

DroneControlSystem

Adan Formisano 1813917

Marco Castellani 1718187

Rory Ucer 1934865

Indice

- [Indice](#)
- [DroneControlSystem](#)
 - [Esecuzione](#)
 - [Requisiti di esecuzione](#)
- [Descrizione generale](#)
 - [Fini del sistema](#)
 - [Modello concettuale ed illustrazione del sistema](#)
 - [Modello concettuale del sistema](#)
 - [Struttura dell'area sorvegliata](#)
 - [Stati del sistema](#)
 - [Stati di guasto dei droni e TestGenerator](#)
 - [Visualizzare il sistema](#)
 - [Area da sorvegliare](#)
 - [To starting line](#)
 - [Working](#)
 - [To base](#)
 - [Contesto del sistema](#)
- [User requirements](#)
 - [Use case utente](#)
- [System requirements](#)
 - [Architectural system diagram](#)
 - [Activity diagram creazione Wave e droni](#)
 - [State diagram Drone](#)
 - [Message sequence chart diagram carica Drone](#)
- [Monitors](#)
 - [WaveCoverageMonitor \(funzionale\)](#)
 - [AreaCoverageMonitor \(funzionale\)](#)
 - [Drone Charge \(non-funzionale\)](#)
 - [Drone Recharge \(funzionale\)](#)
 - [System Performance \(non-funzionale\)](#)
- [Implementation](#)
 - [Implementazione software](#)
 - [Implementare il sistema](#)

- [Componente ScannerManager](#)
- [Componente DroneControl](#)
- [Componente ChargeBase](#)
- [Componente Drone](#)
- [Componente Wave](#)
- [Componente TestGenerator](#)
- [Componente Database](#)
- [Altre componenti](#)
 - [Componente Main](#)
 - [Componenti Monitor](#)
- [Schema del Database](#)
 - [Tab `drone_logs`](#)
 - [Tab `wave_coverage_logs`](#)
 - [Tab `area_coverage_logs`](#)
 - [Tab `system_performance_logs`](#)
 - [Tab `drone_charge_logs`](#)
 - [Tab `drone_recharge_logs`](#)
- [Connessioni Redis](#)
- [Risultati Sperimentali](#)
 - [Correttezza del sistema](#)
 - [Avvio del sistema](#)
 - [Vita completa dei droni](#)
 - [TestGenerator comparato a DB](#)
 - [Scenario *Everything is fine*](#)
 - [Scenario *Drone failure*](#)
 - [Scenario *High consumption*](#)
 - [Scenario *Charge rate malfunction*](#)
 - [Scenario *Connection lost*](#)
 - [Monitor](#)
 - [Copertura dell'area](#)
 - [Consumo anomalo](#)
 - [Ricarica anomala](#)
 - [Performance del sistema](#)

DroneControlSystem

DroneControlSystem è un progetto simulante un sistema di sorveglianza basato su droni volanti che monitorano un'area di 6×6 Km.

Il sistema è sviluppato come progetto d'esame per [Ingegneria del software](#), corso tenuto dal prof [Enrico Tronci](#) a [La Sapienza](#), ed è basato sul progetto gentilmente proposto dal prof nel `main.pdf` [qui](#), al punto 4.2 *Controllo formazione droni*.

Esecuzione

Il progetto possiede un `run.sh` che permette di eseguire il sistema in maniera semplice. Per eseguire il sistema, basta eseguire il comando `./run.sh` da terminale (lo script potrebbe richiedere i privilegi elevati per la creazione del database e del suo owner).

In alternativa è possibile effettuare `build` manualmente il utilizzando `cmake` e poi eseguire il binario generato.

Requisiti

- PostgreSQL
- [redis-plus-plus](#)

Descrizione generale

Fini del sistema

Il sistema progettato è basato, come detto, su una delle tracce di progetto fornite dal prof Tronci. La traccia è la seguente:

Si progetti il centro di controllo per una formazione di droni che deve sorvegliare un'area di dati. Ogni drone ha un'autonomia di 30 minuti di volo ed impiega un tempo di minimo $2h$ e massimo $3h$ per ricaricarsi. Il tempo di ricarica è scelto ad ogni ricarica uniformemente a random nell'intervallo $[2h, 3h]$. Ogni drone si muove alla velocità di $30Km/h$. L'area da monitorare misura 6×6 Km. Il centro di controllo e ricarica si trova al centro dell'area da sorvegliare. Il centro di controllo manda istruzioni ai droni in modo da garantire che per ogni punto dell'area sorvegliata sia verificato almeno ogni 5 minuti. Un punto è verificato al tempo t se al tempo t c'è almeno un drone a distanza inferiore a 10 m dal punto. Il progetto deve includere i seguenti componenti:

1. Un modello (test generator) per i droni
2. Un modello per il centro di controllo
3. Un DB per i dati (ad esempio, stato di carica dei droni) ed i log
4. Monitors per almeno tre proprietà funzionali
5. Monitors per almeno due proprietà non-funzionali

Il sistema si occupa quindi di verificare che ogni punto dell'area sia sorvegliato ogni cinque minuti, e, in caso contrario, segnala eventuali anomalie.

Modello concettuale ed illustrazione del sistema

Modello concettuale del sistema

Il sistema si compone di una **base centrale** (composta da componenti come `ChargeBase`, `DroneControl`, `ScannerManager`) situata al centro dell'area, che funge da punto di partenza e ricarica per i droni.

La `ChargeBase` è l'unico punto dell'intera area in cui i droni si trovano in uno stato di **non volo** e gestisce la ricarica di ciascun drone dopo un suo giro di perlustrazione. Più precisamente gli stati di non volo sono due, ossia `CHARGING` e `IDLE`, e il giro di perlustrazione corrisponde allo stato di `WORKING` del drone. Per spiegare come ogni drone adempie alla verifica dei punti, vediamo come l'area è concettualmente strutturata.

Struttura dell'area sorvegliata

L'area da sorvegliare è un quadrato di $6 \times 6 \text{ Km}$ (36 Km^2), suddiviso in una griglia regolare composta da quadrati di lato `20m` ciascuno. La griglia ha quindi 300 righe e 300 colonne, ed un totale di 90.000 quadrati.

Partendo dalla richiesta della traccia abbiamo pensato di vedere questi quadrati come delle *celle* con al proprio centro il punto da verificare per il drone. Quest'ultimo condivide infatti l'istante di tempo t in cui è coperto con ogni altro punto nella cella, facendo sì che al passaggio del drone sul punto al t -esimo istante di tempo, l'intera area del quadrato della griglia risulti simultaneamente coperta - dove il tempo è rappresentato, nel nostro sistema, da un'unità di tempo chiamata `tick` (un tick equivale a `2.4 s`).

Chiamiamo quindi `starting_line` la colonna di celle coincidenti col lato sinistro dell'area.

Per rispettare il requisito di sorveglianza di ogni punto almeno ogni 5 minuti, raggruppiamo i droni in gruppi di 300 che chiamiamo **onde**. Una volta formata l'onda, questa parte dalla base verso la `starting_line`, ovviamente i droni si muoveranno in diagonale e quindi non tutti arriveranno alla `starting_line` allo stesso istante.

È importante considerare che i primi 212 tick sono quelli in cui i droni raggiungono la `starting_line`, e non sono dedicati alla verifica dei punti.

Quando ogni drone è arrivato alla `starting_line`, l'onda parte col sorvegliare l'area. Questo processo si ripete ogni cinque minuti. È importante notare come il momento in cui l'ultimo drone della nuova onda arriva alla `starting_line` coincide col momento in cui l'onda precedente avrà lavorato per esattamente cinque minuti.

Con onde di droni partenti ogni cinque minuti dalla `starting_line`, ogni punto dell'area è verificato ogni cinque minuti: quando un punto sulla linea di quadrati che il drone percorre sarà stato verificato, esso lo sarà di nuovo entro i prossimi cinque minuti grazie al drone della nuova onda che arriverà a sorvegliarlo.

Questo sistema forma un meccanismo ad onde che è possibile vedere nelle immagini a seguire, in cui nel lifetime di una simulazione è possibile osservare il susseguirsi di diverse onde di droni, ciascuna delle quali copre naturalmente per intero l'area da sinistra a destra.

Stati del sistema

Facciamo ora ordine circa i possibili stati di un drone:

0. Avvio simulazione

Ovviamente questo stato viene eseguito una sola volta all'avvio della simulazione. I primi 300 droni (con carica completa) vengono generati al centro dell'area.

1. Partenza droni (`TO_STARTING_LINE`)

Carichi al 100%, i droni generati scelti da `ScannerManager` partono dalla `ChargeBase` verso il lato sinistro dell'area, per posizionarsi lungo la `starting_line`. Avremo perciò 300 droni pronti a partire dal lato sinistro dell'area, uno per ogni quadrato.

2. Attesa degli altri droni (`READY`)

I droni giunti alla `starting_line` non passano subito a `WORKING`, ma entrano in uno stato di attesa chiamato `READY`, in cui rimangono fin quando ognuno dei 300 droni non è arrivato alla `starting_line` ed è passato a sua volta a `READY`.

3. Copertura dell'area (`WORKING`)

Dopo che tutti i droni sono entrati in `READY`, essi entrano nello stato di `WORKING`. Iniziano quindi il loro volo a 30 Km/h in linea retta (parallela alla base dell'area) verso il lato destro del perimetro dell'area, passando sopra ciascun checkpoint (punto) al centro dei 300 quadrati che separano la `starting_line` dal lato destro dell'area. Ogni volta che un drone sorvola un punto, lo verifica, verificando al contempo tutta l'area del quadrato di cui il punto è il centro.

4. Ritorno alla base (`TO_BASE`)

Quando un drone/onda (possiamo usare i termini in maniera intercambiabile, perché il movimento di un drone è equivalente a quello di un'onda) raggiunge il lato destro dell'area, termina il suo lavoro di verifica dei punti, e passa allo stato `TO_BASE`. In questo stato non fa altro che tornare verso il **centro** dell'area per ricaricarsi, ed essere riutilizzato in un nuovo viaggio di copertura.

5. Ricarica droni (`CHARGING`)

Giunti alla base, i droni vengono ricaricati da `ChargeBase`. In particolare, esso è composto da slot che accolgono i droni che vengono a ricaricarsi - uno slot per drone. Il tempo di ricarica, come richiesto dal requisito di progetto, ha una durata nell'intervallo di $[2, 3] h$ (questo valore è rigenerato ad ogni nuova ricarica del drone), che corrispondono rispettivamente a 3000 e 4500 tick.

6. Attesa in base (`IDLE`)

A carica completa (e non prima), i droni sono messi a disposizione di `ScannerManager` per essere riutati nel creare una nuova onda.

Stati di guasto dei droni e TestGenerator

Durante uno qualsiasi degli stati di volo (`TO_STARTING_LINE`, `READY`, `WORKING`, `TO_BASE`), i droni possono entrare in uno dei seguenti fault state:

- **DEAD**: Il drone subisce un malfunzionamento critico e diventa irrecuperabile.
- **DISCONNECTED**: Il drone perde la connessione con la base e tenta di riconnettersi.
- **HIGH_CONSUMPTION**: Il drone consuma più carica del previsto e continua a operare fino a quando la carica non si esaurisce.

I droni con stato `DISCONNECTED` possono recuperare la connessione (`RECONNECTED`) tornando quindi allo stato precedente la disconnessione, oppure passare a `DEAD` se la connessione non viene ristabilita (in questo caso è la base che prende la decisione di smettere di tentare la riconnessione col drone).

I droni in `HIGH_CONSUMPTION` possono non riuscire ad arrivare alla base. In tal caso passano a `DEAD`.

Si noti che `HIGH_CONSUMPTION` è un "meta-stato". Nel SUD non compare come uno stato vero e proprio (gli altri, ad esempio, sì, essendo definiti in classi apposite), ma è semplicemente una condizione del drone in uno stato di volo che vede il proprio consumo moltiplicato di un fattore casuale scelto in un range di valori plausibile per uno stato di alto consumo di energia. Lo stesso dicasi per lo stato `IDLE`, che è presente nel sistema in qualità di un insieme contenente i droni carichi e disponibili al riuso.

Nel caso in cui il drone sia in `ChargeBase`, c'è un altro scenario che può colpirlo:

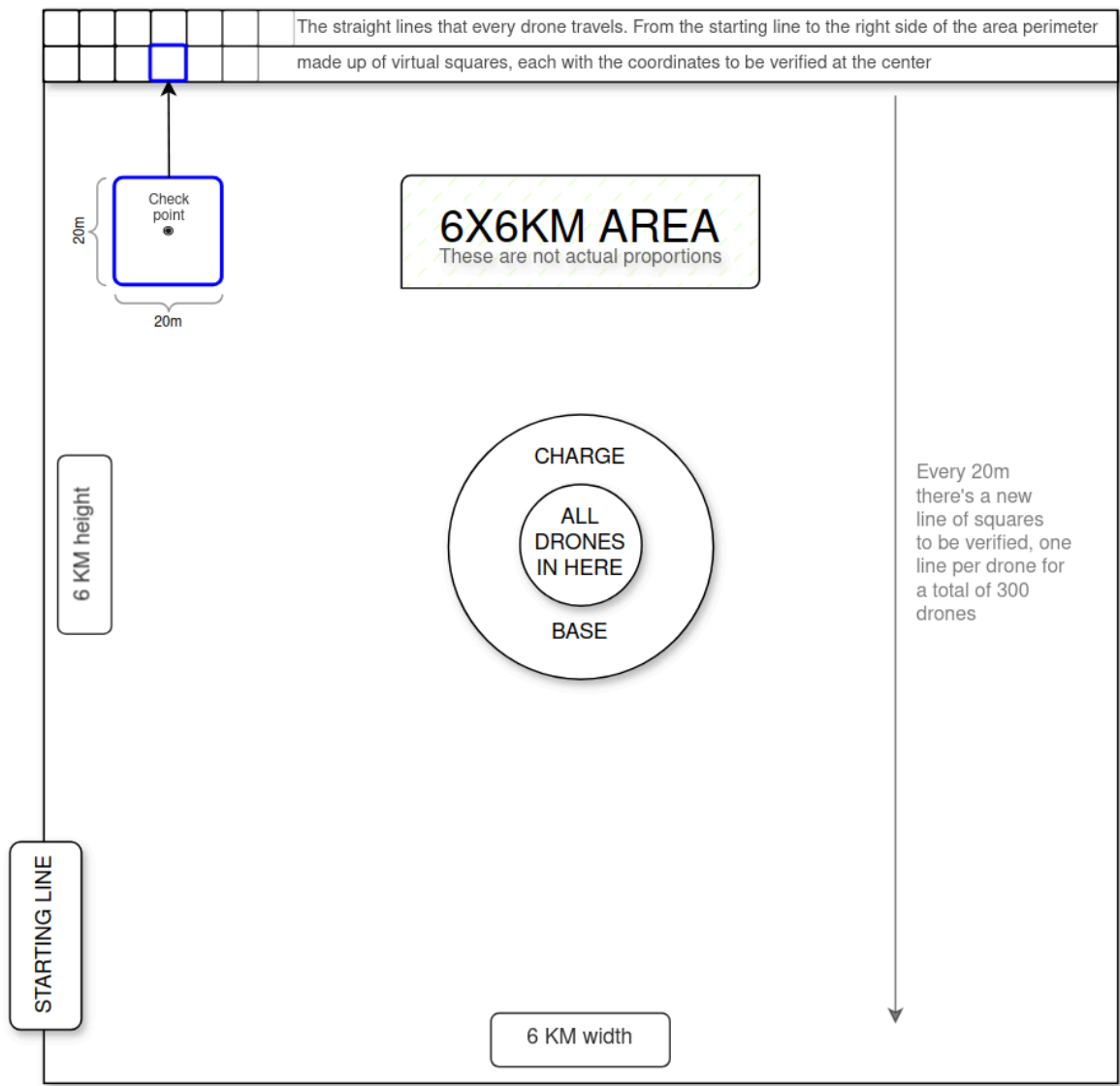
- **RECHARGE_MALFUNCTION:** Il tempo di ricarica è fuori dal range prestabilito (quindi fuori da $[2, 3] h$).

Ogni fault state è conseguenza di uno scenario attivato dal TestGenerator, che è l'entità adibita alla generazione tramite generatori pseudocasuali di avvenimenti riguardanti l'environment di `DroneControlSystem`.

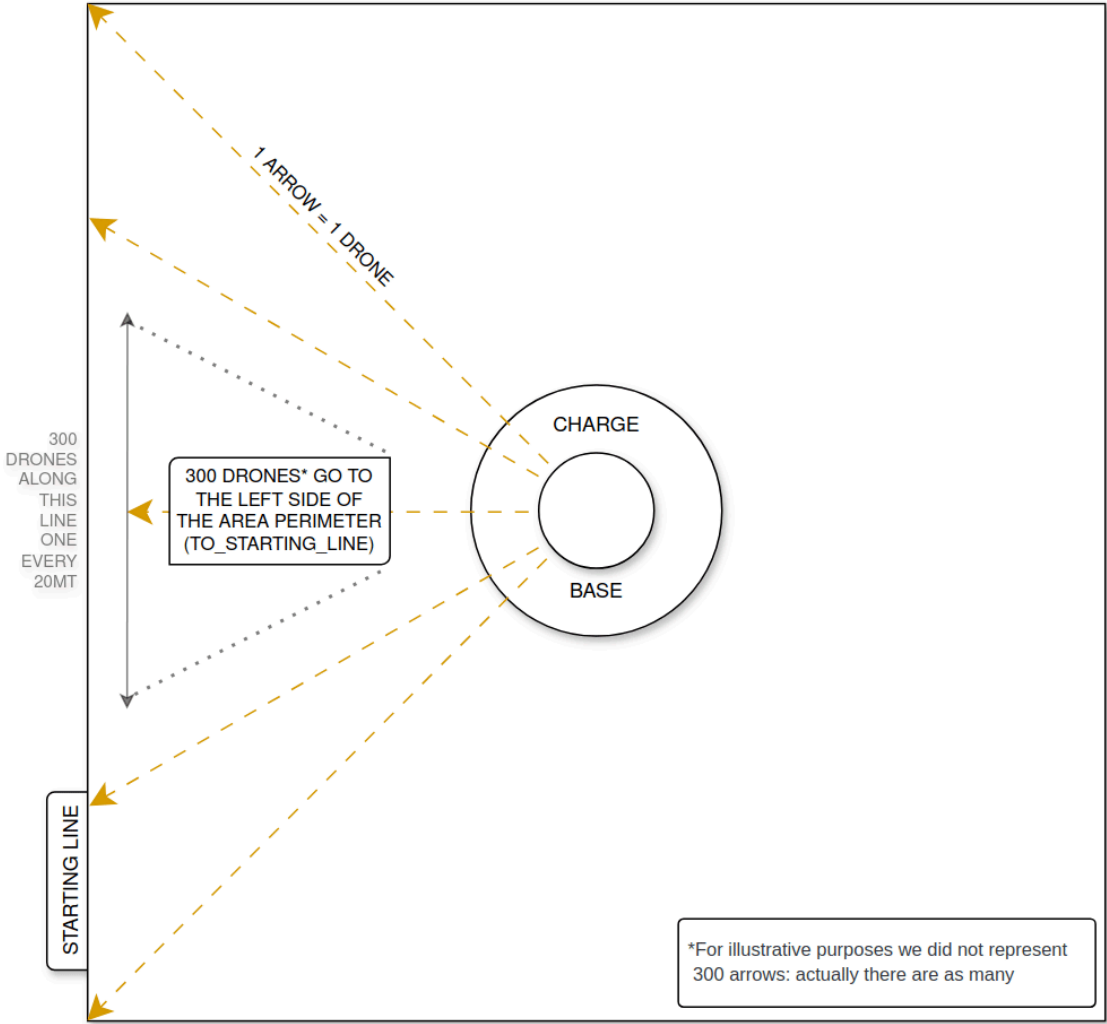
Visualizzare il sistema

La seguente è una vista ad alto livello dell'area, delle componenti del sistema e di alcune delle fasi in cui sono coinvolte

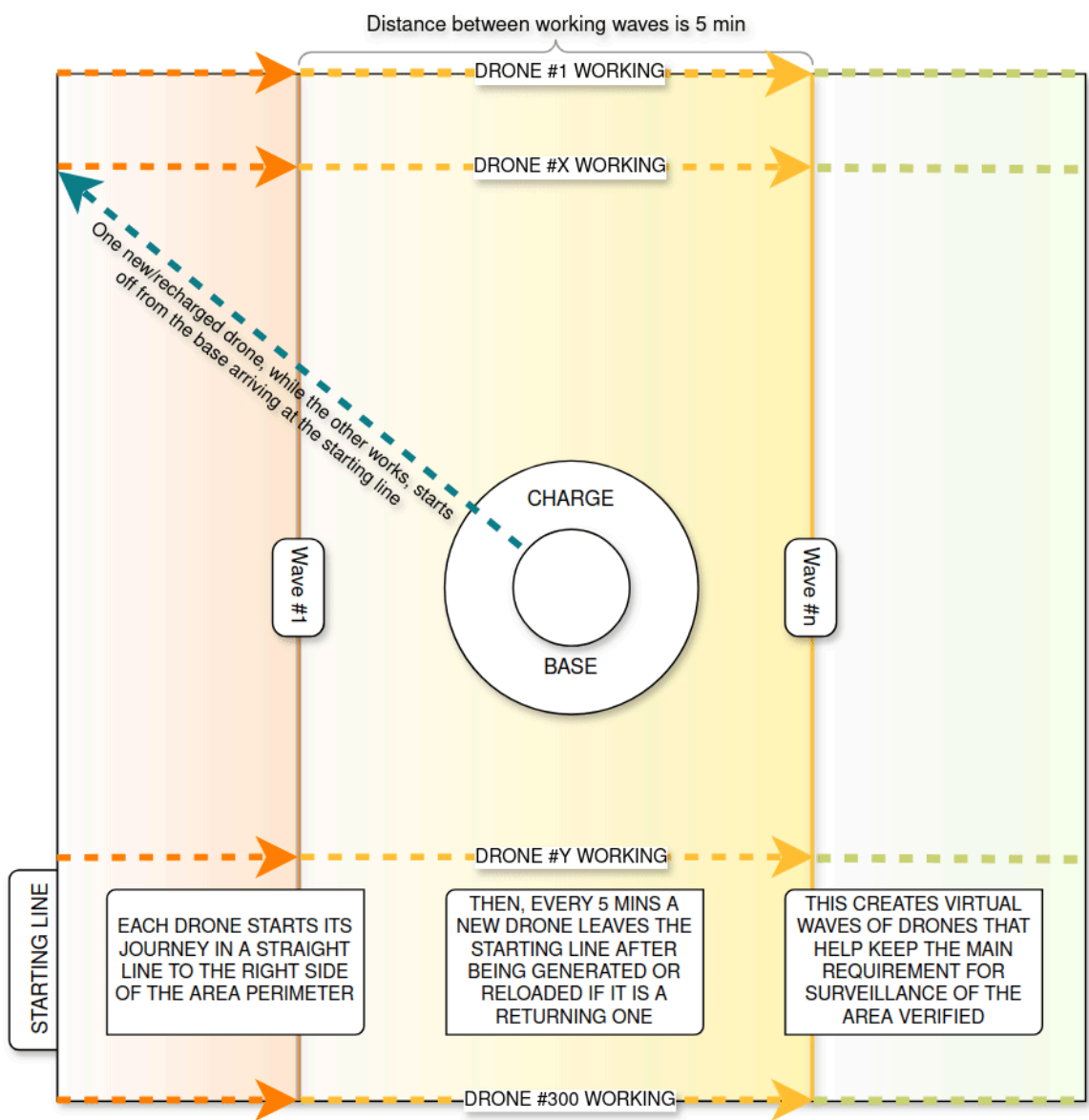
Area da sorvegliare



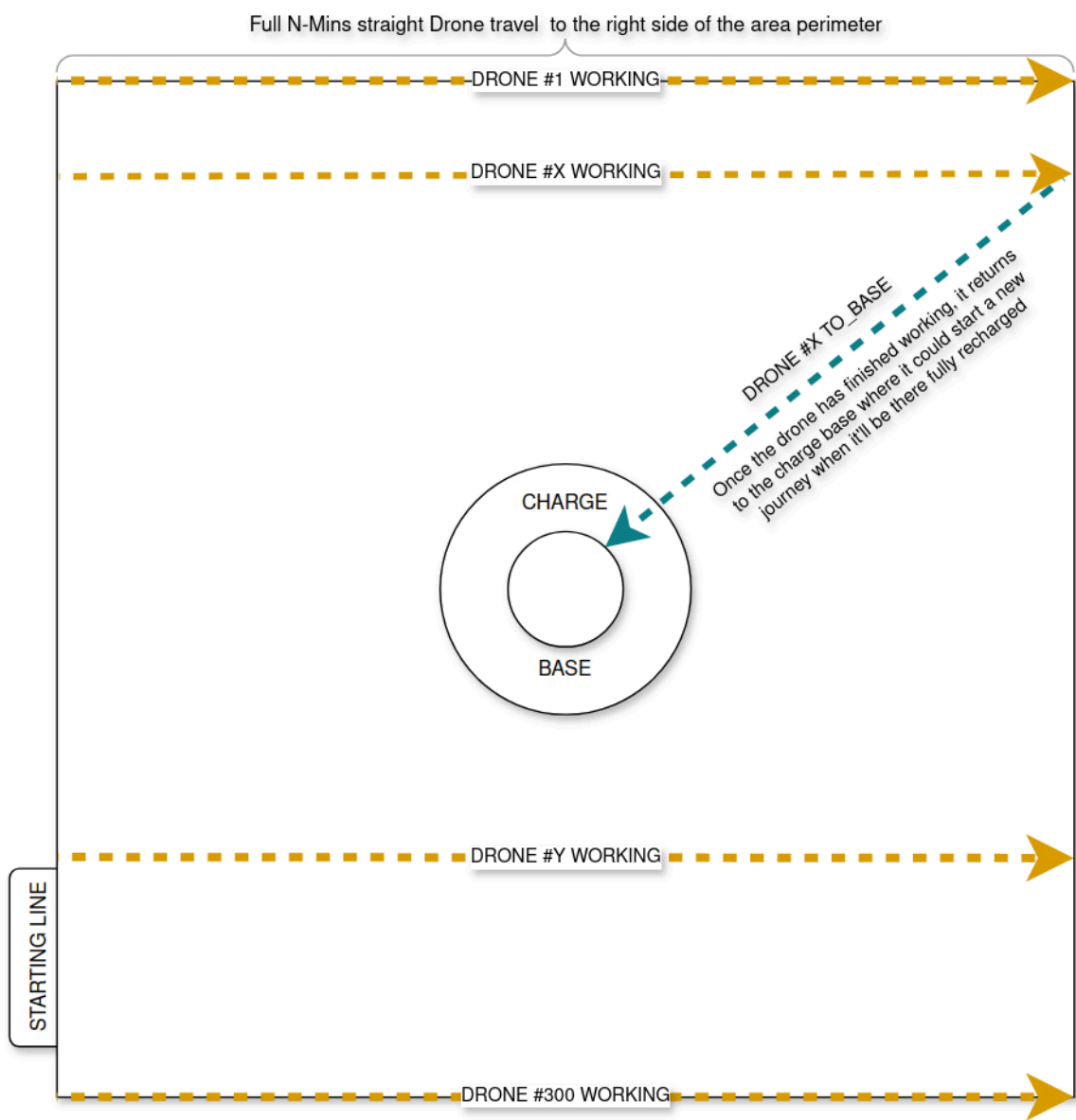
To starting line



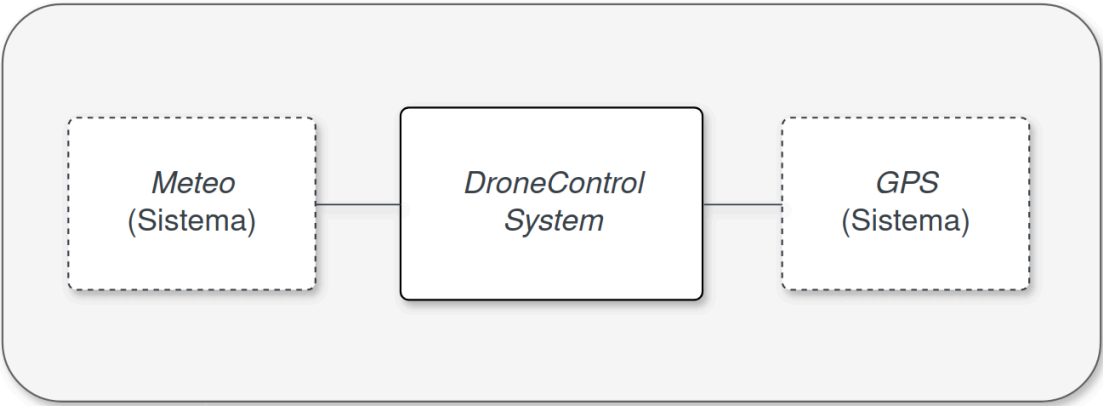
Working



To base



Contesto del sistema



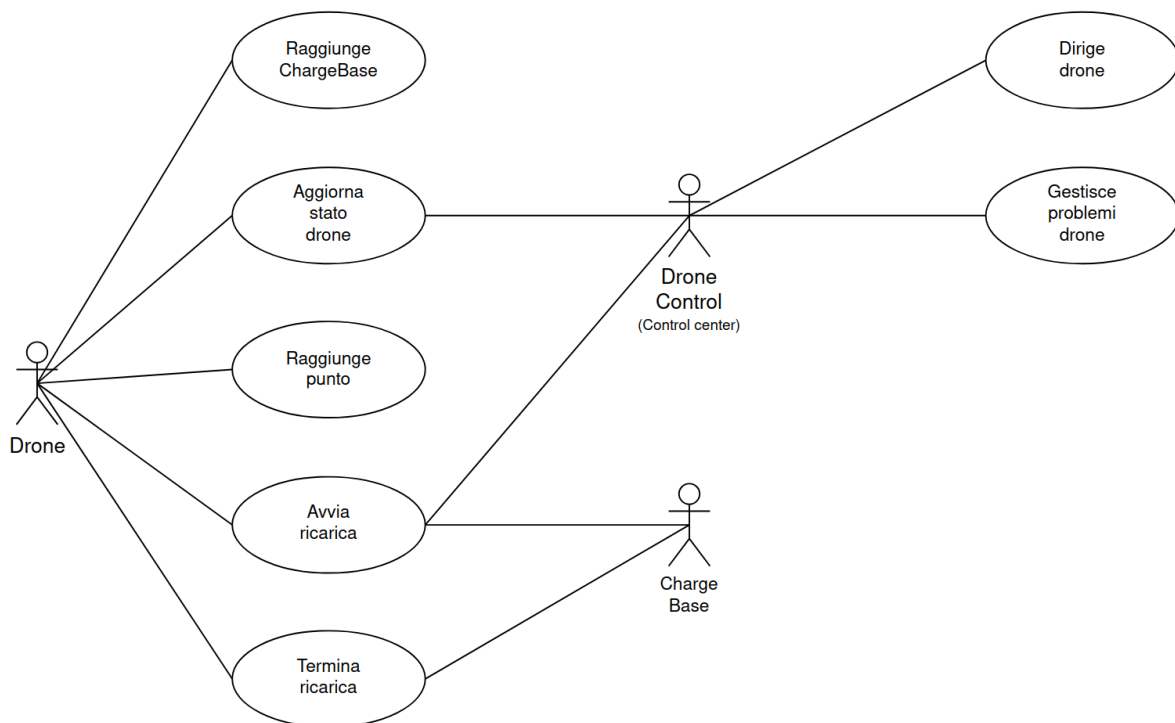
User requirements

Questi sono i requisiti utente che riflettono le esigenze e le aspettative degli utenti del sistema:

- **(1) Area di Sorveglianza:** L'area da monitorare misura 6×6 Km e tutti i suoi punti devono essere verificati ogni 5 minuti.
- **(2) Centro di Controllo e Ricarica:** Il centro di controllo e ricarica si trova al centro dell'area da sorvegliare.
- **(3) Autonomia e ricarica dei droni:** Ogni drone ha 30 minuti di autonomia e deve ricaricarsi in un tempo compreso tra le $[2, 3]$ ore

Use case utente

Questi diagrammi use-case contengono scenari d'uso del sistema da parte dei vari suoi attori



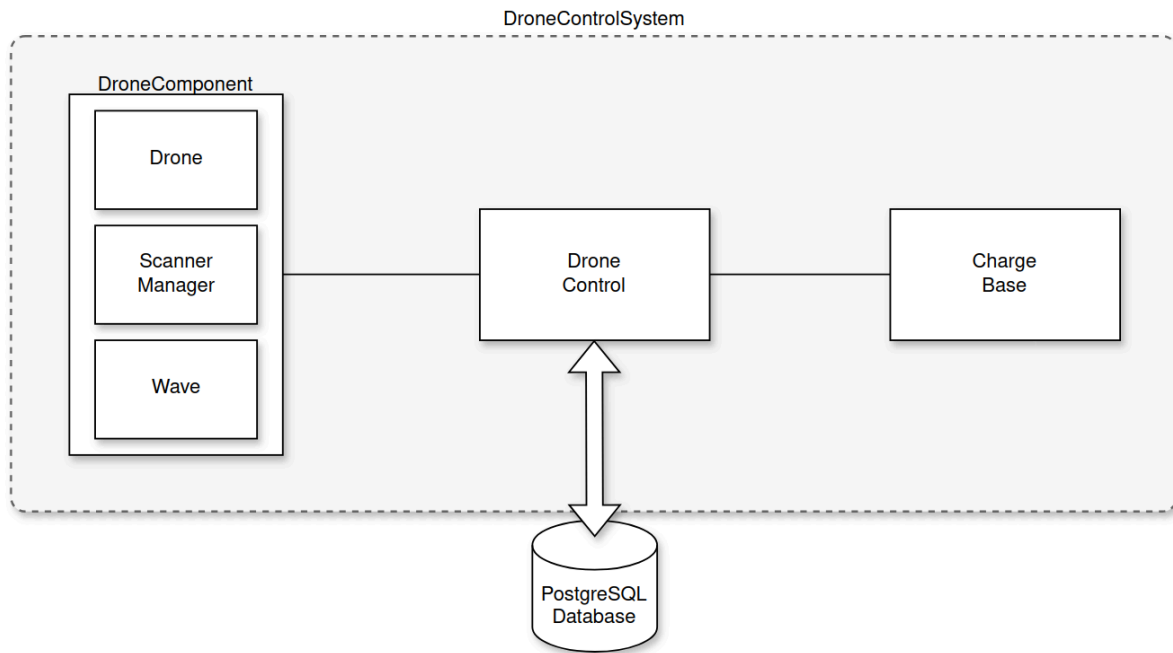
System requirements

Questi requisiti sono i requisiti di sistema che dettagliano le specifiche tecniche e le funzionalità necessarie per implementare il sistema:

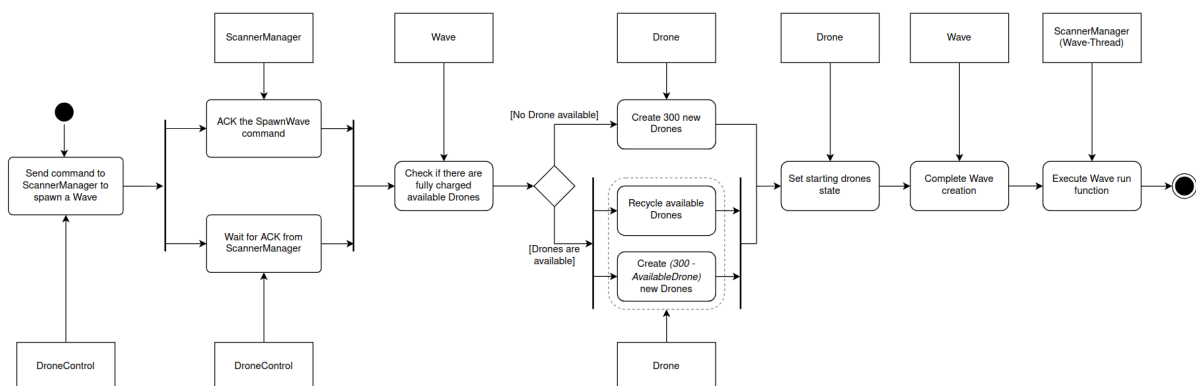
- **(1.1) Sistema di Copertura dell'Area di Sorveglianza:** Il sistema deve programmare e coordinare i percorsi di volo dei droni per garantire una copertura completa e costante dell'area di sorveglianza di 6×6 Km.
- **(1.2) Monitoraggio e Verifica del Territorio:** Il sistema deve assicurare che ogni punto dell'area sia verificato almeno una volta ogni 5 minuti, monitorando la posizione e l'attività di ciascun drone.
Un punto è verificato al tempo t se al tempo t c'è almeno un drone a distanza inferiore a 10 m dal punto

- **(2.1) Implementazione del Centro di Controllo:** Il centro di controllo e ricarica deve essere situato al centro dell'area da sorvegliare. Il sistema deve essere configurato per utilizzare questa posizione centrale come punto di partenza per la pianificazione delle missioni e per l'ottimizzazione dei percorsi di ritorno per la ricarica.
- **(2.2) Funzionalità del Centro di Controllo:** Il centro di controllo, situato al centro dell'area di sorveglianza, deve gestire tutte le operazioni dei droni, inclusa la pianificazione delle missioni.
- **(3.1) Controllo autonomia dei Droni:** Il sistema deve gestire autonomamente l'autonomia di volo di ciascun drone, coordinando i tempi di rientro per la ricarica.

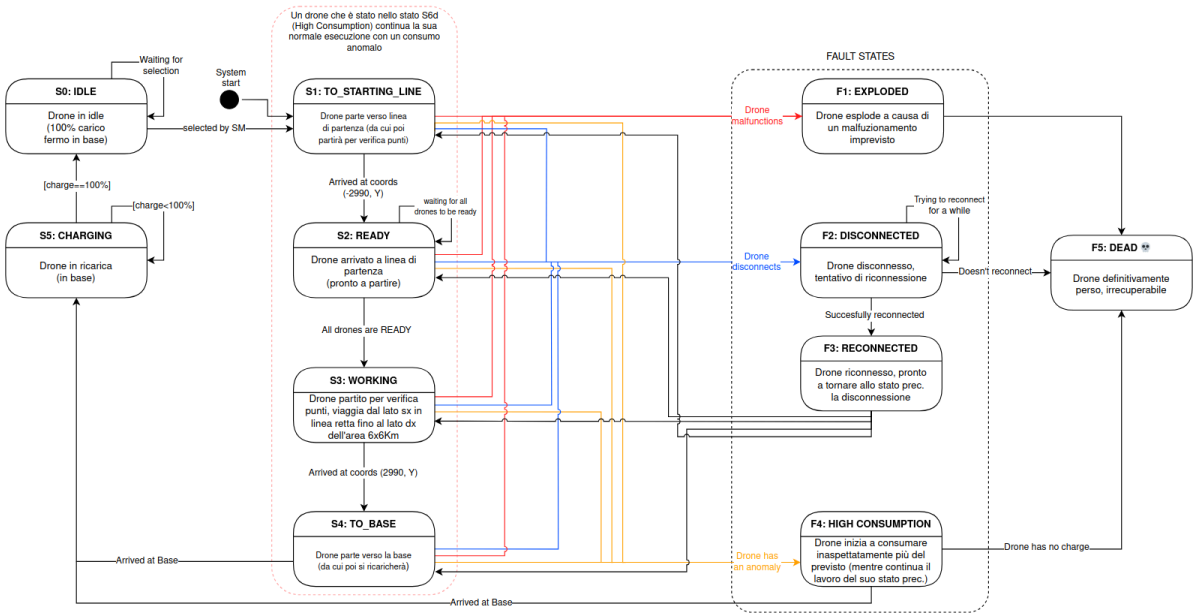
Architectural system diagram



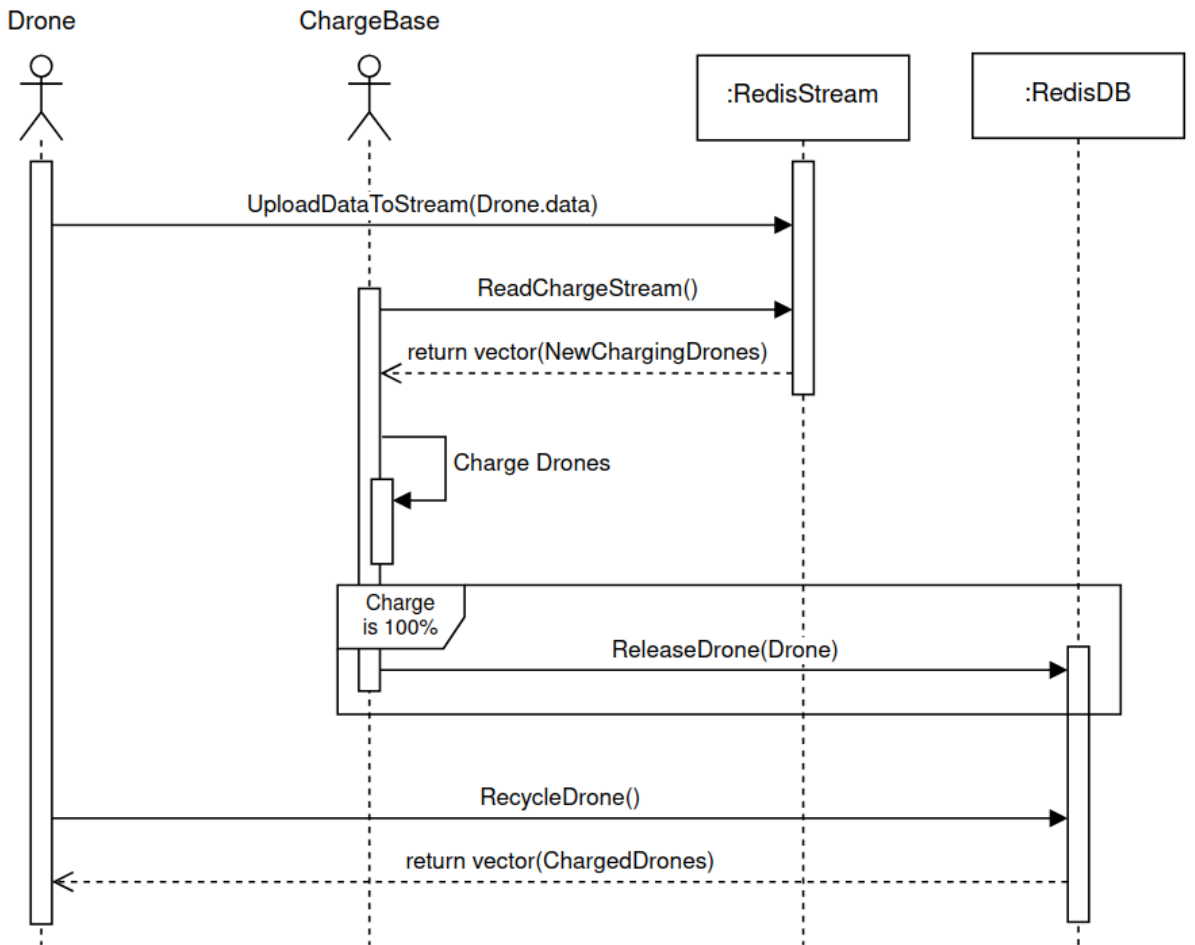
Activity diagram creazione Wave e droni



State diagram Drone



Message sequence chart diagram carica Drone



Monitors

WaveCoverageMonitor (funzionale)

WaveCoverageMonitor si occupa di controllare che, ad ogni tick, ogni drone di un'onda che sta nello stato `WORKING` stia effettivamente verificando il proprio punto. In caso contrario riporterà le informazioni di quale drone abbia fallito la verifica e il suo motivo.

AreaCoverageMonitor (funzionale)

AreaCoverageMonitor si occupa di controllare che, ad ogni tick, tutti i punti dell'area vengano correttamente verificati dai droni. In caso contrario riporterà le coordinate e i tick per la quale l'area non è stata verificata.

Drone Charge (non-funzionale)

Il DroneChargeMonitor verifica che non ci sia alcun consumummo anomalo per i droni del sistema. Nel caso in cui un drone inizi ad avere un consumo elevato, il monitor riporta il valore del consumo per singolo tick del drone e se è riuscito ad arrivare alla base o meno.

Drone Recharge (funzionale)

RechargeTimeMonitor controlla che il tempo di carica del drone sia fuori dal range di $[2, 3]h$. Se ciò accade, il monitor riporta in quanti tick (e anche in quanti minuti) il drone si è ricaricato.

System Performance (non-funzionale)

Per ogni onda a lavoro in un determinato tick, viene verificato se tutti i suoi droni siano effettivamente in uno stato `WORKING`. In caso contrario, contando il numero di droni effettivamente a lavoro se questi ultimi sono meno di quelli previsti (il numero di droni a lavoro è conosciuto per ogni onda, così come il numero di onde) le prestazioni degradano in percentuale.

Il livello di degrado per un dato tick è perciò dato dal numero di droni non a lavoro (`DEAD` o `DISCONNECTED`) in relazione al numero di droni totali che dovrebbero esserlo.

Implementation

Implementazione software

Il sistema è implementato in [C++](#), e fa uso di [PostgreSQL](#) e [Redis](#) (in particolare è stata utilizzata la libreria [redis-plus-plus](#)).

Redis è stato usato per permettere la comunicazione di dati tra le componenti del sistema.

Le componenti principali del sistema sono sincronizzate tra di loro utilizzando semafori, ogni componente ha un semaforo per ricevere il comando di partenza e completamento di un tick. Il componente `ScannerManager` ha anche un semaforo interno che permette la sincronizzazione ad ogni tick di tutti i suoi thread, rappresentati le onde del nostro sistema.

Implementare il sistema

Il sistema è strutturato secondo un'architettura modulare che comprende diverse componenti chiave, ciascuna realizzata attraverso file sorgente specifici.

Componente ScannerManager

`ScannerManager` coordina le operazioni della flotta di droni.

Pseudocodice di ScannerManager:

```
class ScannerManager
    [CheckSpawnWave] Checks whether the DroneControl has sent a command to spawn
    a new Wave
        Save current time
        While loop
            Get "spawn_wave" value from Redis
            If "spawn_wave" == 1 then
                Decrease the value of "spawn_wave" on Redis
                return True
            End if
            Save elapsed_time since start
            If elapsed_time > timeout value then
                If "spawn_wave" == -1 then
                    return False
                End if
                return False
            End if
        End loop

    [SpawnWave] Create a new Wave
        Set the wave ID
        Create and add a new Wave to the vector containing all the other Waves
        Enque the Wave's main run function to the pool of threads
        Increase the Wave ID counter

    [Run] ScannerManager's main execution function
        Create the IPC message queue
        Create or open 2 semaphores used for sync (sem_sync_sc: recieves GO
command, sem_sc: sends tick completed status)
        Signal the synchronizer to add a new thread to the synch process
        While simulation is running loop
            Wait for sem_sync semaphore for GO command
            If current tick is a multiple of 150 and CheckSpawnWave is true then
                SpawnWave
            End if
            If the number of msg in the IPC queue is > 0 then
                For loop on queue size
                    Get the message
                    Append the message to the right/indicated Wave

                End loop
            End if
            Signal the synchronizer that current tick is completed
            Release sem_sc semaphore to comunicate tick completed
            Increase internal tick counter
```

```
End loop
Signal synchronizer to remove a thread from the synch process
Close sem_sync and sem_dc semaphores
Close the IPC message queue
```

Componente DroneControl

Il modulo `DroneControl` si occupa di impartire le istruzioni operative ai droni, garantendo che ogni area sia monitorata in conformità con i requisiti di progetto.

Pseudocodice di DroneControl:

```
class DroneControl
    [ConsumerThreadRun] ConsumerThread main execution function
        While simulation running loop
            Read data from "scanner_stream"
            Parse data from stream
            Write data to DB-Buffer
        End loop

    [WriteDroneDataToDB] DB-Writer thread main execution function
        While max_tick read from DB-Buffer < duration in ticks of simulation
            Read from DB-Buffer the data to write into DB
            Create query string containing the batch of data to write (batch_size
= 15000)
                Execute query
            If there are remaining queries
                Execute remaining queries
            Sleep for 1s
        End loop

    [SendWaveSpawnCommand] Send command to ScannerManager to spawn new wave
        Send command to ScannerManager to spawn a new wave (do it by increasing
"spawn_wave" value on Redis)
        Wait for the ACK from ScannerManager ("spawn_wave" value = 0)

    [GetDronePaths] Calculate Drone's paths
        Calculate for each of the 300 drones in a wave the set of "working"
coords that it will have

    [Run] DroneControl's main execution function
        Create or open 2 semaphores used for sync (sem_sync_dc: recieves GO
command, sem_dc: sends tick completed status)
        Calculate paths for working drones
        Create Consumers
            Create Redis consumer group
            Create threads running ConsumerThread main function
            Detach Consumer-Threads
        Create DB-Writer thread
        While simulation is running loop
            Wait for sem_sync semaphore for GO command
            Every 150 ticks SendWaveSpawnCommand
            Release sem_dc semaphore to comunicate tick completed
            Increase internal tick counter
        End loop
```



```
Join DB-Thread if possible
Close sem_sync and sem_dc semaphores
```

Componente ChargeBase

Questa componente è responsabile per la gestione della ricarica dei droni, assicurando che i droni siano pronti per le operazioni di volo secondo le necessità del sistema.

Pseudocodice di ChargeBase:

```
class ChargeBase
  [Run] ChargeBase's main execution function
    Create or open 2 semaphores used for sync (sem_sync_cb: recieves GO
command, sem_dc: sends tick completed status)
    While simulation is running loop
      Wait for sem_sync semaphore for GO command
      Increase charge for every charging drone [ChargeDrone]
      Check if new Drones need charging [ReadChargeStream]
      Release sem_cb semaphore to communicate tick completed
      Increase internal tick counter
    End loop

  [ReadChargeStream] Check if new drones need to be charged
    Read every element of "charge_stream" on Redis
    Parse every element read
      Update the local drone data [SetChargeData]
    Trim the stream

  [SetChargeData] Update the local data of a drone
    Use the param of the function to update the drone's data saved in an
unordered map
    Calculate the rate of charge for the given drone

  [ChargeDrone] Increase the charge value of a drone by the rate calculated for
a tick
    For every charging drone loop
      If charge value < 100% then
        Increase the value by the calculated rate
      Else
        Release the drone [ReleaseDrone]

    Remove the drone from the charging drones container

  [ReleaseDrone] Remove the given charged drone from ChargeBase
    Add on Redis the drone to the set of charged drones
    Add on "scanner_stream" the info that the drones has completed charging
process

  [SetChargeRate] Creates the charging speed rate
    Choose randomly the total charging duration (between 2 and 3 hours)
    Calculate the charging rate for a single tick
```

Componente Drone

Il `Drone` è un componente fondamentale del sistema e di una Wave (onda). Ogni drone ha un'autonomia limitata e necessita di ricarica dopo un periodo di attività.

Pseudocodice di Drone:

```
class Drone
  [Drone] Drone's constructor
    Set current state to ToStartingLine

  [setState] Change current state to the given state
    If currentState != nullptr then
      Execute the exit function of the currentState
    End if
    Set currentState to the given new State

  [Run] Drone's main execution function
    Execute currentState's main run function
    Increase internal tick counter

class DroneState
  class ToStartingLine
    [Run]
      If charge value in the next step is <= 0 then
        setState(Dead)
      End if
      If Drone has reached the starting line then
        setState(Ready)
      Else
        If Drone's next step does reach starting line then
          Set X to -2990
          Set Y to Drone's starting_line.y
          Decrease Drone's charge
        Else
          Move Drone towards the starting line following the direction
previously calculated
          Decrease Drone's charge
        End if
      End if

  class Ready
    [Enter]
      Increase Wave's number of ready Drones (# of Drones that reached the
starting line)

    [Run]
      If charge value in the next step is <= 0 then
        setState(Dead)
      End if
      If # of ready Drones < 300 then
        Decrease Drone's charge (Wait for all drones to reach the
starting line)
      Else
        setState(Working)
```

```

        End if

class Working
    [Run]
        If charge value in the next step is <= 0 then
            setState(Dead)
        End if
        If Drone has reached the right border then
            setState(ToBase)
        Else
            Move Drone to the right by 20m
            Decrease Drone's charge
        End if

class ToBase
    [Run]
        If charge value in the next step is <= 0 then
            setState(Dead)
        End if
        If Drone has reached Base then
            setState(Charging)
        Else
            If Drone's next step dose reach the Base then
                Set Drone's coords to (0, 0)
                Decrease Drone's charge
            Else
                Move Drone towards the base following the direction
previously calculated (inverted on the Y-axis)
                Decrease Drone's charge
            End if
        End if

class Charging
    [Enter]
        Upload to Redis stream ("charge_stream") Drone's data needed by
ChargeBase
        Add DroneID to the vector drones_to_delete

class Disconnected
    [Enter]
        Create a "hidden" version of coords and charge value
        Save in disconnected_tick the current tick value

    [Run]
        If reconnect_tick != -1 then
            If charge value in the next step is <= 0 then
                setState(Dead)
            End if
            Switch on Drone's previous state
                case ToStartingLine
                    Hidden_to_starting_line
                case Ready
                    Hidden_ready
                case Working
                    Hidden_working
                case ToBase

```

```

        Hidden_to_base
    End switch
    If current internal tick >= reconnect_tick + disconnected_tick then
        setState(Reconnected)
    End if
End if

[Hidden_to_starting_line]
    If hidden_charge value in the next step is <= 0 then
        setState(Dead)
    End if
    If Drone has reached the starting line then
        Drone's previous state = Ready
        Increase # of ready drones
    Else
        If Drone's next step does reach starting line then
            Set X to -2990
            Set Y to Drone's starting_line.y
            Decrease Drone's charge
        Else
            Move Drone towards the starting line following the direction
previously calculated, using hidden_coords
            Decrease Drone's charge
        End if
    End if

[Hidden_ready]
    Decrease Drone's charge
    If # of ready Drones < 300 then
        Drone's previous state = Working
    End if

[Hidden_working]
    Decrease Drone's charge
    Move Drone to the right by 20m using hidden_coords
    If Drone reaches the right border (using hidden_coords) then
        Drone's previous state = ToBase
    End if

[Hidden_to_base]
    Decrease Drone's charge
    Move Drone towards the base following the direction previously
calculated, using hidden_coords
    If Drone reaches the base then
        setState(Charging)
    End if

class Reconnected
    [Enter]
        Drone coords = hidden_coords
        Drone charge = hidden_charge

    [Run]
        Switch on Drone's previous state
            case None

```

```

        setState(Dead)  #If state is None there was an issue with the
drone's execution
        case ToStartingLine
            setState(ToStartingLine)
        case Ready
            setState(Ready)
        case Working
            setState(Working)
        case ToBase
            setState(ToBase)
        case Charging
            setState(Charging)

class Dead
    [Enter]
        Add DroneID to vector of Drones that need to be deleted

```

Componente Wave

Wave si occupa di creare e in parte gestire le operazioni dell'onda di droni che sorveglia l'area.

Pseudocodice di Wave

```

class Wave
    [Wave] Wave's contructor
        RecycleDrones
        Set starting internal tick counter
        Set WaveID
        For loop on 300 Wave's drones do
            Create Drone with its data (ID, X, Y, starting_X, starting_Y,
tick_drone)
            Calculate drone's direction
        End for
        Self add to "waves_alive" on Redis

    [UploadData] Uploads onto the Redis stream Wave Drone's data
        For each Drone do
            Create stream-ready drone-data
            Add to the pipeline the Redis stream upload function
        End for

    [setDroneFault] Drone fault manager for a given fault
        Calculate the DroneID from WaveID
        Save Drone's current state (into "previous" variable)
        If fault state != NONE then
            Set Drone's state to the given fault state
        End if
        Set Drone's reconnect_tick
        Set Drone's high_consumptio_factor

    [RecycleDrones] Used when creating a new Wave, gets all available fully
charged Drones
        Get from Redis DB at most 300 of the fully charged and available drones

    [DeleteDrones] Delete Drones that are dead

```

```

    For each DroneID to be deleted do
        Calculate the array index of the Drone
        If Drone hasn't be deleted yet
            Remove Drone from array

[AllDronesAreDead] Check if all drones of the wave are Dead
    return True if all drones are Dead

[Run] Wave's main execution function
    Signal the synchronizer to add a new thread to the synch process
    Create a Redis pipeline
    While !AllDronesAreDead loop do
        If the queue for Drone faults is not empty then
            Parse the fault-data from the queue
            setDroneFault
        End if
        For loop on each Drone do
            If Drone != nullptr then
                Execute Drone's Run function
                Create stream-ready drone-data
                Add the Redis stream upload function to the pipeline
            End if
        End for
        Execute Pipeline
        DeleteDrones
        Signal the synchronizer that current tick is completed
        Increase internal tick counter
    End loop
    Remove self from alive Waves on Redis
    Signal synchronizer to remove a thread from the synch process

```

Componente TestGenerator

TestGenerator è il componente che si occupa di generare scenari pseudocasuali che influenzano il sistema. Ogni scenario si verifica con una determinata probabilità, e fra questi vi sono gli scenari in cui il drone consuma di più, si disconnette, ecc.

Pseudocodice di TestGenerator

```

class TestGenerator
    [Constructor] Define scenarios with their percentages
        [All_fine] 20% (Testing)

        [Charge_rate_malfunction] 20%
            ChooseRandomDrone
            Generate charge_rate_factor (between 0.5 and 0.8)
            Send IPC message to ScannerManager

        [High_consumption] 20%
            ChooseRandomDrone
            Generate high_consumption_factor (between 1.5 and 2)
            Send IPC message to ScannerManager

        [Death] 20%
            ChooseRandomDrone

```

```

        Send IPC message to ScannerManager

[Disconnected] 20%
    ChooseRandomDrone
    Calculate probability of reconnecting
    If reconnect < 70% then
        Generate reconnect_tick using ChooseRandomTick
        Send IPC message to ScannerManager
    Else (Drone will not reconnect)
        Send IPC message to ScannerManager
    End if

[Run] TestGenerator's main executing function
    While True do
        Generate scenario probability
        Select correct scenario using generated probability
        Sleep for 5s (Giving time to system to run)
    End loop

[ChooseRandomDrone] Calculates a Drone to select for scenarios
    Select a Wave from "waves_alive" on Redis
    Randomly select a Drone

[ChooseRandomTick] Generates # of tick after which the drone will reconnect
    Randomly select a tick between 1 and 20

```

Componente Database

DroneControlSystem utilizza un database `dcs` PostgreSQL per memorizzare e gestire i dati relativi all'attività e allo stato dei droni durante le missioni di sorveglianza. La gestione del database è implementata nel file `Database.cpp`

Pseudocodice del Database:

```

class Database
    [ConnectToDB] Creates a connection to "dcs" database
        Make unique connection to "dcs"

    [ReadCredentials] Reads the credentials from a config file
        Open the config file
        Read and parse the credentials
        Close the config file

    [CreateDB] Given the credentials, creates a new database
        Check if the database exists
        If it doesn't exist then
            Create a new database with the given credentials
        End if

    [CreateTables] Creates the tables for the database
        Drop existing tables
        Create new tables

    [get_DB] Creates and/or connects to the "dcs" database
        While retry counter < max tries then

```

```

        ReadCredentials
        CreateDB
        ConnectToDB
        CreateTables
    End while

[ExecuteQuery] Executes the indicated query
    If connection to DB is open then
        Execute query

class Buffer
    [WriteToBuffer] Writes the indicated vector into the buffer
        Get the lock for thread safety
        Bulk insert the data vector into the buffer

    [ReadFromBuffer] Reads the entire buffer
        Get the lock for thread safety
        Bulk read the entire buffer by saving the data into a vector
        Clear the buffer

    [getSize] Return the size of the buffer
        return Buffer's size

```

Altre componenti

Componente Main

Il `Main` è il meta-componente che gestisce i processi e le attività di tutto `DroneControlSystem`, eseguendole al momento opportuno.

Pseudocodice di Main

```

[main] Main executing function
    Create Redis' connections options

    Fork to create DroneControl process
        Create Redis connection
        Create DroneControl object
        Run DroneControl

    Fork to create ScannerManager process
        Create Redis' connection pool options
        Create Redis pool
        Create ScannerManager object
        Run ScannerManager

    Fork to create ChargeBase process
        Create Redis connection
        Create ChargeBase
        Create random_device used for creating charging rates
        SetEngine in ChargeBase
        Run ChargeBase

    Fork to create TestGenerator process

```



```

    Create Redis connection
    Create TestGenerator object
    Run TestGenerator

    Fork to create Monitors process
        Create and run Monitors (AreaCoverage, WaveCoverage, DroneCharge,
RechargeTime)
        When simulation ends run SystemPerformanceMonitor

    End forks

    Create Redis connection
    Create semaphores (sem_sync_dc, sem_sync_sc, sem_sync_cb, sem_dc, sem_sc,
sem_cb)
    While simultion is running do
        Post on semaphores (sem_sync_dc, sem_sync_sc, sem_sync_cb) to signal
start of new tick
        Wait on semaphores (sem_dc, sem_sc, sem_cb) that signal end of tick for
simulating processes
        Increase internal tick count
    End loop
    Wait for child processes to finish
    Kill TestGenerator process

```

Componenti Monitor

```

class Monitor
    [CheckSimulationEnd] Checks if the simulation is still running
    If the simulation has ended, then
        Set sim_running to false
    End if

class RechargeTimeMonitor
    [RunMonitor] Runs RechargeTime monitor
    Create a thread that runs the RechargeTime monitor
    ([checkDroneRechargeTime])

    [checkDroneRechargeTime] Main execution function for the monitor
    While sim_running is true
        Sleep for 10 seconds
        getChargingDrones
        getChargedDrones
        For each charging drone do
            If drone has completed charging then
                Calculate duration of recharge (in ticks)
                If duration < 3000 or duration > 4500 then
                    Add entry to monitor table
                End if
            End if
        End for
        Check if simulation has ended
    End while

    [getChargingDrones] Get list of drones that have started charging

```

Adds to the list of charging drones any new drones that have started charging since the last check

[getChargedDrones] Get list of drones that have completed charging

Adds to the list of charging drones any new drones that have completed charging since the last check

class WaveCoverageMonitor

[RunMonitor] Runs WaveCoverage monitor

Create a thread that runs the WaveCoverage monitor ([checkWaveCoverage])

[checkWaveCoverage] Main execution function for the monitor

While sim_running is true

Sleep for 10 seconds

checkCoverageVerification

Check if simulation has ended

End while

[checkCoverageVerification] Check if every wave is verifying the area

getWaveVerification

For each failed wave verification do

Add entry to monitor table

End for

[getWaveVerification] Get list of waves that have failed verification

Adds to the list of failed waves any new waves that have failed verification since the last check

class AreaCoverageMonitor

[RunMonitor] Runs AreaCoverage monitor

Create a thread that runs the AreaCoverage monitor ([checkAreaCoverage])

[checkAreaCoverage] Main execution function for the monitor

Initialize area_coverage_data

Sleep for 10 seconds

Find every new entries for WORKING drone

For each WORKING drone do

If the tick for WORKING drone > area_coverage_data[drone's coords]

last tick then

readAreaCoverage

End if

End for

Check if simulation has ended

InsertUnverifiedTicks

[readAreaCoverageData] Elaborate the given area coverage data

If data corresponds to special "starting case" then (first 210 ticks)

For each tick from 0 to given current tick do

Add tick to list of unverified ticks for that coordinate

End for

Else if (current tick - last verified tick) > 125 then

For each tick from (last verified tick + 126) to current tick do

Add tick to list of unverified ticks for that coordinate

End for

```

    End if
    Updated area_coverage_data[drone's coords] last tick with current tick

[InsertUnverifiedTicks] Insert unverified ticks into monitor table
    For each coord of 6x6 area do
        Add entry to monitor table for each unverified tick
    End for

class SystemPerformanceMonitor
    [RunMonitor] Runs SystemPerformance monitor
        Create a thread that runs the SystemPerformance monitor
    ([checkPerformance])

    [checkPerfomance] Main execution function for the monitor
        getPerformanceData
        If performance < 100% then
            Insert into database data
        End if

    [getPerformanceData] Get data from database
        Get numbers of working drones and waves for each tick from database

class DroneChargeMonitor
    [RunMonitor] Runs DroneCharge monitor
        Create a thread that runs the DroneCharge monitor ([checkDroneCharge])

    [checkDroneCharge] Main execution function for the monitor
        While sim_running is true
            Sleep for 10 seconds
            getDroneData
            ElaborateData
            Check if simulation has ended
        End while

    [getDroneData] Get drone data from Database
        Get first tick of drone
        Get last tick of drone (DEAD or CHARGING)

    [ElaborateData] Parses and checks if drone has higher consumption
        For every complete drone data do
            If consumption is higher than base value then
                Insert into database data
            End if
        End for

```

Schema del Database

Di seguito gli schemi delle tabelle del database `dcx` usato

Tab drone_logs

Column	Data Type	Constraint	Info
tick_n	INT	PRIMARY KEY (tick_n, drone_id)	Il tick attuale della simulazione
drone_id	INT	NOT NULL	ID univoco del drone
status	VARCHAR(255)	-	Stato attuale del drone
charge	FLOAT	-	% di carica attuale del drone
wave	INT	-	L'onda a cui ∈ il drone
x	FLOAT	-	Coord x posizione attuale drone
y	FLOAT	-	Coord y posizione attuale drone
checked	BOOLEAN	-	Indica se drone ha verificato punto
created_at	TIMESTAMP	-	Indica l'istante di tempo in cui il dato è stato scritto sul DB

Tab wave_coverage_logs

Column	Data Type	Constraint	Info
tick_n	INT	PRIMARY KEY (tick_n, drone_id)	Il tick attuale della simulazione
wave_id	INT	-	L>ID dell'onda
drone_id	INT	-	ID univoco del drone
fault_type	VARCHAR(255)	-	Descriz. dell'eventuale fault state

Tab area_coverage_logs

Column	Data Type	Constraint	Info
checkpoint	VARCHAR(20)	PRIMARY KEY	Rappresenta un checkpoint da verificare dell'area
unverified_ticks	INT[]	-	I tick non verificati per un determinato checkpoint

Tab system_performance_logs

Column	Data Type	Constraint	Info
tick_n	INT	PRIMARY KEY	Il tick attuale della simulazione
working_drones_count	INT	-	Droni attualm. a lavoro
waves_count	INT	-	Num. di onde che stanno lavorando
performance	FLOAT	-	Il liv. in % di performance per quel tick

Tab drone_charge_logs

Column	Data Type	Constraint	Info
drone_id	INT	PRIMARY KEY	ID univoco del drone
consumption	FLOAT	-	Valore medio di consumo \forall tick del drone
consumption_factor	FLOAT	-	Il fattore HIGH_CONSUMPTION
arrived_at_base	BOOLEAN	-	Indica se il drone è giunto in base

Tab drone_recharge_logs

Column	Data Type	Constraint	Info
drone_id	INT	PRIMARY KEY	FK di drone_logs(drone_id)
recharge_duration_ticks	INT	-	Durata in tick della ricarica
recharge_duration_min	FLOAT	-	Durata in minuti della ricarica
start_tick	INT	-	Tick di inizio ricarica
end_tick	INT	-	Tick di fine ricarica

Connessioni Redis

Le connessioni e le operazioni Redis sono cruciali per la comunicazione tra i vari processi all'interno del progetto. Le operazioni con Redis sono integrate in diverse parti del codice sorgente.

Eccole qui elencate:

Connessione	Dettagli	Usata da
<code>spawn_wave</code>	VALUE flag per indicare se è necessaria una nuova Wave o meno	ScannerManager, DroneControl
<code>waves_alive</code>	SET per tracciare quali Waves sono attualmente attive nella simulazione	Wave, TestGenerator
<code>charged_drones</code>	SET per tracciare quali droni sono attualmente completamente carichi/disponibili	Wave, ChargeBase
<code>charging_drones</code>	SET per tracciare quali droni sono attualmente in carica	ChargeBase, TestGenerator
<code>scanner_stream</code>	STREAM usato per caricare ogni aggiornamento di stato dei droni	DroneControl, Wave, Drone, ChargeBase
<code>charge_stream</code>	STREAM usato per caricare i dati necessari a ChargeBase	ChargeBase, Drone
<code>connection_pool</code>	Usato da Waves per un utilizzo efficiente del multi-threading Redis	ScannerManager, Wave, Drone
<code>scanner_group</code>	Gruppo di consumer usato per la lettura in blocco di <code>scanner_stream</code>	DroneControl
<code>pipeline</code>	Usato per caricare in blocco i dati su <code>scanner_stream</code>	Wave

Risultati Sperimentali

Per verificare il corretto funzionamento del sistema, abbiamo tenuto conto di diversi parametri, e osservato i dati di diverse simulazioni. Prendendo in esame una delle simulazioni più rappresentative, ossia una di quelle in cui si verificano tutti gli scenari del TestGenerator, abbiamo constatato, comparando l'output del sistema/monitor coi dati nel DB, in primis la correttezza delle informazioni visualizzate ambo i lati, e poi quanto e come il sistema performasse.

Correttezza del sistema

Avvio del sistema

Quando avviamo il sistema i componenti principali vengono creati e sincronizzati tra loro per poter avviare la simulazione. Successivamente, dopo la connessione al Database e alla creazione dell'onda, ci troviamo di fronte alla vista dei tick che scorrono ad indicare l'avvenuto avvio della simulazione.

```

=====
[DroneControl]    DroneControl created
[ChargeBase]      ChargeBase created
[ScannerManager]  ScannerManager created
[TestGenerator]   TestGenerator created
=====
[Database]        Successfully connected to DB on attempt 1
[DroneControl]    Sending wave spawn command
[DroneControl]    DB writer thread started
[DroneControl]    Wave spawned
=====
[ChargeBase]      TICK: 0
[Main]            TICK: 0
[DroneControl]    TICK: 0
[ScannerManager]  TICK: 0
=====
[Main]            TICK: 1
[DroneControl]    TICK: 1
[ScannerManager]  TICK: 1
[ChargeBase]      TICK: 1
=====
[Main]            TICK: 2
[DroneControl]    TICK: 2
[ScannerManager]  TICK: 2
[ChargeBase]      TICK: 2
=====
....
...
..
.

```

Vita completa dei droni

Di seguito verifichiamo che il sistema faccia funzionare correttamente i droni, infatti per un determinato drone possiamo osservare la presenza di tutti i suoi stati a partire da `TO_STARTING_LINE`, che indica una suo corretto "avviamento", fino a `CHARGING`, che indica come il drone sia riuscito a ritornare alla base ed a iniziare il processo di ricarica (e l'eventuale `CHARGING_COMPLETED`).

	123 tick_n	123 drone_id	Az status	123 charge	123 wave_id	123 x	123 y	checked
1	0	0	TO_STARTING_LINE	99,865562	0	-14,142136	-14,142136	[]
2	1	1	TO_STARTING_LINE	99,731125	0	-28,284271	-28,284271	[]
3	2	2	TO_STARTING_LINE	99,596687	0	-42,426407	-42,426407	[]
211	210	0	TO_STARTING_LINE	71,633675	0	-2.983,985107	-2.983,985107	[]
212	211	0	READY	71,499237	0	-2.990	-2.990	[]
213	212	0	WORKING	71,364799	0	-2.990	-2.990	[v]
512	511	0	WORKING	31,167048	0	2.990	-2.990	[v]
513	512	0	TO_BASE	31,032608	0	2.990	-2.990	[]
724	723	0	TO_BASE	2,665804	0	6,014923	-6,014923	[]
725	724	0	CHARGING	2,531364	0	0	0	[]
726	5.466	0	CHARGING_COMPLETED	100	0	0	0	[]

TestGenerator comparato a DB

Osserviamo adesso come nel DB siano stati inseriti i dati di interesse. Poniamo l'attenzione su ogni sequenza di avvenimenti in cui il `TestGenerator` ha generato uno scenario (che non fosse *Everything is fine*), l'output della shell lo ha stampato a video, e il sistema ha scritto nelle opportune tabelle del DB i giusti valori che rispecchiassero l'accaduto.

Scenario *Everything is fine*

Scenario di default in cui tutto funziona senza anomalie. Viene impostato come funzione vuota per il 20% del valore della mappa.
Per vedere che si sia verificato basta guardare l'output loggato di `drone_logs` in cui i tick si succedono normalmente.

```
[DroneControl] Wave spawned
=====
[Main] TICK: 0
[DroneControl] TICK: 0
[ScannerManager] TICK: 0
[ChargeBase] TICK: 0
=====
[Main] TICK: 1
[DroneControl] TICK: 1
[ScannerManager] TICK: 1
[ChargeBase] TICK: 1
=====
[Main] TICK: 2
[DroneControl] TICK: 2
[ScannerManager] TICK: 2
[ChargeBase] TICK: 2
=====
[Main] TICK: 3
[DroneControl] TICK: 3
[ScannerManager] TICK: 3
[ChargeBase] TICK: 3
=====
```

tick_n	drone_id	A-Z status	charge	wave_id	x	y	checked
1	0	0 TO_STARTING_LINE	99,865562	0	-14,142136	-14,142136	[]
2	0	1 TO_STARTING_LINE	99,865562	0	-14,189511	-14,094599	[]
3	0	2 TO_STARTING_LINE	99,865562	0	-14,237046	-14,046583	[]
4	0	3 TO_STARTING_LINE	99,865562	0	-14,284735	-13,998084	[]
5	0	4 TO_STARTING_LINE	99,865562	0	-14,332576	-13,949097	[]
6	0	5 TO_STARTING_LINE	99,865562	0	-14,380569	-13,899612	[]
7	0	6 TO_STARTING_LINE	99,865562	0	-14,42871	-13,849631	[]
8	0	7 TO_STARTING_LINE	99,865562	0	-14,477	-13,799148	[]
9	0	8 TO_STARTING_LINE	99,865562	0	-14,525434	-13,748154	[]
10	0	9 TO_STARTING_LINE	99,865562	0	-14,574013	-13,696648	[]
11	0	10 TO_STARTING_LINE	99,865562	0	-14,622731	-13,644623	[]
12	0	11 TO_STARTING_LINE	99,865562	0	-14,671589	-13,592073	[]
13	0	12 TO_STARTING_LINE	99,865562	0	-14,720583	-13,538998	[]
14	0	13 TO_STARTING_LINE	99,865562	0	-14,769711	-13,485388	[]
15	0	14 TO_STARTING_LINE	99,865562	0	-14,818969	-13,431239	[]
16	0	15 TO_STARTING_LINE	99,865562	0	-14,868356	-13,376547	[]
17	0	16 TO_STARTING_LINE	99,865562	0	-14,917869	-13,321306	[]
18	0	17 TO_STARTING_LINE	99,865562	0	-14,967504	-13,265513	[]
19	0	18 TO_STARTING_LINE	99,865562	0	-15,01726	-13,209161	[]
20	0	19 TO_STARTING_LINE	99,865562	0	-15,067131	-13,152246	[]
21	0	20 TO_STARTING_LINE	99,865562	0	-15,117119	-13,094761	[]

Scenario *Drone failure*

Viene selezionato un drone casuale che "esplode" (o cessa di funzionare). Il suo stato viene impostato su `DEAD` da `ScannerManager`.

```
=====
[TestGenerator] Drone 3264 exploded
[Main] TICK: 749
[DroneControl] TICK: 749
[ScannerManager] TICK: 749
[ChargeBase] TICK: 749
[ScannerManager] TICK 749 Drone 3264 state set to DEAD
=====
```

tick_n	drone_id	A-Z status	charge	wave_id	x	y	checked
1	749	3264 DEAD	49.719944	3	230	2290	[]
2	748	3264 WORKING	49.719944	3	230	2290	[v]
3	747	3264 WORKING	49.854385	3	210	2290	[v]
4	746	3264 WORKING	49.988827	3	190	2290	[v]
5	745	3264 WORKING	50.123268	3	170	2290	[v]
6	744	3264 WORKING	50.25771	3	150	2290	[v]
7	743	3264 WORKING	50.392151	3	130	2290	[v]
8	742	3264 WORKING	50.526592	3	110	2290	[v]
9	741	3264 WORKING	50.661034	3	90	2290	[v]
10	740	3264 WORKING	50.795475	3	70	2290	[v]
11	739	3264 WORKING	50.929916	3	50	2290	[v]

Scenario *High consumption*

Viene scelto un drone casuale e viene aumentato il suo consumo di energia (tra 1.5 e 2 volte rispetto al normale). `ScannerManager` imposta perciò il fattore di consumo elevato del drone.

```
=====
[ScannerManager]   TICK: 1640
[Main]             TICK: 1640
[DroneControl]     TICK: 1640
[ChargeBase]       TICK: 1640
[TestGenerator]    Drone 5212 has high consumption factor of 1.629380
=====
```

	123 drone_id	123 consumption	123 consumption_factor	<input checked="" type="checkbox"/> arrived_at_base
1	5.212	0,153599	1,142511	[]
2	16.006	0,137887	1,025637	[v]

Scenario *Charge rate malfunction*

Viene scelto un drone casuale dalla lista `charging_drones` in Redis e viene aumentato il suo rate di carica. Un messaggio viene inviato alla `ChargeBase` per impostare il fattore di carica.

```
=====
[ScannerManager]   TICK: 3649
[Main]             TICK: 3649
[DroneControl]     TICK: 3649
[ChargeBase]       TICK: 3649
[TestGenerator]    Drone 4107 has charge rate factor of 0.020510
[Wave]             Wave 16 duration: 19413ms
[Wave]             Wave 16 finished
=====
```

	123 drone_id	123 recharge_duration_ticks	123 recharge_duration_min	123 start_tick	123 end_tick
95	4.122	4.627	185,080017	1.165	5.792
96	4.107	4.669	186,76001	1.168	5.837
97	4.100	4.511	180,440002	1.170	5.681

Scenario *Connection lost*

Lo scenario **Connection_lost** comporta due sotto-scenari:

1. **Drone non si riconnetterà**: se la probabilità di riconnessione calcolata è inferiore al 70%, il drone rimane disconnesso. Viene inviato un messaggio a `ScannerManager` che imposta il suo stato su `DISCONNECTED` per 20 tick per poi diventare `DEAD`.
2. **Drone si riconnetterà**: se la probabilità di riconnessione è del 70% o superiore, il drone verrà riconnesso dopo un certo numero di tick, calcolato casualmente (usando `reconnect_tick`). Anche in questo caso, un messaggio a `ScannerManager` aggiorna lo stato del drone su `DISCONNECTED` con il tempo di riconnessione impostato.

```

=====
[ScannerManager]   TICK: 2239
[Main]             TICK: 2239
[TestGenerator]    Drone 16006 disconnected and will reconnect after 18 tick
[DroneControl]     TICK: 2239
[ChargeBase]       TICK: 2239
=====

```

	123 tick_n ↑	123 drone_id	A-Z status	123 charge	123 wave_id	123 x	123 y	checked
1	2.000	16.006	DISCONNECTED	100	16	0	0	[]
2	2.001	16.006	DISCONNECTED	100	16	0	0	[]
3	2.002	16.006	DISCONNECTED	100	16	0	0	[]
4	2.003	16.006	DISCONNECTED	100	16	0	0	[]
5	2.004	16.006	DISCONNECTED	100	16	0	0	[]
6	2.005	16.006	DISCONNECTED	100	16	0	0	[]
7	2.006	16.006	DISCONNECTED	100	16	0	0	[]
8	2.007	16.006	DISCONNECTED	100	16	0	0	[]
9	2.008	16.006	DISCONNECTED	100	16	0	0	[]
10	2.009	16.006	DISCONNECTED	100	16	0	0	[]
11	2.010	16.006	DISCONNECTED	100	16	0	0	[]
12	2.011	16.006	DISCONNECTED	100	16	0	0	[]
13	2.012	16.006	DISCONNECTED	100	16	0	0	[]
14	2.013	16.006	DISCONNECTED	100	16	0	0	[]
15	2.014	16.006	DISCONNECTED	100	16	0	0	[]
16	2.015	16.006	DISCONNECTED	100	16	0	0	[]
17	2.016	16.006	DISCONNECTED	100	16	0	0	[]
18	2.017	16.006	DISCONNECTED	100	16	0	0	[]
19	2.018	16.006	RECONNECTED	97,445686	16	-274,145477	-263,142944	[]
20	2.019	16.006	TO_STARTING_LINE	97,445686	16	-274,145477	-263,142944	[]

Monitor

Andando più sul concreto, iniziamo a snocciolare qualche dato, partendo dai monitor e dalla loro osservazione.

Copertura dell'area

Per quanto riguarda la verifica dei punti dell'area, i due monitor [WaveCoverageMonitor](#) e [AreaCoverageMonitor](#) ci permettono di verificare la corretta copertura dell'area.

Inoltre, fino a quando la prima onda non avrà terminato di lavorare, [AreaCoverageMonitor](#) riporterà la mancata copertura su quei punti non ancora verificati da questa prima onda.

Per questa esecuzione del sistema che abbiamo deciso di analizzare, possiamo osservare come, a causa dell'input ricevuto da TestGenerator, il drone 1102 abbia smesso di funzionare correttamente; quindi il monitor [WaveCoverageMonitor](#) giustamente riporta che dal tick 387 al tick 636 l'onda 1 abbia avuto un drone che ha mancato la verifica dei propri punti.

260	384	1.102	WORKING	65,046234	1	-2.050	-950	[v]
261	385	1.102	WORKING	64,911797	1	-2.030	-950	[v]
262	386	1.102	DEAD	64,911797	1	-2.030	-950	[]

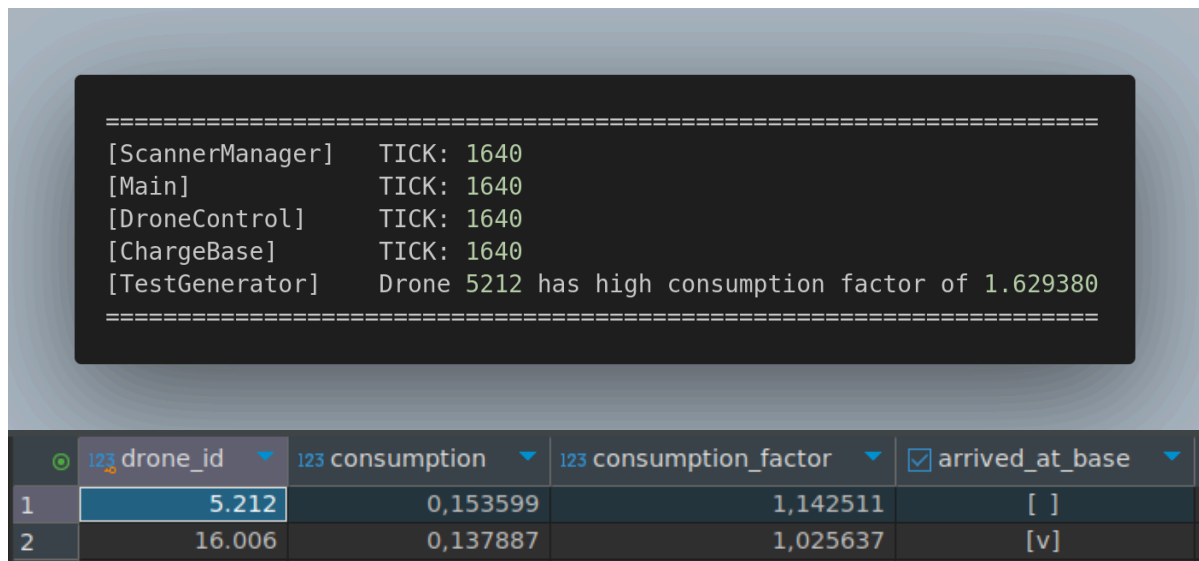
	123 tick_n ↑	123 wave_id	123 drone_id	A-Z issue_type
1	387	1	1.102	DIED_WHILE_WORKING
2	388	1	1.102	DIED_WHILE_WORKING
249	635	1	1.102	DIED_WHILE_WORKING
250	636	1	1.102	DIED_WHILE_WORKING

Consumo anomalo

Come già esposto precedentemente, il TestGenerator può scegliere non solo quale drone avrà un consumo elevato, ma anche di quanto sarà più alto.

Nel nostro caso il drone 512 è stato scelto e come riportato dal monitor [ChargeDroneMonitor](#) il drone ha avuto un consumo medio a tick di 0.153599 con un fattore di 1.142511 volte, rispetto ad un consumo normale. Ovviamente il fattore scelto dal TestGenerator risulta più alto rispetto a quello riportato dal monitor, questo perchè il drone ha avuto un numero significativo di tick per cui il suo consumo non è stato anomalo, andando quindi ad abbassare il valore medio del suo fattore di consumo.

Ci potrebbero essere casi in cui il drone nonostante il suo consumo anomale riesce a tornare alla base.



```
=====
[ScannerManager]    TICK: 1640
[Main]              TICK: 1640
[DroneControl]      TICK: 1640
[ChargeBase]        TICK: 1640
[TestGenerator]     Drone 5212 has high consumption factor of 1.629380
=====
```

	123 drone_id	123 consumption	123 consumption_factor	<input checked="" type="checkbox"/> arrived_at_base
1	5.212	0,153599	1,142511	[]
2	16.006	0,137887	1,025637	[v]

Ricarica anomala

Nel caso in cui un drone abbia un malfunzionamento durante la fase di ricarica, il monitor [RechargeTimeMonitor](#) riporterà un'anomalia nel tempo di ricarica. Ad esempio, il drone 4107 ha avuto un tempo di ricarica fuori dal range prestabilito di $[2, 3] h$ (la sua ricarica è durata 4669 tick), il monitor registra il numero di tick e i minuti effettivi impiegati per la ricarica. Questo permette di identificare eventuali problemi nel processo di ricarica.

```

=====
[ScannerManager]   TICK: 3649
[Main]             TICK: 3649
[DroneControl]     TICK: 3649
[ChargeBase]       TICK: 3649
[TestGenerator]    Drone 4107 has charge rate factor of 0.020510
[Wave]             Wave 16 duration: 19413ms
[Wave]             Wave 16 finished
=====

```

	123 drone_id ↓	123 recharge_duration_ticks	123 recharge_duration_min	123 start_tick	123 end_tick
95	4.122	4.627	185,080017	1.165	5.792
96	4.107	4.669	186,76001	1.168	5.837
97	4.100	4.511	180,440002	1.170	5.681

Performance del sistema

Come riportato dai monitor della copertura, alcuni droni (e quindi onde) non hanno correttamente verificato alcuni punti dell'area. Per questo [SystemPerformanceMonitor](#) ha riportato che per alcuni tick, come il 460, un valore percentuale delle performance sotto il 100%, indicando che per quel tick non tutti i droni "a lavoro" hanno effettuato il loro compito di copertura.

	123 tick_n ↑	123 wave_id	123 drone_id	A-Z issue_type
74	460	1	1.102	DIED_WHILE_WORKING
75	461	1	1.102	DIED_WHILE_WORKING
76	462	1	1.102	DIED_WHILE_WORKING

	123 tick_n ↑	123 working_drones_count	123 waves_count	123 performance
249	460	599	2	99,833333
250	461	599	2	99,833333
251	462	899	3	99,888889
252	463	899	3	99,888889