

# Relazione su Controllo formazione droni

A. Bettoni (1998044),  
A. Coppola (2003964),  
S. Di Cesare (1938649)

July 3, 2024

# Contents

<b>1</b>	<b>Descrizione Generale</b>	<b>3</b>
<b>2</b>	<b>Analisi del software</b>	<b>4</b>
2.1	Requisiti utente . . . . .	4
2.2	Requisiti di sistema . . . . .	4
2.3	Algoritmo . . . . .	5
2.4	Diagrams . . . . .	5
2.4.1	State diagram . . . . .	8
2.5	Scelte progettuali . . . . .	8
<b>3</b>	<b>Protocollo di comunicazione</b>	<b>9</b>
3.1	Scambio di messaggi . . . . .	9
3.2	La classe Channel . . . . .	10
3.2.1	Thread Safety . . . . .	10
3.3	La classe Message . . . . .	11
3.3.1	Costruttori e Metodi . . . . .	11
3.3.2	Lista Messaggi . . . . .	12
3.4	Message Sequence Chart UML . . . . .	13
<b>4</b>	<b>Database</b>	<b>14</b>
4.1	Struttura del Database . . . . .	14
4.2	Implementazione . . . . .	14
4.2.1	Lista DB File Properties . . . . .	14
4.2.2	Procedura di Connessione . . . . .	15
4.2.3	La tabella Blocco . . . . .	15
<b>5</b>	<b>Implementazione</b>	<b>16</b>
5.1	Monitor . . . . .	16
5.2	Altre parti interessanti . . . . .	17

# 1 Descrizione Generale

**Problema:** Si vuole progettare un sistema di controllo di  $n$  droni che sorvegli una data area rettangolare. Il sistema deve garantire che, con i droni forniti, ogni punto dell'area venga sorvegliato il più frequentemente possibile.

I droni partiranno dalla torre di controllo che si trova al centro dell'area da sorvegliare. Ogni drone ha un'autonomia limitata (misurata in minuti in volo) e una velocità massima. La torre dovrà gestire gli spostamenti di ogni drone facendo in modo che tornino alla torre prima che la batteria sia del tutto esaurita. Quando i droni si trovano nella torre di controllo sono considerati in carica e il tempo di ricarica può variare da drone a drone.

**Dati sperimentali:** Si assume che:

1. Ogni drone si muove alla velocità costante di  $30 \text{ km/h}$
2. Il tempo di ricarica varia tra le 2 e le 3 ore
3. Un punto si dice visitato se si trova nel raggio di 10 m da un drone

**Idea di soluzione:** Abbiamo scelto di modellare il problema con un sistema che agisce sulla torre di controllo e ricarica che a sua volta controlla i droni. Dunque la torre invia ad ogni drone le coordinate del punto sull'area da raggiungere secondo un sistema di volo a "tappe". Una volta che il drone arriva alla posizione ricevuta lo comunica alla torre che risponde con la posizione successiva, delineando così un percorso. Quando il drone è scarico lo comunica alla torre e si dirige alla torre di controllo, per essere ricaricato.

Sta al sistema quindi il compito di calcolare, mentre i droni sono in volo, il percorso migliore per far sì che ogni punto dell'area venga sorvegliato il più frequentemente possibile, tenendo conto dei punti visitati, dei droni in volo, e dei punti che visitano prima di scaricarsi.

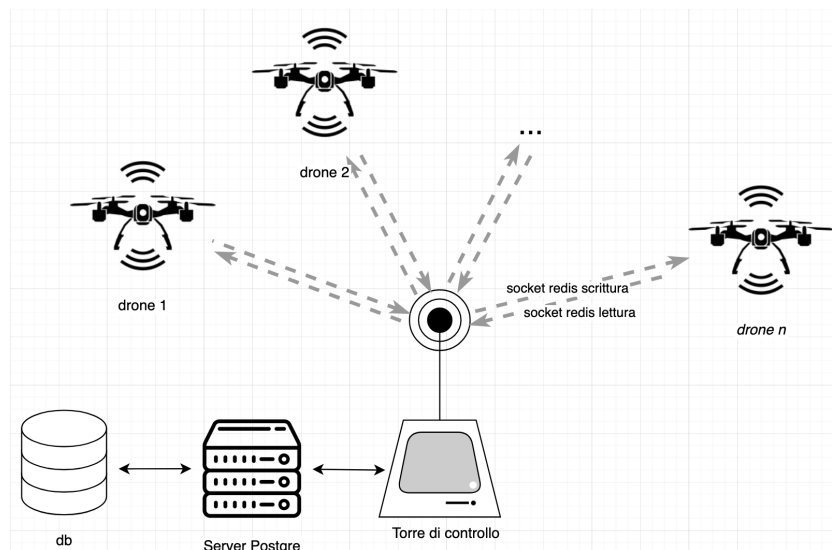


Figure 1: Architettura fisica del sistema

## 2 Analisi del software

### 2.1 Requisiti utente

Per funzionare, il sistema ha bisogno dei seguenti requisiti:

- dimensione dell'area  
Ogni punto dell'area è identificato dalla sua posizione in una matrice di punti e di quei punti ci interessa solo:
  - il tempo trascorso dall'ultima visita.
- numero di droni  
per ogni drone è di interesse:
  - Stato : tra i stati precedentemente elencati
  - posizione
  - carica residua (in minuti)
  - carica massima (in minuti)
  - tempo di ricarica
  - last\_update : tempo trascorso dall'ultimo update

### 2.2 Requisiti di sistema

Il sistema deve rispettare i seguenti requisiti:

1. Area:
  - 1.1 I blocchi sono tutti tra loro disgiunti (non esiste un punto dell'area che si trova in più di un blocco)
  - 1.2 L'unione dei blocchi copre tutti e solo i punti dell'area (un punto appartiene all'area se e solo se esiste un blocco che lo contiene)
  - 1.3 Un blocco può essere assegnato al più ad un drone
2. Drone:
  - 1.1 Un drone a cui è stato assegnato un blocco può trovarsi al di fuori di esso se e solo se:
    - i. il drone è partito dalla torre e si sta posizionando verso il punto di partenza del blocco
    - ii. il drone è scarico e sta tornando alla torre
    - iii. il drone ha visitato tutto il blocco e si sta posizionando verso il punto di partenza di un'altro blocco
  - 1.2 Un drone può volare verso un punto diverso da quello della torre se e solo se il suo tempo di volo residuo è maggiore del tempo che serve al drone per tornare alla torre (*più un valutato margine di errore*)
3. Torre:
  - 1.1 La torre non può terminare il suo processo se esiste un drone connesso che non è tornato nella torre di controllo.
  - 1.2 Tutti i punti che la torre invia ad un drone devono trovarsi all'interno dell'area assegnata al drone
4. Requisiti non-funzionali:
  - 1.1 ogni punto dell'area deve essere visitato il più frequentemente possibile
  - 1.2 Il tempo di risposta della torre per ogni drone deve essere ragionevolmente basso per fare in modo che i droni rimangano fermi sul posto il meno possibile

## 2.3 Algoritmo

Contando di poter ottenere per ogni drone, una tupla contenente almeno  $\{Id, Posizione, Stato, Blocco\}$  di tutti i droni che si sono connessi alla torre e le dimensioni dell'area da sorvegliare, la torre inizia a seguire il seguente algoritmo. Il problema può essere gestito seguendo i seguenti passaggi:

1. La torre di controllo conta i droni che si sono connessi e divide l'area da sorvegliare in  $N$  blocchi tutti della stessa misura secondo una specifica funzione approssima per eccesso i blocchi necessari per far volare tutti i droni.
2. La torre assegna ad ogni drone *"pronto a partire"* un blocco disponibile.
3. Ogni drone si dirige al punto di partenza del blocco assegnatogli. Non appena lo raggiunge lo comunica alla torre.
4. Quindi la torre azzerava il tempo di visita dell'area  $10\sqrt{2} \times 10\sqrt{2}$  (*la scelta della dimensione verrà approfondita in seguito*) centrata nella posizione del drone e gli invia il prossimo punto da raggiungere per scansionare tutto il blocco.
5. Quando il drone ha visitato tutto il blocco gli viene assegnato il blocco disponibile che contiene il punto non visitato da più tempo. Il loop continua dal punto 3.
6. Quando un drone ha carica appena sufficiente per tornare alla torre di controllo parte la procedura di return. Segnala alla torre che sta tornando comunica quindi l'ultimo punto visitato e si dirige alla torre.
7. Quando il drone arriva alla torre di controllo inizia a ricaricarsi. La torre lo marcherà come *"pronto a partire"* non appena la batteria sarà carica.
8. Se la torre non riceve messaggi da un drone in volo per troppo tempo questo verrà segnato come *"morto"*.

Questo è l'idea principale per far sì che tutti i punti vengano visitati il più frequentemente possibile. Questo ciclo viene ripetuto finché l'utente non interrompe il programma o tutti i droni "muoiono".

## 2.4 Diagrams

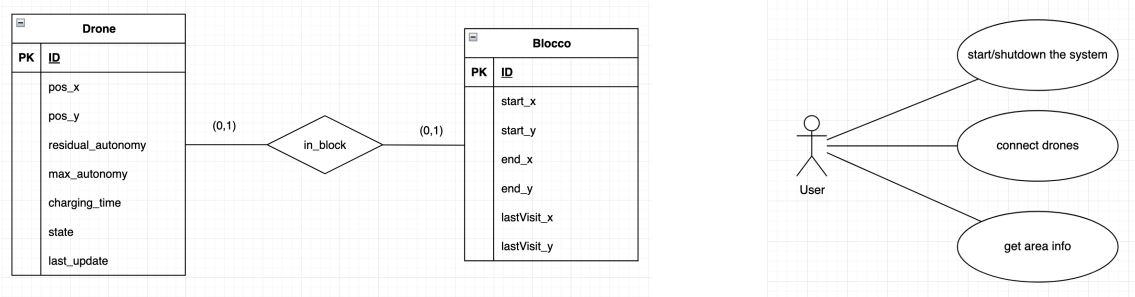


Figure 2: schema ER e UML

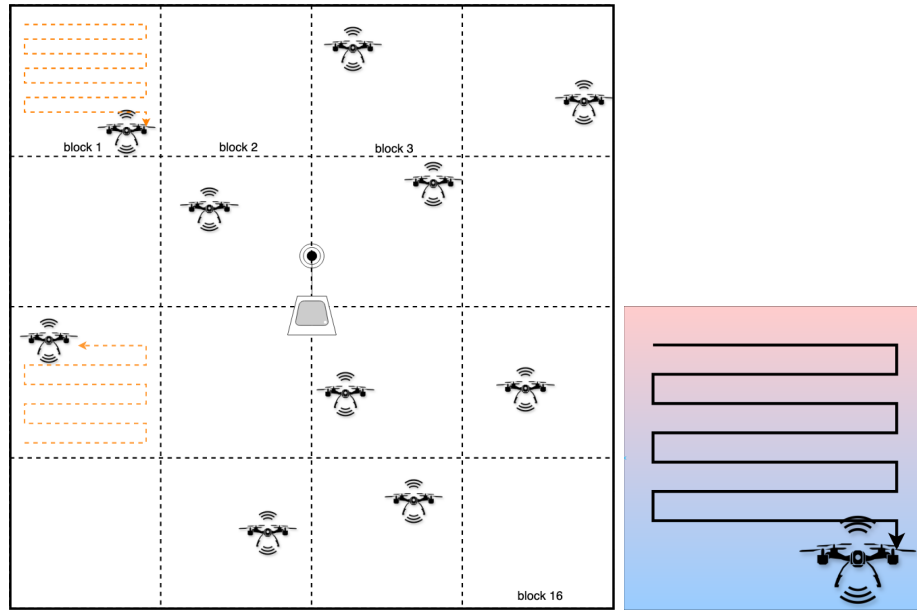


Figure 3: Esempio di area suddivisa in blocchi. Nel dettaglio a destra oltre al percorso seguito dal drone, viene evidenziata come più "calda" l'area visitata da meno tempo e più "fredda" l'area appena vista.

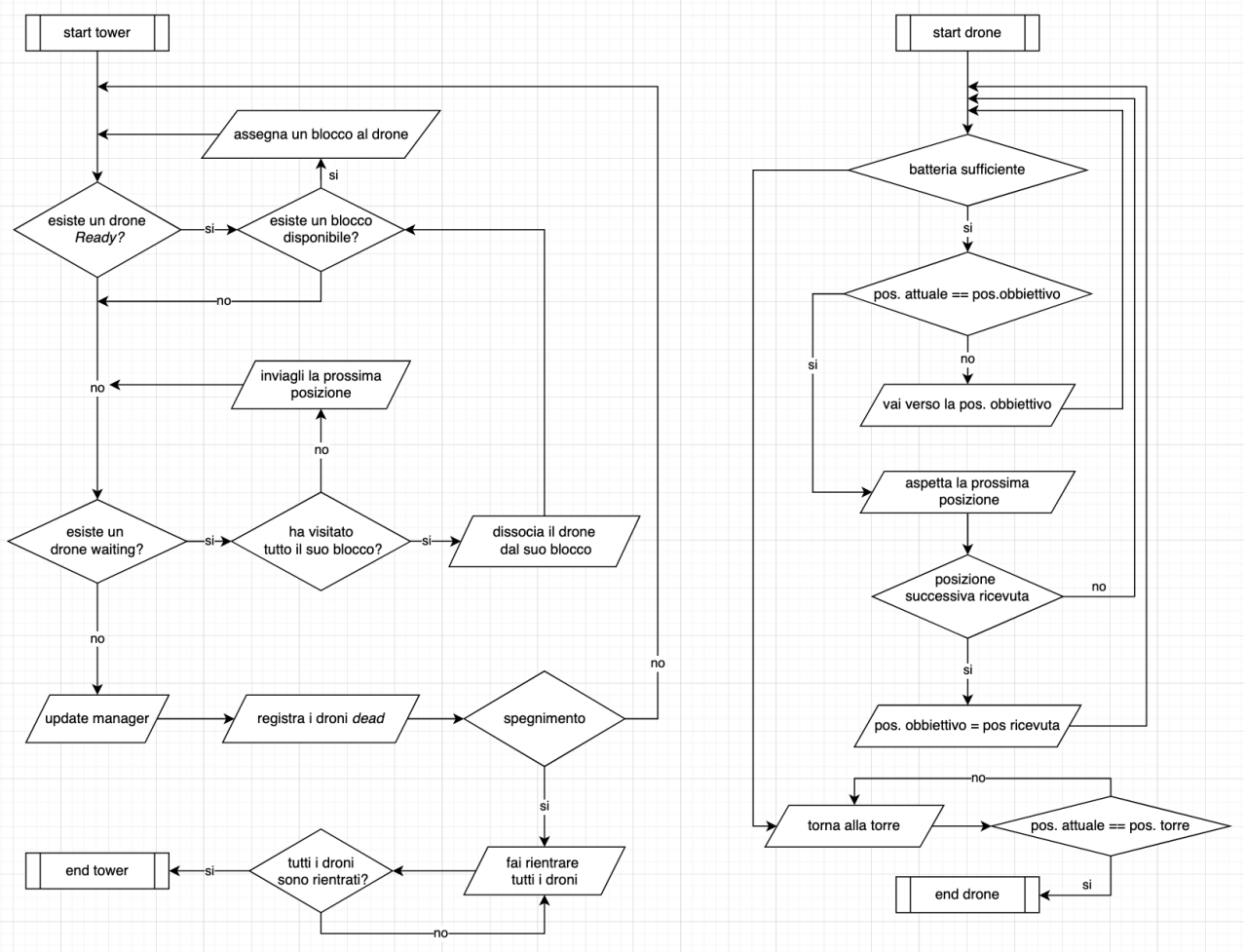


Figure 4: FlowChart della torre e del drone. Nota bene, nel FlowChart viene assunto che il drone e la torre siano già connessi. Il Flow Chart del drone ne rappresenta il ciclo di vita dal momento che parte dalla torre al momento che vi rientra.

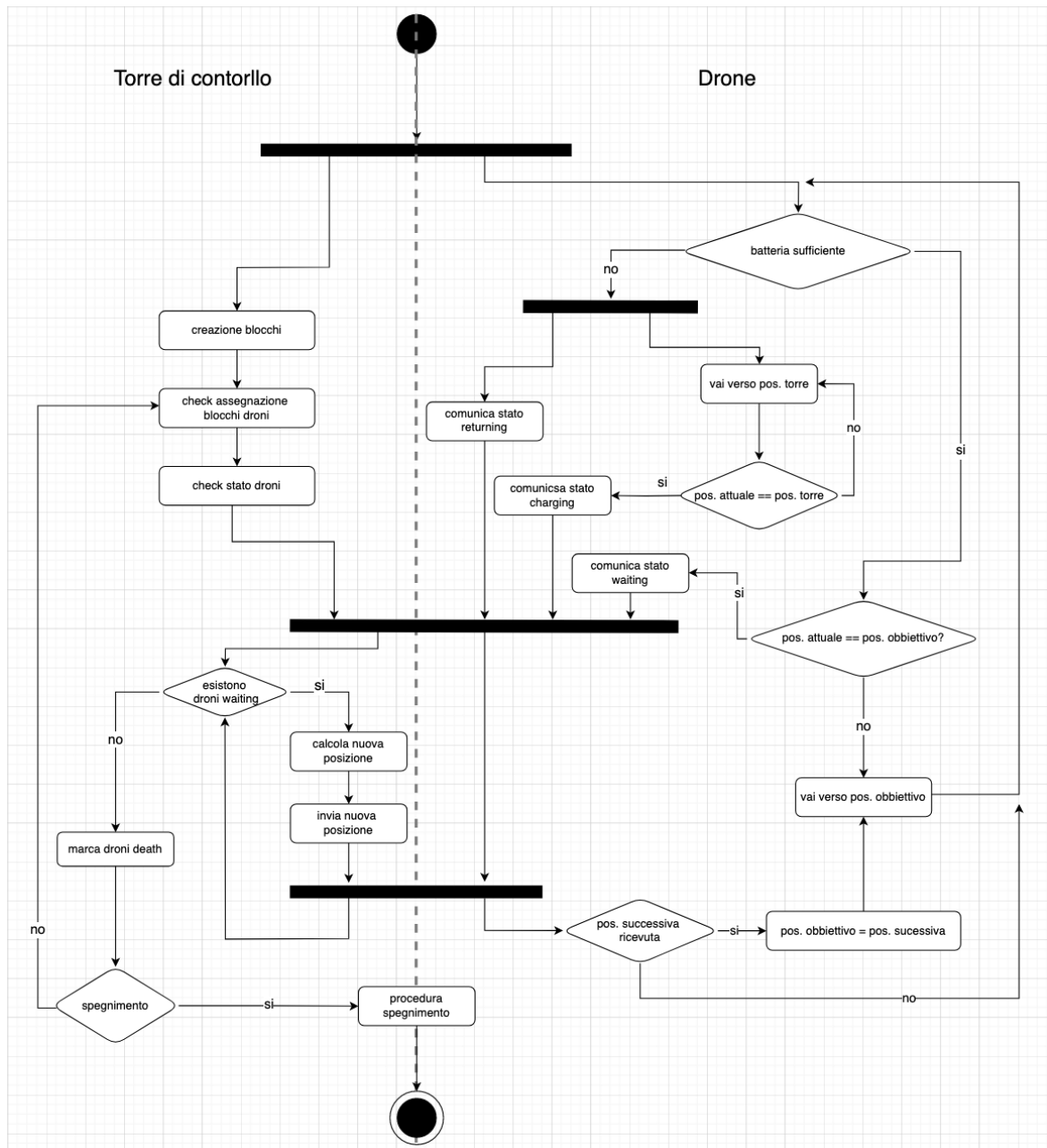


Figure 5: Activity diagram del sistema di interazione e decisione tra torre e drone, nel diagramma sono mostrate due barriere in cui drone e torre si sincronizzano per scambiarsi i messaggi principali mentre è lasciato implicito l'handler message di entrambi per messaggi di check.

### 2.4.1 State diagram

Per decidere l'azione da assegnare ad ogni drone la torre deve riuscire a distinguere la condizione di ogni drone. Definiamo quindi 6 stati in cui ogni drone può trovarsi:

1. **CHARGING:** il drone si trova alla base, e sta ricaricando la sua batteria
2. **READY:** il drone è carico e si trova nella torre di controllo (è pronto a partire)
3. **WAITING:** il drone sta aspettando che la torre gli invii la prossima coordinata
4. **MONITORING:** il drone sta scansionando l'area che gli è stata assegnata seguendo i punti inviati dalla torre
5. **RETURNING:** la carica del drone è sufficiente solo per il suo rientro (il drone sta rientrando)
6. **DEAD:** la batteria del drone si è scaricata prima che questo potesse rientrare oppure la torre di controllo non riesce più a contattarlo (fault).

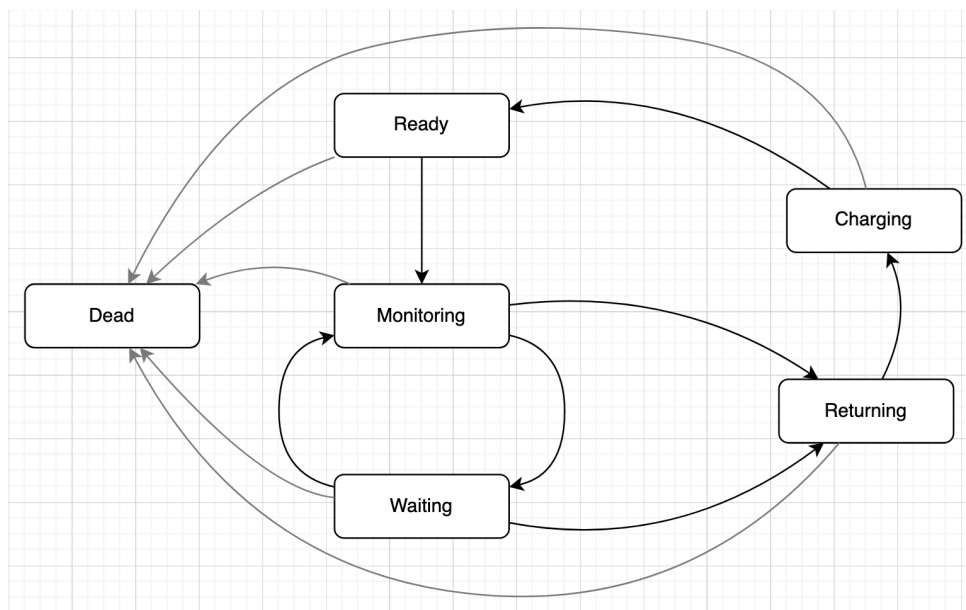


Figure 6: State diagram

## 2.5 Scelte progettuali

**Divisione delle celle** Dato che un punto si considera visitato se si trova nel raggio di 10 metri da un drone, possiamo discretizzare l'area rappresentandola come una matrice di celle quadrate di dimensione  $10\sqrt{2} \times 10\sqrt{2}$

**Divisione dei blocchi** Abbiamo scelto di raggruppare le caselle in blocchi circa della stessa area (la discretizzazione non garantisce la perfetta divisibilità) per garantire equità nelle assegnazioni. Data la variabilità delle dimensioni dell'area e del numero di droni in fase iniziale, non è sempre garantito che il numero di droni divida il numero caselle in un valore utile per costruire una matrice di blocchi. Cerchiamo quindi due valori  $h$  e  $w$  tali che :  $h * w > \text{numero di droni}$  e dividiamo l'area in  $h \times w$  blocchi. Questo permette a tutti i droni *READY* di volare contemporaneamente all'inizio del processo di monitoraggio e quindi ridurre il più possibile il tempo di visita medio dell'intera area. Anche se creare più blocchi ne riduce le dimensioni, questo non influisce sulle prestazioni del sistema in quanto ogni drone quando termina il ciclo di visita del suo blocco inizia a visitarne un altro e continua finché la batteria lo permette.



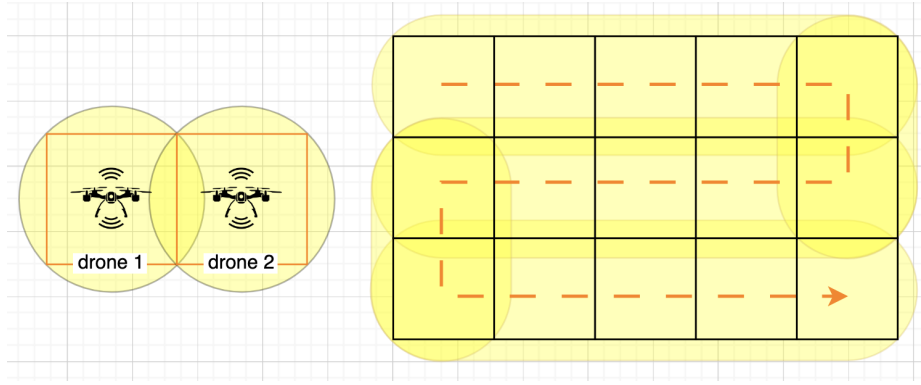


Figure 7: L'area divisa in caselle della dimensione del quadrato inscritto nel cerchio di raggio 10

**Tempi di ricarica** Sebbene l'algoritmo sia stato ideato per massimizzare il tempo di visita in un ciclo di carica della flotta di droni, questo non tiene conto del fatto che ogni drone può lunghi tempi di ricarica che possono superare anche di varie volte il tempo di volo di ogni drone. Per sopperire a questo rallentamento è molto efficace considerare la divisione in blocchi dell'area come se si avesse a disposizione solo un'  $n$ -esima parte dei droni.

Indichiamo con  $n$  il numero di droni necessari per coprire il tempo di ricarica medio di un drone più uno (il tempo, medio, di volo del drone stesso). In questo modo si distribuisce il numero di droni in volo durante il monitoraggio a beneficio sia del tempo di visita medio che massimo.

Le differenze tra i vari approcci sono mostrate nella sezione risultati sperimentali.

**Assegnamento dei blocchi** Per minimizzare il tempo di visita massimo e medio abbiamo sperimentato due criteri di scelta:

1. **FirstAverageTime**: Assegnare ad un drone *free* il blocco con tempo medio trascorso dall'ultima visita più alto (la media viene eseguita su tutte le caselle nel blocco).
2. **FirstMaxTime**: Assegnare ad un drone *free* il blocco che contiene la casella visitata meno di recente.

Attualmente il programma utilizza il secondo approccio (**FirstMaxTime**). Computazionalmente i due approcci sono equivalenti e dai test anche a livello di efficacia non si nota una differenza significativa tranne che su aree grandi con pochi droni, dove è preferibile dare più importanza al valore massimo per essere sicuri di visitare almeno una volta tutti i punti.

### 3 Protocollo di comunicazione

La Torre di Controllo ed i Droni comunicano in maniera asincrona. Le comunicazioni avvengono tramite Redis.

#### 3.1 Scambio di messaggi

Ogni attore ha un suo identificativo univoco. La torre ha sempre  $ID = 0$  mentre i droni hanno id casuali assegnati dalla torre. Ogni attore ha riservato un canale rappresentato su redis da una **coda** con chiave = " $c:id$ "

Il proprietario usa il suo canale solo in lettura mentre gli altri attori lo utilizzano in scrittura. Dunque ogni elemento nella coda rappresenta l'id di un messaggio con il seguente formato ( $m : ID : mID$ ) dove

- $ID$  rappresenta l'id del mittente
- $mID$  rappresenta l'identificatore univoco dell'elemento nel canale.

Il contenuto di un messaggio  $m$  è salvato in una *HashMap* usando come chiave l' $mID$  del messaggio.

### 3.2 La classe Channel

Per gestire agevolmente la ricezione e l'invio dei messaggi è stata implementata la classe *Channel*, la quale gestisce un canale di un attore. Per inviare e ricevere messaggi, la classe *channel* presenta due metodi:

- I messaggi vengono aggiunti alla coda utilizzando `LPUSH c:id message_id` e salvati nell'*HashMap* con `"HSET message_id {params}"`
- In fase di lettura invece, viene utilizzato il metodo `RPOP c:id` per estrarre il `message_id`, e poi `HGETALL message_id` per recuperarne il contenuto.

Che costituiscono le due funzioni principali per inviare e ricevere messaggi:

**void sendMessageTo(int channelId, Message &message)**

Questo metodo permette di creare il messaggio su redis (quindi di creare una *HashMap* con  $id = m : cID : mID$  e con le sue coppie  $(kX, vX)$ ) e di inserire l'ID del messaggio all'interno della coda.

```
1 // Creazione del messaggio su redis
2 hset m:{this.id}:{message.id} {message.parseMessage()}
3
4 // Aggiunta del messaggio al canale
5 lpush c:{channelId} m:{this.id}:{message.id}
```

**Message\* awaitMessage(long timeout = 0)**

Questo metodo serve per aspettare l'arrivo di un messaggio all'interno di un canale, per poi leggerlo e ritornarlo. Il parametro **timeout**, se diverso da -1, imposta un tempo massimo per l'attesa di un messaggio.

```
1 // Letture dell'id da redis -> restituisce un id sulla variabile message_id
2 BRPOP c:{this.id} timeout
3
4 // Lettura contenuto del messaggio
5 HGETALL message_id
```

#### 3.2.1 Thread Safety

La Classe *channel* è thread safe, poiché gestisce le due funzioni **sendMessageTo** e **awaitMessage** tramite due mutex:

- Un mutex in scrittura, che viene utilizzato per fare il lock durante la creazione e scrittura di un messaggio su un canale
- Un mutex in lettura, che viene lockato in attesa di una risposta da redis quando si tenta di leggere un messaggio dal canale.

Si è deciso di dividere le operazioni tra due mutex per questione di responsività, così che mentre un attore è in attesa di un messaggio, possa continuare a fare altre operazioni in scrittura per aggiornare il suo stato agli occhi degli altri attori (nel nostro caso, torna utile per ridurre i tempi di attesa di risposta tra la torre ed i droni)

### 3.3 La classe Message

La classe Message è la classe che rappresenta un messaggio di un canale.

#### 3.3.1 Costruttori e Metodi

I costruttori della classe Message sono due:

- **Message(std::string id)**  
Usato nella creazione di un messaggio già presente. Prende come parametro una stringa del formato **id:mid**, nel quale **id** è l'id del canale che ha inviato il messaggio, e **mid** è l'id del messaggio.
- **Message(int id)**  
Usato nella creazione di un messaggio da creare ed inviare. Prende come parametro un intero che rappresenta l'id del messaggio.

In aggiunta, questa classe presenta due metodi virtuali che deve implementare ogni tipo di messaggio:

- **void parseResponse(RedisResponse\* response)**  
Questo metodo serve a ricevere una Risposta da un Comando Redis, e tramutarla in messaggio. La funzione prende una risposta redis che deve essere di tipo VECTOR, e ne legge i parametri a coppie.  
Per esempio, se si leggesse un messaggio *LocationMessage*, la sua funzione di parsing legge all'interno della risposta la chiave *x*, e setta la propria *x* a quella nella risposta.
- **std :: string parseMessage()**  
Questo metodo serve a trasformare i dati contenuti nel messaggio in una stringa della forma:

type type\_value p1 v1 p2 v2 p3 v3

Nella quale type\_value è un intero che rappresenta il tipo del messaggio, mentre pX kX sono le coppie (parametro, valore) contenute nel messaggio

**Esempio:** LocationMessage, che serve scambiare una posizione tra drone e torre e viceversa, presenterà il corpo della funzione parseMessage() di questo tipo:

```
1  std::string LocationMessage::parseMessage() {
2      std::string data = "type " + std::to_string(this->getType());
3      data = data + " x " + std::to_string(this->x);
4      data = data + " y " + std::to_string(this->y);
5      data = data + " movement_type " + std::to_string(this->movementType);
6      return data;
7  }
```

### 3.3.2 Lista Messaggi

Classe	Tipo	Parametri	Info
AssociateMessage	0	drone_id: long long, x: int, y: int	Messaggio di associazione drone↔torre
PingMessage	1	nessuno	Semplice messaggio di Ping
DroneInfoMessage	2	{params}	Scambio parametri drone→torre
LocationMessage	3	x: int, y: int	Nuova posizione per drone dalla torre
RetireMessage	4	nessuno	Drone con batteria scarica. Rientro necessario
DisconnectMessage	5	nessuno	Disconnette l'associazione drone↔torre

#### AssociateMessage

Messaggio inviato dal drone alla torre per richiedere l'associazione al pool di droni. La torre risponde ritornando l'id che associerà al drone per riconoscimento all'interno della formazione. Il messaggio presenta il campo **drone\_id**, che rappresenta l'id temporaneo de drone in richiesta, oppure l'id associato dalla torre in caso di associazione effettuata. Inoltre, il messaggio contiene la posizione della torre per gestire con precisione il punto di partenza dei droni.

#### PingMessage

Messaggio inviato dalla torre al drone per verificarne l'esistenza. Viene generalmente utilizzato quando la torre non riceve update dal drone per più di 2 minuti. Questo messaggio non presenta parametri.

#### DroneInfoMessage

Messaggio per aggiornare le info di un drone all'interno del db della torre. Viene generalmente mandato dalla torre al drone, per richiedere un update dello stato del drone e verificarne il funzionamento. I parametri del messaggio sono i campi del drone:

(**drone\_id**, **x**, **y**, **drone\_state**, **charge\_time**, **battery\_autonomy**)

#### LocationMessage

Messaggio con duplice scopo:

- Se inviato dalla torre al drone, significa che la torre sta impostando delle nuove coordinate da raggiungere al drone. Dunque, in lettura il drone aggiornerà le coordinate e deciderà cosa fare di conseguenza.
- Se inviato dal drone alla torre, rappresenta un update della posizione del drone, il quale ha completato il movimento assegnatogli in precedenza.

I parametri del drone indicano sempre una posizione sul piano cartesiano (l'area da monitorare)

#### RetireMessage

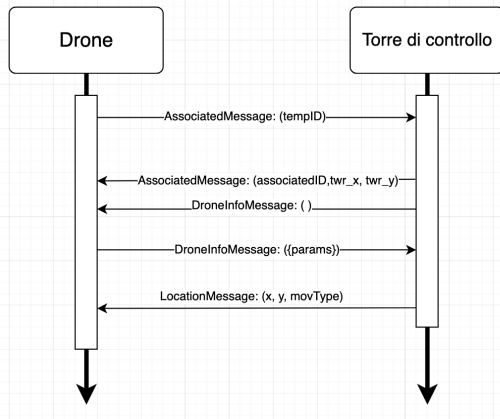
Messaggio inviato dal drone per notificare la necessità di rientro per low battery. In questo caso, la torre invia al drone un LocationMessage con le coordinate precise della torre, e lo mette in stato retiring per ottimizzare il percorso di rientro.

#### DisconnectMessage

Messaggio che rimuove l'associazione tra la torre ed il drone. Permette di liberare risorse su redis e sul database.

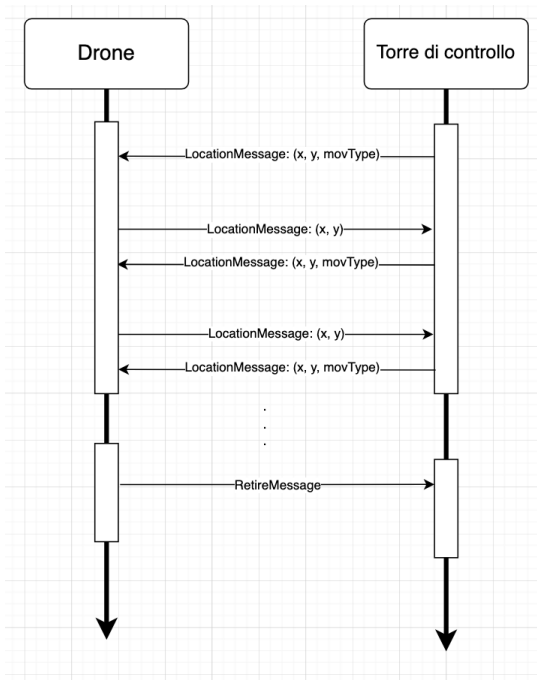
### 3.4 Message Sequence Chart UML

Di seguito vengono mostrati alcuni **Message Sequence Chart UML** che mostrano i principali scenari di scambio di messaggi tra la torre e un drone qualsiasi.



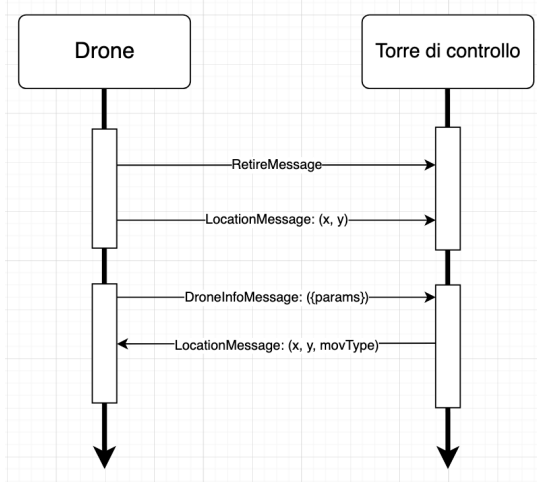
#### Associazione di un drone:

1. Il drone richiede di essere associato
2. La torre aderisce all'associazione
3. La torre richiede le informazioni del drone
4. Il drone risponde alla richiesta di informazioni
5. La torre inizia a guidare il drone



#### Monitoraggio di un drone:

1. La torre invia la posizione da raggiungere
2. Tramite il *LocationMessage* il drone comunica di aver raggiunto il punto identificato
3. La torre prontamente risponde con il nuovo punto calcolato
4. Quando il drone non ha carica sufficiente invia un *RetireMessage* e torna alla torre



#### Ricarica di un drone:

1. Il drone comunica che la sua batteria è scarica
2. Tramite il *LocationMessage* il drone comunica di essere tornato alla torre
3. Il drone comunica di essersi ricaricato inviando un *DroneInfoMessage*
4. La torre ricomincia a guidarlo per monitorare l'area

## 4 Database

All'interno del progetto viene sfruttato un database **PostgreSQL** per salvare dati utili in maniera persistente all'interno della torre.

### 4.1 Struttura del Database

Il Database presenta un'unica tabella: **drone**, la quale contiene le informazioni dei vari droni con il seguente schema:

Nome	Tipo	Info
id	int 64bit	Id > 0 univoco del drone
x	integer	Coordinata x del drone all'ultimo update
y	integer	Coordinata y del drone all'ultimo update
battery_autonomy	int 64bit	Durata della batteria all'ultimo update
charge_time	int 64bit	Durata della ricarica della batteria del drone
dstate	int	Stato del drone all'ultimo update
last_update	int 64bit	Timestamp dell'ultimo update

Il campo **dstate** contiene valori interi associati all'enum del codice C++:

```
1  enum DroneState {
2      CHARGING,
3      READY,
4      WAITING,
5      MONITORING,
6      RETURNING,
7      DEAD
8  }
```

### 4.2 Implementazione

Vengono sfruttate la libreria **pqxx** per creare una connessione ed eseguire query con il database PostgreSQL. L'eseguibile della torre prende come parametro facoltativo un file di properties, con la seguente struttura:

[Opzione]=Valore

Un esempio di file base è il seguente:

[Ip]=http://mydomain.com/db  
[User]=root

#### 4.2.1 Lista DB File Properties

Opzione	Tipo	Info
Ip	String	Ip alla connessione PostgreSQL
Port	Integer	Porta d'accesso del db (solo in caso di URL)
User	String	Username per accedere al db
Password	String	Password in chiaro per accedere al db
DbName	String	Nome del db da utilizzare

#### NOTA:

Le properties di base sono quelle di default di PostgreSQL (ip e port), con un utente chiamato **tower** senza password.

### 4.2.2 Procedura di Connessione

Una volta lette le properties, la torre crea una connessione con il db (in caso di fail, l'eseguibile si interrompe con un errore). Una volta connessa, la torre controllerà l'esistenza della tabella **drone**, e si metterà in attesa di ricevere dei droni per iniziare il controllo dell'area. Ogni qualvolta che un drone si connette alla torre, viene salvata la nuova entry del drone nel db.

#### **ID DRONE:**

Il protocollo di comunicazione prevede che il drone si crei un proprio id da mandare alla torre. Di conseguenza, la torre dovrà controllare se l'id autogenerato dal drone sia valido all'interno del db (non preesistente), e in caso dovrà assegnarne uno nuovo.

### 4.2.3 La tabella Blocco

La tabella block ha senso esclusivamente nel contesto dell'algoritmo di guida dei droni pertanto seppure è presente nel diagramma ER visto in precedenza non è di interesse nel database finale.

## 5 Implementazione

### 5.1 Monitor

seguono i monitor che assicurano che i vincoli funzionali e non-funzionali vengano rispettati

**locationTrigger** Questo trigger opera sul database e assicura che ogni drone rimanga nei limiti dell'area da monitorare

Listing 1: locationTrigger

```
1 CREATE OR REPLACE FUNCTION check_drone_area()
2 RETURNS TRIGGER AS $$
3 DECLARE
4     min_x INT := 0;
5     min_y INT := 0;
6     max_x INT := [this->areaWidth]
7     max_y INT := [this->areaHeight]
8 BEGIN
9     IF NEW.x < min_x OR NEW.x > max_x OR NEW.y < min_y OR NEW.y > max_y
10        THEN
11        RAISE EXCEPTION 'Drone % is out of surveillance area',
12        NEW.id;
13    END IF;
14    RETURN NEW;
15 END;
16 $$ LANGUAGE plpgsql;
17 CREATE TRIGGER drone_location_check
18 BEFORE INSERT OR UPDATE ON drone
19 FOR EACH ROW
20 EXECUTE FUNCTION check_drone_area();
```

**batteryTrigger** Questo trigger opera sul database e si assicura che non esistano droni che volano con la batteria scarica

Listing 2: batteryTrigger

```
1 CREATE OR REPLACE FUNCTION check_drone_battery()
2 RETURNS TRIGGER AS $$
3 BEGIN
4     IF NEW.dstate != CHARGING AND NEW.dstate != DEAD AND NEW.
5        battery_autonomy <= 0 THEN
6        'Drone % has no battery left but is still flying'
7        RAISE EXCEPTION,
8        NEW.id;
9    END IF;
10    RETURN NEW;
11 END;
12 $$ LANGUAGE plpgsql;
13 CREATE TRIGGER drone_battery_check
14 AFTER UPDATE ON drone
15 FOR EACH ROW
16 EXECUTE FUNCTION check_drone_battery();
```



## 5.2 Altre parti interessanti

Di seguito vengono riportate parti del codice implementato

```
1  void Drone::start(float executionSpeed) {
2
3      this->setExecutionSpeed(executionSpeed);
4
5      // Establish connection to tower
6      bool connectedToTower = this->connectToTower();
7      if (!connectedToTower) {
8          loge("Can't connect to tower!");
9          return;
10     }
11     logi("Connected to tower");
12
13     // Start loop
14     std::thread moveThread(&Drone::behaviourLoop, this);
15
16     std::vector<std::thread> threads;
17     this->running = true;
18     this->setState(READY);
19     while (this->running) {
20         Message *message = this->channel->awaitMessage();
21         if (message == nullptr) {
22             // No message received. Maybe ping tower?
23             continue;
24         }
25         logi("Received message m:" + message->getFormattedId());
26         threads.emplace_back(&Drone::handleMessage, this, message);
27         // Free up completed threads
28         for (auto it = threads.begin(); it != threads.end(); ) {
29             if (it->joinable()) {
30                 it++;
31             } else {
32                 it = threads.erase(it);
33             }
34         }
35     }
36     // Disconnect drone
37     // Wait threads to finish
38     for (auto &thread : threads) {
39         thread.join();
40     }
41
42 }
```

Listing 3: da drone.cpp

Il drone dopo aver completato con successo la procedura di connessione, entra in un loop in cui gestisce parallelamente sia i suoi spostamenti [riga 14. `behaviourLoop` è una funzione che simula lo spostamento del drone] sia i messaggi ricevuti e da inviare alla torre di controllo.

Le funzioni utilizzate per suddividere l'area in blocchi

```
1
2 void calculateBlocksInArea(int width, int height, int &numBlocks, int
  &blockWidth, int &blockHeight) {
3   int w = std::sqrt(numBlocks);
4   int h = w;
5   while (h * w < numBlocks) {
6     if ((h + 1) * w <= numBlocks) {
7       h++;
8     } else {
9       w++;
10    }
11  }
12  numBlocks = w * h;
13  blockWidth = std::ceil(static_cast<double>(width) / w);
14  blockHeight = std::ceil(static_cast<double>(height) / h);
15 }
16
17 void Area::initArea(int blockCount, int centerX, int centerY) {
18   this->blocks = new std::vector<Block>();
19   int blockWidth = 0;
20   int blockHeight = 0;
21   calculateBlocksInArea(this->width, this->height, blockCount,
    blockWidth, blockHeight);
22   logDebug("Area", "Area Approximated to " + std::to_string(blockCount)
    + "(" + std::to_string(blockWidth) + "," +
    std::to_string(blockHeight) + ")");
23   int x = 0;
24   int y = 0;
25   for (int i = 0; i < blockCount; i++) {
26     Block b(x, y, blockWidth, blockHeight);
27     x += blockWidth;
28     if (b.getX() >= this->width) {
29       b.setX(0);
30       b.setY(b.getY() + blockHeight);
31       x = blockWidth;
32       y += blockHeight;
33     }
34     if (b.getY() >= this->height) {
35       // Extra blocks
36       break;
37     }
38     if (b.getX() + b.getWidth() >= this->width) {
39       b.setWidth(this->width - b.getX());
40     }
41     if (b.getY() + b.getHeight() >= this->height) {
42       b.setHeight(this->height - b.getY());
43     }
44     b.orientBlock(centerX, centerY);
45     this->blocks->push_back(b);
46   }
47   logDebug("Area", "Fitted " + std::to_string(this->blocks->size()) + "
    blocks");
48 }
```

Listing 4: da area.cpp

La funzione che gestisce il principale comportamento della torre

```
1 void Tower::start() {
2     if (!this->channel->isUp()) {
3         loge("Can't start tower without a connected channel!");
4         return;
5     }
6     // Register signals
7     signal(SIGINT, Tower::handleSignal);
8     signal(SIGTERM, Tower::handleSignal);
9     this->running = true;
10    std::vector<std::thread> threads;
11    threads.emplace_back(&Tower::droneCheckLoop, this);
12    // threads.emplace_back(&Tower::areaUpdateLoop, this);
13    threads.emplace_back(&Tower::drawGrid, this);
14    logi("Tower online");
15    while (this->running) {
16        // 1' of waiting before restarting the cycle
17        Message *message = this->channel->awaitMessage(10);
18        if (message == nullptr) {
19            // If we have no message to handle, we check last updates from
20            // drones
21            // If a last update is > x second (to decide, maybe 1-5' =>
22            // 60-300'') -> ping and wait a response
23            // If the drone is doing nothing, we commit it to monitor a
24            // zone
25        } else {
26            // Handle message received on another thread, and return to
27            // listen
28            logi("Received message from Drone " +
29                std::to_string(message->getChannelId()) + ". Type: " +
30                std::to_string(message->getType()));
31            threads.emplace_back(&Tower::handleMessage, this, message);
32        }
33        // Free finished threads
34        for (auto it = threads.begin(); it != threads.end(); ) {
35            if (it->joinable()) {
36                it++;
37            } else {it = threads.erase(it);}
38        }
39    }
40    logi("Powering Off");
41    // Disconnect Drones
42    std::vector<Drone> drones = this->getDrones();
43    for (Drone& drone : drones) {
44        if (drone.droneState == DEAD) {
45            continue;
46        }
47        DisconnectMessage *message = new
48            DisconnectMessage(this->generateMessageId());
49        this->channel->sendMessageTo(drone.id, message);
50        delete message;
51    }
52    logi("Waiting threads to finish");
53    // Await Spawned Threads
54    for (auto& thread : threads) {
55        thread.join();
56    }
57    // Flush Channel
58    logi("Flushing Channel");
59    bool channelFlushed = this->channel->flush();
60    if (channelFlushed) {
61        logi("Channel Flushed!");
62    } else {
```

```

56         logw("Can't flush redis channel");
57     }
58     logi("Printing Area");
59     std::ofstream areaFile;
60     areaFile.open("area.csv", std::ios_base::app);
61     for(int i = 0; i < this->areaWidth; i++) {
62         for(int j = 0; j < this->areaHeight; j++) {
63             areaFile << this->area->operator [] (i)[j] << ",";
64         }
65         areaFile << "\n";
66     }
67     areaFile.close();
68     logi("Area Saved on area.scv");
69 }

```

Listing 5: da tower.cpp

La funzione che la torre di controllo utilizza per decidere la prossima posizione da inviare ad un particolare drone

```

1  void Tower::calculateDronePath(Drone drone) {
2      // Check Active Block
3      std::vector<Block> *blocks = this->area->getBlocks();
4      for (Block& block : *blocks) {
5          if (block.getAssignment() == drone.id) {
6              // Check block cells
7              if (block.getLastX() != drone.posX || block.getLastY() !=
8                  drone.posY) {
9                  block.setLastX(drone.posX);
10                 block.setLastY(drone.posY);
11             }
12             int x = block.getLastX() + block.getDirX();
13             int y = block.getLastY();
14             if (x == this->x && y == this->y) {
15                 logi("Skipping Tower Cell");
16                 // Safe because the tower is in the middle
17                 x += block.getDirX();
18             }
19             if (x >= block.getX() + block.getWidth() || x < block.getX()) {
20                 y += block.getDirY();
21                 x -= block.getDirX();
22                 if (y > block.getY() + block.getHeight() || y <
23                     block.getY()) {
24                     // Reset Block and Associate a new One
25                     this->associateBlock(drone);
26                     block.reset(this->x, this->y);
27                     return;
28                 }
29                 block.setDirX(-block.getDirX());
30             }
31             LocationMessage *location = new
32                 LocationMessage(generateMessageId());
33             location->setLocation(x, y);
34             this->channel->sendMessageTo(drone.id, location);
35             delete location;
36             return;
37         }
38     }
39     // Not Returned before -> no Block associated
40     this->associateBlock(drone);

```

Listing 6: da tower.cpp