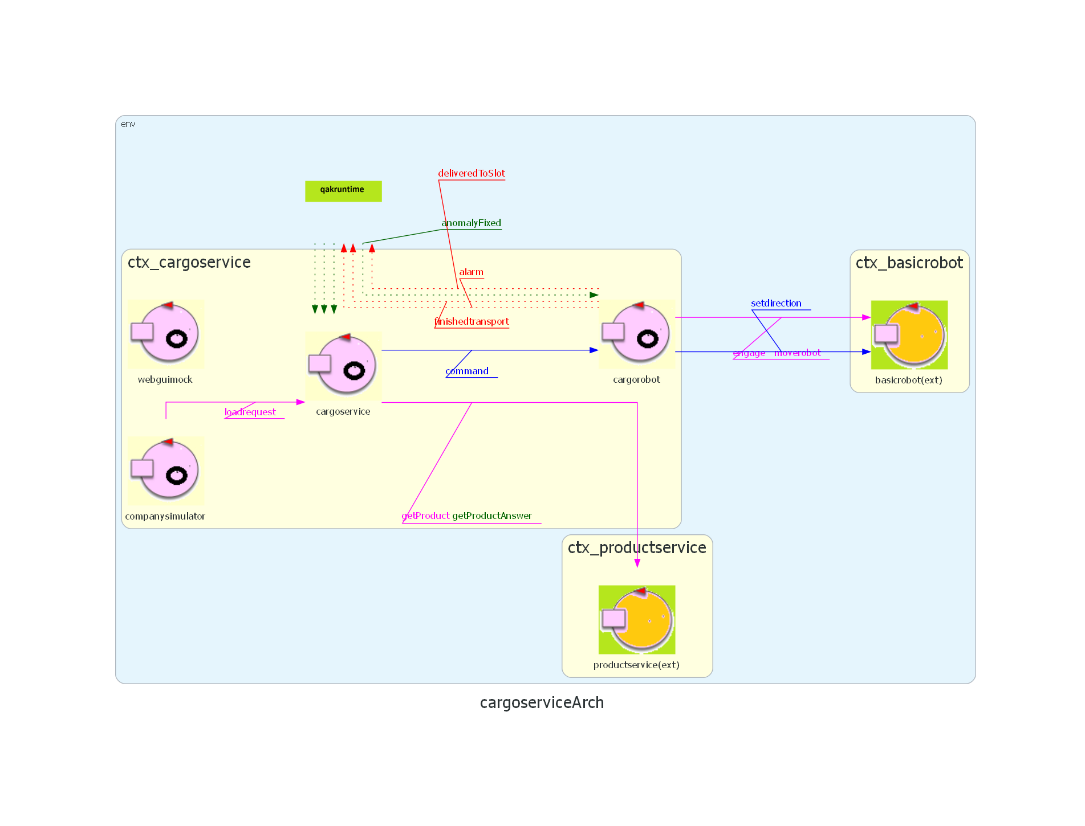
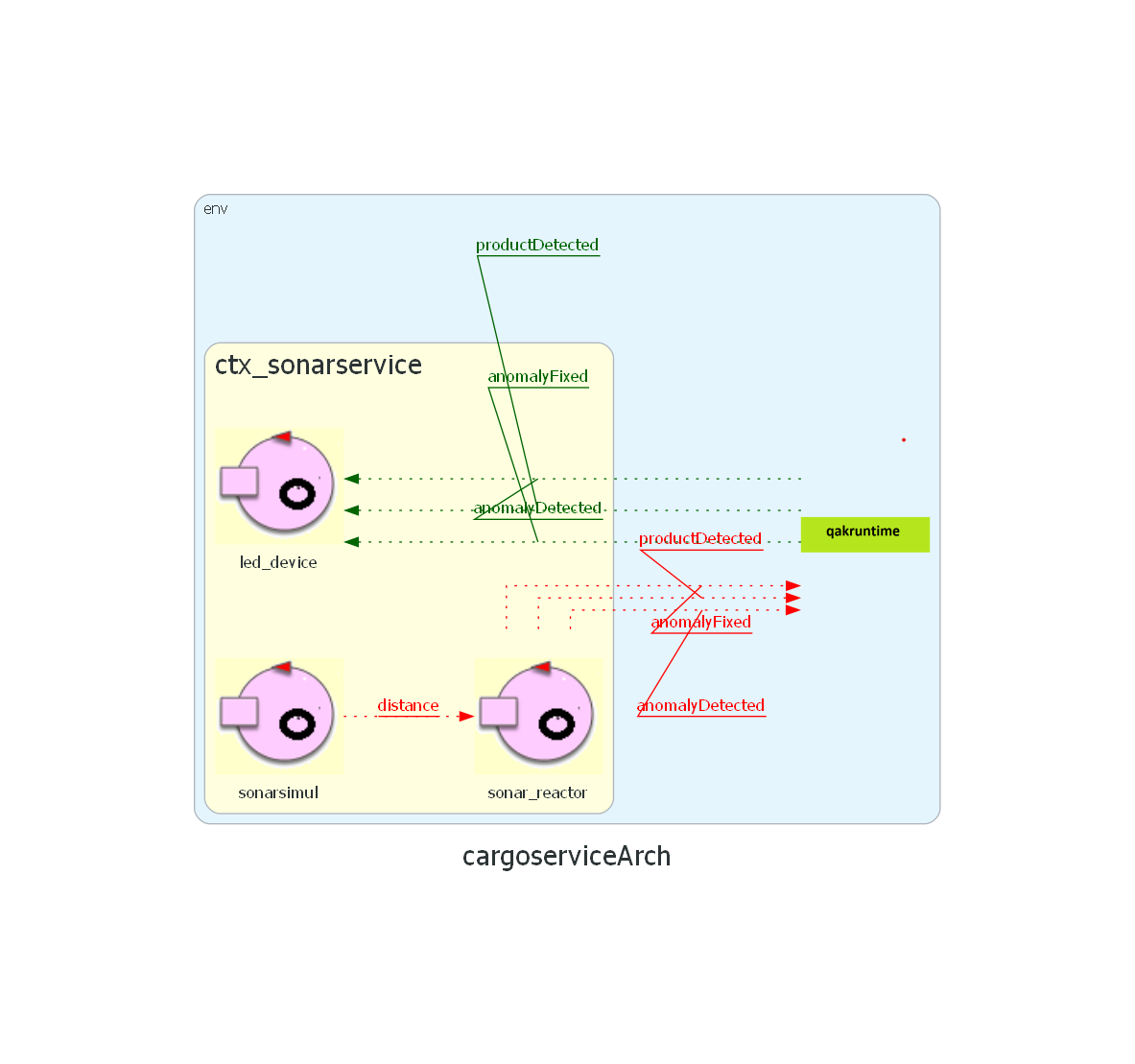
**SPRINT 3:**

**Punto di Partenza:**

****Nello sprint2 il team ha implementato il componente sonarservice, sostituendolo al mock object corrispondente creato durante lo sprint1, ottenendo la seguente architettura:

Si vuole fornire per comprensione un vocabolario riassuntivo di termini definiti nei precedenti sprint: **Vocabolario:**

|  |  |
| --- | --- |
| **Termine** | **Significato attribuito** |
| Container | Contenitore in cui viene inserito il prodotto |
| loadrequest /richiesta di carico | Richiesta mandata dalla compagnia, specificando un PID |
| prodotto | Nel sistema è l’oggetto trasportato dal robot, la cui presenza può innescare diversi eventi |
| prodotto registrato | Prodotto conosciuto da ProductService a cui è associato un PID e un peso(Weight) |
| Microservizio | Componente software progettato per svolgere una specifica funzione del sistema. Ogni microservizio comunica con gli altri tramite messaggi, rendendo il sistema flessibile e scalabile. |
| GUI (Graphical User Interface) /WebGUI | Interfaccia grafica utente accessibile via web, che consente di visualizzare in tempo reale lo stato della stiva e interagire in modo intuitivo con il sistema. |
| Bounded Context | Il "bounded context" (**contesto limitato**) è un concetto fondamentale nel Domain-Driven Design (DDD) e si riferisce a un ambito applicativo ben definito e autonomo all'interno del quale vengono definite entità, regole e logiche di business in modo univoco e chiaro. All'interno di un bounded context, il significato di ogni entità o concetto è inequivocabile e specifico per quel contesto, evitando ambiguità e conflitti con altri contesti. |
| IOPort | Punto fisico (porta) attraverso il quale i contenitori dei prodotti entrano o escono dalla nave. È il punto in cui il sonar rileva la presenza di un prodotto. |
| Sonar | Sensore a ultrasuoni che misura la distanza tra sé e un oggetto. Nel nostro sistema serve per rilevare se un contenitore è presente all’IOPort. |
| DDRobot | è un robot che utilizza due motori indipendenti per muovere le ruote o i cingoli. È il supporto fisico che viene comandato da cargorobot. |
| PID (Product Identifier) | Numero intero univoco assegnato a ciascun prodotto registrato, usato per tracciarne l'identità all'interno del sistema. |
| Slot | Spazio fisico nella stiva della nave dove può essere posizionato un contenitore. Esistono 4 slot disponibili; uno è sempre occupato (slot5). |
| Cargorobot | Robot mobile autonomo (a guida differenziale) incaricato di trasportare i contenitori dall’IOPort fino allo slot assegnato e poi tornare alla posizione HOME. |
| Stiva | Area rettangolare della nave in cui i contenitori vengono caricati. Contiene gli slot e l’IOPort. |
| ProductService | Microservizio che gestisce la registrazione dei prodotti. Verifica i dati e assegna un PID univoco |
| CargoService | Microservizio che riceve richieste di carico, controlla i vincoli, assegna gli slot e coordina il caricamento tramite cargorobot. |
| SonarService | Microservizio che rileva la presenza di un contenitore all’IOPort tramite i dati forniti dal sonar. |
| DFREE | Distanza soglia usata dal sonar: se la distanza misurata è maggiore di DFREE per 3 secondi, si ipotizza un malfunzionamento del sensore. |
| MaxLoad | Peso massimo complessivo che la nave può sopportare. Il sistema rifiuta richieste che farebbero superare questo limite. |
| Worker | Persona che colloca fisicamente i contenitori sull’IOPort dopo che sono stati registrati. |
| Sistema logico di riferimento | Rappresentazione concettuale dell’intero sistema, con attori, componenti e interazioni, usata come base per l’architettura e la progettazione tecnica. |
| attore | Entità che svolge un ruolo attivo nel sistema, eseguendo azioni e comunicando con gli altri attori attraverso messaggi |
| Linguaggio QAK | Linguaggio modellistico usato per descrivere e simulare il comportamento dei componenti del sistema come “attori” |
| POJO | Plain Old Java Object: un oggetto di una classe in java |
| Anomalia | Nel documento è inteso come un comportamento inatteso di un componente hardware, tale da compromettere il normale funzionamento del sistema. |
| Refactoring |  |
| Framework |  |
| CoAP |  |
| Mqtt |  |
| TCP |  |
| UDP |  |
| Http |  |
| Rest |  |

**Goal dello Sprint3:**

* Enunciazione esplicita dei requisiti della webgui
* Analisi del problema
* Definizione dell’architettura logica con modello eseguibile
* Progetto e realizzazione della webgui
* Progettazione per l’invio delle richieste
* Deployment

**Enunciazione esplicita dei requisiti della webgui:**

Il sistema deve fornire un’interfaccia grafica accessibile via web (**webgui**) che consenta di monitorare lo stato della stiva (**hold**) in tempo reale. In particolare:

**RF1. Visualizzazione dello stato della stiva**

* + L’interfaccia deve mostrare lo stato corrente di ciascun **slot** (occupato/libero).
  + Deve essere mostrato anche il **peso complessivo** del carico presente nella stiva.

**RF2. Aggiornamento dinamico**

* + La WebGUI deve aggiornarsi automaticamente al variare dello stato della stiva, senza necessità di ricaricare manualmente la pagina.
  + L’aggiornamento avviene tramite notifiche push, ottenute osservando la risorsa CoAP hold e inoltrate ai client via WebSocket.

**RF3. Accessibilità via Web**

* + La WebGUI deve essere consultabile da un browser attraverso un endpoint HTTP dedicato.
  + Deve essere disponibile almeno una pagina HTML responsiva che consenta all’utente di monitorare lo stato della stiva in maniera chiara e intuitiva.

**Analisi del problema:**

L’obiettivo principale di questo sprint è la realizzazione della **WebGUI**, che consentirà agli utenti di monitorare in tempo reale lo stato della stiva. Per raggiungere questo scopo, è emersa la necessità di centralizzare le informazioni relative alla stiva e al peso del carico in un unico punto di riferimento.

Durante lo sprint1 il team aveva assegnato la responsabilità della gestione dello stato degli slot e del calcolo del peso totale del carico direttamente al *cargoservice*, ma mantenere tale struttura con l’aggiunta della webgui avrebbe reso il sistema estremamente complesso, difficile da mantenere e da far evolvere. La sua funzione principale è quella di coordinare le operazioni, non gestire i dati.

Pertanto, si è deciso di fare un **refactoring** e creare un nuovo componente specifico: il modello **hold**.

**Il modello Hold: Ruolo e responsabilità**

Questo componente agisce come punto di riferimento centralizzato per lo stato della stiva. La sua responsabilità principale è mantenere aggiornate le seguenti informazioni:

* il **peso complessivo** degli elementi presenti nella stiva
* lo **stato (occupato/libero)** di ogni slot.

Questo design riflette il principio di separazione delle responsabilità, semplificando gli altri microservizi e rendendo le informazioni sulla stiva facilmente accessibili, migliorando la robustezza e la scalabilità del sistema.

**Interazione tra Hold e Cargoservice:**

Con il **refactoring**, il *CargoService* non gestisce più i dati della stiva, ma delega a *hold* la responsabilità di valutare le richieste di carico. Questa collaborazione si basa su un preciso scambio di messaggi:

1. **Richiesta di valutazione:** *CargoService* invia a *Hold* una richiesta per verificare se è possibile caricare un prodotto, utilizzando il messaggio *checkIfFits.*

|  |
| --- |
| ***Request checkIfFits : checkIfFits(PID, Weight)*** |

1. **Risposta di Hold:** *Hold* valuta la richiesta in base al peso del prodotto e alla disponibilità degli slot.

o La richiesta viene rifiutata se il peso supera il valore di MaxLoad o se la stiva risulta piena (assenza di slot disponibili).

o In caso contrario, viene assegnato uno slot al prodotto.

|  |  |
| --- | --- |
| ***Reply accepted: accepted(JsonString) for checkIfFits*** | La richiesta di carico viene accettata e viene mandata un oggetto Json in formato stringa, che contiene le informazioni riguardanti il prodotto e lo slot in cui deve essere posizionato |
| ***Reply refused: refused(Reason) for checkIfFits*** | La richiesta di carico viene rifiutata. |

**Hold e la WebGUI:**

*Hold* è fondamentale per il funzionamento in tempo reale della WebGUI. Per garantire che l’interfaccia utente sia sempre aggiornata, Hold invia periodicamente il proprio stato alla WebGUI.

Questa comunicazione avviene tramite un messaggio di tipo Dispatch, che contiene un oggetto JSON che descrive lo stato attuale della stiva (slot occupati/liberi e peso totale).

|  |
| --- |
| ***Dispatch update : update(HoldJsonString)*** |

Mentre la Hold rimane in attesa di nuove richieste, manda il messaggio di update.

Nota: si era pensato di mandare un aggiornamento solo ad ogni consegna avvenuta, ma la natura del canale di comunicazione, fallace, rischierebbe di far perdere messaggi e di rendere inaffidabile l’interfaccia, pertanto, si è optato fare più aggiornamenti.

In questo modo, la WebGUI riceve le informazioni e aggiorna l’interfaccia in modo dinamico, senza bisogno di ricaricare la pagina.

**Comunicazione con la WebGui tramite CoAP:**

Per la comunicazione tra Hold e la WebGUI, si è scelto **CoAP** (Constrained Application Protocol), uno dei protocolli più usati nell'Internet of Things. CoAP è basato su **REST** e utilizza il protocollo **UDP**, che, pur avendo una **alta probabilità di perdita dei messaggi**, consente una trasmissione dei dati rapida e adatta per aggiornamenti dinamici come quelli richiesti dalla WebGUI.

CoAP è un protocollo **request-response**:

* Il client invia richieste **GET** per osservare una risorsa.
* Il server risponde con notifiche che contengono il nuovo stato della risorsa, aggiornando così l’interfaccia senza la necessità di una comunicazione continua.

Questo lo rende ideale per scenari dove è fondamentale l'aggiornamento dinamico e tempestivo dei dati.

Difatti la Hold non manda il dispatch alla WebGui tramite TCP, ma lo fa tramite comunicazione CoAP:

|  |
| --- |
| ***Dispatch update : update(HoldJsonString)*** |

Ovvero:

|  |
| --- |
| **[#**  **hold = HoldData()**  **val HoldJsonString = hold.holdToJson()**  **#]**  **updateResource [#HoldJsonString#]** |

**Il formato per il trasferimento di informazioni:**

Per rendere più facile il trasferimento dei dati, si è optato per l’utilizzo del formato JSON (JavaScript Object Notation).

La *Hold* manda oggetti in formato JSON sotto forma di stringa. Tale formato è stato scelto come standard per lo scambio di dati perché è leggero e basato su testo, caratteristiche particolarmente adatte per l’architettura a microservizi.

Le principali caratteristiche sono:

* **Leggibilità**: la sua sintassi a coppie "chiave-valore" è intuitiva e facile da leggere sia per gli sviluppatori che per le macchine, semplificando il debug e la manutenzione.
* **Indipendenza dal linguaggio**: è un formato universale supportato nativamente dalla maggior parte dei linguaggi di programmazione moderni. Questo permette ai nostri microservizi, anche se scritti in linguaggi diversi, di comunicare senza problemi.
* **Efficienza**: La sua struttura concisa, che non richiede tag di chiusura come altri formati, si traduce in messaggi più piccoli e leggeri. Questo riduce la quantità di dati d1a trasmettere sulla rete, migliorando la velocità e l'efficienza della comunicazione tra i componenti.

**Modello dei dati** [**HoldData.java**](cargoservice/src/main/java/HoldData.java)**:**

Il cuore del microservizio hold è la classe **HoldData.java**. Progettata come un **POJO** (Plain Old Java Object), questa classe incapsula lo stato della stiva e implementa la logica di business necessaria per gestirne i dati. In questo modo, l'entità hold funge da fonte centralizzata e coerente per tutti i componenti che necessitano di informazioni sul carico.

La classe HoldData gestisce due attributi principali:

* **s**: un'istanza della classe Slots che gestisce lo stato di ogni slot fisico della stiva.
* **Cur\_HoldWeight**: un intero che rappresenta il peso totale del carico attualmente a bordo, aggiornato dinamicamente.

I metodi implementati in questa classe sono pensati per supportare le interazioni con il CargoService e la WebGUI:

|  |  |
| --- | --- |
| ***canLoad(int slotID, int productWeight)*** | esegue la **verifica dei vincoli** richiesta dal CargoService. Restituisce true solo se il peso del prodotto non supera il MaxLoad totale della nave e se lo slot specificato è valido, garantendo l'integrità del sistema |
| ***getAvailableSlot()*** | Fornisce il primo **Slot** disponibile per un nuovo carico |
| ***registerProductInSlot(Slot slot, int productWeight)*** | Aggiorna lo stato dello slot una volta che un prodotto è stato registrato e aggiorna il peso totale del carico (Cur\_HoldWeight). |
| ***checkResultToJson(Slot slot, int pid, int weight):*** | Converte il risultato di una verifica di carico in un **oggetto JSON**. Questo metodo crea un messaggio standardizzato che Hold invia al CargoService per comunicare i dettagli dello slot assegnato. |
| ***String holdToJson()*** | Genera un **oggetto JSON completo** che rappresenta lo stato attuale dell'intera stiva. Questo formato è ottimizzato per la visualizzazione sulla WebGUI, includendo il peso totale del carico e lo stato di ogni slot, e facilita l'aggiornamento dinamico dell'interfaccia. |

**La WebGUI:**

è il componente responsabile della presentazione dello stato della stiva (hold), che include il monitoraggio degli slot e del peso totale trasportato. In precedenza, un attore webgui\_mock era utilizzato per emulare questa funzionalità, con **cargoservice** responsabile della comunicazione. Dopo il refactoring, si è deciso che gli aggiornamenti arrivino direttamente dall’attore **Hold** descritto prima.

La WebGui funziona come segue:

* All’avvio comincia ad **osservare** il componente hold come risorsa CoAP e visualizza lo stato iniziale della stiva.
* All’arrivo di ciascun aggiornamento, modifica i valori dell’interfaccia e aggiorna lo stato della stiva dinamicamente.

Possiamo considerarlo un sistema **passivo**, dato che riceve soltanto informazioni.

**Progettazione della WebGUI:**

La webgui è stata progettata utilizzando SpringBoot, un framework che semplifica lo sviluppo di applicazioni web e microservizi in Java. Offre una serie di funzionalità integrate, come il supporto per la configurazione automatica, la gestione delle dipendenze e l’integrazione con vari sistemi di backend, il che permette di sviluppare rapidamente un’applicazione scalabile e facilmente mantenibile.

Le componenti che permettono la comunicazione della WebGUI con il cargoservice sono:

* **WsHandler**.**java**: gestore delle connessioni con il client browser.
* **CoapToWS.java**: il client CoAP che ottiene i messaggi.

[**WSHandler**](webgui/src/main/java/unibo/webgui/ws/WSHandler.java)**.java:**

è il componente di comunicazione WebSocket che si occupa delle connessioni con i client browser.

Gestisce le connessioni WebSocket in entrata, permettendo l’invio di messaggi a tutte le sessioni connesse.

**responsabilità:**

* **tiene traccia delle sessioni connesse**: ogni volta che un nuovo client si connette, viene aggiunto alla lista delle sessioni, e quando una sessione si disconnette, viene rimossa.
* **fornisce il metodo *sendToAll:*** il metodo *sendToAll* è utilizzato per inviare i messaggi ricevuti da CoAP a tutte le sessioni WebSocket attive, assicurando che tutte le WebGUI connesse ricevano gli aggiornamenti in tempo reale.

[**WebSocketConfig**](webgui/src/main/java/unibo/webgui/ws/WebSocketConfig.java).**java:**

Registra il WSHandler su un endpoint specifico, in questo caso “/status-updates”

[**CoapToWS**](file:///C:\Users\Sveva\Documents\GitHub\CargoService\sprint3\webgui\src\main\java\unibo\webgui\coap\CoapToWS.java)**.java:**

Fa da ponte tra il protocollo CoAP e WebSocket.

Si sottoscrive come osservatore dello stato di hold accessibile. Ogni volta che hold viene aggiornata, la WebSocket riceverà un messaggio.

**responsabilità:**

* **monitora il cambiamento dello stato della stiva tramite CoAP (observeResource):** quando il client riceve una risposta dal server CoAP, il contenuto viene parsato (convertito in un oggetto JSON) e inviato a tutte le sessioni WebSocket attive tramite il componente WSHandler.

La comunicazione tra CoAP e WebSocket consente alla WebGUI di ricevere aggiornamenti in tempo reale senza necessità di ricaricare la pagina.

**Parsing**:

I messaggi che arrivano alla WebSocket tramite Coap sono in forma JSON.

La classe [HoldResponseParse](webgui/src/main/java/unibo/webgui/utils/HoldResponseParser.java)r.java fornisce un metodo **parseHoldState(String message)** che interpreta i messaggi ricevuti e li restituisce in formati interpretabili.

**Interfaccia web:**

Per il monitoraggio della stiva è stata sviluppata un’interfaccia **web responsiva** basata su HTML, CSS e JavaScript.  
La pagina fornisce una rappresentazione chiara e immediata dello stato del sistema, mostrando:

* il **peso totale del carico a bordo**, aggiornato in tempo reale;
* lo **stato dei quattro slot del deposito**, distinti come liberi o occupati;
* un **collegamento automatico con l’endpoint /status-updates**, che permette di ricevere notifiche push sugli eventi;
* un **meccanismo di refresh dinamico** dell’interfaccia al verificarsi di variazioni, senza dover ricaricare la pagina.

Al primo accesso, il browser si connette alla websocket, in modo da poter iniziare a ricevere gli aggiornamenti da parte del server.



**Progettazione per l’invio delle richieste:**

[**CompanyRequestReceiver**](cargoservice/src/cargoservice.qak)**:**

Come visto nello sprint0, il sistema deve essere in grado di ricevere le richieste di carico (load request) da parte della compagnia.

Esso è responsabile della gestione delle richieste di carico provenienti dalla WebGUI.   
Riceve le richieste della compagnia e ne coordina l’inoltro verso il CargoService, garantendo al tempo stesso una comunicazione immediata con la GUI.

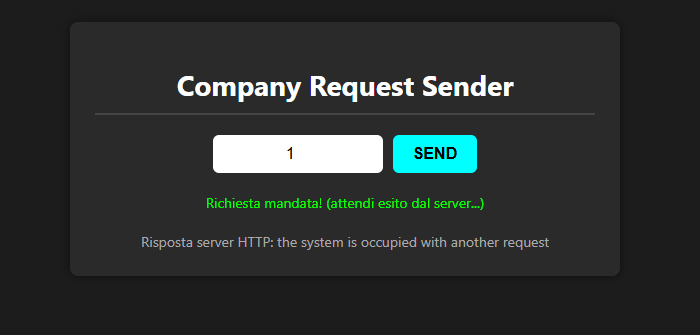
In particolare:

* attende le richieste di carico inviate dalla compagnia;
* verifica la disponibilità del sistema: se libero, accetta la richiesta, altrimenti la rifiuta immediatamente;
* inoltra le richieste accettate al **CargoService** e rimane in attesa dell’esito;
* invia una risposta immediata alla GUI, per confermare o meno la presa in carico della richiesta;
* aggiorna successivamente la GUI, tramite notifiche CoAP, comunicando l’esito finale dell’operazione.

Questo attore sostituisce il simulatore di richieste utilizzato nello **Sprint 0** e rappresenta il punto di ingresso reale delle richieste di carico nel sistema, garantendo sia la gestione esclusiva delle richieste sia un aggiornamento costante dello stato verso l’interfaccia utente.

**RequestSender:**

Per facilitare sia il debug, che l’utente stesso, è stata creata un’interfaccia grafica anche per le richieste.



L’utente può inserire solo valori numerici e cliccare invio.

L’interfaccia segnala se la richiesta è stata mandata correttamente al sistema cargoservice o se quest’ultimo è già occupato e pertanto non può prenderne in carico un’altra.

Segnala se la richiesta è stata completata con un successo o con un fallimento.

[**CallerService.java:**](webgui/src/main/java/unibo/webgui/service/CallerService.java)

Quando l’utente clicca *SEND*, cerca in realtà di mandare la richiesta di carico al sistema cargoservice.

Pertanto, è necessario un punto di accesso che permetta alla WebGUI a mandare richieste tramite TCP.

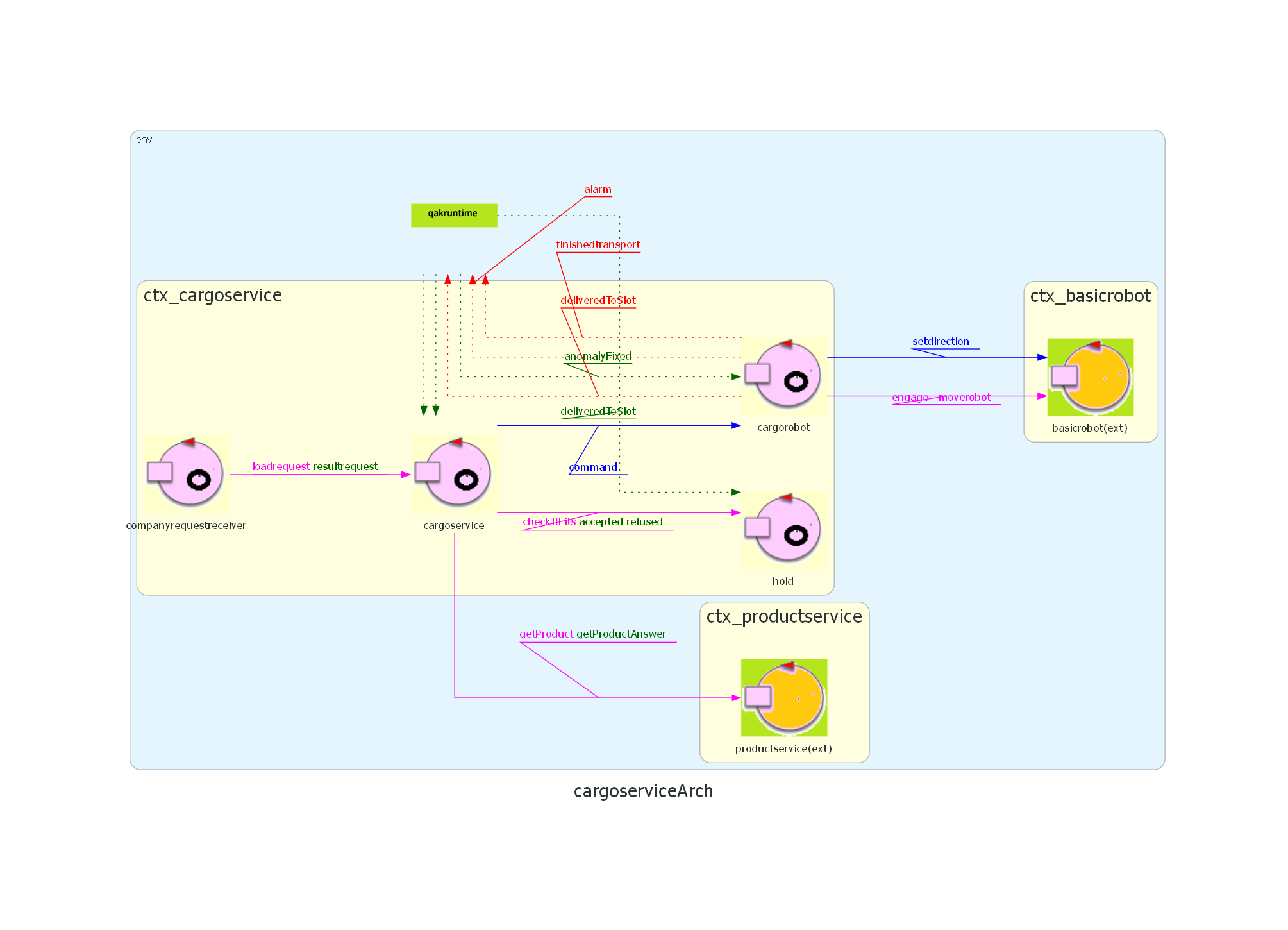
CallerService gestisce quindi le richieste HTTP che provengono dalla WebGUI.

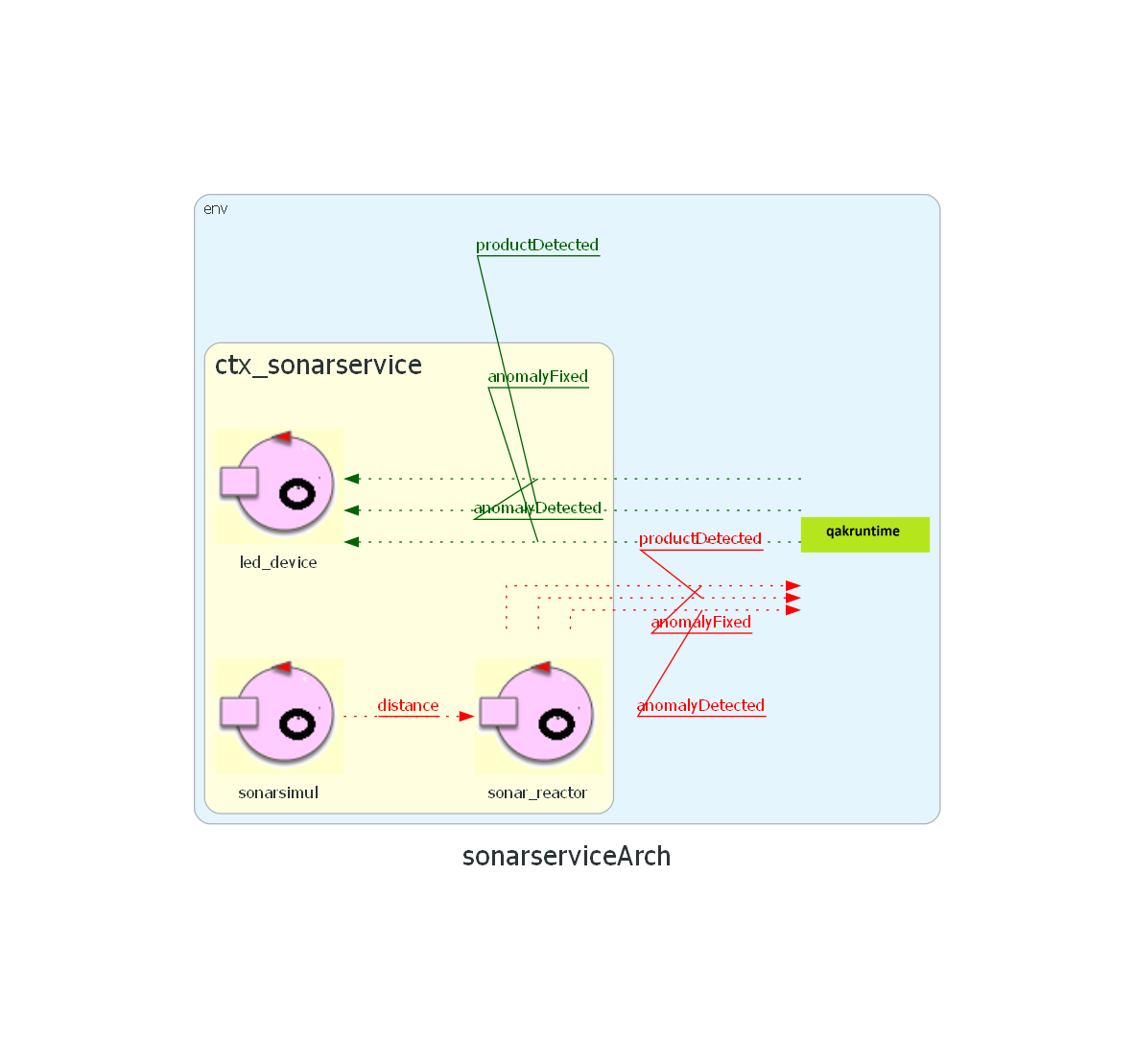
Quando arriva una richiesta dall’interfaccia (l’utente ha cliccato il pulsante *SEND*), il controller HTTP invoca un metodo che invia una richiesta al sistema *cargoservice* tramite una connessione TCP.

Una volta giunta al sistema, l’attore CompanyRequestReceiver se ne occupa.

**Test:**

**Definizione dell’architettura logica con modello eseguibile:**

****

****

**Deployment:**

Si intende