Progetto e implementazione di un sistema distribuito per il coordinamento di droni

Esame di Sistemi Distribuiti

Belliato Riccardo
Lena Emanuele

A.A. 2021-22

Sommario

In questo documento si tratta la proposta (e l'implementazione) di una soluzione al problema del coordinamento di una flotta di droni per le consegne. In particolare, si vuole realizzare un sistema distribuito che permettaa ai droni di coordinarsi in autonomia allo scopo di prevenire potenziali collisioni in volo.

La prima parte di questo elaborato (capitoli 1, 2 e 3) ha come obiettivo la trattazione del problema dal punto di vista progettuale e algoritmico. In questa parte si descrive il problema e si propone una soluzione distribuita, in particolare come questa affronta le varie criticità, quali: la gestione dell'ingresso di un nuovo nodo, la negoziazione per partenza, la definizione di criteri ragionevoli di priorità, la gestione della starvation e del deadlock.

Nella seconda parte dell'elaborato (capitoli 4, 5 e 6) si descrive l'implementazione del sistema con il linguaggio C# e con il framework Akka.NET e la sua validazione mediante l'interfaccia utente da terminale e la suite per i test automatici XUnit. I risultati del lavoro svolto sono disponibili nella repository del progetto: `https://github.com/rik1599/DistributedSystemsExam`

Indice

1 Introduzione **4**
- 1.1 Descrizione del problema 4
  - 1.1.1 Note sul problema e assunzioni di partenza 4
- 1.2 Descrizione generale della soluzione 5

2 Analisi **6**
- 2.1 Requisiti funzionali 6
- 2.2 Requisiti non funzionali 6

3 Progetto **7**
- 3.1 Architettura logica 7
  - 3.1.1 Rete peer-to-peer di droni 7
  - 3.1.2 Il registro dei nodi 8
  - 3.1.3 Altre componenti secondarie (tracciamento delle missioni) 8
- 3.2 Protocolli e algoritmi 8
  - 3.2.1 Note e convenzioni sullo pseudocodice 8
  - 3.2.2 Protocollo di ingresso 9
  - 3.2.3 Protocollo di negoziazione 11
  - 3.2.4 Calcolo delle collisioni e dalla distanza da esse 15
  - 3.2.5 Metrica di priorità e note sulla negoziazione 18
  - 3.2.6 Uscita dei nodi dalla rete, timeout e gestione dei crash . . 19
  - 3.2.7 Altri algoritmi e protocolli 20
- 3.3 Architettura fisica e dispiegamento 20
  - 3.3.1 Architettura fisica e note sulla rete 20
  - 3.3.2 TCP vs UDP 21
  - 3.3.3 Linguaggi e tecnologie 21
  - 3.3.4 Procedure di dispiegamento 23
- 3.4 Note sulle trasparenze e sul CAP Theorem 23
  - 3.4.1 Nota sul CAP Theorem e sulla disponibilità del servizio . 23
  - 3.4.2 Trasparenze 24
- 3.5 Piano di sviluppo 25

4 Sviluppo **27**
4.1 Tier 1: implementazione dei protocolli principali 27
4.1.1 Attore del drone 27
4.1.2 Gli stati del drone e la gestione dei messaggi 29
4.1.3 Strutture dati (missione, priorità, ConflictSet, ecc.) 32
4.1.4 Meccanismo di attesa delle missioni in volo 34
4.1.5 Simulazione del processo di volo 35
4.1.6 Terminazione della missione 36
4.1.7 Spawn dei droni e procedura di pre-inizializzazione tramite registro dei nodi 36
4.1.8 Spawn concreto dell'attore e problema della supervisione . 37
4.2 Tier 2: Gestione degli errori 39
4.3 Tier 3.A e 3.C: Interfacciamento con il sistema e servizi per l'utente 39
4.3.1 Predisposizione messaggi per comandi utente 40
4.3.2 Meccanismi per l'osservazione dello stato delle spedizioni 40
4.3.3 API Object Oriented per il programmatore dell'interfaccia 41
4.3.4 Interfaccia utente da terminale 43

5 Validazione del sistema **45**
5.1 Test unitari con XUnit 45
5.1.1 Testing automatizzato applicato al contesto degli attori . 45
5.1.2 Casi di test sugli attori 45
5.1.3 Casi di test sulle API 46
5.1.4 Commenti sul testing automatizzato 47
5.2 Interfaccia console per il prototipo 48
5.2.1 Come usare l'interfaccia 48
5.2.2 Test con interfaccia Nro 01: Spawn remoto (fig. 5.1 e 5.2) 49
5.2.3 Test con interfaccia Nro 02: attesa di nodo in volo (fig. 5.3) 49

6 Conclusioni **53**
6.1 Risultati ottenuti 53
6.1.1 Tier 1: protocolli e algoritmi fondamentali (COMPLETATO) 53
6.1.2 Tier 2: Gestione degli errori (COMPLETATO) 53
6.1.3 Tier 3: Servizi per l'utente (PARZIALMENTE COMPLETATO) 54
6.1.4 Tier 4: Sistema di interfacciamento aperto (CAMBIATO IN CORSO) 54
6.1.5 Sviluppo della console 55
6.2 Criticità e idee di miglioramenti futuri 55

A Note didattiche **57**
A.1 Alcune note sui messaggi in Akka.NET 57

B Esempio di utilizzo delle API per avviare e monitorare una missione **58**
B.1 Spawn della missione su un nodo remoto 58
B.2 Osservazione di una missione dispiegata su un nodo remoto . . . 59

Capitolo 1

Introduzione

1.1 Descrizione del problema

Il problema affrontato si può riassumere con quanto riportato di seguito.

Si ha una flotta di droni (volanti), che devono spostarti in una certa area geografica per portare a termine delle consegne. I droni del caso sono veicoli autonomi piuttosto semplici, in quanto per portare a termine una consegna si limitano a decollare da un certo punto di partenza e volare in linea retta fino alla destinazione prestabilita (mantenendo velocità e altezza costanti e uguali per tutti i droni). Non si prevede che i droni siano capaci di fare cose complesse come regolare la propria velocità, cambiare direzione mentre sono in volo oppure rilevare oggetti tramite sensori; le loro uniche capacità sono la misurazione della propria posizione e la comunicazione con gli altri velivoli tramite una qualche forma di connessione wireless.

Lo scopo del progetto è realizzare un sistema elastico e distribuito che sfrutti le capacità di comunicazione dei droni per permettere a loro di decidere in autonomia chi deve partire e quando per evitare le collisioni; il tutto evitando di fare affidamento ad un sistema centralizzato (ad es., unica unità di controllo centrale).

Oltre che evitare le collisioni, altre sfide da affrontare sono la prevenzione del *deadlock*[1] e della *starvation*,[2] oltre che la minimizzazione dei tempi di attesa.

1.1.1 Note sul problema e assunzioni di partenza

- Visto che l'altezza di volo è unica per tutti i droni e visto che non si prevede che l'area geografica sia troppo estesa, lo spazio del problema può essere visto come un piano cartesiano (un classico spazio - non curvo - modellabile da due dimensioni).

[1] **deadlock**: situazione nella quale i droni si trovano in una situazione di stallo a causa di un ciclo nel grafo delle precedenze.

[2] **starvation**: uno o più droni non partono o aspettano troppo a causa del continuo ingresso di nuove missioni di priorità maggiore.

- Per questo progetto, si assume che ogni drone sia capace di comunicare con tutti gli altri (tramite un canale diretto oppure tramite una rete commutata).
- Nella proposta di soluzione, si vuole quanto più possibile minimizzare i tempi di attesa complessivi delle missioni.
- Nella proposta di soluzione, si vuole includere anche una qualche forma di gestione dei guasti. Come "guasti" si intendono sia problematiche rilevabili (che portano ad un atterraggio di emergenza), sia problematiche non rilevabili (come lo schianto di un drone o la perdita di connessione).

Si noti infine che non vengono trattati in questo elaborato questioni come:

- la gestione di una procedura di prenotazione e "scelta" del drone esatto che deve svolgere una certa missione;
- la gestione dello spostamento di un drone dalla sua posizione corrente fino alla partenza della consegna.

Per questo progetto, si può assumere che un drone "appaia" fisicamente nello spazio bi-dimensionale al momento della partenza e "scompaia" al momento dell'atterraggio.

1.2 Descrizione generale della soluzione

La soluzione proposta si basa sul far comunicare tra di loro i droni *prima della partenza*, in modo tale da permettere loro di venire a conoscenza delle tratte altrui e calcolare eventuali collisioni.

Per evitare le collisioni, l'idea è che ogni gruppo di droni "in conflitto" decida in autonomia chi può partire subito attraverso un *protocollo di negoziazione*. Tale negoziazione si prevede avvenga tramite lo scambio di una metrica, che tiene conto sia del tempo che drone ha atteso fino adesso, che dell'attesa che la sua vittoria causerebbe agli altri.

L' assenza di collisioni può essere garantita modellando il problema tramite un grafo (dove ogni drone rappresenta un nodo e ogni conflitto un arco) e ricavando da esso (in maniera distribuita) un *insieme indipendente* di nodi che possono partire subito. Più in generale, si vuole ricavare dal grafo dei conflitti un *grafo diretto ed aciclico delle dipendenze* che stabilisce "chi deve attendere chi" per partire.

Capitolo 2

Analisi

2.1 Requisiti funzionali

Requisito	Descrizione	Input	Output
Nuova missione	Viene istanziato un drone per la consegna. La consegna si considera avvenuta quando il drone raggiunge il punto stabilito.	2 coppie di coordinate (x_{start}, y_{start}) e (x_{end}, y_{end})	Messaggio di avvenuta consegna oppure errore
Richiesta posizione drone	Un utente chiede la posizione attuale e lo stato di un drone	ID drone	Coordinate (x, y) attuale del drone e se questo è a terra o in volo

2.2 Requisiti non funzionali

Requisito	Descrizione
Coordinamento droni	I droni devono coordinarsi da soli (possibilmente senza coinvolgere nodi centralizzanti) per decidere chi può volare e quando in modo che questi non si scontrino in volo. Un drone in volo non può essere fermato.
Movimento droni e localizzazione	I droni si muovono in linea retta a velocità costante v, tutti alla stessa altezza e in un area geografica limitata (ad esempio ad un area urbana). Per questo motivo lo spazio di movimento si può modellare come un piano a due dimensioni. I droni sono in grado di conoscere la loro posizione nello spazio.
Minimizzazione tempi	Le consegne devono essere completate nel minor tempo possibile.
Vincoli sulle attese	Vanno assolutamente evitate situazioni quali *deadlock* e *starvation*.
Vincoli sulle nuove missioni	Una nuova missione può essere istanziata in qualsiasi momento. Non sono presenti vincoli sul numero di droni attivi contemporaneamente.
Comunicazione tra droni	Ogni drone può comunicare con tutti gli altri attraverso una layer di rete sottostante. Non è richiesto l'inoltro di messaggi attraverso i droni.
Fault tolerance	I droni possono guastarsi mentre sono in volo. Esistono due tipi di guasto: - guasto *non rilevabile* (il drone si disattiva senza comunicare) - guasto *rilevabile* (il drone segnala il malfunzionamento) Un guasto, possibilmente, non deve compromettere il completamento delle altre missioni.

Capitolo 3

Progetto

3.1 Architettura logica

L'architettura logica del sistema consiste in una rete *peer-to-peer*, affiancata da alcuni processi separati che offrono servizi di supporto per il sistema e per gli utenti.

3.1.1 Rete peer-to-peer di droni

La componente principale del sistema è una rete peer-to-peer, dove ad ogni drone corrisponde un nodo. La topologia della rete è un grafo completo, pertanto si prevede che ogni nodo possa comunicare con tutti gli altri tramite un canale logico bi-direzionale.

Il modello logico della rete è da considerarsi un *puro peer-to-peer*, pertanto ogni nodo svolge i medesimi ruoli. I compiti principali dei nodi sono:

- comunicare il proprio ingresso nella rete a tutti i nodi esistenti;
- comunicare la propria uscita al completamento della missione (oppure in caso di guasto osservabile);
- comunicare le proprie intenzioni (la tratta che si intende percorrere) e negoziare con gli altri le precedenze in caso di conflitti;
- "spegnersi" nel caso in cui ci si accorge di non poter completare la missione a causa di problemi di comunicazione o altri guasti rilevabili.

A tali compiti primari si possono affiancare la fornitura di altri servizi secondari, come la trasmissione in tempo reale della propria posizione per finalità di tracciamento, oppure la possibilità di effettuare snapshot della rete su richiesta di un utente privilegiato.

3.1.2 Il registro dei nodi

La seconda componente del sistema (l'unica centralizzata) è un registro dei nodi. Il *registro dei nodi* è un server (dall'indirizzo noto) che ha il compito di fornire ai droni una lista dei nodi attualmente attivi nella rete in fase di inizializzazione.

Ciascun drone, quando entra nella rete, contatta il registro per reperire una lista aggiornata di tutti i nodi presenti e - allo stesso tempo - comunica il proprio ingresso. Il registro poi in qualche modo monitora l'uscita dei droni, così da mantenerne sempre la propria lista aggiornata per i nuovi arrivati.

In alternativa al registro, si può immaginare una sorta di *processo ambiente*, che, invece di essere contattato dai droni, istanzia tutti i processi e li inizializza con una lista aggiornata degli altri processi già presenti. Tra le due soluzioni, si è scelta l'opzione del registro in quanto:

- semplifica e rende più elastica la procedura di spawn dei processi sui nodi;
- rende potenzialmente più semplice pensare a sviluppi futuri dove il server centrale monolitico viene trasformato in un cluster di processi ridondanti.

3.1.3 Altre componenti secondarie (tracciamento delle missioni)

Si può immaginare di introdurre nel sistema altri componenti secondarie finalizzate ad offrire servizi per l'utente, come ad esempio il tracciamento delle missioni.

Inizialmente, per offrire un servizio del genere si è pensato a inserire uno o più server ai quali i droni trasmettono in tempo reale il proprio stato. Ragionando in ottica più distribuita, si è però concluso che sarebbe più sensato creare per ogni missione un servizio di "notifica su richiesta". Tale servizio può essere immaginato come un sotto-processo al quale degli "osservatori" si "iscrivono" per ricevere uno stream di notifiche.

Per implementare un sistema del genere si potrebbe utilizzare protocolli *publish-subscribe* (ad es., basati su un broker e su un sistema di eventi), oppure implementare meccanismi ad hoc ispirati al *pattern dell'Observer* [3].

3.2 Protocolli e algoritmi

3.2.1 Note e convenzioni sullo pseudocodice

Si elencano ora alcune indicazioni per la lettura dello pseudocodice e dei diagrammi di stati dei vari algoritmi e protocolli.

- Ogni missione è caratterizzata da alcuni elementi di base quali:
  - il proprio identificatore univoco ID;
  - il timestamp t_{spawn} dell'istante in cui il nodo entra nella rete e inizia a comunicare con gli altri (dal quale si può ricavare in ogni momento l'età del nodo AGE);

  - la propria tratta *path*, composta da un punto di partenza *path.start* e un punto di arrivo *path.end* (entrambi rappresentati da una coppia di coordinate x, y).

  Ogni drone (quindi ogni nodo) ha la responsabilità di tenere traccia di queste informazioni.

- Tutti i nodi, dal momento che iniziano a comunicare con gli altri, si caratterizzano da uno *stato interno* (*state*), che può essere:

  - $Init$
  - $Negotiate$
  - $Waiting$
  - $Flying$

  Lo stato del nodo determina come esso reagisce alle varie tipologie di messaggi scambiati. Le dinamiche delle transazioni tra stati sono descritte (ad alto livello) dal diagramma della figura 3.1.

- Un aspetto chiave del problema è la modellizzazione dei conflitti. Ogni nodo può modellare i propri conflitti definendo una serie di insiemi.

  1. Il $ConflictSet$ è l'insieme di nodi (non in volo) la cui tratta si incrocia con quella della missione e con i quali si deve negoziare. Concettualmente, si può partizionare in due sotto-insiemi:
     (a) un $GPTMSet$ (*Greater-Priority-than-Me-Set*), contenente tutti i nodi la cui priorità (nota o assunta) è superiore alla mia;
     (b) un $SPTMSet$ (*Smaller-Priority-than-Me Set*), contenente tutti i nodi la cui priorità (nota o assunta) è inferiore alla mia.

     Come si spiega nelle sotto-sezioni 3.2.3 e 3.2.5, è essenziale che ogni nodo abbia una visione coerente delle priorità, o al massimo entrambi i nodi assumano di avere priorità inferiore all'altro.
  2. Un $FlyingSet$ contenente tutti i nodi che si sanno essere già in volo.

  Una volta che si scopre che c'è un incrocio con la tratta di un altro nodo, esso deve per forza essere inserito in uno di questi insiemi. Allo stato attuale si può invece assumere che si possa "dimenticare" un nodo quando le tratte non sono in conflitto, oppure quando un nodo in volo ha raggiunto un "punto sicuro".

3.2.2 Protocollo di ingresso

Il primo primo problema da affrontare è l'ingresso di un nodo della rete. Come viene spiegato anche nella sotto-sezione 3.4.1, per garantire l'assenza di collisioni è indispensabile che ogni nodo conosca e comunichi (almeno una volta) con tutti gli altri nodi della rete; pertanto, il protocollo di inizializzazione svolge un ruolo chiave. Di seguito, si definiscono le meccaniche del protocollo.

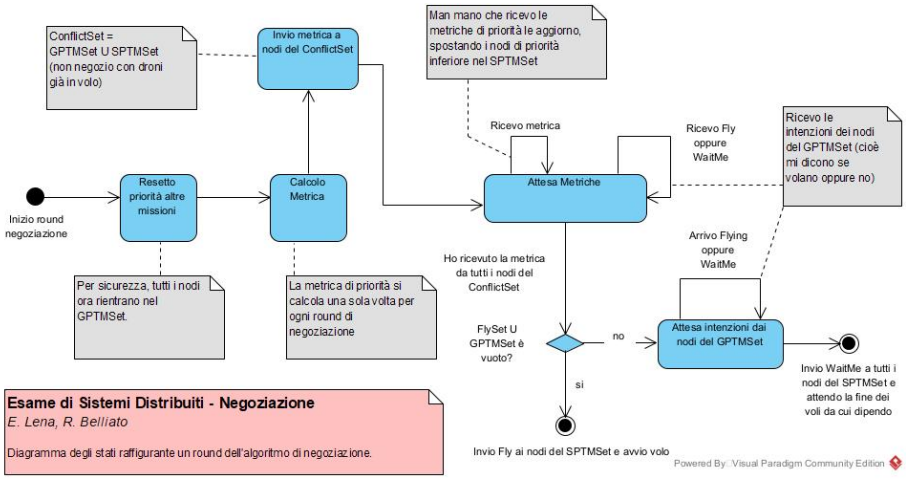

Figura 3.1: Diagramma degli stati del ciclo di vita di un drone.

- Al suo spawn, un nodo viene inizializzato (dall'ambiente o tramite un registro, si veda la sotto-sezione 3.1.2) con una lista di tutti i nodi che si suppongono essere attivi. A quel punto, il nodo deve "presentarsi" comunicando agli altri la tratta che intende percorrere; si aspetta poi di ricevere come risposta la tratta degli altri nodi, assieme ad un'indicazione sul loro stato (in particolare, è importante che venga esplicitato un eventuale stato di "*Flying*").

- Man mano che si ricevono le risposte, si calcolano gli incroci e si aggiungono i nodi in conflitto nel $ConflictSet$ oppure nel $FlyingSet$. Per motivi di sicurezza, si può assumere che ogni nuovo nodo la cui priorità non è nota (e che non si sà essere già in volo) rientri nel $GPTMSet$.

- Dall'altra parte, anche quando un nodo riceve una richiesta di connessione da parte di un altro nodo in conflitto deve aggiungerelo al $ConflictSet$ e coinvolgerlo nelle negoziazioni.

- Di base, se un nodo riceve una richiesta di negoziazione prima di aver terminato l'inizializzazione risponde con una priorità nulla, in modo tale da essere inserito nel $SPTMSet$. Quando avrà terminato la sua inizializzazione, il nuovo nodo avvierà un nuovo round di negoziazione per provare a migliorare la sua situazione e partire prima.

Nell'algoritmo 1 si descrive la struttura del protocollo tramite pseudocodice.

Algorithm 1 Protocollo di inizializzazione

procedure INIT
  $state \leftarrow Init$
  $t_{spawn} \leftarrow now()$
  $FlyingSet \leftarrow newSet()$
  $ConflictSet \leftarrow newSet()$
  send $Connect(path)$ to all nodes
  waiting for $Conflict$, $NoConflict$ or $FlyingConflict$ from all
  execute NEGOTIATEROUND()
end procedure

procedure ON RECEIVE(NoConflict msg)
  that node is not interesting to me
end procedure

procedure ON RECEIVE(Conflict)
  adds $Conflict$ to $GPTMSet$, so then when I am ready I can negotiate
end procedure

procedure ON RECEIVE(FlyingConflict — Flying)
  adds the node to $FlyingSet$, so I am sure I will wait for him
end procedure

3.2.3 Protocollo di negoziazione

La negoziazione è la fase in cui un gruppo di nodi decide quale (o quali) spedizioni possono partire subito e quali invece devono attendere.

Come anticipato nell'introduzione (sezione 1.2), la soluzione proposta per la prevenzione dei conflitti si basa sulla modellazione del problema come un grafo non orientato, dove ogni nodo rappresenta una tratta e ogni arco un conflitto. Da tale grafo, si vuole determinare un *insieme indipendente* di nodi che possono partire subito; per i nodi rimanenti invece si vuole definire un *grafo aciclico diretto* che determina l'ordine delle attese definito in questo modo: dati due nodi a e b è presente un arco da a a b se a deve attendere b per poter partire. I nodi che possono partire al termine di una negoziazione sono quindi quelli con solo archi in ingresso (e che ovviamente non stanno attendendo il termine di nessun altro volo).

Dalla prospettiva del singolo nodo, gli archi in ingresso sono rappresentati dai nodi con priorità minore (quindi i nodi del $SPTMSet$); gli archi in uscita sono quelli i cui nodi hanno priorità maggiore (quindi i nodi del $GPTMSet$).

Di seguito si descrive la struttura del protocollo di negoziazione che porta al suo termine alla costruzione in ogni nodo di una prospettiva *coerente* delle sue dipendenze.

- La negoziazione inizia con il calcolo della propria priorità da parte di ogni

nodo. Tale calcolo avviene una sola volta per ogni round e si fa tramite una specifica metrica, unica per tutti i nodi (vedere sotto-sezione 3.2.5). Una volta calcolata la metrica, questa viene condivisa con tutti i nodi del $ConflictSet$.

- All'avvio della negoziazione, tutto il $ConflictSet$ viene concettualmente considerato $GPTMSet$ (si impostano tutte le priorità ad infinito); man mano che si ricevono le metriche dei vicini, si aggiornano le priorità e quindi si spostano i nodi di priorità minore nel $SPTMSet$. Per come è strutturata la metrica, al termine dello scambio tutti i nodi hanno quindi una visione chiara e consistente di chi ha vinto su chi. Al termine dello scambio della metrica, globalmente, ci sarà sempre almeno un nodo che vince su tutti i propri vicini.

- Se dopo una negoziazione l'insieme degli archi uscenti di un nodo ($FlyingSet \cup GPTMSet$) è vuoto, il nodo è autorizzato a partire (perché ciò vuol dire che ha vinto tutti i conflitti e che non sta attendendo alcun nodo in volo). La sua partenza deve essere notificata a tutti i nodi del $SPTMSet$ in modo tale che questi spostino il nodo dal $GPTMSet$ al $FlyingSet$.

  Tutti i nodi per i quali questa proprietà non è vera, dovranno attendere (o un nodo in volo, o un nodo ancora fermo ma che si è stabilito, per qualche motivo, che ha la priorità).

- Ogni nodo che non parte ha almeno un vicino appartenente a $FlyingSet \cup GPTMSet$; per fare in modo che tutti i nodi abbiano un idea chiara dello stato dei propri vicini, si vuole che ogni nodo che non parte:

  - attenda le intenzioni da tutti i nodi appartenenti a $FlyingSet \cup GPTMSet$ (per scoprire se volano o se stanno fermi);
  - una volta note tali intenzioni, comunichi a loro volta a tutti i nodi del $SPTMSet$ la propria intenzione di stare fermi.

  Questa procedura di retro-propagazione potrebbe essere un occasione per comunicare ai nodi che dovranno stare fermi stime sui tempi di attesa e/o informazioni sulla struttura del grafo delle dipendenze.

Nel suo ciclo di vita, si può immaginare che un nodo partecipi anche a più di un *round di negoziazione*. Si riconosce che ripetere più volte la procedura di negoziazione potrebbe aiutare a costruire set indipendenti di nodi sempre più "ottimali", fino magari a raggiungere un insieme indipendente massimale [5, 6]. Detto ciò, nella prima implementazione presentata in questo elaborato l'unico caso in cui si esegue un nuovo round di negoziazione è l'ingresso di un nuovo nodo nella rete.

L'esecuzione di singolo round del protocollo di negoziazione può essere riepilogato con il pseudo-codice dell'algoritmo 2, oppure con in diagramma degli stati della figura 3.2.

Algorithm 2 Protocollo di negoziazione

```
procedure ExecuteNegotiate
    state ← Negotiate
    myMetricValue ← CalculateMetric()
    GPTMSet ← new Set()
    SPTMSet ← new Set()

    send Negotiate(path, myMetricValue) to all nodes in ConflictSet
    waiting for Negotiate from all nodes in COnflictSet

    if GPTMSet ∪ FlyingSet is empty then
        execute BeginFly()
    else
        execute FinishNegotiation()
    end if
end procedure

procedure On Receive(Negotiate msg)
    compare metrics and decide if it goes in GPTMSet or in SPTMSet
end procedure

procedure BeginFly
    state ← Flying
    send Flying to all nodes in SPTMSet
    execute PerformFly()
end procedure

procedure FinishNegotiation
    waiting for Flying or WaitMe from all nodes in GPTMSet
    send WaitMe to all nodes in SPTMSet
    state ← Wait
end procedure

procedure On Receive(Flying)
    remove node from ConflictSet and GPTMSet
    add node to FlyingSet so I am sure I can wait for him
    when node is safe
    do FlyingSet.remove(node) and check if I can BeginFly()
end procedure
```

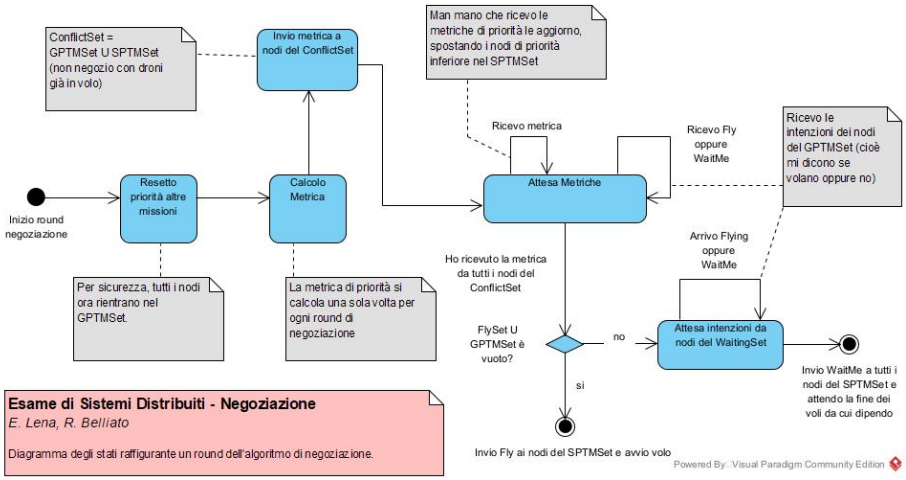

Figura 3.2: Diagramma degli stati del protocollo di negoziazione. È il sotto-diagramma che descrive più nel dettaglio lo stato *Negotiate* della figura 3.1

Attesa dei nodi in volo

Un discorso fortemente correlato alla negoziazione è l'attesa dei nodi in volo. Come è già stato specificato più volte, un nodo che comunica la sua partenza viene inserito in un insieme chiamato *FlyingSet*.

L'uscita da tale insieme avviene nel momento in cui si ha la garanzia che, partendo adesso, l'altro nodo superi il punto di conflitto prima che lo raggiunga il nodo in questione. Per garantire ciò (e allo stesso tempo non fare attese inutili), quando si riceve la comunicazine di una partenza (o più in generale una comunicazione sullo stato di volo di un nodo) si può calcolare un "*tempo di sicurezza*" (come spiegato nella sotto-sezione 3.2.4). Dopo aver atteso un intervallo di tempo maggiore o uguale al tempo di sicurezza, si può rimuovere il nodo dal *FlyingSet* e considerare il conflitto risolto.

Complessità

Dato un grafo dei conflitti composto da N nodi - ognuno dei quali ha in media C archi - ogni nodo invia C messaggi per comunicare la propria metrica e al più C messaggi per comunicare la propria intenzione ai vicini con cui vince. La complessità media nel numero dei messaggi è quindi $\Theta(C * N)$. Il caso peggiore si ha in presenza di grafi completi in cui si raggiunge una complessità di $\Theta(N^2)$.

Per quanto riguarda la complessità in termini di spazio, ogni nodo deve memorizzare $\Theta(C)$ informazioni sulle tratte con cui è in conflitto.

Lo spazio richiesto per ogni messaggio è invece $\Theta(1)$, quindi la complessità in termini di bit rimane $\Theta(C * N)$.

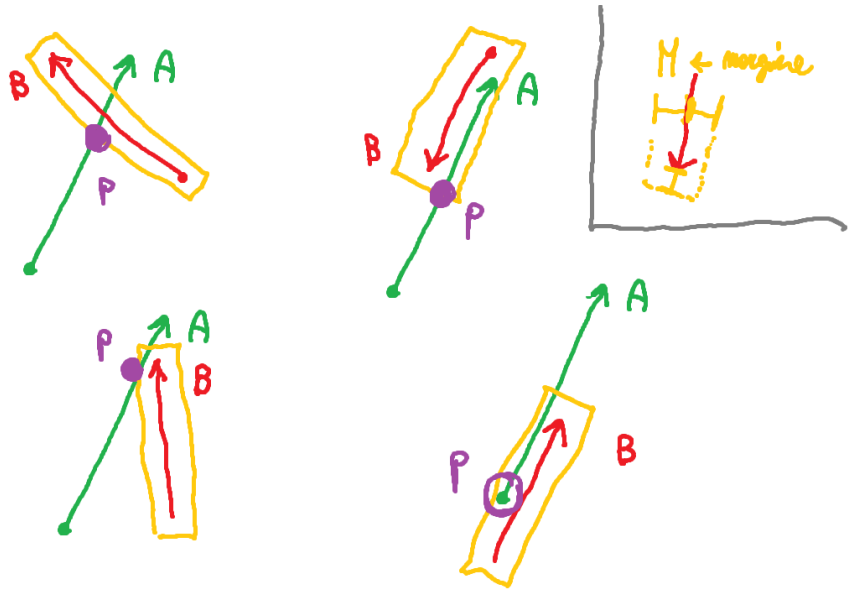

Figura 3.3: Calcolo del primo punto di conflitto tenendo conto del margine.

3.2.4 Calcolo delle collisioni e dalla distanza da esse

Ogni tratta può essere vista come una coppia di punti $T = \langle P_{start}, P_{end} \rangle$ (tra i quali si può tracciare un segmento). Teoricamente, si ha rischio di collisione se e solo se c'è esiste un intersezione tra i segmenti delle tratte. Nel caso l'intersezione sia costituta da più di un punto, ogni tratta seleziona come punto di conflitto il più vicino alla propria partenza (in quanto, dal suo punto di vista, quello è il primo punto del viaggio nel quale si rischia una collisione; due tratte possono quindi avere due prospettive diverse - entrambe conservative - del conflitto).

Ogni nodo può considerare quindi il *primo punto di conflitto* con la tratta $T^{(i)}$:

$$P \in T \wedge T^{(i)} \text{ t.c. } \text{dist}(P, T.Start) \text{ è minima}$$

Calcolo del conflitto con margine

Per garantire il volo in sicurezza, si ritiene necessario inserire nel calcolo del conflitto un *margine*, così da coprire tutte quelle situazioni limite nelle quali si hanno due tratte molto vicine ma non propriamente intersecate (si veda le situazioni della figura 3.3).

Per coprire questi casi si può fare in modo che ogni missione rappresenti le altre tratte come poligoni, o meglio, come rettangoli costruiti mettendo un

margine attorno ai segmenti, come si mostra in figura 3.3. Facendo ciò si può calcolare il conflitto come l'intersezione (più vicina al proprio punto di partenza) tra il proprio segmento e il rettangolo delle altre tratte. Nel concreto, per calcolare il punto di conflitto tra una tratta del drone (T) e una ricevuta per messaggio ($T^{(i)}$), si esegue la procedura spiegata qui di seguito.

- Si ricavano i vertici di un ipotetico rettangolo costruito attorno a $T^{(i)}$:

  $$A = T^{(i)}.Start - M * T^{(i)}.normDirection() - M * T^{(i)}.perpNormDirection()$$
  $$B = T^{(i)}.Start - M * T^{(i)}.normDirection() + M * T^{(i)}.perpNormDirection()$$
  $$C = T^{(i)}.End + M * T^{(i)}.normDirection() + M * T^{(i)}.perpNormDirection()$$
  $$D = T^{(i)}.End + M * T^{(i)}.normDirection() - M * T^{(i)}.perpNormDirection()$$

- Dati tali vertici, si costruisce un poligono e si verifica se il punto di partenza $T.Start$ è contenuto nel poligono (se si, quello è il punto di conflitto).

- Se il punto $T.Start$ non è contenuto nel poligono, si ricavano i quattro (eventuali) punti di incrocio:

  $$T \wedge \overline{AB}$$
  $$T \wedge \overline{BC}$$
  $$T \wedge \overline{CD}$$
  $$T \wedge \overline{DA}$$

  tra questi (se esiste almeno uno), si seleziona il più vicino al punto di partenza $T.Start$.

Calcolo del tempo per la partenza sicura

Dati due nodi A e B, se A riceve una comunicazione di partenza da parte del nodo B, può calcolare un *tempo da attendere per una partenza sicura*, cioè un tempo minimo da attendere prima di considerare la tratta "libera".

Il tempo per la partenza sicura può essere calcolato in diversi modi. Due approcci abbastanza conservatori sono:

- usare la durata della tratta B;

- usare la distanza temporale tra il punto di partenza di B e il punto P_A, cioè $v_B * \text{dist}(B.start, P_A)$.

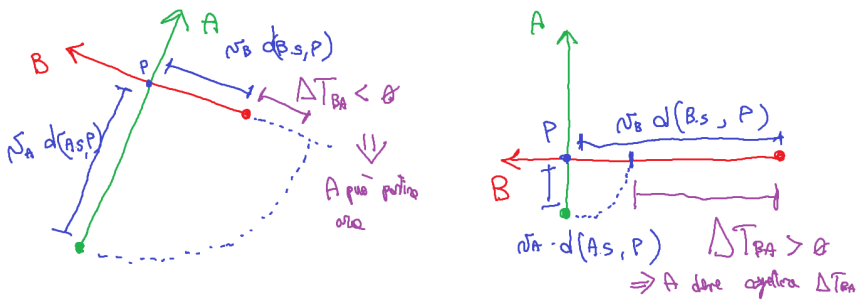

Figura 3.4: Tempo di partenza sicura rappresentato graficamente (senza margine). Si nota che nella figura a sinistra A può partire senza dover aspettare B (in quanto si sa per certo che B raggiungerà il punto di conflitto prima). Al contrario, nella figura a destra A si deve aspettare un tempo $\Delta T_{BA} > 0$ per avere la garanzia che B passi prima.

Entrambi questi approcci sono però in realtà eccessivamente prudenti. In base alle nostre assunzioni, il comportamento di volo dei nodi è estremamente prevedibile: una volta che un nodo parte, si sa per certo che volerà in linea retta ad una certa velocità senza mai rallentare; al massimo può effettuare un atterraggio di emergenza. Data questa assunzione, si può immaginare di prendere un *tempo minimo anche minore* di $v_B * \text{dist}(B.start, P_A)$:

> quando A parte, B continua a muoversi in maniera prevedibile, pertanto se A parte quando B è sufficientemente vicino al punto di conflitto, ha la garanzia che non ci saranno collisioni in quanto B supererà tale punto di conflitto prima che A ci arrivi.

Data questa intuizione, si può immaginare di calcolare il tempo minimo da attendere come la *differenza tra due distanze temporali*, cioè con la formula (rappresentata anche graficamente nella figura 3.4):

$$\Delta T_{BA} = v_B * \text{dist}(B.start, P_A) - v_A * \text{dist}(A.start, P_A) + \Delta T_M$$

(dove ΔT_M è un intervallo di margine).

Si noti che tale formula ha una proprietà piuttosto interessante: se

$$v_A * \text{dist}(A.start, P_A) > v_B * \text{dist}(B.start, P_A) + \Delta T_M$$

il valore ΔT_{BA} ricavato risulta essere negativo; di conseguenza, A può considerare il conflitto risolto in quanto, se partisse ora, al momento in cui raggiungerebbe P_A, B avrebbe già superato tale punto.

Questa formula può essere utilizzata per due scopi: da una parte permette di evitare attese inutili, dall'altra può potenzialmente permettere di calcolare

in fase di negoziazione "quanto" una partenza potrebbe penalizzare le altre (evitando quindi il più possibile i casi rappresentati nella parte destra della figura 3.4).

3.2.5 Metrica di priorità e note sulla negoziazione

La scelta di quali missioni devono partire a seguito della negoziazione è una questione sulla quale c'è molta libertà. Nel nostro caso si è scelto di stimare la priorità di ogni spedizione attraverso una metrica che:

- favorisce missioni vecchie rispetto a missioni nuove (per prevenire la starvation);
- favorisce missioni in grado di partire prima rispetto a missioni che devono attendere prima di partire (ad es., perché sono in conflitto con una missione già in volo);
- favorisce missioni che "liberino prima" lo spazio di volo rispetto a missioni che fanno attendere molto gli altri;
- sia calcolata una sola volta per ogni round e sia confrontata in modo coerente (per prevenire deadlock e/o inconsistenze).

A nostro parere, una proposta sensata potrebbe essere la seguente:

$$m = (age)^k - h_1 * |ConflictSet| * \max\{\text{waits in my } FlyingSet\} - h_2 * \sum\{\text{waits I may cause in my } ConflictSet\} \quad (3.1)$$

Tale metrica abbastanza elementare si basa sulle due informazioni che un nodo dovrebbe conoscere meglio, cioè:

- il tempo che ha già atteso (age);
- il tempo che farebbe attendere ai suoi vicini in caso di sua partenza, che include la più grande delle sue attese già schedulate e la somma delle attese dirette che procura agli altri nodi.

Di seguito, si riporta qualche nota aggiuntiva sulla metrica e su come viene utilizzata per stabilire la priorità.

- Le varie componenti della metrica sono tempi espressi in un'unità consona alla durata di percorrenza delle tratte (ad es., se mediamente una tratta ci mette una decina di minuti per arrivare a destinazione, il minuto è una buona unità di misura).
- k è un parametro arbitrario che permette di aumentare esponenzialmente il peso di age (ci si aspetta che un numero reale, ragionevolmente si può scegliere un valore compreso tra 1 e 2 - ad es., $k = 1.2$).

- h_1 ed h_2 sono parametri arbitrari che consentono di modulare l'effetto della mia massima attesa in corso e delle attese che causerei agli altri.

- Essendo la metrica è un numero reale e visto che in caso di parità di valore si può considerare l'ID univoco della missione, dato un set di nodi coinvolti in un conflitto ciclico, si potrà sempre stabilire un ordinamento e ci sarà sempre almeno un "vincitore" (*non si prevede quindi di avere situazioni di deadlock*). Allo stesso tempo, data una coppia di nodi in conflitto al termine della negoziazione essi avranno sempre una visione coerente su chi ha vinto e chi ha perso.

Si noti infine che la metrica potrebbe essere migliorata empiricamente, anche in base alle situazioni reali che si osservano essere più frequenti.

3.2.6 Uscita dei nodi dalla rete, timeout e gestione dei crash

Un aspetto molto importante da considerare è il problema dell'uscita dei nodi dalla rete. L'uscita può avvenire per diversi motivi, in diversi modi e deve essere gestita dai nodi rimanenti. Alcune di queste uscite sono "pulite e regolari" (cioè avvengono secondo un protocollo prestabilito), mentre altre sono dovute ad errori e vanno in qualche modo gestite.

Le uscite regolari avvengono in due casi: quando un drone ha terminato la sua missione e quando invece, per un qualche motivo, ha deciso di annullare la missione (ad es., perché si rileva un malfunzionamento o per comando esplicito dell'utente). In entrambi i casi, il drone provvede a comunicare a tutti i nodi (da lui noti) la propria uscita tramite un messaggio (così che questi possono rimuoverlo dai vari set delle dipendenze).

Oltre queste due modalità di uscita, si devono gestire anche quelle situazioni (eccezionali) in cui un drone, per una qualunque ragione, non è più raggiungibile (ad esempio perché si è schiantato, si è spento o perché non ha più connessione). In particolare, si immaginano due situazioni:

- in fase di inizializzazione un drone non riesce a raggiungere alcuni nodi, e quindi non riesce a conoscere il loro stato (e quindi sapere se è già in volo, se ci sono potenziali conflitti, ecc.);

- in fase di negoziazione, un drone non riesce più a contattare uno dei nodi con cui è in conflitto.

In entrambi i casi, per evitare collisioni o starvation (degli altri nodi coinvolti nella negoziazione), si prevede che se un drone non riesce a reperire una certa informazione di interesse in un tempo ragionevole, esso debba evitare di partire e annullare la sua missione. In pratica, si prevede un tempo limite entro il cui un nodo deve ricevere tutte le risposte previste durante l'inizializzazione o la negoziazione. L'annullamento viene svolto avvertendo tutti gli altri nodi, in modo tale da non causare terminazioni a catena.

Quanto discusso qui è il motivo principale per il quale nella sotto-sezione 3.4.1 si è stabilito che, in certi casi, non è possibile garantire la disponibilità del servizio.

3.2.7 Altri algoritmi e protocolli

Comandi utente

Si prevede che l'utente possa inviare dei comandi basilari al drone della propria missione, per - ad esempio - ottenere lo stato corrente e annullare la missione.

Servizio di notifica

Per ogni drone si propone di implementare un servizio di notifica. Se un utente (umano o macchina) desidera ricevere uno stream dello stato del drone, può generarsi un processo **Observer**, che:

- si iscrive ad un servizio di notifica del drone;
- riceve notifiche ogni qual volta "succede qualcosa".

Protocollo di snapshot

Per dei super-utenti potrebbe essere utile:

- poter richiedere snapshot dei droni coinvolti in un certo cluster di conflitti;
- poter richiedere snapshot dell'intera rete.

3.3 Architettura fisica e dispiegamento

3.3.1 Architettura fisica e note sulla rete

L'architettura fisica del sistema è fortemente suggerita dai requisiti. Dato che ad ogni nodo logico corrisponde un drone, si può assumere che corrisponda anche un nodo fisico (l'unità di calcolo del drone).

Sempre dai requisiti, si deriva che i nodi fisici (quindi i droni) hanno accesso ad una qualche forma di connessione wireless. Si possono immaginare diverse situazioni, ma in sostanza ci si riconduce a due casi:

1. l'area fisica in cui operano i droni è interamente coperta da una WAN (gestita dall'organizzazione che fornisce il servizio di consegna, oppure da un loro fornitore);
2. i droni dispongono di una scheda telefonica e comunicano passando per Internet.

Scegliere una soluzione del primo tipo facilita la gestione delle problematiche di rete, in quanto permetterebbe di gestire in autonomia l'assegnazione degli indirizzi IP[1] e del routing. Detto ciò tale soluzione comporterebbe anche la gestione di aspetti tediosi come la mobilità dei dispositivi.[2]

Se si decide di optare per la seconda soluzione, probabilmente risulterebbe comodo utilizzare una Virtual Private Network per aver maggior controllo sull'assegnazione degli indirizzi IP.

3.3.2 TCP vs UDP

I protocolli definiti per il coordinamento della rete di droni possono funzionare sia con protocolli connectionless che con protocolli orientati alla connessione. Nel primo caso, ad ogni messaggio scambiato corrisponderebbe un singolo datagramma, mentre nel secondo si avrebbe una connessione per ogni trasmissione (in modo simile a come in HTTP si ha una connessione per ogni richiesta).

La scelta di TCP comporterebbe un minor numero di fallimenti dovuti alla non raggiungibilità dei nodi, ma potrebbe anche essere pesante per la rete (che si ricorda essere basata su un'importante componente wireless). La scelta di UDP invece ridurrebbe il carico di rete, ma si aumenterebbe il numero di missioni che falliscono per la mancata ricezione di un messaggio.

Per il prototipo da realizzare in questo progetto scegliamo di utilizzare TCP (per diminuire la probabilità di complicare la fase di debug con errori di rete imprevedibili), ma per un sistema da usare in produzione bisognerebbe rivedere la scelta in base alle scelte sulla rete e in base a degli esperimenti pratici.

3.3.3 Linguaggi e tecnologie

Per realizzare il progetto, si è scelto di utilizzare il *linguaggio C#* e la libreria *Akka.NET* [2] (con l'estensione *Akka.NET Remote*).

Akka.NET è un framework per la realizzazione di sistemi concorrenti e distribuiti basato sul paradigma dell'*actor base programming*. L'idea alla base del framework è di implementare il modello della *comunicazione asincrona tra processi tramite scambio messaggi* (quindi lo stesso modello tipicamente utilizzato da linguaggi come Erlang), ma attraverso un linguaggio general purpose e orientato agli oggetti come C# (si noti che Akka.NET è un porting di Akka, originariamente pensato per Java e Scala).

In Akka.NET ogni processo logico è rappresentato da un *attore*, cioè un particolare tipo di oggetto capace di inviare e ricevere messaggi. Esso è identificato univocamente da un URI (detto anche *ActorRef*), vive all'interno di un *ActorSystem* e può essere inserito in una gerarchia di supervisione.

[1] L'assegnazione degli indirizzi IP potrebbe essere effettuata tramite un servizio DHCP; detto ciò è importante che il servizio sia configurato in modo tale che l'indirizzo assegnato rimanga valido per tutto il ciclo di vita di una missione.

[2] È essenziale garantire che un dispositivo possa muoversi tra diversi punti di accesso della rete, mantenendo comunque attiva la propria connessione.

Differenze tra il paradigma degli attori e gli oggetti remoti

Si noti che Akka.NET *non* implementa né il paradigma delle *remote procedure call*, né il paradigma degli oggetti remoti. Nonostante un attore sia effettivamente un oggetto, la comunicazione tramite messaggi è qualcosa di ben distinto dalle chiamate di metodo. Di seguito si elencano alcune importanti differenze tra Akka.NET e gli oggetti remoti.

- Per inviare un messaggio non è necessario avere un "puntatore" all'istanza remota, bensì si usa un riferimento univoco gestito dal framework (trasparente per locazione, ma potenzialmente anche identificabile da un URI).
- Semanticamente, l'operazione di invio di un messaggio è distinta dalla chiamata di metodo. Dato un riferimento, posso inviare un certo messaggio tramite il metodo `Tell` (*Fire-And-Forget*), oppure tramite `Ask` (*Send-And-Receive-Future*) che permette di attendere e gestire un'eventuale risposta in maniera asincrona.
- Anche la ricezione dei messaggi è ben distinta dall'esecuzione di un metodo. Intuitivamente, si può dire che i messaggi vengono consegnati ad una mailbox e gestiti dall'attore attraverso una serie di direttive del tipo `Receive<Type>(Condition, Action)`, che permettono di collegare l'esecuzione di una procedura all'arrivo di un certo tipo di messaggi.[3].

  Si noti che a differenza del *receive...end* di Erlang, il `Receive` di Akka.NET non è un'attesa sincrona di un messaggio, bensì è una sorta di direttiva che si dà in fase di creazione dell'attore (oppure durante un cambio di stato) che indica come gestire tutti i messaggi di un certo tipo. Al posto del meccanismo del pattern matching di Erlang, si utilizza il polimorfismo per decidere quale istanza di `Receive` deve gestire quali messaggi.
- I messaggi scambiati sulla rete sono istanze serializzate di oggetti (di default stringhe JSON).

ActorSystem, dispiegamento e Akka.NET Remote

In Akka.NET gli attori vivono all'interno di *ActorSystem*. Un *ActorSystem* è un gruppo di attori visibile come un nodo logico di un sistema. Ogni *ActorSystem* si identifica con un path (o con un riferimento che rende trasparente la locazione) e offre diversi servizi agli attori ospitati (supervisione, meccanismi per la comunicazione di gruppo basata su eventi, ecc.) e agli utenti esterni (ad es., meccanismi di selezione). In Akka.NET il dispiegamento degli attori su più nodi fisici avviene tramite l'estensione Akka.NET Remote [1]. Tale estensione permette sostanzialmente di:

[3] Si noti che l'esecuzione di tale procedura è bloccante per il thread corrispondente all'attore (pertanto garantisce che i messaggi vengano gestiti uno alla volta), ma allo stesso tempo non blocca la consegna di nuovi messaggi alla mailbox.

- assegnare un indirizzo IP e una porta agli *ActorSystem*;
- definire con che protocolli deve avvenire la comunicazione;
- accedere ad *ActorSystem* remoti e ai loro attori (sempre utilizzando il meccanismo dell'$ActorRef$ univoco);
- generare attori su nodi remoti e - nel limite del possibile - "monitorarli".

3.3.4 Procedure di dispiegamento

Nella nostra soluzione per il dispiegamento prevediamo la procedura di dispiegamento descritta di seguito.

- Ad ogni drone fisico corrisponde un *ActorSystem* (remoto, situato sulla macchina fisica del drone e - in una situazione reale - creato all'avvio del dispositivo). Quando un host dell'interfaccia desidera avviare una missione, spawna su uno di questi actor system un attore Akka.NET che rappresenta una missione a livello logico.
- Quando l'attore del drone si avvia, richiede ad un registro una lista di nodi noti. Ricevuta la lista, il registro non serve più e l'attore può iniziare ad eseguire prima il protocollo di inizializzazione e poi quello di negoziazione.
- Per gestire alcuni servizi particolari, gli attori principali possono generare a loro volta dei sotto-attori nel proprio ActorSystem (ad es., si genera un attore per gestire la simulazione del volo e uno per gestire il servizio di notifica).
- Da un qualsiasi host (che sia un interfaccia utente, un altro sistema software, ecc.) è possibile:
  - contattare gli attori dei droni per conoscere il loro stato;
  - inviare comandi di cancellazione missione;
  - iscriversi a servizi di notifica tramite attori Observer locali.

In figura 3.5 si presenta uno schema che illustra come i nodi logici sono collocati nei nodi fisici, come avviene lo spawn e quali sono le macro-dinamiche di comunicazione.

3.4 Note sulle trasparenze e sul CAP Theorem

3.4.1 Nota sul CAP Theorem e sulla disponibilità del servizio

Come in tutti i sistemi distribuiti, anche in questo caso si è dovuto considerare il *CAP Theorem* per decidere come gestire eventuali situazioni di partizionamento della rete. In particolare, in questo progetto il partizionamento della rete diventa un problema in due casi specifici:

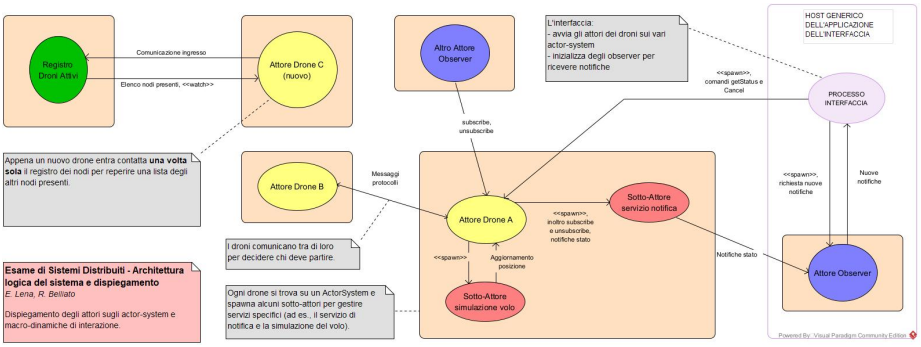

Figura 3.5: Architettura fisica e comunicazione tra i processi.

- quando un nodo entra nel sistema e non riesce, per qualche motivo, a comunicare con gli altri nodi già presenti;
- quando un nodo non riesce a portare a termine una negoziazione.

Si noti che, in entrambi questi casi, se un drone decidesse di partire senza prima aver ricevuto le intenzioni e il consenso di tutti gli altri rischierebbe di avere una collisione con una missione in volo non nota, oppure con una missione in conflitto che a sua volta parte senza terminare la negoziazione. *Garantire la consistenza* diventa quindi una priorità, anche al costo di *rinunciare alla garanzia di disponibilità del servizio*.

Nel concreto, ciò si traduce in un semplice principio:

> se un nodo non riesce a conoscere le tratte di tutti gli altri e/o a portare a termine tutte le negoziazioni entro un tempo ragionevole, esso non deve partire e deve invece annullare la propria missione e comunicare la propria uscita a tutti (per evitare fallimenti a catena).

3.4.2 Trasparenze

In ottica di quanto definito per il modello logico (sezione 3.1), per il modello fisico (sezione 3.3) e per le scelte implementative (sotto-sezione 3.3.3), si prevede di riuscire ad implementare le trasparenze elencate di seguito:

1. *trasparenza di locazione* ← grazie ai riferimenti univoci degli attori di Akka.NET, i nodi possono comunicare tra di loro senza considerare la loro locazione fisica.

   Per l'utente, invece, la trasparenza di locazione nel primo prototipo non sarà implementata, perché si dovrà conoscere almeno l'indirizzo IP e la porta dell'ActorSystem su cui viene eseguito il protocollo. Si noti che in un servizio commerciale si potrebbe inserire un *tier* intermedio che nasconde tutto ciò (magari tramite una sorta di DNS dei droni);

2. *trasparenza di accesso* ← realizzabile attraverso l'implementazione di API Object-Oriented che incapsulano le procedure di avvio e gestione delle missioni;

3. *trasparenza di concorrenza e di parallelismo* ← i nodi della rete accedono alla risorsa condivisa (in questo caso lo spazio aereo) gestendo da soli i conflitti (senza necessità di intervento esterno).

4. *trasparenza di mobilità* ← si prevede che il drone possa spostarsi nello spazio fisico senza modificare il proprio modo di comunicare;

5. *trasparenza di scala* ← all'aumentare delle missioni, le meccaniche di funzionamento del sistema non cambiano (sia per l'utente che internamente). Ovviamente, nel caso il sistema venga sovraccaricato con troppe missioni la qualità del servizio peggiorerebbe (a causa di tempistiche di attesa più lunghe, maggiori probabilità di fallimento, ecc..);

Per quanto riguarda le trasparenze rimanenti:

- nel prototipo la *trasparenza di errore* non viene implementata, ma non si esclude la possibilità che ciò possa essere parzialmente fatto per un servizio commerciale (ad esempio, ri-schedulando una missione che fallisce prima di partire);

- questioni come le trasparenze *di replicazione* e *di performance* non sono concetti legati al tipo di problema (forse potrebbero essere questioni importanti per un eventuale tier intermedio tra l'utente e il sistema, che però non sarà sviluppato più di tanto in questo progetto);

3.5 Piano di sviluppo

Il piano sviluppo prevede le fasi elencate di seguito.

1. Implementazione dei protocolli e degli algoritmi fondamentali, quindi:
   (a) inizializzazione dei nodi;
   (b) calcolo dei conflitti;
   (c) negoziazione tra nodi e schedulazione delle partenze;
   (d) implementazione processo ambiente e creazione missioni.

2. Integrazione nel protocollo delle procedure di timeout e gestione degli errori.

3. Implementazione di ulteriori servizi per l'utente:
   (a) Richiesta dello stato di una missione,
   (b) Richiesta snapshot dello stato della rete,

(c) Implementazione del servizio di notifica.

4. Predisposizione di meccanismi di interfacciamento aperti ed adeguati al collegamento di esso con sistemi esterni (ad es., web server con API REST).

Non si prevede di riuscire a creare un'interfaccia grafica per il sistema, ma si realizzerà un'interfaccia da console adeguata per un prototipo del tier di sviluppo raggiunto.

Capitolo 4

Sviluppo

4.1 Tier 1: implementazione dei protocolli principali

Il primo tier di implementazione raggiunto è l'implementazione dell'algoritmo principale, cioè tutto ciò che un drone fa da quando entra nella rete, fino a quando termina la sua missione. Il primo tier di implementazione quindi include:

- la procedura di spawn di un attore;
- gli algoritmi legati all'ingresso di un nodo nella rete, alla negoziazione e al volo (in pratica, la procedura principale descritta dalla macchina a stati finiti della figura 3.1).

Di seguito, si trattano alcune tematiche strettamente legate all'implementazione tramite Akka.NET e al progetto ad oggetti del sistema; non si ripeteranno invece concetti "di progetto" (inteso come progettazione del sistema distribuito) o "di algoritmo" già trattati nel capitolo 3.

4.1.1 Attore del drone

La componente principale del sistema è l'attore del drone, implementato nella classe `DroneActor`. Tale classe ha sostanzialmente le responsabilità di:

- avviare tutte le procedure di inizializzazione di un drone (quelle immediatamente successive allo spawn e antecedenti al primo contatto con gli altri nodi della rete);
- coordinare ad alto livello i protocolli distribuiti;
- ricevere i vari messaggi scambiati tramite Akka.NET, per darli poi in gestione alle proprie componenti interne specifiche.

In Akka.NET l'invio di messaggi si può fare in qualunque punto del codice tramite il metodo `actorRefDestinatario.Tell(messaggio);`, questa responsabilità (al contrario della ricezione) può invece essere spostata alle varie componenti interne.

In figura 4.1 si riporta un diagramma UML raffigurante la classe `DroneActor` e le sue componenti principali. Qui di seguito, si riporta qualche nota utile a comprendere il diagramma.

- Come si può osservare dal diagramma, la classe `DroneActor` risulta essere piuttosto spoglia. Essa difatti contiene soltanto qualche metodo di inizializzazione e i metodi che incapsulano tutte le varie direttive del tipo:

  ```
  Receive<TipoMessaggio>(msg => componente.Metodo(msg, Sender));
  ```

- A destra della classe `DroneActor` si possono osservare le tre classi principali rappresentanti le componenti interne di più alto livello:

  - `DroneActorContext` incapsula tutte le informazioni di base del drone e della missione, quali l'elenco di tutti gli altri nodi, le specifiche della missione da portare a termine, il timestamp dell'istante di spawn e vari altri dettagli legati ad Akka.NET (l'`IActorContext` ad esempio è uno strumento che permette di accedere a vari dettagli quali il proprio ActorSystem, le primitive per la generazione di attori figli, ecc.);
  - `DroneActorState` è la classe dove si implementa la vera logica dei protocolli primari definiti in fase di progetto;
  - `NotificationProtocol` è una classe ausiliaria utilizzata per gestire il protocollo di notifica.

- Osservando attentamente `DroneActor` si può notare che è una classe astratta (il suo nome è scritto in corsivo), sotto di essa difatti si possono osservare due implementazioni concrete, che differiscono sostanzialmente da come viene repertita la lista dei nodi:

  - `SimpleDroneActor` è l'implementazione che riceve nel costruttore la lista di tutti i nodi noti;
  - `RegisterDroneActor` è l'implementazione che, prima di eseguire i veri e propri protocolli, reperisce la lista di tutti i droni presenti da un registro.

  La tematica dello spawn degli attori e dell'inizializzazione tramite registro viene trattata più approfonditamente nella sotto-sezione 4.1.7.

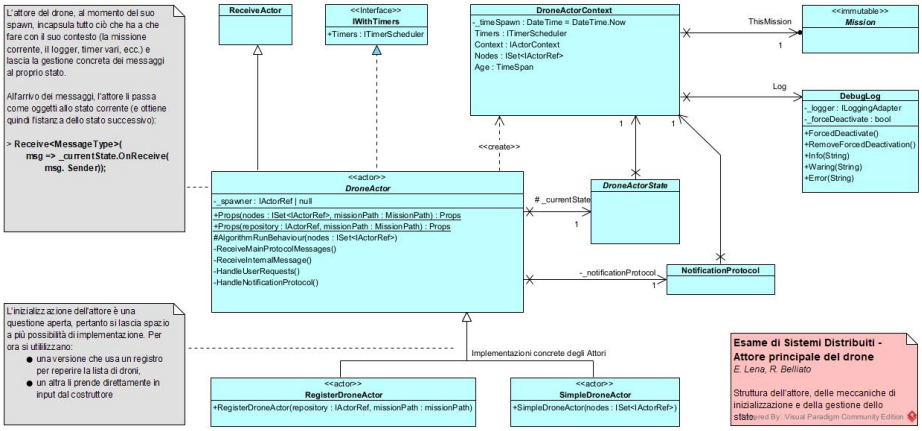

Figura 4.1: Diagramma di classi dell'attore del drone e delle sue componenti primarie

4.1.2 Gli stati del drone e la gestione dei messaggi

Modellazione dell'algoritmo distribuito con il pattern dello State

Come si è spiegato in fase di progetto, l'algoritmo che schedula le partenze può essere modellato come una *macchina a stati finiti* con quattro stati: *Init*, *Negotiate*, *Waiting* e *Flying*; a questi quattro stati si può aggiungere un quinto stato di uscita *Exit* (vedi sezione 4.1.6).

Ciascuno di questi stati può essere visto come una *fase distinta dell'algoritmo distribuito*. Ogni fase è caratterizzata da una serie di necessità comuni, quali:

- l'esecuzione di una procedura di inizializzazione;
- la gestione dei messaggi secondo politiche a volte uguali a quelle delle fasi, a volte totalmente differenti;
- il mantenimento di alcune informazioni interne, in parte utili sempre, in parte utili soltanto per la fase specifica;
- la capacità di decisione dopo ogni evento (quindi dopo l'inizializzazione, dopo la ricezione di un messaggio o dopo un evento interno) di quale deve essere lo stato successivo.

Per implementare tutto ciò, si può fare riferimento al *design pattern* dello *State* [4] il quale, usando i meccanismi dell'ereditarietà e del polimorfismo, fornisce uno schema di base per l'implementazione di macchine a stati finiti. Nella figura 4.2 si può osservare come è stato adattato il pattern per l'uso specifico; di seguito si riporta qualche nota per comprendere il diagramma.

- La classe base `DroneActorState` rappresenta una generica fase dell'algoritmo; le sue classi figlie sono le implementazioni delle cinque fasi specifiche.

- La parte principale della classe di base è una lunga sequenza di metodi `OnReceive(msg, sender)`; tali metodi rappresentano le procedure da eseguire quando viene ricevuto ciascun tipo di messaggio. Per alcuni di questi metodi la classe `DroneActorState` fornisce già un'implementazione di base (ad es., in caso di ricezione di una richiesta di connesione, si aggiunge il mittente alla lista dei nodi noti e gli si risponde con la propria tratta), mentre per altre si obbliga la classe figlia a fornire la sua implementazione concreta.

  In fondo alla classe `DroneActorState` si può osservare anche un metodo astratto `RunState()`, dove le classi figlie possono implementare la procedura di inizializzazione (per esempio, la classe `NegotiateState` può implementare quì l'invio della propria metrica ai nodi in conflitto).

- Osservando attentamente i metodi `OnReceive(msg, sender)` e `RunState()`, si nota che essi ritornano come output un'istanza di `DroneActorState`: sostanzialmente, tale output è il prossimo stato, che può coincidere con lo stato corrente, oppure può essere uno nuovo creato usando i metodi statici `CreateXXXState(...)` forniti dalla classe base.

Date queste premesse, ora si può intuire cosa succede quando un attore `DroneActor` riceve un messaggio (relativo all'algoritmo principale):

- tramite la direttiva `Receive` di Akka.NET, l'attore riceve il messaggio;

- invece che gestire il messaggio da solo, l'attore esegue il metodo `OnReceive` sull'oggetto `_currentState`, così da eseguire le procedure di gestione relative allo stato corrente;

- l'attore riceve poi in output l'istanza dello stato successivo, che va a sostituire quello corrente.

In questo modo, grazie al polimorfismo, l'attore riesce sempre ad eseguire la procedura di gestione dei messaggi corretta senza ricorrere a tediose strutture condizionali del tipo `switch-case` e inoltre, grazie all'ereditarietà, i vari stati possono condividere le parti comuni (ad es., i riferimenti ad alcuni componenti, la gestione di alcuni messaggi) e allo stesso tempo sovrascrivere/definire separatamente le proprie parti specifiche.

Messaggi gestiti

I messaggi gestiti da `DroneActorState` e dai suoi figli sono sostanzialmente tutti quelli dei protocolli primari, già descritti in fase di progetto (capitolo 3). Per ognuno dei messaggi, si è definita una classe-dato contenente tutte le informazioni necessarie (ad es., `ConnectRequest` contiene i dettagli della propria tratta).

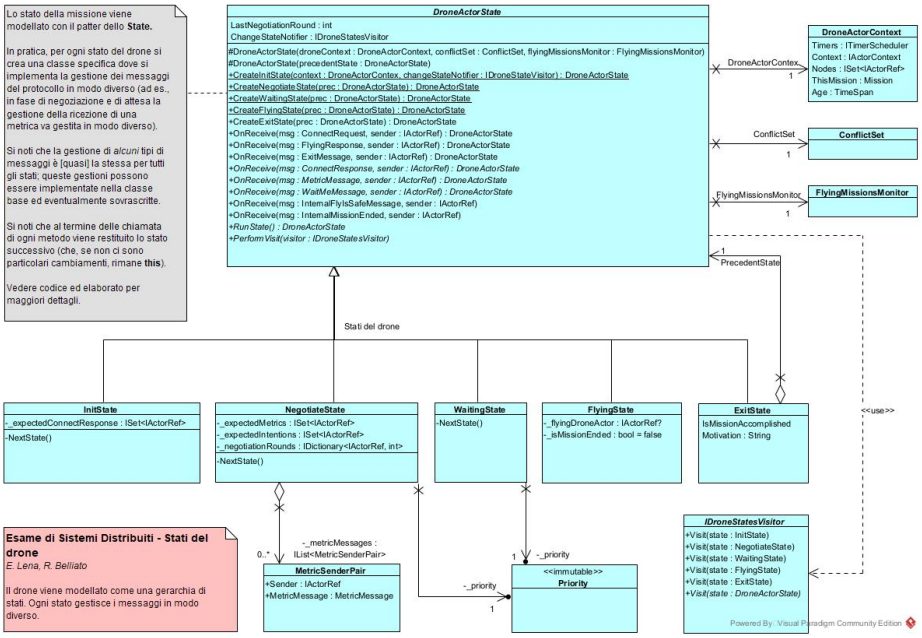

Figura 4.2: Diagramma di classi dello stato del drone (implementazione concreta dei protocolli di inizializzazione, negoziazione e volo).

Gli unici due messaggi che potrebbero apparire estranei sono quelli etichettati come `InternalXXX`. Essi sono messaggi relativi ad eventi interni (ad es., la fine del tempo di attesa verso una missione in volo) e vengono usati sostanzialmente per garantire la thread-safety dell'accesso alle varie strutture dati all'interno dell'attore stesso[1].

Descrizione dei singoli stati

Le singole classi degli stati non sono altro che implementazioni dei protocolli già definiti in fase di progetto, pertanto non si rimanda al capitolo 3 per maggiori dettagli.

4.1.3 Strutture dati (missione, priorità, ConflictSet, ecc.)

Dai diagrammi di alto livello dell'attore e dello stato (figure 4.1 e 4.2) si può osservare come si faccia spesso uso di tipi di dato non primitivi, specifici del problema. Ciascuno di questi tipi di dato è stato implementato come classe; di seguito si riportano i più significativi.

Tipi di dato di base (fig. 4.3)

1. `MissionPath` è la rappresentazione di una tratta. È costituita da un punto di partenza, un punto di arrivo e una velocità e ha la responsabilità di svolgere tutte le operazioni di tipo geometrico (implementate con la libreria *.NET Spatial*); in particolare, si offrono metodi per verificare se esiste conflitto tra due tratte e ricavare la "distanza temporale" dal punto di partenza ad una certa posizione.

   `MissionPath` si definisce *immutabile* in quanto, una volta creata un'istanza, questa non può essere modificata.

2. `Mission` rappresenta una generica missione. È costituita da una tratta e da un rifermento al nodo che la porta a termine. Tale classe è una delle più importanti del progetto, in quanto viene usata sia per rappresentare le missioni portate avanti dagli altri nodi, sia per rappresentare la missione corrente.

   Come si può osservare dal diagramma, la classe è astratta e viene implementata tramite due realizzazioni concrete:

   - `WaitingMission` rappresenta una missione ancora in attesa che, a differenza della classe padre, è caratterizzata da una priorità *mutabile*;

[1] In C# operazioni asincrone sono implementate utilizzando la classe `Task`. Ad ogni `Task` corrisponde un nuovo thread ma non un nuovo attore, per cui una task lanciata da un attore e il thread dell'attore stesso possono (potenzialmente) accedere concorrentemente agli stessi dati. Per evitare ciò uno strategia molto comune è quella di inviare messaggi a se stessi per simulare eventi interni all'attore.

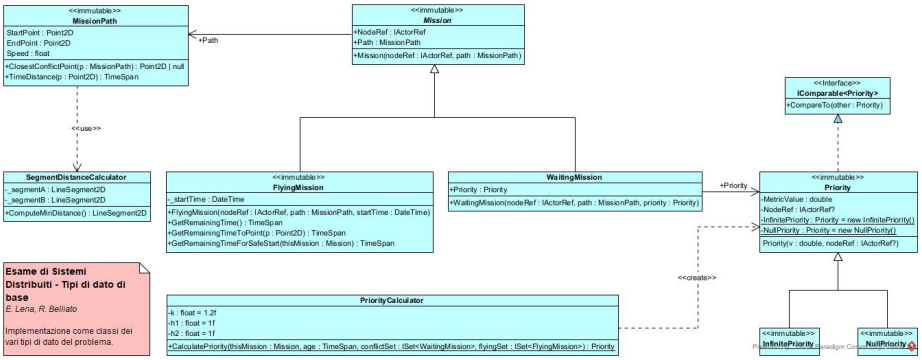

Figura 4.3: Diagramma di classi dei tipi di dato principali del problema.

- `FlyingMission` invece rappresenta una missione che è partita in volo; a differenza della controparte in attesa, non interessa più tenere traccia della sua priorità, ma invece si registra l'istante nel quale si è venuti a conoscenza della sua partenza e si offrono alcuni metodi per calcolare il *tempo di attesa per una partenza sicura*.

3. `Priority` rappresenta una priorità ed è costituita dalla metrica e dal riferimento al nodo. Come si può notare dal diagramma, essa espone un metodo che permette di fare dei confronti. L'identificatore del nodo serve per confrontare due priorità la cui metrica ha lo stesso valore; non si accettano casi in cui due priorità sono uguali.

   Sempre dal diagramma, si può notare anche come:

   - oltre alla classe base `Priority`, si definiscono due classi figlie `NullPriority` e `InfinityPriority`, usate rispettivamente per indicare che un drone perderà o vincerà tutti i conflitti; non si prevedono confronti tra infinito e infinito oppure tra nullo e nullo;
   - la generazione delle istanze viene delegata ad una classe `PriorityCalculator`, che ha il compito di calcolare la priorità di una missione attraverso la metrica spiegata nella sotto-sezione 3.2.5.

   Si noti infine che, come `MissionPath`, anche `Priority` è una classe immutabile.

Strutture per la gestione di set di missioni (fig. 4.4)

Un aspetto fondamentale dell'algoritmo è la gestione di "insiemi di missioni" (per implementare strutture come $ConflictSet$, $FlyingSet$, $GPTMSet$ e $SPTMSet$). Invece di usare direttamente le strutture base, si è preferito implementare due strutture specifiche ad hoc chiamate `ConflictSet` e `FlyingSet`.

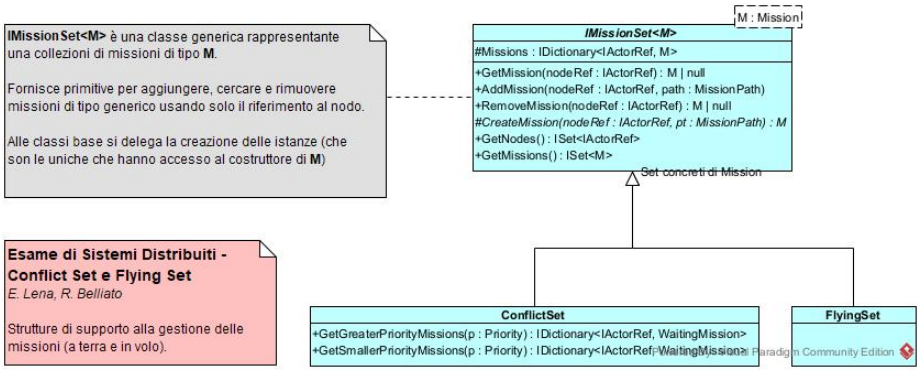

Figura 4.4: Diagramma di classi degli insiemi di missioni (Conflict Set e Flying Set).

Entrambi le strutture estendono una classe astratta e generica chiamata `IMissionSet<M>`, che rappresenta una collezione di missioni dove le istanze si creano internamente e la gestione avviene solo tramite riferimento al nodo (si è scelto di fare così per evitare di creare duplicazioni delle istanze di `Mission` e di lavorare invece soltanto con gli `IActorRef` passati tramite messaggio).

La classe `FlyingSet` è una semplice realizzazione di `IMissionSet<M>` con `M=FlyingMission`. La classe `ConflictSet` invece aggiunge dei metodi che permettono di estrarre facilmente gruppi di missioni di priorità superiore ed inferiore ad una passata in input. In questo modo, invece che implementare manualmente lo spostamento delle missioni tra $GPTMSet$ e $SPTMSet$, ci si può limitare ad aggiornare le priorità ed estrarre quando necessari i vari gruppi di nodi con priorità maggiore o minore.

4.1.4 Meccanismo di attesa delle missioni in volo

Un aspetto delicato dell'algoritmo non trattato nel dettaglio in fase di progetto è la gestione delle attese delle missioni in volo (quindi di tutte quelle missioni che comunicano la loro partenza dopo una negoziazione, ma anche di quelle che si scoprono essere in volo durante la fase di connessione o di waiting). Per gestire tale aspetto, si è usato l'approccio descritto di seguito.

- Si è definita una classe `FlyingMissionMonitor` (figura 4.5), che al suo interno contiene un riferimento alla missione corrente e al `FlyingSet`.
- Quando un nodo mi notifica la sua partenza o il suo stato di volo, passo l'istanza di missione in attesa corrispondente al nodo (che viene rimossa dal `ConflictSet`) al monitor. Il monitor la aggiunge al `FlyingSet` e crea un timer per attendere il tempo per la partenza sicura (ottenuto tramite l'istanza di `FlyingMission`).

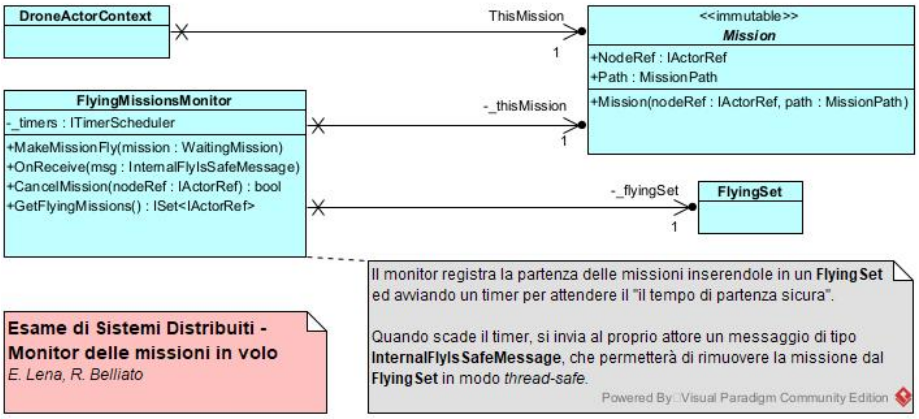

Figura 4.5: Diagramma di classi del monitor dei nodi in volo.

- Terminata l'attesa, il monitor invia all'attore del nodo corrente un messaggio di tipo `InternalFlyIsSafeMessage`, che funge da evento interno.
- l'evento interno viene gestito dallo stato corrente, il quale, tra le varie cose, usa l'istanza di `FlyingMissionMonitor` per rimuovere il nodo dal `FlyingSet`.

Per implementare il timer è stato utilizzata una funzionalità di Akka.NET, lo *Scheduler*, ossia un meccanismo interno all'*ActorSystem* che permette di schedulare l'invio di un messaggio a sè stesso (singolo o ricorrente) in avanti nel tempo. Ad ogni timer creato con questo metodo è associata una chiave univoca assegnata dal programmatore (per poterlo eventualmente annullare o controllare se è attivo).

4.1.5 Simulazione del processo di volo

Un'altra tematica non affrontata nel dettaglio in fase di progetto è la simulazione di un volo: quando il drone entra in stato di *Flying*, ci si aspetta che inizi a spostarsi nello spazio, che la sua posizione vari e che in qualche modo si renda conto quanto ha terminato.

In questo prototipo, la simulazione del volo è stata gestita spawnando - al momento della partenza - un attore figlio che periodicamente aggiorna la propria posizione (sempre con il meccanismo dei messaggi interni) e che:

- su richiesta dell'attore principale, ritorna la propria posizione attuale;
- all'arrivo a destinazione, notifica l'attore principale con un messaggio.

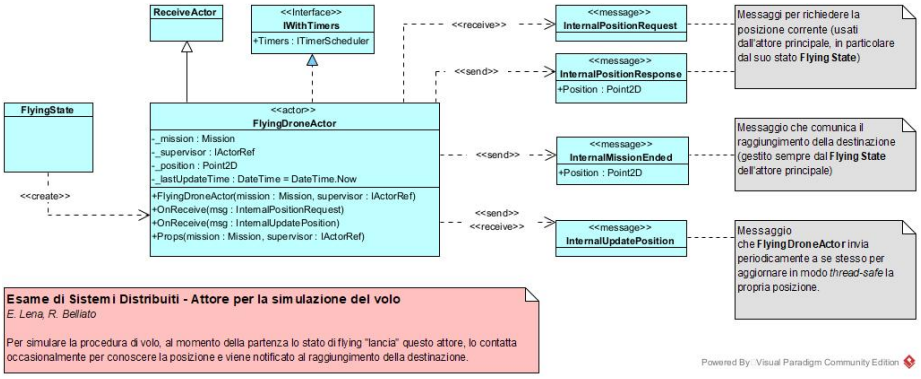

Figura 4.6: Diagramma di classi dell'attore che simula il volo.

Nella figura 4.6 si riporta un diagramma rappresentante l'attore `FlyingDroneActor` e i messaggi che scambia (tutti "interni" al nodo logico). La gestione dello spawn e della comunicazione con il sotto-attore sono incapsulate nello stato `FlyingState`.

4.1.6 Terminazione della missione

Quando un nodo raggiunge la destinazione o avviene un errore osservabile (es. timeout di ricezione messaggi - vedi sezione 4.2) il nodo passa in stato *Exit*. Questo passaggio provoca l'invio di un messaggio `ExitMessage` verso tutti i nodi conosciuti.

Quando un nodo riceve un `ExitMessage` questi rimuove il mittente dalla propria lista dei nodi conosciuti ed, eventualmente, anche dal $ConflictSet$ o dal $FlyingSet$. In quest'ultimo caso viene anche annullato il timer corrispondente alla missione.

4.1.7 Spawn dei droni e procedura di pre-inizializzazione tramite registro dei nodi

L'inizializzazione del `DroneActor` con una lista dei riferimenti agli altri droni è un problema cruciale. L'algoritmo definito in fase di progetto può garantire l'assenza di collisioni se e solo se ogni nuovo nodo che entra nella negoziazione riesce ad avere una lista di **tutti** i nodi entrati prima di lui (al momento attivi). Per come è strutturato l'algoritmo, se si perde anche un solo riferimento, si rischia di non venire a conoscenza di un incrocio e quindi di una potenziale collisione.

Come si può osservare dal diagramma della figura 4.1, si sono esplorare due opzioni per inizializzare la lista degli altri attori presenti.

Soluzione con processo ambiente centralizzato

Nella prima proposta di soluzione si prevede di delegare la creazione di tutti gli attori ad un unico processo ambiente (che potrebbe essere anche un attore). Tale processo ambiente, utilizzando la tecnica dello spawn su *ActorSystem* remoto, avvierebbe quindi tutti gli attori passando direttamente nel costruttore una lista di riferimenti ai nodi già presenti.

Soluzione con registro dei nodi

Come già anticipato in fase di progetto (vedere sotto-sezioni 3.1.2 e 3.3.4) però si è optato per un'altra opzione più elastica, dove gli attori possono essere spawnati da chiunque (dall'host di un interfaccia, da un altro sistema software, ecc.). In questa soluzione prima di iniziare il vero e proprio algoritmo gli attori dei droni contattano un registro dei nodi che:

- fornisce loro una lista dei nodi già presenti;
- registra l'ingresso del nuovo arrivato.

Tale registro è implementato come attore (la classe `DronesRepositoryActor`, non riportata nei diagrammi) e si prevede che venga dispiegato in un Actor-System dall'indirizzo noto. `RegisterDroneActor` è invece l'implementazione di `DroneActor` che prima di eseguire l'algoritmo, effettua una (pre) inizializzazione con il registro.

Gestione dell'uscita dei nodi dall'ambiente

Per il registro è essenziale avere in ogni momento una visione aggiornata di **tutti e soli** i nodi attivi. Pertanto deve essere in grado di aggiornare la lista di questi ultimi anche in caso di uscite non regolari (causa errori non segnalabili). Per fare ciò si è deciso di utilizzare **nell'ambiente** un meccanismo di *Deathwatch* (già implementato in Akka.NET con il metodo `actorContext.Watch(actorRef)`).[2]

4.1.8 Spawn concreto dell'attore e problema della supervisione

In Akka.NET, ci sono diversi modi per dispiegare un attore; a seconda della modalità scelta, ci saranno delle differenze in termini di:

- chi ha la responsabilità di supervisionare l'attore;
- quale sarà l'indirizzo logico dell'attore.

[2] In altre parole l'attore registro genera un attore figlio che invia periodicamente un messaggio (chiamato *heartbeat*) all'attore osservato. Se quest'ultimo non risponde all'*heartbeat* entro un certo tempo o risponde comunicando l'intenzione di terminare, l'attore figlio invia un messaggio `Terminate` al genitore, il quale provvederà a rimuovere il nodo terminato dalla lista. Tutto ciò è gestito da Akka.NET.

Spawn tramite ActorSystem (e variante con spawn remoto)

La prima modalità (di base) è il dispiegamento tramite ActorSystem. Tale modalità prevede semplicemente, data un'istanza del sistema, di spawnare dentro di essa un attore chiamando il metodo `actorSystem.ActorOF(Props, NomeAttore);`. L'attore generato in questo caso:

- viene eseguito sull'ActorSystem;
- ha come indirizzo logico un sotto-path del suo ActorSystem;
- viene supervisionato dal *Guardian* di base dell'ActorSystem secondo una politica configurata (uguale per tutti gli attori del sistema).

Una variante di questa modalità (implementata tramite un'impostazione) permette di dispiegare il nuovo attore in una locazione remota (mantenendo però l'indirizzo logico del sistema spawner e lasciando a quest'ultimo la responsabilità di effettuare la supervisione).

In fase di progetto si immaginava di utilizzare questa modalità di spawn, usando come sistema spawner il processo ambiente centralizzato. In fase di implementazione però si è scelto di rinunciare al processo ambiente; di conseguenza, l'utilizzo dello spawn remoto diventa poco praticabile in quanto questo legherebbe l'attore di una missione al processo dell'interfaccia che ha eseguito lo spawn (causando effetti collaterali spiacevoli, come ad esempio il fatto che la terminazione dell'interfaccia implicherebbe la terminazione anche dell'attore che sta portando a termine la missione).

Delega dello spawn ad un attore

La seconda modalità di spawn offerta da Akka.NET è la delega dello spawn ad un attore. Akka.NET permette agli attori di generare attori figli, i quali:

- vengono eseguiti nell'ActorSystem dell'attore padre;
- hanno come indirizzo logico un sotto-path dell'attore padre;
- vengono supervisionati dal padre, secondo politiche definite via codice.

A causa della rinuncia all'ambiente centralizzato, si è deciso di ricorrere a questa seconda modalità di spawn. Nel concreto:

- per ogni locazione fisica inizializzata (cioè per ogni ActorSystem creato), si crea al suo interno un attore `SpawnerActor`;
- per avviare una missione, invece che agire da remoto sull'ActorSystem, si invia una richiesta allo `SpawnerActor` sotto forma di messaggio;

- lo `SpawnerActor` assume quindi la responsabilità di spawnare attori sul proprio ActorSystem (e di effettuare la supervisione, che al momento avviene secondo una politica `Stop`[3]).

4.2 Tier 2: Gestione degli errori

Come già presentato in sezione 3.2.6 si è previsto di implementare un sistema di scadenze temporali per la ricezione dei messaggi nelle varie fasi di inizializzazione e negoziazione, nello specifico in tre occasioni:

1. in fase di inizializzazione: la ricezione delle tratte dagli altri nodi attivi dopo aver inviato la propria,
2. in fase di negoziazione:
   (a) la ricezione delle metriche dai nodi nel proprio $ConflictSet$ dopo aver inviato la propria
   (b) la ricezione delle intenzioni (volo o sto fermo) dai nodi nel $GPTMSet$ dopo aver ricevuto tutte le metriche

L'implementazione prevede di attivare un timer (utlizzando lo *Scheduler* di Akka.NET) per ognuna delle tre fattispecie calcolando un tempo limite di 10 secondi per ogni messaggio atteso (valore arbitrario eventualmente adattabile). Se tutti i messaggi attesi arrivano prima del tempo la schedulazione viene annullata, altrimenti viene inviato all'attore un messaggio `InternalTimeoutEnded`, il quale provoca il fallimento della missione e il passaggio del nodo in stato $Exit$.

4.3 Tier 3.A e 3.C: Interfacciamento con il sistema e servizi per l'utente

Raggiunto il secondo tier di sviluppo, il sistema di droni funziona correttamente e gestisce anche situazioni di errore dovute alla non raggiungibilità dei nodi. Detto ciò, dall'esterno non si offre ancora alcun modo di osservare il funzionamento del sistema. L'obiettivo di questo tier è predisporre dei meccanismi di base che potranno poi essere usati per accedere al sistema distribuito da un altro sistema software esterno (che potrebbe essere ad esempio un interfaccia, un servizio di prenotazione, ecc.).

[3] Akka.NET offre quattro tipi di politica di supervisione da applicare in caso di crash di un attore: `Resume`, `Restart`, `Stop` ed `Escalate`. Nel nostro caso, per il momento, si è ritenuto che se nell'attore di una missione si verifica un'eccezione non gestita, questo deve terminare (per evitare che finisca in uno stato incoerente e assuma comportamenti bizantini). Per approfondire le politiche di supervisione, si invita a fare riferimento alla documentazione ufficiale: `https://getakka.net/articles/concepts/supervision.html`.

4.3.1 Predisposizione messaggi per comandi utente

Gestione dei messaggi Cancel e GetStatus

La prima forma di interfaccia con il sistema è la predisposizione di un meccanismo *Request-Response* per inviare comandi al drone.

Per il momento, non si prevede di fornire all'utente la possibilità di influire troppo sul funzionamento del sistema, ma ci si limita a implementare due semplici richieste: una richiesta di tipo *GetStatus* (per ottenere un'istantanea dello stato del drone e della missione) e una di tipo *CancelMission* (per richiedere l'annullamento di una missione).

Per ciascuna di queste richieste si sono creati quindi un messaggio `XXXRequest` e un messaggio `XXXResponse`. I messaggi richiesta sono gestiti dall'attore principale del drone `DroneActor` e possono essere inviati da altri attori, oppure anche da un generico processo C# che importa le librerie di *Akka.NET*.

Estrazione dello stato come Data Type Object

Per realizzare la richiesta dello stato è stato necessario implementare un meccanismo che permetta all'attore di estrarre il proprio stato in un formato trasmissibile via messaggio.

Nel nostro caso, si è gestita la rappresentazione dello stato nei messaggi attraverso una gerarchia di classi simmetrica a `DroneActorState`, che sostanzialmente estrare le informazioni più rilevanti sui vari stati in una serie di `DroneStateDTO`. La generazione di questi *Data Type Object* viene gestita da un componente separato chiamato `StateDTOBuilderVisitor`.[4]

Nella prima implementazione dei DTO si sono riscontrati dei problemi di serializzazione; questi problemi si sono risolti applicando le buone norme elencate nell'appendice A.

4.3.2 Meccanismi per l'osservazione dello stato delle spedizioni

Oltre all'interfacciamento *Request-Response*, si è ritenuto utile implementare un meccanismo che permetta di "osservare" una spedizione e ricevere notifiche ogni qual volta che accade un evento rilevante. Per realizzare ciò, si è implementata una soluzione ispirata al desgin pattern dell'*Observer* [3] (adattato ad un contesto distribuito con attori, come mostrato nel diagramma della figura 4.7).

- Ogni attore del drone, al momento del suo spawn, genera (tramite il componente `NotificationProtocol`) un attore figlio a cui delega la gestione del servizio di notifica.

[4] Il componente `StateDTOBuilderVisitor` è stato implementato attraverso il design pattern del Visitor, che sostanzialmente permette di definire una procedura specifica *esterna* per ogni sotto-tipo di una gerarchia di classi (per approfondire: `https://refactoring.guru/design-patterns/visitor`). Nel nostro caso, si usa questo pattern come stratagemma per generare un DTO diverso per ogni stato, evitando però allo stesso tempo di creare una dipendenza diretta tra le classi stato e i loro DTO corrispondenti.

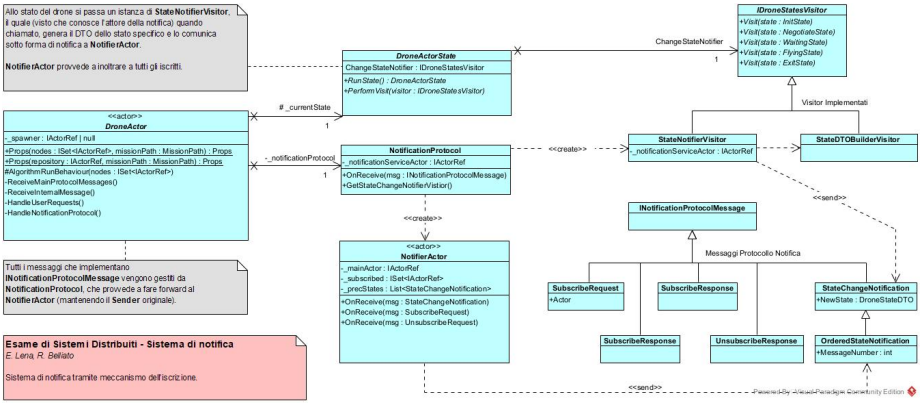

Figura 4.7: Diagramma di classi del servizio di notifica.

- Questo attore (di tipo `NotifierActor`) implementa un classico servizio di notifica basato sul design pattern dell'*Observer*:
  - l'attore figlio ha il compito di gestire una lista di iscrizioni, cioè una lista di attori che hanno affermato la loro volontà di ricevere aggiornamenti attraverso l'invio di un messaggio `SubscribeRequest`;
  - ogni qualvolta accade un evento (comunicato dall'attore principale all'attore figlio come messaggio `StateChangeNotify`), quest'ultimo inoltra la notifica a ciascun iscritto.
- Per fare in modo che dall'esterno si comunichi solo con l'attore base:
  - tutti i messaggi che `NotifierActor` invia, hanno come mittente l'attore base (in Akka.NET ciò si realizza specificando come secondo argomento del metodo `Tell()` l'`IActorRef` del mittente);
  - i messaggi del protollo vengono ricevuti dall'attore base, ma ventono poi inoltrati (con il metodo `Forward()`, che lascia invariato il mittente) a `NotifierActor` per la gestione concreta.
- Per garantire che chi si iscrive in ritardo possa ricevere tutti i messaggi precedenti, `NotifierActor` tiene una lista di tutte le notifiche passate (in modo da poterle inoltrare ai nuovi iscritti).

4.3.3 API Object Oriented per il programmatore dell'interfaccia

Per semplificare l'accesso al sistema a chi dovrebbe implementarne un'interfaccia, si sono realizzate una serie di *classi API*. Tali classi hanno lo scopo di

incapsulare il dispiegamento degli attori e i sistemi di monitoraggio delle missioni predisposti (spiegati nelle sotto-sezioni 4.3.1 e 4.3.2).

Nell'appendice B si riporta un esempio di come le API possono essere usate per avviare e monitorare una missione, anche da diversi dispositivi.

Interfacciamento con i droni delle missioni (fig. 4.8)

`IMissionAPI` è l'interfaccia base delle API di monitoraggio delle missioni. `IMissionAPI` definisce i metodi delle operazioni che si possono svolgere su una missione (che coincidono praticamente con i messaggi utente che l'attore può ricevere); `SimpleMissionAPI` implementa queste operazioni di base usando la primitive di Akka.NET `Ask()` per inviare i messaggi (con un attore temporaneo) e gestire la risposta.

`ObserverMissionAPI` invece è un'API più sofisticata che fornisce al programmatore la possibilità di monitorare le notifiche emesse dal drone (si veda la sotto-sezione 4.3.2). Ciò viene realizzato dispiegando nell'ActorSystem locale un attore di tipo `ObserverActor`, il quale:

- si iscrive al servizio di notifica del drone;
- accumula le notifiche che riceve in una coda;
- su richiesta dell'API, "svuota" la coda ritornando tutte le notifiche ricevute (alla chiamata del metodo `AskForUpdates()`, l'API effettua tramite `Ask()` una richiesta all'attore observer; se ci sono notifiche in coda, l'observer risponde subito, altrimenti tiene la richiesta in attesa fintanto ché non arrivano notifiche da comunicare).

Si noti che tutti i metodi delle API delle missioni *sono asincroni* e *possono fallire per timeout* (ad eccezione di `AskForUpdates()`, che è predisposto per essere una sorta di "attesa sincrona di notifica").

API per il dispiegamento (fig. 4.9)

Le procedure di dispiegamento degli attori del sistema (attori del drone e attori del registro) sono state anch'esse incapsulate in classi API. In particolare, si prevede che il programmatore lavori come descritto di seguito.

- Nel codice che viene eseguito sui droni, si inizializza un ActorSystem e si genera all'interno di esso l'attore spawner (per farlo, si può utilizzare una classe `ActorSystemFactory` - non riportata nei diagrammi - che permette sostanzialmente di creare il sistema e avviare lo spawner con un unica chiamata di metodo).
- Nel codice dell'interfaccia/del servizio di prenotazione il programmatore usa la classe `DroneDeliverySystemAPI` per:
  - impostare un registro (oppure, se non esiste, avviarlo in una certa locazione remota);

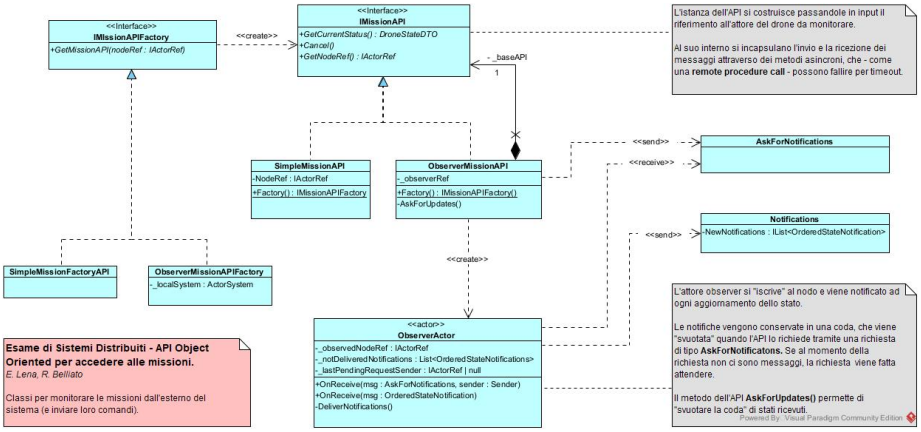

Figura 4.8: Diagramma di classi delle API per le missioni

- avviare missioni tramite il metodo `SpawnMission` (usando il parametro `Host` per specificare indirizzo e porta del drone);
- connettersi a missioni esistenti con il metodo `ConnectToMission`;
- utilizzare poi le istanze di `IMissionAPI` resituite in output per monitorare le missioni.

4.3.4 Interfaccia utente da terminale

Come interfaccia utente si è deciso di implementare un'applicazione da terminale nel quale si inseriscono dei comandi e si ottiene un output da ciascuno di essi. L'applicazione lavora in modalità *stateful*, ossia i comandi manipolano lo stato interno dell'applicazione e da questo dipende anche l'output degli stessi. In particolare vengono memorizzati gli ActorSystem generati (con le relative porte TCP utilizzate) e le missioni generate e/o "osservate" da remoto.

Per il parsing dei comandi è stata utilizzata la libreria `CommandLine`;[5] lo stile dei comandi è quello utilizzato nei sistemi UNIX.

Per maggiori informazioni sui comandi si rimanda alla repository del progetto `https://github.com/rik1599/DistributedSystemsExam`.

[5] `https://github.com/commandlineparser/commandline`

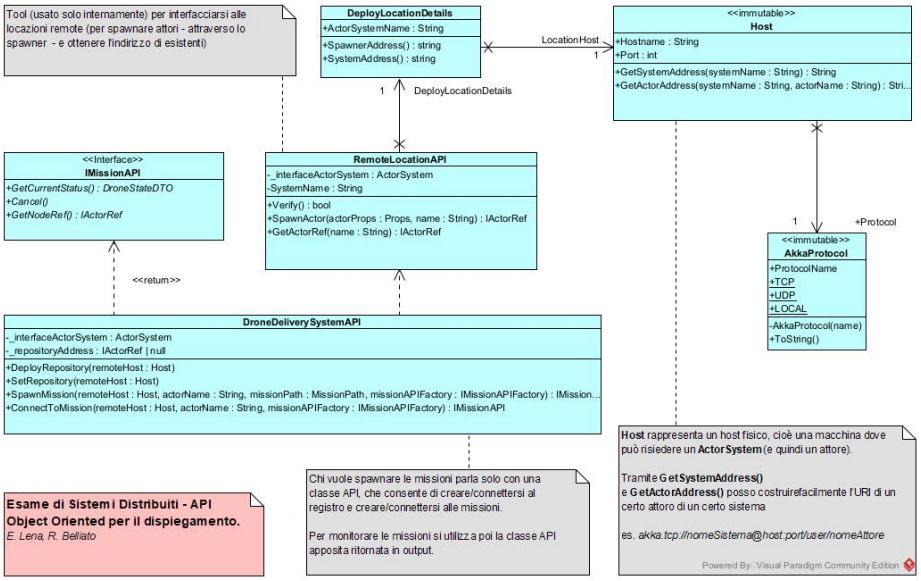

Figura 4.9: Diagramma di classi delle API per lo spawn di missioni e del registro.

Capitolo 5

Validazione del sistema

5.1 Test unitari con XUnit

5.1.1 Testing automatizzato applicato al contesto degli attori

La prima forma di validazione introdotta sono i test unitari automatizzati sviluppati con `xUnit` (e con il corrispondente `TestKit` fornito da Akka.NET).

Questi test consistono sostanzialmente nella valutazione del comportamento degli attori (con un approccio black box). In pratica, per ogni caso di test, si svolgono tre attività:

1. si dispiegano (e si inizializzano) uno o più attori in un ambiente controllato;
2. si interagisce con essi, fingendo di essere un altro attore, un utente, un registro, ecc.
3. si verifica la corretta esecuzione dei protocolli tramite una sequenza di asserzioni (che verificano ad esempio che i messaggi ricevuti siano quelli attesi).

Queste tre attività vengono eseguite tutte in automatico, in quanto sono definite via codice all'interno di metodi C#. In questo modo, si rende possibile eseguire con un unico click un intera batteria di test **sempre uguali** (da qualche decina, fino anche a centinaia), senza che vi sia necessità di intervento umano. Grazie a ciò, ogni qual volta si effettua una modifica al codice, si può testare rapidamente la sua retro-compatibilità.

5.1.2 Casi di test sugli attori

Di seguito, si elencano le principali situazioni rappresentate.

1. *Test di base sugli attori delle missioni*

(a) spawn di una missione in uno spazio libero (ci si aspetta che parta);

(b) spawn di una missione in uno spazio non libero, ma comunque privo di conflitti (ci si aspetta che parta);

(c) simulazione di un conflitto vinto (ci si aspetta che l'attore esegua correttamente il protocollo di negoziazione, e poi parta);

(d) simulazione di un conflitto perso (ci si aspetta che l'attore attenda);

(e) simulazione dello spawn in un ambiente dove c'è già una missione in volo a rischio collisione (ci si aspetta che l'attore effettui la sua attesa, poi parta);

(f) simulazione di un conflitto dove due nodi hanno la stessa metrica (ci si aspetta parta quello con ID minore).

2. *Test di situazioni più complesse*

   (a) simulazione di un caso in cui è previsto un secondo round di negoziazione;

   (b) simulazione di un conflitto senza vera e propria intersezione delle tratte (per verificare il corretto calcolo della collisione con margine).

3. *Test sulla capacità degli attori di gestire errori*

   (a) simulazione di un timeout in fase di inizializzazione;

   (b) simulazione di un timeout in fase di negoziazione;

   (c) simulazione di un timeout nella ricezione delle intenzioni dopo la negoziazione.

4. *Test sul corretto funzionamento del registro* ← si ripetono alcuni dei test precedenti, ma introducendo anche il regsitro dei nodi e verificando il suo corretto funzionamento

5. *Test vari sul servizio di notifica e sulle richieste di stato*

   (a) verifica delle procedure di iscrizione e disiscrizione dal servizio di notifica;

   (b) verifica di ricezione di tutte le notifiche attese;

   (c) verifica della richiesta di stato in alcune delle situazioni precedenti (cielo libero, conflitto vinto, conflitto perso, ecc.).

5.1.3 Casi di test sulle API

Oltre che i test sugli attori, si sono eseguiti anche alcuni test sulle API object oriented.

1. *Test sulla creazione e sull'impostazione del registro*

(a) avvio del registro dei nodi tramite API e verifica del suo corretto funzionamento;

(b) avvio del registro dei nodi senza API, connesione via API e verifica del suo corretto funzionamento.

2. *Test sulle funzioni legate alle missioni*

   (a) avvio di due missioni tramite API ed esecuzione delle richieste di stato;

   (b) avvio di una missione senza API e connessione;

   (c) avvio di una missione e cancellazione di essa;

   (d) tentativo di connessione a missione non esistente;

   (e) tentativo di spawn senza registro (ci si aspetta fallisca);

   (f) tentativo di connesione a missione senza aver fissato il registro (ci si aspetta funzioni);

3. *Test sulle missioni con l'API Observer*

   (a) avvio di una missione e verifica che le notifiche che arrivino siano quelle attese;

   (b) avvio di una missione, creazione di una connessione e verifica che vengano ricevute le notifiche (quelle vecchie e quelle nuove);

   (c) test di retro-compatibilità (si ripetono tutti i test già fatti per le API normali).

5.1.4 Commenti sul testing automatizzato

Il testing automatizzato si è rivelato essere uno strumento molto utile per portare a termine lo sviluppo. Detto ciò, è importante chiarire alcune cose:

1. **In questo progetto NON si è fatto testing unitario esaustivo**. A causa di mancanza di tempo, in questo progetto non è stato fatto "unit testing" nel senso letterale del termine. La metodologia del testing unitario prevede che *per ogni componente* del sistema (*partendo dalle componenti più elementari*, come ad es., le singole classi interne, i metodi, ecc.) si definisca una serie di test che coprano *tutte* le tipologie di input previste (quindi, anche - e soprattutto - un'ampia gamma di situazioni limite diverse, situazioni d'errore, ecc.). Nel nostro caso invece:

   - non si è fatto testing esaustivo sulle singole classi;
   - si sono considerati un numero relativamente piccolo di casi.

   Scrivere dei buoni test più esaustivi avrebbe migliorato significativamente la qualità e la manutenibilità del progetto, ma avrebbe anche richiesto molto più tempo (oltre che un'attenzione molto più importante per ogni singolo componente).

2. **L'ambiente simulato non è quello reale**. Si noti che tutti i test descritti sono stati eseguiti in un ambiente controllato, con un unico ActorSystem piuttosto particolare (quello fornito dal TestKit di Akka.NET).

3. **Di per sé, il testing unitario non garantisce la correttezza**. È importante notare che è sempre valido il principio di base del testing, cioè che il superamento dei test non è una garanzia di correttezza del programma; al contrario, il fallimento di un test è una prova di non correttezza.

5.2 Interfaccia console per il prototipo

5.2.1 Come usare l'interfaccia

L'interfaccia da console può essere usata per validare il prototipo in un ambiente simil-reale, avviando una o più istanze della console (su una o più macchine).

Con la console si può operare in due modi: esclusivamente in locale, oppure facendo comunicare più macchine. Nel test esclusivamente in locale, la procedura tipica è la seguente:

- si avvia una o più istanze della console;
- si inizializzano su più ActorSystem usando porte diverse, ripendendo il comando

  `create-actor-system -pPORTA`

  (non specificando l'host, viene assegnato come valore `localhost`);
- si dispiega il registro su un certo ActorSystem con il comando

  `spawn-repository -pPORTA`

  Sulle altre console, si imposta il registro con il comando

  `set-repository -pPORTA`
- si dispiegano le missioni con il comando

  `spawn-mission x1 y1 x2 y2 -nNOME -pPORTA`

  e vi ci si connette dalle altre console con

  `set-mission -nNOME -pPORTA`
- si utilizzano i comandi

  `ping -nNOME -pPORTA`

  e

  `log -nNOME -pPORTA`

per monitorare le missioni.

Per il test su più macchine invece si aggiunge ai vari comandi l'opzione `-h` per specificare gli indirizzi IP. Si noti che un actor system a cui è stato assegnato `localhost` come indirizzo, non sarà visibile in rete.

Per maggiori dettagli su tutti i comandi, si può consultare il README della repository del progetto.[1]

5.2.2 Test con interfaccia Nro 01: Spawn remoto (fig. 5.1 e 5.2)

Il primo test effettuato tramite interfaccia prevede, dati due host (appartenenenti alla stessa rete locale), di:

1. creare un paio di actor system;
2. creare un registro dei nodi;
3. connettersi al registro da un'altra interfaccia;
4. spawnare (in remoto) una missione su un ActorSystem;
5. utilizzare il comando `log` (per verificare la partenza del nodo);
6. utilizzare il comando `ping` (per verificare lo spostamento nello spazio);
7. creare una connessione alla missione da un host diverso da quello che lo ha creato (e ripetere `log`);
8. cancellare la missione (e verificare tramite `log` lo stato di terminazione).

5.2.3 Test con interfaccia Nro 02: attesa di nodo in volo (fig. 5.3)

Il secondo test è stato effettuato proseguendo dalla stessa disposizione del test Nro 01. In ordine, si sono svolte le seguenti operazioni:

9. si spawna una missione ($B1$), che partirà subito in volo;
10. si spawna una seconda missione $B2$, la cui tratta incrocia quella della missione in volo; con il comando `log` verifico che $B2$ si sia messa in attesa;
11. con `ping` verifico che il tempo d'attesa decresca;
12. dopo un po', verifico che $B2$ sia partita;
13. dopo un po', verifico che $B2$ sia terminata (per raggiungimento della destinazione).

[1] `https://github.com/rik1599/DistributedSystemsExam`

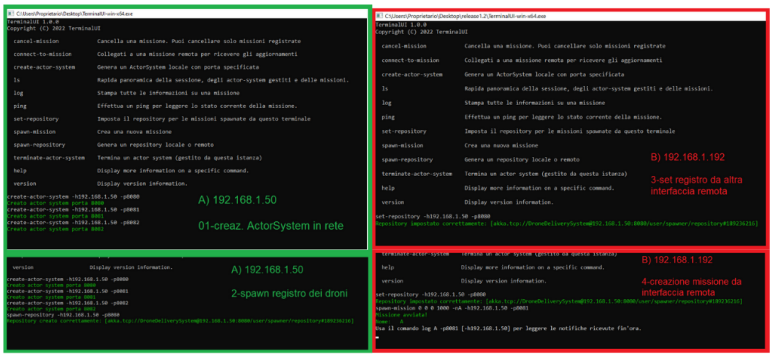

Figura 5.1: Test con interfaccia Nro 01 - I parte: spawn degli ActorSystem, del registro e della missione.

Figura 5.2: Test con interfaccia Nro 01 - II parte: comandi sulla missione.

Figura 5.3: Test con interfaccia Nro 02: attesa di una missione in volo.

Capitolo 6

Conclusioni

6.1 Risultati ottenuti

Di seguito, si riepilogano i tier di sviluppo prefissati (sezione 3.5) e si commentano i risultati ottenuti.

6.1.1 Tier 1: protocolli e algoritmi fondamentali (COMPLETATO)

Il primo tier di sviluppo prevedeva di implementare il protocollo definito in fase di progetto e la logica di dispiegamento degli attori. Tale tier è stato **completato interamente**.

La parte degli algoritmi è stata implementata con successo nei primi dieci giorni di lavoro, mentre per quanto riguada lo spawn degli attori ci sono stati alcuni intoppi. La logica di spawn attualmente implementata è frutto di almeno tre fasi (spalmate in tre/quattro settimane di lavoro).

1. Nella prima fase, si è partiti con l'idea originale del processo ambiente che dispiega le missioni con lo spawn remoto Akka.NET.
2. Nella seconda fase, si è deciso di sostituire l'ambiente con un registro dei nodi (come già spiegato in fase di progetto). Il dispiegamento diventa quindi responsabilità delle varie interfacce; la modalità di dispiegamento rimane lo spawn remoto (funzionante in ambiente di test).
3. Nella terza fase, ci si è resi conto che lo spawn remoto non era praticabile (per le motivazioni già spiegate in fase di implementazione), quindi si è implementata la soluzione con spawner (contemporaneamente ad un refactoring delle API).

6.1.2 Tier 2: Gestione degli errori (COMPLETATO)

Il secondo tier prevedeva di implementare la gestione degli errori.

Il meccanismo del timeout è stato completato subito dopo l'implementazione dell'algoritmo (in circa due giornate di lavoro). La parte sull'uscita pulita del nodo invece è stata implementata durante l'implementazione dell'algoritmo.

6.1.3 Tier 3: Servizi per l'utente (PARZIALMENTE COMPLETATO)

Il terzo tier di sviluppo prevedeva di realizzare una serie di servizi per l'utente:

1. comandi di richiesta stato e cancellazione missione; ← **completato**, dopo l'algoritmo in circa due/tre giornate di lavoro
2. possibilità di richiere snapshot; ← **non iniziato** (per mancanza di tempo)
3. servizio di notifica. ← **completato**, dopo l'algoritmo e durante lo sviluppo delle API, in circa cinque giorni di lavoro

A questo terzo tier si deve aggiungere anche una parte di refactoring dei DTO per i messaggi, la cui serializzazione ha causato diversi problemi (in quanto erano stati progettati ignorando le buone norme elencate nell'appendice A). Il refactoring dei DTO ha richiesto almeno due giornate di lavoro.

6.1.4 Tier 4: Sistema di interfacciamento aperto (CAMBIATO IN CORSO)

Il quarto tier di sviluppo prevedeva di realizzare una qualche forma di API per interfacciarsi al sistema da un qualche altro sistema software. Inizialmente si aveva pianificato di realizzare un'API REST, ma durante l'implementazione si è deciso di creare invece un'API object oriented. L'API ad oggetti è stata preferita alla web API REST in quanto:

- permette di interfacciarsi al sistema in modo diretto ed elastico, senza passare per un server intermedio;
- potrebbe potenzialmente essere usato in una più ampia gamma di contesti, quali:
  - la realizzazione di diverse interfacce di vario tipo;
  - l'integrazione con altri sistemi software che (in una situazione reale) sarebbero strettamente correlati (es., sistemi che coordinano a più alto livello le tratte, servizi di prenotazione, ecc.);
  - la stessa realizzazione di servizi web o applicazioni web in C#.

Una prima implementazione delle API è stata effettuata in circa quattro/cinque giorni di lavoro. Dopo ti ché è stato fatto un pesante refactoring che ha richiesto altri cinque/sei giorni di lavoro, ma che ha portato ad avere un'API molto più semplice, razionale e coesa.

6.1.5 Sviluppo della console

Come previsto in fase di progetto, si è realizzata un'interfaccia da console. L'interfaccia attuale è frutto di circa due settimane di lavoro (spalmato assieme alla realizzazione dei servizi per l'utente e allo sviluppo e al refactoring delle API).

6.2 Criticità e idee di miglioramenti futuri

Quello che è stato sviluppato è da ritenersi soltanto un prototipo, pertanto presenta qualche criticità e potrebbe essere migliorato in diversi modi. Di seguito, si elencano le principali criticità/possibilità di miglioramento future.

Ri-valutare il processo ambiente

L'eliminazione del processo ambiente ha aiutato a rendere il sistema meno centralizzato, ma ha anche influito negativamente sulla trasparenza di locazione.

Con il processo ambiente che dispiega missioni attraverso il remoting di Akka.NET, si sarebbe potuto mantenere tutti gli attori sotto un unico indirizzo logico. Ciò avrebbe semplificato le procedure di `ActorSelection` per la connessione ad una missione e avrebbe evitando all'utente di dover conoscere host e porta del drone per connettersi ad una missione esistente.D'altro canto però la rinuncia al processo ambiente in favore del registro ha permesso di rendere i nodi più indipendenti e di evitare situazioni spiacevoli nelle quali il processo ambiente fallisce e - a catena - falliscono tutte le missioni. Nella situazione attuale invece il fallimento del registro pregiudica solo l'ingresso di nuove missioni.

In alternativa alla re-introduzione del processo ambiente si potrebbe creare un servizio di "mission discovery" (magari legato al registro dei nodi) che permetta di ottenere l'`IActorRef` delle missioni a partire da un nome.

Effettuare alcuni ultimi aggiustamenti sulle API O.O.

Con le API object oriented riteniamo di aver raggiunto un buon risultato in termini di incapsulamento, semplicità, astrazione e trasparenza. Detto ciò, alcuni aspetti sono ancora migliorabili. In particolare, riteniamo vadano migliorati:

- il meccanismo di creazione degli ActorSystem (quindi l'API "lato drone");
- la gestione degli errori e la generazione di eccezioni (sarebbe interessante modellare con una gerarchia di eccezioni tutti i possibili casi di errore, specie quelli legati ad aspetti di rete);
- la configurabilità (quindi, predisporre un meccanismo di configurazione del sistema).

Migliorare il meccanismo di notifiche

Allo stato attuale i nodi inviano una notifica solo quando si ha un cambiamento dello stato interno, però non si riesce ad osservare nel dettaglio cosa accade nelle singole sottofasi del protocollo.

Sarebbe opportuno rivedere questo meccanismo aumentando la granularità delle notifiche emesse.

Semplificare ed estendere l'interfaccia

L'interfaccia da console sviluppata è solo un prototipo, pertanto è migliorabile sotto diversi aspetti. Uno dei problemi più rilevanti è la verbosità dei comandi; si potrebbe sfruttare meglio il meccanismo dei nomi per evitare di ripetere host e porta. Un secondo aspetto migliorabile riguarda l'uso dell'interfaccia per il dispiegamento in rete: sarebbe utile non dover inserire esplicitamente il proprio indirizzo IP.

Oltre che l'interfaccia da console sarebbe inoltre interessante sviluppare un'interfaccia grafica che permetta di visualizzare (in tempo reale) i droni sul piano cartesiano.

Effettuare un testing di sistema esaustivo

Allo stato attuale la validazione del sistema è stata effettuata prevalentemente a livello di unità (verifica dell'esecuzione corretta del protocollo da parte del singolo attore); riteniamo che vi sia la necessità di integrare tutto ciò con una validazione più esaustiva a livello di sistema, dove si definiscono una serie di situazioni di test, le si esegue (via interfaccia o via codice) e si usa lo strumento delle notifiche per verificare la corretta esecuzione dell'algoritmo.

Questa parte non è stata trattata per mancanza di tempo.

Implementare le parti di tier mancanti (snapshot, API REST)

Per concludere, sarebbe interessante completare i tier di sviluppo prefissati implementando lo snapshot e l'API REST.

Appendice A

Note didattiche

A.1 Alcune note sui messaggi in Akka.NET

Per garantire la corretta serializzazione dei messaggi, in Akka.NET è buona norma che gli oggetti spediti:

- siano immutabili (quindi siano costituiti da proprietà scritte una sola volta nel costruttore e accessibili in sola lettura);
- abbiano tutti i campi accessibili con visibilità `public` (in lettura);
- forniscano almeno un costruttore pubblico che prenda in input tutti i campi (che Akka.NET userà in fase di de-serializzazione);
- specie in caso di uso dell'ereditarietà, evitare di usare i valori di default dei campi e di abusare del meccanismo dell'`override`.

Non rispettare queste pratiche può portare a problemi di serializzazione dei messaggi.

Queste buone norme sono frutto di diverse problematiche riscontrate in fase di sviluppo. Si è ritenuto interessante riportarle, visto che nella documentazione di Akka.NET [2] non si riporta molto sul tema.

Appendice B

Esempio di utilizzo delle API per avviare e monitorare una missione

Di seguito, si presenta un caso d'uso abbastanza tipico delle API. Immaginiamo di voler implementare - **in due sistemi software differenti** - lo spawn di una missione e il suo monitoraggio.

B.1 Spawn della missione su un nodo remoto

Immaginiamo di avere un ActorSystem già avviato su un drone con indirizzo e porta noti (ad esempio, resi pubblici tramite una qualche forma di *registro dei servizi*). Un sistema di prenotazione potrebbe avviare una missione su di esso con il seguente codice:

```
// definisco la tratta da percorrere
MissionPath missionA = new MissionPath(
    new Point2D(100, 250),  // partenza
    new Point2D(500, 350),  // arrivo
    speed);

// creo l'API
DroneDeliverySystemAPI api = new DroneDeliverySystemAPI(
    actorSystemSpawner, "DroneDeliverySystem", "repository");

// imposto al registro (di cui conosco l'indirizzo)
api.SetRepository(new Host("192.168.0.1", 8080));

// avvio la missione
```

```
MissionAPI? missionAPI = api.SpawnMission(
    new Host("192.168.0.123", 8081),
    missionA, "Consegna1234",
    SimpleMissionAPI.Factory());

if (missionAPI is null) throw new Exception("Unable to spawn mission.");
```

B.2 Osservazione di una missione dispiegata su un nodo remoto

Dopo aver avviato la missione, si immagina che il cliente si collega da una qualche forma di interfaccia per monitorare la sua spedizione. Il codice per il collegamento e per la gestione delle notifiche assomiglierebbe al seguente:

```
// creo un'istanza dell'API di base
DroneDeliverySystemAPI api = new DroneDeliverySystemAPI(
    actorSystemInterfaccia, "DroneDeliverySystem", "repository");

// mi collego ad una missione usando l'API di tipo observer
ObserverMissionAPI? obsAPI = (ObserverMissionAPI?)
    api.ConnectToMission(
        new Host("192.168.0.123", 8081), // host del drone
        "Consegna1234", // nome dell'attore
        ObserverMissionAPI.Factory(actorSystemInterfaccia));

if (obsAPI is null) throw new Exception("Unable to connect to mission.");

IList<DroneStateDTO> notifications = new List<DroneStateDTO>();

// ripeto la richiesta di notifiche fintanto
// ché non ricevo un ExitState
do {
    var newNotifications = obsAPI!.AskForUpdates().Result;
    foreach (var n in newNotifications)
    {
        notifications.Add(n);
        HandleNotification(n);  // (gestisco in qualche modo la notifica)
    }
} while (notifications.Last() is ExitStateDTO)
```

Bibliografia

[1] Akka.NET Documentation. Akka.net remote overview. https://getakka.net/articles/remoting/index.html.

[2] Akka.NET Documentation — Akka.NET Documentation. Akka.net documentation: Akka.net documentation. https://getakka.net/.

[3] Observer design pattern. https://refactoring.guru/design-patterns/observer.

[4] State design pattern. https://refactoring.guru/design-patterns/state.

[5] Wikipedia. Insieme indipendente massimale — Wikipedia, the free encyclopedia. http://it.wikipedia.org/w/index.php?title=Insieme%20indipendente%20massimale&oldid=110829433, 2022. [Online; accessed 01-September-2022].

[6] Distributed Computing ETH Zürich. Maximal indipendent set, 2022. https://disco.ethz.ch/courses/podc_allstars/lecture/chapter7.pdf.