Sprint 2

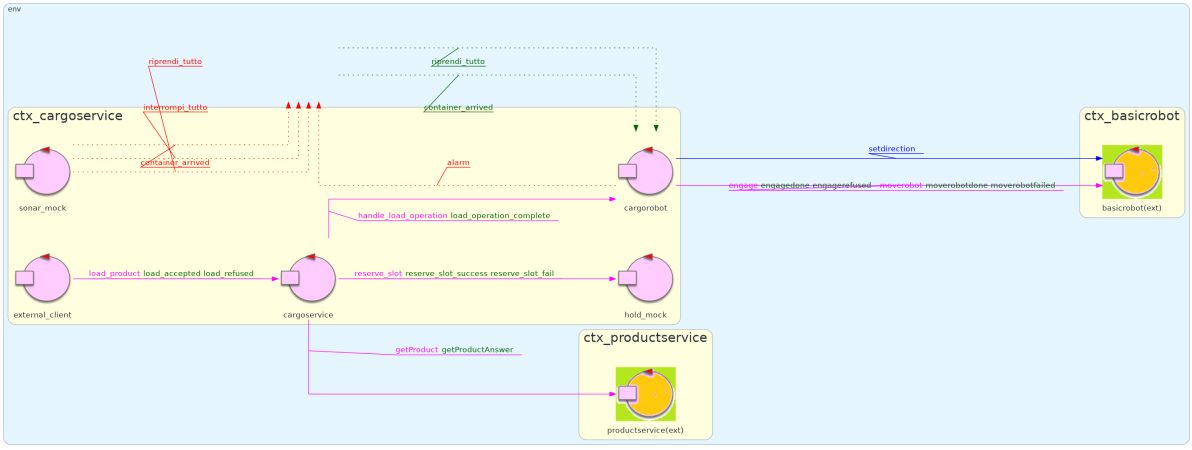
Indice

Punto di Partenza

Nello sprint 1 si sono implementati i componenti che definiscono il corebuisness del sistema: *cargoservice* e *cargorobot*.

Nel far questo si sono anche definite le interfaccie per i componenti *hold* e *sonar* da svilupparsi in questo sprint.

L'architettura del sistema risultante dallo sprint 1 è la seguente.



arch1Arch

Obiettivi

L'obiettivo dello sprint 2 sarà affrontare il sottoinsieme dei requisiti relativi ai componenti *sonar* e *hold*, 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 2 saranno i seguenti:

- implementare un sistema in grado di rilevare la presenza/assenza di un *container* presso l'*IO-port*
- implementare un sistema in grado di rilevare e gestire malfunzionamenti del sonar
- implementare un sistema in grado di tenere traccia dello stato del deposito. Questo include:
  - lo stato libero/occupato di ogni *slot*
  - il peso totale dei container caricati all'interno del deposito
- implementare un sistema in grado di condividere con la *web-gui* (o a qualunque altro componente interessato) lo stato del deposito

Sonar

Analisi del Problema

L'attore *sonar* è responsabile di effettuare **periodicamente** delle misurazioni di distanze allo scopo di rilevare la presenza dei container che arrivano all'*IO-port*.

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

1. *sonar* effettua una misurazione *m* comandando il sensore fisico.
2. *sonar* controlla in quale intervallo ricade *m*, le possibilità sono tre:
   - *0 < m < DFREE/2*
   - *DFREE/2 <= m <= DFREE*
   - *m > DFREE*
3. *sonar* considera l'intervallo a cui appartiene *m*, e le misurazioni precedenti effettuate **negli ultimi 3 secondi**, per decidere cosa fare. Le possibilità sono quattro:
   - se le misurazioni effettuate sono state consistentemente *> 0* e *< DFREE/2*, **significa che è presente un *container*** presso l'IO-port e *sonar* notifica questo evento al resto del sistema.
   - se le misurazioni effettuate sono state consistentemente *>= DFREE/2* e *<= DFREE*, **significa che NON è presente un *container*** presso l'*IO-port* e *sonar* notifica questo evento al resto del sistema.
   - se le misurazioni effettuate sono state consistentemente *> DFREE*, **significa che il sonar fisico è guasto** e *sonar* (il componente software) **emette l'evento 'interrompi_tutto'** introdotto nello sprint 1 per interrompere le attività del resto del sistema.
   - se le misurazioni effettuate NON sono state consistenti, **non si può dedurre nulla**. *sonar* non fa nulla.
4. solo nel caso in cui la misurazione corrente *m* abbia portato al passo precedente alla rilevazione di un guasto nel sonar fisico, ***sonar* cambia di stato** e attende la prima misurazione *m' < DFREE* prima di

tornare ad uno stato di corretto funzionamento. All'arrivo della misurazione *m'*, *sonar* fa ripartire il resto del sistema **emettendo l'evento 'riprendi_tutto'** introdotto nello sprint 1.

Considerazioni

Il ciclo di attività dell'attore *sonar* è divisibile in due fasi:

- fase di recupero della misurazione
- fase di processamento della misurazione

Risulta quindi possibile separare *sonar* in **due attori distinti**, uno per fase. Questo porta ad avere come vantaggio il poter **produrre misurazioni fittizie** sostituendo l'attore che recupera le misurazioni dal sonar fisico con un attore mock, oppure con una test unit, rendendo facilmente testabile la logica di processamento.

Problematiche

L'analisi fatta fino ad ora fa sorgere le seguenti domande.

Come fa *sonar* a comandare il sonar fisico per ottenere le misurazioni?

Il caro committente ha fornito uno script python che fa proprio questo. Più nel dettaglio, lo script fornito comanda i **pin GPIO** di un **Raspberry PI** a cui il sonar fisico è collegato, ottenendo **una misurazione al secondo**.

```
# File: sonar.py
import RPi.GPIO as GPIO
import time
import sys

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
TRIG = 17
ECHO = 27

GPIO.setup(TRIG, GPIO.OUT)
GPIO.setup(ECHO, GPIO.IN)
GPIO.output(TRIG, False)   # TRIG parte LOW

print ('Waiting a few seconds for the sensor to settle')
time.sleep(2)

while True:
    GPIO.output(TRIG, True)    #invia impulso TRIG
    time.sleep(0.00001)
    GPIO.output(TRIG, False)

    pulse_start = time.time()
    #attendi che ECHO parta e memorizza tempo
    while GPIO.input(ECHO)==0:
        pulse_start = time.time()
    # register the last timestamp at which the receiver detects the signal.
    while GPIO.input(ECHO)==1:
```

```
        pulse_end = time.time()
    pulse_duration = pulse_end - pulse_start

    # velocità del suono ~= 340m/s
    # distanza = v*t
    # il tempo ottenuto misura un roundtrip -> distanza = v*t/2
    distance = pulse_duration * 17165
    distance = round(distance, 1)
    print ('Distance:', distance,'cm')
    sys.stdout.flush()
    time.sleep(1)
```

Come fa *sonar* a capire se le misurazioni effettuate negli ultimi 3 secondi sono state consistenti?

È evidente che *sonar* dovrà mantenere delle informazioni nel suo stato riguardanti le misurazioni precedenti. Più nel dettaglio, *sonar* avrà bisogno di:

- una variabile che conta il numero di misurazioni consistenti effettuate.
- una variabile che indica in quale intervallo è ricaduta la misurazione precedente per capire quale intervallo considerare nel decidere se la misurazione corrente è consistente o meno.

Siccome le misurazioni vengono effettuate una volta al secondo, se il contatore raggiunge il valore 3 questo significa che le misurazioni sono state consistenti per almeno tre secondi e può quindi venire emesso l'evento corrispondente all'intervallo in cui le misurazioni sono ricadute.

Se una misurazione non è consistente, o se le misurazioni sono state consistenti per 3 secondi, il contatore viene resettato.

Messaggi

sonar emette tutti gli eventi definiti durante l'analisi di *cargorobot* fatta nello sprint 1

```
Event container_arrived : container_arrived(X)
Event container_absent  : container_absent(X)
Event interrompi_tutto  : interrompi_tutto(X)
Event riprendi_tutto    : riprendi_tutto(X)
```

Oltre a questi, siccome si è deciso di separare *sonar* in due attori distinti, si introduce un evento corrispondente ad una misurazione del sonar fisico.

```
Event measurement       : measurement(CM)
```

Modello

L'analisi fatta fino ha portato al seguente modello.

```
QActor sonardevice context ctx_iodevices {
    [#
        lateinit var reader : java.io.BufferedReader
        lateinit var p : Process
        var Distance = 0
    #]

    State s0 initial{
        println("$name | start")
        [#
            p       = Runtime.getRuntime().exec("python sonar.py")
            reader  = java.io.BufferedReader(
                        java.io.InputStreamReader(p.getInputStream())
                      )
        #]
    }
    Goto readSonarData

    State readSonarData{
        [#
            var data = reader.readLine()

            if( data != null ){
                try{
                    val vd = data.toFloat()
                    val v  = vd.toInt()

                    // filter the data maybe?
                    if(v <= 100)
                        Distance = v
                    else
                        Distance = 0
                }catch(e: Exception){
                    CommUtils.outred("$name readSonarDataERROR: $e")
                }
            }

        #]

        if [# Distance > 0 #] {
            println("$name | misurato $data cm") color yellow
            emitlocalstream measurement : measurement($Distance)
        }
    }
    Goto readSonarData
}

QActor measure_processor context ctx_iodevices {
    import "main.java.IntervalliMisurazioni"

    [#
        val DFREE = 30
        // uso degli enumerativi
```

```
        var CurrentIntervallo = IntervalliMisurazioni.PRIMA_MISURAZIONE
        var LastIntervallo = IntervalliMisurazioni.PRIMA_MISURAZIONE
        // conta quanti misurazioni di fila sono cadute nello stesso intervallo
        var CounterIntervallo = 1
        // flag che mi dice se sono in uno stato di malfunzionamento
        var Guasto = false
    #]

    State s0 initial{
        println("$name | start")
        subscribeTo sonardevice for measurement
    }
    Goto listen_for_measurement


    State listen_for_measurement {
        //aspetto
    }
    Transition t0
        whenEvent measurement -> process_measurement


    State process_measurement {
        onMsg(measurement : measurement(X)) {
            [#
                val M = payloadArg(0).toInt()
                CounterIntervallo++
            #]

            if [#  M < DFREE/2 #] {
//              println("$name | container presente") color blue // DEBUG
                [#
                    CurrentIntervallo =
                        IntervalliMisurazioni.CONTAINER_PRESENTE
                #]

                if [# Guasto #] {
                    println("$name | sonar ripristinato") color green
                    [# Guasto = false #]
                    emit riprendi_tutto : riprendi_tutto(si)
                }
            }

            if [# M >= DFREE/2 && M <= DFREE #] {
//              println("$name | container assente") color blue // DEBUG
                [#
                    CurrentIntervallo =
                        IntervalliMisurazioni.CONTAINER_ASSENTE
                #]

                if [# Guasto #] {
                    println("$name | sonar ripristinato") color green
                    [# Guasto = false #]
                    emit riprendi_tutto : riprendi_tutto(si)
```

```
                }
            }

            if [# M > DFREE #] {
//              println("$name | guasto!!!") color blue // DEBUG
                [# CurrentIntervallo = IntervalliMisurazioni.GUASTO #]
            }

            [#
                if(CurrentIntervallo==LastIntervallo &&
                   LastIntervallo!=IntervalliMisurazioni.PRIMA_MISURAZIONE)
                {
                    // switch di Kotlin
                    when(CurrentIntervallo) {
                        IntervalliMisurazioni.CONTAINER_PRESENTE -> {
                            if(CounterIntervallo == 3) {
                                CommUtils.outmagenta("Container presente
                                                      consistentemente")
            #]
                                emit container_arrived :
                                     container_arrived(si)
            [#
                                CounterIntervallo = 0
                            }
                        }
                        IntervalliMisurazioni.CONTAINER_ASSENTE -> {
                            if(CounterIntervallo == 3) {
                                CommUtils.outmagenta("Container assente
                                                      consistentemente")
            #]
                                emit container_absent :
                                     container_absent(si)
            [#
                                CounterIntervallo = 0
                            }
                        }
                        IntervalliMisurazioni.GUASTO -> {
                            if(CounterIntervallo == 3) {
                                CommUtils.outred("Guasto consistente")
                                Guasto = true
            #]
                                emit interrompi_tutto :
                                     interrompi_tutto(si)
            [#
                                CounterIntervallo = 0
                            }
                        }
                        else -> {
                            // ci vuole se no kotlin si lamenta in quanto
                            // i casi sopra non sono esausitivi
                        }
                    }
                }
                else {
```

```
                    // 1 in quanto questa è la prima misurazione
                    // appartenente al suo intervallo
                    CounterIntervallo = 1
                }

                LastIntervallo = CurrentIntervallo
            #]
        }
    }
    Goto listen_for_measurement
}
```

Piano di test

Scenario 1: container presente per 3 secondi

```
@Test
public void testContainerArrived() throws Exception {
    // versione Java dei waitgroup di Go.
    // Serve a bloccare il main thread fino a quando
    // i child thread non completano
    CountDownLatch latch = new CountDownLatch(1);
    // osservo il coap endpoint per ricevere gli eventi di reazione
    // agli eventi che genero nel test
    CoapClient client = new CoapClient(SonarTest.CoapEndopoint);
    CoapObserveRelation relation = client.observe(
        new CoapHandler() {
            @Override
            public void onLoad(CoapResponse response) {
                String content = response.getResponseText();
                CommUtils.outgreen("ActorObserver | value=" + content );

                assertTrue("TEST: container_arrived non ricevuto",
                            content.contains("container_arrived"));

                latch.countDown();
            }
            @Override
            public void onError() {
                CommUtils.outred("OBSERVING FAILED");

                fail("errore nella osservazione del sonar");

                latch.countDown();
            }
        }
    );

    // container presente per tre misurazioni
    IApplMessage measurement =
        CommUtils.buildEvent("tester", "measurement", "measurement(10)");
```

```
    conn.forward(measurement);
    conn.forward(measurement);
    conn.forward(measurement);

    // Aspetto la risposta del coap endpoint.
    // latch.await() restituisce false se scade il timeout
    boolean arrived = latch.await(5, TimeUnit.SECONDS);
    relation.proactiveCancel();
    client.shutdown();
    // verifico anche che il timeout non sia scaduto
    assertTrue("onLoad non è stato invocato entro il timeout", arrived);
}
```

Scenario 2: container presente per 3 secondi e poi assente per 3 secondi

```
@Test
public void testContainerArrivedThenAbsent() throws Exception {
    // versione Java dei waitgroup di Go.
    // Serve a bloccare il main thread fino a quando
    // i child thread non completano
    CountDownLatch latch = new CountDownLatch(2);
    // osservo il coap endpoint per ricevere gli eventi di reazione
    // agli eventi che genero nel test
    CoapClient client = new CoapClient(SonarTest.CoapEndopoint);
    CoapObserveRelation relation = client.observe(
        new CoapHandler() {
            int counter = 0;
            @Override
            public void onLoad(CoapResponse response) {
                String content = response.getResponseText();
                CommUtils.outgreen("ActorObserver | value=" + content );

                if(counter==1) {
                    assertTrue("TEST: container_arrived non ricevuto",
                        content.contains("container_arrived"));
                }
                else if(counter==2) {
                    assertTrue("TEST: container_absent non ricevuto
                        dopo container_arrived",
                        content.contains("container_absent"));
                }
                latch.countDown();

                counter++;
            }
            @Override
            public void onError() {
                CommUtils.outred("OBSERVING FAILED");

                fail();
```

```
                latch.countDown();
                counter++;
            }
        }
    );

    // container presente per tre misurazioni
    IApplMessage present_measurement = CommUtils.buildEvent("tester",
                                        "measurement", "measurement(10)");
    IApplMessage absent_measurement = CommUtils.buildEvent("tester",
                                        "measurement", "measurement(20)");

    conn.forward(absent_measurement);
    conn.forward(absent_measurement);
    conn.forward(absent_measurement);
    conn.forward(present_measurement);
    conn.forward(present_measurement);
    conn.forward(present_measurement);

    // Aspetto la risposta del coap endpoint.
    // latch.await() restituisce false se scade il timeout
    boolean arrived = latch.await(5, TimeUnit.SECONDS);
    relation.proactiveCancel();
    client.shutdown();
    // verifico anche che il timeout non sia scaduto
    assertTrue("onLoad non è stato invocato entro il timeout", arrived);
}
```

Scenario 3: rilevazione guasto e ripristino

```
@Test
public void testFaultySonarAndRecovery() throws Exception {
    // versione Java dei waitgroup di Go.
    // Serve a bloccare il main thread fino a quando
    // i child thread non completano
    CountDownLatch latch = new CountDownLatch(2);

    // osservo il coap endpoint per ricevere gli eventi di reazione
    // agli eventi che genero nel test
    CoapClient client = new CoapClient(SonarTest.CoapEndopoint);
    CoapObserveRelation relation = client.observe(
        new CoapHandler() {
            int counter = 0;
            @Override
            public void onLoad(CoapResponse response) {
                String content = response.getResponseText();
                CommUtils.outgreen("ActorObserver | value=" + content );

                if(counter==0) {
                    assertTrue("TEST: guasto non ricevuto",
                        content.contains("guasto"));
```

```
                }
                else if(counter==1) {
                    assertTrue("TEST: ripristino non ricevuto",
                        content.contains("ripristinato"));
                }

                latch.countDown();

                counter++;
            }
            @Override
            public void onError() {
                CommUtils.outred("OBSERVING FAILED");

                fail();

                latch.countDown();
                counter++;
            }
        }
    );

    // container presente per tre misurazioni
    IApplMessage guasto_measurement = CommUtils.buildEvent("tester",
                                        "measurement", "measurement(31)");
    IApplMessage recovery_measurement = CommUtils.buildEvent("tester",
                                        "measurement", "measurement(20)");

    conn.forward(guasto_measurement);
    conn.forward(guasto_measurement);
    conn.forward(guasto_measurement);
    conn.forward(recovery_measurement);


    // Aspetto la risposta del coap endpoint.
    // latch.await() restituisce false se scade il timeout
    boolean arrived = latch.await(5, TimeUnit.SECONDS);
    relation.proactiveCancel();
    client.shutdown();
    // verifico anche che il timeout non sia scaduto
    assertTrue("onLoad non è stato invocato entro il timeout", arrived);
}
```

Successivamente, si è testato il sonar anche utilizzando i seguenti attori mock.

sonar_simul

```
QActor sonar_simul  context ctx_iodevices{
    State s0 initial{
    }
    Goto work
```

```
State work{
    delay 1000 // attendo l'avvio di sonar_listener

    // misurazioni non consistenti
    emitlocalstream measurement      : measurement(30)
    delay 1000
    emitlocalstream measurement      : measurement(15)
    delay 1000
    emitlocalstream measurement      : measurement(10)
    delay 1000
    emitlocalstream measurement      : measurement(0)
    delay 1000
    emitlocalstream measurement      : measurement(40)

    // assente per 4 secondi
    emitlocalstream measurement      : measurement(20)
    delay 1000
    emitlocalstream measurement      : measurement(20)
    delay 1000
    emitlocalstream measurement      : measurement(20)
    delay 1000
    emitlocalstream measurement      : measurement(20)
    delay 1000

    // presente per 3 secondi
    emitlocalstream measurement      : measurement(10)
    delay 1000
    emitlocalstream measurement      : measurement(10)
    delay 1000
    emitlocalstream measurement      : measurement(10)
    delay 1000

    // di nuovo presente per 3 secondi
    emitlocalstream measurement      : measurement(10)
    delay 1000
    emitlocalstream measurement      : measurement(10)
    delay 1000
    emitlocalstream measurement      : measurement(10)
    delay 1000

    // guasto per 5 secondi
    emitlocalstream measurement      : measurement(31)
    delay 1000
    emitlocalstream measurement      : measurement(31)
    delay 1000
    emitlocalstream measurement      : measurement(31)
    delay 1000
    emitlocalstream measurement      : measurement(31)
    delay 1000
    emitlocalstream measurement      : measurement(31)
    delay 1000

    // di nuovo presente per 2 secondi, assente per 3
```

```
        emitlocalstream measurement      : measurement(10)
        delay 1000
        emitlocalstream measurement      : measurement(10)
        delay 1000
        emitlocalstream measurement      : measurement(20)
        delay 1000
        emitlocalstream measurement      : measurement(20)
        delay 1000
        emitlocalstream measurement      : measurement(20)
        delay 1000
    }
}
```

sonar_listener

```
QActor sonar_listener context ctx_iodevices {
    State s0 initial{
        println("$name | start")
    }
    Goto work


    State work {
        println("$name | working") color blue
        delay 1000
    }
    Transition t0
        whenEvent container_arrived -> container_arrived
        whenEvent container_absent -> container_absent
        whenEvent interrompi_tutto -> bloccato


    State container_arrived {
        println("$name | container_arrived") color green
        delay 1000
    }
    Goto work


    State container_absent {
        println("$name | container_absent") color red
        delay 1000
    }
    Goto work


    State bloccato {
        println("$name | bloccato!") color red
    }
    Transition t0
        whenEvent riprendi_tutto -> ripristinato
```

```
    State ripristinato {
        println("$name | ripristinato!") color green
    }
    Goto work
}
```

Hold

Analisi del problema

L'attore *hold* è responsabile di effettuare la prenotazione degli slot di carico nella stiva, garantendo che la capacità massima della nave (MaxLoad) non venga superata.

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

1. *hold* riceve da *cargoservice* una richiesta di prenotazione di uno slot.
2. *hold* valuta la possibilità di effettuare l'intervento di carico. Le casistiche previste sono le seguenti:
    - Se il carico cumulativo (peso del nuovo container + carico attuale nel deposito) supera MaxLoad, non è possibile caricare il container. In tal caso *hold* risponde a *cargoservice* con **reserve_slot_fail**
    - Se il carico cumulativo non supera MaxLoad ma non sono presenti slot liberi, anche in questo caso non è possibile caricare il container. In tal caso *hold* risponde al *cargoservice* con **reserve_slot_fail**
    - Se Il carico cumulativo non supera MaxLoad e vi è almeno uno slot libero è possibile procedere con l'intervento di carico. La risposta a *cargoservice* sarà **reserve_slot_success_**

Considerazioni

Dall'analisi dei requisiti si è evinto che **non è necessario implementare la casistica in cui gli slot di *hold* si liberino**.

Si può notare un legame tra *hold* e il componente *web-gui*. Già dall'analisi dei requisiti è risultato chiaro che questi due componenti dovessero interagire; ora che si è esplicitata la sequenza di attività di hold. questo è ancora più chiaro.

In particolare, la web-gui deve recuperare lo stato della stiva mantenuto da *hold* e ricevere da quest'ultimo aggiornamenti periodici quando questo stato viene modificato. *hold* è quindi anche in grado di:

1. Rispondere a query riguardanti il suo stato
2. Emettere eventi di aggiornamento quando il suo stato subisce una modifica

Messaggi

```
Request reserve_slot         : reserve_slot(WEIGHT)
Reply   reserve_slot_success : reserve_slot_success(SLOT) for reserve_slot
Reply   reserve_slot_fail    : reserve_slot_fail(CAUSA) for reserve_slot

Request get_hold_state       : get_hold_state(X)
```

```
Reply   hold_state          : hold_state(JSonString) for get_hold_state

Event   hold_update         : hold_update(JSonString)
```

Modello

L'analisi fatta finora ha portato al seguente modello.

```
QActor hold context ctx_cargoservice{
    //Variabili di stato di Hold MaxLoad:Massimo peso caricabile sulla nave ,
Currentload: Peso del carico in un dato istante , slots: presenza o assenza di
carico in un determinato slot
    [#
        var MaxLoad = 500
        var currentLoad = 0
        val slots = hashMapOf(
            "slot1" to true,
            "slot2" to true,
            "slot3" to true,
            "slot4" to true
        )

        fun getHoldStateJson(): String {
                val slotsJson = slots.map { (key, value) ->
                "\"$key\": \"${if (value) "free" else "occupied"}\""
            }.joinToString(", ")

            val rawJson = """{"currentLoad":$currentLoad,"slots":{$slotsJson}}"""

            // println("DEBUG raw JSON: $rawJson")

            return "'${rawJson.replace("'", "\\'")}'"
        }
    #]


    State s0 initial {
        println("$name | STARTS - MaxLoad: $MaxLoad kg, Slots: $slots") color
yellow

    }
    Goto wait_request


    State wait_request{
        println("$name | waiting for reservation requests") color yellow

    }
    Transition t0
       whenRequest get_hold_state -> serving_get_hold_state
       whenRequest reserve_slot -> check_reservation
```

```
    /*Verifico la presenza di slot liberi all'interno della stiva e che il MaxLoad
della nave non venga superato */

    State check_reservation{
        onMsg(reserve_slot : reserve_slot(WEIGHT)){
            [#
                val weight = payloadArg(0).toInt()
                var FreeSlot: String ?= null
                var Cause = ""

                if (currentLoad + weight > MaxLoad){
                    Cause= "Exceeds MaxLoad"

                }else{
                    FreeSlot = slots.entries.find {it.value}?.key // restituisce
la chiave del  primo elemento della entry con valore true oppure restituisce null
                    if (FreeSlot == null){
                        Cause = "All slots are occupied"
                    }
                }
            #]

            if [# FreeSlot != null #]{

                println("$name | reserving $FreeSlot for weight $weight") color
green
                [#
                    slots[FreeSlot]=false
                    currentLoad +=weight
                    val JsonState = getHoldStateJson()

                #]
                emit hold_update : hold_update($JsonState)
                replyTo reserve_slot with reserve_slot_success :
reserve_slot_success($FreeSlot)

            }else{
                println("$name | reservation refused: $Cause") color red
                replyTo reserve_slot with reserve_slot_fail :
reserve_slot_fail($Cause)

            }
        }
    }
    Goto wait_request


    //Stato dell'attore che si occupa di rispondere con lo stato iniziale del
deposito
    State serving_get_hold_state{
        onMsg(get_hold_state : get_hold_state(X)){
            [#
                val JsonState = getHoldStateJson()
```

```
            #]
            println("$name | sending hold state") color yellow
            println("$name | DEBUG wrapped   = $JsonState") color red
            replyTo get_hold_state with hold_state : hold_state($JsonState)
        }

    }
    Goto wait_request
}
```

Piano di test

Scenario 1: Test prenotazione riuscita

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

        String requestStr = CommUtils.buildRequest("tester",
                "reserve_slot", "reserve_slot(100)",
                "hold").toString();

        System.out.println("Richiesta Test 1: " + requestStr);

        String response = conn.request(requestStr);

        System.out.println("Risposta Test 1: " + response);

        assertTrue("TEST 1: prenotazione riuscita",
                 response.contains("reserve_slot_success"));
    }
```

Scenario 2: Test intervento di carico rifiutato per superamento di MaxLoad

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

        String requestStr = CommUtils.buildRequest("tester",
                "reserve_slot", "reserve_slot(600)",
                "hold").toString();

        System.out.println("Richiesta Test 2: " + requestStr);

        String response = conn.request(requestStr);

        System.out.println("Risposta Test 2: " + response);

        assertTrue("TEST 2: Exceeds MaxLoad",
                 response.contains("reserve_slot_fail") &&
```

```
                response.contains("Exceeds MaxLoad"));
    }
```

Scenario 3: Prenotazione fallita data da nessuno slot libero

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

        String state = conn.request(CommUtils.buildRequest("tester",
                "get_hold_state", "get_hold_state(X)",
                "hold").toString());

        JSONObject slots = extractSlots(state);
        for (String slot : slots.keySet()) {
            if (slots.getString(slot).equals("free")) {

                String requestStr = CommUtils.buildRequest("tester",
                        "reserve_slot", "reserve_slot(50)",
                        "hold").toString();

                 System.out.println("Richiesta Test 3: " + requestStr);

                 String response = conn.request(requestStr);

                 System.out.println("Risposta Test 3: " + response);
            }
        }  // Occupa gli slot liberi ai fini del test

        String response = conn.request(CommUtils.buildRequest("tester",
                "reserve_slot", "reserve_slot(50)",
                "hold").toString());

        assertTrue("TEST 3: prenotazione dovrebbe fallire",
                 response.contains("reserve_slot_fail"));
        assertTrue("TEST 3: motivo dovrebbe essere 'All slots are occupied'",
                 response.contains("All slots are occupied"));
    }
```

Scenario 4: Richiesta dello stato corrente del deposito

```
@Test
    public void testGetHoldState() throws Exception {
        String requestStr = CommUtils.buildRequest("tester",
                "get_hold_state", "get_hold_state(X)",
                "hold").toString();

        System.out.println("Richiesta Test 4: " + requestStr);

        String response = conn.request(requestStr);
```

```
        System.out.println("Risposta Test 4: " + response);

        assertTrue("TEST 4: stato hold restituito correttamente",
                 response.contains("hold_state") &&
                 response.contains("currentLoad") &&
                 response.contains("slots"));
    }
```

Scenario 5: Verifica aggiornamento dello stato dopo una prenotazione

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


        String stateRequest = CommUtils.buildRequest("tester",
                "get_hold_state", "get_hold_state(X)",
                "hold").toString();

        String initialState = conn.request(stateRequest);
        System.out.println("Test 5 --> Stato iniziale: " + initialState);

        int initialLoad = extractCurrentLoad(initialState);  //Carico attuale
della nave

        int reservationWeight = 150;    //Peso del container nuovo

        String reserveRequest = CommUtils.buildRequest("tester",
                "reserve_slot", "reserve_slot("+reservationWeight+")",
                "hold").toString();

        String reserveResponse = conn.request(reserveRequest);
        System.out.println("Test 5 --> Risposta prenotazione: " +
reserveResponse);

        String updatedState = conn.request(stateRequest);
        System.out.println("Test 5 --> Stato aggiornato: " + updatedState);

        int updatedLoad = extractCurrentLoad(updatedState);     //Peso dopo
intervento di carico

        assertFalse("TEST 5: stato dovrebbe essere diverso dopo prenotazione",
                initialState.equals(updatedState));
        assertEquals("TEST 5: currentLoad dovrebbe essere aumentato di " +
reservationWeight,
                 initialLoad + reservationWeight, updatedLoad);
    }
```

Progettazione

Come per lo sprint 1 la modellazione tramite il DSL QAK ha prodotto dei modelli eseguibili. In questo caso, non c'è stato addirittura bisogno di una fase di progettazione. I componenti modellati sono già soddifacenti nella loro forma da modello eseguibile.

Deployment

I modelli QAK sviluppati in questo sprint sono recuperabile alla seguente repository github, dentro alle cartella "system2/" e "iodevices/".

Per avviare il progetto:

1. eseguire `docker compose -f arch2.yaml up` per lanciare i componenti relativi alla logica di buisness del sistema
2. aggiungere qualche prodotto al db mongo appena avviato, eseguendo con node lo script setupmongo.js
3. nel lanciare i componenti relativi ai dispositivi di I/O si hanno due alternative
    - se si ha a disposizione un raspberry pi, e quindi dei dispositivi fisici da comandare, si può copiare la distribuzione ottenuta con `./gradlew run` all'interno del raspberry e lanciarla al suo interno
    - se non si ha a disposizione un raspberry si possono lanciare i simulatori con `./gradlew run`
4. aprire il browser e digitare `localhost:8090` per visualizzare l'ambiente virtuale WEnv e il robot che effettua i suoi interventi di carico

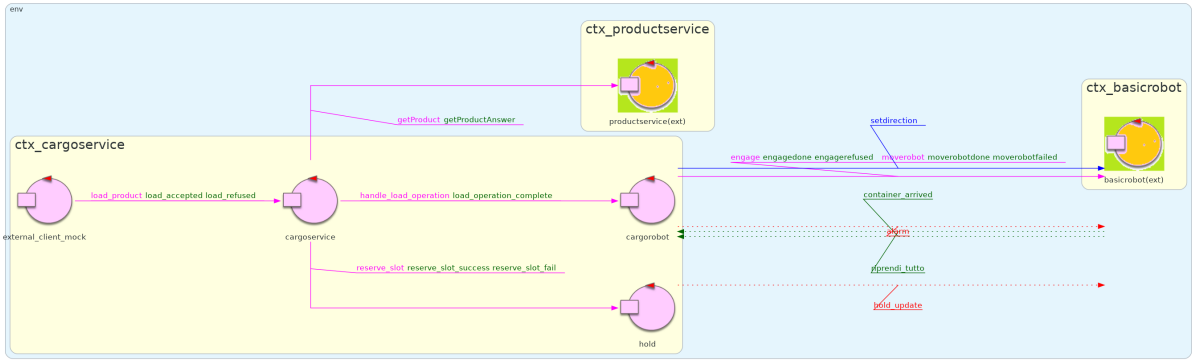
Sintesi Finale e Nuova Architettura

In questo sprint 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.

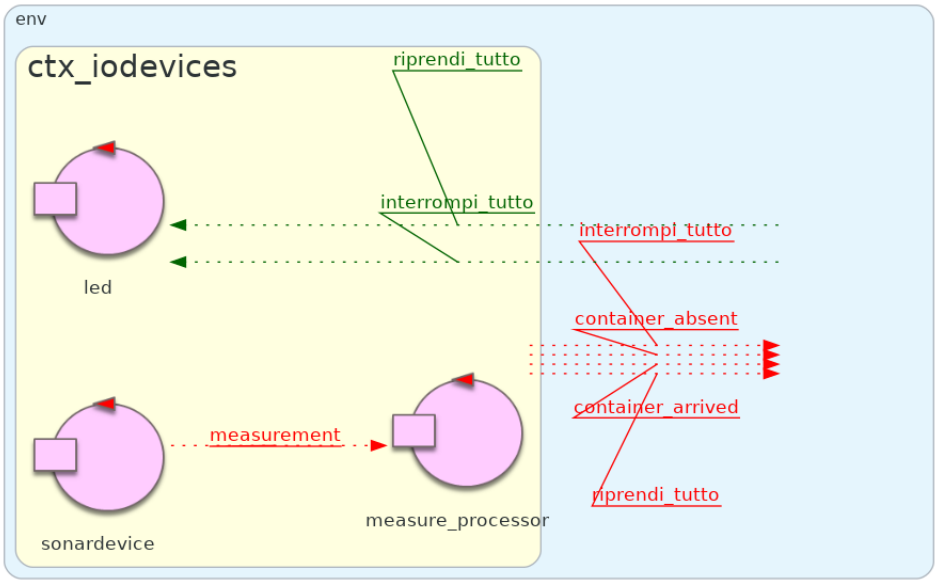
L'architettura del sistema risultante da questo sprint è suddivisibile in due macrocontesti.

Servizio principale



arch2Arch

Dispositivi di I/O



iodevicesArch

Tempo Impiegato e Ripartizione del Lavoro

Tempo Impiegato

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

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.