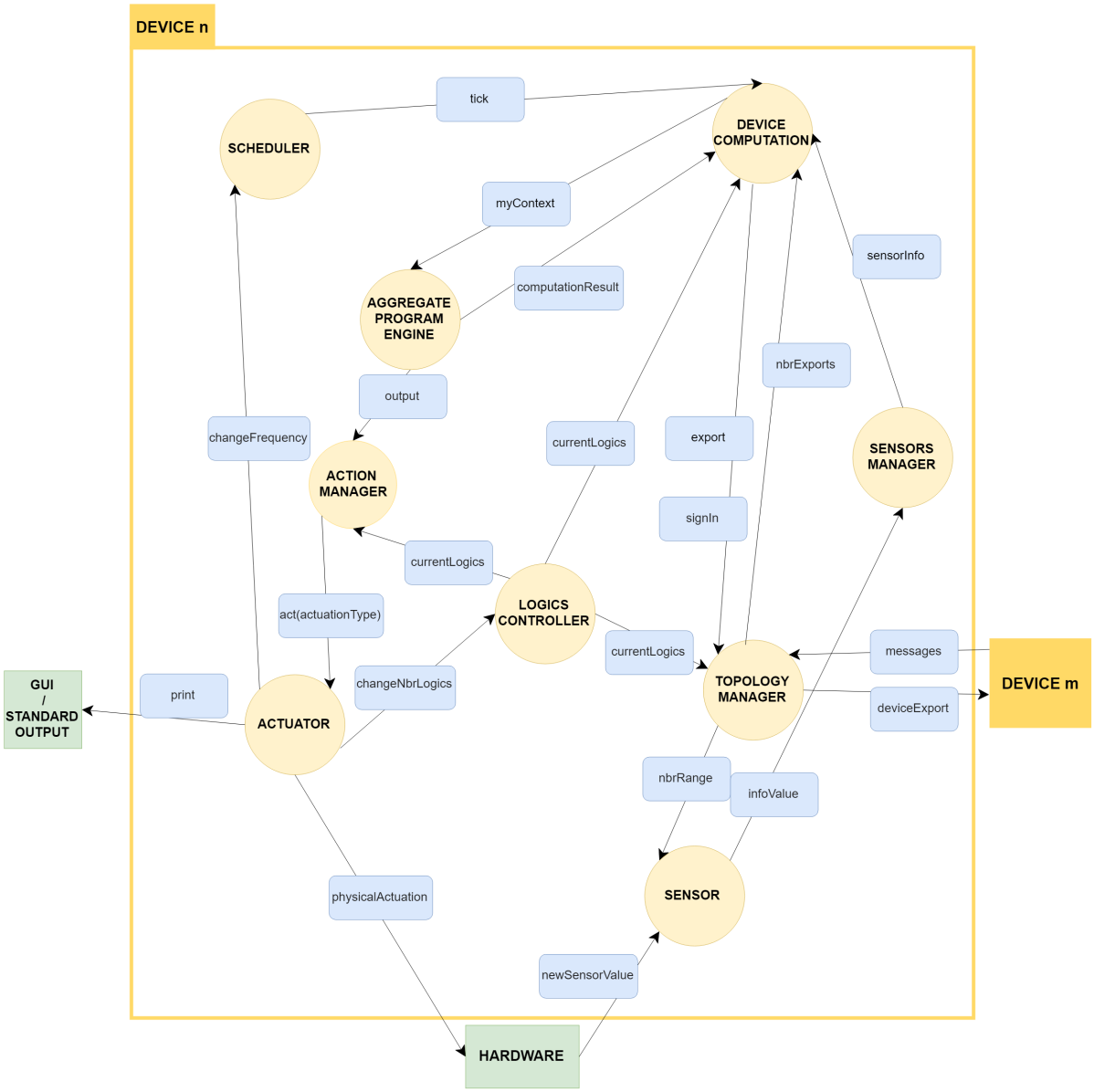


Figura 3.2: Architettura del middleware modello p2p

3.2 Design dettagliato

Di seguito si procede ad approfondire alcuni elementi di progettazione del middleware e vengono proposte le soluzione adottate per alcune sottoparti rilevanti del progetto.

3.2.1 Progettazione di Sensor

Un attore `Sensor` viene istanziato per ogni tipo di sensore presente sul dispositivo e per questo deve essere facilmente adattabile alle esigenze. Nel metodo `receive` di `Sensor`, si è fatto uso del pattern Strategy che consente di differenziare le azioni da eseguire in base al tipo di sensore istanziato: locale o ambientale. Grazie alla parola chiave di Scala *type*, Scafi differenzia due tipi stringhe: `LSNS` per i sensori locali e `NSNS` per quelli ambientali. Anche il middleware evidenzia questa separazione utilizzando la classe generica `ST` da cui ereditano `LocalSensor`, che effettua il binding con `LSNS` e `EnvironmentalSensor`, che fa la stessa operazione con `NSNS`.

(Figura 3.3).

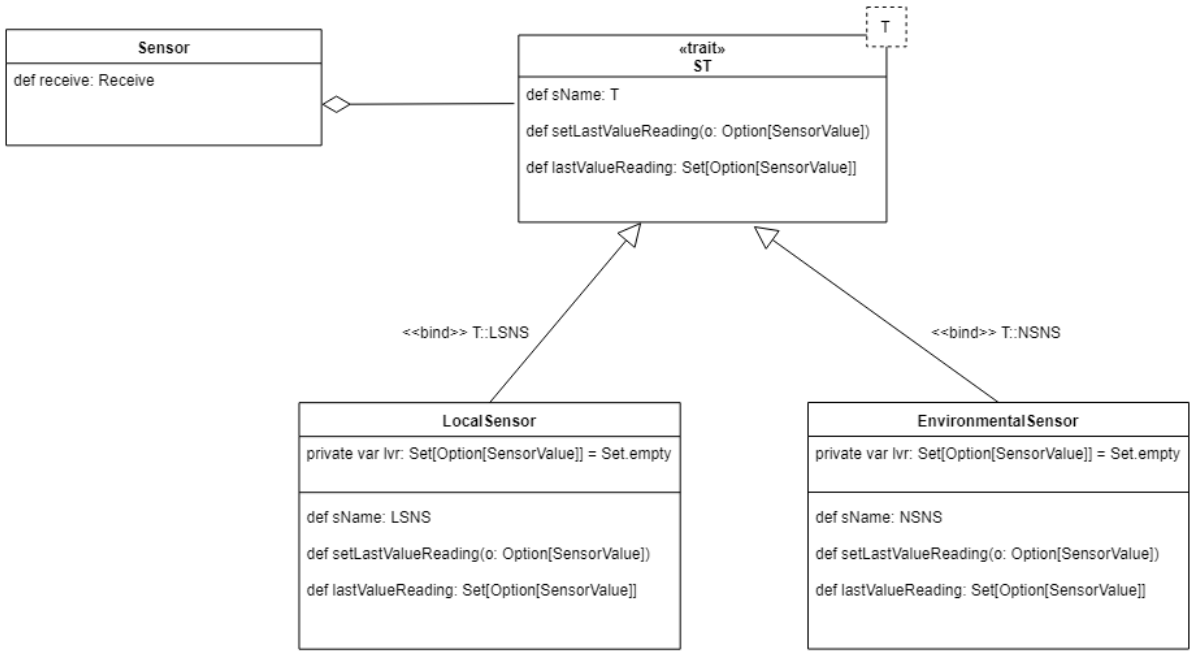


Figura 3.3: Pattern Strategy utilizzato all'interno di Sensor

3.2.2 Progettazione Topology Manager

Anche per quanto riguarda l'attore `Topology Manager` è stato utile utilizzare il pattern Strategy.

Infatti, a seconda della logica di vicinato corrente, il criterio di determinazione dei vicini cambia in modo sostanziale. Quando un potenziale vicino invia la richiesta, il metodo `isANeighbour` effettua il controllo e restituisce una tupla contenente un Boolean e opzionalmente un valore. In figura 3.4 la rappresentazione di tale pattern. Allo scopo di testare il funzionamento del sistema, sono state utilizzate due logiche di vicinato: spaziale e di "colore". La prima assume un dispositivo come vicino se si trova ad una certa distanza, la seconda attribuisce la caratteristica di vicinato ai device il quale colore (semplice enumerazione, costituita da nomi di colori, che viene utilizzata come metro di paragone) è lo stesso del nodo su cui si sta effettuando la valutazione.

Figura 3.4: Pattern Strategy utilizzato all'interno di TopologyManager

3.2.3 Attivazione servizi

Visti il numero di attori e la complessità di inizializzazione delle sottoparti, per l'avvio dell'applicazione è stato utilizzato il pattern Façade.

Questo schema permette di fornire un'unica interfaccia di comunicazione tra i sottosistemi del middleware facilitandone così l'utilizzo.

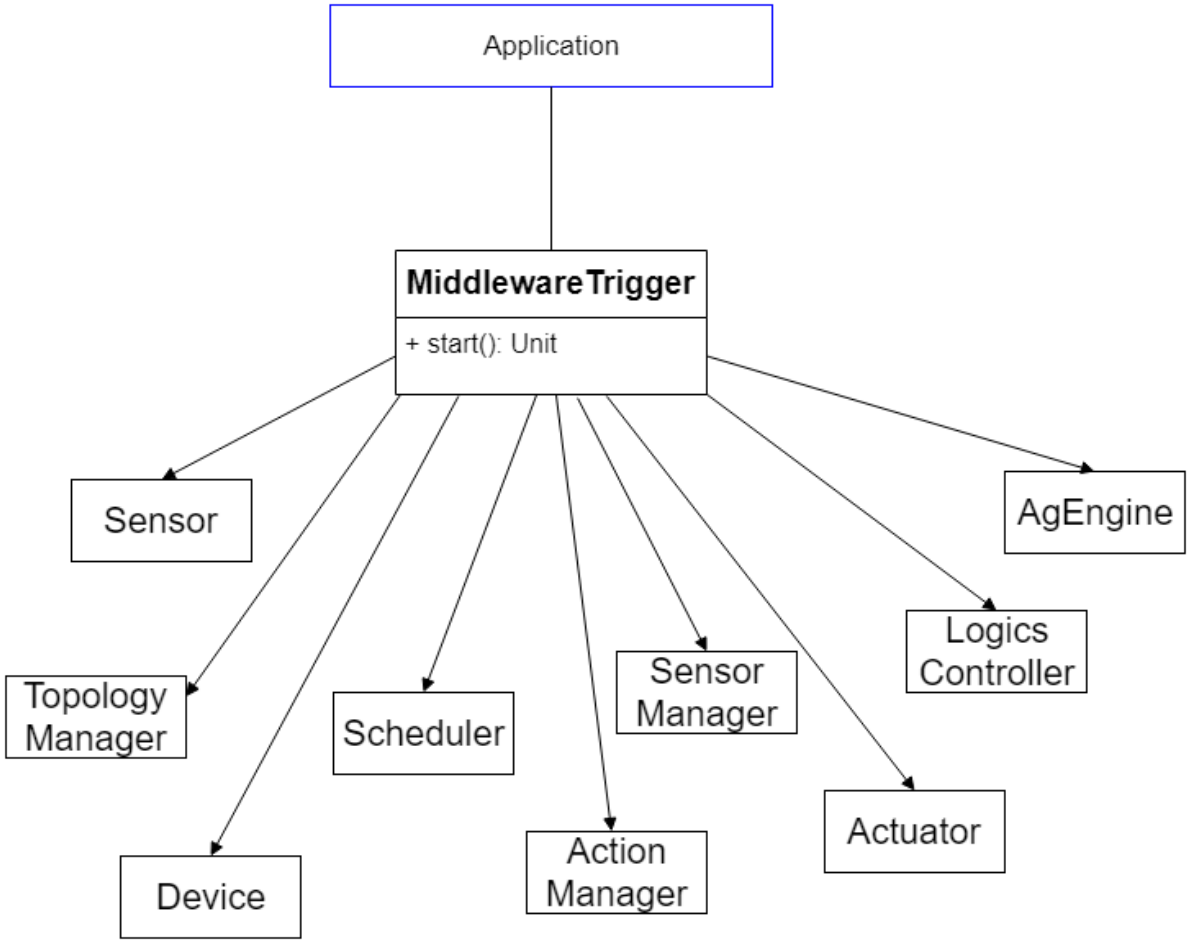


Figura 3.5: Pattern Façade

Capitolo 4

Implementazione

In questo capitolo vengono illustrate in modo approfondito alcune scelte fatte durante lo sviluppo del sistema. Questo renderà più comprensibili alcuni dettagli descritti in fase di progettazione.

Il codice riportato non è esaustivo ma è volto ad esplicitare alcuni passaggi fondamentali dell'implementazione del middleware.

4.1 Avvio applicazione

Fondamentale prima dell'inizio della computazione, è la configurazione delle logiche di sistema.

```
//frequenza iniziale dello scheduler
  private val INITFREQUENCY = 500

  //creazione actor system
  val system = ActorSystem("AggregateMiddleware")

  //inizializzazione programma aggregato
  val aggregateProgram = /*....*/

  //inizializzazione logica di vicinato
  val nbrLogics = /*....*/

  //inizializzazione logica che determina il valore per effettuare il
     confronto tra vicini
```

```
  val currentLogics = /*....*/
```

Al momento dell'avvio, la classe `ApplicationLauncher`, esegue il metodo statico `MiddlewareTrigger.start()` che contiene la configurazione degli attori. Gli 8 attori base necessari al funzionamento sono :

- Scheduler
- Sensor Manager
- Topology Manager
- Actuator
- Action Manager
- AgEngine
- Device
- Logics Controller

Viene poi creato un attore per ogni sensore da rappresentare.

È sufficiente specificarne tipo e nome e possono essere, ad esempio, un sensore che raccolga i dati inviati da un Gps, oppure che conservi un dato logico come il colore.

```
//attore che rappresenta sensore fisico Gps
 val localGps = system.actorOf(Sensor.props("1", LocalSensor("Gps")),
 "LocalGps")

//attore che rappresenta sensore logico Color
  val localColor = system.actorOf(Sensor.props("1", LocalSensor("Color")),
      "LocalColor")
```

Il TopologyManager crea anche il sensore ambientale `NbrRange`. Così facendo, secondo la gerarchia di Akka, ne diventa genitore e può monitorarlo.

Il primo messaggio che riceve `NbrRange` è il valore 0.0 in corrispondenza dell' id del nodo, che rappresenta la distanza del device da se stesso.

```
private var nsnsSensor = context.actorOf(Sensor.props("NS",
    EnvironmentalSensor("nbrRange")), "NbrRange"+actorId)
```

A questo punto, lo scheduler riceve il riferimento al device a cui mandare periodicamente il `Tick` e viene istruito il `Logics Controller` ad inviare, agli attori che ne necessitano, le logiche del programma.

L'attore `Device` si registra al `TopologyManager` in modo da permettere a quest'ultimo di utilizzarne il riferimento.

```
localScheduler ! localDevice
localLogContr ! SendProgramLogics()
localLogContr ! SendProgramStrategy()

//il device si registra al topologymanager
localDevice ! ConnectTo(localTopMan)
```

A fini dimostrativi della corretta configurazione del programma, si riporta il modo in cui viene simulato il sensore fisico Gps: viene manualmente inviato all'attore che lo rappresenta il dato contenente le coordinate.

Infine, il sensore si registra al `Sensor Manager` che lo inserirà tra i sensori locali del dispositivo.

```
//all'attore sensor arriva un nuovo valore dal sensore fisico
localGps ! NewValue(0, Option(GpsCoordinates(44.0701799,12.5649016)))

//il sensore si registra al sensor manager
localSensorMan ! SensorRef(localGps)
```

4.2 Scheduler

Oltre allo scheduler interno di ogni Actor System, Akka fornisce la possibilità di pianificare l'invio di messaggi agli attori.

È necessario un oggetto `ExecutionContext` implicito per lanciare in background attività come `system.scheduler.scheduleWithFixedDelay()`.

Specificando `Duration.Zero` nel metodo sopracitato, lo scheduling parte al momento della creazione. Si devono specificare anche frequenza, riferimento all'attore a cui deve inviare il messaggio e il tipo di quest'ultimo.

```
implicit val executionContext = context.dispatcher
val cancellable = context.system.scheduler.scheduleWithFixedDelay(Duration
    .Zero, currentFrequency.milliseconds, rf, Tick)
```

4.3 Rilevamento vicini

Come visto in precedenza, il `TopologyManager` controlla se i dispositivi che ne fanno richiesta, sono definibili vicini. A questo attore, viene periodicamente inviato il valore del device che deve essere confrontato quello degli altri, come ad esempio il valore della posizione in cui esso si trova. In seguito, secondo la strategia di determinazione del vicinato attualmente in atto, vengono effettuati i dovuti controlli.

Se un device viene ritenuto valido, il `TopologyManager` invia al sensore ambientale i dati utili: ad esempio nel caso si stia utilizzando `NbrRange`, `NewValue` sarà composto dall'id del vicino e dalla sua distanza con il nodo.

```
/*..
..*/
!currentCompareValue.isEmpty && !potentialNbrs.isEmpty match {
    case true =>
      potentialNbrs.foreach(x =>
      currentStrategy.get
      .isANeighbour(currentCompareValue.get(currentLogics).get,
      x.get.value) match {

       case (z: Boolean, y: Option[SensorValue]) if z =>
          nsnsSensor ! NewValue(x.get.id, y)
          devNeighbour ++= Map(x.get.id -> Map(x.get.ar -> Option.empty))
          potentialNbrs = potentialNbrs.filter(h => h.get.equals(x.get))

       case (z: Boolean, _) if !z => log.warning("{} isn't a nbr", x.get.
          id)
      })
```

```
/*..
..*/
```

Una volta registrati, i vicini possono iniziare ad inviare i propri *export*, il `TopologyManager` li intercetta e li invia all'attore `Device Computation`.

```
override def receive: Receive = {
/*..
..*/
    case NbrExport(id, exp) =>
            devNeighbour.contains(id) match {
        case true =>
        /*..

          ..*/
        nbrExports ++= Map(id -> exp.get)
        device.get ! ExportsFromTopMan(nbrExports)
        case false =>
          /*..

          ..*/
      }
/*..
..*/
}
```

Una volta che è stato elaborato il programma aggregato, il `Device Computation` manda il proprio *export* al `TopologyManager` che lo può così instradare ai vicini inviando ad ognuno di essi il messaggio: `NbrExport(deviceId, deviceExport)`.

```
override def receive: Receive = {
/*..

..*/
  case DeviceExport(devId, de) =>
         devNeighbour.foreach(x =>
            x._2.foreach(y =>
            y._1 ! NbrExport(devId, de)
            ))
```

```
 /*..

..*/
}
```

4.4 Contesto ed esecuzione

All'interno del progetto sono stati utilizzati alcuni costrutti che Scafi mette a disposizione, utili all'esecuzione di un programma aggregato. Essi sono:

```
ID
EXPORT
CONTEXT
LSNS
NSNS
```

In particolare, all'interno dell'attore `Device Computation`, per creare il contesto è stato utilizzato il metodo di Scafi `factory.context()` chiamato ogni volta che il middleware entra in possesso di tutte le informazione necessarie.

```
  def createContext(exp: Map[ID, EXPORT], ls: Map[LSNS, Any], ns: Map[NSNS
      , Map[ID, Any]]): CONTEXT = {
    factory.context(
    selfId = actorId,
    exports = exp,
    lsens = ls,
    nbsens = ns
    )
  }
```

A questo punto all'attore che si occupa dell'effettiva esecuzione del programma aggregato, viene inviato il messaggio contentente il contesto appena creato.

```
  myEngine ! Compute(createContext(exports, lsnsSens, nsnsSens))
```

All'interno di `AgEngine`, viene eseguito il *round* e dal risultato dell'esecuzione, ovvero l'*export*, viene estratto l'*output*.

Il primo viene inviato al `Device`, mentre il secondo viene spedito ad `Action Manager`, che decide come elaborarlo.

```
override def receive: Receive = {
    case Compute(cx) =>
        val res = agp.round(cx)
        sender() ! MyExp(res)
        am ! OutputMessage(res.root())
}
```

Capitolo 5

Testing

La fase di testing è stata utile per verificare sia le singole parti del sistema, facendo uso di *Unit Test*, che le interazioni tra i componenti, appoggiandosi a framework capaci di supportare applicazioni concorrenti.

Trattandosi di un sistema ad attori infatti, oltre a controllare le funzioni di base, è fondamentale capire se i messaggi vengono inviati e ricevuti correttamente e se il comportamento dell'attore rispecchia le previsioni.

5.1 Strumenti di testing

Per il testing automatico e asincrono del sistema, Akka fornisce un modulo dedicato chiamato *ActorTestKit*. Questa libreria è stata utilizzata all'interno del progetto in quanto permette di testare gli attori a diversi livelli e in un ambiente controllato.

Per predisporre un test con questo Toolkit, è necessario creare un sistema ad attori estendendo la classe `TestKit(ActorSystem("SystemName")`.

Si è fatto ampio utilizzo della classe `TestProbe` che permette di creare un attore "fittizio" in grado di intercettare ed inviare messaggi, garantendo un controllo accurato dello scambio degli stessi.

Per fare questo, è possibile utilizzare alcune asserzioni *built-in* come `expectNoMessages()`, `expectMsgType[]` ecc.

Durante lo sviluppo del progetto ci si è avvalsi anche di un tool di valutazione basato su Scala: *ScalaTest*. Questo strumento è progettato, come il linguaggio in cui è scritto,

in modo da crescere in base alla complessità di un sistema: più essa aumenta, più è possibile estendere le soluzioni di testing.

Tra le possibilità fornite dalla suddetta libreria, è stata sfruttata la classe `AnyWordSpecLike`, che permette di predisporre i test in simil-linguaggio naturale. Ad esempio, per verificare che un attore risponda correttamente ad una richiesta di invio dati, si può scrivere:

```
"A DataCollector" should {
    "reply with empty reading if no value is known or the value if exist"
        in {
         /*...*/
    }
}
```

Questo comporta un vantaggio per quanto riguarda la facilità di comprensione delle azioni che vengono intraprese per verificare il funzionamento del sistema.

5.2 Esempio applicativo

Per verificare il funzionamento del middleware su un caso di studio vero e proprio, si è sfruttato l'esempio del Gradiente: programma aggregato presente in Scafi che, dati un insieme di nodi sorgenti, identifica le distanze degli altri nodi da questi.

Tutti i dispositivi si trovano sullo stesso ActorSystem e sulla stessa JVM (Figura 5.1), e ogni dispositivo è stato configurato come illustrato nella sezione 4.1.

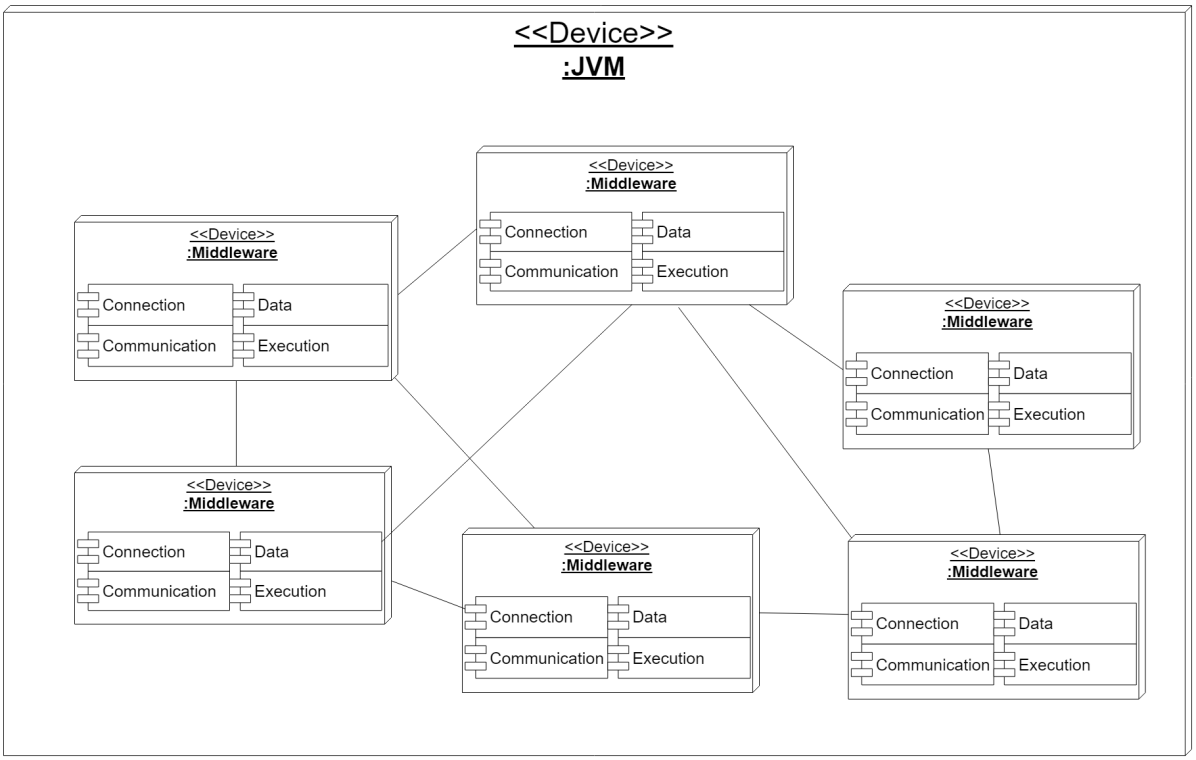


Figura 5.1: Deployment della simulazione.

Lo scenario illustrato in 5.2 prevede che: le logiche di vicinato siano di tipo *Spatial* e i valori che devono essere confrontati tra i dispositivi siano le coordinate Gps.

```
val aggregateProgram = Gradient()
val nbrLogics = SpatialLogics()
val currentLogics = "Gps"
```

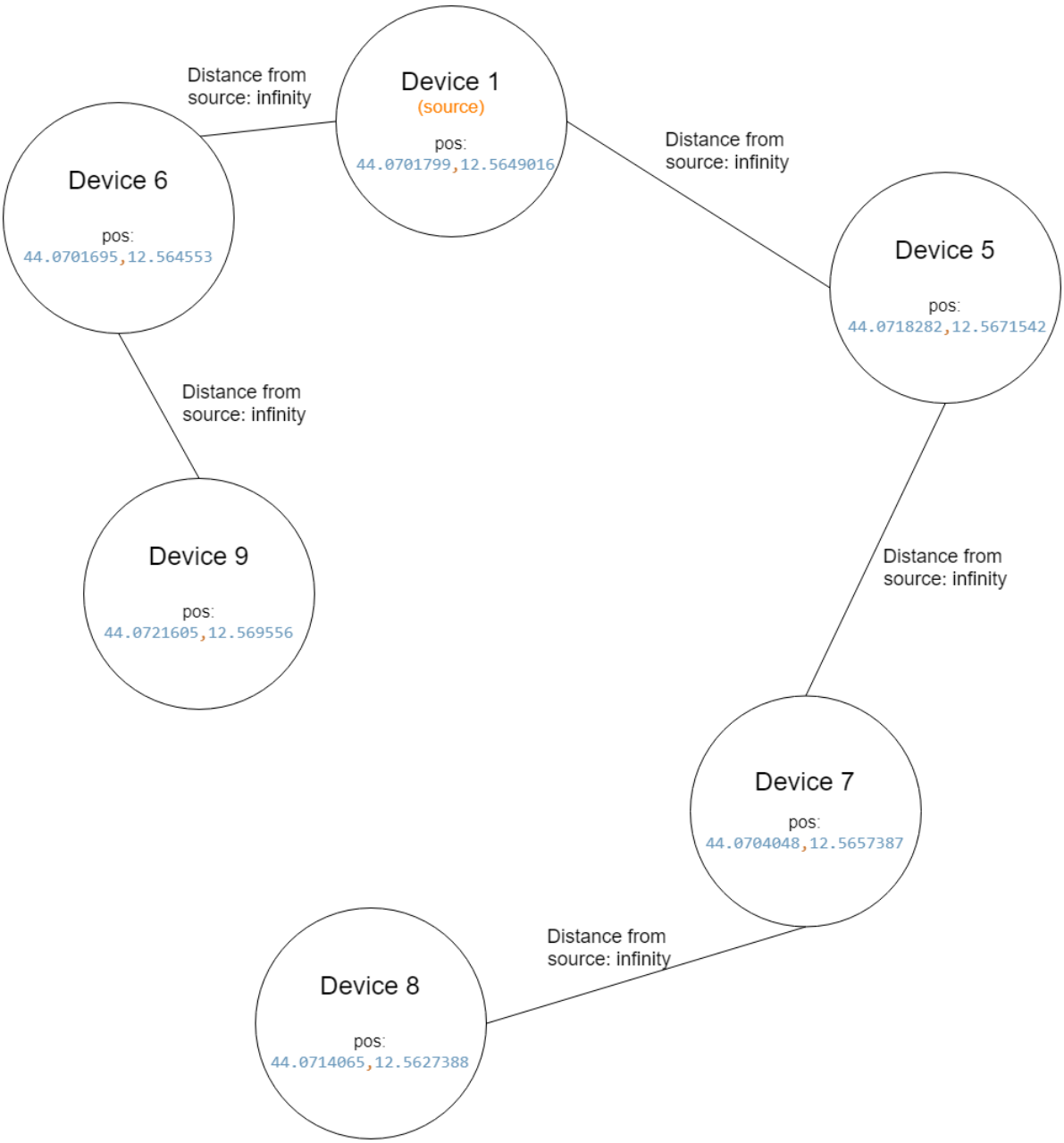


Figura 5.2: Scenario iniziale.

La distanza tra due nodi è calcolata in kilometri da un semplice algoritmo che considera il raggio di curvatura della Terra. Un Device è ritenuto vicino se questa distanza è minore di 10. Ogni nodo invia all'altro un messaggio di richiesta di vicinato con i propri id e posizione corrente. A questo punto i processi di riconoscimento dei vicini, scambio di

export ecc. iniziano in automatico scanditi da un tempo di 500 millisecondi.

```
localTopMan.tell(RequestTrackNbr(5, Option(GpsCoordinates(44.0718282,
    12.5671542))), remoteTopMan2)
remoteTopMan2.tell(RequestTrackNbr(1, Option(GpsCoordinates(44.0701799,
    12.5649016))), localTopMan)

localTopMan.tell(RequestTrackNbr(6, Option(GpsCoordinates
    (44.0701695,12.564553))), remoteTopMan3)
remoteTopMan3.tell(RequestTrackNbr(1, Option(GpsCoordinates(44.0701799,
    12.5649016))), localTopMan)

remoteTopMan4.tell(RequestTrackNbr(5, Option(GpsCoordinates
    (44.0718282,12.5671542))), remoteTopMan2)
remoteTopMan2.tell(RequestTrackNbr(7, Option(GpsCoordinates
    (44.0704048,12.5657387))), remoteTopMan4)

remoteTopMan4.tell(RequestTrackNbr(8, Option(GpsCoordinates
    (44.0714065,12.5627388))), remoteTopMan5)
remoteTopMan5.tell(RequestTrackNbr(7, Option(GpsCoordinates
    (44.0704048,12.5657387))), remoteTopMan4)

remoteTopMan3.tell(RequestTrackNbr(9, Option(GpsCoordinates
    (44.0721605,12.569556))), remoteTopMan6)
remoteTopMan6.tell(RequestTrackNbr(6, Option(GpsCoordinates
    (44.0701695,12.564553))), remoteTopMan3)
```

Il Device 1 è segnato come nodo sorgente e inizialmente, l'output del programma mostra l'unica distanza conosciuta sia proprio quella del Device 1, che ha distanza zero da se stesso. Gli altri Device non sono ancora entrati in possesso delle informazioni necessarie a determinare la distanza, per cui il loro valore è *infinity*.

```
[akka://AggregateMiddleware/user/LocalActuator] Device 1 distance from
    source: 0.0

[akka://AggregateMiddleware/user/RemoteActuator4] Device 7 distance from
    source: Infinity
```

```
[akka://AggregateMiddleware/user/RemoteActuator2] Device 5 distance from
   source: Infinity

[akka://AggregateMiddleware/user/RemoteActuator3] Device 6 distance from
   source: Infinity

[akka://AggregateMiddleware/user/RemoteActuator5] Device 8 distance from
   source: Infinity

[akka://AggregateMiddleware/user/RemoteActuator6] Device 9 distance from
   source: Infinity
```

Dopo alcuni round, quindi dopo che i device si sono scambiati vicendevolmente gli *export*, si ottengono i risultati sperati: il Gradiente determina la distanza di un nodo dalla sorgente sommando tutte le distanze accumulate dagli altri nodi. Più nodi sono presenti nel sistema, più i valori sono accurati. Di seguito e in figura 5.3, si mostra il risultato ottenuto dopo alcune iterazioni.

```
[akka://AggregateMiddleware/user/RemoteActuator3] Device 6 distance from
   source: 0.027874444472462158

[akka://AggregateMiddleware/user/RemoteActuator2] Device 5 distance from
   source: 0.25686401666981584

[akka://AggregateMiddleware/user/LocalActuator] Device 1 distance from
   source: 0.0

[akka://AggregateMiddleware/user/RemoteActuator6] Device 9 distance from
   source: 0.4847865837368665

[akka://AggregateMiddleware/user/RemoteActuator5] Device 8 distance from
   source: 0.7156714544297231

[akka://AggregateMiddleware/user/RemoteActuator4] Device 7 distance from
   source: 0.4513873026591636
```

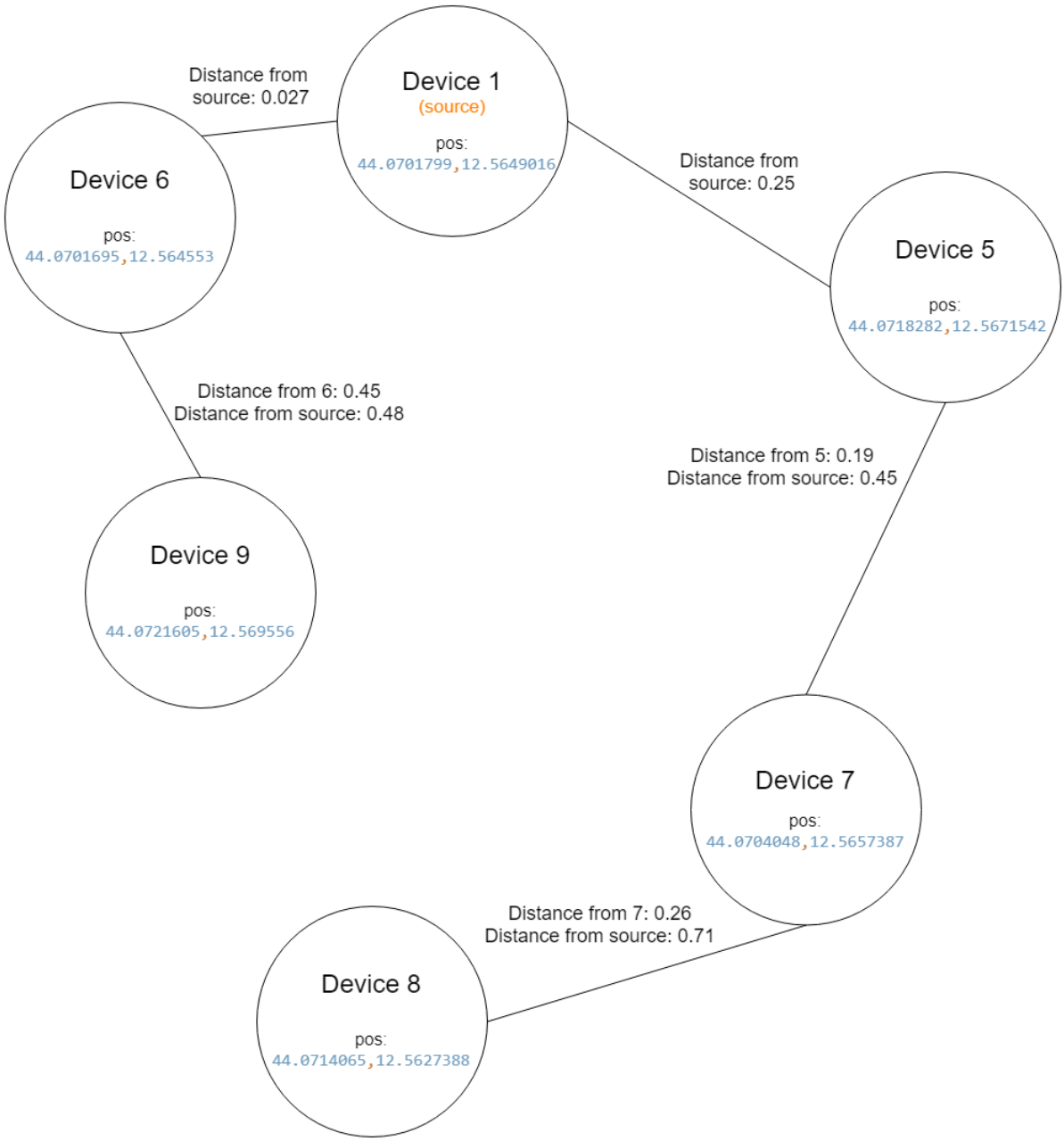


Figura 5.3: Scenario finale.

Conclusioni

Durante il processo di progettazione e realizzazione di un middleware per applicazioni aggregate, è necessario tenere conto di una grande quantità di fattori.

Molti di questi caratterizzano tutti i sistemi pervasivi o Iot, mentre altri sono specifici del modello di Aggregate computing e il software deve permettere di sfruttarne appieno le potenzialità. Per quanto riguarda la parte generale della tesi, sono emersi aspetti di analisi che hanno dimostrato la grande complessità del creare sistemi di questo tipo e la necessità di fornire un design di qualità, in modo da garantire una facile e ordinata implementazione. La fase preliminare ha poi aiutato ad approfondire la conoscenza del problema, stimolando la nascita di spunti diversi per affrontarlo.

L'utilizzo di un modello ad attori ha permesso di risolvere facilmente problemi legati alla concorrenza. Inoltre, al momento dell'implementazione, grazie alla modalità di gestione degli attori fornita da Akka, raramente si sono riscontrati errori o problemi a *run-time*. Questo modello ha consentito anche di sperimentare tecniche sofisticate di progettazione architetturale.

Nell'ultima parte, riguardante la realizzazione del modello a comunicazione diretta, sono stati concretizzati alcuni aspetti del middleware che hanno permesso un ulteriore approfondimento dei problemi legati alla comunicazione "peer to peer" tra device. Buona parte dei requisiti prefissati, per quanto riguarda questo modello, sono stati soddisfatti, mentre gli altri potranno essere facilmente implementati nel processo di completamento del middleware.

Per quanto riguarda lavori futuri, si potrebbero affrontare i seguenti sviluppi:

- ampliamento ed integrazione del middleware della sua parte "client/server" (o a comunicazione indiretta).

- test su un caso di studio reale, portando il middleware su diversi dispositivi fisici e quindi su diverse JVM. Questo non dovrebbe comportare problemi, anche grazie alla caratterisitca della *location transparency*, e permetterebbe così di ottenere una comunicazione tra attori distribuita.
- testing sul Gradiente aumentando sostanzialmente il numero di nodi, in modo da notare l'incremento dell'accuratezza nel determinare le distanze.
- definizione di nuove configurazioni per stabilire qual è il vicinato di un dispositivo.
- implementazione di altri programmi aggregati.

Bibliografia

[1] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the internet of things."

[2] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for internet of things: A survey."

[3] V. Raychoudhurya, J. Caob, M. Kumarc, and D. Zhangd, "Middleware for pervasive computing: A survey."

[4] R. Casadei and M. Viroli. Scafi repository. [Online]. Available: https://github.com/scafi/scafi

[5] V. Vernon, *Reactive Messagging Patterns With The Actor Model.* Addison-Wesley Professional, 2015.

[6] Akka documentation. [Online]. Available: https://akka.io/

[7] Gamma, Helm, Johnson, and Vlissides, *Design Patterns, Elementi per il riuso di software ad oggetti.* Pearson, 2008.

[8] R. Casadei, "Aggregate programming in scala: a core library and actor-based platform for distributed computational fields."

[9] G. Aguzzi, "Sviluppo di un front-end di simulazione per applicazioni aggregate nel framework scafi."

[10] M. Viroli, R. Casadei, and D. Pianini, "On execution platforms for large-scale aggregate computing."

[11] G. Aguzzi, “Sviluppo di un front-end di simulazione per applicazioni aggregate nel framework scafi.”

[12] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima Press, 2016.

[13] N. Dao. Actor-based concurrency and akka fundamentals. [Online]. Available: https://www.slideshare.net/ngocdaothanh/actor-and-akka-fundamentals

[14] Gradle documentation. [Online]. Available: https://gradle.org/

[15] Scala documentation. [Online]. Available: https://www.scala-lang.org/