Sprint 3

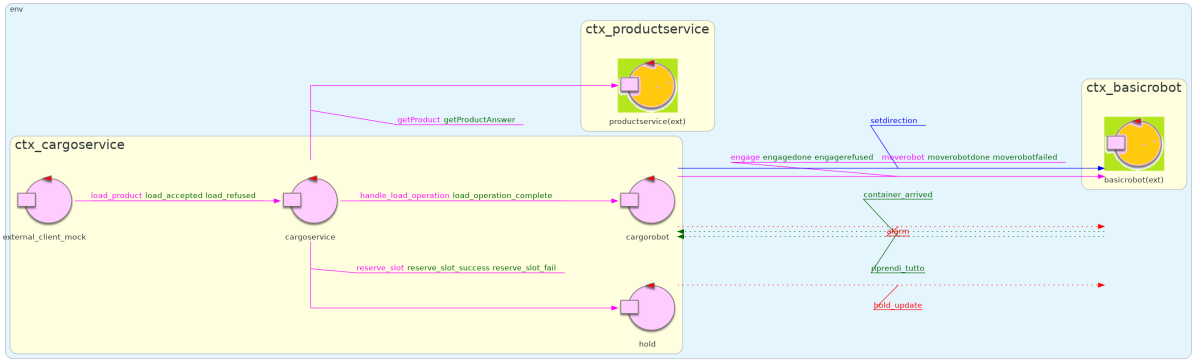
Indice

Punto di Partenza

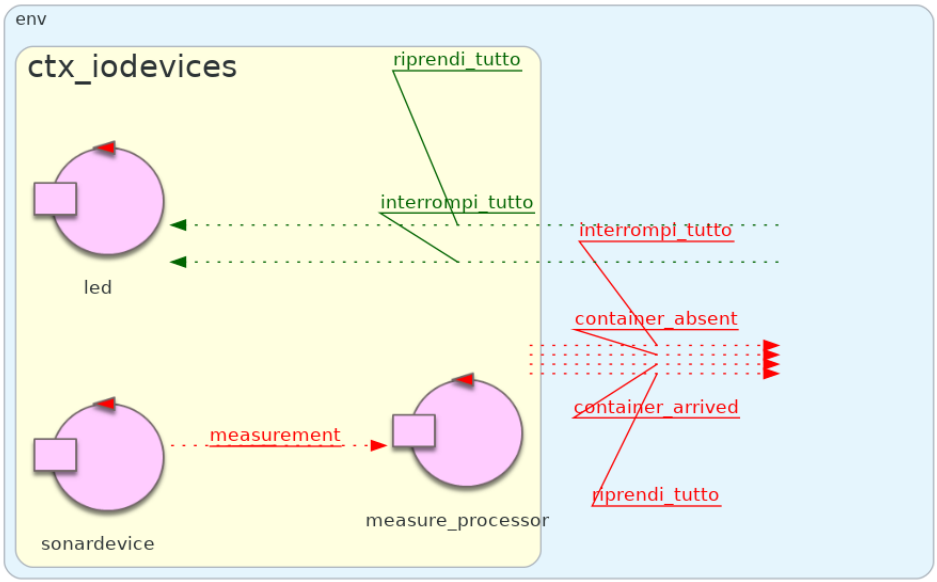
Nello sprint precedente si sono implementati i componenti: sonar e hold. Grazie al primo, è diventato possibile rilevare la presenza/assenza dei container, grazie al secondo è diventato possibile gestire lo stato del deposito completando in questa maniera la logica del sistema.

Durante l'analisi del componente Hold si sono anche definiti i messaggi che quest'ultimo dovrà scambiarsi con la web-gui, componente che si implementerà nello sprint 3.

L'architettura del sistema risultante da questo sprint è suddivisibile in due macrocontesti.



arch2Arch



iodevicesArch

Obiettivi

L'obiettivo dello sprint 3 sarà affrontare il sottoinsieme dei requisiti relativi ai componenti *webgui* e *led*, effettuando l'analisi del problema e la progettazione. Particolare importanza verrà data alle **interazioni** che questi componenti dovranno avere con il resto del sistema.

I requisiti affrontati nello sprint 3 sono i seguenti:

- is able to show the current state of the hold, by means of a dynamically updated web-gui,
- interrupts any activity and turns on a led if the sonar sensor measures a distance D > DFREE for at least 3 secs (perhaps a sonar failure) and the service continues its activities as soon as the sonar measures a distance D <= DFREE.

Led

Analisi del Problema

L'attore *led* è responsabile di segnalare l'eventuale malfunzionamento del sonar.

L'analisi del problema di *led* è banale. Quando viene emesso un evento di guasto si accende il led fisico, e quando viene emesso l'evento di ripristino quest'ultimo viene spento.

L'unica cosa non ovvia è come comandare l'accensione/spegnimento del led fisico. Similemente al caso del sonar, questo si può fare con uno script python che comanda i GPIO pin di un raspberry pi.

Nota: Data la semplicità del componente, non è stato previsto un piano di test.

Modello

```
QActor led context ctx_iodevices {
    State s0 initial{
    }
    Goto spento

    State spento {
        println("$name | sono spento") color red
        [# machineExec("python ledPython25Off.py") #]
    }
    Transition t0
        whenEvent interrompi_tutto -> acceso

    State acceso {
        println("$name | sono acceso") color green
        [# machineExec("python ledPython25On.py") #]
    }
    Transition t0
        whenEvent riprendi_tutto -> spento
}
```

WebGui

Analisi del Problema

L'attore *webgui* è responsabile di presentare lo stato degli slot e il peso complessivo dei container nel deposito.

Si era già accennato nello sprint2 che lo stato del deposito mostrato dalla *webgui* viene mantenuto dal componente *hold*, ma recuperato passando per *cargoservice*. Sempre nello stesso, si erano anche delineati dei messaggi che *webgui* utilizzerà per recuperare lo stato del deposito e eventuali aggiornamenti di quest'ultimo(*get_hold_state* e *hold_update*).

Il tipico ciclo di attività di *webgui* è il seguente:

1. All'avvio, *webgui* richiede lo stato corrente del deposito inviando una richiesta a *cargoservice* e comincia ad **osservare** il componente *hold* come risorsa CoAP (vedi progettazione sprint2)
2. Ricevuta la risposta da *cargoservice*, *webgui* può cominciare a mostrare lo stato del deposito
3. Ogni volta che si verifica una modifica al deposito, con conseguente aggiornamento della risorsa CoAP *hold*, *webgui* mostra i cambiamenti.

Modello

```
QActor webgui context ctx_webgui {
    [#
        var CurrentState = "{}"

        fun stateUpdate(json: String){
            CurrentState = json
            println("Hold State: $CurrentState")
        }
    #]

    State init initial {
        delay 1000
        println("$name | START")

        // webgui osserva hold per aggiornamenti
        observeResource hold msgid hold_update

        println("$name | getting hold state for the first time")
        // chiedo a cargoservice che girerà ad hold
        request cargoservice -m get_hold_state : get_hold_state(si)
    }
    Transition t0
        whenReply hold_state -> handleHoldState


    State handleHoldState {
        println("$name | processing reply")

        onMsg( hold_state : hold_state(JSON) ) {
            [#
                val receivedState = payloadArg(0)
```

```
                println("$name | initial hold state: $receivedState")
                stateUpdate(receivedState)
            #]
        }
    }
    Goto listening


    State listening {
        println("$name | waiting for hold updates")
    }
    Transition t1
        whenEvent hold_update -> update_webgui
        // Con CoAP l'evento si traduce in un dispatch agli observer
        whenMsg hold_update -> update_webgui


   State update_webgui {
        println("$name | update $currentMsg") color red

        [#
            var UpdateJson = payloadArg(0)
            println("$name | hold update received: $UpdateJson")
            stateUpdate(UpdateJson)
        #]
    }
    Goto listening
}
```

Piano di Test

Scenario 1: Invio evento hold_update e verifica ricezione con CoAP

```
@Test
public void testHoldUpdateEventReceptionWithCoap() throws Exception {

    // Serve a bloccare il main thread fino a quando i child thread non
completano
    CountDownLatch latch = new CountDownLatch(1);

    CoapClient client = new CoapClient(COAP_ENDPOINT);

    CoapObserveRelation relation = client.observe(new CoapHandler() {
        @Override
        public void onLoad(CoapResponse response) {
            String content = response.getResponseText();
            System.out.println("CoAP notification received: " + response);

            // Controllo se il messaggio contiene il carico aggiornato
            if (content.contains("\"currentLoad\":100" )
                && content.contains("\"slot1\":occupied"))
```

```
            {

                latch.countDown();
            }
        }

        @Override
        public void onError() {
            System.err.println("CoAP observing failed");
            fail("Errore durante l'osservazione CoAP");
            latch.countDown();
        }
    });

    // Costruisco e invio l'evento hold_update
    String updateJson = "{\"currentLoad\":100, \"slots\":
{\"slot1\":\"occupied\", \"slot2\":\"free\", \"slot3\":\"free\",
\"slot4\":\"free\"}}";
    String event = CommUtils.buildEvent("tester", "hold_update",
"hold_update('" + updateJson.replace("\"", "\\\"") + "')").toString();

    System.out.println("Invio evento hold_update: " + event);
    conn.forward(event);

    //Rimane bloccato qui finchè non viene eseguito latch.countDown()
    latch.await();

    // Pulizia
    relation.proactiveCancel();
    client.shutdown();

    assertTrue("Evento hold_update correttamente ricevuto via CoAP", true);
}
```

Progettazione

L'applicazione *webgui* è un sistema sviluppato utilizzando SpringBoot un framework Java-based scelto per la sua semplicità di configurazione, l'integrazione nativa con pattern MVC e l'uso di annotazioni per la gestione semplificata dei Controller.

La *webgui* è stata progettata col fine di monitorare lo stato della stiva (*hold*) e visualizzarne i cambiamenti in tempo reale attraverso un'interfaccia web. La *Webgui* è un sistema passivo, capace di ricevere e inoltrare aggiornamenti all'interfaccia web.

Ogni componente della *webgui* è responsabile di una specifica funzionalità, come voluto dal principio di singola responsabilità. Ritroviamo tre principali componenti quali:

- HoldStateService: Accesso TCP e interrogazione su richiesta dello stato.
- WSHandler: Gestione WebSocket.
- CoapToWS: Client CoAP e gestione degli aggiornamenti.

Componenti e responsabilità

WSHandler

Il WSHandler è il componente di comunicazione WebSocket occupandosi della gestione delle connessioni con i client browser. Le sessioni attive sono mantenute in memoria, e ogni aggiornamento ricevuto dal sistema viene immediatamente inoltrato ai client tramite broadcast. Le sue funzioni principali sono quelle di:

- Gestire tutte le sessioni client WebSocket connesse
- Fornire un metodo **sendToAll**, che trasmette il messaggio JSON ricevuto da **HoldStateService** o **CoapToWS** a tutti i client attivi.

```
@Component
public class WSHandler extends TextWebSocketHandler {
    private final List<WebSocketSession> sessions = new
CopyOnWriteArrayList<>();

    @Override
    public void afterConnectionEstablished(WebSocketSession session) {
        sessions.add(session);
    }

    @Override
    public void afterConnectionClosed(WebSocketSession session, CloseStatus
status) {
        sessions.remove(session);
    }

    public void sendToAll(String message) throws IOException {
        for (WebSocketSession session : sessions) {
            if (session.isOpen()) {
                session.sendMessage(new TextMessage(message));
            }
        }
    }
}
```

WebSocketConfig

Implementa WebSocketConfigurer per registrare l'handler WebSocket su un endpoint specifico (/status-updates).

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {
    private final WSHandler handler;

    public WebSocketConfig(WSHandler handler) {
        this.handler = handler;
```

```
    }


    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry
registry) {
        registry.addHandler(handler, "/status-updates")
                .setAllowedOrigins("*");
    }
}
```

CoapToWS

Questa componente si comporta come un osservatore CoAP. Alla sua inizializzazione si sottoscrive come osservatore dello stato interno di *hold* accessibile tramite un path simile al seguente: `coap://localhost:8000/ctx_cargoservice/hold`.

Ogni volta che lo stato di *hold* si aggiorna, il CoAP client riceve l'aggiornamento corrispondente tramite una callback. L'aggiornamento viene elaborato estraendo le informazioni e costruendo un nuovo JSON che viene successivamente inviato via WebSocket ai client, tramite WSHandler.

```
@Component
public class CoapToWS {
    // nome che viene risolto da dentro il container
//  private static final String COAP_ENDPOINT =
"coap://arch3:8000/ctx_cargoservice/hold";
    private static final String COAP_ENDPOINT =
"coap://127.0.0.1:8000/ctx_cargoservice/hold";

    private CoapClient client;
    private CoapObserveRelation observeRelation;

    @Autowired
    private WSHandler wsHandler;

    @PostConstruct
    public void init() {
        client = new CoapClient(COAP_ENDPOINT);
        observeRelation = client.observe(new CoapHandler() {
            @Override
            public void onLoad(CoapResponse response) {
                String content = response.getResponseText();
                CommUtils.outblue("CoAP payload: " + content);

                try {
                    JSONObject payload =
HoldResponseParser.parseHoldState(content);
                    if (payload != null) {
                        wsHandler.sendToAll(payload.toString());
                    } else {
                        CommUtils.outred("Evento CoAP non valido: " +
```

```
content);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }

            @Override
            public void onError() {
                System.err.println("Errore nell'osservazione CoAP.");
            }
        });
        System.out.println("Iniziata osservazione CoAP su: " +
COAP_ENDPOINT);
    }
}
```

HoldStateService

Questa componente ha il compito di inviare una richiesta TCP all'attore esterno *hold* per ottenere il suo stato. Il flusso è il seguente:

1. L'utente accede alla pagina web.
2. La pagina invia una richiesta HTTP GET all'endopoint **/holdstate** fornito dal backend SpringBoot.
3. Il controller delega a HoldStateService, che aprendo una connessione TCP verso l'attore _hold, inoltra una richiesta **get_hold_state(X)**.
4. La risposta ricevuta dal controller viene convertita in un oggetto JSON.
5. Il JSON successivamente viene inviato via WebSocker alla pagina web.

```
@RestController
public class HoldStateService {

    @Autowired
    private WSHandler wsHandler;

    private Interaction conn;

    public HoldStateService() {
        try {
            // usa questo da dentro i container
            // conn =
ConnectionFactory.createClientSupport23(ProtocolType.tcp, "arch3", "8000");
            conn =
ConnectionFactory.createClientSupport23(ProtocolType.tcp, "127.0.0.1",
"8000");
        } catch (Exception e) {
            System.err.println("Errore nella connessione TCP iniziale: " +
e.getMessage());
        }
    }
```

```
    @GetMapping("/holdstate")
    public String getHoldState() {
        try {
            IApplMessage request = CommUtils.buildRequest("webgui",
"get_hold_state", "get_hold_state(X)", "hold");
            IApplMessage response = conn.request(request);
            CommUtils.outblue("hold-state query response:" +
response.msgContent());

            String jsonString = response.msgContent().substring(
                    "'hold_state(".length(),
                    response.msgContent().length() - 2 // Rimuovi il
wrapper "'hold_update(" e il suffisso ")'"
                );

            JSONObject payload =
HoldResponseParser.parseHoldState(jsonString);
            if (payload != null) {
                wsHandler.sendToAll(payload.toString());
                return payload.toString();
            } else {
                return "{\"error\":\"payload nullo\"}";
            }
        } catch (Exception e) {
            e.printStackTrace();
            return "{\"error\":\"" + e.getMessage() + "\"}";
        }
    }
}
```

Interfaccia web

La pagina principale di monitoraggio è una semplice interfaccia HTML+CSS+JS responsiva. Include:

- Visualizzazione del carico totale della nave
- Stato dei 4 slot del deposito
- Connessione automatica all'endpoint /status-updates
- Aggiornamento automatica dell'interfaccia all'arrivo di eventi sulla websocket

Al caricamento della pagina viene eseguita anche una chiamata fetch a /holdstate per ottenere lo stato iniziale.

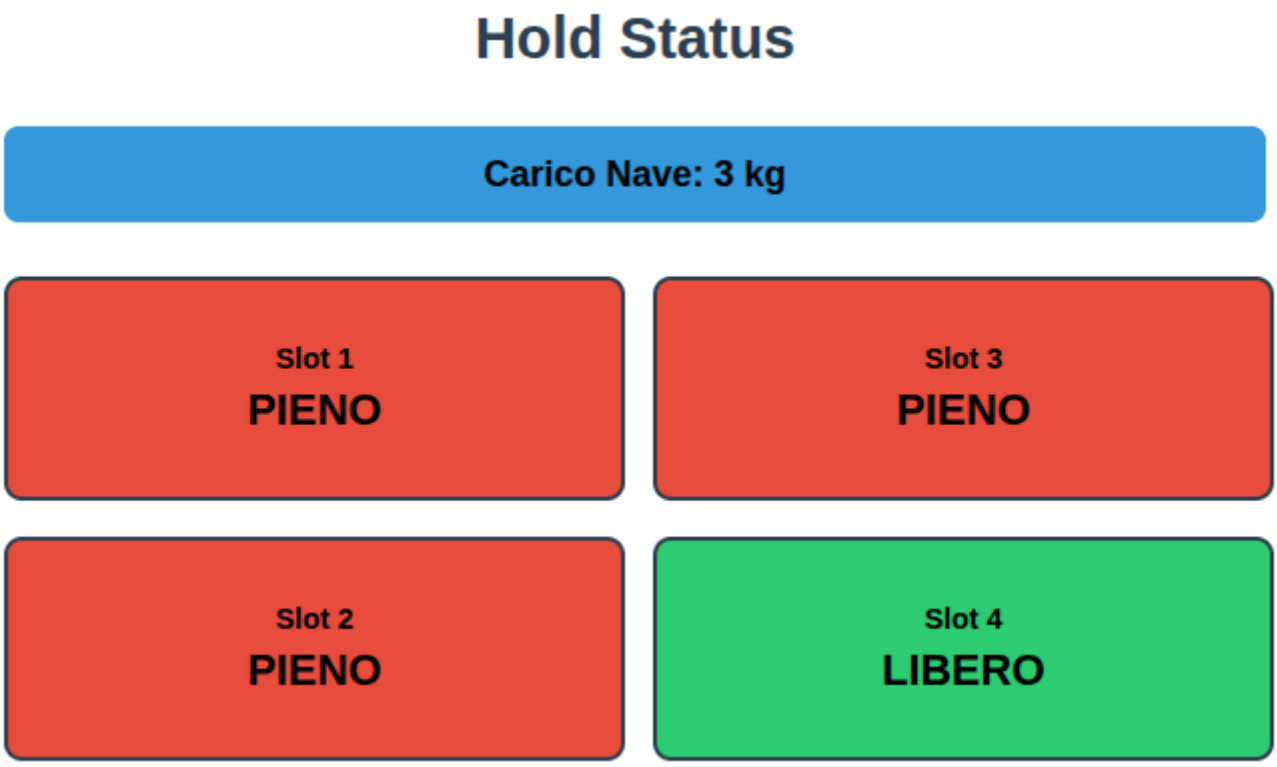
Queste funzionalità vengono implementata dal seguente codice javascript.

```
let socket;

//Funzione per connettersi al WebSocket
function connectWebSocket() {
    const socketUrl = "ws://" + window.location.host + "/status-updates";

    //Crea la connessione WebSocket
```

```
    socket = new WebSocket(socketUrl);

    //Gestione eventi WebSocket
    socket.onopen = function(event) {
        console.log("Connessione WebSocket stabilita");
        updateConnectionStatus(true);
    };

    socket.onmessage = function(event) {
        console.log("Messaggio ricevuto:", event.data);
        try {
            const data = JSON.parse(event.data);
            updateUI(data);
            updateLastUpdate();
        } catch (e) {
            console.error("Errore nel parsing del JSON:", e);
        }
    };

    socket.onclose = function(event) {
        console.log("Connessione WebSocket chiusa");
        updateConnectionStatus(false);

        setTimeout(connectWebSocket, 5000);
    };

    socket.onerror = function(error) {
        console.error("Errore WebSocket:", error);
        updateConnectionStatus(false);
    };
}

// Funzione per aggiornare l'interfaccia con i nuovi dati
function updateUI(data) {
    // Aggiorna il carico della nave
    if (data.shipLoad !== undefined) {
        document.getElementById("shipLoadValue").textContent =
data.shipLoad;
    }

    // Aggiorna lo stato degli slot
    if (data.slots && Array.isArray(data.slots)) {
        for (let i = 0; i < 4; i++) {
            const slotElement = document.getElementById(`slot${i+1}`);
            const statusElement = slotElement.querySelector(".slot-
status");
            const status = data.slots[i] || "libero"; // Default a "libero"
se non specificato

            statusElement.textContent = status.toUpperCase();
            slotElement.className = "slot " + status;
        }
    }
}
```

```
window.onload = function() {
    connectWebSocket();
    // Ogni volta che l'utente si collega, viene recuperato tramite
    // richiesta GET fatta a holdstate, lo stato attuale del deposito
    fetch("/holdstate")
            .then(response => response.json())
            .then(data => {
                console.log("Stato iniziale ricevuto:", data);
                updateUI(data);           // aggiorna l'interfaccia
                updateLastUpdate();       // aggiorna data/ora ultimo
aggiornamento
            })
            .catch(error => {
                console.error("Errore nel recupero dello stato iniziale:",
error);
            });
};
```

In seguito, come si presenta l'interfaccia web.



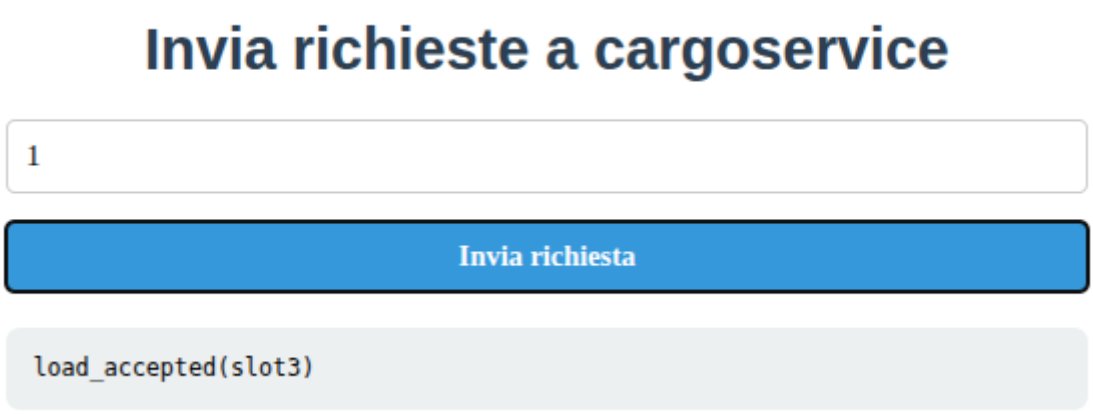
CallerService

Sebbene non esplicitamente richiesto dai requisiti, si è ritenuto opportuno aggiungere anche un altro endpoint al server springboot, e relativa pagina web, che permettesse di **inviare richieste di carico a *cargoservice***.

L'endpoint esposto è /caller?pid=XX. Richieste GET verso questo endpoint vengono tradotte in richieste di tipo **load_product(pid)** verso *cargoservice* e le risposte vengono servite al cliente nella relativa pagina html.

```
@GetMapping("/caller")
    public String callCargoservice(@RequestParam("pid") String pid) {
        try {
            CommUtils.outblue("send request to cargoservice");
            IApplMessage getreq = CommUtils.buildRequest ("webgui",
"load_product","load_product("+pid+")", "cargoservice");
            IApplMessage answer = conn.request(getreq);  //raises exception
            CommUtils.outgreen("response" + answer);
            return answer.msgContent();
        } catch (Exception e) {
            e.printStackTrace();
            return "{\"error\":\"" + e.getMessage() + "\"}";
        }
    }
```

In seguito, come si presenta l'interfaccia per inviare richieste a cargoservice.



Deployment

I modelli QAK sviluppati in questo sprint sono recuperabile alla seguente repository github, dentro alle cartella "system3/" e "iodevices/".

L'intero sistema è stato **containerizzato**, il deployment risulta quindi immediato. Si hanno due possibilità: una se si ha a disposizione un raspberry pi per il sonar, alternativamente si è anche predisposta una versione del sistema con un simulatore del sonar.

Cargoservice con sonar-simulator

1. eseguire `docker compose -f arch3-local.yaml up` per lanciare l'intero sistema
2. aggiungere qualche prodotto al db mongo appena avviato, eseguendo con node lo script setupmongo.js
3. aprire il browser e digitare `localhost:8090` per visualizzare l'ambiente virtuale WEnv e il robot che effettua i suoi interventi di carico

4. aprire un'altra scheda del browser e digitare `localhost:8080` per visualizzare la GUI che mostra lo stato del deposito
5. aprire una terza e ultima finestra e digitare `localhost:8080/caller.html` per visualizzare una GUI con cui mandare le richieste di carico

Cargoservice con sonar su raspberry pi

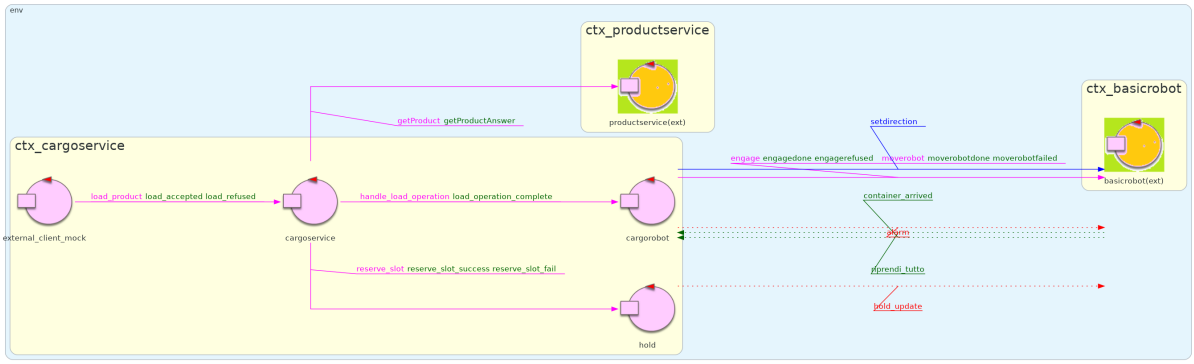
1. creare una distribuzione del progetto /iodevices eseguendo `./gradlew distTar`
2. copiare la distribuzione ottenuta dentro al raspberry, estrarre i contenuti, ed avviare i componenti sonar eseguendo il binario dentro alla cartella /bin
3. eseguire `docker compose -f arch3-rpi.yaml up` per lanciare l'intero sistema (apparte il sonar e il led)
4. aggiungere qualche prodotto al db mongo appena avviato, eseguendo con node lo script setupmongo.js
5. aprire il browser e digitare `localhost:8090` per visualizzare l'ambiente virtuale WEnv e il robot che effettua i suoi interventi di carico
6. aprire un'altra scheda del browser e digitare `localhost:8080` per visualizzare la GUI che mostra lo stato del deposito
7. aprire una terza e ultima finestra e digitare `localhost:8080/caller.html` per visualizzare una GUI con cui mandare le richieste di carico

Sintesi Finale e Nuova Architettura

In questo sprint si sono implementati i componenti: led e webgui. Grazie al primo, è diventato possibile segnalare la presenza di un malfunzionamento del sonar, grazie al secondo è diventato possibile visualizzare lo stato degli slot e il peso complessivo dei container nel deposito.

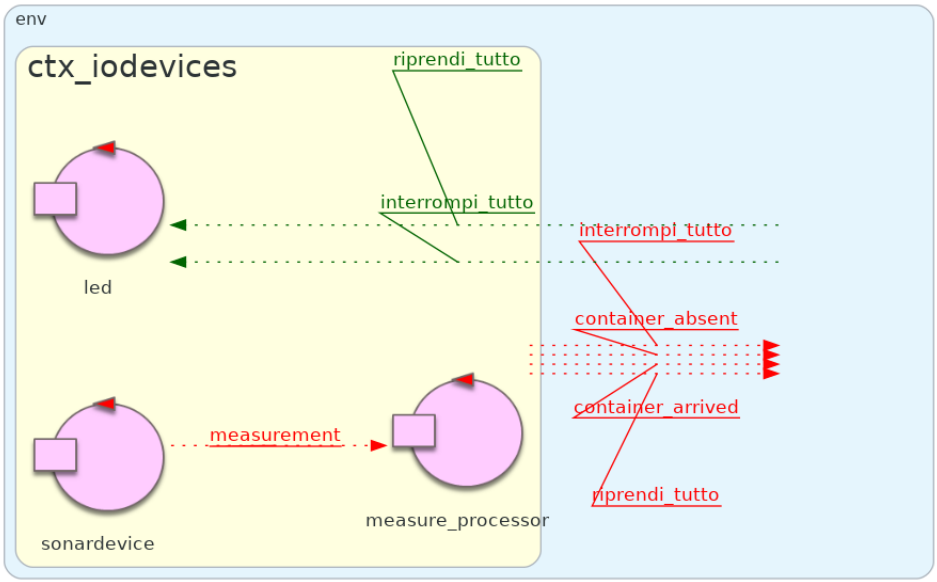
L'architettura del sistema risultante da questo sprint ha definito il nuovo macrocontesto della webgui.

Servizio principale



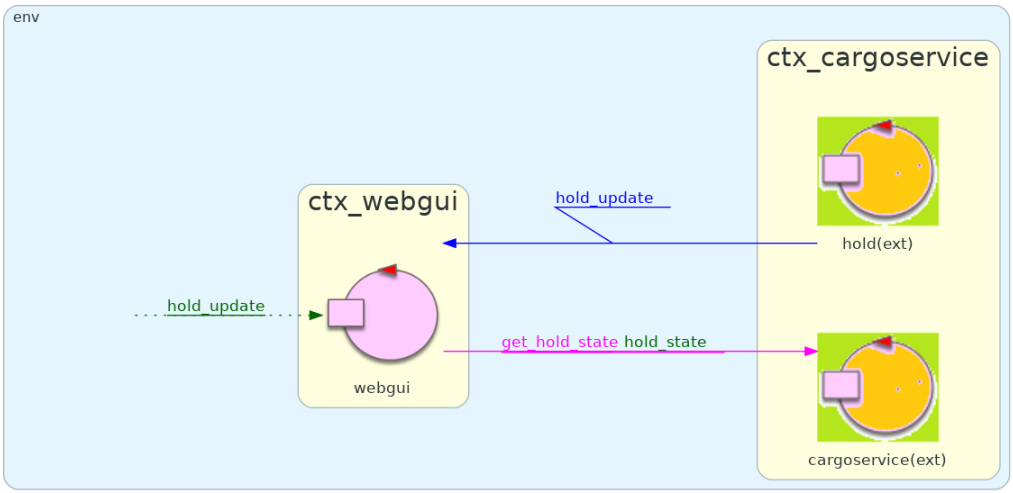
arch2Arch

Dispositivi di I/O



iodevicesArch

WebGui



webguimodelArch

Tempo Impiegato e Ripartizione del Lavoro

Tempo Impiegato

Lo sprint ha richiesto poche ore in più del previsto. Avevamo previsto di procedere al ritmo di uno sprint alla settimana, e così è stato.

Ripartizione del Lavoro

Non c'è nessuna differenza tra la ripartizione del lavoro in questo sprint e la ripartizione del lavoro nel precedente.

Come previsto, tutti i membri del gruppo hanno partecipato attivamente a tutte le fasi dello sviluppo. Questa modalità organizzativa si è rivelata particolarmente soddisfacente, poiché le principali difficoltà riscontrate durante gli sprint hanno riguardato soprattutto la fase di analisi e progettazione, più che quella di implementazione.

In questo contesto, si è dimostrato molto efficace affrontare le problematiche attraverso sessioni di brainstorming collettivo. È infatti raro che un singolo componente riesca a cogliere da solo tutte le sfaccettature di una tematica complessa, mentre il confronto tra punti di vista diversi porta spesso alla sintesi di soluzioni condivise, in grado di soddisfare l'intero team.

Particolarmente utile si è rivelata la necessità di esporre e argomentare le proprie idee fin dalle prime fasi di analisi: il confronto verbale permette di individuare tempestivamente eventuali errori o fraintendimenti, e assicura che il gruppo proceda in maniera coerente, evitando che si consolidino interpretazioni discordanti che potrebbero compromettere il lavoro futuro.