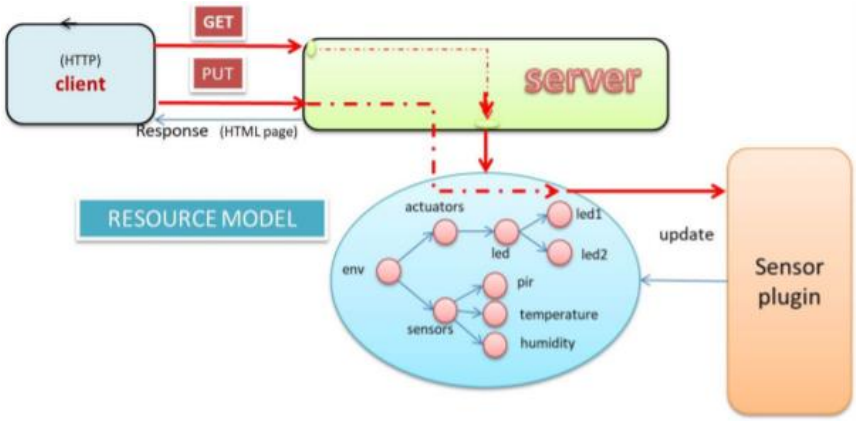
Un Frontend server -3-

Obiettivo: costruire il prototipo di un sistema software la cui architettura è ispirata al modello HTTP/REST adottato nel Web Of Things (WoT). L'architettura logica informale è questa:



Nel WoT i dispositivi e i loro servizi sono pienamente integrati nel web perché usano gli stessi standard e tecniche come le tradizionali websites. Possiamo scrivere applicazioni che interagiscono con servizi integrati nella stessa maniera di interagire con qualsiasi altro servizio che usa web API, tipo le architetture RESTful. Gli aspetti principali che qualificano questo tipo di architettura sono:

- **Integration patterns**: le cose possono essere integrate nel web in modi diversi: usando REST per mezzo di Gateway o server che usino il cloud (MQTT)
- **Resource model design**: ogni cosa deve poter offrire funzionalità che possano essere modellate e organizzate in una gerarchia (di solito le risorse fisiche sono mappate in risorse REST con descrizioni JSON)
- **Representation design**: ogni risorsa può essere associata con rappresentazione come JSON, HTML, ecc.
- **Interface design**: ogni servizio può essere usato con un insieme di comandi (nel modello REST i comandi sono verbi http, come GET, PUT, ecc.)
- **Resource linking design**: Le risorse devono essere logicamente collegate nel web.

Ora ci occupiamo di:

1. Sviluppare il frontend server con Node.js e Express producendo un ambiente strutturato in *model*, *control* e 1 o più *views.*
2. Definire il codice dell'entry point del server con i pattern di Express. L'entry point è un server http in cui carichiamo la logica applicativa (una richiesta piò essere http://localhost:3000/pi/sensors/temperature)
3. Poi definiamo il codice dell'applicazione secondo Express
4. O modelliamo il resource model in JSON e introduciamo dei sensor/actuator plugins che simulano le risorse o usiamo il server come frontend per l'applicazione e i plugins publlicheranno nel topic unibo/qasys MQTT messaggi del tipo

```
msg(ctrlEvent,event,js,none,ctrlEvent(leds, led1, VAL,1)      //for actuators (Led)
msg(inputCtrlEvent,event,js,none,inputEvent(temperature, t1, VAL,1) //for sensors (Temperature)
```

Definisce quindi la logica applicativa in applCode.js con i pattern di Express con una funzione per gestire le richieste, gli eventi e le eccezioni.

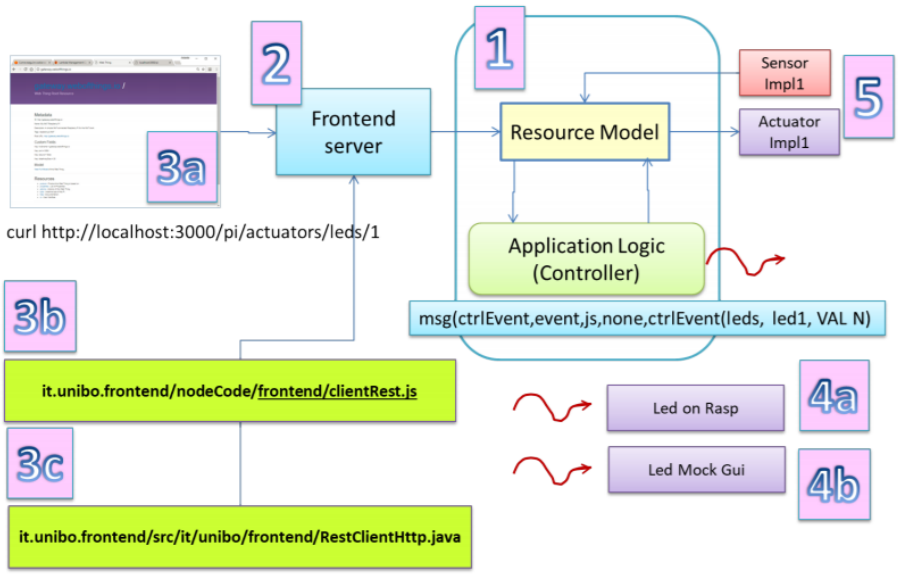
ApplCode.js invece è il codice dell'applicazione che fa routing cioè associa a un indirizzo una funzione che lo gestisca. Quindi i sensori saranno gestiti da appServer/routes/sensor.js che si occuperà semplicmente di ritornare il valore del modello mentre gli attuatori da appServer/routes/actuators.js che può modificare il modello e propagare l'informazione che il valore di un led è stato modificato.

A sto punto abbiamo un server che emette informazioni via http riguardo al cambiamento dello stato di un attuatore. Quando il server gestisce un comando PUT del genere:

```
curl -H "Content-Type: application/json" -X PUT -d "{\"value\": \"false\" }"
     http://192.168.43.229:3000/pi/actuators/leds/1
curl http://192.168.43.229:3000/pi/actuators/leds/1 (to read)
```

Emette un comando ctrlEvent : ctrlEvent(leds,led1,V). Il led come abbiamo visto riesce a reagire a questo evento con javaRun.

Per ora abbiamo fatto un sistema così:



1. Resource model definito in Prolog o JSON + application logic riferita a quel modello
2. Un server frontend REST per leggere o modificare il resource model
3. Clienti HTTP - REST(umani o macchine) possono aceddere al modello
4. Le risorse fisiche possono essere mock o concreti
5. Implementazioni di sensori e attuatori

Modellare in JSON

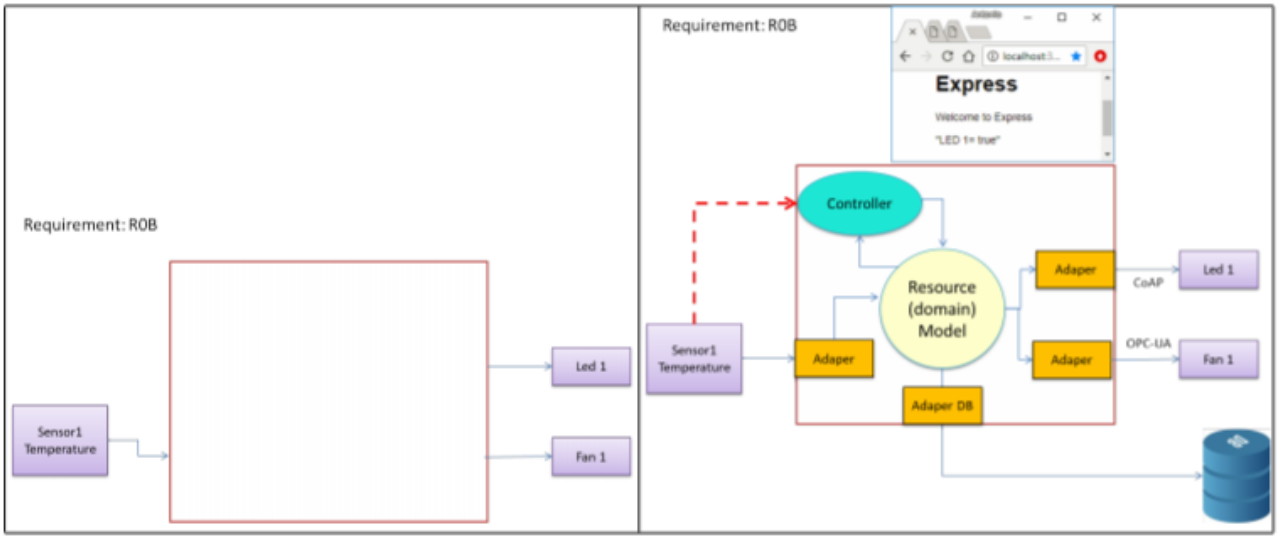
Per modellare sensori e attuatori si può scrivere un file JSON, come nell'estratto riportato qui sotto:

```
"sensors": {
  "temperature": {
    "name": "Temperature Sensor",
    "description": "An ambient temperature sensor.",
    "unit": "celsius",
    "value": 0,
    "gpio": 12
  },
  "humidity": {
    "name": "Humidity Sensor",
    "description": "An ambient humidity sensor.",
    "unit": "%",
    "value": 0,
    "gpio": 12
  },
  "pir": {
    "name": "Passive Infrared",
    "description": "A passive infrared sensor. When 'true' someone is present.",
    "value": true,
    "gpio": 17
  }
},
"actuators": {
  "leds": {
    "1": {
      "name": "LED 1",
      "value": false,
      "gpio": 25
    },
    "2": {
      "name": "LED 2",
      "value": false,
      "gpio": 9
    }
  }
```

Node on Rapsberry -4-

L'obiettivo è far blinkare un led del raspberry, quindi procede creando due file javaScript lanciati dal frontend server, uno che simuli il cambiamento dello stato del led o addirittura farlo nel vero rapsberry se il server è su quest'ultimo, l'altro per simulare cambiamenti di temperatura e umidità.

Possiamo quindi ridisegnare il sistema riguardo al requisito R0b della sezione 1 così:



Un nuovo robot virtuale – 5-

Parla del robot di Soffritti, la cui web page è servita dal server Node.js sulla porta 8999. Fa vedere un esempio di client js che invia e riceve messaggi con una connessione TCP.

Il servizio di Soffritti può essere modellato con un qactor **weGuiExecutor**. Il servizio gestisce:

- i dispatch moveRobot e altri eventi
- mappa gli eventi usercmd in dispatch moveRobot.

A quel punto il QActor player trasforma un dispatch moveRobot in chiamate TCP al Server 8999.

```
Plan execMove resumeLastPlan[
    printCurrentMessage;
    onMsg moveRobot : usercmd( robotgui(h(X)) ) ->
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'alarm' }");
    onMsg moveRobot : usercmd( robotgui(w(X)) ) ->
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveForward', 'arg': -1 }");
    onMsg moveRobot : usercmd( robotgui(s(X)) ) ->
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveBackward', 'arg': -1 }");
    onMsg moveRobot : usercmd( robotgui(a(X)) ) ->
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'turnLeft', 'arg': 800 }");
        onMsg moveRobot : usercmd( robotgui(d(X)) ) ->
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'turnRight', 'arg': 800 }");
    onMsg moveRobot : usercmd( robotgui(x(X)) ) -> forward robotpfrs -m startAppl : startAppl(go)
]
```

Arg: -1 sta per indicare un'azione che non termina e l'ultimo comando indica il forward del messaggio a un attore robotpfrs che fa alcuni lavori specifici per l'applicazione.

Nello stesso .qa c'è anche un altro attore sonarguidetector che gestiste gli eventi emessi dal Server8999:

```
QActor sonarguidetector context ctxWebGuiExecutor {
Rules{
    checkEnd( D ) :-
        eval( lt(D,5) ),
        output( checkEnddddddddddddddddddddddddddddddddddddddddddddddddddddddd( D ) ),
        assert( endreached ).
 }
    Plan init normal [ println("sonarguidetector STARTS ") ]
    switchTo waitForEvents

    Plan waitForEvents[
        [ ?? endreached ] forward robotpfrs -m moveRobot : usercmd( robotgui(h(low)) )
        ]
    transition stopAfter 3600000  //1h
        whenEvent sonar : sonar(sonar2, TARGET, DISTANCE ) do demo checkEnd(DISTANCE), // println( sonar(NAME, TARGET,
             DISTANCE ) ),
        whenEvent sonarDetect : sonarDetect( TARGET ) do println( sonarguidetector( TARGET ) )
    finally repeatPlan
}
```

Viene definita anche una classe clientTcp.java che il server usa per interagire con il robot virtuale. Questa offre anche il metodo sendMesg che abbiamo visto nell'immagine prima per inviare un messaggio JSON a in una certaSocket (indirizzo e porta vengono definiti con un metodo initClientConn). La stessa classe offre anche un modo per ricevere informazioni legate ai sonar e le collisioni dal server e generare di conseguenza i corrispettivi eventi sonar e sonarDetected.

Dopo aver fatto ciò, si può interagire con il robot virtuale.

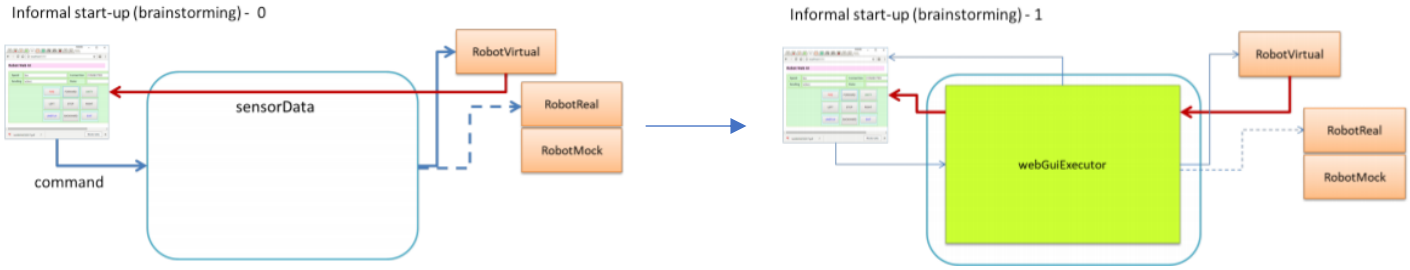
Ci accorgiamo che webGuiExectuor può essere usato anche per applicazioni più complesse!

Il nuovo problema diventa: "*pensare a un robot come una "smart thing", costruire un sistema software che permette di inviare comandi al robot e ricevere dati da esso o dall'ambiente*". Prima di partire a scrivere codice osserviamo che:

- il robot logicamente è un interprete di comandi e un emittore di informazioni (sensor)
- un robot è una cosa logica o fisica che esegue software dotato dal cliente. Usualmente il robot può essere modellato come un POJO che offre API come un insieme di procedure. Un esempio è il REAL ROBOT.

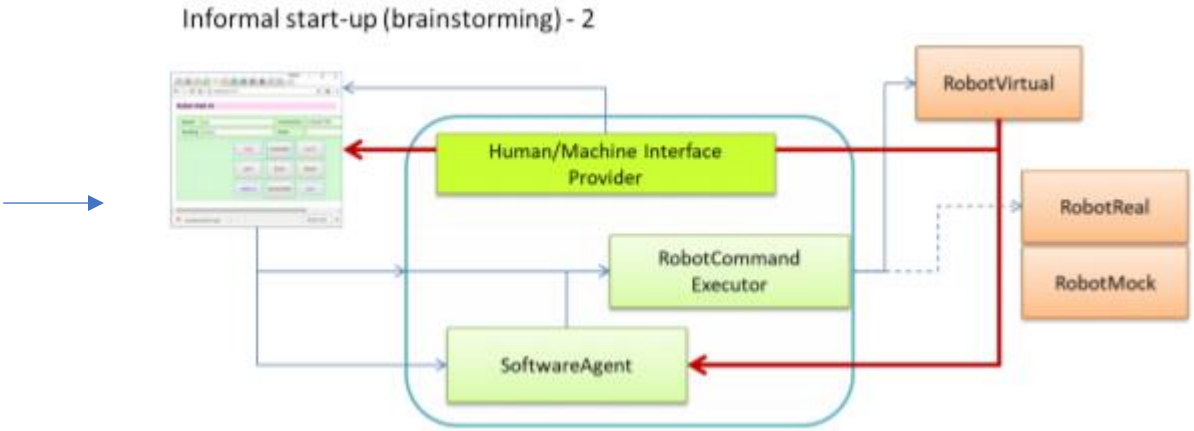
- Nel nostro sistema un robot dovrebbe essere modellato come un'entità che offre API remote che permettono allo sviluppatore di manipolarlo con dei controlli, senza fregarsene del linguaggio di implementazione. Un esempio è il VIRTUAL ROBOT.
- Il sistema dovrebbe includere un supporto per l'interazione sia via macchina sia via uomo con il robot.

Queste considerazioni portano a questo iniziale modello informale dell'architettura logica del sistema.



Il webGuiExecutor permette al team di definire un prototipo funzionante in un tempo ragionevole per sistemare i requisiti tra cliente e utenti finali.

Un ulteriore passo può essere quello di fornire una parte "technology-independent" dedicata alla logica applicativa che puà essere facilmente adattata alle parti "technology-dependent". In quest'ottica il webGuiExecutor può essere visto come un adapter (**RobotCommandExecutor**) tra l'application agent di alto livello e il robot virtuale. Lo **Human/MachineInterface Provider** invece inietta nel sistema un componente più avanzato provvisto dall'infrastruttura qactor che vediamo nel prossimo capitolo.



Un server frontend per i robot -6-

Il nuovo requisito è:

> **RobotRest**: Thinking of a robot as a '*smart thing*' (in the domain of **IOT**), provide a **RESTful** interface to send commands to the robot.

Quindi:

1. Definiamo un modello del robot
2. Definiamo un client per il robot virtuale e testiamolo
3. Definiamo in Node/Express un server frontend per il controllo del robot virtuale
4. Aggiungiamo una parte per l'autenticazione del cliente

Il modello per il robot può essere scritto in JSON:

```
{
  "robot": {
        "name": "WoT Robot",
        "description": "A simple robot model",
        "port": 8484,
        "state": "stopped",
        "sonar1": {
            "name" : "sonar1",
            "value": 0
        },
        "sonar2": {
            "name" : "sonar1",
            "value": 0
        },
        "sonarRobot": {
            "name" : "sonarRobot",
            "value" : 0
        }
    }
}
```

Il server frontend per il robot virtuale è un server http che supporta l'interazione pub-sub. È un file js che ha una funzione showResourceModel che restituisce lo stato del resource model del robot a una determinata frequenza inviandolo nella socket.

Il codice dell'applicazione applCodeRobot.js viene scritto usando EJS per generare pagine HTML con JavaScript. In particolare definisce la variabile externalActuator che se vera significa che il server delegherà delle azioni richieste ad altre applicazioni esterne pubblicando un messaggio MQTT e la variabile withAuth che se vera significa che il server permette l'accesso a un gruppo di utenti registrati.

Poi fa vedere come realizzare la pagina ejs per inviare comandi di movimento al robot.

I comandi sono action resources rappresentati così:

```
/robot/actions/commands/CMD, with
    CMD = appl | w | s | a | d | h
```

Il codice applicativo se externalActuator è falso eseguirà immediatamente l'azione, sennò la delega.

L'immediata esecuzione sfrutta clientRobotVirtual.js per inviare un comando al robot virtuale. Lo stesso può anche ricevere dati dal robot virtuale.