chronicles_sprint1_v0

Il team

Diego Bruno	Marco Crisafulli	Sebastiano Giannitti

GITHUB DEL TEAM

Introduzione allo Sprint 1 e obiettivi attesi

L'obiettivo dello Sprint 1 è quello di definire e produrre la business logic dei componenti principali del sistema software sviluppato.
Al fine di garantire la robustezza di questa parte, ci poniamo anche l'obiettivo di scrivere dei test che ne verifichino il corretto funzionamento, in modo da rendere iterabile e affidabile questa base di codice.

Link allo sprint precedente:
<add link>

Goal dello Sprint 1:

- Analisi del problema
- Definizione e produzione della business logic
- Test per la business logic finalizzati a garantirne il corretto funzionamento

Analisi del problema

Lo Sprint 1 segna il passaggio dall'analisi dei requisiti all'analisi concreta del problema.
A partire dal modello concettuale individuato nello Sprint 0, abbiamo implementato una prima versione funzionante del sistema, volta a gestire il flusso completo di una richiesta di carico.

L'obiettivo principale è comprendere come gli attori interagiscono realmente e quali vincoli o comportamenti emergano durante l'esecuzione del sistema.

`Comunicazione`
Al fine di instaurare un canale di comunicazione di tipo CoAP viene sfruttato il servizio Eureka per permettere ai servizi di "trovarsi" e conoscere così le rispettive porte esposte.

I componenti forniti dal committente sono già dotati del codice che permette di registrarsi su eureka.
Nei componenti da noi sviluppati è invece stata implementata una funzione per la

registrazione dei client su eureka e il codice qak si occupa di fare il “discovery” ogni qual volta che un contesto è dichiarato come segue:

```
Context ctxbasicrobot ip [host="discoverable" port=8020]
Context ctxcargoservice ip [ host="discoverable" port=8111]
```

Funzione usata per registrare i servizi su eureka:

```
//REGISTER
fun register(){

if( CommUtils.ckeckEureka( ) ){
    val discoveryclient = CommUtils.registerService( main.java.EurekaServiceConfig() )
    CommUtils.outmagenta("discoveryclient=$discoveryclient ")
    CommUtils.outmagenta("EUREKA")

}else{CommUtils.outmagenta("NON VA EUREKA")}
```

I file di eurekaserviceconfig sono stati modificati per permettere ai vari servizi di trovare il server eureka su cui registrarsi e sul quale sono già registrati cargorobot e productservice.

La documentazione di eureka è disponibile qui.

ProductService (PRD-1, P-1)
L’attore productservice è fornito dal committente, sviluppato in linguaggio qak e si occupa di fornire il PID e peso dei prodotti registrati in database (anch’esso fornito dal committente).

```
Request getProduct : product( ID )
Reply   getProductAnswer: product( JSonString ) for getProduct
```

Apprendiamo inoltre il formato dei prodotti in database e come ci verranno restituiti:

```
//String JSON '{"productId":31,"name":"p31","weight":311}'
```

Per distribuire le responsabilità e per gestire al meglio le richieste è stato scelto di sfruttare un attore adapter per il database chiamato productpolice.

Questo attore è un attore di tipo dinamico (viene creato al momento da cargoservice, instaura la comunicazione e restituisce i dati, al termine del flusso si distrugge), questa scelta garantisce l’atomicità dell’operazione di comunicazione con productservice.
Ogni nuova richiesta a cargoservice implicherà la creazione di un nuovo adapter capace di comunicare con productservice.

Il formato dei messaggi di tipo JSON richiede il parsing dei messaggi dentro l’attore per estrarre i valori di peso e PID (realizzato con libreria JSON: com.googlecode.json-simple).

CargoRobot (ROB-1, ROB-2)
CargoRobot è l'attore che opera nel contesto cargoholdservice e si occupa di comunicare con il software basicrobot24, tramite modello di comunicazione request/reply.

```
Request engage         : engage(OWNER, STEPTIME)
Reply   engagedone     : engagedone(ARG)     for engage
Reply   engagerefused  : engagerefused(ARG)  for engage

Dispatch disengage     : disengage(ARG)
```

Alla ricezione della richiesta di carico il cargorobot va ad estrarre le coordinate dello slot assegnato.

```
onMsg(moverobotinternal : moverobot( SLOT )) {
    [#
        CommUtils.outblue("-------- Received request with SLOT ---------")
        var slot = payloadArg(0).toInt()
        println( slot )
        var coords = when(slot) {
            1 -> Pair(1, 1)
            2 -> Pair(4, 1)
            3 -> Pair(1, 3)
            4 -> Pair(4, 3)
            else -> Pair(0, 0)
        }
        TARGETX = coords.first
        TARGETY = coords.second
        CommUtils.outblue("CargoRobot | Slot=$slot -> coords=($TARGETX,$TARGETY)")
    #]
    request basicrobot -m engage : engage($MyName, 330)
}
```

Ad ingaggio avvenuto, al basicrobot viene comunicato di dirigersi all'IOport per prelevare il carico, successivamente gli vengono comunicate le coordinate dello slot assegnato.

Una volta completato il percorso il robot torna alla HOME e termina la comunicazione.
Queste mosse sono supportate anche da un algoritmo di pathfinding A* che permette di trovare la strada verso delle coordinate (x,y).
Il robot rimane allineato con la griglia della HOLD perchè al termine di ogni percorso torna in HOME e fa delle manovre per allinearsi nuovamente alla griglia.

Hold (H-1, H-2, H-3, H-4)

L'attore Hold opera nel contesto ctxcargoholdservice e si occupa di modellare la stiva. Esso è progettato come un FSM che gestisce le richieste di allocazione di slot (holdrequest) e utilizza del codice Kotlin per il mantenimento dello stato interno della stiva.

La gestione degli slot è basata su 4 array Kotlin: slotState, slotPid e slotPeso, e definisce tre stati possibili per ogni slot; FREE (0), RESERVED (1) e OCCUPIED (2). Viene inoltre definito il carico massimo (MAXLOAD) pari a 1000. Le funzioni kotlin sono

findFree() che restituisce l'indice del primo slot disponibile, o -1 se nessuno lo è;

reserve(i, pid, peso) che imposta lo stato dello slot i come RESERVED e memorizza il PID e il peso del prodotto;

occupy(i) che imposta lo slot i come OCCUPIED se è RESERVED;

release(i) che libera lo slot i ponendolo come FREE resettando i valori di PID e peso.

Vengono utilizzate le variabili lastPeso, lastPid, lastIdx, invalid, full e overweight per scambiare dati con le guardie QAK.

La logica della FSM ha uno stato iniziale dove l'attore inizia nello stato s0 e passa allo stato ready dove attende richieste; alla ricezione di una richiesta holdrequest(PESO, PID) l'attore estrae i valori e valida PID e Peso. In base ai loro valori aggiorna le condizioni invalid, overweight e full, se una delle condizioni è vera l'attore transita nella condizione corrispondente che invia una risposta di fallimento con l'argomento appropriato e ritorna nello stato di ready. Se la transizione ha successo (tutte le guardie sono false), l'attore transita a reserve, stato in cui viene invocato il codice Kotlin sullo slot libero trovato, invia al richiedente l'assegnazione dello slot e torna in ready.

Architettura generale

L'architettura è composta da più attori QAK, ognuno con responsabilità specifiche, collegati tramite messaggi `dispatch`, `request`, `reply` o `event`.
I principali componenti implementati sono:

`cargoservice`

È il cuore del sistema e gestisce l'intero ciclo di vita di una richiesta `requesttoload(PID)`.

Coordina i componenti interni (`productpolice`, `hold`, `cargorobot`) e comunica con i servizi esterni (`productservice`, `basicrobot`).
Gestisce tutte le fasi operative:

1. Ricezione e validazione del PID.
2. Verifica della disponibilità e del peso nella hold.
3. Attesa del segnale del sonar (fase simulata in questo sprint).
4. Ingaggio del robot per la movimentazione del carico.
5. Gestione degli errori e stato di fallback requestrefused.

`productpolice`

È un attore dinamico creato dal cargoservice per gestire la verifica del prodotto.
Richiede i dati al `productservice` e, se il PID è valido, restituisce il peso.
In caso contrario, genera un messaggio di fallimento e termina la propria esecuzione.

`hold`

Rappresenta la stiva, con 4 slot di carico modellati tramite tre array paralleli che memorizzano stato, PID e peso.
Gestisce i casi di:

- payload non valido,
- sovrappeso rispetto a `MAXLOAD`,
- stiva piena.

Quando possibile, assegna il primo slot libero e restituisce `holdreply(SLOT, PID)`.

`cargorobot`

Riceve la richiesta di movimento `moverobotinternal(SLOT)` e calcola le coordinate della destinazione.
Segue tre fasi di movimento:

1. Da HOME all'IOPort (prelievo del prodotto);

2. Da IOPort allo slot assegnato;
3. Ritorno alla HOME.
   Il comportamento è simulato tramite richieste `engage` e `moverobot` al `basicrobot`.

`Attori esterni`

- `productservice`: fornisce i dati dei prodotti.
- `basicrobot`: gestisce i movimenti fisici del robot.
- `user`: rappresenta un operatore o test mock.

Tabella riassuntiva

Componente	**Requisito soddisfatto definito in Sprint 0**	**Funzionalità**	**Stato di sviluppo Sprint 1**
productservice	PRD-1	Fornisce PID e peso dei prodotti registrati in database. Interrogato dal `productpolice` tramite `getProduct(ID)`.	Attore esterno fornito; integrato e testato.
productpolice	PRD-1 P-1	Verifica la validità del PID e ottiene il peso dal `productservice`. Risponde al `cargoservice` con `productreply` o `productreplyfailed`.	Attore dinamico sviluppato in QAK.
products	P-1	Ogni prodotto ha PID e peso recuperati da `productservice`.	Gestiti in forma di messaggio; nessuna classe separata.

hold	H-1 H-2 H-3 H-4	Gestisce 4 slot, ciascuno con stato, PID e peso. Verifica payload, sovrappeso e disponibilità. Restituisce slot assegnato o errore.	Attore sviluppato e funzionante.
sensor (sonar)	SNS-1 SNS-2	Non ancora integrato. Previsto sviluppo per Sprint 2	Non implementato (mock).
LED	LED-1	Funzione di notifica non ancora implementata - Sprint 2	Non implementato.
cargorobot	ROB-1 ROB-2	Riceve `moverobotinternal(SLOT)` dal cargoservice. Calcola coordinate, muove il robot verso IOPort, slot e ritorna a HOME.	Attore sviluppato; movimenti simulati via basicrobot.
basicrobot	(da contesto esterno)	Riceve comandi `engage` e `moverobot` dal `cargorobot`.	Comunicazione con attore esterno integrata.
cargoservice	CRS-1 CRS-2 CRS-3 CRS-4 CRS-5 CRS-6	Coordina l'intero flusso di carico: verifica PID, richiede slot alla hold, avvia il cargorobot. Gestisce errori e stato di attesa.	Attore principale sviluppato e testato.
web-GUI	GUI-1	Interfaccia non ancora realizzata. Prevista integrazione allo Sprint 3.	Non implementato.

ctxusermock	(nuovo)	Simula l'utente che invia `requesttoload(PID)` e riceve risposte.	Mock attivo per test manuali.

Flusso operativo

1. L'utente invia una `requesttoload(PID)` al `cargoservice`.
2. Il `cargoservice` crea `productpolice` e richiede la validazione del prodotto.
3. Se il prodotto è valido, ottiene il peso da `productservice`.
4. Viene inviata una `holdrequest(PESO, PID)` alla `hold`.
5. Se uno slot è disponibile, `hold` risponde con `holdreply(SLOT, PID)`.
6. Il `cargoservice` richiede al `cargorobot` di spostare il prodotto nello slot corrispondente.
7. Il robot effettua il percorso IOPort → SLOT → HOME.
8. In caso di successo, `cargoservice` stampa "Delivery completed successfully!" e torna in attesa.
9. Qualsiasi fallimento intermedio porta allo stato `requestrefused`.

Testing???????????

Durante lo sprint abbiamo progettato test di unità e integrazione, utilizzando messaggi QAK per simulare il comportamento dei vari attori.

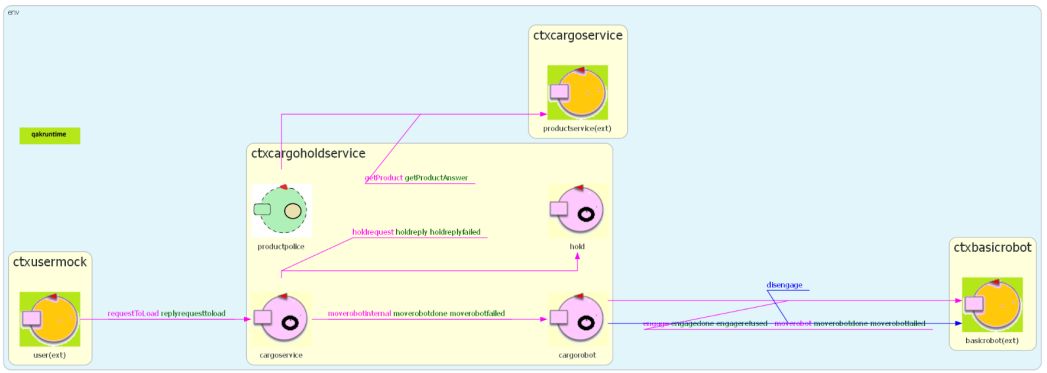
Test ID	Caso di test	Input	Risultato atteso
T1	PID valido, peso entro MAXLOAD, slot disponibile	`requesttoload(1)`	Delivery completata
T2	PID valido ma peso > MAXLOAD	`requesttoload(2)`	`holdreplyfailed(OVERWEIGHT)`
T3	PID valido ma nessuno slot libero	`requesttoload(3)`	`holdreplyfailed(FULL)`

T4	PID non registrato	`requesttoload(99)`	`productreplyfailed`
T5	Errore in movimento robot	simulato	`moverobotfailed` + ritorno a `waitRequest`

I test sono stati eseguiti tramite il contesto `ctxusermock`, con verifica visiva dei log su console.

Proposta di modello

Lo Sprint 1 ha permesso di realizzare la prima business logic completa del sistema, in grado di gestire l'intero ciclo di carico.
L'architettura è pronta per essere estesa con componenti fisici e funzionalità di monitoraggio.



cargoserviceArch

Piano di lavoro futuro

Sprint	Obiettivo principale	h/uomo
2	IOManager, in questa fase si svilupperà la parte di gestione del sonar, il rilevamento del product container nella porta di IO e l'interfacciamento hardware dei dispositivi fisici di sonar e led con il sistema.	20