ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Zenoh network per Aggregate Computing in Rust: verso comunicazioni peer-to-peer su reti eterogenee

Tesi di laurea in:
Programmazione Ad Oggetti

Relatore
Prof. Mirko Viroli

Correlatori
Dott. Nicolas Farabegoli
Dott. Gianluca Aguzzi

Candidato
Ludovico Maria Spitaleri

I Sessione di Laurea
Anno Accademico 2025-2026

Sommario

I sistemi embedded distribuiti, come gli sciami robotici e le reti Internet of Things (IoT), richiedono un coordinamento autonomo e resiliente, senza un orchestratore centrale, operando spesso con risorse computazionali e di memoria limitate. Aggregate Computing (AC) è un paradigma di macroprogrammazione adatto a questi scenari, ma molti framework esistenti, come Scala Fields (ScaFi), sono pensati per dispositivi general-purpose e risultano incompatibili con i microcontrollori a basso costo.

Yet Another Aggregate Implementation in Rust (YAAIR) è un framework che nasce per superare questo limite, proponendo un'architettura a strati che disaccoppia la logica del programma aggregato dal livello di rete. Sfruttando Rust, garantisce prestazioni predicibili, previene errori derivanti da un uso improprio della memoria e mette a disposizione astrazioni di alto livello minimizzando l'impatto sulle risorse disponibili.

Il contributo principale di questa tesi consiste nella progettazione ed implementazione di un livello di rete decentralizzato per questo framework basato sul protocollo Zenoh, pensato per superare i vincoli topologici di soluzioni tradizionali come Message Queuing Telemetry Transport (MQTT) e per supportare dispositivi eterogenei, da embedded a sistemi general-purpose.

Il risultato finale è stato valutato tramite test sperimentali, volti a verificare il corretto funzionamento della comunicazione tra dispositivi di diverso tipo e l'integrazione della rete con YAAIR per il suo utilizzo nell'esecuzione di un programma aggregato.

Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

Introduzione

Negli ultimi anni, la diffusione di dispositivi Internet of Things (IoT) e di sistemi pervasivi ha reso sempre più centrale il problema del coordinamento di reti distribuite su larga scala. Sciami robotici, reti di sensori e sistemi embedded interconnessi devono spesso operare in modo autonomo e resiliente, senza la possibilità di affidarsi ad un orchestratore centrale, mentre gli approcci tradizionali alla programmazione distribuita faticano a garantire un coordinamento trasparente tra centinaia o migliaia di nodi. A questo si aggiunge la natura intrinsecamente vincolata di molti di questi dispositivi, che dispongono di risorse di calcolo e di memoria limitate, limitando le possibilità di scelta delle tecnologie e di strategie di implementazione.

Aggregate Computing (AC) si è affermato come un paradigma di macroprogrammazione capace di rispondere a queste esigenze, astraendo l'esecuzione di un singolo programma su un'intera rete di dispositivi e trattandola come un'unica entità collettiva. Framework esistenti come Scala Fields (ScaFi) hanno permesso di sperimentare efficacemente i principi di AC in ambito di simulazione e su dispositivi general-purpose, ma la loro dipendenza da runtime e Garbage Collector (GC) li rende incompatibili con i microcontrollori a basso costo, che richiedono invece una gestione deterministica della memoria e garanzie di esecuzione in tempo reale.

In questo contesto si inserisce Yet Another Aggregate Implementation in Rust (YAAIR), un framework open-source pensato per superare i limiti prestazionali dei framework di AC basati su linguaggi ad alto livello. Questa libreria è infatti implementata nel linguaggio Rust, che grazie ad un sistema di compilazione pro-

gettato per la produzione di programmi eseguibili ottimizzati ed autosufficienti soddisfa i requisiti fondamentali per la programmazione di sistemi embedded.

I protocolli IoT tradizionali come Message Queuing Telemetry Transport (MQTT)[1] richiedono la presenza obbligatoria di un broker centrale, il quale costituisce un singolo punto di fallimento e un possibile collo di bottiglia per la rete, in contrasto con la natura decentralizzata che si vorrebbe garantire ai sistemi di AC. Zenoh si pone come alternativa a questi protocolli, proponendo un'infrastruttura di rete decentralizzata orientata a basse latenze e alla scalabilità su dispositivi eterogenei, capace di supportare comunicazioni Peer to Peer (P2P) senza vincoli topologici stretti.

Contributo della tesi YAAIR adotta un'architettura a strati che disaccoppia la logica di calcolo del programma aggregato dalla serializzazione e dai protocolli di rete sottostanti, permettendo al sistema di operare in modo agnostico rispetto al trasporto fisico dei dati e di essere facilmente estendibile a diversi protocolli di comunicazione. Il contributo di questa tesi consiste nella progettazione e nell'implementazione di un livello di rete decentralizzato per il framework, basato sul protocollo Zenoh, in grado di supportare la comunicazione tra dispositivi eterogenei, da microcontrollori che dispongono di risorse fortemente limitate fino a sistemi general-purpose dotati di un sistema operativo completo.

Struttura dei capitoli Il Capitolo 2 introduce i concetti di background necessari alla comprensione del contributo: vengono presentati i principi di AC, con particolare attenzione alla loro applicazione su dispositivi a risorse limitate, il framework YAAIR, il protocollo di rete Zenoh e l'ecosistema offerto da Rust per la programmazione embedded.

A seguire, il Capitolo 3 analizza gli obiettivi e i requisiti del contributo, evidenziando i vincoli imposti a livello hardware e software dalle tecnologie e dai dispositivi utilizzati.

Con il Capitolo 4 si descrivono le scelte architetturali adottate nella realizzazione dei vari componenti del progetto, che devono soddisfare i requisiti e i vincoli definiti nel capitolo precedente.

[1] `https://mqtt.org/`

Nel Capitolo 5 si illustra in dettaglio l'implementazione dell'architettura progettata, dall'adattamento della libreria Zenoh a diverse piattaforme eterogenee alla creazione della rete e la sua integrazione con YAAIR, motivando le scelte tecniche effettuate e le librerie utilizzate.

All'interno del Capitolo 6 vengono presentati i test sperimentali condotti per validare i componenti principali del contributo di questa tesi, descrivendone gli obiettivi, l'implementazione e i risultati ottenuti per ciascuno di essi.

Infine, il Capitolo 7 riassume il lavoro svolto rispetto agli obiettivi fissati, e propone alcuni possibili sviluppi futuri per il progetto.

Capitolo 2

Background

Questo capitolo introduce le tecnologie, i paradigmi e i concetti fondamentali necessari per la comprensione del contributo della tesi. Vengono presentati i principi di AC, con particolare attenzione all'implementazione nel contesto dei dispositivi con risorse limitate, seguiti dal framework YAAIR, dal protocollo di rete Zenoh[1] e dall'ecosistema di Rust per la programmazione embedded.

2.1 Aggregate computing

La proliferazione di dispositivi IoT e sistemi pervasivi ha reso sempre più complessa la gestione di reti distribuite su larga scala, e gli approcci tradizionali faticano a garantire coordinamento trasparente e resiliente tra centinaia o migliaia di nodi [ABB+25].

AC è un paradigma di *macroprogrammazione* che astrae l'esecuzione di un singolo programma su una rete di dispositivi, trattandola come un'unica entità collettiva: invece di programmare separatamente il comportamento di ogni singolo nodo, si descrive il comportamento collettivo, lasciando che sia il framework a occuparsi di tradurlo in azioni eseguite localmente da ciascun dispositivo.

Al centro di questo modello vi sono il *field calculus* ed i *campi computazionali* [ABB+25]. Un campo computazionale è un'astrazione che associa un valore ad ogni dispositivo della rete in un determinato istante di tempo: concettualmente, è

[1] `https://zenoh.io/`

una funzione che mappa coppie spazio-temporali a valori di un certo tipo, come numeri, booleani o strutture dati più complesse. Il *field calculus* è il linguaggio formale che definisce come questi campi possano essere costruiti e combinati tra loro tramite operatori funzionali, permettendo di esprimere computazioni globali sulla rete a partire da semplici operazioni locali. Ad esempio, in uno sciame robotico, si possono definire diversi campi computazionali che associano ad ogni robot un valore come la distanza minima da una sorgente o un vettore rappresentante il movimento desiderato [Ric].

A livello operativo, ogni dispositivo esegue ripetutamente lo stesso programma aggregato secondo un ciclo, detto *round*: ad ogni iterazione, il dispositivo raccoglie i messaggi ricevuti dai propri vicini durante il round precedente, calcola un nuovo valore del campo computazionale combinando questi messaggi con il proprio stato locale, ed infine esporta il risultato ottenuto, sia per processarlo localmente (ad esempio, comandando un attuatore) sia per inviarlo ai dispositivi con cui è in relazione, così che possa essere utilizzato nel loro round successivo [Ric]. La nozione di *vicinato* di un dispositivo, ovvero l'insieme dei nodi con cui scambia messaggi, determina quali altri nodi contribuiscono al calcolo del campo in un determinato punto della rete. Ogni applicazione può definire il vicinato di un nodo secondo strategie differenti, per esempio in base alla distanza fisica tra i dispositivi o tramite una relazione di rete esplicita.

Un esempio classico di programma aggregato è la costruzione di un gradiente spaziale, utilizzato per misurare la distanza di ogni nodo da una sorgente, dove le relazioni tra i nodi sono basate sul concetto di vicinato definito tramite distanza cartesiana tra loro [ABB+25]. In questo caso, il campo computazionale calcolato da ogni nodo è la propria distanza minima dalla sorgente: ad ogni round, ciascun dispositivo osserva i valori di distanza ricevuti dai propri vicini, vi somma la distanza che lo separa da essi, e si propone come nuova distanza il valore minimo ottenuto, così da propagare progressivamente l'informazione a partire dalla sorgente verso il resto della rete (Figura 2.1).

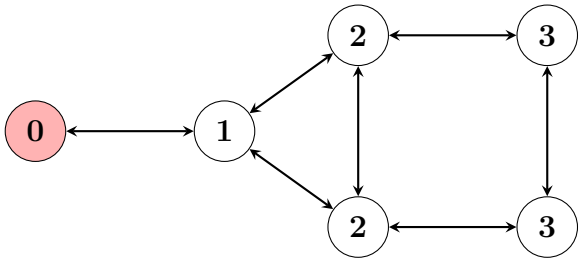

Figura 2.1: Rappresentazione di un gradiente spaziale tramite un grafo. Il nodo sorgente (in rosso) ha distanza 0, mentre i nodi adiacenti calcolano la propria distanza propagando il risultato del programma aggregato attraverso il vicinato.

2.1.1 Aggregate computing su sistemi embedded

Il modello di computazione distribuita offerto da AC si pone come soluzione naturale per reti di dispositivi IoT e sciami robotici, mentre i framework preesistenti sono stati concepiti per simulazioni o esecuzioni su dispositivi general-purpose. Questo li rende incompatibili con i microcontrollori a basso costo che non dispongono di risorse sufficienti e sistemi operativi tradizionali, in quanto spesso si affidano a macchine virtuali o meccanismi di gestione della memoria basati su GC.

Per colmare questa lacuna e portare l'AC sui dispositivi embedded, il linguaggio Rust rappresenta una soluzione ideale. Rust è un linguaggio di programmazione compilato, nato nel 2006 e sviluppato inizialmente da Mozilla[2], pensato per offrire un livello di controllo sull'hardware e sulle prestazioni paragonabili a quello di linguaggi di basso livello come C e C++, senza però rinunciare alla sicurezza nella gestione delle risorse. A differenza di questi ultimi, infatti, Rust non richiede una gestione manuale dell'allocazione e deallocazione della memoria, né si appoggia ad un GC per farlo automaticamente: il compilatore verifica staticamente, tramite il suo rigoroso sistema di tipi ed il modello di *ownership*, che ogni porzione di memoria abbia esattamente un proprietario responsabile della sua liberazione, eliminando così a tempo di compilazione intere classi di errori comuni nei linguaggi di basso livello, come accessi ad indirizzi non più validi o perdita di puntatori ad aree di memoria non precedentemente liberate. Questo controllo avviene interamente in fase di compilazione e non introduce alcun costo aggiuntivo a runtime, permettendo

[2] `https://www.mozilla.org/it/`

a Rust di fornire astrazioni di alto livello a costo zero: costrutti comodi da usare in fase di sviluppo che, una volta compilati, producono codice macchina con prestazioni equivalenti a quello scritto manualmente in un linguaggio di più basso livello. Queste caratteristiche garantiscono prestazioni predicibili, latenze minime e un controllo sicuro e granulare sulle risorse hardware, aspetti fondamentali per la programmazione embedded, ambito in cui Rust ha guadagnato un'adozione crescente negli ultimi anni.

2.2 YAAIR

YAAIR[3] è un framework Rust open-source per l'AC, ideato per risolvere le limitazioni prestazionali delle alternative basate su linguaggi ad alto livello che non permettono l'esecuzione su determinate tipologie di sistemi. Sfruttando la natura flessibile di Rust data dal concetto di *trait*, ovvero interfacce che definiscono un comportamento comune implementabile da diversi tipi di oggetti, YAAIR propone un modello architetturale a strati che disaccoppia la logica di calcolo del programma aggregato dalla serializzazione e dai protocolli di rete sottostanti. Questo livello di isolamento consente al sistema di operare indipendentemente dall'hardware, eseguendo cicli di valutazione in modo totalmente agnostico rispetto al trasporto fisico dei dati.

Il contributo di questa tesi si concentra sull'implementazione di una alternativa decentralizzata per lo strato di rete basata sul protocollo Zenoh. Essendo YAAIR relativamente giovane, la libreria si trova in una fase di sviluppo attivo, la sua struttura interna può ancora evolvere in maniera sostanziale, e non è ancora disponibile una documentazione estesa.

2.3 Zenoh

Zenoh è un protocollo di comunicazione di nuova generazione per la realizzazione di infrastrutture di rete decentralizzate ad alte prestazioni. Ottimizzato per offrire un throughput elevato e latenze minime, Zenoh permette di unificare la comunicazione

[3] `https://github.com/nicolasfara/yaair`

pub/sub e *query/reply* all'interno di un'unica astrazione. A differenza degli approcci tradizionali dell'ecosistema IoT come MQTT, che si appoggiano obbligatoriamente ad un broker centrale, Zenoh è progettato per comunicazioni *edge-to-cloud* senza vincoli topologici stretti [Zen26b], permettendo di creare reti versatili e efficienti.

2.3.1 Modello di comunicazione

Per la comunicazione Zenoh adotta un approccio ibrido altamente flessibile in cui i nodi sulla rete possono assumere ruoli differenti:

- **Client**: dispositivi con risorse minime, che si connettono direttamente ai router o ad altri peer per l'inoltro dei pacchetti senza occuparsi della topologia globale.
- **Peer**: nodi in grado di comunicare in maniera diretta fra loro e di gestire topologie P2P auto-organizzanti grazie a meccanismi integrati di scoperta automatica dei nodi nella rete (*scouting*).
- **Router**: elementi strutturali utilizzati per unire reti isolate o scalare la topologia su geografie estese.

Indipendentemente dal ruolo di un nodo, Zenoh permette di astrarre la comunicazione sulla rete facendo sì che ognuno di essi possa agire in maniera indipendente. Questo lo rende una soluzione ideale per i programmi aggregati in quanto permette di comunicare in maniera diretta con altri nodi sulla rete in maniera agnostica rispetto alle connessioni e salti necessari per arrivarci.

2.3.2 Attori e concetti fondamentali

Sfruttando un approccio incentrato sui dati, il sistema di indirizzamento è dettato tramite *topic*, stringhe che seguono una struttura ad albero simile a quella utilizzata dai sistemi operativi per i percorsi dei file, e *key expressions*, che integrando dei costrutti ispirati alle espressioni regolari permettono di rappresentare insiemi di topic.

Zenoh permette di utilizzare modelli di comunicazione *pub/sub* e *query/reply* in maniera ibrida all'interno della stessa sessione, così da poter sfruttare la soluzione

più adatta per ogni problema senza vincoli sulla strategia di interazione. Ogni nodo della rete mantiene una sessione, che rappresenta la connettività del nodo, dalla quale dichiara liberamente entità per i differenti modelli di comunicazione che entreranno a far parte della rete.

Nel modello *pub/sub*, un *publisher* pubblica dati su un topic specifico senza conoscere né attendere la presenza di alcun destinatario, mentre uno o più *subscriber* possono iscriversi a molteplici topic tramite key expressions per reagire alla pubblicazione di nuovi valori. Questo modello è quindi adatto a scenari in cui un nodo deve notificare continuamente un dato, senza necessità di una risposta esplicita da parte di chi è in ascolto (Figura 2.2).

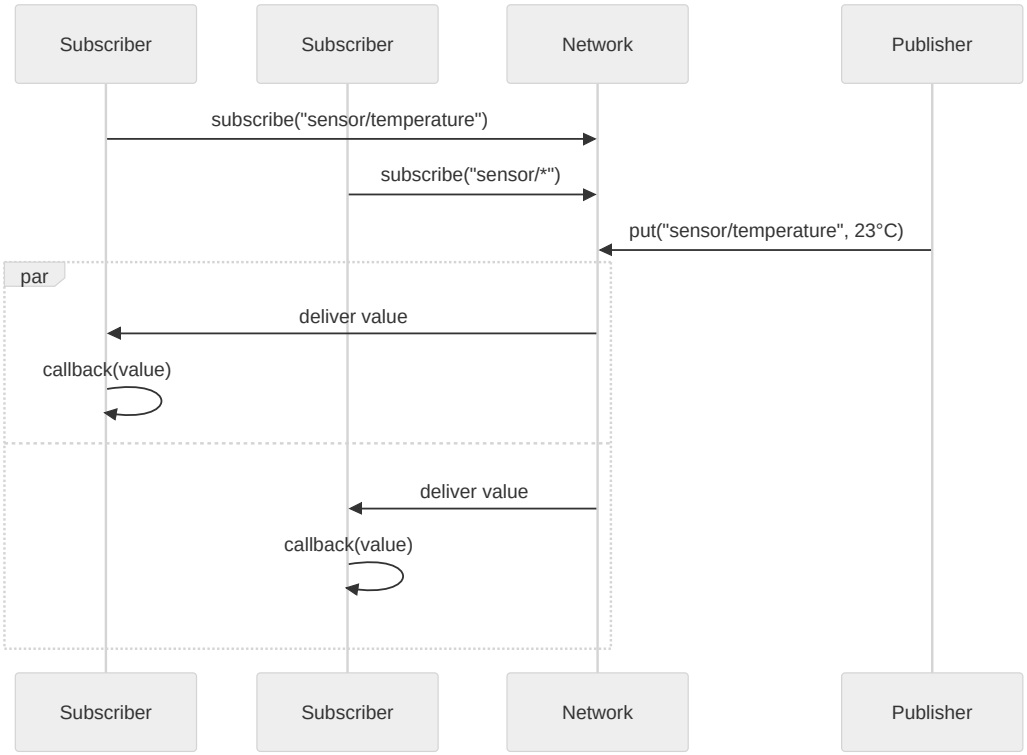

Figura 2.2: Diagramma di sequenza del modello *pub/sub* in Zenoh. L'entità *Network* non rappresenta un'unità centrale, bensì la rete P2P su cui ogni nodo comunica le proprie azioni, senza necessità di connessioni dirette tra i partecipanti.

Nel modello *query/reply*, invece, un *querier* effettua attivamente una richiesta su un topic, alla quale rispondono i *queryable* registrati su key expression di

competenza. A differenza della comunicazione *pub/sub*, qui è il richiedente a guidare lo scambio, rendendo l'interazione tra le entità simile al modello Remote Procedure Call (RPC). Questo modello è adatto a scenari in cui un nodo necessita di un dato specifico nel momento stesso in cui lo richiede (Figura 2.3).

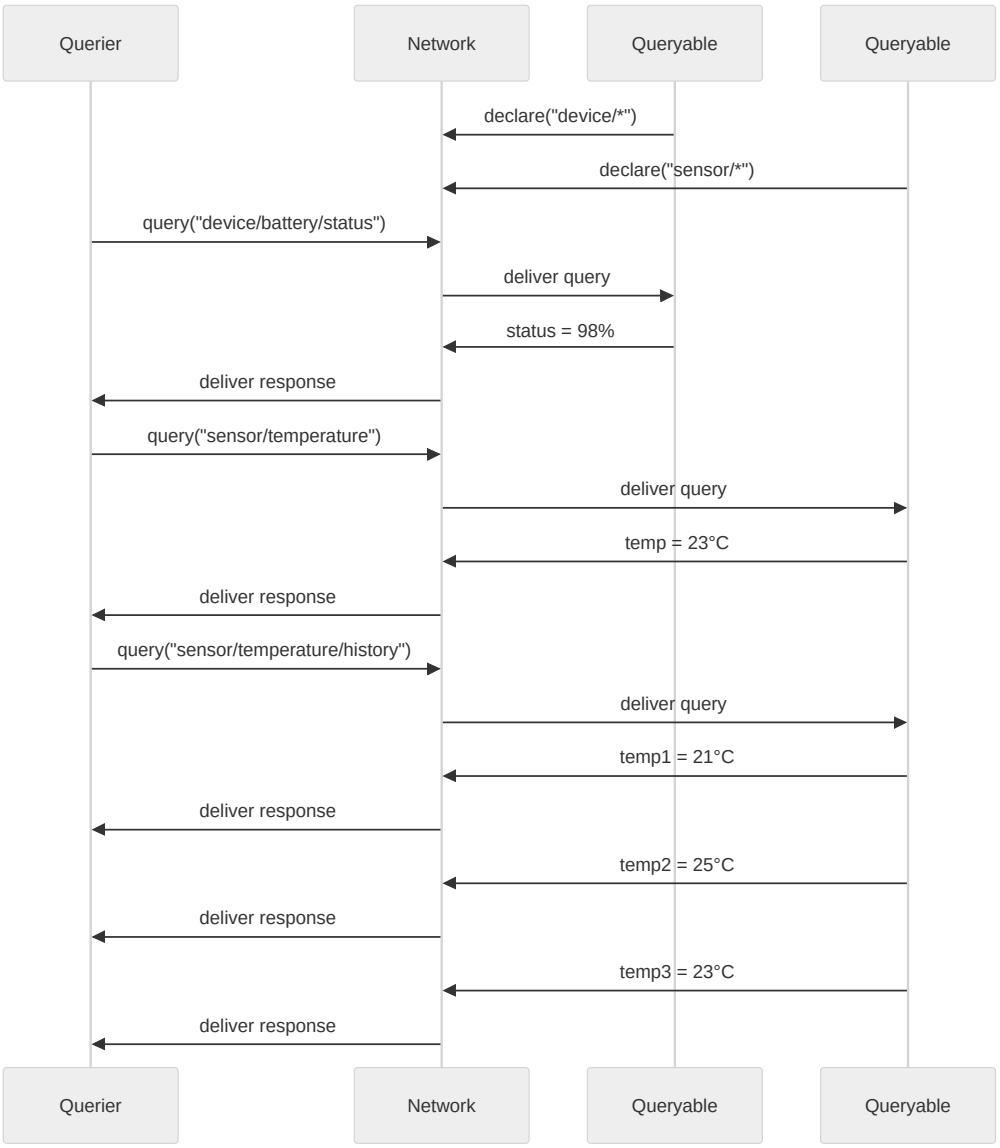

Figura 2.3: Diagramma di sequenza del modello *query/reply* in Zenoh. Come nella fig. 2.2, i messaggi verso l'entità *Network* rappresentano la comunicazione decentralizzata con i vari nodi presenti.

2.3.3 Zenoh per sistemi embedded

Nonostante la flessibilità di Rust come linguaggio che ne permette l'utilizzo anche per la programmazione di dispositivi *bare-metal*, come diversi dispositivi utilizzati per la programmazione IoT, la versione standard della libreria per Zenoh dipende da alcune funzionalità specifiche che richiedono la presenza di un sistema operativo completo sul dispositivo.

Per colmare questa lacuna è stato introdotto Zenoh pico[4], un porting leggero della libreria in C appositamente concepito per sistemi embedded dotati di vincoli severi. Questa versione copre quasi completamente lo standard definito dalla libreria originale, omettendo alcune funzionalità incompatibili con le risorse limitate offerte dai dispositivi embedded come la possibilità di utilizzare un nodo come *router*.

Si menziona, a corredo dell'ecosistema, il nascente progetto `zenoh-nostd`[5], un'iniziativa volta a fornire un'implementazione del protocollo interamente in Rust indipendente dalla libreria standard e dalla piattaforma su cui viene utilizzata. Tuttavia, trovandosi ancora in una fase sperimentale, per la creazione del contributo di questa tesi è stato utilizzato Zenoh pico creando un wrapper Rust per la libreria.

2.4 Embedded Rust

Rust offre un ecosistema robusto e maturo per la programmazione di sistemi embedded, con un supporto nativo del compilatore per diverse architetture di CPU e piattaforme, possibilità di utilizzare funzioni provenienti da librerie esterne precompilate ed accesso alla libreria standard completamente opzionale per i dispositivi che non ne offrono un'implementazione.

Un altro aspetto che rende questo linguaggio particolarmente versatile per la creazione di programmi che devono supportare sistemi eterogenei è la scelta di non imporre un'implementazione specifica per funzionalità che, su altri linguaggi, spesso sono parte integrante della tecnologia. Un esempio rappresentativo è il supporto per la programmazione asincrona tramite i costrutti `async/await`: generalmente l'esecuzione di funzioni concorrenti richiede un runtime specifico fornito dal lin-

[4] `https://github.com/eclipse-zenoh/zenoh-pico`
[5] `https://github.com/ZettaScaleLabs/zenoh-nostd`

guaggio, mentre in Rust questi costrutti sono implementati tramite *trait*, e la scelta dell'implementazione concreta che effettivamente esegue e pianifica i task è lasciata completamente allo sviluppatore, permettendo di scegliere la libreria più adatta alle esigenze specifiche del progetto.

2.4.1 Cross-compilation

Il compilatore `rustc` supporta nativamente la compilazione per piattaforme diverse da quella della macchina host su cui il programma viene sviluppato, permettendo la creazione di artefatti binari autosufficienti ed eseguibili su qualsiasi dispositivo compatibile, senza necessità di dipendenze esterne non esplicitamente definite dal programmatore. L'ecosistema intorno a questo linguaggio permette inoltre di scaricare ed utilizzare versioni alternative della *toolchain*, ovvero l'insieme dei componenti che definiscono un'installazione di Rust, estendendo la compatibilità ad ulteriori piattaforme.

Il linguaggio fornisce inoltre un sistema di compilazione condizionale che permette la definizione di moduli e porzioni di codice in base alle caratteristiche del target per cui viene compilato il programma, come l'architettura, il sistema operativo o la presenza di determinate funzionalità. L'affidabilità di Rust in questo ambito è dimostrata dalla sua crescente adozione nell'industria per lo sviluppo di sistemi critici e con standard di qualità elevatissimi, come il sistema operativo embedded *TockOS* [SCG+25] o la sua recente integrazione per la scrittura di moduli nel kernel Linux [PE24].

2.4.2 Embassy

Embassy[6] è un framework per lo sviluppo di programmi embedded che fornisce interfacce di astrazione verso l'hardware in maniera indipendente dalle specifiche del dispositivo.

In particolare, per la creazione del contributo di questa tesi, è stato sfruttato il runtime per la programmazione di task concorrenti fornito dalla libreria, che

[6] `https://embassy.dev/`

permette l'utilizzo di costrutti per la programmazione asincrona grazie ad un sistema di task concorrenti.

2.4.3 ESP-IDF

Espressif Systems[7], azienda produttrice di microcontrollori della famiglia ESP, supporta lo sviluppo di programmi in Rust per i propri dispositivi fornendo implementazioni della toolchain per le architetture da loro utilizzate, permettendo così l'utilizzo della libreria standard su tutta la loro gamma di prodotti.

Tramite il framework Espressif IoT Development Framework (ESP-IDF), viene fornita un'implementazione dell'astrazione dell'hardware definita da Embassy che permette l'interazione con le periferiche disponibili sul microcontrollore tramite un'unica Application Programming Interface (API) compatibile con tutti i dispositivi da loro prodotti. Sfruttando poi la natura *memory-safe* ed il sistema di gestione degli errori del linguaggio, viene garantito un accesso facile e sicuro a funzionalità come Wi-Fi e comunicazione seriale, spesso utilizzate nel campo dell'IoT.

[7] `https://www.espressif.com/`

Capitolo 3

Analisi

In questo capitolo viene effettuata un'analisi degli obiettivi e requisiti del contributo di questa tesi, concentrandosi anche sui vincoli imposti a livello hardware e software dalle tecnologie e piattaforme coinvolte.

3.1 Obiettivi

L'obiettivo primario del progetto consiste nello sviluppo di una rete decentralizzata per il framework YAAIR che utilizzi il protocollo Zenoh e che supporti sistemi eterogenei: la natura del problema necessita un'analisi non banale delle tecnologie e dell'ecosistema di lavoro che rende il contributo finale il risultato di una serie di processi di studio e sviluppo.

O1 *Studio di Zenoh.* Zenoh è la tecnologia che funge da base fondamentale per la comunicazione nella rete implementata: è quindi necessario conoscere a fondo il suo funzionamento, i suoi costrutti, elementi primari e funzionalità che offre con relative limitazioni.

O2 *Integrazione dell'ecosistema embedded.* I sistemi embedded sono trattati come piattaforma di primaria importanza in questo progetto, e l'integrazione con essi richiede l'utilizzo di diverse librerie e framework oltre che un'attenzione particolare all'utilizzo delle risorse e l'interazione con i componenti del dispositivo. Inoltre, la scelta di Rust come linguaggio comporta la necessità

di creare un wrapper intorno alla libreria C di Zenoh pico che definisca una API memory-safe e più simile alla versione standard di Zenoh.

O3 *Implementazione della rete.* Il contributo principale di questa tesi è la creazione della rete stessa, che deve permettere una comunicazione decentralizzata e affidabile tra i nodi partecipanti in maniera compatibile con YAAIR.

3.2 Vincoli

La natura eterogenea e trasparente della rete richiesta dagli obiettivi del contributo impone alcuni vincoli su alcune scelte progettuali e di architettura.

V1 *Compatibilità con sistemi Espressif.* Nonostante supporti i dispositivi a risorse limitate, anche Zenoh pico richiede l'accesso ad alcune primitive di sistema che non ne permettono l'utilizzo su dispositivi puramente bare-metal. La libreria utilizza PlatformIO[1] come strumento per la compilazione su diverse piattaforme, così da supportare facilmente tutte le famiglie di dispositivi embedded più popolari. Nonostante una così vasta gamma di sistemi supportati fosse l'obiettivo originale anche per questo progetto, l'integrazione di questo strumento all'interno del processo di compilazione Rust non è semplice, di conseguenza per mancanza di tempo si è optato per ridurre il vincolo di compatibilità ai sistemi Espressif, che offrono nativamente un supporto eccellente per il linguaggio scelto tramite il framework ESP-IDF, adattando l'integrazione di Zenoh pico al processo di compilazione Rust senza passare per PlatformIO.

V2 *Compatibilità con YAAIR.* Per sfruttare un'architettura a strati flessibile, YAAIR impone che l'implementazione della rete utilizzata segua un'interfaccia molto semplice ma specifica. La realizzazione di questo componente da parte di questo contributo deve essere perfettamente compatibile con essa e non deve richiedere trattamenti speciali che impediscano al resto del framework di agire normalmente.

[1] `https://platformio.org/`

3.3 Requisiti funzionali

RF1 *Comunicazione tra nodi.* Ciascun nodo deve essere in grado di pubblicare aggiornamenti ed ascoltare quelli dei nodi con cui è in relazione per esportare il proprio risultato in maniera agnostica alla topologia della rete.

RF2 *Scoperta automatica dei nodi.* Al momento dell'ingresso in rete, un nodo deve essere in grado di individuare autonomamente gli altri nodi presenti senza richiedere configurazione manuale o per mezzo di un'unità dedicata, preservando così la natura decentralizzata del sistema.

RF3 *Rilevamento della perdita di un nodo.* La rete deve rilevare l'uscita o il fallimento di un nodo entro un intervallo di tempo configurabile. Alla scadenza del timeout, gli altri partecipanti devono essere notificati e lo stato interno della rete deve essere aggiornato di conseguenza, consentendo al framework di reagire opportunamente all'evento e di non dedicare più risorse al nodo perso.

3.4 Requisiti non funzionali

RNF1 *Portabilità tra Zenoh e Zenoh pico.* L'architettura della rete deve essere strutturata in modo da poter essere compilata e utilizzata sia su sistemi desktop o server tramite la libreria Zenoh standard, sia su sistemi embedded Espressif tramite Zenoh pico, senza duplicare la logica applicativa. Le differenze tra le due librerie devono essere incapsulate in un livello di astrazione dedicato.

RNF2 *Strategie alternative di gestione della perdita di nodi.* Il meccanismo di rilevamento e gestione dell'uscita di un nodo dalla rete deve essere progettato con una struttura modulare, così da permettere l'adozione di strategie diverse a seconda del contesto applicativo. Ad esempio, in scenari con risorse limitate può essere preferibile una strategia semplificata rispetto a una più robusta ma costosa in termini di memoria o banda.

Capitolo 4

Progettazione

Nel seguente capitolo vengono illustrate le scelte architetturali adottate nella realizzazione del sistema per soddisfare tutti i requisiti e raggiungere gli obiettivi definiti nel Capitolo 3, rispettando al contempo i vincoli imposti.

4.1 Architettura generale

In YAAIR, il cuore dell'esecuzione del programma aggregato è lo `Engine`, che date le implementazioni dei vari componenti da utilizzare per il corretto funzionamento del framework permette l'esecuzione di un programma aggregato generico. Tra queste implementazioni è richiesta una rete per la comunicazione, la quale deve soddisfare un'interfaccia ben definita. La rete Zenoh deve quindi soddisfare questo livello di compatibilità (V2), mantenendo internamente l'implementazione specifica di Zenoh utilizzata (standard o pico) ed esponendo delle API alla rete che siano trasparenti rispetto alla tecnologia utilizzata.

Riguardo la configurazione di Zenoh stesso (RNF1), le due versioni offrono opzioni con requisiti differenti, rendendo quindi necessario esporre almeno in parte l'implementazione utilizzata. Per minimizzare questa necessità, viene definita una API comune che segue il *builder pattern* [GHJV94] per la configurazione delle opzioni comuni e più importanti nel contesto di questo progetto. Entrambe le versioni di Zenoh offrono un'implementazione di questa interfaccia, producendo solo al momento della creazione finale un oggetto specifico che potrà essere ulterior-

mente configurato utilizzando funzioni apposite della libreria originale. Per ridurre ulteriormente l'interazione diretta con l'implementazione del protocollo utilizzata, il progetto è strutturato in maniera tale da scegliere tra Zenoh standard e pico a tempo di compilazione e riesporre eventuali tipi e funzioni necessari sotto lo stesso nome (Figura 4.1).

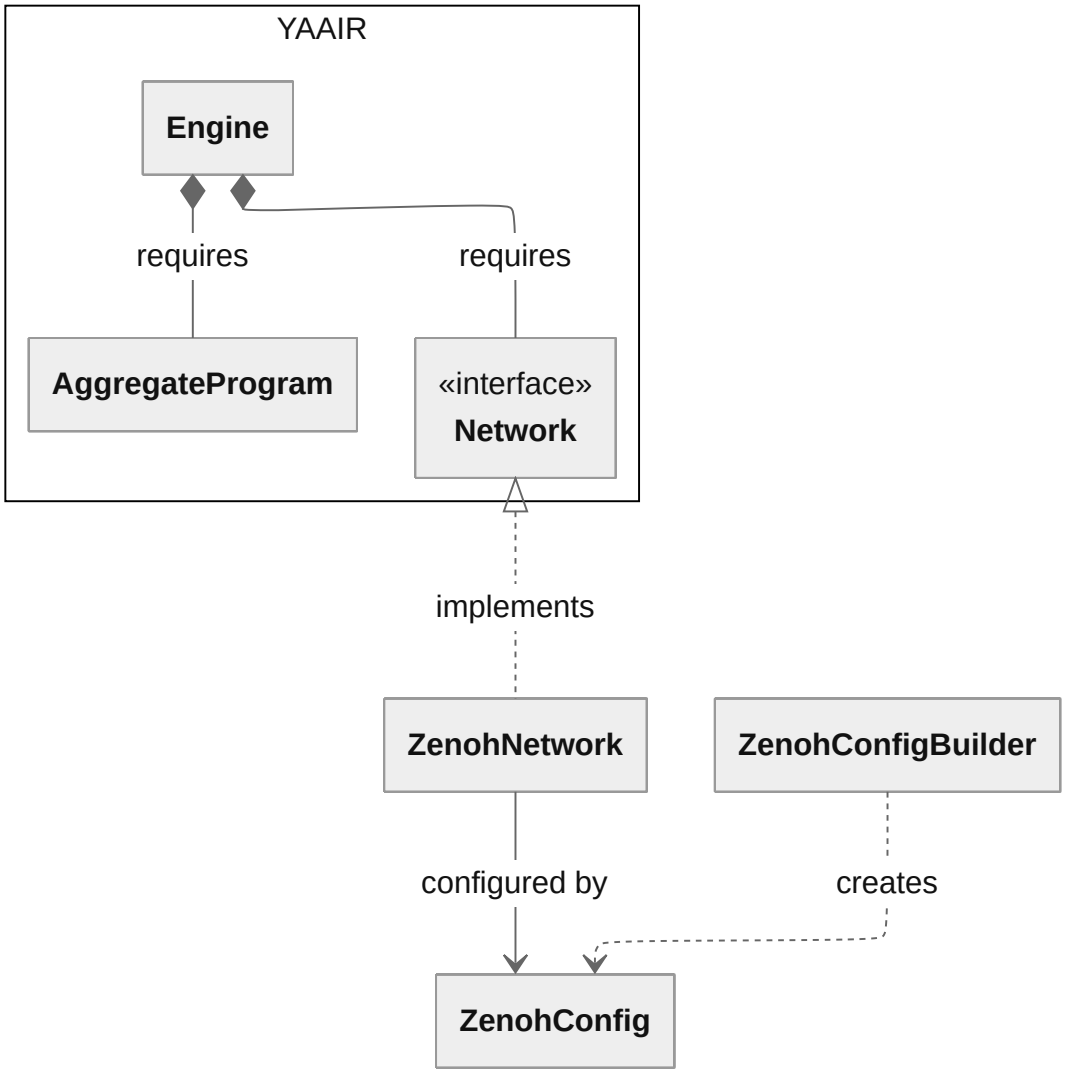

Figura 4.1: Diagramma dell'architettura generale della rete.

4.2 La rete Zenoh

La rete Zenoh implementata è costituita internamente da diverse parti che interagiscono tra loro: al centro di tutto è presente una base di dati che si occupa dell'accesso ed il salvataggio dei messaggi ricevuti dal nodo, con la quale interagiscono dei subscriber in ascolto sulla rete per nuovi messaggi da parte dei nodi partecipanti. Questi messaggi possono essere di due tipi: i risultati aggiornati (RF1) o le notifiche per segnalare la presenza del nodo stesso (RF3).

4.2.1 Memorizzazione e ciclo di vita dei messaggi

La comunicazione distribuita pone un problema fondamentale di disaccoppiamento temporale: i messaggi provenienti dagli altri nodi della rete arrivano in modo asincrono, guidati dalla rete e non dall'avanzamento del programma aggregato. YAAIR, al contrario, interroga la rete in momenti precisi del proprio ciclo di esecuzione, aspettandosi di ricevere una visione coerente e aggiornata dello stato della rete. Occorre quindi un componente che faccia da intermediario, salvando i messaggi al loro arrivo e rendendoli disponibili su richiesta.

Il `MessagesStore` è progettato per assolvere esattamente questo ruolo: si interpone tra le entità Zenoh interne, che scrivono i messaggi alla loro ricezione, e YAAIR, che li legge ogni ciclo del programma, mantenendo una struttura dati che associ ogni nodo al suo ultimo risultato. Poiché Zenoh mantiene i suoi subscriber su thread separati, l'accesso alla mappa è gestito in modo da garantire la correttezza delle operazioni in presenza di concorrenza.

Oltre al contenuto del messaggio, il `MessagesStore` associa ad ogni voce due metadati: la durata di vita attesa (*lifespan*) e l'istante dell'ultimo aggiornamento (*timestamp*). Queste informazioni sono necessarie per rispondere al requisito di rilevamento dei nodi offline (RF3): quando YAAIR interroga la rete per eventuali nuovi risultati ricevuti, viene confrontato il tempo trascorso dall'ultimo aggiornamento con il lifespan dichiarato, così da non fornire risultati obsoleti (Figura 4.2).

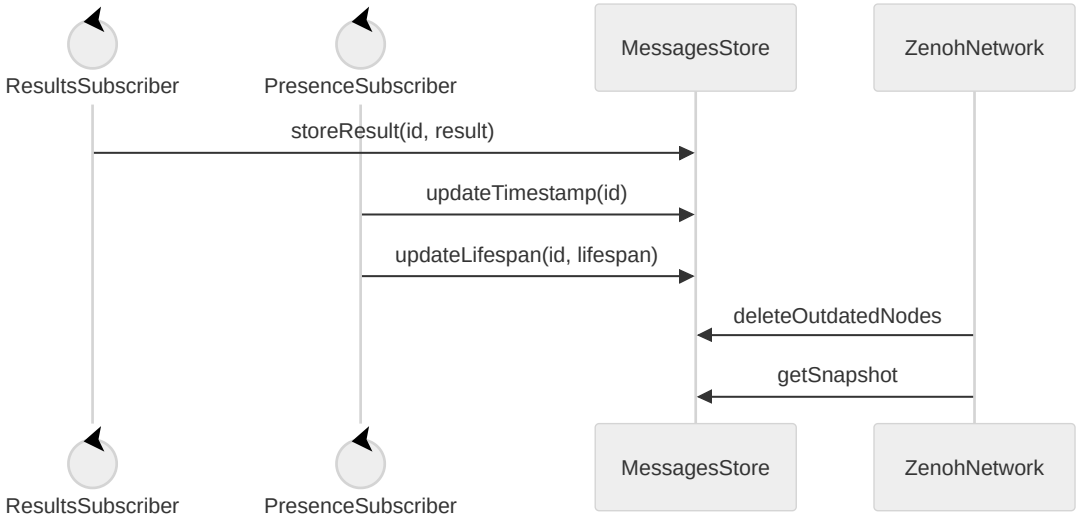

Figura 4.2: Flusso di interazione tra i componenti della rete. I subscriber Zenoh aggiornano il `MessagesStore` alla ricezione di ogni messaggio tramite le operazioni dedicate, mentre la rete può richiedere una fotografia dei risultati attuali o di eliminare eventuali nodi scollegati dalla rete i cui risultati sono ormai obsoleti.

4.2.2 Zenoh e la comunicazione pub/sub

L'interoperabilità con le versioni standard e pico di Zenoh all'interno della rete è gestita secondo una scelta strutturale fondamentale: la rete definisce delle interfacce per l'interazione con i vari elementi offerti dal protocollo, ed interagisce con l'implementazione effettiva solo tramite esse. Questa strategia permette di centralizzare la scelta della versione e di poter interagire con la tecnologia in maniera agnostica alle specifiche.

Internamente, la rete mantiene una sessione Zenoh, la quale definisce la presenza e permette la scoperta automatica dei nodi nella rete (RF2). Da quest'ultima si ha la possibilità di dichiarare publisher e subscriber su diversi topic, utilizzati per l'esportazione dei risultati e gli aggiornamenti sullo stato dei nodi. Ogni nodo della rete pubblica informazioni su topic che condividono il proprio id come radice, e si iscrive a tutti i topic equivalenti degli altri membri della rete.

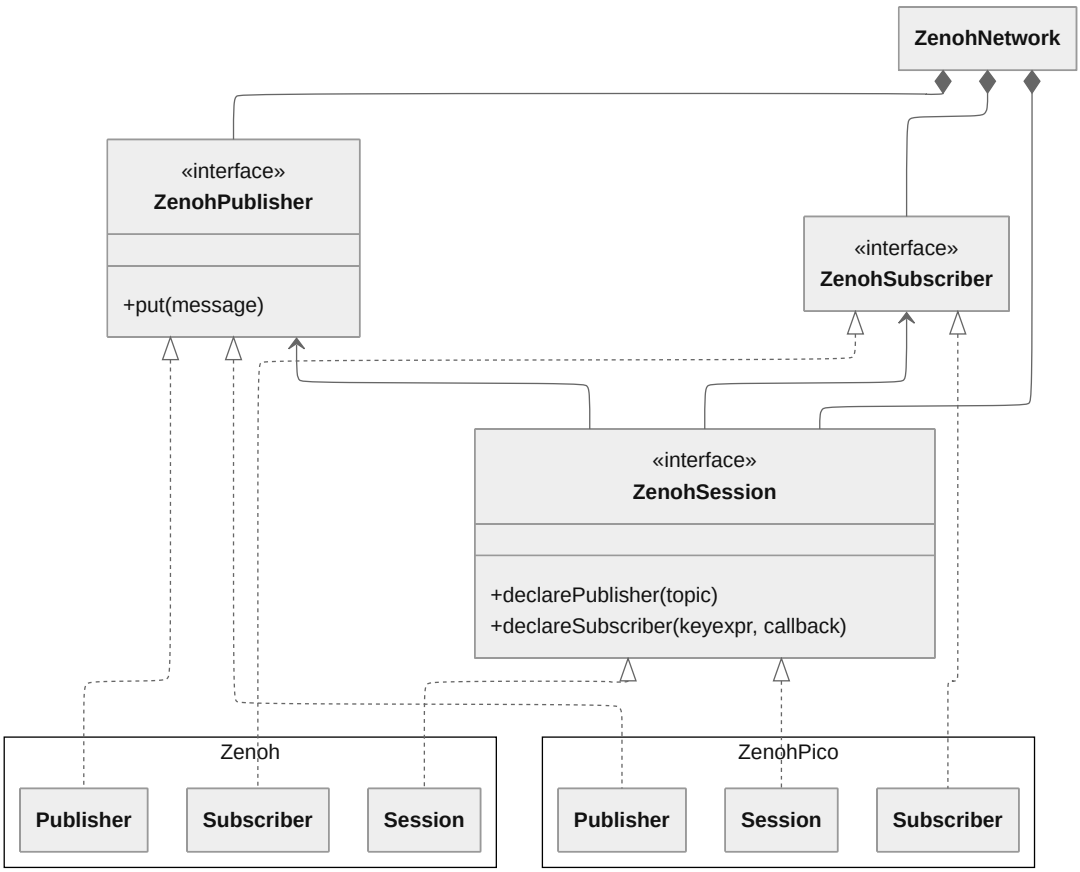

Figura 4.3: Schema dei componenti della `ZenohNetwork`. L'interfaccia per i subscriber non definisce metodi in quanto la funzione da chiamare alla ricezione di un messaggio viene fornita al momento della creazione tramite il parametro *callback* del metodo `ZenohSession.declareSubscriber`. Lo scopo di un oggetto di questo tipo è quello di rappresentare l'esistenza del subscriber in background.

4.2.3 L'interazione con YAAIR

YAAIR richiede che la rete esponga due operazioni, una per la pubblicazione del risultato del programma aggregato, ed una che restituisca una mappa associativa con l'ultimo risultato ricevuto da ogni nodo con cui si è in relazione (Sezione 2.1).

Mentre la pubblicazione del risultato si tratta di un'operazione banale tramite i costrutti di comunicazione *pub/sub* utilizzati dalla *ZenohNetwork*, la creazione della mappa associativa richiede l'eliminazione dal `MessagesStore` di eventuali

nodi offline per evitare di restituire dati ormai obsoleti (Figura 4.4).

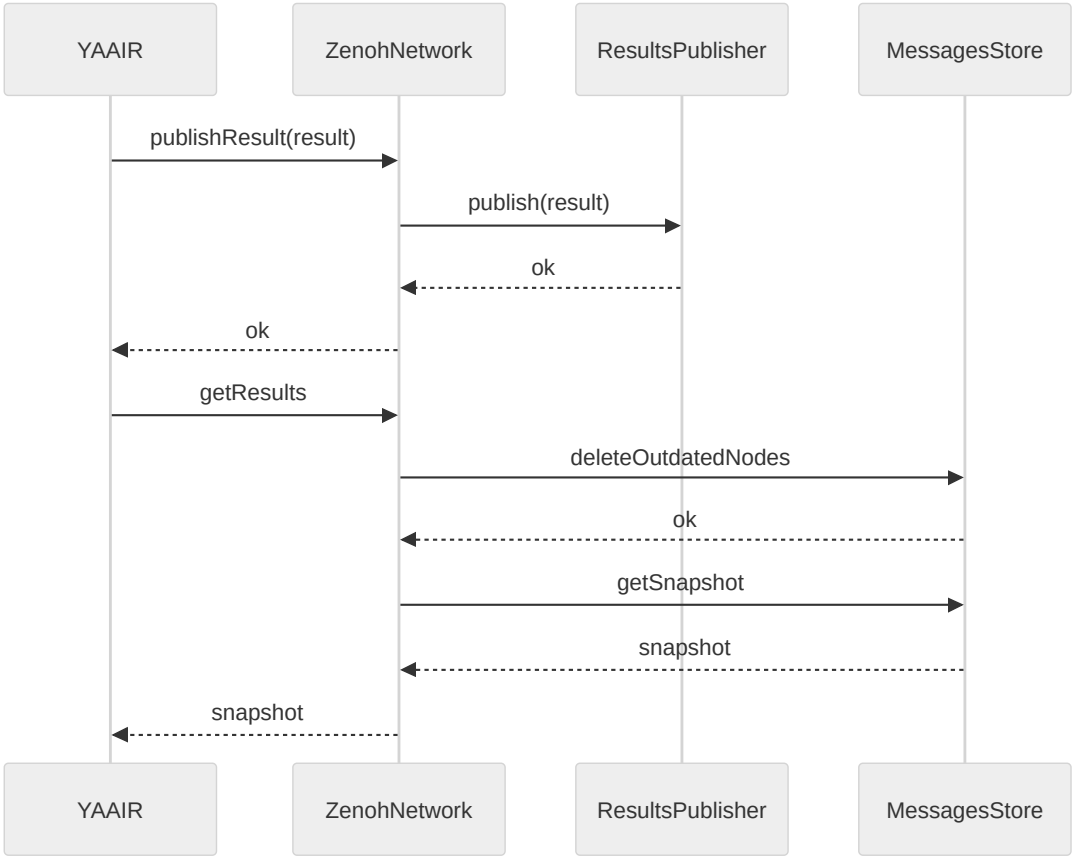

Figura 4.4: Flusso di interazione tra YAAIR e la `ZenohNetwork`. YAAIR richiede due operazioni alla rete: la pubblicazione di un nuovo risultato del programma, delegata direttamente al publisher apposito, e la lettura dei risultati degli altri nodi, che passa per il `MessagesStore`. Prima di restituire una fotografia dei risultati attuali, la rete elimina le voci scadute, garantendo che YAAIR riceva solo dati ancora validi.

La pubblicazione di aggiornamenti sullo stato dei nodi, invece, viene mantenuta esterna a YAAIR, tramite la possibilità di dichiarare publisher dedicati, in quanto completamente facoltativa.

Capitolo 5

Implementazione

In questo capitolo viene descritto come sono state implementate le varie parti del contributo di questa tesi, in maniera coerente con la progettazione svolta nel Capitolo 4.

5.1 Il wrapper di Zenoh pico

A differenza della versione standard di Zenoh, che offre una libreria nativa Rust, Zenoh pico dispone solo di una libreria C, in modo da poter sfruttare PlatformIO come strumento per supportare facilmente tutte le famiglie di dispositivi embedded più popolari. Per utilizzare questa versione di Zenoh è quindi necessario creare dei *binding* che permettano di interagire con la libreria tramite chiamate a funzioni inter-linguaggio.

La creazione di un wrapper Rust, inoltre, consente di nascondere le API C dietro strutture più adeguate e coerenti con lo stile di programmazione e con il modello di gestione della memoria propri del linguaggio.

5.1.1 Integrazione della libreria

Per semplificare l'integrazione della libreria e ridurre i tempi di sviluppo, il contributo di questa tesi limita i vincoli di compatibilità, nel contesto dei sistemi embedded, ai soli dispositivi della famiglia Espressif e compatibili (V1). Questa limitazione permette di sfruttare il framework ESP-IDF, che offre un sistema di componenti per

integrare librerie esterne nel processo di compilazione: definendo semplicemente i file sorgenti da considerare, questi vengono compilati con le impostazioni necessarie per avere accesso alle funzioni della libreria standard C offerte dalla piattaforma, per poi essere esposti pubblicamente come modulo all'interno del framework.

Nonostante Zenoh pico sia una libreria pensata per i sistemi embedded, la sua compatibilità è in realtà limitata a quelli che dispongono di un'implementazione, anche se parziale, della libreria standard C; risulta quindi necessario sfruttare questa funzionalità.

Per rendere poi accessibili le funzioni e i tipi del framework originale, ESP-IDF utilizza *bindgen*[1], uno strumento per la generazione automatica di binding Rust per le chiamate a funzioni di librerie esterne. In questo modo il programma può sfruttare un'unica libreria sia per interagire con il sistema tramite astrazioni di alto livello, sia per utilizzare le primitive C per un controllo più diretto, sia per accedere ai componenti aggiuntivi esposti tramite moduli dedicati (Listato 5.1).

Listato 5.1: Esempio di utilizzo del framework ESP-IDF in Rust: astrazioni di alto livello, primitive C e utilizzo di componenti extra tramite i moduli generati automaticamente da bindgen.

```
// The ESP-IDF Rust library exposes high-level abstractions of the hardware
// components with a single API that supports different devices.
let wifi = AsyncWifi::new(...);
wifi.connect().await;

// The original C primitives of the framework are also provided for a more
// fine-grained control over the system.
esp_idf_svc::sys::xTaskCreatePinnedToCore(...);

// The declared components bindings are also generated in dedicated
// modules.
esp_idf_svc::sys::zenoh_pico::z_open(...);
```

5.1.2 Le macro in Rust

Rust fornisce delle primitive di *metaprogrammazione* [Wik26] per la generazione di codice a tempo di compilazione. Alla base di questo sistema si trovano le *macro*, un costrutto concettualmente simile a una funzione, che riceve in input e produce in

[1] https://rust-lang.github.io/rust-bindgen/

output dei flussi di simboli (*token stream*) elaborati dal compilatore (Listato 5.2). Poiché l'espansione delle macro da parte del compilatore avviene prima della fase di validazione delle espressioni, il codice generato viene automaticamente incluso nei controlli dei tipi e di correttezza del programma, garantendo così lo stesso livello di sicurezza e robustezza di quello scritto manualmente.

Listato 5.2: Implementazione semplificata della macro `vec`, che inizializza un vettore con le espressioni fornite in input. [KNK26]

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}

let v = vec!(1, 2, 3);
// let v = {
//       let mut temp_vec = Vec::new();
//       temp_vec.push(1);
//       temp_vec.push(2);
//       temp_vec.push(3);
//       temp_vec
// };
```

Le macro si dividono in due categorie principali: dichiarative e procedurali. Le prime permettono di definire il codice da generare in maniera statica rispetto all'input, fornendo solo qualche costrutto di *pattern matching* sulla struttura delle espressioni in ingresso. Le macro procedurali, invece, sono effettive funzioni che operano su flussi di simboli, che operano su delle `TokenStream` interpretabili dal compilatore [KNK26]. Queste ultime vengono utilizzate come attributi da applicare a definizioni di simboli per modificarne la struttura o il comportamento. Un esempio sono le *derive macro*, un caso particolare di macro procedurali che permette di implementare automaticamente un *trait* per un tipo di dato.

Poiché la struttura `TokenStream` è molto primitiva e priva di operazioni di alto livello per la manipolazione e la generazione dei simboli, nel tempo sono nate

diverse librerie per semplificarne l'utilizzo, come *syn*[2] e *darling*[3], che permettono il parsing automatico e strutturato dell'input, oppure *quote*[4], che consente di generare l'output a partire da codice standard (Listato 5.3).

Listato 5.3: Esempio di implementazione di una derive macro. Vengono sfruttate le librerie syn, quote e darling per la manipolazione e la generazione dei simboli.

```rust
/// This is an example trait that we want to be able to implement automatically.
trait HelloWorld {
    fn hello_world();
}

// This struct describes the parameters of the optional customization attribute
// that can sit on top of the struct being derived, which darling populates
// accordingly.
#[derive(darling::FromDeriveInput, Default)]
struct HelloWorldOpts {
    ident: Ident, // special field containing the identifier of the derived struct
    greeting: Option<String>,
}

// This function is the entry point for the derive macro, which is called
// automatically by the compiler when the `HelloWorld` trait is derived for type.
// It also defines an associated `hello_world` attribute that can be used for
// extra options.
#[proc_macro_derive(HelloWorld, attributes(hello_world))]
fn hello_world_derive(input: TokenStream) -> TokenStream {
    // Here syn and quote can be used to parse the input tokens stream into an
    // AST and to generate the output tokens stream directly from code.
    let derive_input = syn::parse_macro_input!(input as syn::DeriveInput);

    let opts = HelloWorldOpts::from_derive_input(&derive_input);
    let ident = opts.ident;
    let greeting = opts.greeting.unwrap_or("Hello".to_string());
    let message = format!("{greeting} world from {ident}!");

    quote! {
        impl HelloWorld for #ident {
            fn hello_world() {
                println!(#message);
            }
        }
    }
}

```

[2]https://docs.rs/syn/2.0.118/syn/
[3]https://docs.rs/darling/0.23.0/darling/
[4]https://docs.rs/quote/1.0.45/quote/

```
// The derive attribute calls the associated procedural macro to implement the
// trait automatically.
#[derive(HelloWorld)]
// The `hello_world` attribute can be also added to override the default
// behaviour.
#[hello_world(greeting = "Good morning")]
struct Foo;

Foo::hello_world(); // output: Good morning world from Foo!
```

5.1.3 L'ownership model in Zenoh pico

Nonostante Zenoh pico sia una libreria distinta e indipendente da Zenoh standard, oltre a essere compatibile a livello logico, la sua API cerca di riprodurre la versione originale anche a livello strutturale nella maniera più fedele possibile. Questo comportamento è ottenuto fornendo non solo tipi e operazioni equivalenti, ma anche funzioni per la gestione della memoria secondo lo stesso modello di ownership che distingue Rust dagli altri linguaggi di basso livello. Sebbene questi vincoli sull'utilizzo delle risorse siano in realtà raggirabili, non essendo imposti direttamente dal compilatore, la libreria mette a disposizione funzioni e macro per emulare i concetti di proprietario, riferimento, spostamento dei valori e mutabilità esplicita, rendendo così la gestione manuale della memoria più semplice e sicura (Listato 5.4).

Listato 5.4: Esempio di gestione delle risorse in Zenoh pico [Zen26a].

```
z_owned_string_t s, s1;
z_string_copy_from_str(&s, "Hello, world!");
// notice that the prototype of z_string_clone is
// void z_string_clone(z_owned_string_t* dst, const z_loaned_string_t* src);
// I.e. the only way to pass the source string is by loaning it
z_string_clone(&s1, z_loan(s));
//...
z_drop(z_move(s));
z_drop(z_move(s1));
```

5.1.4 L'implementazione tramite macro

Unendo la struttura simil-Rust di Zenoh pico con i costrutti di metaprogrammazione offerti dal linguaggio, la creazione del wrapper per la libreria può essere quasi completamente automatizzata tramite generazione di codice. È stata quindi

implementata una macro, chiamata `zwrap` per rimanere coerente con il prefisso utilizzato dalla libreria, che permette di generare automaticamente strutture Rust contenenti il corrispondente tipo C e che implementano i trait necessari per la gestione delle risorse, integrando così il tipo generato nel modello di ownership del linguaggio (Listato 5.5).

Listato 5.5: Esempio di utilizzo di macro per la generazione automatica del wrapper Rust per Zenoh pico.

```
// Traits are defined to represent the memory management operations on the
// library side.
trait ZValue, ZOwn, ZClone, ...

// Since we need to modify the internal structure of the type to hold the
// corresponding wrapped value, we cannot implement the traits using just derive
// macros.
// Instead, a standard procedural macro is used that modifies the internal
// structure of the tagget type to transform it into a wrapper around a C type
// from the Zenoh pico library.
// Using the provided arguments, the automatic implementations of the traits can
// be configured: a base name must be provided to choose automatically the
// internal types and functions, then all traits that should be implemented must
// be listed explicitly, with the possibility to override the identifiers to use
// and to disable the auto implementation of the corresponding standard Rust
    traits.
#[zwrap(
    base(name = "string"),
    zvalue,
    zown(impl_drop = false), // (*)
    zclone(clone_zfn = some_other_clone_function)
)]
// The struct is expanded into a newtype `ZString(_z_string_t)` that wraps the C
// type and implements all the traits defined in the `zwrap` attribute arguments.
struct ZString;

// (*) The corresponding Rust trait can be later implemented manually to add extra
// functionality on top of the Zenoh pico library, accessible through the defined
// traits functions.
impl Drop for ZString { ... }
```

5.1.5 Il sistema di closure

Zenoh pico utilizza pervasivamente il concetto di chiusura per tutte quelle operazioni che forniscono un riscontro ritardato rispetto al momento dell'invocazione, ad esempio quando richiedono comunicazioni con gli altri nodi della rete, permettendo

così di operare in maniera asincrona senza bloccare il thread chiamante. Queste chiusure sono implementate come strutture che contengono tutti gli elementi necessari per fornire alla funzione associata un contesto mutabile a cui può accedere liberamente, rispettando sempre l'uso delle risorse e le regole di gestione della memoria utilizzate anche nel resto della libreria.

Nel wrapper Rust, oltre alla possibilità di utilizzare normalmente queste strutture, sono implementate funzioni che ne permettono l'uso anche tramite i costrutti per la programmazione concorrente `async/await`, offrendo così una API alternativa, moderna e versatile (Listato 5.6).

Listato 5.6: Esempio di utilizzo della API asincrona per le chiusure di Zenoh pico.

```rust
// The closure types are composed of a callback function and a context,
// that can be put behind an Atomically Reference Counted (Arc) smart pointer
// to allow sharing it in a thread-safe way.
// Any object using a closure can leverage an embassy `Signal` to get notified
// whenever a new value is received and provide a modern async/await API.
struct AsyncSubscriber {
    signal: Arc<Signal<Mutex, Sample>>,
    sample_closure: SampleClosure,
}

impl AsyncSubscriber {
    async fn recv_async(&self) -> T {
        self.signal.wait().await
    }
}

let subscriber = AsyncSubscriber::new();
loop {
    // This waits for the next value from the subscriber callback.
    let sample = subscriber.recv_async().await;
    // ...
}
```

5.2 Implementazione della rete

Un requisito fondamentale per il contributo di questa tesi è la compatibilità con YAAIR (V2), che definisce un'interfaccia da rispettare per implementare una rete utilizzabile dal framework. È quindi necessario analizzare le firme e il

comportamento atteso di queste operazioni, in modo da poter adeguare la rete di conseguenza.

Quest'interfaccia è definita da un trait `Network` composto da tre metodi (Listato 5.7):

- `get_local_id`: restituisce il valore identificativo del nodo corrente nella rete.
- `prepare_outbound`: prende in input l'ultimo risultato del programma aggregato, già serializzato sotto forma di sequenza di byte, e lo pubblica sulla rete per renderlo disponibile agli altri nodi presenti.
- `prepare_inbound`: restituisce un'istanza di una struttura dati di YAAIR, `InboundMessage`, che associa ogni nodo raggiungibile con l'ultimo messaggio da esso ricevuto.

Listato 5.7: Definizione del trait `Network` del framework YAAIR. È presente un argomento generico per il tipo dell'identificativo utilizzato dalla rete, il quale deve soddisfare diversi vincoli, tra cui l'implementazione dei trait di serializzazione e deserializzazione definiti dalla libreria `serde`.

```
pub trait Network<Id: Ord + Hash + Copy + Serialize + for<'de> Deserialize<'de>> {
    fn get_local_id(&self) -> Id;
    fn prepare_outbound(&mut self, outbound_message: Vec<u8>);
    fn prepare_inbound(&mut self) -> InboundMessage<Id>;
}
```

La `ZenohNetwork` implementata dovrà rispettare questo trait, sfruttando Zenoh e i suoi costrutti per la comunicazione con gli altri nodi. La rete mantiene un `MessagesStore` centrale per la memorizzazione dei messaggi, accessibile dai vari subscriber in background. Per i nodi della rete viene definito un tipo `ZenohNodeId`, costituito da una sequenza di 16 byte, utilizzato da tutti i componenti come identificativo sia interno che nel framework, tramite il parametro generico `Id` del trait `Network`.

5.2.1 Il sistema di messagistica

Al centro della comunicazione nell'implementazione troviamo il `MessagesStore`, una struttura che mantiene una mappa associativa dei nodi con l'ultimo risultato

da loro pubblicato, arricchito con metadati sull'istante in cui è stato inserito e la durata della sua validità.

Internamente utilizza una `HashMap` protetta da un mutex, per permettere un accesso sicuro anche da più thread paralleli, ed espone diversi metodi per interagire con essa: l'aggiunta di un risultato per un nodo (`store_message`), l'aggiornamento dell'istante di ultima attività (`keep_alive`) e della durata di validità di un risultato (`store_lifespan`), l'eliminazione dalla mappa di tutte le coppie chiave-valore contenenti risultati ormai obsoleti (`clear_dead`) e la creazione di una fotografia dei risultati attuali (`create_snapshot`).

Per un'architettura più versatile, il `MessagesStore` è implementato in maniera generica rispetto ai valori specifici utilizzati dalla rete, ed è configurabile tramite parametri di inizializzazione per definire opzioni come la durata di validità dei risultati predefinita.

Listato 5.8: Implementazione semplificata del `MessagesStore`, con lo scopo di mostrare un'approssimazione del suo funzionamento.

```
// A store entity associates with a value the metadata necessary for the
// expiration calculations.
// The timestamp is set to the current system time upon initialization, and each
// time the object is updated it is reset, making the check for the validity of
// the result a simple comparison between the time passed since the timestamp
// and the lifespan associated.
struct StoreEntity<T> {
    message: Option<T>,
    lifespan: Duration,
    timestamp: SystemTime,
}

struct MessagesStore<Id, T> {
    // The internal structure of the `MessagesStore` is an hash map behind a mutex
    // for thread safety.
    // Each operation that interacts with the values of the map automatically
    // updates the `StoreEntity` lifespan, keeping the node alive.
    storage: Mutex<HashMap<Id, StoreEntity<T>>>,
}
```

5.2.2 Zenoh e la comunicazione

Insieme al `MessagesStore`, la rete è composta da diversi subscriber che interagiscono con esso in maniera diretta secondo i messaggi ricevuti dai vari nodi e publisher

per comunicare nuovi risultati o aggiornamenti di stato sulla rete.

Tutte le entità Zenoh devono essere utilizzate attraverso delle interfacce per rendere la logica applicativa indipendente dalla versione utilizzata. Vengono quindi definiti diversi trait per rappresentare la sessione, i publisher, i subscriber e le key expressions, implementati sia da Zenoh standard che da Zenoh pico in appositi moduli. Questi vengono poi selezionati e utilizzati dal modulo principale della libreria in base alle caratteristiche del target di compilazione specificato tramite i costrutti di compilazione condizionale forniti dal linguaggio (Listato 5.9).

Listato 5.9: Definizione dei trait per le entità Zenoh utilizzate dalla rete.

```
// The session is the center of the communication, and its trait also defines
// the concrete types used by the Zenoh implementation.
trait ZenohSession {
    type Err, Config, KeyExpr, Subscriber, Publisher;

    // The nodes id type must be the same in all implementations for consistency
    // in serialization of messages sent between nodes using different versions
    // of Zenoh.
    fn node_id(&self) -> ZenohNodeId;

    fn init(zenoh_config: Self::Config) -> Result<Self, Self::Err>;
    fn declare_subscriber<T>(&self, keyexpr: Self::KeyExpr, callback: fn(T, &
        NetworkContext), context: Arc<NetworkContext>) -> Result<Self::Subscriber,
         Self::Err>;
    fn declare_publisher(&self, keyexpr: Self::KeyExpr) -> Result<Self::Publisher,
         Self::Err>;
}

// Other traits can access the implementation types through the Session trait.
trait ZenohKeyExpr, ZenohSubscriber, ZenohPublisher;

// The main library module that defines the `ZenohNetwork` can than import the
// correct Zenoh implementation based on the target characteristics.
#[cfg(target_os = "espidf")]
#[path = "zenoh_pico/mod.rs"]
mod zenoh_impl;

#[cfg(not(any(target_os = "espidf", target_os = "none")))]
#[path = "zenoh_standard/mod.rs"]
mod zenoh_impl;
```

5.2.3 L'integrazione con YAAIR

L'implementazione della `ZenohNetwork` è vincolata dal trait `Network` definito da YAAIR, in modo da rendere la rete utilizzabile dal framework e quindi dal programma aggregato. Sfruttando le varie entità Zenoh e il `MessagesStore`, vengono implementati i metodi necessari per rendere la comunicazione tra i nodi semplice ed efficace.

Quando la rete viene inizializzata, viene creata la sessione Zenoh, che rappresenta il nodo come peer per la comunicazione decentralizzata. Da questa sessione viene poi derivato l'identificativo del nodo all'interno della rete. Attraverso la sessione vengono poi dichiarate le entità per la comunicazione *pub/sub* dei risultati: un subscriber sulla key expression `*/messages` che ascolta i nuovi messaggi contenenti i risultati aggiornati di tutti i nodi della rete, ed un publisher corrispondente che pubblica questi messaggi tramite il metodo `prepare_outbound` sul topic `<nodeid>/messages/`, dove `<nodeid>` corrisponde ad una rappresentazione testuale dell'identificativo definito in precedenza.

Quando il subscriber riceve un messaggio da un altro nodo, ne deserializza il contenuto e aggiorna il `MessagesStore` con il nuovo risultato: nel momento in cui il metodo `prepare_inbound` viene chiamato da YAAIR, verrà fatta pulizia dei risultati obsoleti e creata un'istanza della mappa associativa tra i nodi ed il loro ultimo risultato a partire da una fotografia dello stato attuale (Listato 5.10).

Listato 5.10: Implementazione del trait `Network` per la rete Zenoh.

```
impl Network<ZenohNodeId> for ZenohNetwork {
    // The id of the node inside YAAIR is the same as in the Zenoh network.
    fn get_local_id(&self) -> ZenohNodeId {
        self.session.node_id()
    }

    fn prepare_outbound(&mut self, outbound_message: Vec<u8>) {
        // The new result is simply published on the network on the dedicated
            topic.
        self.messages_publisher.put_message(outbound_message)
    }

    fn prepare_inbound(&mut self) -> InboundMessage<ZenohNodeId> {
        // Get the `MessagesStore` instance used by the network.
        let messages = &self.context.messages;
        // Clear the obsolete results and create a snapshot of the current
        // state of the store.
```

```
17          let snapshot = match messages.clear_dead().and_then(|| {
18              messages.create_snapshot()
19          }) {
20              // If something fails, a default empty map is given to YAAIR.
21              Ok(s) => s,
22              Err(_) => return Default::default(),
23          };
24          // Create the YAAIR associative map from the snapshot of the current
25          // node results.
26          InboundMessage::new(snapshot)
27      }
28  }
```

I messaggi per la notifica manuale dell'esistenza del nodo e per l'aggiornamento della longevità dei risultati, invece, sono gestiti in maniera indipendente dal framework, tramite una comunicazione *pub/sub* analoga a quella usata per i risultati sul topic `heartbeat`. La rete offre la possibilità di dichiarare publisher appositi prima di passare l'istanza all'`Engine` del framework, così da lasciare libertà sulla strategia di utilizzo. Tutti i tipi, le operazioni e la logica applicativa associati a questa categoria di messaggi vengono definiti dietro una *feature* dedicata del crate, in modo da renderne l'integrazione completamente opzionale.

5.2.4 Configurazione ed utilizzo

La configurazione della rete avviene al momento dell'inizializzazione, tramite una struttura dedicata che definisce le opzioni disponibili. Nel caso della `ZenohNetwork`, questa si divide in due categorie principali: le opzioni per la rete e per il suo comportamento nel contesto di YAAIR, e la configurazione per la sessione Zenoh utilizzata internamente.

La configurazione di Zenoh avviene secondo il *builder pattern*, definendo un trait per tutte le opzioni comuni implementato da entrambe le versioni di Zenoh. Per tutte le opzioni specifiche di Zenoh standard o pico, invece, è possibile semplicemente continuare a modificare il valore di configurazione prodotto dal builder prima di fornirlo in input alla rete per l'inizializzazione. Inoltre, se sono presenti parametri obbligatori per il corretto funzionamento di una determinata versione di Zenoh, l'implementazione del builder può definire eventuali argomenti necessari per la sua creazione.

Per la configurazione della rete stessa viene usata una normale struttura che racchiude tutte le opzioni possibili. Nonostante sia buona norma fornire un'implementazione di default per le strutture che rappresentano collezioni di campi o opzioni di configurazione, in questo caso non è previsto, poiché è necessaria almeno la configurazione di Zenoh creata dal builder: come alternativa, sono stati implementati gli opportuni trait per convertire automaticamente la configurazione Zenoh in una per la rete, impostando i campi restanti con valori predefiniti.

Oltre alla sua configurazione, la rete necessita per l'inizializzazione anche di un serializzatore compatibile con quello usato da YAAIR per il programma aggregato, così da poter deserializzare correttamente i messaggi inoltrati sotto forma di sequenze di byte (Listato 5.11).

Listato 5.11: Esempio di configurazione ed utilizzo della rete.

```
// The config builder may require different initialization options based
// on the Zenoh version used.
let zenoh_config = ZenohConfigBuilder::with_default_options()
    // Shared options can be set from the builder, while implementation-specific
    // ones can be set directly on the configuration when built.
    .scouting_timeout(Duration::from_secs(15))
    .build();

// The default network config can be created from the Zenoh one, by converting it
// directly or by using it as a base.
let network_config = ZenohNetworkConfig {
     lifespan: Duration::from_secs(10),
     ..zenoh_config.into(),
};
let zenoh_network = ZenohNetwork::new(network_config, JsonSerializer);

// If the program needs some heartbeat publishers, they must be declared before
// giving the network to the `Engine`.
let heartbeat_publishers = zenoh_network.declare_heartbeat_publisher();

// And the network it's ready to be used in YAAIR.
// The serializer used by the engine must be the same of the network or a
// compatible one.
let engine = Engine::new(zenoh_network, JsonSerializer, ...);
```

Capitolo 6

Valutazione dei risultati

La natura distribuita della rete e il coinvolgimento di dispositivi embedded rendono difficile automatizzare in maniera efficace la validazione dei risultati. Per ridurre i tempi di sviluppo è stato quindi adottato un approccio più manuale: sono stati creati dei programmi specifici che utilizzano i vari componenti implementati coprendo diversi casi e piattaforme, per poi verificare manualmente che il risultato sia quello atteso. In questo capitolo, vengono presentati i test effettuati per verificare il corretto funzionamento delle componenti principali del contributo di questa tesi.

6.1 Il wrapper Zenoh pico: test del ping-pong

Il test del ping-pong[1] consiste nel far comunicare due nodi in un ciclo infinito: ad ogni iterazione il nodo attende di ricevere un nuovo valore sul topic opposto, lo modifica a piacere e lo pubblica sul suo topic dedicato, dove l'altro nodo sarà in attesa con una logica analoga. Per evitare situazioni di *deadlock* nella comunicazione, la partenza sarà asimmetrica, con un nodo che inizia inviando il valore e l'altro in attesa di ricevere (Figura 6.1).

[1] `https://github.com/lspita-academic/yaair-zenoh-network/tree/main/examples/ping-pong`

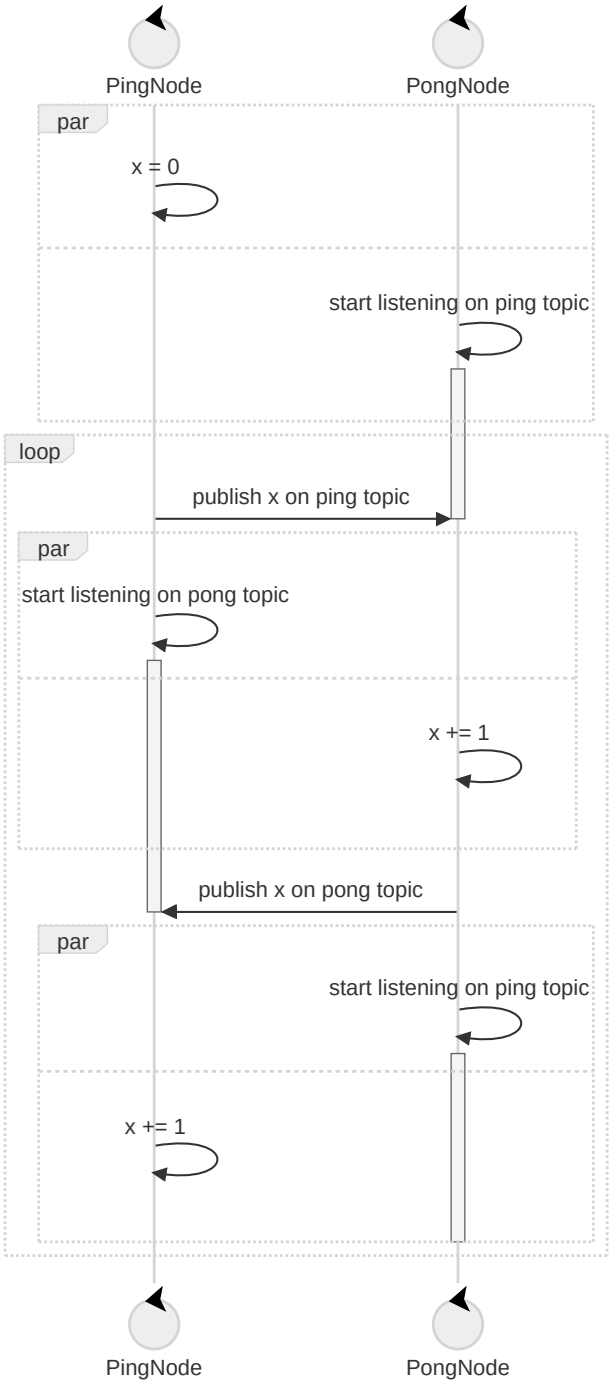

Figura 6.1: Diagramma di sequenza del test del ping-pong.

Obiettivi Questo test non utilizza né YAAIR né la rete per esso creata, bensì ha lo scopo di verificare il corretto funzionamento del wrapper per Zenoh pico anche al di fuori del contesto di un programma aggregato.

Implementazione Essendo un test per il wrapper di Zenoh pico, è stato pensato per i sistemi embedded che andranno poi ad utilizzare questa implementazione del protocollo all'interno della rete per YAAIR. Sono quindi previsti programmi distinti, eseguibili su dispositivi separati, rappresentanti i due diversi nodi descritti in precedenza.

Dopo una fase di inizializzazione per avviare la sessione Zenoh e dichiarare i publisher e subscriber necessari, il nodo `ping` imposta un valore `x=0`, mentre il nodo `pong` si mette in ascolto per nuovi messaggi sul topic `ping/value` sfruttando la API asincrona fornita dal wrapper. A questo punto inizia il ciclo infinito di scambio del valore:

1. il nodo `ping` pubblica il valore di `x` sul topic `ping/value`, e si mette in ascolto sul topic analogo `pong/value`.

2. il nodo `pong`, che era già in ascolto per nuovi valori sul topic `ping/value`, riceve il valore di `x`, lo incrementa e lo pubblica nuovamente sul topic `pong/value`.

3. Analogamente, il nodo `ping` riceve il nuovo valore aggiornato, lo incrementa a sua volta e ricomincia il ciclo infinito fino a quando non viene interrotto il programma manualmente.

Risultati Il test è stato eseguito su due ESP32-C3 Mini 1, dispositivi della famiglia Espressif dotati di un processore con architettura RiscV a 32 bit e di moduli per la connettività tramite WiFi. Ognuno di questi dispositivi esegue un solo programma tra `ping` e `pong`, caricato via cavo dalla macchina host.

Tramite i messaggi di log dei programmi in esecuzione è stato verificato che il valore viene trasferito tra un nodo e l'altro, di conseguenza la comunicazione *pub/sub* fornita dal wrapper Zenoh pico funziona correttamente.

6.2 La rete Zenoh: test del gradiente

Il test del gradiente[2] consiste in un'implementazione del noto esempio di programma aggregato per il calcolo di un gradiente spaziale tra i nodi della rete. Nel contesto di questo test, il calcolo delle distanze minime viene effettuato tra 3 nodi i cui collegamenti nel grafo della rete sono definiti in maniera statica, così da poter verificare facilmente la correttezza del risultato ottenuto.

Obiettivi L'obiettivo principale di questo test non è verificare la correttezza del programma aggregato stesso, il quale implementa un problema noto e facilmente risolvibile, bensì si tratta di valutare il corretto funzionamento della rete Zenoh quando utilizzata in un contesto reale da YAAIR, facendo interagire nodi di diverso tipo e verificando la correttezza dei messaggi scambiati.

Implementazione Come per il test del wrapper di Zenoh pico discusso nella sezione 6.1, anche per questo vengono definiti diversi programmi rappresentanti i vari nodi necessari per la comunicazione, così da poter verificare il supporto per l'interazione tra dispositivi eterogenei.

L'implementazione del gradiente è basata su quella fornita dagli esempi nella repository originale di YAAIR[3], modificata per implementare in maniera statica e prevedibile le distanze tra i vari nodi secondo il grafo rappresentato dalla fig. 6.2.

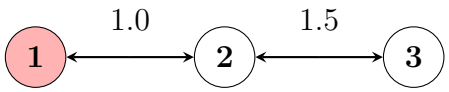

Figura 6.2: Gradiente spaziale utilizzato per il test della rete Zenoh. Gli archi rappresentano i collegamenti tra i vari nodi, mentre i valori su di essi la distanza tra i nodi. Il nodo sorgente è evidenziato in rosso. Il risultato atteso per ogni nodo corrisponde alla somma degli archi per arrivare ad esso dalla sorgente tramite il percorso più breve, ovvero 0 per la sorgente, 1 per il nodo 2 e 2.5 per il nodo 3.

[2] `https://github.com/lspita-academic/yaair-zenoh-network/tree/main/examples/gradient`

[3] `https://github.com/nicolasfara/yaair/blob/master/examples/gradient.rs`

Risultati Il test è stato eseguito con un dispositivo diverso per ogni nodo, per verificare il supporto della rete per la comunicazione tra sistemi eterogenei: il primo è rappresentato da un ESP32-C3 Mini 1, lo stesso utilizzato nella sezione 6.1, il secondo è una Raspberry Pi Zero W, che rientra nella categoria di sistemi embedded che utilizzano la versione standard di Zenoh in quanto dotati di un sistema operativo compatibile, ed infine il terzo è una macchina Linux standard, che rappresenta un dispositivo general-purpose.

Grazie ai messaggi di log dei programmi in esecuzione è stato verificato che i dispositivi si sono identificati nella rete autonomamente senza connessioni dirette, per poi correttamente condividere i risultati del gradiente ad ogni iterazione, ottenendo i valori finali previsti di 0 per il nodo 1 sorgente, 1 per il nodo 2 e 2.5 per il nodo 3, dato dalla somma di tutti gli archi percorsi. Inoltre, si possono anche identificare gli istanti in cui sono stati utilizzati dei messaggi di heartbeat per notificare la propria presenza nella rete. In particolare, dopo aver interrotto un nodo manualmente i suoi risultati sono diventati obsoleti e di conseguenza eliminati dalla rete.

Capitolo 7

Conclusioni

Questo capitolo conclude la tesi riprendendo gli obiettivi fissati e valutandone il raggiungimento alla luce del lavoro di progettazione, implementazione e validazione svolto nei capitoli precedenti. Vengono inoltre proposti alcuni possibili sviluppi futuri del progetto, volti ad ampliarne le funzionalità o a colmare le limitazioni emerse durante lo sviluppo del contributo.

In questo progetto è stata realizzata una rete di comunicazione decentralizzata basata su Zenoh, pensata per permettere a YAAIR di funzionare su dispositivi eterogenei. L'obiettivo principale era capire a fondo come funziona Zenoh, renderlo utilizzabile anche sui dispositivi più limitati e costruire intorno ad esso una rete decentralizzata affidabile che possa fornire un sistema di comunicazione per i programmi aggregati.

Per riuscire a far funzionare Zenoh anche sui microcontrollori, che dispongono solo di una versione della libreria limitata sviluppata in C, è stato necessario crearne un wrapper, in modo da poterci interagire tramite una API moderna che segue i paradigmi di programmazione del linguaggio Rust. A partire da questo, è stata costruita la rete vera e propria, capace di scegliere autonomamente quale implementazione di Zenoh utilizzare in base alle caratteristiche del dispositivo per cui viene compilata, così da mantenere l'esecuzione del programma aggregato da parte di YAAIR agnostica alla propria piattaforma e quella dei diversi nodi partecipanti. La rete si occupa anche di tenere traccia dei messaggi ricevuti e di capire quando uno di essi non è più raggiungibile, scartandone i risultati ormai

obsoleti.

La verifica del corretto funzionamento del contributo è stata effettuata tramite dei test sperimentali. Per quanto non possano essere valutati in maniera automatica, quest'approccio ha ridotto drasticamente i tempi di sviluppo del progetto, in quanto l'automatizzazione della verifica dei risultati richiederebbe l'integrazione di ambienti virtualizzati per simulare la presenza di nodi distinti sulla rete e la comunicazione tra di loro. Come alternativa, è stata prestata molta attenzione nella creazione dei problemi risolti in questi programmi, imponendo dei vincoli che permettessero di definire facilmente i risultati attesi e permetterne il confronto manuale con quelli ottenuti.

7.1 Sviluppi futuri

Sebbene il contributo presentato soddisfi tutti gli obiettivi prefissati, il lavoro svolto apre la strada a diversi possibili sviluppi futuri.

Test di performance e benchmark energetici su hardware reale. I test condotti in questa tesi si sono concentrati sulla correttezza funzionale della rete; un'analisi più approfondita delle prestazioni e del consumo energetico su dispositivi embedded reali permetterebbe di valutare l'impatto della rete in scenari con vincoli energetici stringenti, tipici di molte applicazioni IoT. Un benchmark di questo tipo avrebbe un duplice scopo: da un lato verificare che l'overhead introdotto dal wrapper e dal `MessagesStore` non comprometta le proprietà di basse latenze ed elevato throughput che motivano la scelta di Zenoh; dall'altro fornire dati concreti su consumo energetico e occupazione di memoria, dimensioni critiche per i microcontrollori utilizzati per l'esecuzione di programmi aggregati. Una strategia per determinare queste metriche potrebbe essere l'esecuzione dei programmi di test già implementati con carichi e frequenze crescenti.

Ulteriori verifiche potrebbero essere effettuate variando la configurazione della rete stessa, introducendo dei router Zenoh o modificando parametri come il numero di nodi partecipanti e la dimensione e frequenza dei messaggi. Confrontando i risultati ottenuti con quelli di un protocollo di riferimento come MQTT, si potrebbe quantificare concretamente i vantaggi, in termini di prestazioni e di assenza di un

singolo punto di fallimento, dell'architettura decentralizzata proposta rispetto ad una tradizionale basata su broker.

Indipendenza completa dall'ambiente di esecuzione. Il progetto `zenoh-nostd`, precedentemente menzionato nella sezione 2.3.3, propone una versione di Zenoh priva di dipendenze esterne. A differenza di Zenoh pico, che è scritto in C e richiede quindi il wrapper sviluppato in questa tesi per essere integrato, `zenoh-nostd` è implementato completamente in Rust e pensato per ambienti *bare-metal* vincolati da risorse limitatissime, in quanto non utilizza né la libreria standard né l'allocazione dinamica della memoria. Una soluzione di questo tipo permetterebbe di eliminare interamente la necessità di un livello di interoperabilità tra Rust e C, rimuovendo così sia l'overhead introdotto dal wrapper sia l'intera superficie di codice non sicuro richiesta per garantire la compatibilità fra i due modelli di ownership.

Trovandosi attualmente in una fase sperimentale, non è stato possibile utilizzarlo per il contributo di questa tesi, ma una volta raggiunta una maggiore stabilità, la sua adozione permetterebbe di ottenere una soluzione Rust pura funzionante su qualsiasi tipo di dispositivo, in quanto completamente autosufficiente. Un possibile sviluppo futuro consisterebbe quindi nel monitorare la maturazione di questa libreria e, una volta raggiunto un livello di stabilità e copertura del protocollo sufficiente, nello sviluppare un nuovo strato di interfacciamento con la rete che utilizzi questa implementazione. Le caratteristiche di questa versione di Zenoh permetterebbero di scegliere liberamente se mantenere questa implementazione come unica base per la comunicazione nella rete o se utilizzarla come terza alternativa insieme a Zenoh standard e pico.

Esplorazione di strategie alternative per la gestione della perdita di nodi. Il requisito non funzionale RNF2 prevede la possibilità di adottare strategie diverse per la rilevazione dell'uscita di un nodo dalla rete, a seconda del contesto applicativo. L'implementazione della rete attuale utilizza una strategia basata su una semplice scadenza configurabile, mentre sviluppi futuri potrebbero esplorare meccanismi più sofisticati, ad esempio per scenari con particolari esigenze di affidabilità o di risparmio di banda.

Una direzione concreta sarebbe l'adozione di un algoritmo di identificazione del fallimento di un nodo di tipo cumulativo, come il *φ-accrual failure detector* proposto da Hayashibara et al. [HDYK04]: invece di classificare un nodo come irraggiungibile sulla base di una singola soglia fissa, questo tipo di meccanismo calcola un valore continuo di sospetto a partire dalla distribuzione storica degli intervalli tra gli heartbeat ricevuti da un nodo, adattandosi così automaticamente alla variabilità della rete e permettendo all'applicazione di scegliere la propria soglia di sensibilità in base al contesto. Questo algoritmo ha già avuto occasione di dimostrare la sua robustezza ed efficacia essendo utilizzato in sistemi distribuiti decentralizzati molto popolari come Cassandra[1] e Akka[2].

Un'alternativa altrettanto valida riguarda l'uso di tecniche di tipo *gossip* per la propagazione delle informazioni di stato: invece di basarsi solo su heartbeat diretti tra coppie di nodi, ogni nodo potrebbe scambiare periodicamente con i propri vicini anche le informazioni che ha raccolto sullo stato degli altri nodi della rete, riducendo i falsi positivi dovuti a perdita occasionale di singoli messaggi e migliorando la scalabilità del meccanismo di rilevazione al crescere del numero di nodi. Zenoh utilizza già dei messaggi di *gossip* per la scoperta automatica dei nodi sulla rete, di conseguenza una strategia simile potrebbe essere facilmente integrata nel progetto utilizzando le funzionalità già offerte dal protocollo.

[1] `https://cassandra.apache.org/`
[2] `https://akka.io/`

Bibliografia

[ABB+25] Gianluca Aguzzi, Lorenzo Bacchini, Martina Baiardi, Roberto Casadei, Angela Cortecchia, Davide Domini, Nicolas Farabegoli, Danilo Pianini, and Mirko Viroli. A demonstrator for self-organizing robot teams. In Cinzia Di Giusto and António Ravara, editors, *Coordination Models and Languages*, pages 230–244, Cham, 2025. Springer Nature Switzerland.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[HDYK04] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The /spl phi/ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 66–78, 2004.

[KNK26] Steve Klabnik, Carol Nichols, and Chris Krycho. *The Rust Programming Language*. No Starch Press, 3 2026.

[PE24] Shane Panter and Nasir Eisty. Rusty linux: Advances in rust for linux kernel development. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '24, page 496–502, New York, NY, USA, 2024. Association for Computing Machinery.

[Ric] Nicholas Ricci. *Controllo di sciami robotici in ambiente ibrido fisico-virtuale: un dimostrativo*. PhD thesis.

[SCG+25] Leon Schuermann, Brad Campbell, Branden Ghena, Philip Levis, Amit Levy, and Pat Pannuto. Tock: From research to securing 10 million

computers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 36–49, New York, NY, USA, 2025. Association for Computing Machinery.

[Wik26] Wikipedia contributors. Metaprogramming — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Metaprogramming&oldid=1335302027`, 2026. [Online; accessed 20-June-2026].

[Zen26a] Zenoh contributors. Concepts — zenoh pico 1.9.0 documentation. `https://zenoh-pico.readthedocs.io/en/1.9.0/concepts.html`, 2026. [Online; accessed 21-June-2026].

[Zen26b] Zenoh contributors. What is zenoh? — zenoh. `https://zenoh.io/docs/overview/what-is-zenoh/`, 2026. [Online; accessed 11-June-2026].