Relazione su
Controllo formazione droni

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

June 27, 2024

Contents

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.

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 raggiunge 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 direziona 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.

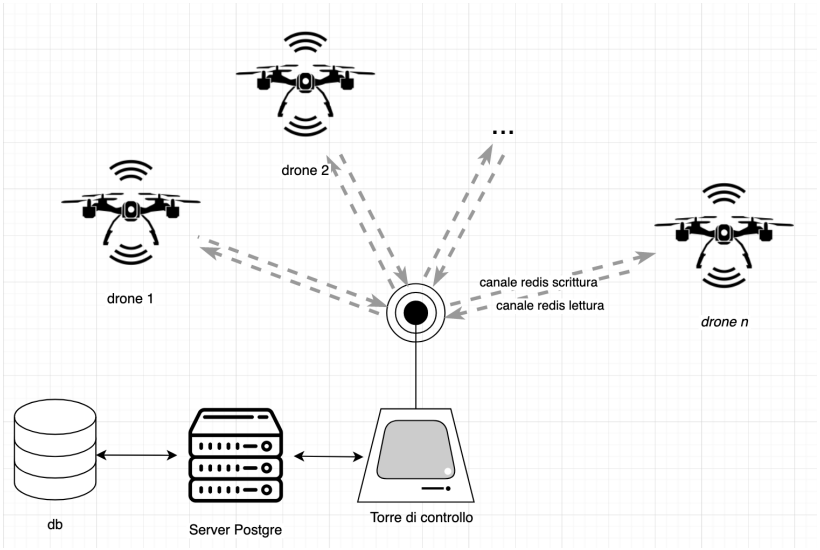


Figure 1: Architettura fisica del sistema

2 Analisi del software

2.1 Requisiti utente

Per funzionare, il sistema ha bisogno dei seguenti requisiti:

- dimensione dell'area
  dato che l'area verrà divisa in blocchi, per ogni blocco è di interesse:
  - limite alto superiore
  - limite basso inferiore
  - punto di partenza (o ultimo punto visitato)
  - 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:
  - id: identificativo univoco seriale
  - 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 andare dalla sua posizione attuale a quella della torre
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 Diagrams

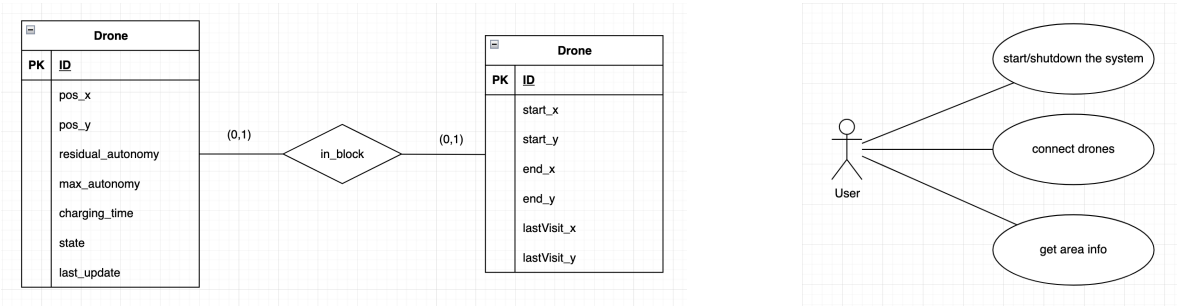


Figure 2: schema ER e UML

2.4 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 che fa in modo di avere più blocchi che 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 azzera il tempo di visita dell'area 20×20 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 marchera 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 si che tutti i punti vengano visitati il più frequentemente possibile. Questo ciclo viene ripetuto finche l'utente non interrompe il programma o tutti i droni "muoiono".

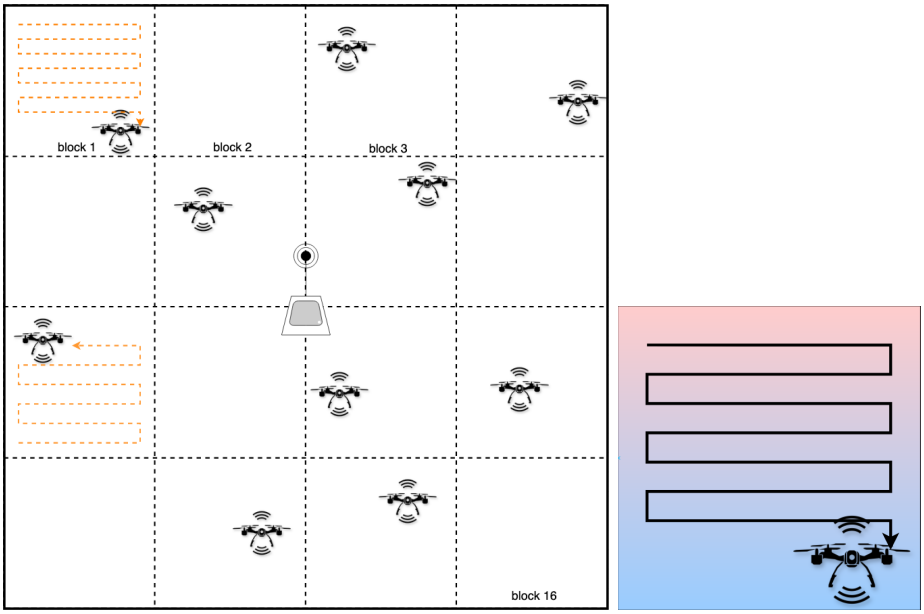


Figure 3: Esempio di area suddivisa in blocchi. Nel dettaglio a destra oltre al percorso seguito dal drone, viene evidenziata come più ”calda” l’area visitata da meno tempo e più ”fredda” l’area appena vista.

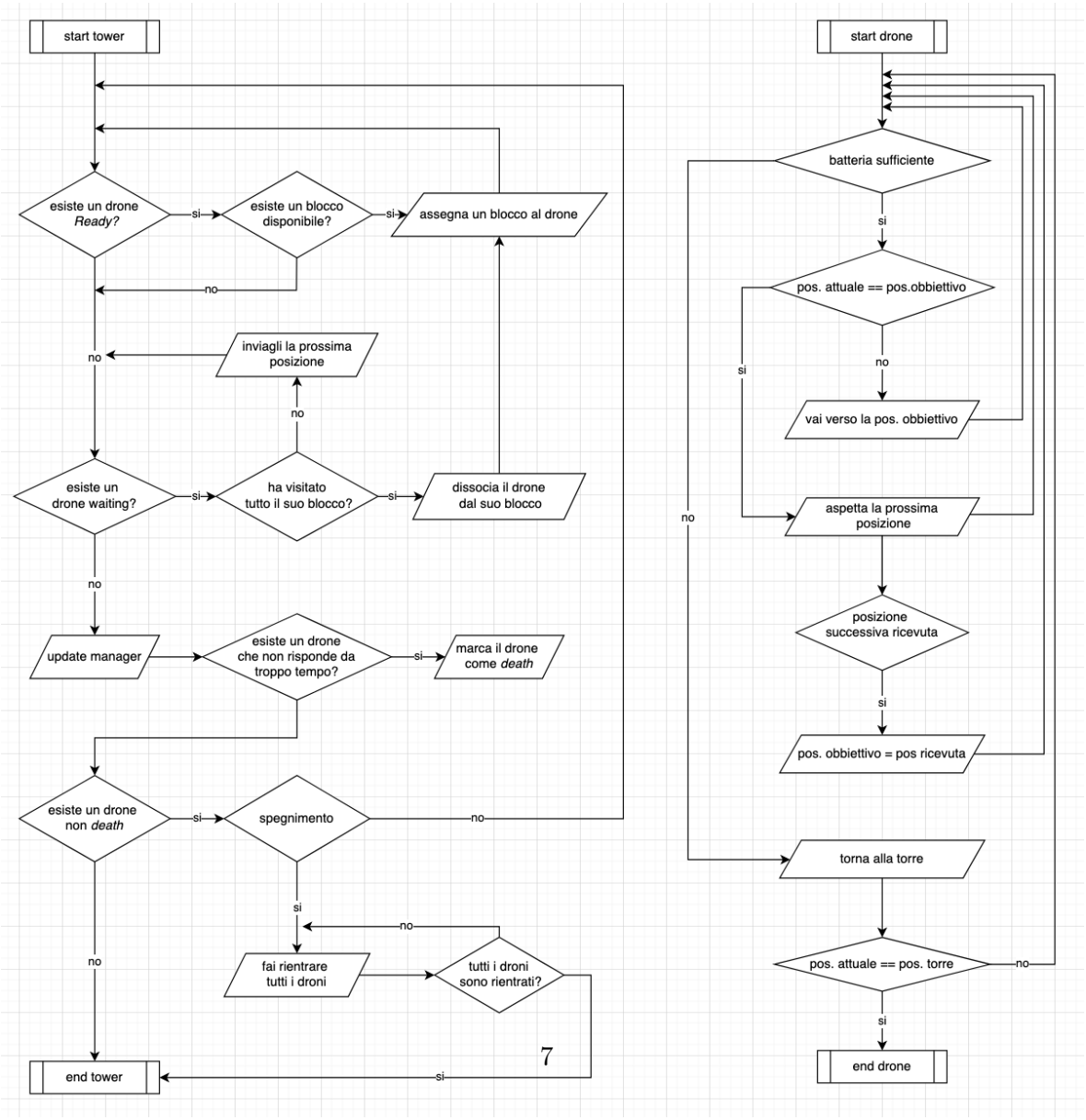


Figure 4: FlowChart della torre e del drone. Nota bene, nel FlowChart viene assunto che il

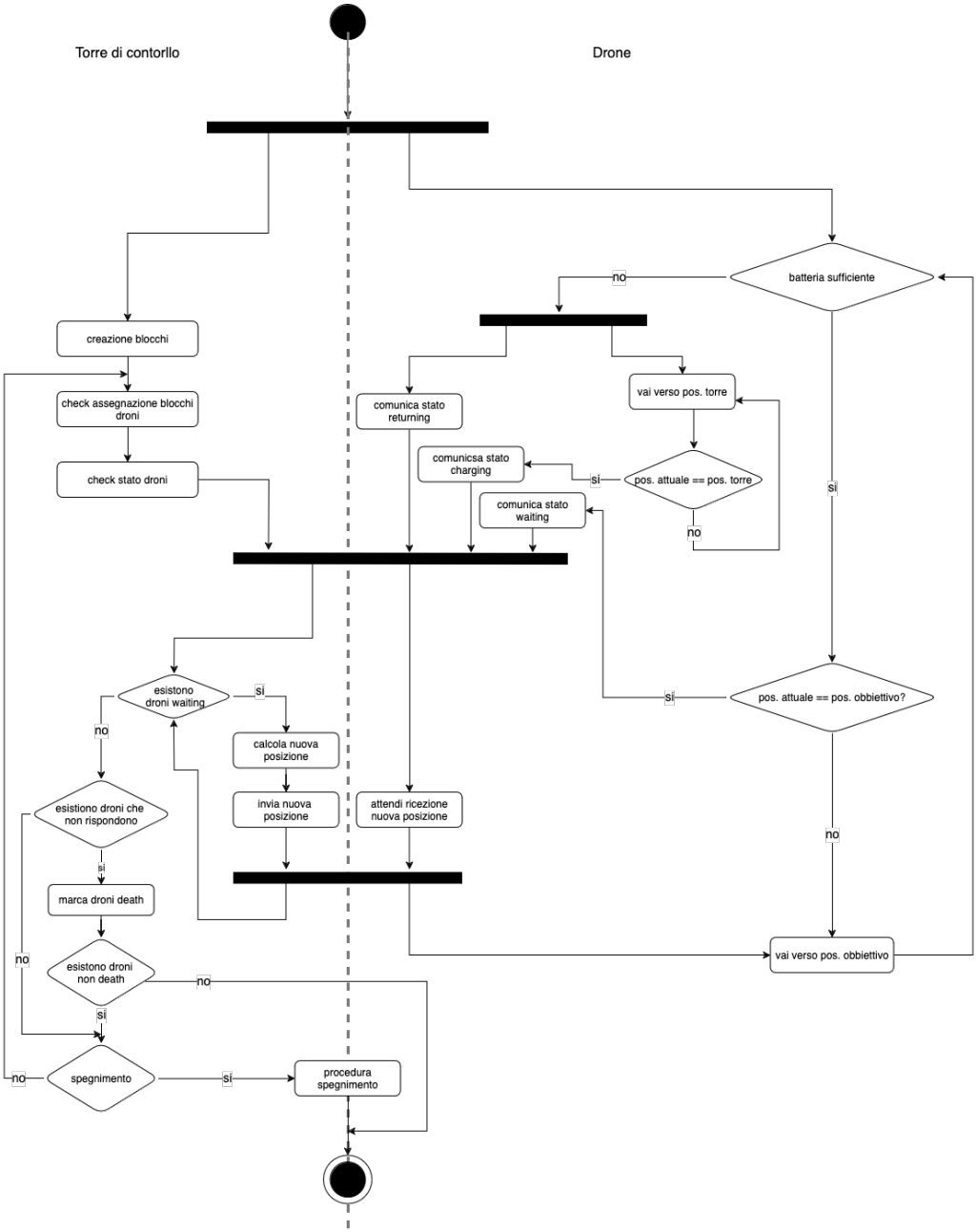


Figure 5: Activity diagram del sistema di interazione e decisione tra torre e drone, nel diagramma sono mostrate due barriere in cui drone e torre si sincronizzano per scambiarsi i messaggi principali mentre è lasciato implicito l'handler message di entrambi per messaggi di check.

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).

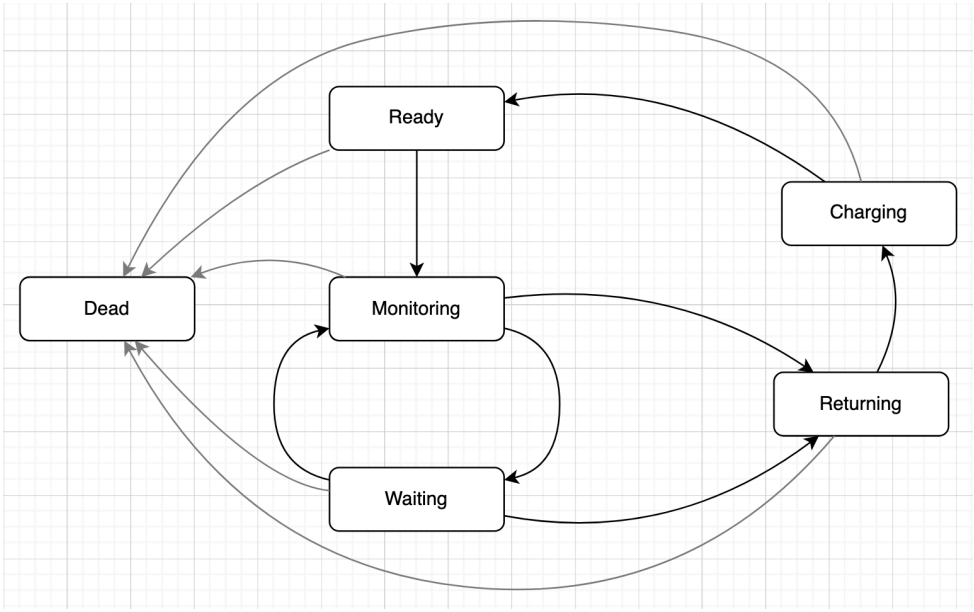


Figure 6: State diagram

2.4.2 Scelte progettuali

Divisione dei blocchi 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 case quadrate di dimensione $10\sqrt{2} \times 10\sqrt{2}$

Abbiamo scelto di raggruppare le caselle in blocchi tutti uguale area 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, utilizziamo un valore di blocchi maggiore per garantirne la perfetta di visibilità in sottoaree uguali. 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.

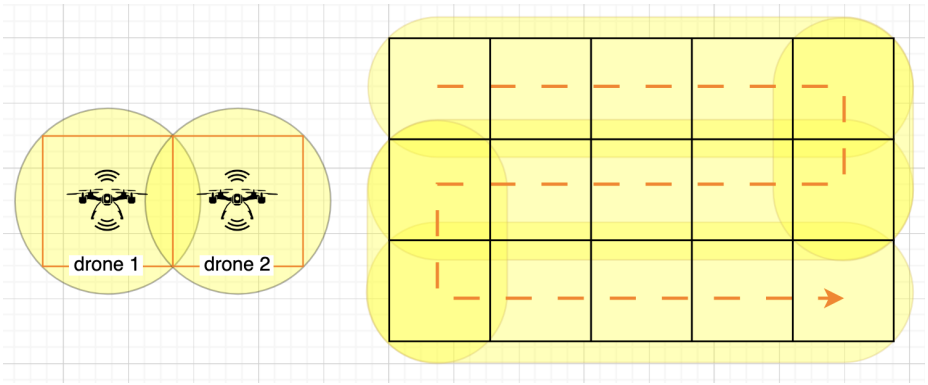


Figure 7: L'area divisa in caselle della dimensione del quadrato inscritto nel cerchio di raggio 10

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**). Se valutiamo il criterio di assegnamento basandoci su tempo di visita medio e tempo di visita massimo raggiunto durante una simulazione, possiamo osservare che a parità di area e di droni il FirstAverageTime ottiene un tempo di visita medio leggermente più basso di quello del FirstMaxTime ma un tempo massimo più alto in quanto tende a non riassegnare dei blocchi (magari quelli più ai margini) che hanno pochi punti con un tempo di visita molto alto ma molti con tempo di visita molto basso. Al contrario il FirstMaxTime al costo di aumentare leggermente il tempo medio di visita, riesce a diminuire sensibilmente il tempo di visita massimo dimostrandosi più equo.

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 del messaggio si ottiene inserendo in un apposito **HashMap** l'mID del messaggio.

3.2 La classe Channel

Per gestire agevolmente la ricezione e l'invio dei messaggio 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 letti e scritti dalla coda utilizzando **"LPUSH c:id message_id"** per inserire un messaggio, ed **"RPOP c:id"** per estrarre il primo messaggio arrivato.

void sendMessageTo(int channelId, Message &message)
Questo metodo permette di aggiungere alla lista di messaggi di channelId il messaggio message tramite i due comandi:

```
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}
```

Listing 1: da tower.cpp

Message* awaitMessage(long timeout = -1)
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.

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 efficienza, 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

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. Dietro le quinte, quello che viene fatto è:
  1. Inviare un comando a redis del tipo *hgetall n:id:mid*
  2. Controllare la validità della risposta
  3. Creare un messaggio tramite il costruttore:
     Message(std::string id)
  4. Parsare la risposta ottenuta tramite:
     message.parseResponse(response)
- **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: La classe PositionMessage, che serve per mandare la posizione di un drone alla torre, presenterà il corpo della funzione parseMessage() di questo tipo:

```
1     return "type 2 x " + std::to_string(this->x) + " y " +
          std::to_string(this->y);
```

Listing 2: da tower.cpp

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.

Il messaggio presenta i seguenti parametri:

- x: La posizione sull'asse x
- y: La posizione sull'asse y

RetireMessage

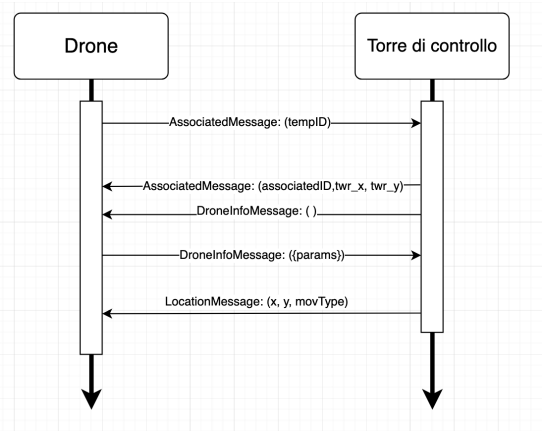
Messaggio inviato dal drone per notificare la necessità di rientro per low battery. In questo caso, la torre invia al drone un LocationMessage con le coordinate precise della torre, e lo mette in stato retiring per ottimizzare il percorso di rientro.

DisconnectMessage

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

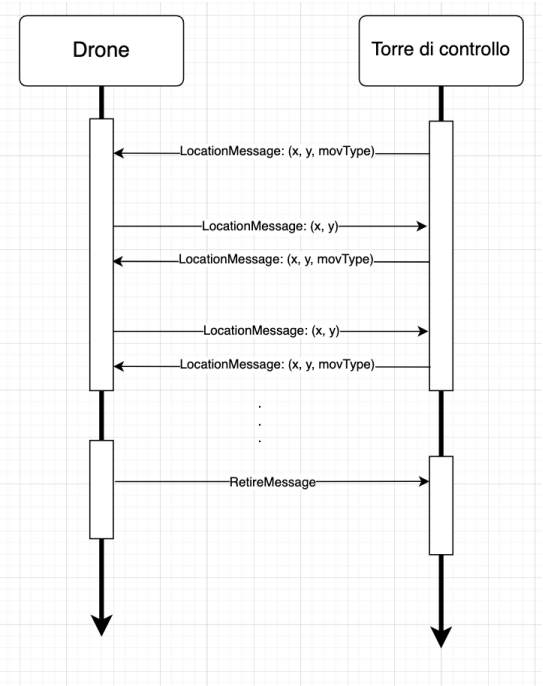
3.4 Message Sequence Chart UML

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.



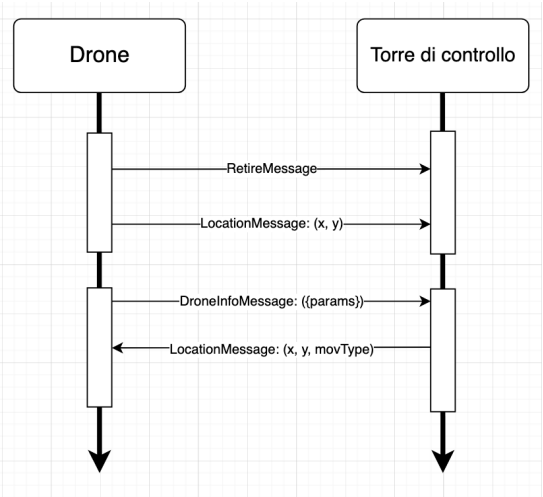
Associazione di un drone:

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 Implementazione

Di seguito vengono riportate parti del codice implementato

```
1   void Drone::start(float executionSpeed) {
2
3       this->setExecutionSpeed(executionSpeed);
4
5       // Enstablish 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
                    drones
20              // If a last update is > x second (to decide, maybe 1-5' =>
                    60-300'') -> ping and wait a response
21              // If the drone is doing nothing, we commit it to monitor a
                    zone
22          } else {
23              // Handle message received on another thread, and return to
                    listen
24              logi("Received message from Drone " +
                    std::to_string(message->getChannelId()) + ". Type: " +
                    std::to_string(message->getType()));
25              threads.emplace_back(&Tower::handleMessage, this, message);
26          }
27          // Free finished threads
28          for (auto it = threads.begin(); it != threads.end(); ) {
29              if (it->joinable()) {
30                  it++;
31              } else {it = threads.erase(it);}
32          }
33      }
34      logi("Powering Off");
35      // Disconnect Drones
36      std::vector<Drone> drones = this->getDrones();
37      for (Drone& drone : drones) {
38          if (drone.droneState == DEAD) {
39              continue;
40          }
41          DisconnectMessage *message = new
                DisconnectMessage(this->generateMessageId());
42          this->channel->sendMessageTo(drone.id, message);
43          delete message;
44      }
45      logi("Waiting threads to finish");
46      // Await Spawned Threads
47      for (auto& thread : threads) {
48          thread.join();
49      }
50      // Flush Channel
51      logi("Flushing Channel");
52      bool channelFlushed = this->channel->flush();
53      if (channelFlushed) {
54          logi("Channel Flushed!");
55      } 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::calcolateDronePath(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() !=
                        drone.posY) {
8                       block.setLastX(drone.posX);
9                       block.setLastY(drone.posY);
10                  }
11                  int x = block.getLastX() + block.getDirX();
12                  int y = block.getLastY();
13                  if (x == this->x && y == this->y) {
14                      logi("Skipping Tower Cell");
15                      // Safe because the tower is in the middle
16                      x += block.getDirX();
17                  }
18                  if (x >= block.getX() + block.getWidth() || x < block.getX()) {
19                      y += block.getDirY();
20                      x -= block.getDirX();
21                      if (y > block.getY() + block.getHeight() || y <
                            block.getY()) {
22                          // Reset Block and Associate a new One
23                          this->associateBlock(drone);
24                          block.reset(this->x, this->y);
25                          return;
26                      }
27                      block.setDirX(-block.getDirX());
28                  }
29                  LocationMessage *location = new
                        LocationMessage(generateMessageId());
30                  location->setLocation(x, y);
31                  this->channel->sendMessageTo(drone.id, location);
32                  delete location;
33                  return;
34              }
35          }
36          // Not Returned before -> no Block associated
37          this->associateBlock(drone);
38      }
```

Listing 6: da tower.cpp