Sprint 2

Sommario

Obiettivi

Nello sprint 1, da obiettivo, sono stati progettati gli attori **cargoservice** e **cargorobot**. Nello sprint 2, invece, si vuole effettuare l'analisi del problema e progettazione dei componenti relativi alla gestione dell'**IOPort**.

Sottoinsieme di requisiti considerati

In questo sprint2 ci siamo concentrati nel sottoinsieme di requisiti relativi alla gestione dell'IOPort. In particolare, facciamo riferimento a quelli che nello sprint0 abbiamo indicato "**a priorità media"**.

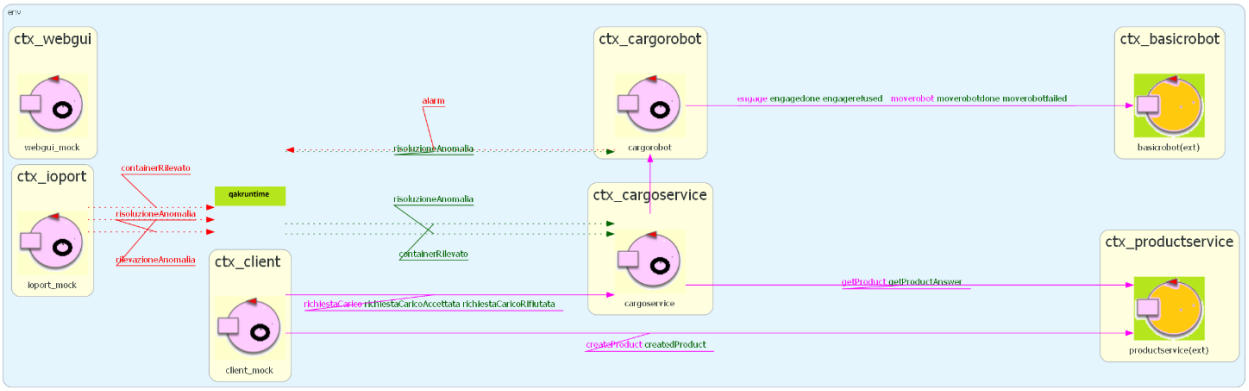
Di seguito vengono riportati i requisiti relativi a questo sprint in cui come già detto verranno sviluppate le funzionalità riguardanti l'**IOPort**. Li abbiamo riportati per comodità in modo tale che in fase di analisi riusciamo ad avere chiaro il contesto generale.

- L'IOPort dovrà rilevare la presenza di un container al suo ingresso tramite un sonar
- Il sonar rileva il container nel momento in cui misura una distanza < DFREE/2 per un certo tempo (*viene detto per esempio 3s, ma nei requisiti si intende che questo tempo possa essere anche diverso*)
- DFREE è la distanza massima che il sonar può misurare se non sono presenti malfunzionamenti
- Se il sonar rileva una distanza > DFREE per almeno 3s (in questo caso 3s è specificato nei requisiti), il sistema deve interrompere ogni attività (il committente ha specificato che con questo si intende che si devono fermare cargorobot e cargoservice in pratica). Questa situazione verrà denominata anomalia o malfunzionamento
- Dopo aver misurato D > DFREE deve essere acceso un led che segnalerà il problema
- Il sistema dovrà tornare a funzionare normalmente quando il sonar rileverà di nuovo D <= DFREE

Note sui requisiti: nei requisiti non è scritto esplicitamente che il led si debba spegnere dopo la risoluzione dell'anomalia, ma è abbastanza ragionevole pensare che sia così.

Altra cosa non esplicitamente descritta nei requisiti è il fatto che una volta rilevato il container, tale avvenimento dovrà in qualche modo dare il via all'effettivo caricamento in stiva del prodotto.

Architettura dello sprint 1



cargorobotArch

Analisi del Problema

Si può notare che, come i componenti analizzati nello sprint 1, l'**IOPort**, sia un'entità di tipo **attore** che può essere considerata in gran parte autonoma rispetto al resto del sistema e può essere racchiusa in un **bounded context**. Ovviamente, questa dovrà comunicare con gli altri attori presenti nel sistema di suo interesse, nello specifico **cargorobot** e **cargoservice**.

Preliminarmente, si possono suddividere i compiti di questo in tre categorie: **interazione** con i componenti cargoservice e cargorobot, **misura effettiva** della distanza e **gestione** del led. Si può pensare di "dividere" il componente formando tre componenti strettamente interagenti tra di loro, ognuno dedito a realizzare una categoria di compiti.

In questo modo le categorie diventano di fatto **indipendenti**: in fase di progettazione eventuali modifiche di una caratteristica non andranno ad influire sull'altra e viceversa; cosa meno probabile nel caso di un solo componente "monolite".

L'obiettivo dell'analisi del problema qui descritta è quello di chiarire e proporre le possibili alternative per la **modellazione** e la **realizzazione** dei seguenti punti non descritti esplicitamente dai requisiti:

- In quale modo è meglio strutturare e dividere le tre categorie di responsabilità appena introdotte?
- Come farà l'IOPort a conoscere DFREE? Lo potrà calcolare in qualche modo attraverso una sorta di calibrazione? O basterà inserirne il valore manualmente?
- Come farà l'IOPort a comunicare l'interruzione delle attività? E la ripresa?
- Come farà il sonar a "dare il via" all'operazione di carico nel momento in cui rileverà un container?

La seguente analisi si baserà sulla modellazione tramite **linguaggio QAK** in quanto, come già specificato negli sprint precedenti è uno strumento estremamente utile per l'analisi e la realizzazione di prototipi di sistemi in cui è predominante la presenza di attori ed entità autonome e interagenti.

Suddivisione delle responsabilità: come già accennato le responsabilità principali dell'IOPort sono la ***rilevazione delle misure tramite il sonar***, la ***gestione del led*** e la ***gestione delle interazioni*** con gli altri componenti. Sono possibili principalmente due approcci: un approccio a oggetti, in cui pur tenendo un attore autonomo, vengono realizzate per mezzo di oggetti (ad esempio dei POJO) gestiti da tale attore, oppure un approccio maggiormente orientato agli attori, in cui ogni responsabilità è affidata a un attore diverso che, pur essendo strettamente legato agli altri, possiede comunque una certa autonomia.

- ✓ Nel **primo caso** si ottiene un modello più monolitico in cui è necessario modellare gli oggetti e i meccanismi necessari. Potrebbe risultare più efficiente a livello di prestazioni, in quanto prevede minori interazioni tramite messaggi, ma porta con sé tutti i problemi legati alla progettazione: è necessario, infatti, gestire eventuali **latenze** o **parti bloccanti** nei messaggi e, anche se il led può essere gestito abbastanza bene anche attraverso una struttura a oggetti, la lettura delle distanze da parte del sonar è un'attività di natura più asincrona, difficilmente realizzabile in maniera efficace tramite oggetti a meno che non si preveda di integrare anche la gestione di diversi processi e/o thread.
- ✓ Il **secondo caso** risolve tutti questi problemi a fronte di una minima perdita in efficienza: gli attori sono più simili a **processi/thread** e permettono, di conseguenza, la divisione delle tre responsabilità semplicemente affidando ciascuna ad un attore diverso e scegliendo le giuste interazioni tra essi. Oltretutto, il linguaggio QAK offre queste caratteristiche in maniera nativa e facile da realizzare.

Viene quindi già fatta una scelta in questi termini orientata alla soluzione ad attori. Nello specifico si propone la realizzazione dell'IOPort tramite 3 attori QAK: ***IOPort***, responsabile della comunicazione con i componenti esterni e del governo delle azioni da realizzare; ***sonar***, responsabile della lettura delle distanze, di una loro gestione di basso livello e della comunicazione di tali dati all'attore IOPort; e ***led***, responsabile, sotto comando, di accendere e spegnere fisicamente il led. In fase di progettazione si deciderà meglio quali azioni concrete affidare a ognuno di questi attori.

***Comunicazione tra i componenti interni*:** posta la suddivisione in attori sopra citata, si pone il problema di come questi debbano comunicare per svolgere il ruolo dell'IOPort nel suo complesso. La comunicazione interna può essere divisa in due interazioni:

- **IOPort – sonar**: l'unica interazione che dovrà avvenire tra questi due componenti è quella relativa all'invio, da parte di **sonar**, di informazioni sulla lettura delle distanze all'**IOPort** (quali informazioni dello specifico verrà trattato in fase di progettazione). Le rilevazioni verranno fatte presumibilmente a certi intervalli di tempo e si figurano logicamente come rappresentazione di eventi nel mondo reale. L'iniziativa della comunicazione partirà dal sonar e non necessiterà di una risposta; perciò, si esclude una comunicazione tramite richiesta-risposta. Prendendo come riferimento QAK, ci si orienta più su una semantica di tipo *dispatch* oppure *event*.
  Tra i due si protende di più verso la scelta di un *event*, in quanto si valuta accettabile il fatto che l'IOPort possa "perdere" una rilevazione se occupata in un'altra attività; anzi, è potenzialmente dannoso l'accumulo di rilevazioni vecchie che potrebbero fornire all'IOPort un'immagine del mondo incoerente con quella reale, portando alla possibilità di ritardi nell'attuazione di azioni. Andando più nello specifico, potrebbe essere ideale l'utilizzo di un streamed event, poiché solamente l'IOPort è interessata a tali eventi.

- **IOPort – led**: l'**IOPort** dovrà semplicemente comandare le operazioni da compiere al **led**: i messaggi saranno quindi finalizzati alla richiesta di accensione o di spegnimento del led. Considerando i messaggi diretti, in questo caso ci sono due possibilità per il tipo di messaggio (facendo riferimento al QAK): tipo *request/reply* e tipo *dispatch*. In questo caso la decisione è meno netta, perché non sembra esserci nessuna risposta informativa che il led possa fornire all'IOPort a seguito del comando se non "OK".
  La differenza è relativa alla semantica: *dispatch* porterebbe a una semantica asincrona, mentre la *request/reply* offrirebbe la possibilità anche di una semantica sincrona. Tra i due tipi si protende leggermente verso l'utilizzo del tipo *dispatch,* in quanto si ritiene più semplice a fronte del fatto che non appare necessaria l'attesa della risposta da parte del led. In conclusione, si può notare come sia possibile non utilizzare messaggi specifici aggiuntivi tra IOPort e led: i momenti di accensione e spegnimento del led, infatti, coincidono con l'emissioni degli eventi legati uno alla segnalazione dell'anomalia e l'altro della sua fine. Questa soluzione risulta preferibile a tutte le altre citate, perché

disaccoppierebbe maggiormente l'IOPort dal led e permetterebbe il corretto riutilizzo dei messaggi già presenti.

Comunicazione con i componenti esterni: Quindi abbiamo deciso che l'attore IOPort sarà responsabile della comunicazione con le entità esterne, ovvero **cargoservice** e **cargorobot**. Tali interazioni avverranno a seguito della rilevazione di un malfunzionamento e, presumibilmente, al momento della rilevazione di un container: quest'ultimo avvenimento, ragionevolmente, dovrà essere in qualche modo fatto percepire anche ai componenti responsabili della gestione della stiva.

- **Malfunzionamento:** l'avvenimento di un malfunzionamento è logicamente un evento che può verificarsi in qualunque momento portando all'interruzione del sistema. Il "sistema" IOPort (con tale termine si intende l'insieme di tutti gli attori trattati nei punti precedenti) rileverà il malfunzionamento nel mondo reale e dovrà comunicare questo evento a cargorobot e cargoservice. Ovviamente tale messaggio è escluso che sia una richiesta, perciò la scelta potrebbe ricadere nuovamente tra tipo *event* e tipo *dispatch*. Si è deciso di procedere scegliendo una semantica (QAK) di tipo *event,* per via del fatto che si tratta di un evento che ha lo scopo di interrompere nel modo più veloce possibile il sistema (una sorta di *interrupt*) e che la percezione in un momento successivo a quello di ricezione potrebbe essere inutile, se non dannosa.

- **Rilevazione container:** anche in questo caso la rilevazione di un container è un evento che viene percepito dall'IOPort e dovrà essere notificato a qualche componente che agisce direttamente sulla stiva. Dall'analisi effettuata nello sprint 1, è emerso come sia più appropriato che la sensibilità alla notifica di rilevazione del container sia esaustivamente del cargoservice: esso, in questo modo, mantiene il pieno controllo sul processo di caricamento e sull'ordine delle operazioni. Per i motivi già elencati in altri punti in questa analisi e per quelli dello sprint 1, si ritiene opportuno modellare la comunicazione del container rilevato dall'IOPort al cargoservice tramite un *event* con semantica QAK. Fare ciò permetterà anche di ignorare in maniera semplice il posizionamento di un container in un momento diverso da quello opportuno durante la fase di caricamento (ad esempio a seguito di un errore da parte dell'operatore).

In quest'ultimo caso l'evento può avere una semantica di *streamed event* (QAK), in quanto non interessa a tutti i componenti del sistema, ma solo a cargoservice. Per quel che riguarda gli eventi di malfunzionamento, come analizzato anche nel punto

precedente, potrebbe essere utile che essi vengano percepiti anche dai componenti interni, oltre che da cargoservice e cargorobot), perciò un normale *event* QAK può essere una scelta più efficace.

NOTA: Da segnalare in questo caso il fatto che QAK non permetta streamed event se non locali al contesto, il che risulta un problema in quanto cargoservice e ioport sono due componenti molto diversi che è molto probabile che eseguiranno su contesti e macchine diverse

Scelta valore di DFREE: DFREE è il parametro usato sia per decidere **presenza** (presenza se `D < DFREE/2` per 3s) sia per rilevare un possibile **malfunzionamento sonar** (failure se `D > DFREE` per 3s). Dunque, la scelta del suo valore influisce su due scenari diversi: quando consideriamo che un container sia arrivato e quando consideriamo che il sonar stia sbagliando. Di conseguenza, è necessario che la sua taratura sia adeguata.

Alcune considerazioni pratica prima di analizzare questa problematica:

1. DFREE/2 deve essere un valore compreso *tra* le misure tipiche con container presente e quelle tipiche con ioport vuota
2. DFREE/2 esso deve essere un valore minore delle misure tipiche a ioport vuota, così che un `"D > DFREE"` sia effettivamente anomalo e non la normale condizione “vuoto”.

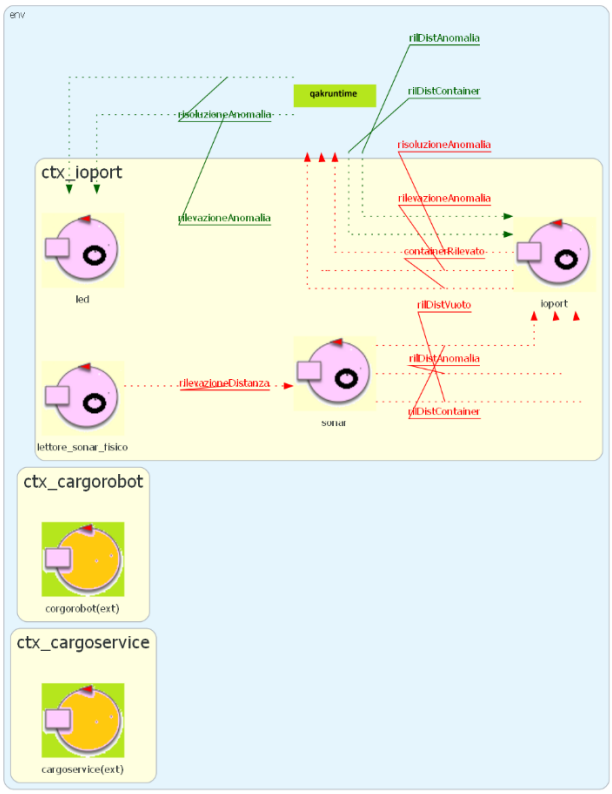
In fase di analisi abbiamo pensato a due possibili approcci per la determinazione di questo parametro:

1. Impostare DREE a priori: valore definito in codice o configurazione, uguale per tutti i sistemi.
    - **VANTAGGI**: Semplicità di implementazione, riproducibilità, manutenzione ridotta, facilità di test.
    - **SVANTAGGI**: Scarsa adattabilità all’ambiente, vulnerabilità a variazioni nel tempo.
2. Calibrare DREE dinamicamente: valore misurato e impostato tramite procedura di setup (decisa poi in fase di progettazione).
    - **VANTAGGI**: Adattabilità reale, precisione maggiore, robustezza nel tempo, scalabilità.
    - **SVANTAGGI**: Maggior complessità SW, tempi di setup iniziale più lunghi

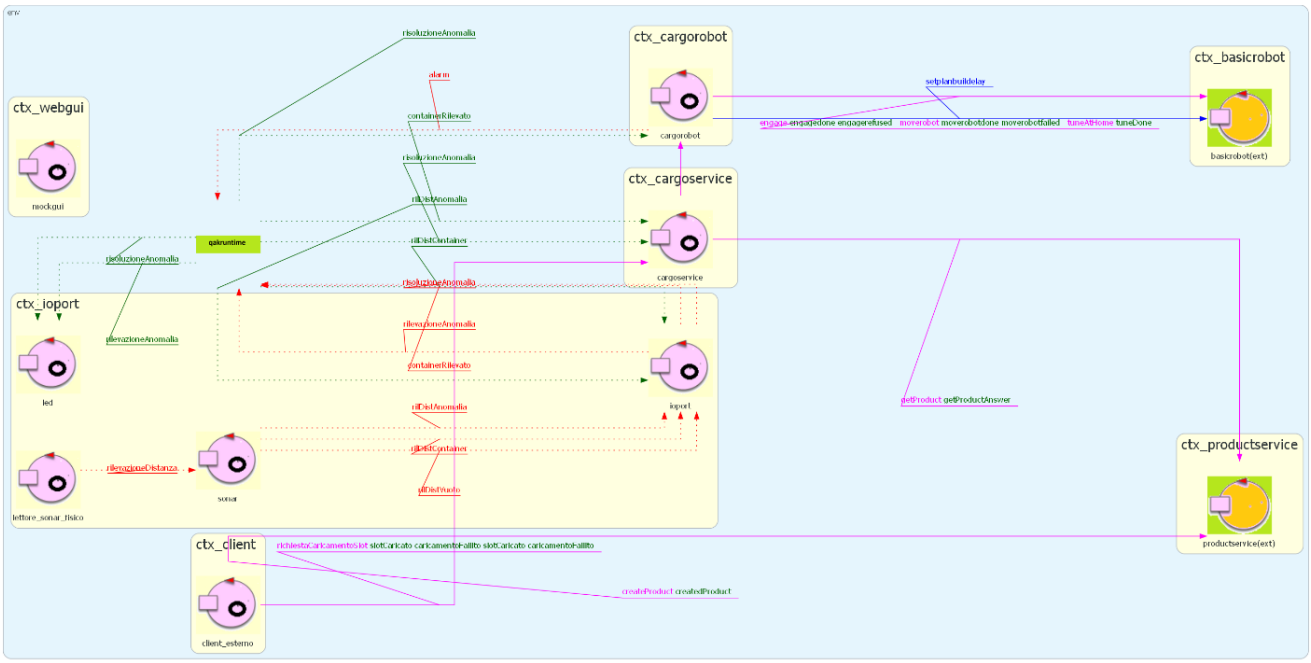
La **calibrazione di DFREE** viene vista come un processo che permette al sistema di adattarsi al contesto fisico reale, migliorando la precisione del rilevamento e l'affidabilità del sonar.
È un'attività che può essere **manuale** o **automatica**, ma in ogni caso deve assicurare che il valore di DFREE rappresenti fedelmente la distanza libera davanti al sensore.

Modello QAK

Di seguito vengono riportato il modello QAK dell'IOPort. Questo modello appartiene chiaramente a un bounded context diverso da quello di cargoservice e cargorobot (visti appunto come attori esterni). La comunicazione tra gli attori dell'IOPort e quelli esterni avverrà solamente attraverso gli eventi già descritti. Successivamente viene riportato anche il modello dell'intero sistema sviluppato fino a questo momento



ioportArch



Riepilogo messaggi

Di seguito viene riportato un riepilogo dei messaggi inerenti all'IOPort

Messaggio	**Tipo**	**Inviato da**	**Ricevuto da**	**Parametri**
RilevazioneAnomalia	Event	IOPort	Cargoservice Cargorobot	
RisoluzioneAnomalia	Event	IOPort	Cargoservice Cargorobot	
ContainerRilevato	Streamed Event	IOPort	Cargoservice	
RilevazioneDistanza	Streamed Event	Sonar	IOPort	Dati rilevazione (da definire in progettazione)

Progettazione

Anche in questo caso, come nello sprint 1, la progettazione è semplificata grazie ai modelli eseguibili realizzati con il DSL QAK. Si riporta comunque il modello dell'attore che si occupa di gestire l'interazione con il sensore e i servizi cargoservice e cargorobot, con i suoi stati e messaggi. I punti principali che abbiamo discusso in fase di progettazione sono i seguenti:

- Caratteristiche dei messaggi di rilevazione
- Scelte progettuali relative alla calibrazione di DFREE

- Scelte progettuali relative alla consistenza delle misurazioni fatte
- Gestione degli attori QAK nei contesti
- Interazione con i componenti fisici (Raspberry)

***Dettaglio messaggi rilevazione**:* nonostante logicamente ogni rilevazione possa essere considerata come un evento *RilevazioneDistanza*, si nota in fase di progettazione che può essere utile dividere tale evento in più tipologie di eventi, riportate di seguito: **rilevazione a vuoto** (DFREE/2 < D <= DFREE), **rilevazione container** (D <= DFREE/2), **rilevazione anomalia** (D > DFREE). Tale suddivisione semantica offre alcuni vantaggi:

- Permette di sfruttare al meglio le funzionalità di transizione tra stati di QAK
- Permette di processare i valori dei dati letti esclusivamente dall'attore **sonar,** garantendo una maggiore divisione di responsabilità
- Permette di facilitare il lavoro dell'attore **IOPort** fornendogli una maggiore astrazione di ciò che sta accedendo nel modo reale. In questo modo è anche possibile per l'IOPort ignorare certe rilevazioni non interessanti in determinati momenti

Di seguito vengono riportati i tre eventi rilevanti che sostituiranno *RilevazioneDistanza*

Messaggio	**Tipo**	**Inviato da**	**Ricevuto da**	**Parametri**
RilDistContainer	Streamed Event	Sonar	IOPort	Valore rilevato
RilDistAnomalia	Streamed Event	Sonar	IOPort	Valore rilevato
RilDistVuoto	Streamed Event	Sonar	IOPort	Valore rilevato

Calibrazione DFREE: Il funzionamento della calibrazione dinamica si baserebbe sul fatto che il software, al primo avvio o dopo reset, assuma l'area dell'IOPort come libera e:

- Legga il sonar per alcuni secondi;
- Tenga in considerazione la più grande delle misure;
- Imposti automaticamente DFREE;
- Registri il valore per i successivi utilizzi.

Per garantire flessibilità e robustezza nella gestione di DFREE, il sistema prevede un **meccanismo di selezione tra due modalità operative**: la prima utilizza un valore a priori definito in configurazione; la seconda è una calibrazione dinamica automatica. La scelta tra le due modalità avviene tramite uno **switch configurabile** a livello software.

- ✓ In modalità **valore a priori**, il sistema legge il valore di DFREE direttamente dalla configurazione e lo adopera come soglia per la rilevazione di presenza di un container e per il monitoraggio di eventuali malfunzionamenti del sonar, senza eseguire alcuna calibrazione iniziale. Tale modalità risulta utile per test, simulazioni o installazioni in ambienti controllati, dove il valore fisso è noto e affidabile.

- ✓ In modalità **calibrazione dinamica**, come accennato prima, il sistema esegue, al primo avvio o a seguito di un reset, una procedura di misura automatica che arte dal presupposto che l'IOPort rimanga libera durante tutto il processo dic alibrazione: come primo passo acquisisce una serie di misurazioni del sonar a vuoto per un certo tempo (si ritiene che 10 secondi possano essere sufficenti), tiene in considerazione la distanza maggiore, infine, imposta il valore di DFREE in base ai dati raccolti. Il valore così determinato viene memorizzato e utilizzato come riferimento stabile fino a successiva ricalibrazione. Il vantaggio di tenere in considerazione la **distanza più grande** è di evitare la detection di un elevato numero di anomalie spurie ed essere robusti a misurazioni imprecise del sonar. Questo approccio funziona meglio rispetto a una soluzione in cui viene fatta la media delle rilevazioni effettuate.

Consistenza rilevazioni: Al fine di avere la correttezza e la stabilità delle rilevazioni del sonar, il sistema deve assicurare che le condizioni di presenza o malfunzionamento vengano confermate in modo consistente. (presenza confermata se D < DFREE/2 per almeno 3 secondi).

Il sonar è gestito da uno script Python fornito dal committente, il quale esegue una **misurazione al secondo** fornendo così un flusso discreto di dati. In fase di progettazione, sono state considerate due possibili strategie per valutare la condizione di soglia:

1. **Basarsi sul tempo**: cioè, verificare che la condizione sia mantenuta per un intervallo di tempo predeterminato (ad esempio 3 secondi), indipendentemente dal numero di misurazioni disponibili.
    a. **Vantaggio**: generalità e indipendenza dalla frequenza di campionamento. Se il sistema sottostante di acquisizione dovesse cambiare o la frequenza variasse, non sarebbe necessario modificare la logica di valutazione.
    b. **Svantaggio**: in presenza di latenza nelle misurazioni, si rischia di non avere il numero minimo di letture necessarie per confermare la condizione, con possibili falsi negativi.

2. **Basarsi sul conteggio di misurazioni**: cioè si considera che la condizione deve essere verificata su un numero minimo di letture consecutive (ad esempio 3 letture corrispondenti a circa 3 secondi).
    a. **Vantaggio**: maggiore robustezza rispetto a latenze occasionali o picchi di ritardo nella raccolta dei dati, perché il controllo è legato a misurazioni effettivamente ricevute.
    b. **Svantaggio**: meno generale, legato alla frequenza di campionamento attuale. Se in futuro dovesse venire modificata la frequenza dello script, la logica del conteggio dovrebbe essere aggiornata di conseguenza.

Nel nostro contesto, dato che lo script Python attuale fornisce una misura al secondo, la strategia basata sul **conteggio di misurazioni consecutive** è preferibile per ridurre la sensibilità a ritardi o picchi sporadici.

Tuttavia, la logica implementativa mantiene la possibilità di calcolare il numero di misurazioni a partire dal tempo desiderato e dalla frequenza di campionamento (nel caso il software di interazione con il sonar fisico dovesse cambiare), parametri forniti in fase di configurazione.

Gestione attori qak nei contesti: gli attori QAK inerenti all'IOPort sono autonomi in una certa misura, e potrebbero essere potenzialmente legati a tre contesti QAK diversi; e quindi potenzialmente a tre microservizi o nanoservizi autonomi. Per il momento, però, si decide, di inserirli in un unico contesto, poiché strettamente legati fra loro in un proprio bounded context. Se in futuro vi sarà necessità di dividerli, sarà possibile porre rimedio semplicemente grazie alle funzionalità offerte da QAK.

Si segnala anche il fatto che a livello implementativo risulta utile aggiungere un nuovo attore *lettore_sonar_fisico* il cui compito sarà quello di interfacciarsi direttamente con il sonar fisico e trasformare le rilevazioni in eventi (QAK) di lettura interni di tipo *rilevazioneDistanza*. La pipeline diventa quindi: *lettore_sonar_fisico* legge i valori del sonar e li trasforma in eventi QAK (streamed) interni al contesto -> *sonar* riceve gli eventi *rilevazioneDistanza* e crea degli eventi QAK (streamed) in base ai tipi visti in precedenza -> *ioport* riceve gli eventi legati al tipo della rilevazione (introdotti nei punti precedenti) e li elabora al fine di emettere eventi legati al malfunzionamento e alla presenza di container verso gli attori esterni cargoservice e cargorobot

Interazione con i componenti fisici: Il sistema cargoservice è progettato per interagire con componenti fisici reali tramite un **Raspberry Pi**, che funge da controller

locale per il sensore sonar e il LED di stato. L'interazione con questi dispositivi è gestita tramite **script Python forniti dal committente**, che espongono le funzionalità di lettura e controllo in modo semplice e affidabile.

- *Sensore sonar*: Il file **`sonar.py`** gestisce il sensore ad ultrasuoni collegato ai pin del Raspberry Pi. Lo script esegue un ciclo continuo di misurazioni, inviando un impulso tramite TRIG e misurando il tempo necessario al ritorno dell'eco tramite ECHO. La distanza calcolata in centimetri viene stampata sullo standard output ogni secondo, garantendo così un flusso di dati regolare per il consumo da parte del sistema principale. L'uso di `sys.stdout.flush()` permette di rendere immediatamente disponibili le misurazioni a qualsiasi processo che legga lo stdout dello script, assicurando consistenza nelle misurazioni.

- *Led di segnalazione*: Il sistema prevede un LED collegato al pin del Raspberry, utilizzato per indicare anomalie del sensore o stati di allarme. La gestione del LED è affidata a due script distinti:
  - **`ledPythonOn.py`**: accende il LED impostando il pin GPIO in stato HIGH.
  - **`ledPythonOff.py`**: spegne il LED impostando il pin GPIO in stato LOW.

  Questi script vengono lanciati dall'attore **led** che, come definito in fase di analisi, è responsabile di accendere e spegnere fisicamente il led. Ad esempio, quando viene rilevata dal sonar una distanza maggiore di DFREE per più di 3 secondi segnalando un'anomalia.

- *Integrazione con il sistema:* Il Raspberry Pi, tramite questi script, costituisce il livello di **interfaccia fisica** tra il mondo reale (sensore sonar e LED) e la logica di controllo di `cargoservice`. Il servizio principale può:
  - **Leggere le distanze**: invocando `sonar.py` e interpretando le misurazioni in stdout per rilevare la presenza di container all'IOPort.
  - **Gestire il LED di allarme**: invocando `ledpythonon.py` o `ledpythonoff.py` in base alle condizioni rilevate dal sonar.

Questa architettura separa chiaramente la **logica di controllo** (cargoservice) dalla gestione diretta dei dispositivi fisici, rendendo il sistema modulare, con funzionamento facile da verificare attraverso l'impiego di test, e mantenibile. In aggiunta, lo script di lettura sonar campiona a intervalli di 1 secondo, il che deve essere considerato nella logica di validazione delle misurazioni (ad esempio nel conteggio di letture consecutive per confermare presenza o failure).

Piano di Test

L'obiettivo è quello di verificare il **corretto funzionamento e l'integrazione del sonar** (simulandolo in ambiente di test JUnit) con il servizio `cargoservice` e, indirettamente, con il `cargorobot`, garantendo che:

- le misurazioni del sonar siano correttamente interpretate dal sistema;
- le condizioni di presenza e failure siano rilevate e gestite come da requisiti;
- le azioni conseguenti (come accensione/spegnimento LED o invio di comandi al robot) siano coerenti con gli eventi generati.

Test automatici con sonar simulato (JUnit): I test vengono realizzati come **test JUnit** sul sistema QAK, senza hardware fisico.
La simulazione si basa sulla generazione di eventi che rappresentano le misure del sonar e sulla verifica empirica della risposta da parte del servizio ioport. In questo caso risulta difficile intercettare gli eventi QAK emessi dall'ioport tramite JUnit, quindi si preferisce un approccio basato sull'osservazione di come il sistema reagisce tramite terminale.

Test non automatici (hardware): I test manuali vengono eseguiti sul Raspberry Pi per verificare il corretto comportamento fisico del LED e del sonar reale.

Test di integrazione: infine si prevede di testare il sistema nel complesso, mettendo in funzionamento l'ioport sul Raspberry Pi e i servizi cargoservice e cargorobot sviluppati nello sprint precedente.

Deployment

Il componente realizzato viene rilasciato come microservizio installato su una Raspberry PI versione 4, alla quale è collegato il sonar e il led. Il microservizio conterrà tutti gli attori relativi al contesto dell'ioport (come mostrato nell'immagine dell'architettura del sistema)

Per caricare il container su docker:

1. Creare l'immagine docker (una solva volta oppure di nuovo quando cambia il progetto). Entrare nella cartella del progetto di cui si vuole fare l'immagine e fare:
   a. ***./gradlew distTar***
   b. ***docker build -t ioport:1.0 .***
2. Entrare nelle varie cartelle dove ci sono i file yaml e lanciare il comando docker-compose giusto

docker-compose -f ioport.yaml up

Si permette di configurare il microservizio attraverso le seguenti variabili d'ambiente:

- **DFREE_CALIBRATION_FLAG**: 0 = attivazione modalità DFREE fissa (importante impostare DFREE correttamente), 1 = attivazione modalità DFREE con calibrazione
- **DFREE**: valore di DFREE fisso (rilevante solo se DFREE_CALIBRATION_FLAG=1)
- **RIL_CONTAINER_SECS**: tempo (in secondi) necessario per rilevare la presenza di un container
- **RIL_ANOMALIA_SECS**: tempo (in secondi) necessario per rilevare la presenza di un'anomalia
- **SONAR_MIS_PER_SEC**: numero di misurazioni della distanza del sonar al secondo
- **SONAR_SCRIPT_PATH**: percorso script python che legge i dati dal sonar
- **LED_OFF_SCRIPT_PATH**: percorso script python che spegne il led
- **LED_ON_SCRIPT_PATH**: percorso script python che accende il led

Ripartizione del lavoro

Lo sviluppo dello **sprint 2** è stato organizzato attraverso una chiara suddivisione delle attività tra i tre membri del gruppo, al fine di garantire un avanzamento parallelo e coordinato delle diverse componenti del progetto. Ognuno dei membri del team si è occupato di una macroarea, anche se ogni punto è stato ampiamente discusso fra tutti i membri:

- **Brighi Valerio: Stesura generale e revisione del documento e deployment:** Redazione della fase di analisi del problema, definizione delle soluzioni architetturali, stesura della documentazione sulle decisioni progettuali. Inoltre, ha gestito la parte sul deployment.
- **Zoccadelli Lorenzo: Implementazione codice QAK e gestione interazione Raspberry**
  Scrittura e validazione del codice QAK, realizzazione degli attori `IOPort`, `sonar` e `led`, definizione di messaggi, eventi e interazioni interne ed esterne previste dal modello. Inoltre, si è occupato dell'interazione con i componenti fisici: simulazione del sonar e fase di integrazione del Raspberry Pi nel sistema, gestione del collegamento e del funzionamento del sonar e del LED fisici.
- **Guiducci Daniele: Analisi, Progettazione e Test:**
  Chiaramente l'analisi e la progettazione sono state fatte e discusse da tutti e tre i membri del team, andando sullo specifico Guiducci Daniele si è occupato dell'analisi della problematica sulla scelta del valore di DFREE e della sua

calibrazione in fase di progettazione (oltre che alla progettazione della consistenza dei valori). Inoltre, si è occupato dello sviluppo di codice di test JUnit per i test automatici con il sonar simulato.

Questa distribuzione del lavoro ha permesso di coprire in modo completo le fasi di analisi, progettazione, sviluppo e integrazione, mantenendo una chiara responsabilità di ogni membro del gruppo. Come già accennato alcune parti sono state fatte totalmente insieme, come la revisione finale del documento, test manuali eseguiti sul Raspberry Pi e anche i test di integrazione per testare il sistema nel complesso (ioport sul Raspberry Pi insieme ai servizi cargoservice e cargorobot sviluppati nello sprint precedente).

Conclusioni

Lo **sprint2** ha consentito di completare l'analisi e la progettazione del sottosistema relativo all'**IOPort**, integrando la gestione del sonar, del LED di segnalazione e delle comunicazioni con i componenti `cargoservice` e `cargorobot`. Il lavoro ha portato a una chiara suddivisione delle responsabilità in attori QAK distinti e cooperanti, migliorando la modularità e la manutenibilità del sistema.

Dal punto di vista funzionale, sono stati definiti i meccanismi di rilevazione della distanza e di gestione delle anomalie, introducendo anche un approccio flessibile per la determinazione del parametro **DFREE**, selezionabile tramite switch tra modalità a priori e calibrazione dinamica.
Questo consente al sistema di adattarsi sia a contesti controllati che a condizioni operative reali, mantenendo robustezza e affidabilità.

La progettazione ha inoltre affrontato il tema della consistenza delle misurazioni e dell'interazione con i componenti fisici su **Raspberry Pi**, garantendo la coerenza tra il mondo reale (sonar e LED) e la logica software. I test automatici JUnit hanno validato la logica di rilevazione e di risposta del sistema, mentre i test manuali hanno confermato il corretto comportamento dei dispositivi fisici.

Con questo sprint si conclude la parte di analisi e sviluppo dei componenti logici e fisici principali del sistema. **L'ultimo sprint previsto** sarà dedicato allo **sviluppo della WebGUI**, che avrà il compito di visualizzare in modo dinamico lo stato della stiva, i messaggi dei servizi e gli eventi generati dal sistema, offrendo così un'interfaccia utente completa e interattiva per la supervisione dell'intero processo di carico.