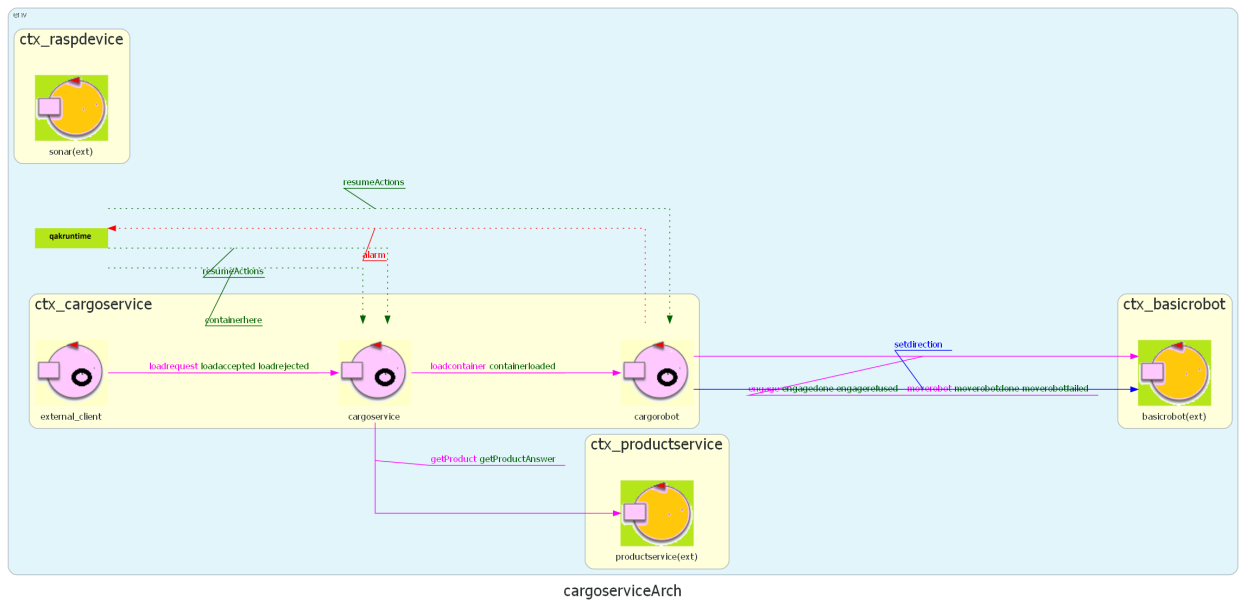
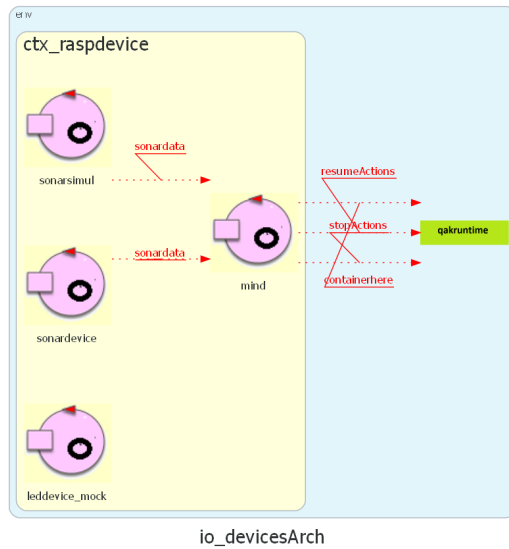


# Sprint 3

## Architettura iniziale dello sprint



# Obiettivi

Sviluppare un'interfaccia grafica e un dispositivo LED per monitorare visivamente lo stato del sistema.  
In particolare i requisiti su cui ci concentreremo in questo sprint sono:

- The **slots5** area is **permanently occupied**, while the other slots are initially empty
- 4. Shows the current state of the **hold**, by means of a dynamically updated **web-GUI**.
- 5. **Interrupts** any activity and turns on a LED if the **sonar sensor** measures a distance  $D > D_{FREE}$  for at least 3 seconds (possibly a sonar failure).  
The service continues its activities as soon as the sonar measures a distance  $D \leq D_{FREE}$

## Analisi del Problema

### CargoServiceStatusGui

Inizialmente, nel modello di alto livello dello Sprint 0, `cargoserviceStatusGui` era stato concepito come un componente strettamente accoppiato a `cargoservice`, potenzialmente residente nello stesso contesto. Tuttavia, un'analisi più approfondita ha rivelato la necessità di un'architettura più robusta e disaccoppiata. Per questo motivo, si è deciso di implementare la GUI e il suo backend in un **contesto separato** (`ctx_cargoservicestatusgui`). Questa scelta strategica garantisce la separazione delle responsabilità (logica di business vs. logica di presentazione) e migliora la manutenibilità e la scalabilità future del sistema, trattando i due contesti come microservizi indipendenti.

Oltre al requisito originale di visualizzazione dello stato della stiva, si è aggiunto un nuovo requisito funzionale:

La Web GUI deve permettere a un utente esterno di **inviare una richiesta di carico** (`loadrequest(PID)`) direttamente dall'interfaccia, ricevendo una notifica di successo (`loadaccepted`) o fallimento (`loadrejected`).

Questo requisito introduce una doppia responsabilità per il backend della GUI:

1. **Flusso in uscita (Push):** Ricevere passivamente gli aggiornamenti di stato dal `cargoservice` e inoltrarli all'interfaccia web.
2. **Flusso in entrata (Request):** Accettare attivamente comandi dall'interfaccia web, inoltrarli al `cargoservice` e gestire il ciclo di richiesta/risposta.

Per gestire questa duplice natura in modo pulito e aderire al **Principio di Singola Responsabilità** anche a un livello più granulare, si è deciso di suddividere il backend della GUI in **tre attori distinti**, ognuno con un compito altamente specializzato.

Il flusso di operazioni si articola quindi come segue:

- **gui\_api\_gateway (API Gateway):**
  - Agisce come unico punto di ingresso per tutte le comunicazioni provenienti dal mondo esterno (il WebSocket Handler della GUI).
  - Nella sua fase di inizializzazione, configura un meccanismo di **delega** (`delegate`): istruisce l'infrastruttura Qak a inoltrare automaticamente tutte le future richieste di tipo `loadrequest` all'attore `gui_request_handler`.
  - Dopo la configurazione, rimane in uno stato passivo, agendo da puro router di messaggi.
- **gui\_state\_observer (Osservatore dello Stato):**
  - La sua unica responsabilità è mantenere la GUI aggiornata.
  - In fase di inizializzazione, si sottoscrive come "osservatore" (`observeResource`) dell'attore `cargoservice`.
  - Rimane in attesa di notifiche di aggiornamento. Quando `cargoservice` pubblica un nuovo stato della stiva, questo attore lo riceve e lo inoltra a tutti i client web connessi.
- **gui\_request\_handler (Gestore delle Richieste):**
  - La sua unica responsabilità è gestire il ciclo di richiesta/risposta per i comandi inviati dalla GUI.
  - Riceve le richieste `loadrequest` tramite la delega configurata dal Gateway.
  - Inoltra la richiesta al `cargoservice`.
  - Attende la risposta (`loadaccepted` o `loadrejected`) da `cargoservice`.

- Una volta ricevuta la risposta, la inoltra al client web originale che ha avviato la richiesta.

Questa architettura a tre attori, basata sul pattern **API Gateway** con worker specializzati, garantisce il massimo disaccoppiamento, una chiara separazione dei compiti e una notevole robustezza, poiché un eventuale malfunzionamento in un attore (es. nel gestore delle richieste) non influenzerà l'operatività degli altri (l'osservatore dello stato continuerà a funzionare).

## leddevice

`leddevice` deve controllare il led fisico, accendendolo, in caso di malfunzionamenti segnalati da `sonarDevice` e spegnendolo a fine segnalazione. Essendo, dunque, un componente reattivo e proattivo lo andremo a considerare come attore.

Il flusso di `leddevice` è il seguente:

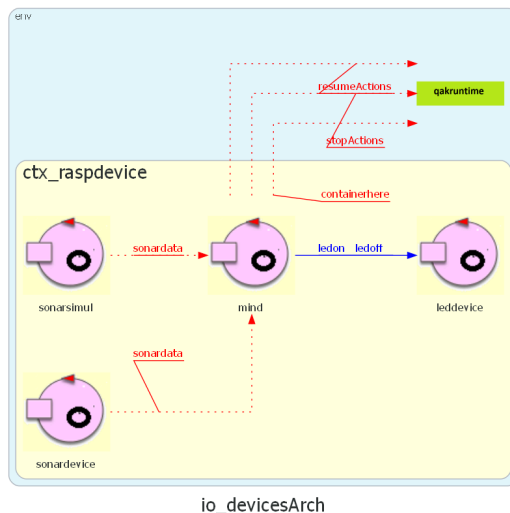
- in fase di inizializzazione passa direttamente all'attesa di messaggi dal sonar
- appena il sonar invia un messaggio di guasto( `ledon` ) accende il led fisico
- quando il sonar riceve un messaggio di spegnimento del led lo spegne ( `ledoff` )

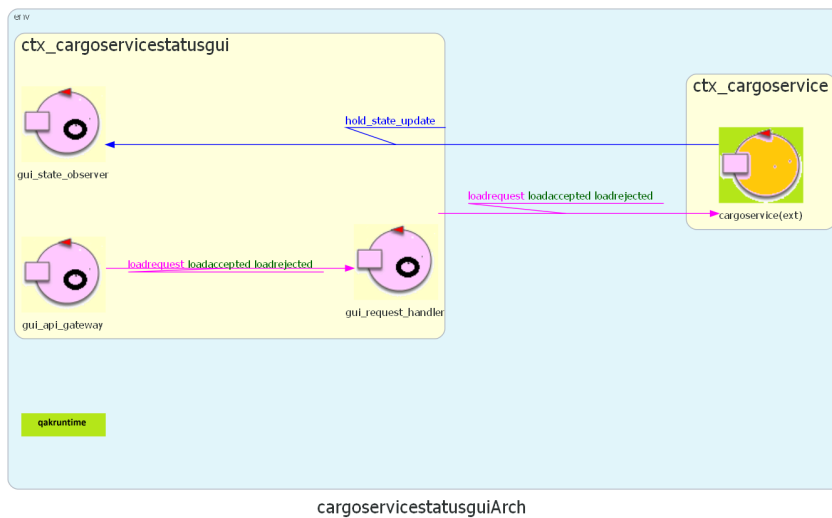
Si sono quindi modellati altri due tipi di messaggi. Si è optato per messaggi di tipo `dispatch`, in quanto `sonardevice` non ha bisogno di una risposta da parte di `leddevice`

```
//Sonardevice -> leddevice
Dispatch ledon : ledon(M)
Dispatch ledoff : ledoff(M)
```

## Modello

L'analisi confluisce nei seguenti due modelli logici





## Piano di testing

Avendo ora un formato definito per la visualizzazione dello **stato della stiva**, è possibile verificarne con maggiore precisione il corretto aggiornamento, oltre che il corretto invio del messaggio di update per la GUI.

A tal fine, sono stati ideati i seguenti **test plan**:

### Nuovi test aggiornamento stiva - [SlotManagementTest.java](#)

- Verifica rappresentazione della stiva in formato JSON (caso stiva vuota):

```

@Test
public void testEmptyHoldJsonRepresentation() {
    String stateJson = slotManagement.getHoldState(true);
    assertNotNull("Lo stato JSON non deve essere null", stateJson);

    try {
        JSONObject root = (JSONObject) new JSONParser().parse(stateJson);

        // Verifica totalWeight
        assertEquals("Il peso totale deve essere 0", 0, ((Long) root.get("totalWeight")).intValue());

        // Verifica slots array
        JSONArray slots = (JSONArray) root.get("slots");
        assertEquals("Devono esserci esattamente 4 slot", 4, slots.size());

        // Ogni slot deve avere product = null
        for (Object obj : slots) {
            JSONObject slotObj = (JSONObject) obj;
            assertTrue("Ogni slot deve avere un nome valido", ((String) slotObj.get("slotName")).startsWith("Slot"));
            assertNull("Ogni slot deve essere vuoto (product = null)", slotObj.get("product"));
        }

    } catch (ParseException e) {
        fail("Formato JSON non valido: " + e.getMessage());
    }
}

```

- Verifica rappresentazione della stiva in formato JSON (caso dopo un update):

```

@Test
public void testJsonStateAfterUpdate() {
    Product prod = new Product(5, "SpecialItem", 42);

    try {
        slotManagement.updateHold(prod, "Slot1");
    } catch (Exception e) {
        fail("Unexpected exception: " + e.getMessage());
    }

    String stateJson = slotManagement.getHoldState(true);
    assertNotNull("Lo stato JSON non deve essere null", stateJson);

    try {
        JSONObject root = (JSONObject) new JSONParser().parse(stateJson);

        // Verifica peso totale aggiornato
        assertEquals("Il peso totale deve riflettere il prodotto inserito",
            42, ((Long) root.get("totalWeight")).intValue());

        // Verifica che Slot1 contenga il prodotto corretto
        JSONArray slots = (JSONArray) root.get("slots");
        JSONObject slot1 = null;
        for (Object obj : slots) {
            JSONObject slotObj = (JSONObject) obj;
            if ("Slot1".equals(slotObj.get("slotName"))) {
                slot1 = slotObj;
                break;
            }
        }
        assertNotNull("Slot1 deve esistere", slot1);

        JSONObject product = (JSONObject) slot1.get("product");
        assertNotNull("Slot1 deve contenere un prodotto", product);
        assertEquals(5, ((Long) product.get("productId")).intValue());
        assertEquals("SpecialItem", product.get("name"));
        assertEquals(42, ((Long) product.get("weight")).intValue());

        // Verifica che gli altri slot siano vuoti
        for (Object obj : slots) {
            JSONObject slotObj = (JSONObject) obj;
            if (!"Slot1".equals(slotObj.get("slotName"))) {
                assertNull(slotObj.get("slotName") + " deve essere vuoto", slotObj.get("product"));
            }
        }

    } catch (ParseException e) {
        fail("Formato JSON non valido: " + e.getMessage());
    }
}

```

## Test messaggio di update con CoAP - CoapUpdateTest.Java

- Verifica della ricezione del messaggio:

@Test

```
public void testUpdateResourceCoap() throws Exception {

    CoapClient client = new CoapClient("coap://localhost:8000/ctx_cargoservice/cargoservice");

    CountDownLatch latch = new CountDownLatch(1); //usato per impostare un timeout

    Logger califLogger = (Logger) LoggerFactory.getLogger("org.eclipse.californium");
    califLogger.setLevel(Level.INFO);

    CoapObserveRelation relation = client.observe(new CoapHandler() {
        @Override
        public void onLoad(CoapResponse resp) {
            System.out.println("response: "+resp);
            String c = resp.getResponseText();
            System.out.println("content: "+c);
            if (c != null && !c.isBlank() && !"nonews".equalsIgnoreCase(c)) {
                content=c;
                latch.countDown();
            }
        }

        @Override
        public void onError() {
            latch.countDown();
        }
    });

    //mock di una loadrequest
    String req = CommUtils.buildRequest("mock", "loadrequest", "loadrequest(1)", "cargoservice").toString();

    String response = conn.request(req);
    System.out.println("richiesta inviata: "+response);

    if (!response.contains("loadaccepted"))
        fail("unexpected rejection");

    //mock dell'evento containerhere per poter testare senza preoccuparsi del sonar
    IApplMessage ev = CommUtils.buildEvent(
        "test",           // Nome mittente
        "containerhere", // Nome evento
        "containerhere(ok)" // Contenuto
    );

    conn.forward(ev);

    // Attendo la notifica CoAP con timeout per sicurezza
    boolean updated = latch.await(600, TimeUnit.SECONDS);
    relation.proactiveCancel();

    assertTrue("Nessun update CoAP ricevuto entro il timeout", updated);
    assertNotNull("Ricevuto update nullo", content);
    System.out.println("Update ricevuto: " + content);
}
```

# Progettazione

## Leddevice

Grazie all'utilizzo del framework QAK l'implementazione di `leddevice` è stata banale: all'arrivo dei dispatch `ledon` e `ledoff`, modellati in fase di analisi, l'attore fa partire il file python datoci dal committente per l'accensione/spegnimento del led fisico e transita nello stato successivo in attesa del messaggio complementare.

```
QActor leddevice context ctx_raspdevice{
    State init initial{
        println("$name | led ready") color yellow
    }Goto ledoff_state

    State ledoff_state{
        [# Runtime.getRuntime().exec("python ./resources/python/ledPython250ff.py") #]
        println("$name | led is off") color yellow
        println("$name | led waiting for messages") color yellow
    }Transition t0 whenMsg ledon -> ledon_state

    State ledon_state{
        [# machineExec("python ./resources/python/ledPython250n.py") #]
        println("$name | led is on") color yellow
    }Transition t0 whenMsg ledoff -> ledoff_state

}
```

## CargoserviceStatusGui

Il sottosistema QAK `cargoservicestatusgui` è il backend “logico” della GUI e vive in un contesto separato dal `cargoservicecore`. È composto da tre attori indipendenti, come stabilito in fase di modellazione:

- `gui_api_gateway` : Punto di ingresso per le comunicazioni provenienti dal mondo esterno. All’avvio delega tutte le `loadrequest` al worker specializzato. Si è deciso di definire una nuova richiesta `client_loadrequest` per differenziare le `loadrequest` ricevute tramite la GUI da quelle provenienti da altre fonti, e per poter passare l'id della sessione websocket come spiegato in seguito.

```
State s0 initial {
    println("$name | Gateway avviato.")
    delay 100
    delegate client_loadrequest to gui_request_handler
} Goto idle_state
```

- `gui_state_observer` : Osservatore. Si sottoscrive via CoAP allo stato del `cargoservice` e inoltra gli update al livello applicativo.

```
State s0 initial {
    println("$name | Avvio e inizio ad osservare cargoservice...") color green

    // Questa azione usa CoAP per sottoscrivarsi.
    // La notifica asincrona viene gestita da un CoapHandler
    // che invia i dati al WebSocketManager.
    // Per semplicità nel modello, l'implementazione esatta
    // dell'handler è delegata a una classe Java helper.
    observeResource cargoservice msgid hold_state_update
}

// La logica di questo attore si è spostata nell'handler CoAP
// e nel WebSocketHandler. Qui rimane solo il setup.
```



- `gui_request_handler` : Attore gestore delle richieste/risposte. Riceve `client_loadrequest(PID, SESSION_ID)` delegata da `gui_api_gateway` e inoltra `loadrequest(PID)` a `cargoservice`. Attende `loadaccepted(SLOT)` oppure `loadrejected(REASON)`, poi costruisce un JSON di risposta e lo inoltra a Spring come `load_response : response($Last_Request_ID, $ResponseJson)` .

## Cargoservicestatusgui\_model

Questo componente è un'applicazione Spring Boot che funge da ponte tra i browser (che comunicano via WebSocket) e i QAK (che comunicano via TCP + CoAP). Questo modulo è implementato con Spring, sfruttando Inversion of Control (IoC) e Dependency Injection (DI) per ottenere componenti debolmente accoppiati, riusabili e facilmente testabili. In particolare:

- il container Spring crea e gestisce il ciclo di vita dei bean;
- le dipendenze vengono iniettate (di norma via constructor injection), evitando new sparsi nel codice;
- le annotazioni (`@Component`, `@Service`, `@Configuration`, `@PostConstruct`, `@Value/@ConfigurationProperties`, ecc.) rendono il wiring esplicito e conciso.

La web GUI vera e propria è una single-page statica (HTML + CSS + JS vanilla) che fornisce una vista in tempo reale dello stato della stiva e consente l'invio di `loadrequest`.

- Stato in tempo reale: connessione WebSocket a `ws://localhost:8080/status-updates` con auto-reconnect (3s). Un indicatore rosso/verde mostra lo stato della connessione.
- Vista "hold": griglia responsiva degli slot aggiornata in tempo reale.
- Stato della richiesta: la risposta alle `loadrequest` inviate tramite GUI viene visualizzata temporaneamente sopra la griglia degli slot, e sparisce automaticamente dopo un breve timeout per evitare confusione con le risposte a richieste successive.

## Invio delle richieste tramite web GUI

Per quanto riguarda la funzionalità aggiuntiva introdotta in questo sprint, ovvero il poter inviare richieste di carico direttamente dalla GUI, essa è stata implementata tramite un ponte WebSocket → QAK realizzato in Spring Boot e l'attore QAK `gui_request_handler` introdotto in precedenza.

Alla pagina della GUI è stato aggiunto un form in cui inserire il PID desiderato e un pulsante Submit per inviare la richiesta.

Flusso dei messaggi:

Lato Spring (`cargoservicestatusgui_model`), il browser invia su WebSocket un messaggio contenente un tipo (in questo caso, `loadrequest`) e il PID inserito nell'apposito form. Il server recupera l'id della sessione WebSocket e lo passa al componente [ClientCaller](#), che a sua volta apre una connessione TCP verso il contesto QAK in cui si trovano i tre attori responsabili della gui e vi inoltra una `client_loadrequest(PID, SESSION_ID)`:

```

@PostConstruct
public void setup() {
    try {
        CommUtils.outblue("ClientCaller | Connecting to QAK context...");
        qakConnection = ConnectionFactory.createClientSupport(
            ProtocolType.tcp, guiContextHost, String.valueOf(guiContextPort));
        CommUtils.outgreen("ClientCaller | Connected to QAK context at " + guiContextHost + ":" + guiContextPort);
    } catch (Exception e) {
        CommUtils.outred("ClientCaller | Connection to QAK context FAILED: " + e.getMessage());
    }
}

public void sendLoadRequest(int pid, String sessionId) {
    if (qakConnection == null) {
        CommUtils.outred("ClientCaller | Cannot send request, no connection to QAK context.");
        return;
    }
    try {
        String msgId = "client_loadrequest";

        // Includiamo il sessionId nel payload, racchiudendolo tra apici singoli per la sintassi Prolog
        String payload = String.format("client_loadrequest(%d,'%s')", pid, sessionId);

        IApplMessage request = CommUtils.buildRequest(
            "websocket_client", // sender
            msgId,              // msgId
            payload,             // content
            gatewayActorName    // receiver
        );

        CommUtils.outblue("ClientCaller | Sending request to QAK: " + request);
        qakConnection.forward(request);

    } catch (Exception e) {
        CommUtils.outred("ClientCaller | Error sending request: " + e.getMessage());
    }
}

```

Lato QAK (cargoservicestatusgui), l'attore `gui_api_gateway` riceve la `client_loadrequest` e ne delega la gestione a `gui_request_handler`, che:

- estrae PID e racchiude tra apici il `SESSION_ID` per evitare errori del motore prolog,
- invia la loadrequest a cargoservice con `request cargoservice -m loadrequest : loadrequest(PID)`,
- una volta ricevuta una risposta `loadaccepted(SLOT)` o `loadrejected(REASON)` costruisce un JSON e lo inoltra a Spring con:  
`forward springboot_gui -m load_response : response(SESSION_ID, JSON_STRING)`

Ritorno a Spring: il server TCP su 8002 ([QakResponseServer](#)) riceve `load_response`, ne effettua il parsing e invia il JSON grezzo alla specifica sessione WebSocket indicata dal `SESSION_ID`.

Dopodiché il browser mostra una notifica del risultato della richiesta.

In questo modo, la GUI può eseguire l'azione end-to-end senza conoscere dettagli interni del dominio QAK.

## Deployment

1. Andare nella cartella [CargoServiceCore](#)
2. Seguire le [istruzioni](#) per caricare l'immagine Docker di cargoservicecore

3. Eseguire il comando `docker load -i basicrobot24.tar` per caricare l'immagine Docker del basicrobot
4. Creare la rete `docker network create iss-network`
5. Eseguire il comando `docker compose -f arch3.yaml up` per far partire i componenti del sistema
6. Aprire il browser su [localhost:8090](http://localhost:8090) per visualizzare l'ambiente WEnv in cui lavorerà il DDR robot
7. Eseguire il comando `./gradlew run` oppure `gradle run` nella cartella [logicModel\\_IODevices](#) per far partire il resto del sistema  
RaspDevice

*Note:*

- a. Per far eseguire il punto 2 è bene ricordarsi di far partire il demone Docker
- b. Il sistema cargoservice si appoggia a productservice che ha un database Mongo per la persistenza dei prodotti, questo si può riempire con opportuni prodotti di test attraverso il file [setup\\_mongo.js](#) (eseguire `node setup_mongo.js` )

## Raspberry Deployment

Se si è in possesso di un Raspberry Pi, si possono usare componenti fisici per il controllo dei dispositivi di I/O.

Per farlo:

1. Eseguire fino al punto 6 della sezione precedente
2. Copiare sul Raspberry Pi la distribuzione dei componenti relativi ai dispositivi di I/O generata con `./gradlew run` utilizzando, ad esempio, il comando `scp` oppure clonando direttamente il repository da Git.
3. Sul Raspberry Pi, assicurarsi di avere installato Java 17 e Python 3:  
`sudo apt update && sudo apt install -y openjdk-17-jdk python3 python3-pip`
4. Verificare che lo script [sonar.py](#) sia leggibile ed eseguibile: `chmod a+rx /percorso/del/progetto/resources/python/sonar.py`
5. Assicurarsi che il file gradlew abbia il permesso di esecuzione (necessario se il progetto è stato copiato da Windows o scaricato in un formato che perde i permessi): `chmod +x gradlew`
6. Lanciare il sistema direttamente sul Raspberry Pi con: `./gradlew run`