DSmart Flood Monitoring System

Bruno Esposito
bruno.esposito2@studio.unibo.it

Andrea Ingargiola
andrea.ingargiola@studio.unibo.it

Cecilia Teodorani
cecilia.teodorani@studio.unibo.it

7 settembre 2023

DSmart Flood Monitoring System è un'applicazione distribuita che permette di monitorare i possibili allagamenti in una città. In particolare, si tratta di un applicativo pensato in un contesto di smart city, che si interfaccia con una serie di sensori che controllano il quantitativo di pioggia caduta nel tempo. Lo scopo di questa applicazione è di semplificare il monitoraggio e le azioni di intervento da parte dei pompieri, permettendo loro sia di controllare lo stato dei pluviometri sia di intervenire in caso di allarmi di allagamenti.

Indice

1 Obiettivi del progetto

L'obiettivo principale di questo elaborato è quello di approfondire i principi della programmazione distribuita basata sugli attori attraverso lo sviluppo di una proof of concept di un sistema di monitoraggio della pioggia in tempo reale, implementato tramite un'architettura peer to peer che garantisca alta availability e coerenza interna dei dati, senza presentare i colli di bottiglia tipici di un'architettura client-server centralizzata.

1.1 Scenari di utilizzo

Il target di utenti per cui l'applicazione è stata pensata corrisponde alle autorità competenti in materia di gestione delle emergenze nell'ambito delle alluvioni. Le interazioni da parte degli utenti con l'applicazione dovranno avvenire attraverso un'interfaccia grafica sufficientemente intuitiva, che dovrà rappresentare efficacemente e a colpo d'occhio le informazioni più importanti sui rilevamenti dei sensori, sulla loro posizione e sullo stato delle varie zone in cui la città sarà stata suddivisa.

Tramite l'applicativo, un utente avrà modo sia di segnalare la presa in carico delle emergenze, che di silenziare gli allarmi mostrati dal sistema ritenuti non pertinenti.

1.2 Politica di autovalutazione

La qualità del software prodotto verrà attestata tramite un uso estensivo di unit testing, che consentirà di ottenere una percentuale di code coverage ritenuta sufficiente: ogni requisito funzionale avrà un apposito test che ne verifichi il soddisfacimento in termini di correttezza, di affidabilità e di robustezza.

Gli obiettivi dell'elaborato, in termini di risultati finali, verranno raggiunti e considerati soddisfacenti se l'implementazione ottenuta costituirà un'efficace proof of concept per un eventuale applicativo commissionato nel mondo reale: questo comporterà un risultato che verifichi i requisiti funzionali e sia effettivamente distribuito (anche se solo a indirizzi diversi di una sola macchina fisica), ma che non sia necessariamente interfacciato a veri sensori e dislocato su veri nodi remoti fisici.

2 Analisi dei requisiti

2.1 Semplificazioni della simulazione

Per questo elaborato si è scelto di rappresentare una città come un insieme di zone contigue di forma regolare (griglia) in cui:

- Tutte le zone hanno lo stesso numero di pluviometri;
- Una zona entra in allarme alluvione se più della metà dei suoi sensori registra un valore oltre la soglia di guardia (che può cambiare da sensore a sensore);
- Ogni zona ha al suo interno una sola caserma dei vigili del fuoco in grado di risolvere i problemi rilevati in quella specifica zona.

2.2 Requisiti di business

- L'applicazione fornita deve rappresentare un efficace proof of concept che dimostri il buon funzionamento dell'architettura peer-to-peer progettata per soddisfare i requisiti funzionali del programma;
- Dev'essere fornito un client (con interfaccia grafica) che renda l'idea del funzionamento a regime in tempo reale dell'applicativo di monitoraggio, che consenta di provarne le funzionalità principali e che dimostri la correttezza dell'architettura utilizzata.

2.3 Requisiti utente

- Visualizzare i rilevamenti dei pluviometri all'interno della città;
- Interagire con il sistema per prendere in carico/gestire o silenziare gli allarmi;
- Poter configurare una rappresentazione della città personalizzata, con un numero prefissato di zone e una quantità fissa di sensori per zona.

2.4 Requisiti funzionali

- Le zone devono essere in grado di distinguere un'informazione parziale a causa dell'arresto di uno o più pluviometri da una completa in cui tutti i pluviometri hanno comunicato nei tempi previsti i propri rilevamenti;
- Una zona deve segnalare lo stato di crisi in caso la metà più uno dei sensori rilevati in quel momento segnali dei valori fuori soglia;
- Ogni caserma deve avere una visione completa di tutta la città;
- Una caserma deve poter silenziare o prendere in carico gli allarmi della propria zona; se una zona è silenziata dalla propria caserma, risulterà silenziata anche alle altre;

- Ogni client deve poter visualizzare lo storico delle ultime rilevazioni effettuate in tutte le zone della città.

2.5 Requisiti non funzionali

- Usabilità: facilità nell'utilizzo dell'applicativo per monitorare pluviometri distribuiti su larga scala;
- Cross Platform: l'applicativo dev'essere platform-independent ed eseguibile sui 3 principali sistemi operativi: Linux, Windows, MacOs;
- Scalabilità orizzontale: non devono esserci limiti dipendenti dal software nel numero di zone e sensori collegabili al sistema;
- L'applicazione deve sfruttare un'architettura decentralizzata peer-to-peer in cui l'arresto di un singolo componente non compromette il funzionamento generale;
- I componenti legati all'ambito della gestione dei rilevamenti (quindi zone e caserme) devono disporre di un meccanismo di fault tolerance che ne preservi le funzionalità anche in caso uno di quei nodi (hardware o software) non sia più disponibile;
- Il carico di lavoro legato alla gestione dei rilevamenti dev'essere bilanciato tra tutti i nodi della rete peer-to-peer;
- Dev'essere possibile mantenere la persistenza dei dati attraverso un database senza che questo diventi un collo di bottiglia per il funzionamento in tempo reale;
- Dev'essere possibile avviare più client in una stessa caserma senza creare conflitti.

2.6 Requisiti implementativi

- L'applicativo verrà sviluppato in Java (JDK 17);
- Verrà usato il framework ad attori Akka Cluster[3] per la gestione della logica distribuita e di scambio dei messaggi;
- Ci sarà un web service RESTful composto dal client (Vue.js[11] e Typescript) e da un server (Vert.x-Web[10] e Java);
- Per il testing verranno usate le librerie AkkaTestKit e JUnit[7];
- Il database sarà di tipo non-relazionale e verrà implementato tramite MongoDB[8] (e relativo driver Java).

3 Design

Il modello distribuito scelto è quello ad attori, implementato dal framework Akka Cluster. Gli attori sono flussi di controllo che reagiscono ai messaggi che ricevono (processando continuamente la propria coda) senza effettuare busy-waiting dei thread su cui sono eseguiti. In questo modo è possibile garantire una buona scalabilità orizzontale tramite un utilizzo ottimale delle risorse hardware messe a disposizione dalla macchina su cui vengono eseguiti. Akka Cluster rende trasparente la gestione della distanza fisica tra le macchine su cui vengono eseguiti gli attori, collegandoli tutti a uno stesso gruppo (per l'appunto il cluster) che agisce come se fosse eseguito localmente tutto in una stessa macchina.

La filosofia adottata per scegliere la suddivisione dei compiti dell'applicativo è quella di modellare le entità del dominio come attori diversi il più possibile indipendenti tra loro, adattando la struttura del cluster secondo la configurazione di nodi fisica che ci si aspetta verrà utilizzata.

Gli ActorSystem, ovvero i nodi software del cluster, sono suddivisi come in figura:

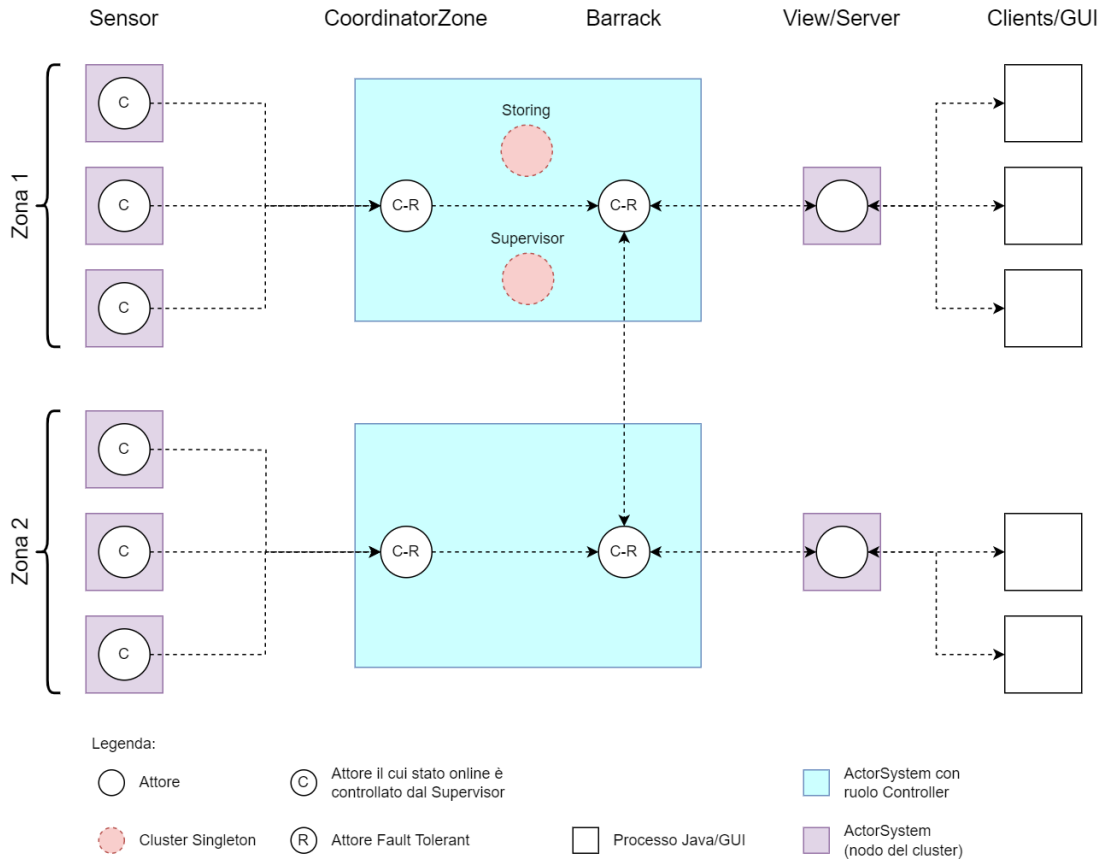

Figura 1: Architettura degli attori e degli ActorSystem.

- Dentro ogni ActorSystem viene istanziato un attore Deployer (non rappresentato in figura) che si occupa della creazione degli altri attori presenti in quel nodo;
- Ogni pluviometro è contenuto all'interno di un ActorSystem diverso;
- Ogni zona ha un rispettivo ActorSystem (di un tipo diverso rispetto a quelli di sensori e gui) che contiene una possibile istanza dei singleton Supervisor e Storing, l'attore CoordinatorZone e l'attore Barrack;
- Ogni attore View è all'interno di un proprio ActorSystem e contiene un proprio server (quindi uno per zona) del web service.

3.1 Struttura

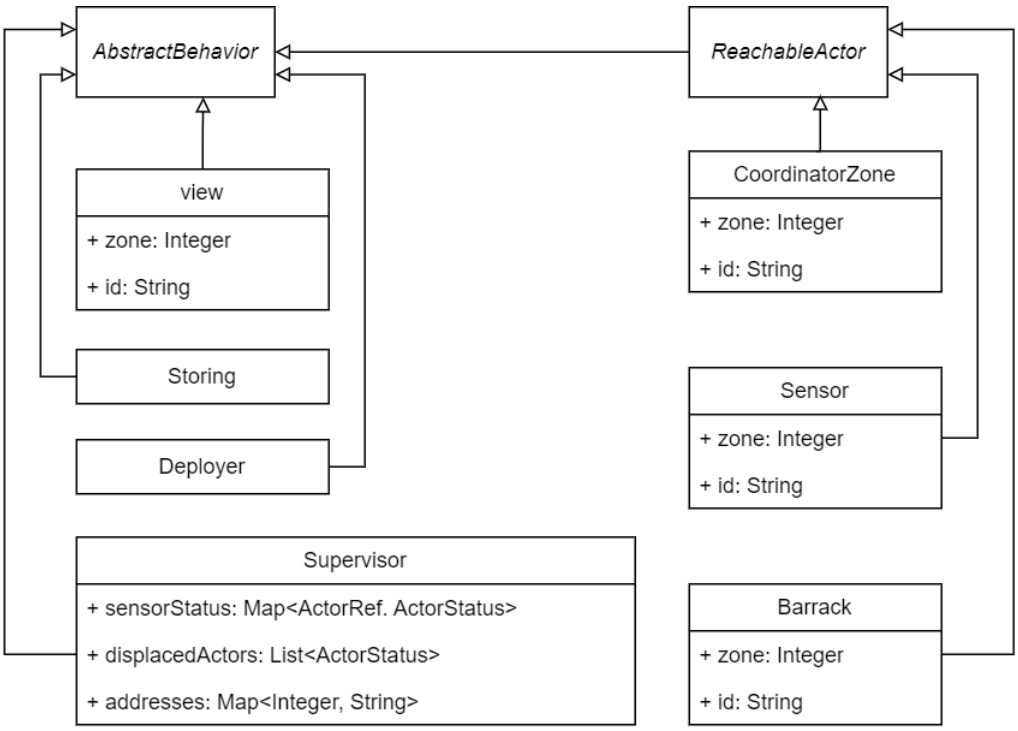

Figura 2: Diagramma UML delle classi degli attori.

Gli attori, che rispecchiano le funzionalità principali del programma, sono:

- **Deployer** è l'attore che si occupa di creare gli altri attori schierati in quel nodo, a seconda dei messaggi che riceve dal Builder (la classe Java che decifra la struttura della città e inizializza il Cluster all'avvio del sistema) o da Supervisor;
- **Sensor** è la rappresentazione nel sistema del singolo pluviometro. Nel contesto della nostra simulazione, i valori rilevati vengono generati casualmente ogni n secondi e mandati al controllore di zona solo dietro richiesta da parte di quest'ultimo;

- **CoordinatorZone** è l'attore responsabile dell'elaborazione dei dati grezzi da parte dei sensori che fanno parte di una zona specifica. Ogni n secondi genera una chiave di iterazione che viene incapsulata nel messaggio di richiesta dei valori rilevati mandato in broadcast ai sensori di quella zona. Nell'intervallo di tempo tra una richiesta e l'altra colleziona i valori ricevuti dai sensori, ritenendoli validi solo se contengono la chiave dell'iterazione corrente. Allo scadere del tempo, prima di richiedere nuovamente i valori, il CoordinatorZone elabora tutti i valori ricevuti, determinando lo stato ("FLOOD" oppure "OK") della zona secondo le regole della simulazione (quindi la metà più uno dei valori rilevati che superano le soglie dei rispettivi sensori) e lo invia alla rispettiva Barrack. Nel caso non tutti i sensori abbiano risposto nei tempi previsti, l'elaborazione avviene soltanto sui valori parziali, marcando il risultato inviato alla caserma come frutto di una vista parziale del sistema;

- **Barrack** è l'attore che rappresenta la singola caserma dei vigili del fuoco adibita al controllo di una specifica zona. Questo attore si occupa di ricevere i dati dal proprio controllore di zona, incrociarli con i possibili comandi ricevuti dagli utenti attraverso l'interazione con l'interfaccia grafica (quindi l'eventuale silenziamento o la presa in carico degli allarmi provenienti da quella zona) e mandarli all'attore View che costituisce il server all'interno di quella caserma, all'attore che si occupa di salvare i dati storici su MongoDB e agli attori delle altre caserme, in modo che tutte le caserme abbiano una visione d'insieme dello stato dell'intera città;

- **Storing** è l'attore che si occupa di salvare su un database locale i dati ricevuti dalle caserme tramite il Mongo Java Driver;

- **Supervisor** è l'attore che implementa i meccanismi di fault tolerance e di bilanciamento del carico di lavoro tra nodi diversi;

- **View** è l'attore che costituisce il server del servizio web di una certa zona. Esso riceve in tempo reale dalla propria caserma tutti i dati rilevati all'interno della città ed interagisce con i client, cioè tutte le interfacce grafiche su cui operano i pompieri della caserma.

3.2 Comportamento e interazioni

I vari attori all'interno del cluster riescono a comunicare attraverso il Receptionist: ogni attore, a seconda della sua classe, si registra al Receptionist usando una o più chiavi (ognuna delle quali rappresenta un canale broadcast diverso) e il suo indirizzo verrà restituito agli altri attori ogniqualvolta uno di questi attori esegue la Receptionist.find di una delle chiavi con cui si è registrato.

Le chiavi utilizzate nell'applicazione sono:

- gossipProtocol, a cui si collegano tutti i gli attori che estendono ReachableActor (quindi sensori, zone e caserme) e che viene usata per lo scambio dei messaggi di Ping e Pong dal Supervisor;
- sensors:{#zona}, a cui si collegano tutti i sensori di una determinata zona per essere raggiunti dal controllore di zona;
- barrack:{#zona}, a cui si collega la caserma di una certa zona per essere raggiunta dal controllore di zona;
- barracks, a cui si collegano tutte le caserme e il singleton dello Storing per ricevere dati dalle altre caserme e ricostruire lo stato dell'intera città;
- guis:{#zona}, a cui si collega l'attore View di una specifica zona per essere raggiunto dalla rispettiva caserma;
- deployers, a cui si collegano tutti i deployer del cluster, che devono essere raggiungibili dal Supervisor per generare gli attori displaced.

L'esatta definizione di queste chiavi avviene nelle istanze di una singola classe immodificabile (campo di tutti gli attori dell'applicazione) chiamata IdGenerator.

3.2.1 CoordinatorZone

Il CoordinatorZone si occupa di elaborare i dati dei sensori, rimanendo in un ciclo infinito in cui:

1. Viene generata una chiave di iterazione univoca;
2. Vengono richiesti al Receptionist gli indirizzi dei sensori sintonizzati sulla chiave sensors:{#zona}, cioè tutti i sensori di quella zona;
3. Se il numero di indirizzi ottenuto è inferiore a quello nominale, l'elaborazione avverrà comunque marcando i dati come parziali;
4. Una volta ottenuti i dati da tutti i sensori raggiunti dal Receptionist, viene computata l'istantanea dello stato della zona e mandata alla rispettiva caserma (raggiunta sempre via Receptionist).

Degno di nota è il fatto che il CoordinatorZone non tiene traccia dello stato in cui si trovava all'iterazione precedente e non tiene conto delle interazioni dell'utente (allarme silenziato, in gestione o risolto).

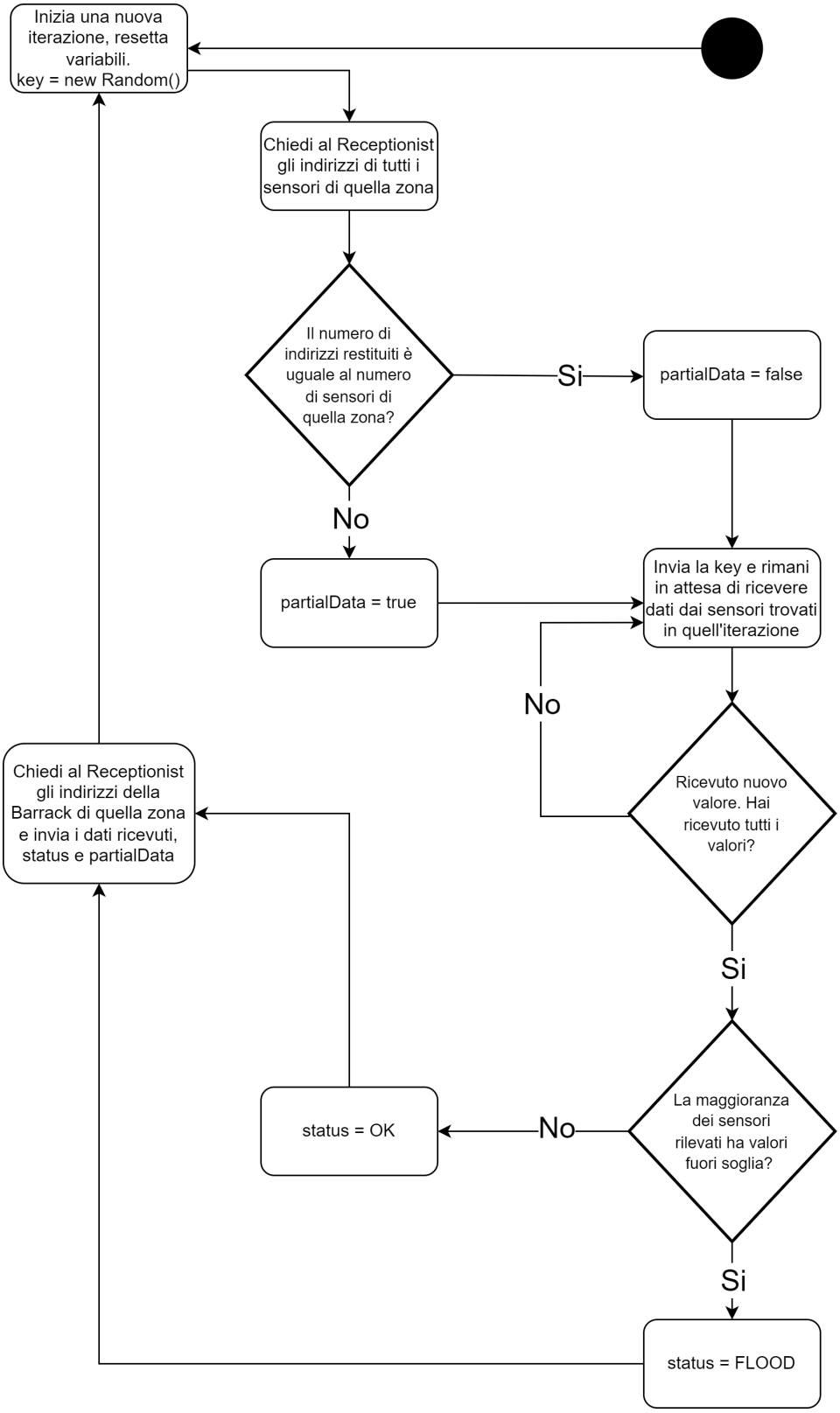

Figura 3: Activity-diagram del CoordinatorZone.

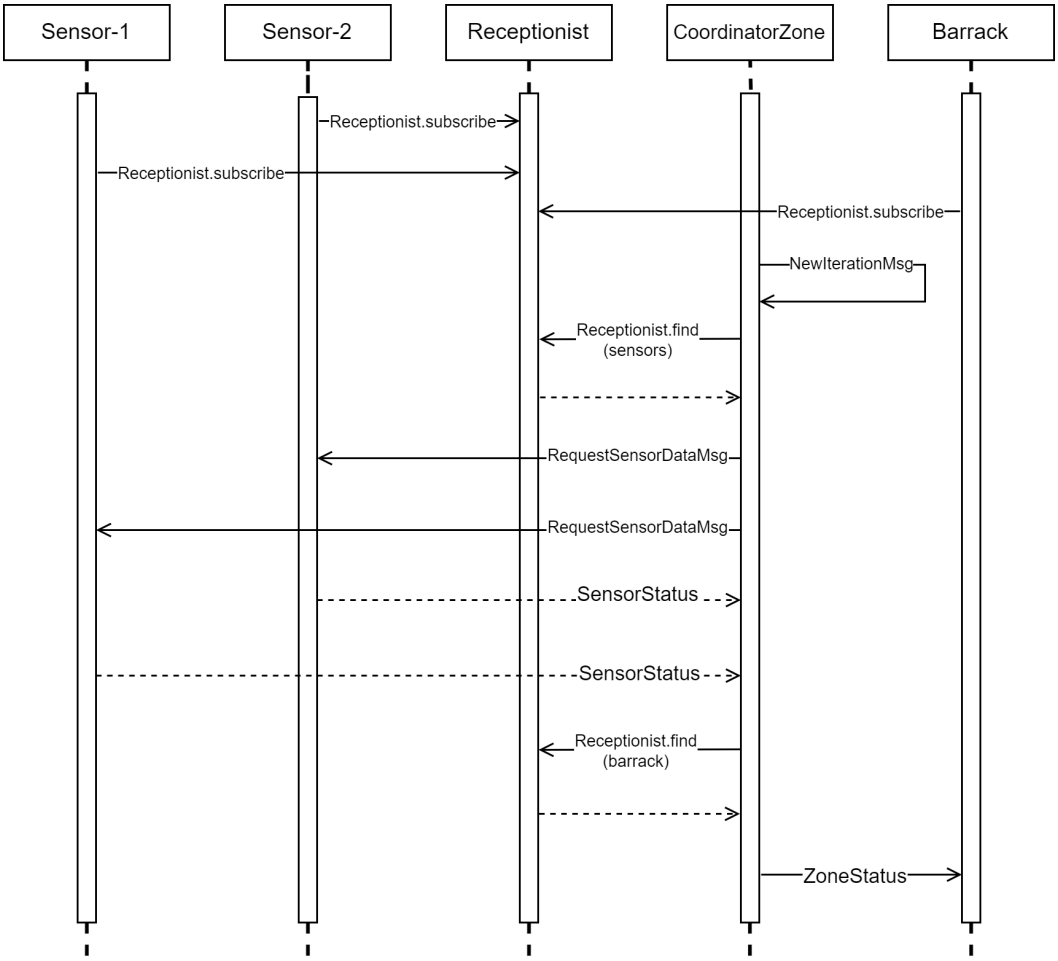

Figura 4: Sequence-diagram delle interazioni tra CoordinatorZone e istanze di Sensor.

3.2.2 Barrack

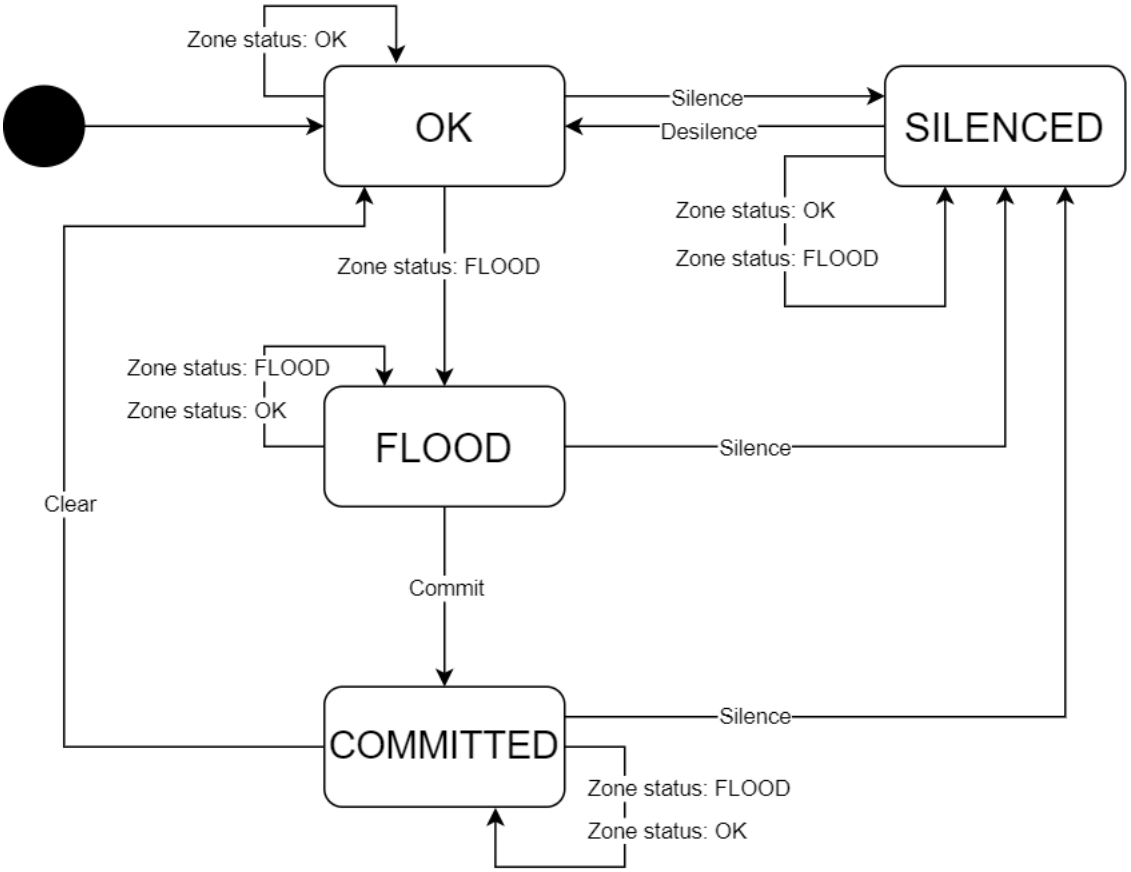

Figura 5: State-diagram dei possibili risultati della computazione dell'attore Barrack.

La Barrack si occupa di incrociare le computazioni istantanee ricevute dal Coordinator-Zone con le possibili interazioni degli utenti, determinando lo stato complessivo di quella porzione di città, da cui i client possono inferire sia lo stato di occupazione della caserma stessa che l'eventuale allarme della zona. Una volta calcolato lo stato complessivo, questo viene diffuso in broadcast sia agli altri attori sintonizzati sulla chiave Receptionist *barracks* (cioè le caserme delle altre zone e l'attore che si occupa del salvataggio dello storico su MongoDB), che all'attore View corrispondente (e cioè il server del servizio web di quella specifica caserma).

Come facilmente intuibile, una Barrack è in grado di ricevere i dati dalle altre Barrack, in modo da poter fornire al proprio client una vista complessiva dello stato di tutte le zone della città.

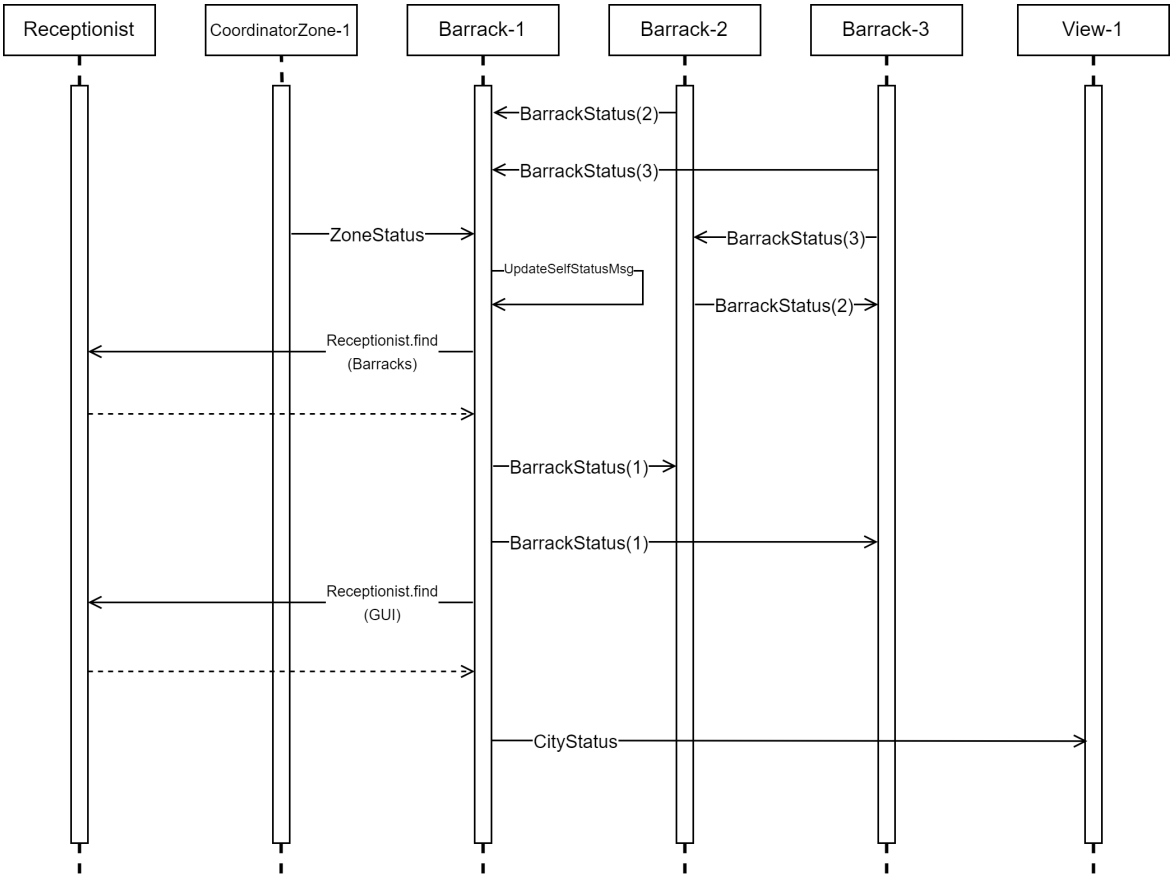

Figura 6: Sequence-diagram delle interazioni tra caserme e tra caserma e zona.

3.2.3 Supervisor

La logica di funzionamento del Supervisor è ispirata al protocollo Gossip di Akka: ogni n secondi spedisce un messaggio di Ping contenente una chiave di iterazione univoca (che dovrà essere restituita dal Pong) a tutti gli attori che ereditano dalla classe astratta ReachableActor, che implementa i metodi di risposta a quello specifico tipo di messaggio. Possiede tre strutture dati principali:

- clusterStatus, che tiene traccia degli attori di cui ha ricevuto un Pong entro le ultime tre iterazioni;
- displacedActors, che identifica gli attori "displaced", ovvero quegli attori ricreati per volere del Supervisor che vanno a coprire i ruoli degli attori ritenuti irraggiungibili;
- addresses, che contiene una mappa degli indirizzi degli ActorSystem in cui sono normalmente operativi Barrack e CoordinatorZone.

Le elaborazioni che avvengono per gli attori che non hanno risposto allo scadere del tempo sono:

- Per ogni Barrack e CoordinatorZone, riduce di uno le vite disponibili (vengono ripristinate a tre ogni volta che rispondono correttamente) nella clusterStructure;
- Per ogni attore della clusterStructure che ha raggiunto zero vite e che non sia un Sensor o un attore displaced di cui è tornato disponibile l'attore originale, crea il rispettivo attore displaced;
- Per ogni zona di cui ci sono ancora dei Sensor presenti e che non ha Barrack o CoordinatorZone, crea i rispettivi attori displaced.

Il supervisor è in grado di accorgersi e creare attori sostituivi per le Barrack e le CoordinatorZone nei casi in cui:

- Un intero ActorSystem si disconnetta dal Cluster prima o durante l'esecuzione del Supervisor;
- Un'attore appartenente a quelle due classi si arresti per qualunque motivo prima o durante l'esecuzione del Supervisor;
- Vengano connessi al Cluster attori di tipo Sensor che non hanno i rispettivi Barrack e/o CoordinatorZone.

La logica di deployment degli attori "displaced" segue il principio del load balancing, cercando di posizionare i nuovi attori laddove ne fossero già presenti in numero minimo.

Nello schema UML della pagina seguente viene illustrato lo scambio di messaggi del caso in cui l'ActorSystem con ruolo "Controller" della zona 2 (quello contenente la rispettiva Barrack e il rispettivo CoordinatorZone) si fosse scollegato dal Cluster prima dell'esecuzione del Supervisor, e, dopo un certo intervallo di tempo, tornasse operativo.

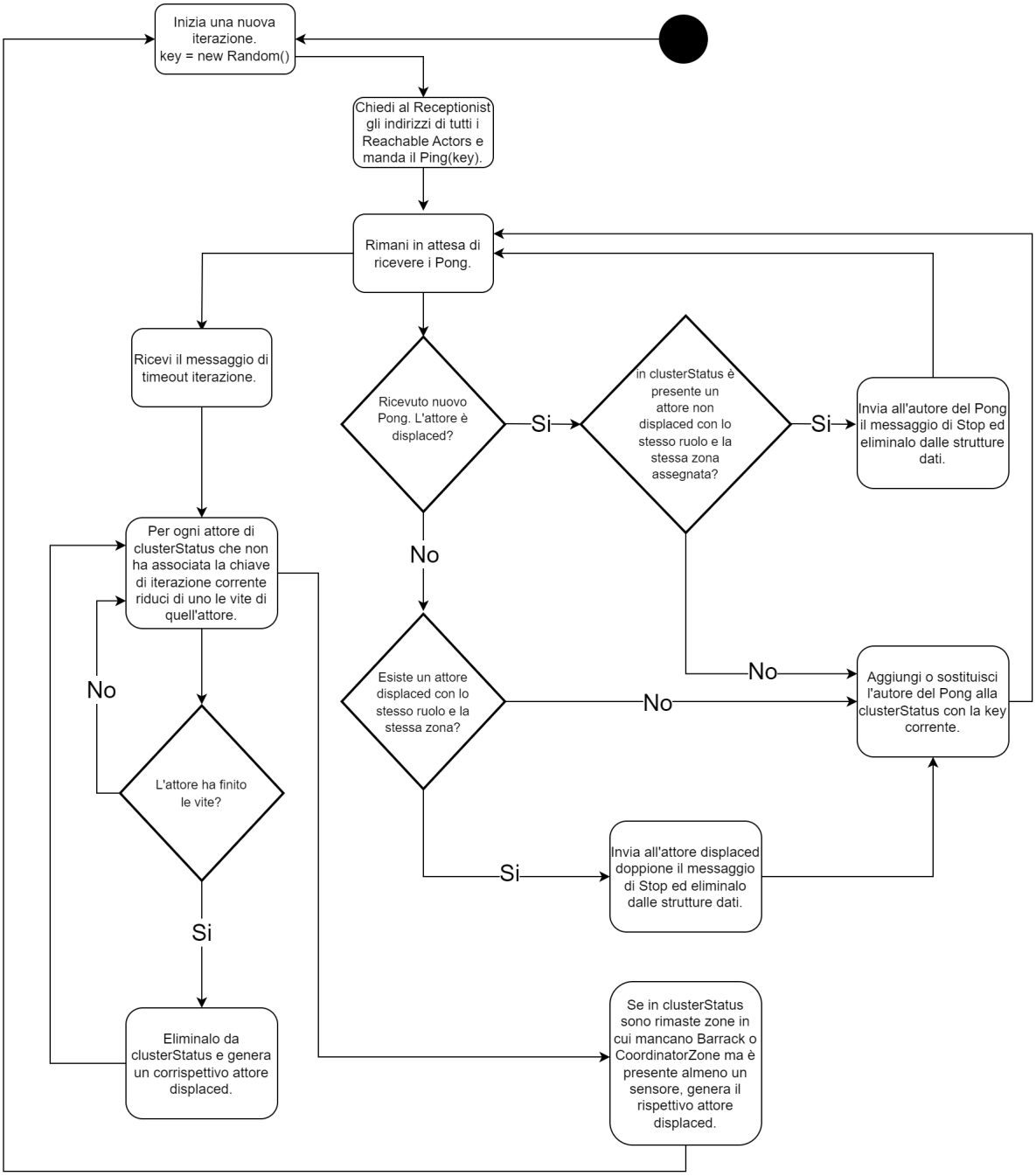

Figura 7: Activity-diagram dell'iterazione del Supervisor.

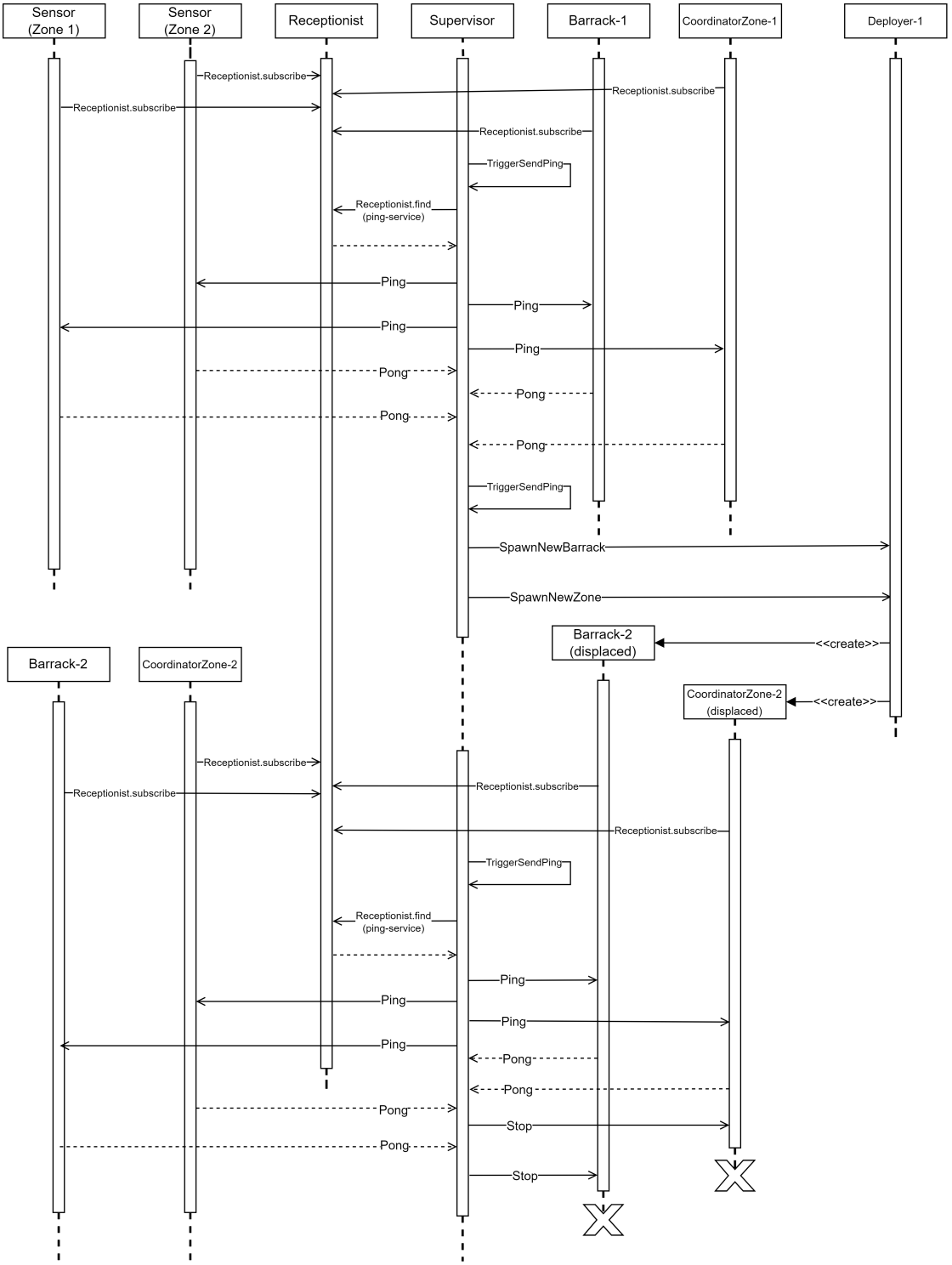

Figura 8: Sequence-diagram del funzionamento del Supervisor.

3.2.4 View

Come accennato in precedenza, l'attore View svolge il ruolo di intermediario tra il sistema di monitoraggio degli allagamenti, composto da attori distribuiti, e i client utilizzati dai vigili del fuoco della caserma nella loro zona di competenza. All'interno di questo attore, viene creato un Server per il servizio Web RESTful. Per ottenere i dati in tempo reale, il client effettua opportune richieste API al Server ogni n secondi.

La View riceve continuamente messaggi dalla Barrack della zona di riferimento, che contengono informazioni sullo stato attuale dell'intera città e sui valori rilevati da ciascun sensore di ogni zona. Queste informazioni vengono inviate alla GUI solo quando questa effettua una specifica chiamata API al Server.

L'utente può interagire con il sistema tramite l'interfaccia grafica per eseguire azioni nella propria zona di competenza:

- silenziare/desilenziare gli allarmi in arrivo;
- mettere un allarme in gestione, se la zona è in allarme;
- terminare la gestione di un allarme.

La View viene informata di queste azioni tramite una richiesta API fatta dal client, in cui viene specificata l'azione desiderata. Una volta ricevute queste indicazioni, la View invia un messaggio alla propria Barrack, comunicando le modifiche apportate. Successivamente, la Barrack modificherà il proprio stato in base alle informazioni ricevute.

Ogni qual volta il client richieda lo storico delle rilevazioni, il Server effettua una connessione con il database centrale, nel quale vengono salvate tutte le misurazioni avvenute in città. Le informazioni visualizzate dall'utente sono in tempo reale ed è stato scelto di estrarre le ultime cento rilevazioni raccolte.

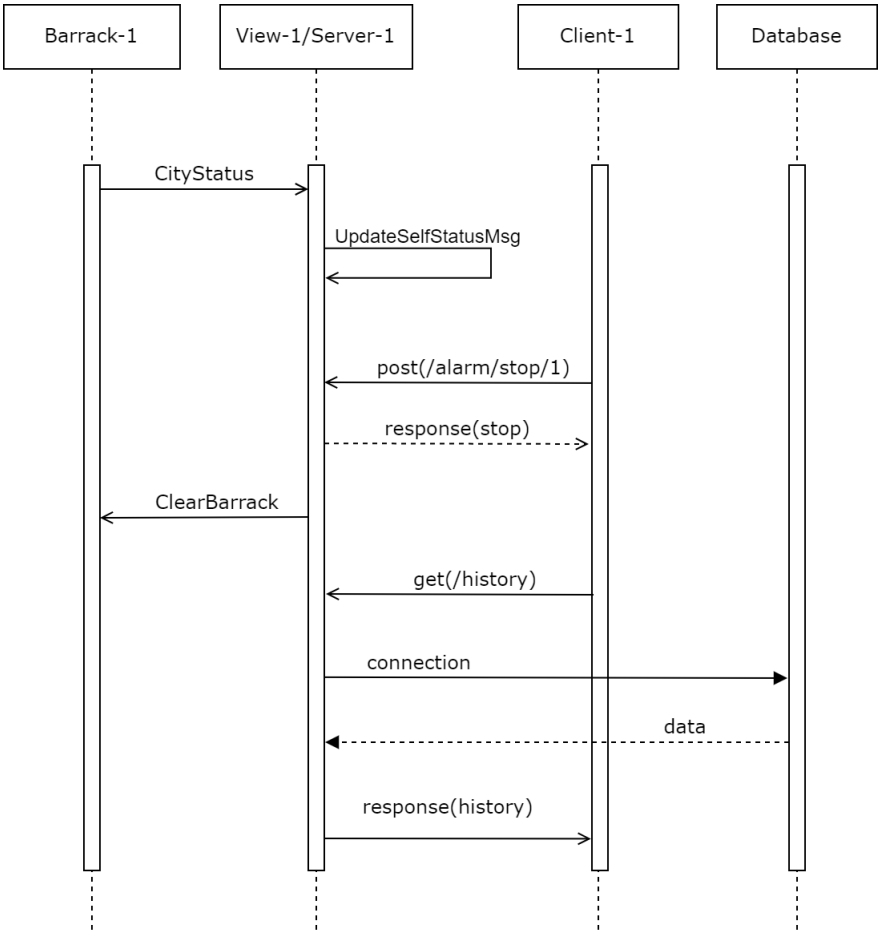

Figura 9: Sequence-diagram del funzionamento della View.

4 Dettagli implementativi

4.1 Strumenti di Akka utilizzati

4.1.1 Receptionist

Akka Receptionist è un modulo del framework Akka[1], progettato per semplificare la gestione delle richieste di servizio e la distribuzione di informazioni tra attori all'interno di un sistema basato su Akka.

In sostanza, Akka Receptionist fornisce un'implementazione pronta all'uso di un modello di servizio di registrazione e ricerca. Permette agli attori di registrarsi come fornitori di servizi e di cercare servizi registrati da altri attori all'interno del sistema.

Quando un attore desidera fornire un servizio, può registrarsi presso il Receptionist fornendo una chiave univoca associata al servizio. Allo stesso modo, quando un attore desidera utilizzare un servizio, può cercarlo nel Receptionist specificando la chiave corrispondente al servizio di interesse.

Il Receptionist funge da intermediario, mantenendo un registro dei servizi registrati e consentendo agli attori di richiederli in modo efficiente.

4.1.2 Cluster Singleton

Quando si utilizza il Cluster Singleton, viene creato un attore di una specifica classe nel cluster che assume il ruolo di Singleton. Questo attore sarà responsabile di gestire il compito assegnato e avrà un nome univoco all'interno del cluster. Se l'istanza Singleton dovesse terminare per qualsiasi motivo, il Cluster Singleton garantirà che una nuova istanza venga creata automaticamente in un altro nodo del cluster.

Ciò significa che il Cluster Singleton offre una soluzione robusta per la gestione di attività che richiedono un'unica istanza all'interno del cluster Akka, come ad esempio l'accesso a una risorsa condivisa o il coordinamento di operazioni complesse tra più nodi.

Il Cluster Singleton di Akka si basa sul protocollo di elezione leader utilizzato per selezionare il nodo che ospiterà l'istanza Singleton. In caso di fallimento di un nodo o di problemi di connettività, il protocollo di elezione garantirà che un nuovo leader venga selezionato per mantenere attiva l'istanza Singleton nel cluster.

All'interno della nostra applicazione gli attori Singleton sono Supervisor e Storing, entrambi progettati per funzionare correttamente anche in caso di ripetuti riavvii dovuti alla disconnessione dei nodi leader su cui sono inizialmente dispiegati.

4.2 Configurazione di Akka Cluster

Per quanto riguarda la creazione del Cluster e dei relativi componenti, è stato necessario definire programmaticamente all'interno di *Builder* la loro configurazione. In particolare, è stata considerata una configurazione di base (*initBasicConfig*) - in modo tale da inizializzare il cluster - che permette lo scambio dei messaggi sulla rete tramite il protocollo TCP, la loro relativa serializzazione mediante il serializzatore di Akka *jackson-json* e la simulazione di più nodi fisici distribuiti nella stessa JVM.
In seguito, a partire da questa, sono state implementate due diverse estensioni: una relativa alla semplice aggiunta di ulteriori nodi all'interno del cluster (*setConfig*) e l'altra per la definizione e l'aggiunta dei cluster singleton (*setClusterConfig*).
Dunque, la creazione del Cluster avviene attraverso la creazione di ActorSystem aventi ciascuno una delle suddette configurazioni, a seconda del ruolo che devono assumere all'interno dell'architettura del sistema.

4.3 Load balancing e fault tolerance

In questa sezione verranno illustrate nel dettaglio le tre funzioni private del Supervisor che si occupano di generare gli attori displaced, bilanciando il carico di lavoro tra i diversi nodi del cluster.

```
1  private Behavior<ValueMsg> onListing(ListingResponse msg) {
2      String receptionistKey = msg.listing.getKey().id();
3      switch(receptionistKey){
4          case "gossipProtocol" -> {
5              msg.listing.getServiceInstances(ServiceKey.create(
                   ValueMsg.class, idGenerator.getPingKey()))
6                        .forEach(b -> b.tell(new Ping(getContext().
                             getSelf(), this.actualKey)));
7          }
8          case "deployers" -> {
9              Map<String, ActorRef<ValueMsg>> deployers = msg.listing.
                   getServiceInstances(ServiceKey.create(ValueMsg.class,
                   "deployers"))
10                       .stream()
11                       .map(d -> new Pair<>(d.path().address().hostPort
                             ().equals(this.clusterStructure.getClusterName
                             ()) ?
12                                       this.addresses.get(this.
                                             supervisorDeployedInZone) :
13                                       d.path().address().hostPort(), d))
14                       .filter(pair -> this.addresses.containsValue(pair
                             .first()))
15                       .collect(Collectors.toMap(Pair::first, Pair::
                             second));
16
17             this.spawnNewActor(deployers, this.toSpawn);
18         }
19     }
20
21     return Behaviors.same();
22 }
```

Questa funzione è parte del pattern Adapter che consente al Supervisor di gestire i messaggi Listing del Receptionist. Nel caso in cui la chiave di broadcast sia "gossipProtocol" invia a tutte le ActorRef ricevute il nuovo Ping a cui dovranno rispondere; invece, nel caso in cui sia "deployers" estrai la lista di ActorRef appartenenti agli attori Deployer istanziati su un nodo con ruolo Controller. Questo implica tenere soltanto gli attori il cui indirizzo corrisponde a una delle possibili zone salvate nella lista addresses. In questa lista filtrata è incluso anche l'attore il cui indirizzo corrisponde al nome del cluster, che tramite prove sperimentali si è visto essere il Deployer del nodo in cui è attivo il Supervisor.

Una volta ottenuta la lista di Deployer attualmente attivi sul cluster, viene richiamata la funzione ricorsiva spawnNewActor, passandogli come argomenti anche la lista di coppie ruolo-zona di cui è necessario generare un attore displaced.

```
23 private void spawnNewActor(Map<String, ActorRef<ValueMsg>> nodes,
       List<Pair<Integer, String>> toSpawn) {
24         if(toSpawn.isEmpty()){
25             this.toSpawn.clear();
26             return;
27         }
28         Pair<Integer, String> actorToSpawn = toSpawn.get(toSpawn.size
               () - 1);
29         this.spawnNewActor(nodes, toSpawn.subList(0, toSpawn.size() -
               1));
30         List<Integer> triedZones = new ArrayList<>(actorToSpawn.first
               ());
31         Integer zone = actorToSpawn.first();
32         for(int i = 0; i < this.addresses.size(); i++){
33             Integer zoneToDeploy = this.getLessCrowdedZone(triedZones
                   );
34             if(!nodes.containsKey(this.addresses.get(zoneToDeploy)))
                   triedZones.add(zoneToDeploy);
35             else {
36                 ActorRef<ValueMsg> deployer = nodes.get(this.
                       addresses.get(zoneToDeploy));
37                 switch (actorToSpawn.second()){
38                     case "CoordinatorZone" -> {
39                         System.out.println("SENDING MESSAGE TO
                               DEPLOYER TO SPAWN A NEW ZONE");
40                        deployer.tell(new SpawnZoneActor(idGenerator.
                              getZoneId(zone), zone, clusterStructure.
                              getSensorPerZone()));
41                     }
42
43                     case "Barrack" -> {
44                         System.out.println("SENDING MESSAGE TO
                               DEPLOYER TO SPAWN A NEW BARRACK");
45                         deployer.tell(new SpawnBarrackActor(zone));
46                     }
47                 }
48                 this.displacedActors.add(new ActorStatus(zone,
                       actorToSpawn.second(), zoneToDeploy,
                       DEFAULT_MAX_LIVES));
49                 return;
50             }
51         }
52         throw new IllegalStateException("Cluster down");
53     }
```

Questa funzione ricorsiva rappresenta il nucleo del funzionamento del Supervisor: dopo aver cercato la zona con meno attori displaced tra quelle il cui Deployer ha risposto alla chiamata via Reecptionist, invia il relativo messaggio al relativo Deployer, generando la rispettiva rappresentazione nella collezione displacedActors. Ogni passo della ricorsione genera un attore diverso, e termina nel momento in cui la lista toSpawn è vuota.

4.4 Servizio Web RESTful

Come già ampiamente spiegato in precedenza, per ottenere i dati e le informazioni rilevate dal sistema di monitoraggio, è stato creato un web service RESTful composto da un client e da un server. Questo servizio è stato progettato in modo che ogni zona abbia il proprio server e possa accogliere più client che si connettono ad esso.

Come client è stata sviluppata un'applicazione web single page, al fine di consentire all'utente (un pompiere) di interagire con il sistema di monitoraggio degli allagamenti distribuito tra attori. La web app è stata realizzata utilizzando il framework *Vue.js* e *Typescript*, ma per avere un'interfaccia con uno stile moderno, reattiva e di facile utilizzo sono stati usati componenti della libreria *PrimeVue*[9]. Per quanto riguarda la scelta dei colori dell'applicazione e l'aspetto generale di essa (ad esempio, le icone, la spaziatura tra i componenti, i font dei testi), sono stati seguiti quanto più possibile i principi espressi dalla libreria *Material Design*[4], che esprime linee guida per la creazione di interfacce utente intuitive, coerenti e coinvolgenti.

Il server, sviluppato con *Vert.x-Web* e *Java*, è ospitato all'interno dell'attore View. Al momento della creazione del Server vengono dichiarate ed esposte le API, così che il client possa effettuare le richieste in un qualsiasi momento. Le API che vengono esposte dal server sono:

- get: /zones (per avere la lista di tutte le zone della città)
- get: /myzone (per sapere qual è la propria zona)
- get: /sensors (per avere il numero di sensori presenti nella zona selezionata)
- get: /barrackstatus (per avere lo stato della zona selezionata)
- get: /citysize (per avere le dimensioni della città)
- get: /sensorscoords (per avere le coordinate di tutti i sensori delle varie zone)
- get: /history (per avere le ultime rilevazioni effettuate nella città)
- post: /zone/:numZone (per selezionare una specifica zona)
- post: /alarm/:action/:numZone (per cambiare lo stato della zona specificata)

L'effettivo reperimento ed utilizzo dei dati nel frontend dell'applicazione viene effettuato tramite la libreria *Axios*[2], la quale permette in modo semplice di effettuare delle richieste HTTP e comunicare quindi con il server attraverso la sua API. *Axios* utilizza le Promise o chiamate async per permettere la computazione asincrona quando si riceve eventualmente la risposta dal server. A seguito di una risposta dal server, è poi possibile utilizzare nel client i dati, che sono restituiti dal server direttamente in un formato JSON-like.

5 Verifica e autovalutazione

Il nostro obiettivo è stato quello di realizzare una proof of concept di un eventuale applicativo relativo al mondo reale. In particolare, il risultato ottenuto verifica tutti i requisiti funzionali espressi ed è testato a livello distribuito, sia in termini di correttezza e affidabilità che di robustezza.
Per quanto riguarda i test, ci siamo concentrati innanzitutto sulla correttezza della costruzione del sistema distribuito verificando in che modo tutti i componenti principali, come i diversi tipi di attori e di cluster, venissero schierati alla creazione. In seguito, abbiamo testato i meccanismi di interazione tra gli attori principali del sistema, in modo tale da verificare l'affidabilità nel loro scambio di informazioni anche a livello distribuito. Infine, abbiamo testato con successo anche i meccanismi di fault tolerance e load balancing considerando alcuni casi particolari ed altri generici, garantendo così una maggiore affidabilità e robustezza del sistema.
Tutti gli unit tests implementati vengono superati con successo e la test coverage complessiva è più che sufficiente, poiché supera il 95% di classi e l'84% di metodi e di linee di codice:

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	95,3% (41/43)	84,4% (221/262)	84,3% (786/932)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
distributed	75% (3/4)	76,7% (23/30)	80,9% (106/131)
distributed.messages	100% (4/4)	100% (13/13)	100% (22/22)
distributed.messages.barrack.commands	100% (4/4)	100% (4/4)	100% (4/4)
distributed.messages.selftriggers	100% (5/5)	100% (5/5)	100% (6/6)
distributed.messages.spawn	80% (4/5)	76,5% (13/17)	75,9% (22/29)
distributed.messages.statuses	100% (4/4)	95,5% (21/22)	97,4% (38/39)
distributed.model.actors	100% (8/8)	91,4% (74/81)	95,3% (384/403)
distributed.model.utility	100% (4/4)	96,6% (28/29)	97,7% (42/43)
distributed.utils	100% (3/3)	92% (23/25)	97,3% (71/73)
distributed.view	100% (2/2)	47,2% (17/36)	50% (91/182)

generated on 2023-07-24 17:22

Figura 10: Test Coverage.

La modalità test-driven di sviluppo ci ha permesso di risolvere da subito importanti problematiche relative sia ai requisiti funzionali e sia allo sviluppo distribuito del sistema. Infatti, nelle fasi intermedie di sviluppo, è stato possibile addirittura migliorare i risultati delle nostre implementazioni partendo proprio da alcune estensioni dei test relativi alle fasi iniziali. In conclusione, riteniamo che il sistema ottenuto soddisfi alcuni dei requisiti più importanti di un'architettura distribuita come la scalabilità, l'affidabilità, la fault tolerance, la gestione dei dati, la comunicazione e il load balacing.

6 Istruzioni per il deployment

Per poter provare in locale l'intero sistema, è necessario aver installato *Gradle*[6], *Docker*[5] e avviare il sistema.

Come avviare il sistema:

1. Posizionarsi in

   ```
   ./
   ```

2. Avviare il sistema con il comando

   ```
   docker-compose up
   ```

3. Per visualizzare la web app, aprire il browser e collegarsi a

   ```
   http://localhost:3000/
   ```

7 Esempi di utilizzo

L'unico caso d'uso previsto è quello in cui il sistema è attivato sulla rete di sensori e caserme dell'autorità preposta al controllo delle alluvioni (vigili del fuoco e/o protezione civile). Il funzionamento a regime consiste in una serie di client web dispiegati in una o più postazioni fisse all'interno delle caserme, in cui gli addetti interagiscono con il programma attraverso l'apposita interfaccia grafica nel caso in cui siano presenti degli allarmi.

Di seguito sono riportate le schermate della web app sviluppata.

La pagina iniziale presenta diversi elementi, tra cui il titolo del progetto e un menu, il quale permette di spostarsi verso la pagina dedicata allo storico delle rilevazioni (Figura 19) e di ritornare alla pagina iniziale. Inoltre, è indicata la zona di competenza della caserma che ha aperto l'interfaccia e la mappa della città in cui ci si trova.

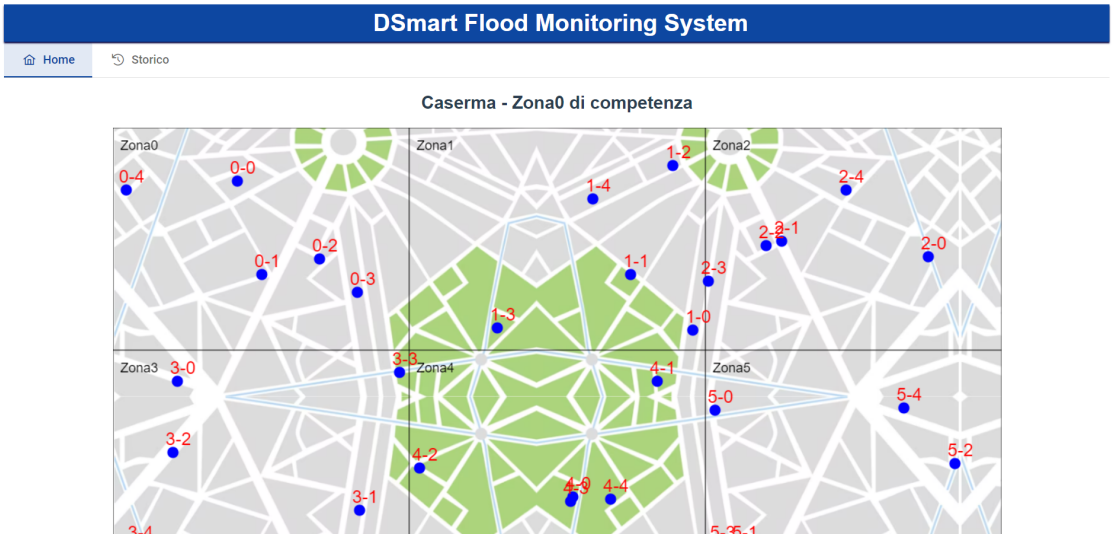

Figura 11: Schermata pagina iniziale.

Nella sezione successiva della pagina iniziale, è presente la restante parte della mappa della città e un'area dedicata alla selezione di una zona specifica. Dopo aver effettuato la selezione, tutte le informazioni relative a quella zona vengono visualizzate in tempo reale nella parte inferiore della pagina. Oltre alle informazioni sulla zona, sono riportati i nomi dei componenti del progetto e l'anno accademico in cui è stato seguito il corso di riferimento del progetto.

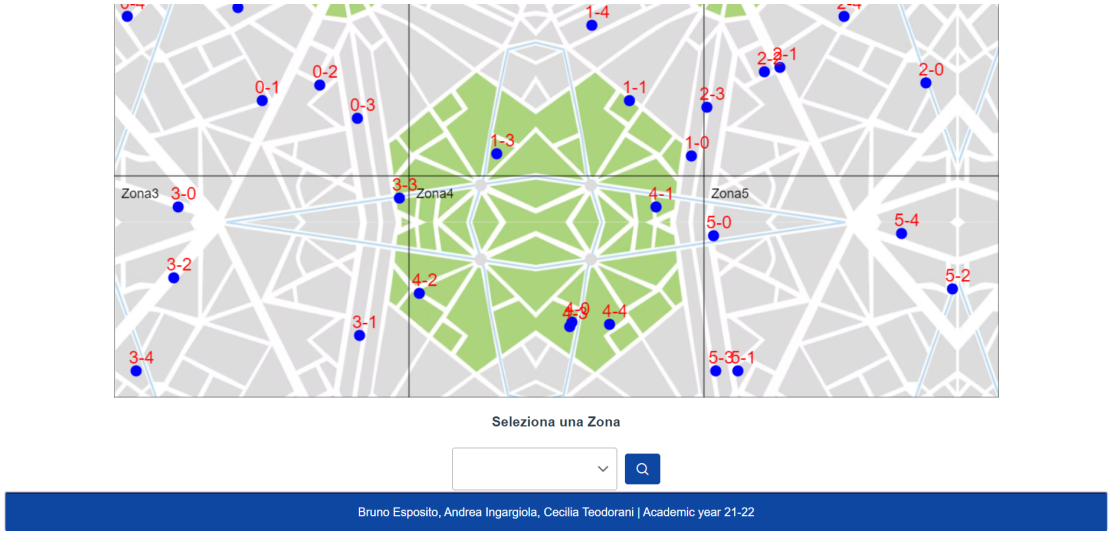

Figura 12: Schermata pagina iniziale.

In modo più dettagliato, l'intera città è rappresentata attraverso una mappa esemplificativa che permette di visualizzarla nella sua interezza. Ogni zona rappresentata ha un certo numero di sensori distribuiti nella sua area, i quali vengono indicati da un cerchio blu e da un testo rosso, seguendo il formato *numeroZona-numeroSensore*.

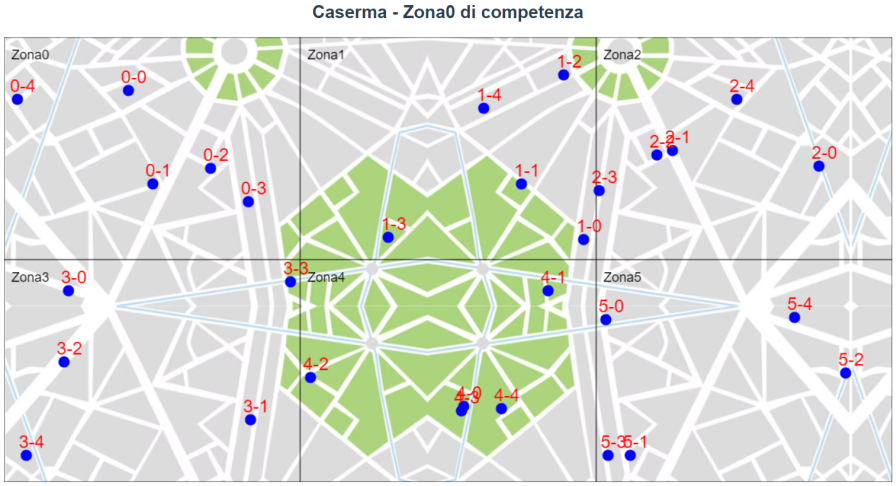

Figura 13: Dettaglio della mappa della città.

Una volta selezionata una zona, vengono visualizzate e aggiornate tutte le informazioni relative in tempo reale:

- stato della zona;
- stato della caserma;
- numero di pluviometri presenti in quella zona;
- se i dati in arrivo sono parziali o meno;
- tutti i valori rilevati dai sensori con la relativa soglia, un limite che oltrepassato indica troppe precipitazioni.

In caso venga selezionata la zona di competenza, c'è anche la possibilità di silenziare tutti gli allarmi attraverso un bottone.

Figura 14: Schermata zona di competenza senza allarme.

Come già spiegato in precedenza, nel momento in cui più della metà dei sensori rileva valori di pioggia sopra la propria soglia, scatta l'allarme di allagamento di quella zona specifica. Lo stato della zona rimarrà in allarme anche se più della metà dei pluviometri dovesse successivamente misurare valori al di sotto della soglia. L'unico modo per modificare lo stato della zona è gestire l'allarme, premendo il rispettivo bottone.

Figura 15: Schermata zona di competenza con allarme.

Una volta che è stato premuto il bottone "Gestisci allarme", la zona ha lo stato "In gestione" e la caserma è occupata nella sua gestione. Una volta che sono stati effettuati gli interventi sul territorio da parte dei pompieri, allora la gestione dell'allarme può terminare, riportando la zona in situazione di normalità, come in Figura 14.

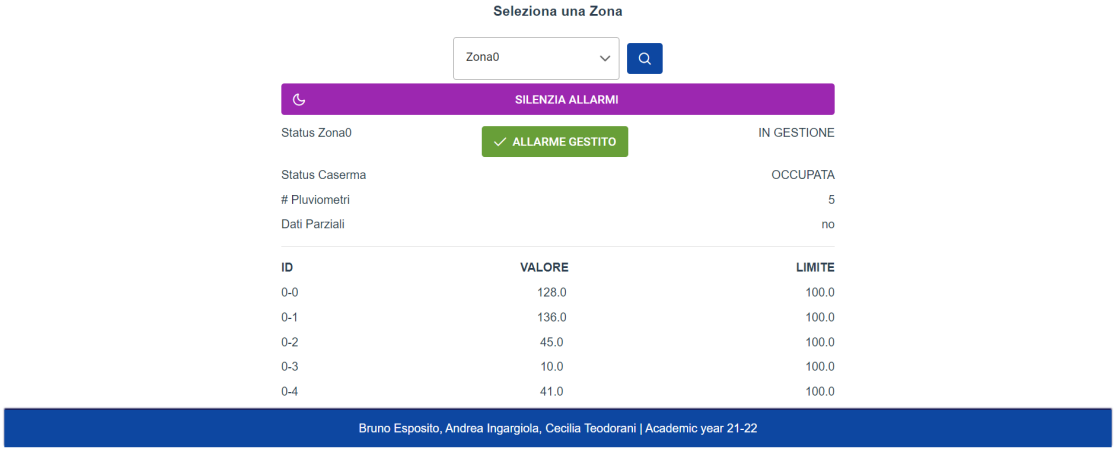

Figura 16: Schermata zona di competenza con allarme gestito.

Nel caso in cui vengano silenziati gli allarmi, lo stato della zona diventa e rimane OK fino a quando non si decide di de silenziarli. I valori rilevati dai pluviometri vengono comunque visualizzati e modificati dopo ogni misurazione.

Figura 17: Schermata zona di competenza con allarmi silenziati.

Se viene selezionata una zona non di competenza della caserma che ha aperta l'interfaccia, vengono visualizzati comunque tutte le informazioni in tempo reale, ma non c'è la possibilità di gestire o silenziare gli allarmi.

Figura 18: Schermata zona non di competenza della caserma.

Le ultime rilevazioni della città vengono visualizzate nella pagina dello storico, la quale contiene tutte le informazioni necessarie, tra cui la zona con il suo stato, il giorno e l'ora della misurazione e tutti i dati relativi a ciascun sensore presente.

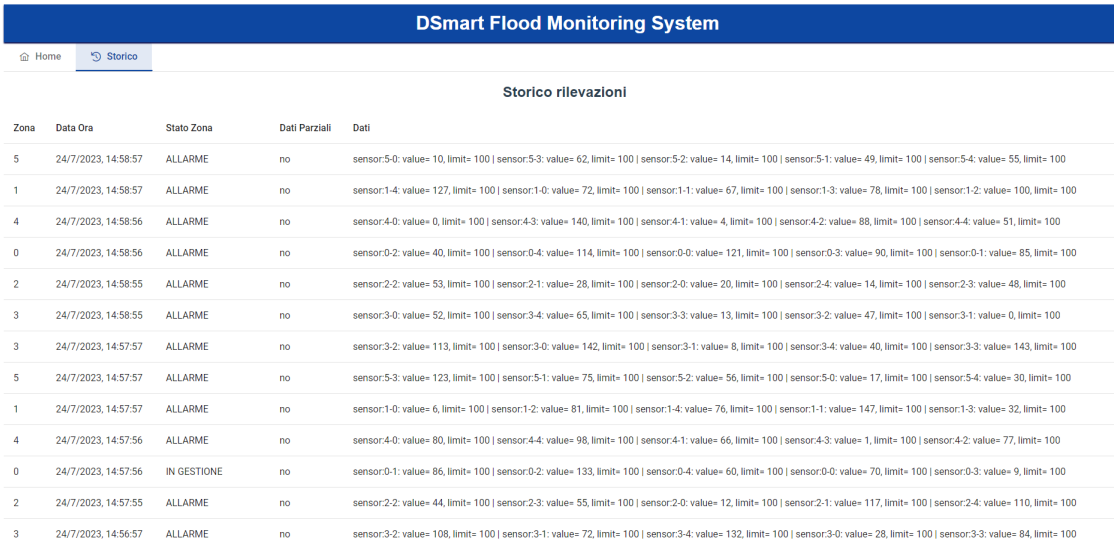

Figura 19: Schermata storico rilevazioni città.

8 Conclusioni

Al termine di questo elaborato è stato ottenuto un applicativo distribuito basato sugli attori che, oltre a rispettare i requisiti funzionali appartenenti al dominio della gestione delle alluvioni, è in grado di garantire fault tolerance e load balancing sulle componenti software che possono essere rimpiazzate (quindi gli attori di controllo non direttamente dipendenti dall'hardware dei sensori).

8.1 Sviluppi futuri

Lo sviluppo futuro consiste nel passaggio dell'applicativo da proof-of-concept a vero e proprio prototipo, funzionante sia su veri sensori IoT che su una rete costituita da nodi fisici distribuiti nel territorio della città. Dal momento che in questo progetto sono state considerate semplificazioni per la simulazione, per renderlo coerente con la realtà sono necessarie alcune modifiche:

- le zone possono avere dimensioni differenti tra loro e anche rispettivamente un numero diverso di sensori;
- ogni città può presentare un numero di caserme di vigili del fuoco diverso rispetto al numero di zone; ciò implica che ciascuna zona è dotata di un coordinatore che invia i dati pertinenti alla caserma più vicina o assegnata, secondo criteri di load balancing.

8.2 Cosa abbiamo imparato

Attraverso questo progetto è stato possibile consolidare le competenze relative alla gestione degli attori e della programmazione asincrona basata sullo scambio di messaggi, con un particolare approfondimento sui possibili meccanismi (e problematiche connesse) di fault tolerance, load balancing e consistenza dei dati registrati. In più, è stato possibile migliorare le competenze di gestione di una web app real time e la creazione di un servizio web RESTful all'interno di un contesto distribuito.

Riferimenti bibliografici

[1] Akka. https://akka.io.

[2] Axios. https://axios-http.com/docs/intro.

[3] Akka Cluster. https://doc.akka.io/docs/akka/current/typed/index-cluster.html.

[4] Material Design. https://m2.material.io/design.

[5] Docker. https://www.docker.com.

[6] Gradle. https://gradle.org.

[7] JUnit. https://junit.org/junit5/.

[8] MongoDB. https://www.mongodb.com/it-it.

[9] PrimeVue. https://primevue.org/.

[10] Vert.x-Web. https://vertx.io/docs/vertx-web/java/.

[11] Vue.js. https://vuejs.org/.