Alma Mater Studiorum
Università di Bologna

Scuola Di Ingegneria e Architettura

Dipartimento Guglielmo Marconi - DEI

Ingegneria Elettronica

Tesi di Laurea

in

Ingengeria Elettronica per l'Energia e l'Informazione

Interfacciamento di un sistema embedded Arduino Yùn alla piattaforma semantica SEPA

Autore:
Lucafrancesco
Raduzzi

Relatore:
Prof. Luca Roffia
Correlatori:
Dott. Simone Sindaco

Anno Accademico 2020/2021
Sessione III

Indice

Introduzione

Nell'ultimo decennio, il panorama dell'Elettronica è andato incontro a un notevole sviluppo. Tutto ciò ha permesso ai dispositivi elettronici di diventare sempre più presenti all'interno della realtà che viviamo continuamente.

Una delle conseguenze di tale cambiamento è la nascita dell'Internet of Things che consiste nell interfacciare oggetti appartenenti alla quotidianità con il mondo di Internet, rendendoli in grado di interagire con noi o di darci delle informazioni di vario tipo provenienti che ci circonda, come per esempio il valore di alcune grandezze fisiche quali la temperatura, l'umidità o la qualità dell'aria. La peculiarità di tali sistemi è quella di poter poi condividere i dati in questione via internet, inviandoli a un server o a un dispositivo in particolare (come uno smartphone o un PC) rendendo così l'oggetto interconnesso con la rete. Una delle principali caratteristiche legate a un dato è come esso venga rappresentato agli utenti, ovvero il modo in cui una persona riesce a visualizzare effettivamente un valore calcolato dal proprio device. Prendiamo come esempio la temperatura: un sensore misura la temperatura presente in una stanza e, per esempio, la visualizza su una pagina internet. Così facendo stiamo creando un Web client (la pagina internet) il quale richiede un dato (temperatura) a un device (sensore), il quale risponderà inviando dei pacchetti (HTTP POST per esempio) all'indirizzo IP della nostra pagina WEB. E' facile intuire da questo esempio, come oggigiorno la rappresentazione di un dato, sia quindi diventata così importante da discriminarne la sua qualità e di come la visualizzazione dei dati richieda un grande impiego di risorse dal punto di vista Web. Pensiamo ora a un altro esempio: un utente, anziché sapere il valore di temperatura, vorrebbe che l'impianto di riscaldamento della propria casa si accenda automaticamente quando la temperatura scende sotto i 10 °C; in questa situazione il sensore non ha necessità di visualizzare il dato, dal momento che nessuno è interessato a sapere la temperatura della

casa, tranne l'impianto di riscaldamento al quale però interessa solamente la natura del dato, il suo valore, non come viene rappresentato. Così facendo, l'intero sistema automatizza la gestione del valore di temperatura correlandolo direttamente con l'attivazione dell'impianto di riscaldamento; in questo modo è possibile notare come la comunicazione tra questi devices abbia automatizzato un processo, rendendo possibile non solo la semplificazione di alcune operazioni da parte dell'uomo ma anche aumentandone la reattività. In questo contesto, la caratteristica più importante di un dato, non sarà più la rappresentazione, bensì la sua natura, nasce così il termine di Semantic Web. Per trattare i dati semanticamente ci sono svariate sintassi e tecnologie. Quelle che verranno approfondite in seguito sono il modello RDF, tramite il quale vengono organizzati i dati, e il linguaggio SPARQL il quale permette di interrogare i sistemi sulla base delle RDF: il nostro obiettivo è stato quello di interfacciare il SEPA (un server PublishSubscribe basato su SPARQL) con un semplice sistema embedded quale è Arduino Yùn, dove per semplice si intende dotato di una capacità di calcolo relativamente bassa.
Nel primo capitolo affronteremo l'architettura di rete sulla quale si basa il server SEPA, illustrando le tecnologie proprie di questo ambiente quali Resource Description Framework e SPARQL Protocol. Nel secondo capitolo sarà presentato l'ambiente Arduino, concentrandosi specialmente sul modello Arduino Yun Rev. 2, e dimostrando le sue caratteristiche fisiche e tecniche. Infine nel terzo capitolo verrà illustrato un case study tramite il quale sarà dimostrata l'applicazione di tali tecnologie ad una situazione pratica: qui si potranno trovare i codici e le librerie utilizzate per l'implementazione dell'esempio di funzionamento.

Capitolo 1

Architettura di Rete

1.0.1 Semantic Web

Nella società odierna, la concezione di "dato" è fortemente legata a quella di "rappresentazione", uno dei principali obbiettivi dei software infatti, è quello di rappresentare i dati, renderli fruibili e facilmente interpretabili dall'essere umano: basti pensare all HyperText Markup Language (HTML), un linguaggio di programmazione interamente volto alla rappresentazione di pagine Web.
Ultimamente però è nata l'esigenza di dare la possibilità ad agenti software e macchine, di navigare all'interno dei dati, in modo da poter cooperare meglio tra di essi e con l'uomo. Con questo obiettivo nasce il Web Semantico, il quale non è un'alternativa al Web, bensì una espansione: esso nasce dalla struttura di Internet ma anziché rappresentare i dati (visualizzare pagine web tramite HTML5 per esempio) si pone l'obiettivo di attribuire loro un significato semantico in modo che diventino più gestibili da parte delle macchine. Nella concezione di Web Semantico i dati vengono collegati tra di loro associandogli delle proprietà di collegamento che indicano le relazioni che li interconnettono: in questo modo le macchine, una volta che conoscono tali relazioni, sono in grado di navigare autonomamente ed automaticamente all'interno di una moltitudine di dati. L'adozione delle tecnologie basate sul Semantic Web ha due principali obiettivi: il primo è quello di esprimere il proprio potenziale automatizzando sempre più alcuni processi e migliorando l'attività degli Autonomous Systems, mentre la seconda si concentra su soluzioni basate sulla flessibilità del modello, ovvero al modo in cui vengono gestiti i dati (Resource Description Framework, RDF). L'uso del Semantic Web negli Autonomous

Systems permette alle macchine di generare, pubblicare ed utilizzare nuove informazioni autonomamente, a partire dalle più semplici applicazioni IoT. Allo stesso tempo, l'utilizzo di un modello dotato di una grande flessibilità (RDF data model) è pensato per intervenire in quelle situazioni in cui è difficile definire il modello dei dati. Mentre i database relazionali non forniscono intrinsecamente questo livello di flessibilità, essendo per lo più degli ambienti "statici", l'RDF data model garantisce un'evoluzione della conoscenza di base del modello, aggiornando continuamente la struttura RDF: tutto ciò dona al sistema una dinamicità mai vista prima. Al giorno d'oggi l'eterogeneità semantica dei dati prodotti e consumati sul Web è davvero alta, basti pensare al fatto che oramai, qualsiasi tipo di informazione è fruibile via internet e corrisponde a un insieme di bit. Da questa prospettiva, l'utilizzo di tale modello, può essere ampliato: possiamo passare dalle piccole applicazioni IoT, ai grandi Big Data ottenendo gli stessi benefici espressi precedentemente ma più in grande. Il Semantic Web per i Big Data da una parte permette ai software di muoversi autonomamente all'interno di più siti, garantendo un arricchimento dell'informazione finale data all'utente, in modo totalmente autonomo. Dall'altra garantisce ancora più dinamicità di quanto il Web non ne avesse prima. Alla base di questo ragionamento abbiamo il concetto di RDF il quale permette di organizzare le strutture dei dati secondo precise relazioni dalle quali è possibile ottenere i benefici computazionali sopra elencati.
Berners-Lee et al. [2001]

1.0.2 RDF Data Model

RDF (Resource Description Framework) è un formato che permette di rappresentare i dati sul Web basandosi sulla loro natura e sulle relazioni che intercorrono tra di essi. RDF è pensato per quelle situazioni in cui i dati non hanno necessità di essere visualizzati e devono essere processati nel modo più veloce e computazionalmente semplice possibile, come per esempio succede nei calcolatori. W3C [2004]

Questa formato può essere sfruttato per varie finalità:

- Processing automatico dei dati da parte di applicazioni terze, dal momento che utilizza informazioni leggibili da macchine.

- Creare aggregati di dati attorno a specifici argomenti: semplificando notevolmente il modo di accesso ai dati, potremmo arricchire il dataset di alcune informazioni collegandole tra di loro (per esempio collegando ai quadri, le liste degli artisti correlati, il movimento al quale appartiene l'autore ecc..) creando delle vere e proprie strutture di dati correlati tra loro.

- Collegare le API e facendo in modo che il client possa accedere a più informazioni contemporaneamente in modo più semplice, anche grazie alla comunicazione tra database.

- Creare dei collegamenti all interno di un'organizzazione facendo in modo di poter ottenere i dati utilizzando delle queries attraverso l'uso di SPARQL.

Come è possibile vedere in Figura 1.1 RDF permette di fare delle asserzioni (Statement) seguendo la struttura delle triplette: ogni dichiarazione esprime un rapporto (chiamato predicato) tra due risorse (chiamate oggetto e soggetto); il predicato quindi esprime la natura della relazione che lega le due risorse.

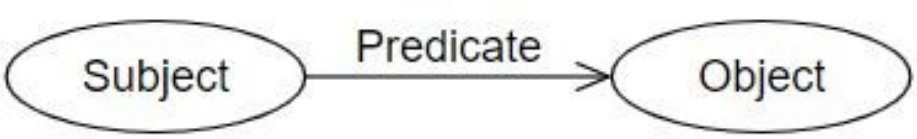

Figura 1.1: Struttura di una Tripletta

E' possibile visualizzare le triplette in grafi all'interno dei quali i soggetti e gli oggetti rappresentano i nodi, mentre i predicati rappresentano gli archi orientati; ci possono essere tre tipi di nodi: IRIs, literals e blank nodes:

- Gli IRIs (International Resource Identifier) sono una generalizzazione degli URI, dal momento che accettano anche caratteri non ASCII, e identificano una risorsa senza specificarne locazione o modalità di accesso. Essi possono occupare ciascuna delle tre posizione di una tripletta, dal momento che sono identificatori globali, ovvero qualsiasi persona può utilizzare lo stesso IRI per indicare solo e soltanto l'oggetto a cui esso è univocamente assegnato.

- I literals sono valori base non IRI, utilizzati per esprimere stringhe, date, numeri ecc... Essi possono apparire solamente come oggetti ed insieme ai precedenti permettono la scrittura delle triplette RDF.

- I blank nodes sono paragonabili (in algebra) a delle semplici variabili: rappresentano qualcosa, senza dire espressamente quale sia il suo valore (come per esempio un oggetto sullo sfondo di una figura)

L'obiettivo principale di RDF è quello di riuscire a fondere informazioni utili provenienti da risorse multiple con l'obiettivo di formare una collezione di dati più grande e dinamica: da ciò ne deriva un grande vantaggio dal momento che i sistemi possono fare deduzioni logiche basate sulle relazioni che intercorrono tra i nodi (predicati) aumentando notevolmente l'autonomia e la dinamicità del sistema preso in considerazione. Una volta definite una serie di triplette all'interno di un calcolatore, il sistema è in grado di intuire quali altre triplette, sulle quali nulla è stato detto, sono vere attraverso una deduzione logica basata sulle proprietà che intercorrono tra i nodi, ovvero i predicati. Una volta avuti i dati organizzati in questo può essere usato SPARQL per fare delle queries e ottenere un dato in particolare con molta facilità, inoltre una volta inserito un nuovo nodo dichiarandolo attraverso una tripletta, esso, in base alla sua natura entrerà a far parte della struttura già presente, legandosi semanticamente ai dati precedenti attraverso le proprietà preesistenti, ampliando il contenuto del Dataset e soprattutto rendendo dinamico l'intero ambiente.
Nella Figura 1.2 si può vedere un esempio di RDF Dataset, ovvero l'insieme di due grafi RDF i quali combinano diversi tipi di informazioni, con diversi significati semantici, ma tutte interconnesse tra loro attraverso asserzioni logiche che ne definiscono il rapporto. Nel caso in cui un grafo possa essere identificato con un IRI(il quale è visibile sopra di esso), esso prende il nome di Named Graph,in caso opposto esso prende il nome di Default Graph il quale è il grafo all'interno del quale vengono reindirizzate le queries e gli updates nell'ipotesi in cui l'IRI del grafo al quale sono indirizzate tali operazioni non sia presente nel Dataset. W3C [2014a]
Dalla Figura 1.2 è possibile identificare tutti e tre gli elementi sopra descritti:

- Ogni cerchio rappresenta un nodo, in particolare quelli in cui entra la freccia sono gli oggetti, quelli da cui parte sono i soggetti.

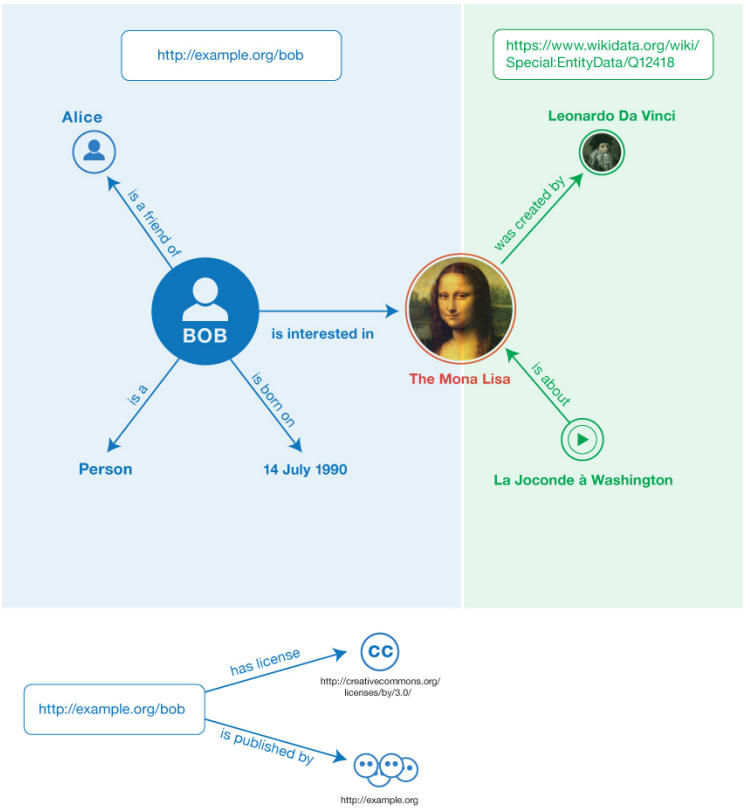

Figura 1.2: Un esempio di RDF Dataset dato dall'insieme di due Grafi W3C [2014a]

- Ogni arco rappresenta un predicato e, come detto precedentemente, ciascuno di essi rappresenta la relazione che intercorre tra il soggetto e gli oggetti.
- Infine possiamo riconoscere la struttura di un grafo formato dagli elementi sopracitati.
- Gli IRI corrispondono rispettivamente ai due Named Graphs

Se volessimo aggiungere un nodo (dato) basterebbe inserire una nuova tripletta che abbia una relazione con i nodi già presenti, per esempio: "Persona (Soggetto) è un (predicato) essere vivente(oggetto)".
Approfondendo ulteriormente, un grafo RDF non è altro che un'insieme di statements che ne definiscono il contenuto; essi possono essere scritti secondo svariate sintassi tra le quali troviamo N-Triplets,TriG, Turtle, JSON-LD, ecc

non sono altro che modi differenti di scrivere il grafo. Il più semplice e intuitivo è sicuramente N-Triplets:

```
<http://example.org/bob#me> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
<http://example.org/bob#me> <http://xmlns.com/foaf/0.1/knows> <http://example.org/alice#me> .
<http://example.org/bob#me> <http://schema.org/birthDate> "1990-07-04"^^<http://www.w3.org/2001/XMLSchema#date> .
<http://example.org/bob#me> <http://xmlns.com/foaf/0.1/topic_interest> <http://www.wikidata.org/entity/Q12418> .
<http://www.wikidata.org/entity/Q12418> <http://purl.org/dc/terms/title> "Mona Lisa" .
<http://www.wikidata.org/entity/Q12418> <http://purl.org/dc/terms/creator> <http://dbpedia.org/resource/Leonardo_da_Vinci> .
<http://data.europeana.eu/item/04802/243FA8618938F4117025F17A8B813C5F9AA4D619> <http://purl.org/dc/terms/subject> <http://www.wikidata.org/entity/Q12418> .
```

Figura 1.3: Esempio di linguaggio usato per scrivere RDF Graphs

Come possiamo notare dalla Figura 1.3 esso non è altro che una successione di triplette, scritte secondo un particolare linguaggio, le quali insieme formano il grafo rappresentato nella Figura 1.2. Ogni riga rappresenta una tripletta, dove il primo IRI (ciascuno racchiuso entro le $< >$) rappresenta il soggetto della tripletta, il secondo il predicato e il terzo l'oggetto; il punto alla fine della tripla rappresenta la fine di essa. Nella riga 3 vediamo un esempio di literal (una data in questo caso): esso è seguito dagli apici * e dalla convenzione della data XML. W3C [2014b]
In conclusione, considerando la struttura appena esposta, possiamo riassumere tali concetti dicendo che:

- Triple RDF: statements che descrivono il dato che vogliamo inserire, seguono il modello soggetto,predicato e oggetto.

- RDF Graph: insieme dinamico di triple RDF (dove soggetto e oggetto sono chiamati "nodi del grafo" e il predicato "archi del grafo") interconnesse tra loro, si dividono in Named Graphs (se è possibile identificarli con un IRI) e Default Graph (non hanno un IRI che li identificano, ricevono le richieste senza destinazione).

- RDF Terms: possibili valori di un nodo del grafo, possono essere IRI, literals o blank nodes (nodi vuoti).

- RDF Dataset: insieme di più RDF Graph.

- RDF Document: documento che codifica secondo una precisa sintassi un RDF Graph.

1.0.3 SEPA

Il SEPA(SPARQL Event Pocessing Architecture) è un'architettura dati Publish-Subscribe creata per supportare l'interoperabilità tra informazioni di più livelli. Essa è basata sul modello di dati RDF precedentemente introdotto e sfrutta il linguaggio SPARQL per gestirle; essendo una struttura basata sul concetto di Web of Data, esso è fortemente legato al campo semantico dei dati interpretandoli e comunicando i cambiamenti delle triple RDF attraverso un sistema di notifiche. Roffia et al. [2018a]
Osservando la Figura 1.4 è possibile capire come il SEPA utilizzi le funzionalità proposte dallo SPARQL 1.1 Protocol e le estende andando a creare un nuovo tipo di protocollo chiamato SPARQL 1.1 SE Protocol.

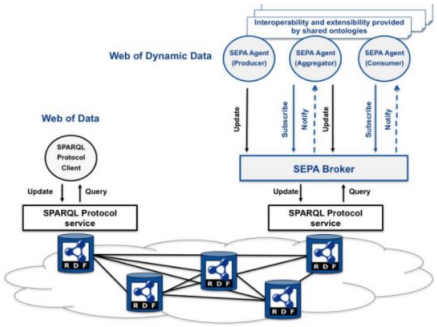

Figura 1.4: L'architettura del SEPA Roffia et al. [2018a]

Esso è configurabile attraverso dei file JSAP, scritti secondo la sintassi JSON, i quali contengono i parametri fondamentali sulla base dei quali agisce il server, come per esempio l'indirizzo IP sul quale si trova, i numeri di porta ai quali inoltrare le richieste ecc..
Il SEPA presenta una Dashboard attraverso la quale è possibile configurarlo e gestirlo, attuando le operazioni fondamentali (Query,Update e Subscribe) di modo da monitorare i cambiamenti attuati su di esso.
Dalla Figura 1.5 è possibile notare come siano presenti quattro sezioni (non considerando la sezione "Help" all'interno della quale sono presenti dei documenti contenenti nozioni teoriche riguardanti il SEPA): nella prima è possibile inserire i parametri configurativi manualmente oppure caricare il file JSAP attraverso il quale vengono impostati automaticamente. Nella seconda, terza e quarta sezione sono presenti le operazioni fondamentali che l'utente può fare inserendo le stringhe SPARQL che ne identificano il funzionamento.

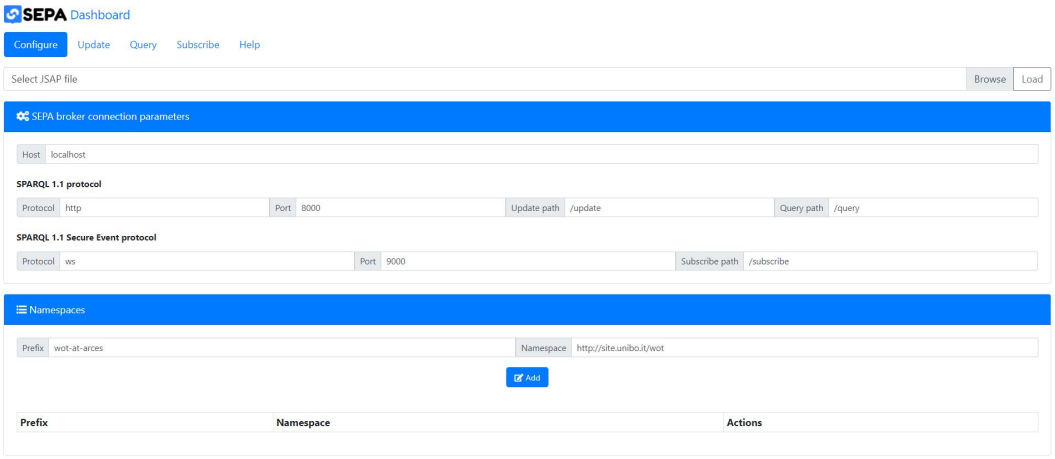

Figura 1.5: Dashboard del SEPA

SPARQL 1.1 Protocol: Query e Update

Lo SPARQL1.1 Protocol è uno standard riconosciuto dalla W3C che ha come obiettivo quello di gestire l'ambente dati RDF fornendo due operazioni che permettono l'interazione con questo sistema: la Query e l'Update. W3C [2013] Tale protocollo definisce una combinazione di:

- Metodi tramite i quali è possibile inviare la richiesta.
- I parametri da includere negli URI delle richieste HTTP.
- Il contenuto dei messaggi HTTP di richiesta e risposta in seguito coinvolti nell'implementazione di una delle due operazioni.

Query e Update sono due operazioni abbastanza semplici concettualmente e a livello realizzativo, dal momento che consistono in delle HTTP Request che il client effettua verso il server SEPA: a giorno d'oggi il protocollo HTTP è uno dei più diffusi all'interno degli sviluppi di applicazioni Web based.

HTTP (hypertext transfer protocol) è il protocollo a livello di applicazione del Web e ne costituisce il cuore. Questo protocollo è implementato in due programmi, client (colui che richiede il servizio) e il server (colui che risponde alla richiesta del client fornendo il servizio richiesto), in esecuzione su sistemi periferici diversi che comunicano tra loro scambiandosi messaggi HTTP. Esso,

per esempio, è utilizzato dai PC per ottenere le pagine web: quando un PC vuole connettersi a una pagine Web non fa' altro che attuare una HTTP Request, in particolare richiede il File HTML principale di tale pagina WEB (oltre che a tutti gli altri contenuti), il quale sarà locato in un server con un certo IP. Di default le HTTP Request vengono attuate sulla porta TCP 80. Per fare una HTTP Request, il client deve inviare un messaggio contenente determinati parametri ed indicazioni al server, il quale, riconoscendone la struttura, capirà che si stratta di una richiesta HTTP, leggerà ciò che il client sta' richiedendo, risponderà con un codice e, nel caso in cui la richiesta sia andata a buon fine, con l'informazione richiesta dal client.Kurose and Ross [2017]

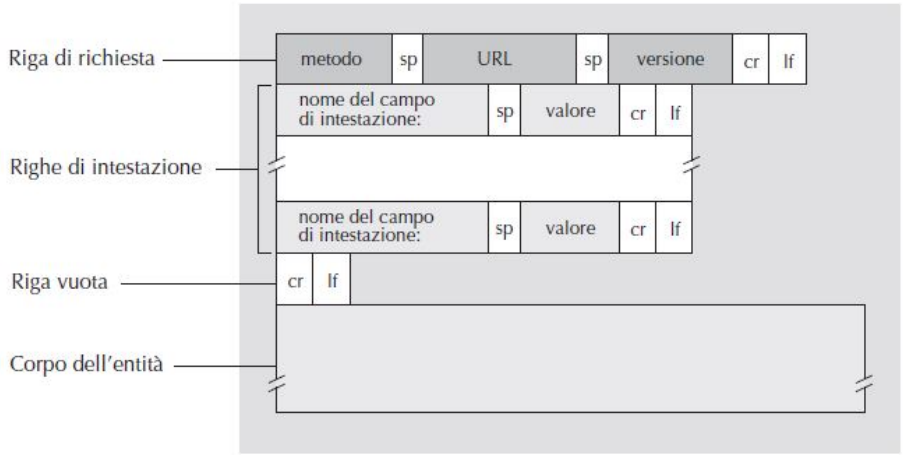

Figura 1.6: Formato generale di una richiesta HTTP Kurose and Ross [2017]

Come possiamo vedere dalla Figura 1.6 in un messaggio HTTP sono presenti alcuni parametri che devono essere presenti e, in base ai quali, il server interpreta la richiesta e risponde.

L'operazione di Query, come anche quella di Update, segue questo schema, all'interno del quale sarà necessario inserire dei determinati valori per far sì che la nostra richiesta sia interpretata correttamente dal SEPA, il quale interpreta i messaggi secondo il protocollo SPARQL 1.1 SE, basato sulla sintassi SPARQL. Quindi sarà immediato dedurre che il corpo della richiesta HTTP che faremo dal nostro client, verso il SEPA, non sarà altro che una stringa SPARQL che il server dovrà leggere e interpretare.

Secondo lo SPARQL 1.1 Protocol, l'operazione di Query presenta tre differenti possibilità per essere realizzata (Figura 1.7): una HTTP GET, una HTTP POST codificata e una HTTP POST non codificata. A seconda della modalità che scegliamo avremo dei differenti parametri da inserire nell header, piuttosto che nel corpo della nostra richiesta. Una volta che il SEPA riceve una Query, risponde al client inviandogli i dati che esso richiede.

	HTTP Method	Query String Parameters	Request Content Type	Request Message Body
query via GET	GET	`query` (exactly 1) `default-graph-uri` (0 or more) `named-graph-uri` (0 or more)	None	None
query via URL-encoded POST	POST	None	`application/x-www-form-urlencoded`	URL-encoded, ampersand-separated query parameters. `query` (exactly 1) `default-graph-uri` (0 or more) `named-graph-uri` (0 or more)
query via POST directly	POST	`default-graph-uri` (0 or more) `named-graph-uri` (0 or more)	`application/sparql-query`	Unencoded SPARQL query string

Figura 1.7: Parametri necessari per fare una Query

Come viene presentato nella Figura 1.8, l'Update a differenza della precedente, presenta solamente due modalità di realizzazione che sfruttano solamente le HTTP POST, queste due modalità si differenziano, oltre che per l'header, per la codifica del messaggio di richiesta presente nel Body (Figura 3.3). Mentre nel primo caso è codificato secondo la codifica URL, nel secondo possiamo inserire come corpo del messaggio direttamente il testo SPARQL corrispondente alla richiesta. Una volta che il SEPA riceve un Update, risponde al client con un codice che lo informa della corretta riuscita dell'operazione fatta.

	HTTP Method	Query String Parameters	Request Content Type	Request Message Body
update via URL-encoded POST	POST	None	`application/x-www-form-urlencoded`	URL-encoded, ampersand-separated query parameters. `update` (exactly 1) `using-graph-uri` (0 or more) `using-named-graph-uri` (0 or more)
update via POST directly	POST	`using-graph-uri` (0 or more) `using-named-graph-uri` (0 or more)	`application/sparql-update`	Unencoded SPARQL update request string

Figura 1.8: Parametri necessari per fare una Update

A livello concettuale è possibile interpretare le Queries come delle domande che un client fa' verso il server, solitamente sono richieste di dati; le Updates sono degli inserimenti di informazioni che il client fa' nel server, andando ad inviare dei veri e propri dati. Collegandolo con la sintassi RDF precedentemente esposta, è possibile affermare che in una Query, andiamo a richiedere il valore di un particolare nodo, o volendo anche tutti i valori appartenenti a un determinato grafo. Con l'Update andiamo ad inserire una nuova tripla

RDF la quale andrà a collegarsi a quelli preesistenti e insieme ad esse formerà l'architettura di dati di un server. W3C [2013]

SPARQL 1.1 SE Protocol

Lo SPARQL 1.1 SE Protocol si basa sul protocollo esposto nella precedente sezione: sfrutta le operazione di Query e Update ma ne formula una terza chiamata Subscribe che si basa sulle precedenti ma che, come vedremo, aumenta notevolmente la dinamicità di questo protocollo rispetto al suo predecessore; inoltre fornisce ai client che utilizzano questo protocollo, una autenticazione basata su OAuth2.0, una criptazione dei dati e una autenticazione da parte del server; tutto ciò garantisce al client un aumento della sicurezza e un'integrità del messaggio, cose che non erano presenti nel precedente protocollo. Roffia et al. [2018b]

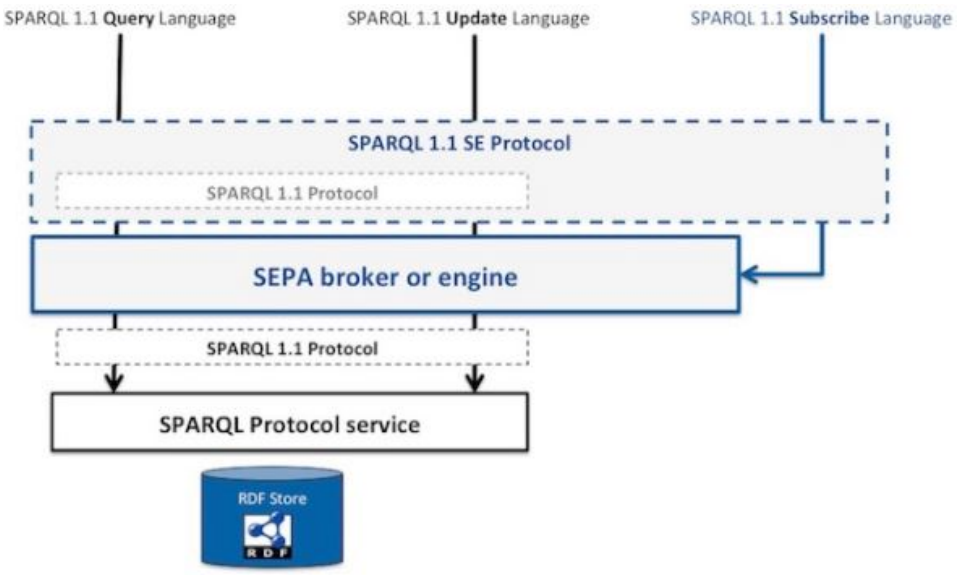

Figura 1.9: Struttura dello SPARQL 1.1 SE Protocol Roffia et al. [2018b]

Dalla Figura 1.9 è possibile vedere come il SEPA Broker, il quale non è altro che il server che implementa tale protocollo rappresentando il cuore della architettura SEPA, si trova frapposto tra il client e l'insieme dei dati RDF. Il compito del Broker è quello di riportare al client le operazioni di Query e Update fornite dallo SPARQL 1.1 Protocol, e di implementare l'operazione di Subscribe.

L'operazione di Subscribe però, a differenza delle due precedentemente citate, sfrutta un protocollo differente dall' HTTP, chiamato Websocket.
Il WebSocket è un protocollo internet che consiste nella creazione di un canale full duplex basato su una singola connessione TCP persistente che trasmette i dati in tempo reale tra client e server. Tale canale rimane attivo finchè uno dei due agenti non lo chiude (o finchè non scade per Timeout): la sua principale caratteristica sta' nel fatto che, oltre ad essere un metodo comunicativo real time bidirezionale, crea un canale diretto tra client e server che non si esaurisce subito dopo una richiesta come succede nel caso delle Query e delle Update.
Lo SPARQL 1.1 Subscribe Language, definisce il contenuto delle primitive di Subscribe e Unsubscribe (attraverso le quali ci sottoscriviamo e annulliamo la sottoscrizione) e di notifiche.
Tutte e tre queste primitive vengono scritte secondo la sintassi JSON tramite la quale vengono definiti i parametri di configurazione. E' possibile interpretare una Subscribe come un'iscrizione: un client può iscriversi a una categoria, a un dato e, ogni volta che questo varia, il SEPA invia a tutti gli iscritti una notifica che li informa della variazione. Al momento della sottoscrizione, il client riceve il risultato della Query contenuta nella Subscribe Request, successivamente essi riceveranno notifiche contenenti i risultati corrispondenti alla Query aggiunti o rimossi dal server. Con questo approccio i subscribers possono facilmente tracciare l'evoluzione dei risultati della Query e quindi l'evoluzione del dato interessato, con il minor impatto sulla banda internet occupata. Nel corpo della nostra Subscribe metteremo una stringa SPARQL contenente il testo corrispondente alla Query della quale vogliamo la risposta. Così facendo, una volta instaurata la WS tra client e server, sul client verrà visualizzata una notifica ogni qualvolta venga aggiunto o rimosso un elemento al grafo sul quale abbiamo fatto la Query

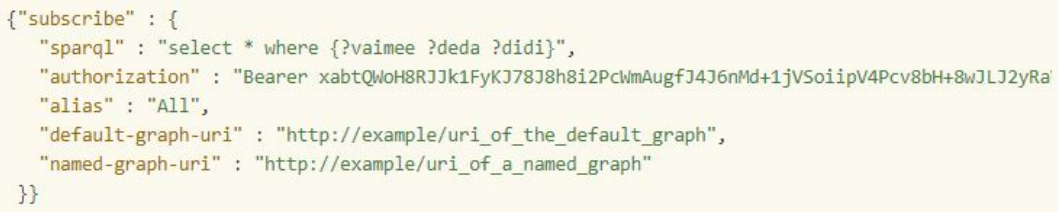

Figura 1.10: Un Esempio di Subscribe Request

```
{"notification":{
   "spuid" : "sepa://subscription/0d057ca5-cc10-4e8a-a5d9-59d7b36f71d6",
   "sequence" : 1,
   "alias" : "All",
   "addedResults" :{
     "head": { "vars": ["vaimee","deda","didi"] },
     "results": {
       "bindings": [{
        "didi": {"type": "literal","value": "ზმკლვვ"} }]}},
   "removedResults":{
     "head": { "vars": ["vaimee","deda","didi"] },
     "results": {
       "bindings": [{
        "didi": {"type": "literal","value": "ვაიმეე"} }]}}}
```


Figura 1.11: Un esempio di Subscribe Notification

Dalla Figura 1.10 e Figura 1.11 è possibile capire come agisce l'operazione di Subscribe: nella richiesta, ci si sottoscrive a un determinato dato specificando in "sparql", la Query della quale vogliamo i risultati. All'interno della notifica, come si può vedere, il messaggio che riceve il client contiene una sezione chiamata "addedResults" all'interno della quale sono contenuti i dati aggiunti al valore del quale siamo interessati, in "removedResults" invece avremo quelli rimossi.
In conclusione possiamo quindi dire che il SEPA è un server costruito sulla base di uno SPARQL endpoint (nel nostro caso useremo Blazegraph) quindi, oltre alle queries e agli updates tipici di questo protocollo (SPARQL 1.1 Protocol) esso "aggiunge" l'operazione di Subscribe per esprimere le notifiche e le richieste di sottoscrizione a particolari dati, formalizzando lo SPARQL 1.1 SE Protocol. Roffia et al. [2018b]
Il SEPA è quindi in grado di gestire le seguenti operazioni:

- Una Query non è altro che una domanda: un dispositivo può fare una Query al SEPA (inerente a un dato presente al suo interno) il quale gli risponderà con il dato richiesto.
- Un Update è l'inserimento di un nuovo dato all'interno del SEPA.
- Una Subscribe possiamo vederla come un'iscrizione: un agente può

iscriversi a una categoria, a un dato e, ogni volta che questo varia, il SEPA invia a tutti gli iscritti una notifica che li informa della variazione.

In questo modo il SEPA si propone come una buona soluzione alla dinamicità dei dati, basato sull'architettura SPARQL (la quale rappresenta il mondo del Web Of Data) ne amplia le potenzialità, riuscendo al meglio a gestire anche l'ormai più diffuso Web Of Dynamic Data. Nell'ipotesi in cui un client sia interessato a monitorare l'andamento di un dato, anzichè iterare una serie di Query per verificare se sia cambiato, potrà sottoscriversi ad esso e il SEPA Broker gli invierà una notifica solo nel caso in cui esso sia cambiato (ovvero se è stato aggiunto un nuovo valore per quel dato). Da ciò ne consegue una possibilità di semplificare notevolmente il carico di lavoro affidato al client, diminuire inoltre il suo consumo energetico e la sua occupazione di banda. Inoltre ne risulta un'architettura server in grado di sfruttare i benefici apportati dal mondo RDF e SPARQL, aumentandone ulteriormente la dinamicità.

Capitolo 2

Sistema Embedded: Arduino Yùn Rev 2

2.0.1 Cos'è Arduino?

Arduino, nato nel 2005 presso l' Interaction Design Institute Ivrea in provincia di Torino, è al giorno d'oggi la più grande piattaforma software e hardware open-source basata sull'uso dei microcontrollori.

Un microcontrollore è un dispositivo elettronico integrato su singolo circuito dotato generalmente di una CPU, di memoria e di PIN di I/O che lo rendono capace di interagire autonomamente con l'ambiente esterno, è utilizzato generalmente nella realizzazione di sistemi embedded ovvero per applicazioni specifiche di controllo digitale e in applicazioni IoT. Ard [2018a]

Figura 2.1: Logo ufficiale di Arduino Ard [2018a]

Con Arduino, si trova un vero e proprio ambiente user-friendly in cui l'utente, collegando la propria scheda al PC, riesce a configurarla attraverso il software dedicato fornito gratuitamente dalla piattaforma (Arduino IDE) in modo semplice e diretto.
Tutto ciò ha reso possibile la diffusione di queste schede grazie alla loro facilità di installazione, al costo relativamente moderato e alla versatilità (sia dal punto di vista software che hardware): essendo infatti un progetto open-source questi dispositivi vengono utilizzati all'interno di numerose situazioni che richiedono l'impiego di tecnologie diverse.
Esistono molti modelli di Arduino, che si differenziano per dimensioni, architettura e potenzialità, per il progetto ho scelto il modello Arduino Yùn Rev 2.

2.0.2 Arduino Yùn Rev 2

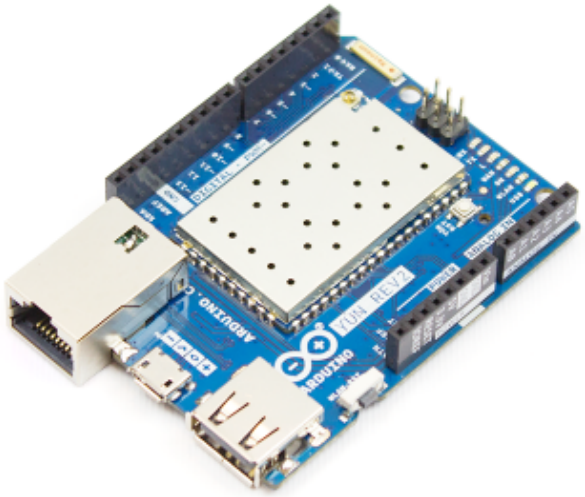

Figura 2.2: Scheda Arduino Yùn Rev 2

Arduino Yun Rev 2 è una scheda Arduino che si distacca da tutte le altre dal momento che presenta due microprocessori anziché uno: infatti oltre al

classico ATMega32U (microprocessore più diffuso della linea Arduino, uguale a quello presente sul Leonardo come anche su molti altri modelli) ne ha un altro addizionale chiamato Atheros9331 (AR9331) il quale supporta come sistema operativo Linino, una versione Linux basata su Open-WRT.
Arduino Yùn presenta un ingresso Ethernet e un'antenna Wifi grazie ai quali, non solo possiamo connetterla ad Internet (sia su LAN che su WLAN) ma anche configurarla tramite Wi-fi (OTA, Over The Air) utilizzandola come Access Point; già da queste due caratteristiche possiamo subito intuire come questa scheda sia perfetta per realizzazioni IoT in cui ci potrebbe essere la possibilità di non riuscire a raggiungere il device.
Grazie alla presenza dei due microprocessori, Arduino Yún Rev 2 combina la facilità d'uso di Arduino con la potenza di Linux: con l'Atheros9331 si trovano numerosi comandi presenti di default su questo sistema (come, per esempio, cURL) ma soprattutto è possibile utilizzare software addizionali su di esso, installandoli da linea di comando tramite 'opkg'.
Nel progetto, come vedremo, giocherà un ruolo fondamentale il software Python2.7, il quale grazie alla sua potenza e versatilità, aggiunge molti benefici al nostro device.

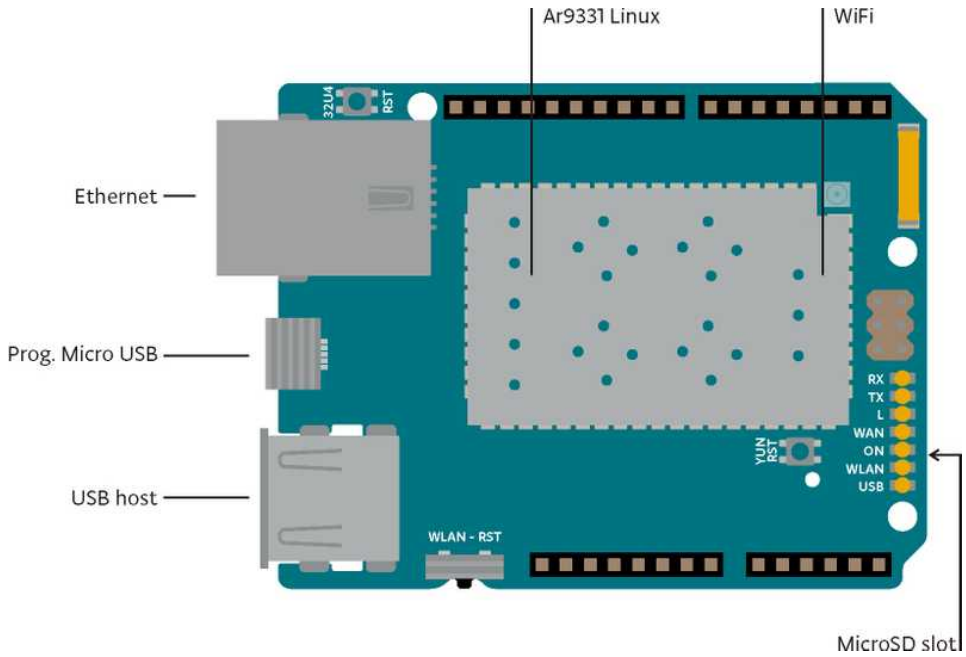

Figura 2.3: Schematico di una scheda Arduino Yùn Rev 2 Ard [2018b]

Arduino Yún Rev 2 è dotato di

- Porta Ethernet 10/100 Mbit/s.
- Interfaccia Wi-Fi 802.11 b/g/n (che permette la connessione a un router wireless o di funzionare come Access Point).
- Porta USB-A.
- Slot per micro SD card.
- 20 pin di ingresso / uscita digitali (di cui 7 possono essere utilizzati come uscite PWM e 12 come ingressi analogici).
- Oscillatore al quarzo da 16 MHz.
- Connettore micro USB per alimentazione a 5V e programmazione.
- Header ICSP e 3 pulsanti di reset (uno per l'ATmega32u4, uno per il Wi-Fi e uno per la sezione Linux).
- Alimentazione: 5V.
- Dimensioni (mm): 68,60x53,30X12,2.
- Peso: 34,5 grammi.

Quando il PC è collegato alla stessa rete WiFi alla quale è collegata anche Arduino, è possibile accedere al Webpanel e alla shell di Atheros9331 tramite SSH.
Per accedere al Webpanel (Figura 2.4) è sufficiente, una volta configurato il device sulla nostra rete WiFi, digitare il nome che gli è stato assegnato in fase di configurazione (in figura vediamo ArduinoLFR) per poter accedere a questa pagina all'interno della quale troviamo molte informazioni utili. Come prima cosa si può notare un paragrafo chiamato "System" all'interno del quale sono elencate informazioni di carattere generale riguardanti il dispositivo, successivamente vengono riportati dei dati inerenti alla Memoria, comprendendo anche la Swap; infine si nota come vengano esposti dati che si riferiscono alla rete alla quale il device è collegato, evidenziando informazioni relative a indirizzo IP, netmask, Gateway della rete, ecc..

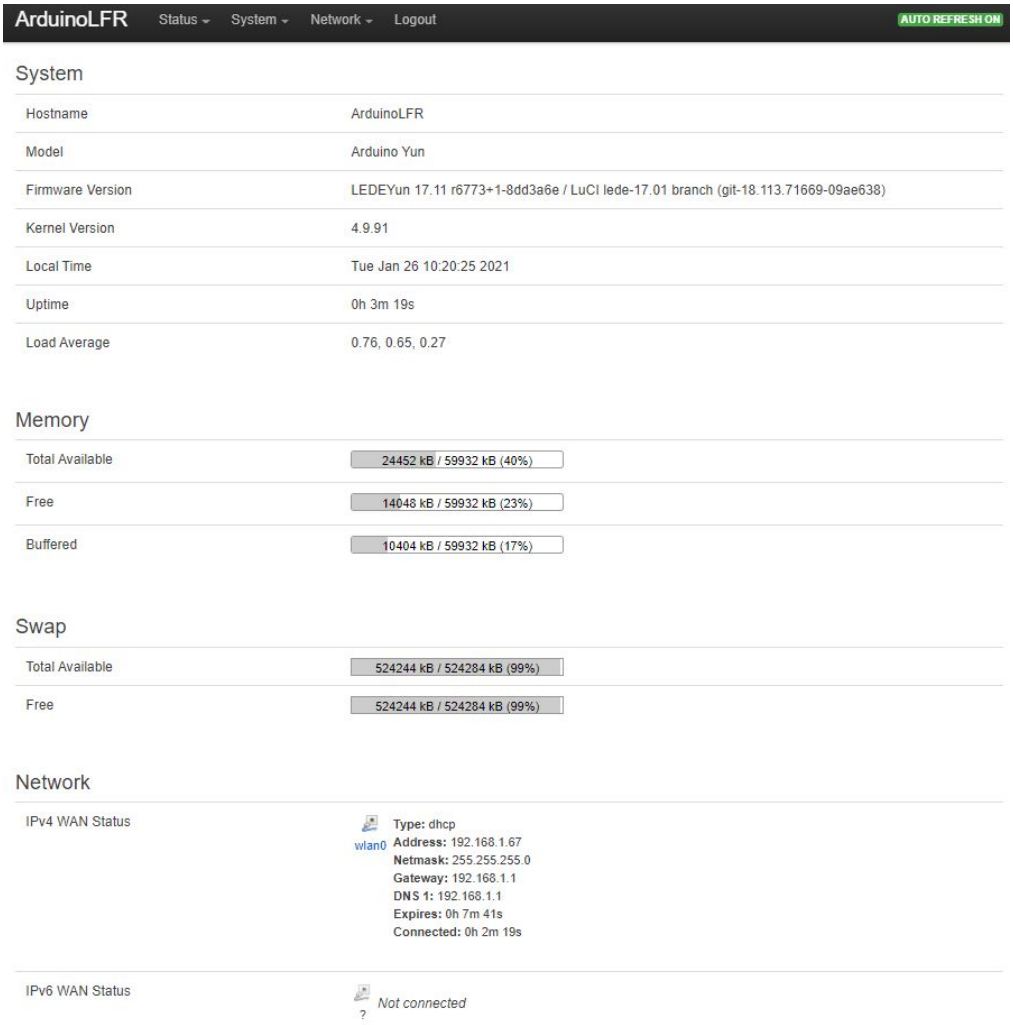

Figura 2.4: Webpanel di Arduino Yun

Una volta identificato l'indirizzo IP del nostro sistema, è possibile tramite SSH connettersi alla shell di OpenWRT (Figura 2.5) dalla quale si possono visualizzare i dati contenuti all'interno di Arduino, e lanciare degli script direttamente da command line.

E' inoltre possibile far comunicare Arduino e OpenWRT-Yun tramite una apposita libreria chiamata Bridge: essa permette lo scambio di dati tra i due processori i quali si trovano a poter lanciare dei processi in modo autonomo ma con la possibilità di collaborare. Bri [2018]

Mentre l'Atheros, grazie alla sua versatilità datagli dall'ambiente di sviluppo

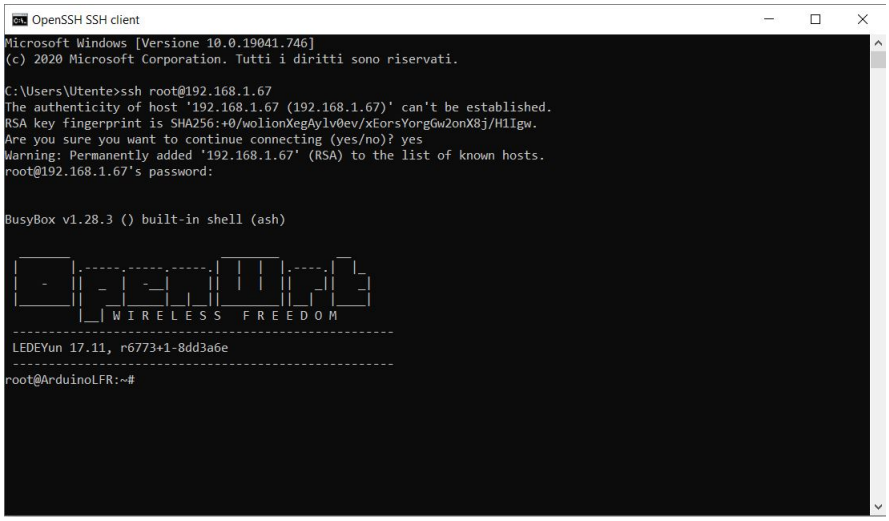

Figura 2.5: Shell di OpenWRT

Linux, riesce a fornire un sostanziale supporto dal punto di vista software (e anche networking), l'ATMega gestisce tutte le operazioni di Input/Output che permettono al microcontrollore di interagire con l'ambiente esterno.

Grazie a questa libreria, è possibile, come si vedrà successivamente, lanciare un processo su Arduino IDE, il quale invocherà uno script, per esempio Python, su OpenWRT; quest'ultimo produrrà una serie di dati i quali, tramite il Bridge arriveranno all'ambiente iniziale e saranno visualizzati sul monitor seriale.

Insieme formano un ambiente che non si limita alla sola interazione fisica con l'esterno, ma che è dotato di una base software la quale allarga di molto le potenzialità di questa scheda.

Il tutto viene riassunto in figura dalla quale è possibile ancora meglio capire quanto questo ambiente sia semplice ma allo stesso tempo permetta la realizzazione di applicazioni più elaborate rispetto a quelle che avremmo nel caso in cui utilizzassimo un Arduino dotato solamente della parte di microcontrollore (parte azzurra in Figura 2.6)

Considerando l'intero ambiente capiamo come questa scheda si presti molto bene ad applicazioni di tipo IoT: una volta collegati dei sensori al device, riusciremo a programmarlo per far sì che acquisisca informazioni da essi e le invii tramite la libreria Bridge e il collegamento internet dato dall'interfaccia Wifi/Ethernet, a un qualsiasi server (privato o non) presente sulla nostra o su altre reti. Ard [2018b]

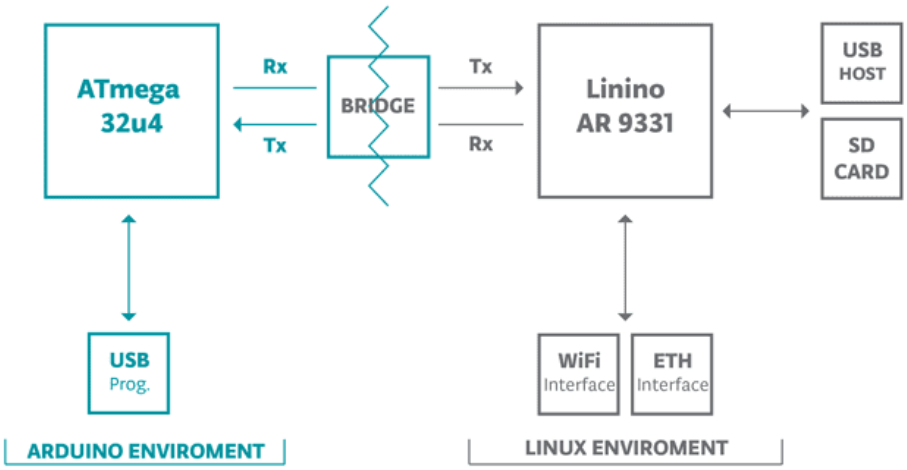

Figura 2.6: Diagramma a blocchi della comunicazione tra AR9331 e ATMEGA Ard [2018b]

2.0.3 OpenWRT-Yun, Python2.7 e SWAP Memory

Come già introdotto precedentemente, OpenWRT-Yun presenta di default Python2.7 il quale dona al sistema una libertà e una flessibilità notevole dal momento che in questo ambiente di sviluppo è possibile trovare numerosi moduli necessari ad implementare molte funzioni, tra le quali le Websocket e le HTTP Requests.
Uno dei principali problemi legati all'utilizzo di Python e delle sue librerie, risiede nella scarsa memoria del nostro dispositivo Arduino il quale spesso non riesce a terminare i processi per via di problemi legati ad essa: in Figura 2.7 è possibile vederne un esempio.
Il microprocessore AR9331 presenta:

- 16MB di memoria flash la quale è possibile espandere con una micro SD esterna.
- 64MB di RAM dei quali però solamente circa 18/20MB sono disponibili ad essere utilizzati per la gestione dei processi.

Secondo i dati sopra riportati, si può quindi intuire che i principali problemi di memoria, non sono legati alla memoria flash, dal momento che è possibile espanderla con una micro SD, bensì è coinvolta la RAM: quest'ultima spesso

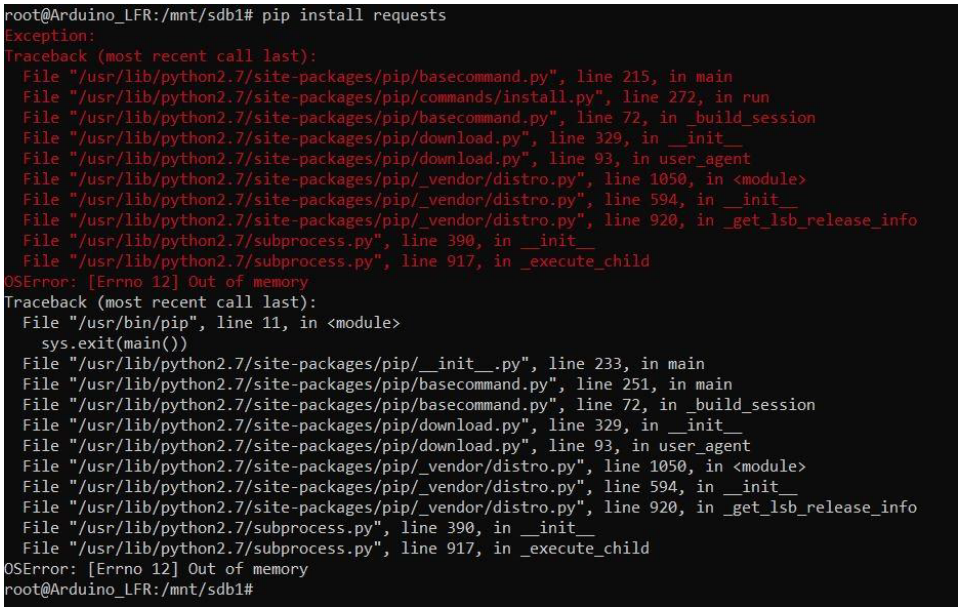

Figura 2.7: Un esempio di errore di memoria

non è abbastanza estesa da poter portare a buon fine alcuni processi, causandone l'interruzione.

Tra di essi risiede anche il comando "pip", necessario all'installazione di moduli Python: si intuisce che ciò rappresenta un problema sostanziale per il nostro dispositivo il quale non può sfruttare pienamente le potenzialità e le migliorie apportate dall'ambiente Linux dal momento che si ritrova a non riuscire a portare a buon fine l'installazione di programmi.

La SWAP memory è una porzione dello spazio di archiviazione (scheda SD, disco fisso, ecc..) che viene allocata come memoria RAM e usata come tale nei casi in cui la RAM del dispositivo che stiamo usando non sia abbastanza per supportare i processi che vogliamo lanciare. Il suo utilizzo però non indica una soluzione ottimale, quando possibile è ovviamente preferibile estendere la RAM, per il semplice fatto che utilizzare la SWAP Memory causa una grossa dilatazione dei tempi di esecuzione dal momento che quando viene utilizzata, il microprocessore deve accedere al disco fisso: recuperare una informazione o un dato in memoria richiede più cicli di clock rispetto all'accesso in memoria RAM.

Di solito la SWAP viene utilizzata dal sistema per gestire processi in parallelo, cercando di riservarla a processi che vengono utilizzati raramente per evitare

un rallentamento complessivo dell'ambiente. Di conseguenza, l'utilizzo della SWAP causa sì un rallentamento del sistema ma, al tempo stesso, permette di gestire processi richiedenti una quantità di RAM superiore a quella che si ha realmente, riuscendo a portarli a termine.
A questa memoria, di default, non è assegnata alcuna risorsa. Tuttavia ci viene data la possibilità di espanderla: così facendo Arduino riesce, nonostante alcune limitazioni, a gestire più processi di quelli che riusciva a gestire precedentemente, dandoci la possibilità di utilizzare il comando "pip" per installare le librerie che ci servono. Palanisamy [2014]

Capitolo 3

Esempio di funzionamento

In questo capitolo si dimostra un esempio di funzionamento il quale sfrutta le tecnologie precedentemente introdotte per realizzare una rete di sensori: attraverso l'uso di due dispositivi della famiglia Arduino, sensori e una versione locale del server SEPA, vengono monitorate temperatura ed umidità in una stanza. In particolari condizioni, viene generato un segnale di allerta che, in questo caso, è rappresentato dall'accensione di due LED. In Figura 3.1 viene presentato uno schematico dell'intero sistema, mentre la Figura 3.2 rappresenta la foto dell'ambiente.

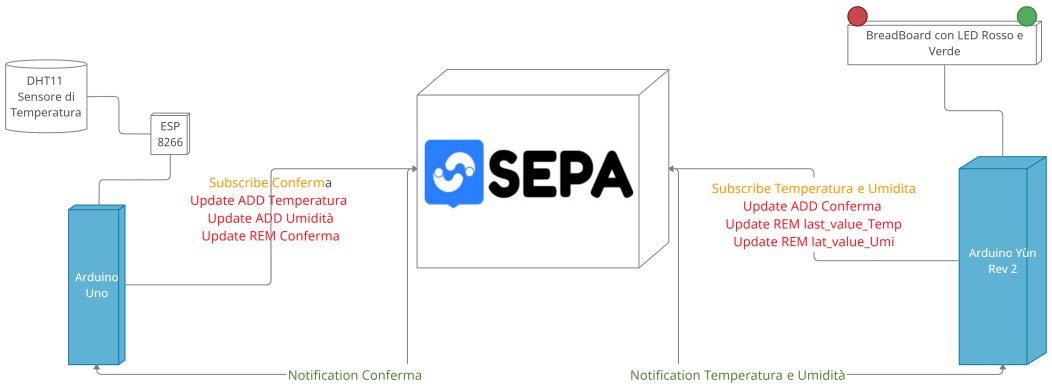

Figura 3.1: Diagramma della DEMO

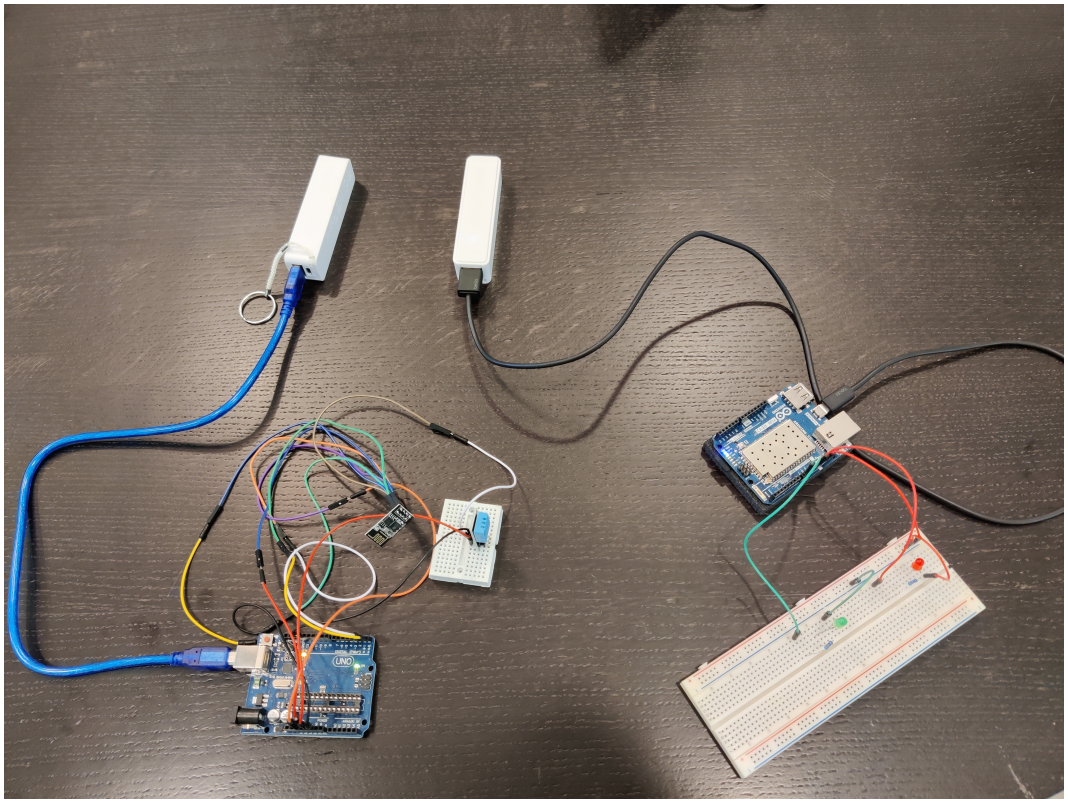

Figura 3.2: Fotografia della rete di sensori

3.0.1 Primo nodo sensore: Arduino Uno e ESP8266

Il primo nodo sensore che viene presentato (Figura 3.3) ha come scopo quello di misurare temperatura ed umidità dell ambiente, inviando continuamente i dati calcolati a un grafo presente sul SEPA al quale è sottoscritto il secondo nodo sensore.
Dal punto di vista hardware il primo nodo sensore presenta:

- DHT11: sensore di temperatura e umidità.
- ESP 8266: microcontrollore che permette all'intero nodo di potersi connettere alla rete Wifi e di poter accedere al SEPA; in questo modo garantisce la possibilità di inviare e ricevere notifiche dal server attuando le operazioni fondamentali.
- Arduino Uno: permette di configurare e gestire il funzionamento del sensore di temperatura e del microcontrollore i quali sono collegati

ad esso, permette quindi all'intero nodo di poter essere programmato tramite Arduino IDE

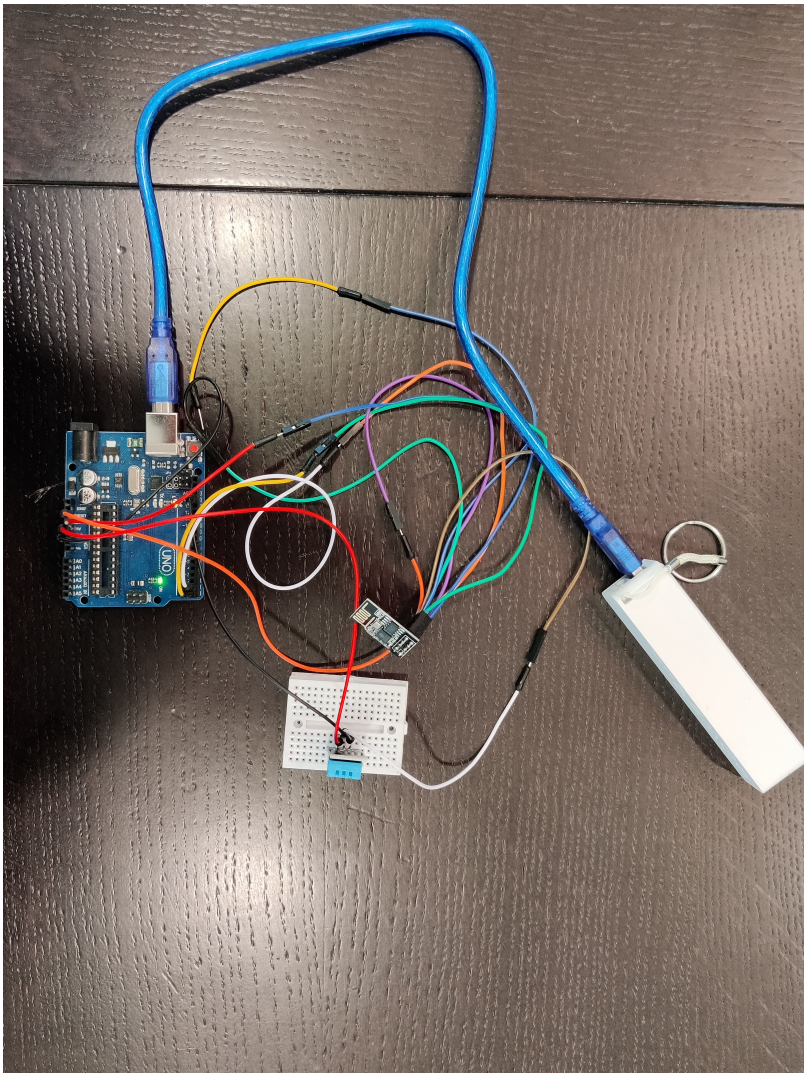

Figura 3.3: Primo nodo sensore

Di seguito viene riportato il codice che viene eseguito sul primo nodo sensore .

```
1  /*Programma per ESP8266 per essere parte di un sistema SEPA. Il nodo è un
   ↪  rilevatore di temperatura e umidità, attraverso il
2   *sensore DHT11. Esso va a leggere la temperatura e l'umidità, inserisce i
   ↪  valori nel relativo grafo, attraverso degli UPDATE in POST.
3   *Inoltre è sottoscritto allo stesso grafo, interessato però solo ad un
   ↪  particolare predicato, ovvero la conferma. Infatti gli altri
```

```
 *nodi dovranno confermare la ricezione dei dati, aggiornando il grafo con predicato "Conferma". In questo modo mostrerà a schermo la
 *notifica di ricezione del dato, e chi l'ha ricevuto
  */


//inclusione delle librerie utilizzate dal sistema
#include <Arduino.h>
#include <ArduinoWebsockets.h> //libreria per la websocket
#include <ESP8266WiFi.h>  //libreria per la connessione al wifi
#include <ESP8266HTTPClient.h>  //libreria per le richieste HTTP
#include <Adafruit_Sensor.h> //librerie per la lettura del sensore
#include <DHT.h>
#include <ArduinoJson.h>  //libreria per la deserializzazione, utilizzata per parsare la notifica ricevuta


//_-_-_-DICHIARAZIONE DELLE VARIABILI_-_-_-//

//variabili per la connessione alla rete e al server
const char* ssid =                 // Inserire il valore di SSID;
const char* password =             // Inserire la password del Wifi al quale vogliamo connetterci;
const char* host_server =          // Inserire l'indirizzo IP del server SEPA;
const uint16_t websocket_port =    // Inserire il numero di porta del SEPA sul quale aprire la Websocket;
const uint16_t update_port =       //Inserire il numero di porta del SEPA sul quale inviare le HTTP Requests;

//variabili per la gestione del tempo
unsigned long int t,dt;

//variabili per la lettura delle temperature
int temp,hum;      //conterranno la temperatura letta
int preTemp = 0;  //contiene la temperatura della lettura precendente
int preHum = 0;   //contiene l'umidità della lettura precendente

//stringa che contiene il nome del grafo a cui sottoscriversi
String grafo = "test3";

//variabili di controllo
```

```
40  int intervallo = 30;  //ogni quanti secondi deve andare a leggere la
    ↪  temperatura
41  int scartoT = 2;      //variazione della temperatura considerata
    ↪  significativa
42  int scartoH = 5;      //varizione dell'unidità considerata significativa
43
44  //oggetto che ci permette di aprire e utilizzare la WebSocket
45  using namespace websockets;
46  WebsocketsClient w_client;
47
48  //oggetto che conterrà il parsing della notifica
49  DynamicJsonDocument doc(1024);
50
51  //oggetto che rappresenta il sensore DHT11
52  DHT sensor(2, 11); //collegato al pin 2, modello 11 (DHT"11")
53
54  //_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_//
55
56  //dichiarazione della funzione utilizzata per effettuare l'update
57  void update_SEPA(String server, String port, String grafo, String
    ↪  soggetto, String predicato, String oggetto, String comando){
58
59      //Creazione di un client HTTP WiFi
60      WiFiClient client;
61      HTTPClient http;
62
63      //connessione al server
64      http.begin(client, "http://"+ server +":"+ port +"/update");
65
66      //invio dell'header
67      http.addHeader("Content-Type", "application/sparql-update");
68
69      //invio del body i POST
70      http.POST(comando +" DATA { GRAPH <http://"+ grafo +"> { <"+ soggetto
        ↪  +"> <"+ predicato +"> "+ oggetto +" } }");
71
72      //messaggio di conferma su seriale
73      Serial.println("Update effettuato!!");
74
75      //chiusura della connessione
76      http.end();
77  }
```

In questa prima parte, in seguito all'inserimento dei parametri di configurazione della rete Wi-Fi alla quale viene collegato il dispositivo, definiamo la funzione "Update", grazie ad essa è possibile effettuare l'omonima operazione: si attua una HTTP Request che viene personalizzata, modificandone i parametri di header e il corpo della richiesta (che corrisponde a una stringa SPARQL), in base alle variabili date come argomenti in ingresso.

```
1  void setup(){
2    //inizializzazione della seriale con baud 115200, in accordo con le
     ↪  specifice dell'ESP8266
3    Serial.begin(115200);
4
5    //inizializzazione del sensore
6    sensor.begin();
7
8    //***************CONNESSIONE AL WiFi*****************//
9    WiFi.begin(ssid, password);
10   //attendo finchè non si connette
11   for(int i = 0; i < 10 && WiFi.status() != WL_CONNECTED; i++) {
12         Serial.print(".");
13         delay(1000);
14     }
15   //controllo che la connessione sia avvenuta correttamente
16   if(WiFi.status() != WL_CONNECTED) {
17         Serial.println("Nessun Wifi Trovato!");
18         return;
19     }
20   Serial.println("Connesso al WiFi!! Tentativo di connessione al
     ↪  server...");
21  //************************************************//
22
23  //************APERTURA DELLA WebSocket*************//
24  bool connected = w_client.connect(host_server, websocket_port,
    ↪  "/subscribe"); //tentativo di connessione al server SEPA
25  if(connected){
26     Serial.println("WebSocket aperta!!");
27     w_client.send("{'subscribe' : {'sparql' : 'select * from <http://"+
       ↪  grafo +"> where { <http://ArduinoYun> <http://Conferma> ?o }'}}");
       ↪  //sottoscrizione a ciò che siamo interessati
28  }
```

```
29  else{
30     Serial.println("Connessione al server fallita. Resettare per
        ↪  riprovare");
31     while(1) ;
32  }
```

A seguire, nel Setup (che si ricorda essere la parte dello sketch eseguita una volta sola) si attua la connessione alla rete Wi-Fi e l'apertura di una Websocket. E' importante evidenziare come tale websocket sia necessaria al nodo sensore per essere in grado di ricevere una notifica di conferma di ricezione del valore di temperatura (inviata dal secondo nodo sensore una volta che è stato correttamente ricevuto il valore); infatti nella stringa SPARQL della Subscription che viene attuata in questo caso, è specificato come soggetto "<http://ArduinoYun>", come predicato "<http://Conferma>" e come oggetto "?o".
Secondo la sintassi SPARQL ciò significa che il nostro primo nodo sensore, sarà sottoscritto al SEPA accettando solamente le notifiche che hanno come predicato <http://Conferma>, provenienti da "<http://ArduinoYun>" il quale rappresenta il soggetto dell'operazione (in questo caso si tratta del secondo nodo sensore, quello al quale viene inviato il valore di temperatura) ciò significa che nel caso in cui l'aggiornamento dei dati vada a buon fine, Arduino Yun aggiunge tramite un Update di conferma, una tripla che ha la seguente forma: <http://ArduinoYun> <http://Conferma> <http://ArduinoUno>.
A questo punto il SEPA invia una notifica a tutti i dispositivi sottoscritti a questo tipo di dato, tra i quali Arduino Uno il quale stamperà sul suo monitor seriale un messaggio di conferma della corretta ricezione del valore di aggiornamento: a ogni valore inviato dal primo sensore corrisponderà un valore di conferma inviato dal secondo sensore che indica la corretta riuscita dell'operazione. (Architettura Publish Subscribe).

```
1  //funzione di callback che viene richiamata ogni volta che viene ricevuta
   ↪  una notifica dal server SEPA
2  w_client.onMessage([&](WebsocketsMessage message) {
3     //leggo il testo della notifica inserendola in una stringa
4     String json = message.data();
5
6     //creo un array con la stringa per poterlo deserializzare (la
      ↪  deserializzazione funziona bene solo con un char in ingresso
```

```
7       int len = json.length();
8       char dati[len];
9       json.toCharArray(dati,len);
10
11      //deserializzo la stringa letta
12      deserializeJson(doc, dati);
13
14      //Salvo in delle variabili i valori ricevuti di soggetto e oggeto;
        ↪  saranno poi utilizzati successivamente.
15      //il predicato è già noto, in quanto siamo sottoscritti solo agli
        ↪  elementi con predicato "Conferma"
16      String soggetto =
        ↪  doc["notification"]["addedResults"]["results"]["bindings"][0]
17      ["s"]["value"];
18      String oggetto =
        ↪  doc["notification"]["addedResults"]["results"]["bindings"][0]
19      ["o"]["value"];
20
21      //mostro a schermo la notifica ricevuta
22      Serial.println("Ricevuta conferma di lettura da "+ soggetto +" con
        ↪  oggetto "+ oggetto);
23      //effettuo un UPDATE per eliminare la notifica appena ricevuta
24      update_SEPA(host_server, String(update_port), grafo, soggetto,
        ↪  "http://Conferma", "<"+ oggetto +">", "DELETE");
25  });
26
27  //leggo il tempo a cui siamo, necessario per temporizzare il sistema
28  t = millis();
29  }
```

Successivamente è definita la funzione "onMessage" la quale viene invocata ogniqualvolta il dispositivo riceva una notifica generata dalla sottoscrizione, ovvero dal SEPA. Quando si verifica questa situazione, ovvero quando il dispositivo riceve una conferma di aggiornamento del dato, Arduino Uno attua un Update che cancella la tripla di conferma appena inserita sul grafo da Arduino Yun. Il motivo di questa operazione risiede nel fatto che la Subscribe, invia una notifica ogni qual volta viene inserito un valore differente da quelli già presenti (in modo da segnalare le variazioni che sono attuate all'interno del grafo di interesse), per cui se non si cancellasse la tripla ogni volta che si riceve un messaggio di conferma, non sarebbe possibile vedere le continue

conferme corrispondenti ai continui aggiornamenti di temperatura e umidità, dal momento che il SEPA riconoscerebbe la tripla <http://ArduinoYun> <http://Conferma> <http://ArduinoUno> già presente nel grafo e quindi non invierebbe la notifica al nostro primo nodo sensore. Grazie a questa operazione avremo una conferma per ogni dato inserito.

```
void loop(){
  // il sistema rimane in ascolto in caso di notifiche dal server
  if(w_client.available()) {
        w_client.poll();
  }

  //ogni n secondi entra in questo loop, dove va a leggere la temperatura e l'umidità
  //se abbiamo un cambiamento significativo, va ad aggiornare il grafo

  //verifica quanto tempo è passato
  dt = millis() - t;
  //se è trascorso più dell'intervallo richiesto, vado a leggere la temperatura e l'umidità. Quindi questa routine è chiamata ogni n secondi
  //in questo modo non si va a bloccare l'ascolto delle notifiche
  if(dt > intervallo){
    temp = sensor.readTemperature();
    hum = sensor.readHumidity();

    if(temp > (preTemp+scartoT) || temp < (preTemp - scartoT)){
      preTemp = temp;
      //invio l'update Temperatura al SEPA
      update_SEPA(host_server, String(update_port), grafo, "http://ESP8266", "http://Temperatura", String(preTemp), "INSERT");
      Serial.println("Inviato un nuovo valore di Temperatura: "+ String(preTemp));
    }

    if(hum > (preHum+scartoH) || hum < (preHum - scartoH)){
      preHum = hum;
      //invio l'update Umidità al grafo
      update_SEPA(host_server, String(update_port), grafo, "http://ESP8266", "http://Umidita", String(preHum), "INSERT");
```

```
29        Serial.println("Inviato un nuovo valore di Umidità: "+
          ↪  String(preHum));
30      }
31
32      //reset del valore di t
33      t = millis();
34    }
35    delay(500);
36  }
```

In seguito è riportata la fase di loop dello sketch (che ricordiamo essere la fase ripetuta iterativamente finché il microcontrollore non viene spento): all'interno di questa parte il nodo sensore calcola continuamente temperatura e umidità e, per evitare di inviare costantemente aggiornamenti ad Arduino Yun, attua delle operazioni di Update nei soli casi in cui queste due grandezze subiscano una variazione sostanziale dei loro valori.

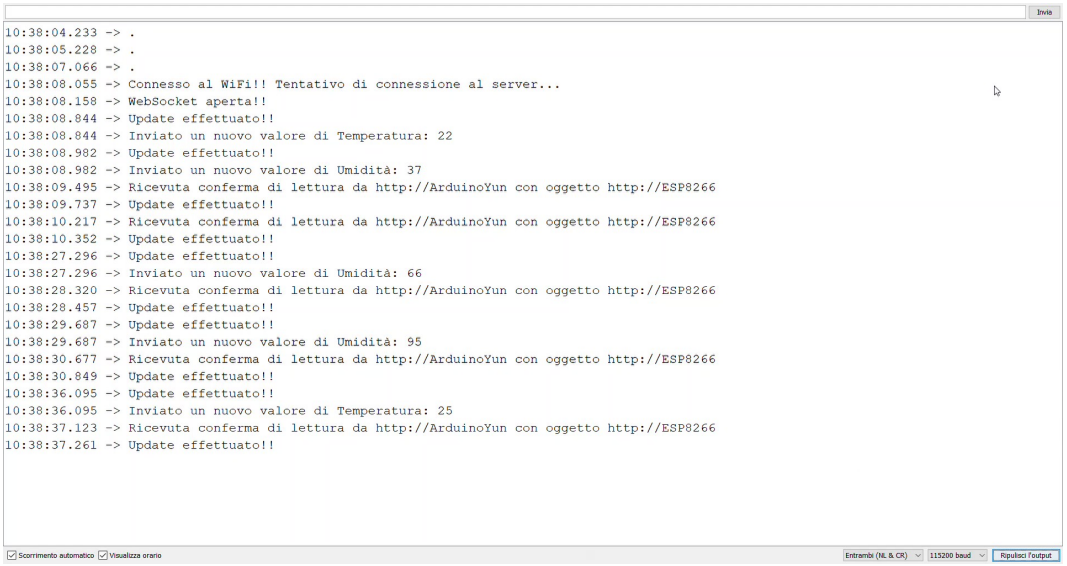

Figura 3.4: Monitor Seriale Arduino Uno

La Figura 3.4 rappresenta il monitor seriale del dispositivo Arduino Uno all'interno del quale possiamo individuare tutte le operazioni elencate precedentemente.

3.0.2 Secondo nodo sensore: Arduino Yun con LED

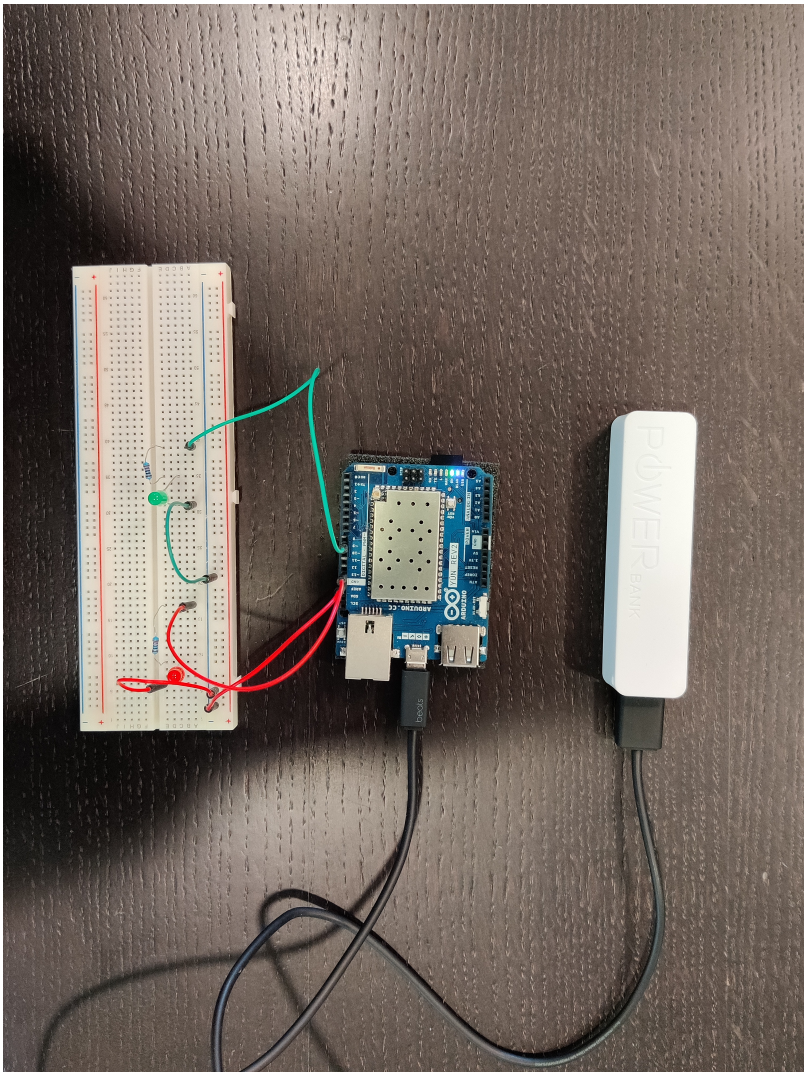

Figura 3.5: Secondo nodo sensore

L obbiettivo del secondo nodo sensore (Figura 3.5) è quello di sottoscriversi ai dati di temperatura e umidità in modo da ricevere le notifiche dal primo nodo sensore. Una volta ricevuti, il dispositivo andrà a verificare che la temperatura non superi i 30°C e l'umidità non superi il 50 per cento. Nel caso in cui queste condizioni non vengano rispettate, il device andrà ad accendere un LED rosso nel caso in cui la temperatura sia oltre la soglia impostata e un LED verde nel caso in cui sia l'umidità a superarla.

Di seguito riportiamo il codice che deve essere lanciato sul secondo nodo sensore:

```
#include <FileIO.h>
#include <Bridge.h>

Process myscript;
Process p;
Process chmod;


// Funzione che scrive uno script in memoria SD nella cartella /mnt/sda1/ chiamato testwebsocket3.py e lo rende eseguibile

void uploadScript() {

  File script = FileSystem.open("/mnt/sda1/testwebsocket3.py", FILE_WRITE);

  script.print("#!/bin/python\n");
  script.print("\n");
  script.print("import sys\n");
  script.print("sys.path.insert(0, '/mnt/sda1/SEPINO')\n");
  script.print("from SEPINO import SEPA\n");
  script.print("from JSAP import JSAP_file\n");
  script.print("SEPA(JSAP_file).Subscribe('select * from <http://demoSEPINO> where { ?s ?p ?o }')");
  script.close();  // close the file

  // Inizia il processo chmod attraverso il quale rende lo script eseguibile da chiunque

  chmod.begin("chmod");                                  // chmod: change mode
  chmod.addParameter("755");                               // 755 indica le autorizza chiunque a leggere, eseguire e decidere l'accesso allo script.
  chmod.addParameter("/mnt/sda1/testwebsocket3.py");     // path fino al file al quale è riferito il processo
  chmod.run();
}
```

In questa prima parte del codice dopo aver inizializzato tre processi (i quali verranno invocati poi successivamente), viene definita la prima funzione dello sketch chiamata "uploadScript()": il compito di tale funzione è quello di creare un file di testo in memoria il quale possiamo personalizzare attraverso il metodo .print(str) al quale vengono passate riga per riga le stringhe di testo che vogliamo inserire nel nostro documento.
Lo scopo di tale operazione si identifica nel fatto che grazie alla classe Process, presente nella Yun Bridge Library, è possibile eseguire un processo direttamente sulla shell Linux del AR9331, in questo modo si possono invocare programmi (come Python nel nostro caso) che aggiungono molta flessibilità e potenzialità agli script Arduino i quali altrimenti possono fare affidamento solamente sulle librerie locali. Bri [2018] Lo scopo della funzione "uploadScript()" sarà quindi quello di scrivere uno script che verrà poi eseguito successivamente.
Il file di testo che scriviamo in memoria e che verrà poi eseguito è il seguente:

```
1 import sys
2 from JSAP import JSAP_file
3 from SEPINO import SEPA
4
5 sys.path.insert(0,/mnt/sda1/SEPINO)
6
7 SEPA(JSAP_file).Subscribe("SELECT * FROM <http://demoSEPINO> WHERE {?s ?p
  ↪  ?o}")
```

dalla sintassi possiamo riconoscere che, come detto in precedenza, si tratta di uno script Python che sfrutta la libreria SEPINO. Inizialmente si inserisce il path in corrispondenza del quale troviamo il modulo Python SEPINO, successivamente viene inizializzata l'operazione di Subscribe inserendo come argomento il testo SPARQL richiesto dall'operazione.

Sepino: libreria Python per Arduino e SEPA

Per automatizzare le operazioni che sono richieste da un device nella comunicazione con il SEPA (Update, Query e Subscribe) è stato realizzato un modulo Python (chiamato SEPINO) contenente una classe chiamata SEPA la quale, come si vede dallo script sopra riportato, viene inclusa nello script

e tramite essa, è possibile effettuare una operazione di Subscribe con una sola riga di codice, semplificando molto il programma a livello computazionale.

```
import json
import websocket
import sys
from requests import post

"""
Questa classe permette di istanziare degli oggetti attraverso i quali posso effettuare delle Update, Query e Subscribe,
i parametri necessari ad effettuare tutte le operazioni verranno presi dal File JSAP passato come argomento in ingresso
alla classe, init infatti come prima cosa effettua un parsing del JSAP (scritto in JSON) trasformandolo in dict Python.
"""

class SEPA:
"""
Inizializzo la classe SEPA facendo il parsing dei file
"""
def __init__(self, file_to_parse):
    self.parametri = json.loads(file_to_parse)
"""
Faccio un Update sfruttando il metodo post di requests, esso accetta come parametri in ingresso: URL a cui fare la richiesta,
testo SPARQL passato come stringa in ingresso al metodo, header
"""
def Update(self, update_txt):
    update_URL = str(self.parametri["sparql11protocol"]["protocol"]) + "://" + str(self.parametri["host"]) + ":" + str(
        self.parametri["sparql11protocol"]["port"]) + str(self.parametri["sparql11protocol"]["update"]["path"])
    r = post(update_URL, data=update_txt, headers={'Content-Type': 'application/sparql-update'})
    print(r.text)
    sys.stdout.flush()
"""""
Fa' una Query con una HTTP post molto simile alla precedente
"""
def Query(self, query_txt):
```

```
32      query_URL = str(self.parametri["sparql11protocol"]["protocol"]) +
        ↪  "://" + str(self.parametri["host"]) + ":" + str(
33          self.parametri["sparql11protocol"]["port"]) +
            ↪  str(self.parametri["sparql11protocol"]["query"]["path"])
34      r = post(query_URL, data=query_txt, headers={'Content-Type':
        ↪  'application/sparql-query'})
35      print(r.text)
36      sys.stdout.flush()
37
38  """
39  Apre una Websocket con il SEPA sfruttando la libreria WebSocket_client;
    ↪  prende l'URL a cui aprire la Websocket sul file JSAP,
40  """
41  def Subscribe(self, websocket_txt):
42
43      websockets_URL = str(self.parametri["sparql11seprotocol"]["protocol"])
        ↪  + "://" + str(self.parametri["host"]) + ":" + str(
44          self.parametri["sparql11seprotocol"]["availableProtocols"]
45          ["ws"]["port"]) + str(
46          self.parametri["sparql11seprotocol"]["availableProtocols"]
47          ["ws"]["path"])
48
49      try:
50          import thread
51
52      except ImportError:
53          import _thread as thread
54
55      def on_message(ws, message):
56          notifica = json.loads(message)
57
58          if (notifica["notification"]["sequence"] == 0):
59              print("Abbiamo aperto una Websocket!")
60              sys.stdout.flush()
61          else:
62              callback(notifica)
63              sys.stdout.flush()
64
65      def callback(notifica):
66
67      if (bool(notifica["notification"]["addedResults"]["results"]
68      ["bindings"][0]["s"]["value"]) and
```

```
69              notifica["notification"]["addedResults"]["results"]
70              ["bindings"][0]["s"]["value"] == "ESP"):
71          soggetto =
            ↪  notifica["notification"]["addedResults"]["results"]["bindings"]
72          [0]["s"]["value"]
73          predicato =
            ↪  notifica["notification"]["addedResults"]["results"]["bindings"]
74          [0]["p"]["value"]
75          oggetto =
            ↪  notifica["notification"]["addedResults"]["results"]["bindings"]
76          [0]["o"]["value"]
77
78          soggetto_parsed = (
79          notifica["notification"]["addedResults"]
80          ["results"]["bindings"][0]["s"]["value"]).replace('http://', '')
81          predicato_parsed = (
82          notifica["notification"]["addedResults"]["results"]["bindings"][0]
83          ["p"]["value"]).replace('http://', '')
84          oggetto_parsed = (
85          notifica["notification"]["addedResults"]["results"]["bindings"][0]
86          ["o"]["value"]).replace('http://', '')
87          print("Abbiamo una notifica da " + soggetto_parsed)
88          sys.stdout.flush()
89
90          if (predicato_parsed == 'Temperatura'):
91              print("E' stata aggiornata la temperatura, ora ci sono " +
                ↪  oggetto_parsed + " gradi")
92              if (int(oggetto_parsed) > 30):
93                  print("Attenzione, troppo caldo!")
94          else:
95              print("E' stato aggiornata l'umidita, ora vale" +
                ↪  oggetto_parsed + " percentuale")
96              if (int(oggetto_parsed) > 50):
97                  print("Attenzione, troppo umido!")
98
99          sys.stdout.flush()
100         post("http://" + str(self.parametri["host"]) + str(
101         self.parametri["sparql11protocol"]["port"]),
102              data='DELETE DATA { GRAPH <http://demoSEPINO> { <' + soggetto
                ↪  + '> <' + predicato + '> ' + oggetto + ' } }',
103              headers={'Content-Type': 'application/sparql-update'})
104         post("http://" + str(self.parametri["host"]) + str(
```

```
105         self.parametri["sparql11protocol"]["port"]),
106             data='DELETE DATA { GRAPH <http://demoSEPINO> {
                ↪  <http://ArduinoYun> <http://Conferma>  <' + soggetto + '>
                ↪  } }',
107             headers={'Content-Type': 'application/sparql-update'})
108         print("Ho aggiornato il grafo!")
109         sys.stdout.flush()
110 
111     def on_error(ws, error):
112         print(error)
113 
114     def on_close(ws):
115         print("### closed ###")
116 
117     def on_open(ws):
118         print("Apertura della Websocket, inviamo la richiesta...")
119         ws.send(json.dumps({"subscribe": {"sparql": websocket_txt}}))
120 
121     websocket.enableTrace(True)
122     ws = websocket.WebSocketApp(websockets_URL, on_message=on_message,
        ↪  on_error=on_error, on_close=on_close)
123     ws.on_open = on_open
124     ws.run_forever()
```

Quando si istanzia un oggetto appartenente alla classe SEPA è necessario inserire come parametro in ingresso un file di tipo JSAP: esso segue la sintassi JSON e il suo compito è quello di elencare i valori di configurazione per il SEPA. La prima cosa che viene fatta dalla classe una volta invocata infatti, è prendere il file passato in ingresso, ed effettuare una deserializzazione secondo la sintassi JSON. Tale operazione viene fatta sfruttando la libreria JSON, la quale è stata inclusa all'inizio dello script: sfruttando il metodo ".loads()" è possibile infatti deserializzare un file JSON passato in ingresso, trasformandolo in un dict Python. JSO [2014] Così facendo creiamo il field self.parametri il quale è un dict Python e tramite il quale è possibile accedere ai singoli campi del file JSON sfruttandone i dati contenuti in esso. JSO Tutto ciò aumenta la facilità di configurazione del SEPA e non solo, in questo modo viene data la possibilità al device per esempio, di potersi connettere a un rest server presente sulla rete, e scaricare direttamente da lì il file JSAP di configurazione, facendo risultare il tutto più automatizzato.

Si possono vedere poi i primi due metodi della classe SEPA i quali sono necessari ad effettuare le operazioni rispettivamente di Query e di Update. Entrambi sono organizzati nello stesso modo dal momento che rappresentano la medesima operazione, ovvero delle HTTP Requests. Per realizzarli è stata impiegata la libreria "requests" la quale tramite il suo metodo .post() permette di attuare tali richieste inserendo parametri personalizzati. Notiamo che i parametri di configurazione di tale richiesta vengono presi dal dict parametri il quale è stato deserializzato precedentemente. HTT [2019]

Infine, nel terzo metodo, viene implementata l'operazione Subscribe aprendo una websocket con il SEPA (attraverso la libreria websocket importata all'inizio dello script): esso presenta 4 funzioni ciascuna delle quali viene invocata ogniqualvolta si verifichi una particolare condizione come per esempio onopen() che viene chiamata quando si apre la websocket, piuttosto che onnotification() quando riceviamo una notifica. Web [2019]

Viene evidenziata la funzione callback nella quale risiedono i comandi necessari ad attuare la nostra dimostrazione: ogni volta che si riceve una notifica dalla websocket, essa viene deserializzata (dal momento che anche le notifiche sono scritte secondo la sintassi JSON) e, nel caso in cui non si tratti del messaggio iniziale contenente tutti i dati già presenti nel grafo al momento della sottoscrizione, viene chiamata la funzione callback dandogli come parametro in ingresso la dict Python deserializzata.

L'obiettivo di questo secondo nodo sensore è quello di verificare continuamente se sono stati aggiunti valori di temperatura e umidità da Arduino Uno, per questo motivo, nel caso in cui questa condizione sia verificata, la funzione inizia ad analizzare il messaggio di notifica. Come prima cosa vengono parsati i valori di soggetto, predicato ed oggetto, dal momento che il nostro device, riceve, in linea con la sintassi SPARQL1.1 SE, gli URI corrispondenti a questi tre valori, i quali avrebbero il prefisso "http://" davanti. Roffia et al. [2018b]

Successivamente si analizza il valore calcolato e si stampa un messaggio che indica la corretta ricezione.

Poi la funzione attua due Update verso il SEPA: il primo cancella il valore appena ricevuto dal grafo di modo che, nel caso in cui si riceva un valore uguale successivamente, il SEPA invii comunque la notifica al nostro device (seguendo lo stesso ragionamento per il quale il primo nodo sensore cancella continuamente i messaggi di conferma che riceve), la seconda invia un

messaggio di conferma al primo nodo sensore o, più generalmente, al device che gli ha inviato la notifica.

E' importante ricordare che tutti i dati generati, vengono inseriti in un buffer di uscita dal processore AR9331 chiamato STDOUT, da qui attraverso il Bridge vengono letti dal metodo .read() proprio dei Process di Arduino e successivamente stampati a video sul monitor seriale. Per evitare che tale buffer si saturi e non riesca ad inviare più i dati all ATMEGA, all'interno dello script sono presenti numerosi flush che svuotano STDOUT e lo rendono in grado di accettare nuovi dati prodotti dagli script Python. Bri [2018]

Tornando allo sketch Arduino iniziale, nelle ultime righe della funzione uploadString() viene istanziano un processo "chmod 755" (infatti grazie al metodo .addParameter() possiamo aggiungere parametri ai processi che lanciamo) tramite il quale forniamo le autorizzazioni necessarie all'utente per modificarlo ed eseguirlo, infatti 755 significa che vengono date le autorizzazioni di: lettura, scrittura ed esecuzione (7) per il proprietario (prima cifra), e lettura ed esecuzione (i due 5) per gruppo (seconda cifra) ed altri (terza cifra). Piccardi [2015]

```
1  // Funzione che lancia un processo il quale invoca l'interprete di python
   ↪  ed esegue lo script scritto precedentemente
2
3  void runScript() {
4
5    myscript.begin("python" );
6    myscript.addParameter("/mnt/sd/testwebsocket3.py");
7    myscript.runAsynchronously();
8
9  }
```

La funzione successiva si chiama runScript() e non fa' altro che lanciare lo script precedentemente scritto attraverso il processo "myscript", si potrebbe tradurre come effetturare "python /mnt/sda1/testwebsocket3.py " dalla command line.

```
void setup() {


  // Come prima cosa cancelliamo il vecchio script per fare in modo che
  ↪  non ci siano confilitti nel caso di modifica
  p.begin("rm /mnt/sda1/testwebsocket3.py");

  // Inizializziamo i PIN 9 e 13 settandoli come OUTPUT
  pinMode(13,OUTPUT);
  pinMode(9,OUTPUT);

  // Inizializziamo il Bridge, la comunicazione col monitor seriale e il
  ↪  FileSystem
  Bridge.begin();
  Serial.begin(9600);
  while (!Serial); // wait for Serial port to connect.
  FileSystem.begin();

  // Effettuiamo l'upload dello script e successivamnte lo lanciamo
  uploadScript();
  runScript();

}
```

E' riportata poi la parte di Setup dello sketch Arduino all'interno del quale, oltre ad inizializzare le varie forme di comunicazione (Bridge, Seriale e FileSystem) e i LED presenti sui pin 9 e 13, vengono eseguite le due funzioni illustrate precedentemente: uploadScript() e successivamente runScript().

```
void loop() {

  // Finchè il processo ha dei dati che vengono inviati attraverso il
  ↪  bridge e devono essere letti li immagazzina in una stringa c
  while (myscript.available() > 0) {
    String c = myscript.readString();
    Serial.print(c);
    // Se vengono identificate le parole caldo o umido, il device attiva i
    ↪  LED per 5 secondi e poi li spegne
    if (c.indexOf("caldo") > 0) {
      digitalWrite(13, HIGH);
```

```
10        delay(5000);
11        digitalWrite(13, LOW);
12      }
13      if (c.indexOf("umido") > 0) {
14        digitalWrite(9, HIGH);
15        delay(5000);
16        digitalWrite(9, LOW);
17      }
18    }
19   // Nel caso in cui non ci siano dati da leggere, sul monitor vengono
     ↪  stampati dei punti
20   Serial.println(".");
21   // Viene continuamente fatto il flush del buffer seriale per evitare che
     ↪  si saturi e non sia in grado
22   // di accettare dati in ingresso
23   Serial.flush();
24   delay(3000);
25  }
```

Lo sketch termina con la fase di loop nella quale il programma controlla iterativamente se il processo "myscript" (che consiste nell'esecuzione dello script Python precedentemente esposto) ha dei dati disponibili a essere letti: ciò è reso possibile grazie al metodo .available() che restituisce un numero intero maggiore di zero nel caso in cui tale condizione si sia verificata.
Successivamente i dati ricevuti vengono immagazzinati in una stringa per poi essere stampati a video.
Nel caso in cui la temperatura sia maggiore di 30 °C, la funzione di callback presente nella classe SEPA stampa il messaggio "Attenzione, troppo caldo!", questo messaggio viene immagazzinato nella stringa c dello skech Arduino (tramite il metodo .readString()). In seguito si analizza tale stringa e nel caso in cui venga rilevata la presenza della parola "caldo" all'interno di essa, è attivato un LED rosso per la durata di 5 secondi.
Secondo lo stesso procedimento, viene valutata continuamente l'umidità e, nel caso in cui venga rilevato un valore troppo alto, sarà stampata a video la stringa "Attenzione, troppo umido": la parola "umido" innescherà l'accensione di un LED Verde per 5 secondi.
In Figura 3.6 è possibile vedere il monitor seriale di Arduino Yun all'interno del quale sono stampate le operazioni elencate precedentemente.

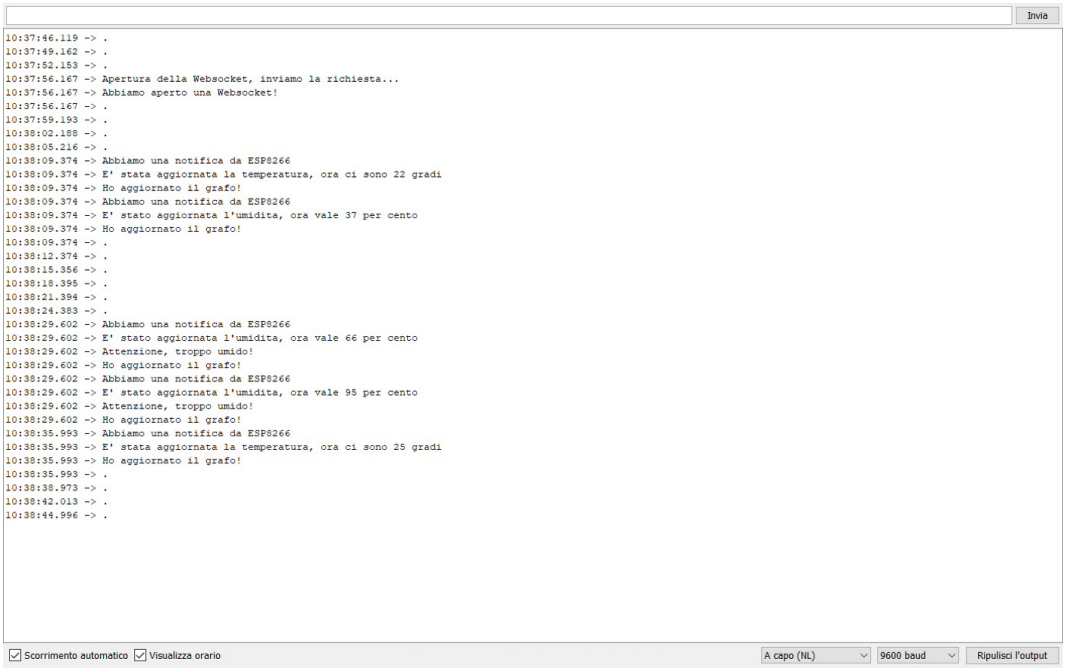

Figura 3.6: Monitor Seriale Arduino Yun

Capitolo 4

Osservazioni e conclusioni

In conclusione, in seguito al caso di studio riportato si può affermare che Arduino Yun, nonostante la sua semplicità, presenta sufficienti requisiti tecnici da eseguire correttamente le operazioni fondamentali di Update, Query e Subscribe. Esse sono state rese più semplici e user friendly grazie all'implementazione della libreria Python SEPINO e tramite la presenza del processore AR9331 il quale ha permesso l'utilizzo di Python2.7, semplificando e aumentando notevolmente le potenzialità del nostro dispositivo. Arduino Yun inoltre presenta tutte le qualità proprie di tale ambiente, tra le quali evidenziamo l'interoperabilità dimostrata soprattutto dalla presenza della libreria YunBridge la quale permette l'interfacciamento tra Arduino IDE e la shell Linux.

Tuttavia in alcune occasioni il device ha dimostrato di aver specifiche tecniche limitate riguardanti specialmente la memoria (RAM e flash): come esposto in precedenza, nella prima fase non era possibile installare alcun software aggiuntivo e spesso i processi lanciati si arrestavano poco dopo il loro inizio. Nonostante tali problemi siano stati risolti aggiungendo una microSD da 16GB ed aumentando le capacità della RAM tramite SWAP Memory, Arduino presenta capacità più limitate rispetto ad altri sistemi embedded (come per esempio Raspberry PI).

L'obbiettivo è quello di implementare e sviluppare meglio la libreria SEPINO rendendola in grado di gestire i "Forced Bindigs" nei quali risiede una parte della potenza apportata dallo SPARQL. Tale progetto mira a una situazione all'interno della quale più device siano configurabili semplicemente tramite la presenza di un file JSAP comune ad entrambi (scaricabile per esempio da una

risorsa comunemente accessibile come un rest server o una web page) all'interno del quale siano espressi non solo i parametri di configurazione del SEPA col quale stanno lavorando ma anche il testo SPARQL delle Update e delle Query, evitando all'utente di dover entrare in contatto con tale linguaggio. In questo modo sarà anche possibile sviluppare un'applicazione di modo da garantire un'interfaccia ad alto livello user friendly la quale permetterà all'utente di non entrare in contatto con alcun tipo di linguaggio di programmazione.
Il progetto SEPINO mira ad interfacciare l'architettura del SEPA con il mondo dei sistemi embedded Arduino: in questo elaborato è stato analizzato principalmente il comportamento del modello Arduino Yun Rev 2.
Tale progetto è ancora in fase di sviluppo e presto le tecnologie SEPA saranno disponibili anche per altre versioni di Arduino. L'obbiettivo finale del progetto SEPINO, come si intuisce dal nome, è quello di unire questi due ambienti.
Da una parte il SEPA garantisce tutti i benefici di una piattaforma semantica presentando tecnologie come RDF e SPARQL, dall'altra Arduino assicura una semplicità di utilizzo notevole la quale, unita all'ambiente user friendly e alla economicità dei devices, rendono i dispositivi di questa famiglia molto versatili: capaci di affrontare situazioni di differenti semplicità. Per realizzare tutto ciò il progetto SEPINO continua nello sviluppo di software e librerie che mirano all'interfacciamento di questi due ambienti.

Bibliografia

Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web, 2001.

W3C. Rdf primer. https://www.w3.org/TR/rdf-primer, 2004.

W3C. Rdf 1.1 primer. https://www.w3.org/TR/rdf11-primer, 2014a.

W3C. Rdf 1.1 concepts and abstract syntax. https://www.w3.org/TR/rdf11-concepts, 2014b.

Luca Roffia, Paolo Azzoni, Cristiano Aguzzi, Fabio Viola, Francesco Antoniazzi, and Tullio Salmon Cinotti. Dynamic linked data: A sparql event processing architecture. https://www.mdpi.com/1999-5903/10/4/36/htm, 2018a.

W3C. Sparql 1.1 protocol. https://www.w3.org/TR/sparql11-protocol, 2013.

James F. Kurose and Keith W. Ross. *Reti di calcolatori e Internet*. Pearson, 2017.

Luca Roffia, Cristiano Aguzzi, Fabio Viola, and Francesco Antoniazzi. Sparql 1.1 secure event protocol. http://mml.arces.unibo.it/TR/sparql11-se-protocol.html, 2018b.

What is arduino? https://www.arduino.cc/en/Guide/Introduction, 2018a.

Getting started with the arduino yún rev. 2. https://www.arduino.cc/en/Guide/ArduinoYunRev2, 2018b.

Bridge library for yún devices. https://www.arduino.cc/en/Reference/YunBridgeLibrary, 2018.

Mohan Palanisamy. Swap file for extending the ram on arduino yun. http://mohanp.com/arduino-yun-extending-the-ram-with-swap-file, 2014.

json-json encoder and decoder. https://docs.python.org/2.7/library/json.html, 2014.

Introducing json. https://www.json.org/json-en.html.

Requests: Http for humans. https://requests.readthedocs.io/en/master, 2019.

Websocket client 0.57.0. https://pypi.org/project/websocket_client, 2019.

Simone Piccardi. *Amministrare Gnu/Linux*. Truelite S.r.l, 2015.

Cristiano Aguzzi, Francesco Antoniazzi, Luca Roffia, and Fabio Viola. Json sparql application profile (jsap). https://www.w3.org/TR/rdf-primer, 2018.

Elenco delle figure