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 UNO e ESP8666 alla piattaforma semantica SEPA

Relatore:
Prof. Luca Roffia

Autore:
Matteo Brighi

Correlatori:
Dott. Simone Sindaco

Anno Accademico 2020/2021

Sessione III - Marzo 2021

Indice

Introduzione **5**

1 Piattaforma Semantica SEPA **7**

1.0.1 RDF Data Model 8
1.0.2 SPARQL 1.1 SE Protocol 11
1.0.3 I servizi REST 18

2 Arduino Uno Rev 3 ed ESP8266 **21**

2.0.1 Sistemi Embedded e Arduino 21
2.0.2 Arduino Uno Rev 3 ed Ethernet Shield v. 1.0 22
2.0.3 ESP8266: Modulo ESP-01 27
2.0.4 Il Software: Arduino IDE 32

3 Esempio di Funzionamento **35**

4 Osservazioni e conclusioni **55**

Bibliografia **57**

Elenco delle figure **59**

Introduzione

Il grande sviluppo tecnologico avvenuto nell'ultimo decennio ha portato alla realizzazioni di dispositivi e componenti elettronici sempre più efficienti e dalle dimensioni ridotte. Questo sviluppo ha consentito una delle innovazioni principali degli ultimi anni: poter inserire un apparato di rete in qualsiasi oggetto della quotidianeità, rendendolo connesso al resto del mondo attraverso il Web.

L'IoT o *Internet of Things* è una disciplina nata proprio con l'intento di sfruttare questo ramo tecnologico. Essa cerca di connettere in rete qualsiasi oggetto che ci circonda, rendendolo in grado di rilevare informazioni che possano essere sfruttate al meglio da chiunque ne sia interessato. Supponiamo, ad esempio, un sensore di temperatura che, attraverso una connessione alla rete, condivida il valore rilevato con chiunque si connetta ad esso, ma soprattuto, ovunque esso si trovi nel mondo. Con l'avanzare degli studi, si è notato, inoltre, come questi sistemi siano popolati in maggioranza da dispositivi elettronici piuttosto che da umani. Si è quindi iniziato a ridimensionare il concetto di rappresentazione del dato. Se un dato viene condiviso soltanto attraverso calcolatori, la rappresentazione di quest'ultimo passa in secondo piano, in quanto essi riconoscono il dato semplicemente come una serie di bit e non hanno necessità di visualizzarlo, ad esempio su uno schermo. Nasce così quello che oggi prende il nome di Semantic Web, ovvero una rete in cui i dati vengono valorizzati per la loro natura e non per come essi sono rappresentati. Inoltre si punta all'autonomia dei sistemi, in modo che l'intervento umano sia sempre meno necessario. All'interno dell'elaborato verrà illustrata la piattaforma semantica SEPA, sistema che sfrutta il concetto del Semantic Web, entrando nell'ambito dell'IoT, attraverso un modello di dati RDF e una gestione di quest'ultimo in sintassi SPARQL. L'obiettivo dell'attività di ricerca è stato quello di interfacciare microcontrollori della

famiglia Arduino ed ESP con un server SEPA. In particolare, inizialmente si effettueranno delle prove con Arduino Uno, il modello più basico della famiglia, per poi spostarsi su un microcontrollore ESP8266, per i motivi che verranno illustrati nell'elaborato. L'idea è quella di rendere l'IoT disponibile a tutti, sfruttando proprio la famiglia Arduino, data la sua grande semplicità di utilizzo e il costo contenuto dei prodotti. Si cercherà inoltre di rendere tutto il sistema stand-alone, sfruttando il concetto dei servizi REST.
Nel primo capitolo verranno illustrate le tecnologie alla base del SEPA; in particolare si analizzerà il modello dati utilizzato, che rende il sistema particolarmente adatto all'interfacciamento con il mondo dei microcontrollori. Il secondo capitolo sarà dedicato all'analisi delle schede utilizzate, illustrandone le prestazioni e i problemi riscontrati nei vari test effettuati. Il tutto terminerà con una demo, realizzata come dimostrazione del successo dell'attività di ricerca svolta, contenuta nel terzo capitolo.

Capitolo 1

Piattaforma Semantica SEPA

Il SEPA (*SPARQL Event Processing Architecture*) è un sistema che consente lo sviluppo di applicazioni di tipo *Dynamic Linked Data* sfruttando un'architettura di rete Publish-Subscribe e un'organizzazione dei dati in database di tipo sparso. Per la sua implementazione vengono sfruttate tecnologie applicative quali RDF Data Model, che fornisce una struttura a grafo per l'organizzazione dei dati, e un *Broker* (o *Engine*), componente fondamentale dell'architettura SEPA che consente l'utilizzo di SPARQL 1.1 SE Protocol, mettendo a disposizione tre primitive per la gestione ed il controllo dei dati all'interno. Le tre operazioni prendono il nome di *Update, Query e Subscribe*. Un'ulteriore grande vantaggio si ha nell'utilizzo di file JSAP (*JSON SPARQL Application Profile*) per la configuarzione del server SEPA stesso e di tutti i client che si connettono ad esso. Attraverso l'utilizzo di queste tecnologie, il SEPA si dimostra in grado di adattarsi efficentemente al mondo del Web of Dynamic Data e la sua semplicità di utilizzo permette di sfruttarlo soprattutto in applicazioni di tipo IoT Roffia et al. [2018a].

Nelle pagine successive verrà presentata una breve panoramica della struttura RDF, del linguaggio SPARQL (e quindi delle operazioni di manipolazione dei dati) e dell'architettura del SEPA. Verrà inoltre introdotto il concetto di servizio REST, che permetterà di indirizzare il sistema verso un modello stand-alone.

1.0.1 RDF Data Model

Il termine RDF è l'acronimo di *Resource Description Framework*. Si tratta di un modello di rappresentazione dei dati, in cui il concetto di visualizzazione di quest'ultimo passa in secondo piano. Permette di rappresentare i dati presenti sul Web basandosi sulla loro natura e sulle interconnessioni che ci sono fra essi. RDF è pensato per permettere un'elaborazione dei dati più fluida ed immediata, soprattutto dove non è necesaria una visualizzazione di quest'ultimo. Per questo motivo risulta particolarmente adatto in sistemi di calcolatori, come ad esempio nell'IoT.

Figura 1.1: Logo RDF Data Model

L'elemento fondamentale di una struttura dati RDF è la cosidetta *Tripla* o *Statement*, di cui viene mostrato un esempio in Figura 1.2. Una Tripla esprime un rapporto di interconnessione tra due risorse. In particolare può essere rappresentata con un grafo orientato, composto da due nodi e un arco che li unisce:

- L'arco è chiamato PREDICATO.
- I due nodi sono chiamati SOGGETTO e OGGETTO.

Figura 1.2: Esempio generale di Tripla

L'arco è orientato dal soggetto verso l'oggetto e rappresenta la relazione che intercorre tra le due risorse.

I nodi che compongono una tripla RDF possono essere principalmente di due tipi; in base al loro contenuto vengono infatti suddivisi in *Literals* e *IRI's*. Esistono poi anche i cosidetti *Blanck Nodes*, ovvero nodi paragonabili a delle variabili algebriche, in quanto rappresentano una risorsa, ma senza indicarne il valore. Analizziamo invece i primi due tipi di nodi, che sono quelli più significativi per l'utilizzo nella struttura dati RDF all'interno del sistema realizzato:

- I nodi di tipo **IRI** identificano una risorsa senza specificarne la sua locazione o come ci si può accedere. Gli IRI (International Resource Identifier) sono delle generalizzazione degli URI, in quanto accettano dei caratteri anche non di tipo ASCII. Possono apparire in qualsiasi posizione della Tripla e vengono usati per rappresentare risorse, come documenti, persone, oggetti fisici e concetti astratti. Nei vari linguaggi, gli IRI vengono identificati attraverso le parentesi angolari (< ... >).

- I nodi di tipo **LITERAL** sono valori che rappresentano ad esempio numeri, date, stringhe, ecc.. Essi possono apparire solamente come oggetti.

La grande potenzialità di un dataset di tipo RDF è quella di riuscire a fondere informazioni provenienti da fonti differenti, con l'obbiettivo di formare un insieme di dati dinamico e interconnesso. Si possono creare infatti iterazioni tra Statement, dando origine a quelli che in sintassi RDF vengono chiamati *Grafi*. I grafi sono quindi un'agglomerato di triple, in cui, ad esempio, l'oggetto di un determinato Statement può essere, allo stesso tempo, il soggetto di un'altra Tripla. Questo tipo di struttura fornisce una grande flessibilità, dando la possibilità ad un sistema esterno di muoversi liberamente all'interno del grafo, in completa autonomia e senza la necessità di informazioni esterne.
Avere una struttura organizzata in questo modo introduce un'altro vantaggio notevole, ovvero quello di poter utilizzare tutte le primitive fornite dal linguaggio SPARQL per manipolare il grafo.

Come detto, l'insieme di più triple interconnesse dà origine ad un grafo. Ogni grafo ha un IRI associato che lo identifica. Per questo vengono definiti come *Named Graph* e ciò che li identifica (IRI) viene chiamto *Graph Name*. Quando un grafo viene identificato da un Blank Node e non da un IRI, viene definito come **Default Graph**. Infine, l'insieme di più grafi collegati viene definito come **RDF Dataset** W3C [2014a].

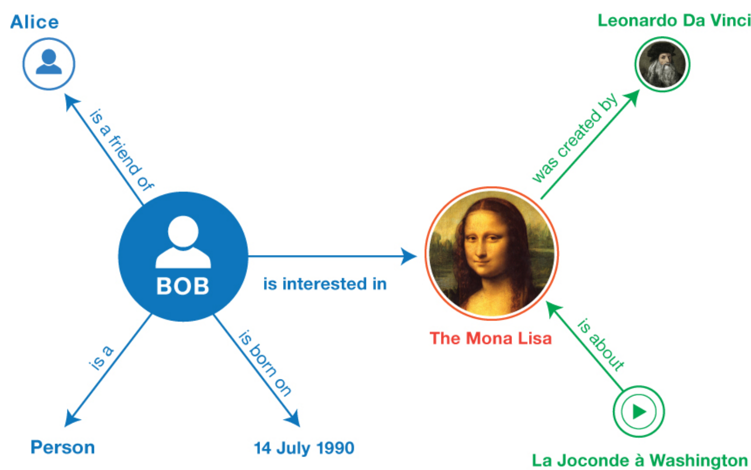

Figura 1.3: Esempio di dataset RDF

In Figura 1.3 viene riportato un semplice esempio di dataset RDF. Come si può notare, esso è composto da due grafi, collegati attraverso un predicato. Se si prende, ad esempio, il nodo che rappresenta l'opera d'arte "The Mona Lisa", si notano le caratteristiche sopra citate; esso infatti fa parte di tre diverse triplette: in due di esse è un oggetto, mentre nella terza rappresenta un soggetto (The Mona Lisa[soggetto] è stata creata[predicato] da Leonardo Da Vinci [oggetto]). Questo esempio mostra la grande potenzialità del modello RDF e l'interconnessione che si crea tra più informazioni, cosa che non sarebbe possibile in un normale database.

Occorre definire anche quello che viene chiamato **RDF Document**. Esso è un documento che codifica un RDF Graph con una sintassi concreta, come può ad esempio essere Turtle o TriG. La realizzazione di un RDF Document permette lo scambio di RDF Graph e RDF Dataset tra sistemi. Il più semplice ed intuitivo è N-Triplet, di cui viene mostrato un esempio di sintassi in

Figura 1.4, che è la sintassi che rappresenta l'RDF Document dell'esempio precedentemente esposto W3C [2014b].

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

Figura 1.4: Sintassi N-Triplets per la rappresentazione del grafo in Figura 1.3

La struttura appena presentata è quella che viene sfruttata all'interno dell'architettura SEPA e fornisce flessibilità, leggerezza e dinamicità nella rappresentazione di dati, consentendo di sfruttare la tecnologia in sistemi IoT. Nell'implementazione della demo dimostartiva, sfruttando la struttura a grafi, verranno implementate diverse operazioni di Update e Subscribe per la manipolazione ed il controllo dei dati contenuti nel server SEPA., tutte gestite autonomamente da microcontrollori.

1.0.2 SPARQL 1.1 SE Protocol

Il protocollo SPARQL viene introdotto dalla *W3C SPARQL Working Group* e permette di gestire, modificare e analizzare database di tipo sparso, come appunto i grafi RDF. Per questo motivo viene sfruttato all'interno dell'architettura SEPA, in quanto permette di gestire gli RDF Document con facilità. In particolare, il SEPA sfrutta quello che viene etichettato come SPARQL 1.1 SE Protocol, estensione dello SPARQL 1.1 Protocol. Questa estensione mette a disposizione, oltre alle due operazioni di Update e Query, una terza operazione, che è appunto quella di Subscribe. Le potenzialità del SEPA derivano in gran parte da questa evoluzione. Inoltre, introduce anche primitive basate su protocollo HTTPS. Da questo deriva l'aggiunta nel nome dell'acronimo SE, Secure Event, che indica appunto il maggior grado di sicurezza offerto da questa versione. Il protocollo definisce parametri e contenuti dei pacchetti HTTP/HTTPS da inoltrare al server per effettuare le operazioni richieste su un determinato grafo o per sottoscriversi al grafo di

interesse. Generalmente si tratta di stringhe di linguaggio SPARQL da inserire all'interno del Body di una richiesta.

Query

L'operazione di Query viene utilizzata per richiedere al SEPA determinati contenuti dell'RDF Graph, come ad esempio un intero grafo, un particolare oggetto o una particolare tripletta. Essendo la richiesta basata su protocollo HTTP, segue il modello client/server:

- Il client è colui che inoltra una HTTP Request e riceve la risposta dal server con le informazioni richieste. Nel caso trattato in questo lavoro di tesi si tratta, ad esempio, del microcontrollore.
- Il Server è colui che riceve la richiesta, la elabora e invia una risposta contenente i dati richiesti. Nel caso di tesi si tratta del server SEPA, in cui è contenuto l'RDF Document.

La richiesta può essere invocata usando i metodi GET o POST: la differenza tra i due risiede nel modo in cui i parametri vengono inoltrati al server. Il corpo di una richiesta è inoltre suddiviso in varie parti. Tra quelle di interesse abbiamo l'*Header* e il *Body*. Affinchè il SEPA interpreti correttamente una richiesta, vengono messi a disposizione tre diversi metodi per effettuare una Query, mostrati in Figura 1.5.

	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 1.5: Metodi per Query

In particolare, il metodo più utilizzato è POST senza codifica ed è lo stesso che viene sfruttato nell'implementazione della demo dimostrativa. Esso deve essere composto come segue:

- Nell'Header viene indicato il tipo di richiesta che il server deve processare. Con la stringa *"application/sparql-update"* si va infatti ad indicare al server che il body dovrà essere interpretato come una stringa di Query in linguaggio SPARQL.
- Il body contiene la stringa in linguaggio SPARQL che dovrà essere interpretata ed eseguita dal server.

Il metodo in codifica URL consente di aumentare leggermente il grado di sicurezza della trasmissione del pacchetto, mentre quello in metodo GET non ha nessun grado di sicurezza ed è per questo scarsamente utilizzato.

Update

Il concetto alla base di un Update HTTP è lo stesso della Query. In questo caso però, il client inoltra una stringa SPARQL non per ricevere dei dati, ma per andare a modificare il grafo a seconda delle necessità. In questo caso, la W3C mette a disposizione soltanto due metodi per potersi interfacciare con il server. Essi vengono illustrati in Figura 1.6.

	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 1.6: Metodi per Update

Come si nota, viene eliminata la possibilità di effettuare una HTTP Request di tipo GET. Sono consentite soltanto richieste POST, una non codificata ed una in codifica URL. Come per le Query, nell'Header andrà specificato il tipo di richiesta, che in questo caso specifico è di tipo *"application/sparql-update"* e nel Body la stringa SPARQL di Update, che conterrà informazioni sul grafo da modificare e su come modificarlo (inserendo ad esempio una nuova tripla o modificando uno specifico nodo).

Generalmente le Update sono legate alle sottoscrizioni: un sistema esterno si sottoscrive infatti ad un determinata sezione del grafo per sapere istantaneamente quando esso viene modificato, senza dover continuamente effettuare delle query verso il server W3C [2013].

Subscribe

L'operazione di Subscribe viene introdotta grazie al passaggio allo SPARQL SE 1.1 Protocol. Essa permette di sottoscriversi ad un certo grafo o nodo per poter ricevere una notifica ogni volta che quest'ultimo viene modificato. L'introduzione di questa operazione è fondamentale nell'efficenza del server SEPA e dei sistemi ad esso collegati; permette infatti un approccio di tipo *Interrupt* con il server. In altre parole non sarà necessarrio effettuare periodicamente delle richieste HTTP per poter conoscere lo stato di un dato, ma il server comunicherà autonomamente la variazione del dato a chiunque sia sottoscritto ad esso.
Le operazioni di tipo Subscribe non sono più basate sul protocollo HTTP, ma sfruttano le cosidette WebSocket. WebSocket è un protocollo Internet che consiste nella creazione di un canale Full Duplex tra due agenti (generalmente un client e un server) basato su una singola connessione TCP persistente che trasmette i dati in tempo reale. Il canale rimane attivo finchè uno dei due agenti non chiude la comunicazione o finchè non scade per Timeout. Oltre a creare un canale real time (appunto la socket), fa in modo che il canale non si esaurisca istantaneamente appena viene terminata la richiesta, mantenendolo attivo attraverso dei pacchetti di ping Melnikov [2011].

Le operazioni di subscribe sono legate a tre primitive, che vengono definite dallo SPARQL 1.1 Subscribe Language. Esse sono:

- Subscribe e Unsubscribe, attraverso le quali è possibile sottoscriversi o annullare l'iscrizione ad un determinato grafo/nodo.
- Notification, ovvero la ricezione di una notifica ogni volta che il dato a cui si è sottoscritti varia.

Le notifiche in seguito ad una sottoscrizione sono di tipo JSAP: seguono quindi la sintassi JSON. Questo fornisce un grosso vantaggio, in quanto,

attraverso un parsing della stringa, permette di accedere separatamente a tutte le informazioni in essa contenute.
L'operazione di Subscribe si divide in due fasi: nella prima fase, il sistema si sottoscrive al grafo, ricevendo in risposta dal server una stringa JSON che contiene lo stato attuale del grafo a cui si è sottoscritto. In altre parole, la prima notifica che si ottiene, non appena viene conclusa lo sottoscrizione, ha lo stesso contenuto di una query. La seconda fase, invece, riguarda l'attesa della ricezione di una notifica: il client ed il server manterranno attiva la *socket* attraverso dei pacchetti di tipo Ping, finchè necessario. In caso di variazione del grafo, il server invierà una stringa JSON contenente le informazioni sulla variazione avvenuta. Un esempio di stringa di notifica è mostrata in Figura 1.7.


```
{"notification":{
   "spuid" : "sepa://subscription/0d057ca5-cc10-4e8a-a5d9-59d7b36f71d6",
   "sequence" : 0,
   "alias" : "All",
   "addedResults" :{
     "head": { "vars": ["vaimee","deda","didi"] },
     "results": {
       "bindings": [{
        "vaimee": {"type": "uri", "value" : "http://wot.arces.unibo.it/example#Subject"},
        "deda": {"type": "uri", "value": "http://wot.arces.unibo.it/example#Predicate"},
        "didi": {"type": "literal","value": "ვაიმეე"} }]}},
   "removedResults":{}}}
```


Figura 1.7: Esempio di Notifica

Come si può notare, una file JSAP di notifica è formato da diverse chiavi. Le due più interessanti sono *"addedResults"* e *"removedResults"*. Esse contengono rispettivamente i valori che vengono aggiunti (generalmente con un operazione di Update di tipo INSERT) e quelli che invece vengono eliminati (attraverso operazioni di tipo DELETE). Inoltre è importante notare come per ogni dato che viene aggiunto/rimosso, viene riportato nel file JSON sia il valore che il tipo di dato, in modo che il sistema di elaborazione che riceve la notifica sia in grado di interpretare correttamente il valore ricevuto.

In conclusione, lo SPARQL SE 1.1 Protocol mette a disposizione tre operazioni fondamentali per operare con gli RDF Graphs contenuti nel SEPA:

- Operazioni di Query, basate su protocollo HTTP, che permettono di conoscere lo stato attuale di un determinato grafo o di una parte di esso.
- Operazioni di Update, basate su protocollo HTTP, che permettono di andare a modificare o aggiungere nodi o triplette ad un determinato grafo.
- Operazioni di Subscribe, basate su protocollo WebSocket, che permettono ad un sistema esterno di sottoscriversi ad un determinato grafo o nodo. Ogni volta che quest'ultimo viene modificato, il SEPA invierà una notifica ad ogni agente sottoscritto, contenente i parametri della modifica effettuata.

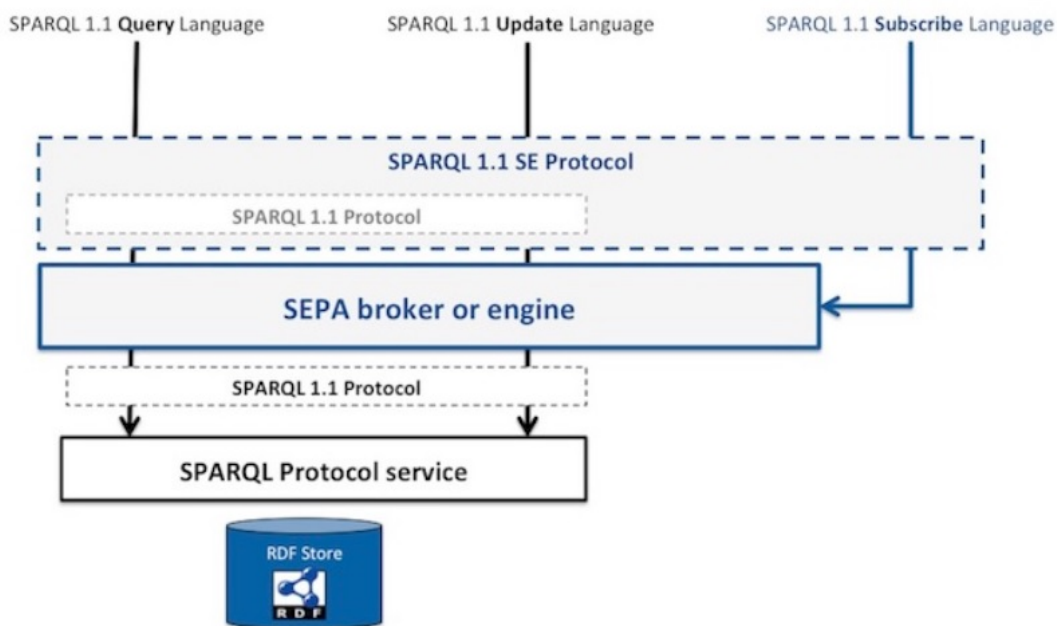

Figura 1.8: Architettura del SEPA

Queste tre operazioni vengono implementate attraverso l'architettura mostrata in Figura 1.8. Il SEPA Broker è l'elemento dell'architettura che consente di passare allo SPARQL SE 1.1 Protocol; esso si occupa delle operazioni di sottoscrizione, permettendo quindi di realizzare un server di tipo Publish-Subscribe. Il Broker ha il compito di elaborare le richieste di

sottoscrizione e di gestire poi le notifiche, andando a comunicare con lo SPARQL Protocol Service. Le operazioni di Query e Update non attraversano invece il broker Roffia et al. [2018b].

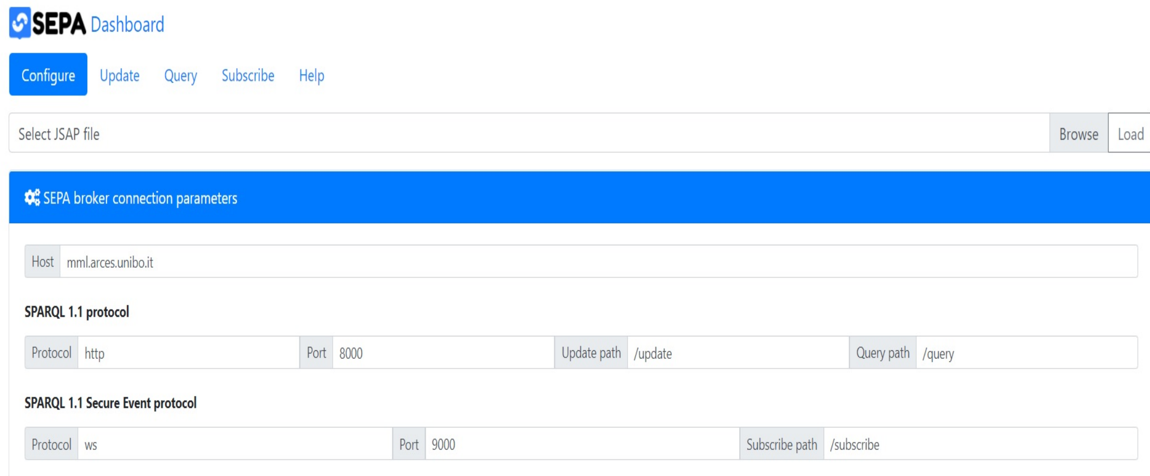

Figura 1.9: DashBoard di gestione del SEPA

Il SEPA mette a disposizione anche un'interfaccia grafica online per poter controllare lo stato del server e per poter gestire operazioni di Update, Query e Subscribe in maniera manuale. Un'immagine della schermata è mostrata in Figura 1.9.

Affinchè le operazioni vadano a buon fine, non è sufficente conoscere solo l'indirizzo IP del server; è importante che siano note anche le porte su cui comunicano i servizi del server. Di default, le richieste di tipo HTTP (Update e Query) sono gestite da un servizio sulla porta 8000, mentre le Subscribe (che sfruttano il protocollo WebSocket) devono essere dirette verso la porta 9000. Per questo motivo sono molto importanti i file JSAP di configurazione, che forniscono ai client tutte le informazioni necessarie per poter effettuare le richieste con successo.

1.0.3 I servizi REST

L'espressione servizio REST, dove REST è acronimo di *REpresentational State Transfer*, fu introdotta intorno al 2000. Indica uno stile di architettura software all'interno di un sistema hypermedia distribuito. Un'architettura di questo tipo è volta ad ignorare i dettagli dell'implementazione dei componenti e dei protocolli utilizzati, concentrandosi prevalentemente sul ruolo dei vari componenti nel sistema, sulla loro interazione con altri componenti e la loro interpretazione.

Un'architettura REST segue il paradigma informatico del SoC (*Separetion of Concerns*). Esso è un principio di progettazzione in cui si punta a dividere il sistema in blocchi distinti, ognuno con il proprio compito. Questo diagramma di tipo SoC viene applicato ad un'architettura di tipo Client-Server. Il Server dovrà ascoltare le richieste del Client e offrire una o più funzionalità a quest'ultimo. Un client invoca un servizio messo a disposizione dal server, inviando il corrispondente messaggio di richiesta. Il server esegue la richiesta, inviando successivamente la risposta al client. La gestione delle eccezioni è delagata al client. Nelle architetture REST è importante il concetto di **Stateless**: la comunicazione tra server e client deve essere senza stato tra le richieste. In altre parole, tutte le richieste inviate da un client devono contenere tutte le informazioni necessarie per comprendere il significato della richiesta legata ad un servizio specifico. Ogni richiesta è come se fosse la prima e non deve essere correlata a nessuna richiesta precedente. Questo porta ad una grande semplicità nella realizzazione del server, in quanto dovrà elaborare soltanto le richieste presenti, senza dover tener traccia di quelle già avvenute. Un'altro importante concetto risiede nella manipolazione delle risorse attraverso rappresentazioni, ovvero ogni risorsa, dato, elemento nel server viene rappresentato in maniera specifica, ad esempio attraverso l'utilizzo della sintassi JSON. In questo modo, un client può interagire con quella risorsa attraverso la sua rappresentazione (nel caso di rappresentazione JSON, ad esempio andando a modificare una specifica key attraverso una richiesta al server). L'implementazione di servizi REST è molto importante nel mondo dell'IoT con sistemi Embedded, in quanto, per la sua semplicità architetturale, permette di utilizzare risorse minime per gestione e configurazione del sistema Fielding [2000].

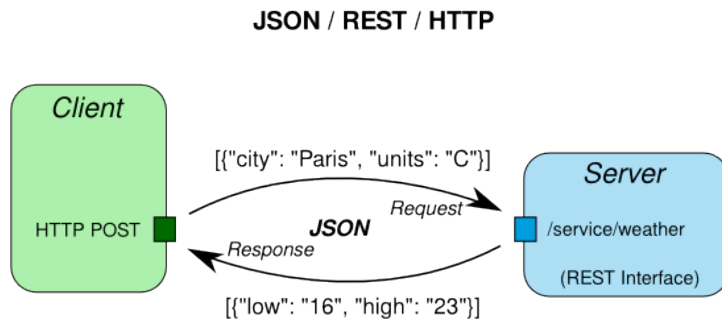

Figura 1.10: Schema di Architettura REST in JSON

Nell'attività di tesi, un'architettura di tipo REST verrà utilizzata per la configurazione del sistema, in modo da renderlo il più autonomo possibile. Il server REST verrà realizzato attraverso l'utilizzo di Arduino Uno. Effettuando una richiesta al server, esso risponderà con un file JSON contenete tutti i parametri necessari per la configurazione del client (ad esempio indirizzo IP del server SEPA, porte attive, ecc.).

Capitolo 2

Arduino Uno Rev 3 ed ESP8266

2.0.1 Sistemi Embedded e Arduino

Il termine *Sistema Embedded* identifica generalmente tutti quei sistemi elettronici di eleaborazione a microprocessore progettati appositamente per un determinato utilizzo. Generalmente sono dispositivi non riprogrammabili, con un hardware progettato ad hoc per relazzire le funzioni richieste. La scelta di utilizzo di schede della famiglia Arduino permette di avere un hardware completo da poter adattare alle varie necessità di progetto, attraverso la possibilità di riprogrammazione, mantenendo intatto il concetto di sistema Embedded.

Figura 2.1: Logo ufficiale di Arduino

La famiglia Arduino nasce nel 2005 presso l'Interaction Design Institute Ivrea in provincia di Torino. Al giorno d'oggi è la più grande piattaforma software/hardware open-source basata sull'utilizzo dei microcontrollori. Un microcontrollore è un dispositivo elettronico integrato su singolo chip e dotato

generalmente di una CPU, un clock interno e vari banchi di memoria. Attraverso i pin di I/O inoltre, riesce ad interagire con l'ambiente circostante, attraverso la connessione con sensori e/o attuatori, che vengono controllati via software. Per questo motivo si adatta molto alla realizzazione di sistemi embedded, in particolare in applicazioni di controllo digitale e IoT Ard [2018].
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. Il linguaggio di programmazione è di tipo C, con alcune varianti necessarie per ottimizzare prestazioni e utilizzo su microcontrollore.
Questa grande versatilità ha permesso la diffusione di queste schede in tutto il mondo, in particolare per la facilità di utilizzo (sia a livello software che hardware) e il costo moderato. Questo fornisce un grandissimo potenziale di utilizzo in molti ambiti di sviluppo.
Esistono molti modelli di Arduino, che si differenziano per dimensioni, architettura e potenzialità; per il progetto è stato scelto inizialmente Arduino Uno Rev 3, per poi passare alle schede ESP8266, per vari motivi che verranno illustrati nei capitoli successivi.

2.0.2 Arduino Uno Rev 3 ed Ethernet Shield v. 1.0

Arduino Uno Rev 3 è la scheda più basica e diffusa che viene offerta dalla famiglia Arduino, oggi arrivata alla sua terza versione (da qui Rev 3). Si tratta di una scheda a microcontrollore basata su ATMega 328P. Contiene tutto il necessario per poter supportare il microcontrollore. Allo stato base è ottima per la realizzazione di sistemi di sensorista e attuatori, ma non è predisposta per connessioni alla rete, ne' di tipo Ethernet, ne' di tipo WiFi.
Le caratteristiche principali, oltre al microcontrollore sopra menzionato, sono:

- 14 pin di I/O, di cui 6 utilizzabili in PWM.
- 6 ingressi analogici.
- Risonatore ceramico a 16MHz (CSTCE16M0V53-R0).
- Porta USB di tipo B, utilizzabile sia per l'alimentazione, ma soprattutto per la connessione al PC.

Figura 2.2: Arduino Uno Rev 3

- Jack di alimentazione esterna (con una batteria ad esempio).
- Un pulsante di Reset per il microcontrollore
- Memoria Flash di 32KB (è quella che contiene il programma ed è uno dei problemi di questa scheda).
- SRAM di 2KB.
- EEPROM di 1KB.
- Dimensioni : 68,6 x 53,4 (Lunghezza x Larghezza).
- Peso : 25g.

Attraverso il cavo USB è possibile non solo programmare, ma ci permette di leggere a monitor il Serial Output della scheda. Arduino Uno permette anche di implementare alcuni protocolli di comunicazione; tra questi è importante ricordare il protocollo SPI e ICMP, che sono fra i più utilizzati soprattutto

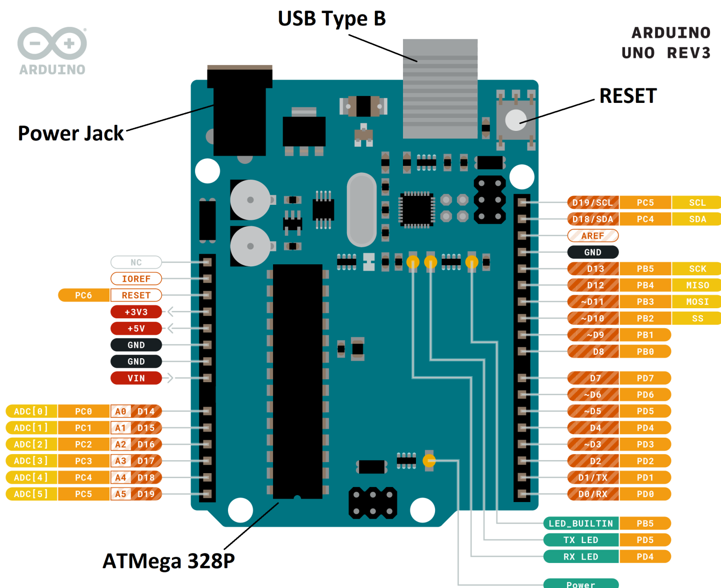

Figura 2.3: Schematico di scheda Arduino Uno Rev 3

per la comunicazioni con schermi esterni,sensori e Shield (ovvero schede di espansione collegabili alla scheda principale) Arduino [2021].

Come detto, l'utilizzo del solo Arduino Uno non consente una connessione in rete. Viene quindi corredata da quello che, nella famiglia Arduino, viene chiamato Ethernet Shield, ovvero una scheda di espansione che integra una connessione di tipo Ethernet.
L'Etherner Shield è basato su Wiznet W5100, chip Ethernet che consente l'utilizzo dei protocolli TCP e UDP e quindi l'utilizzo dei più comuni protocolli Internet. Per il funzionamento dello schield è necessario l'utilizzo della libreria Ethernet.h, fornita direttamente da Arduino. Le caratteristiche principali della scheda di supporto sono:

- Supporto a IEEE802.3af, per cui è possibile un'alimentazione PoE (*Power over Internet*).
- Velocità massima di 100 Mb/s e minima di 10 Mb/s
- Ripple e Rumore in uscita massimo di 100 mVpp.
- Protezione a sovraccarichi e corto circuiti.

- 9V in Output
- Convertitore DC/DC ad alta efficenza.
- Connessione attraverso classico cavo RJ45.
- Tasto di Reset, che resetta sia lo shield che Arduino Uno su cui è connessa.
- Lettore di Schede SD integrato

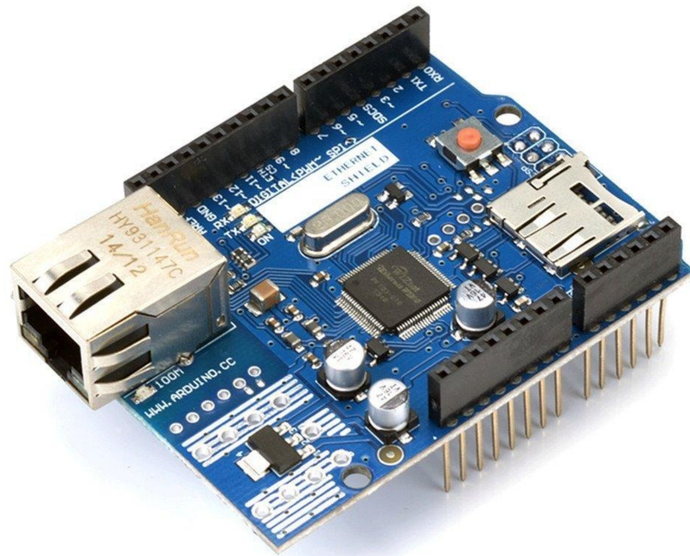

Figura 2.4: Ethernet Shield v 1.0

L'Ethernet Shield viene montato direttamente sulla scheda Arduino. Occupa soltanto i pin 10,11,12 e 13 per la comunicazione in protocollo SPI, dove il pin 10 è utilizzato come CS (Chip Select), Inoltre utilizza il pin 4 nel caso in cui venga sfruttato il lettore di schede SD integrato sullo shield. Tutti gli altri pin possono essere utilizzati normalmente. La presenza dello slot per schede SD è molto utile, in particolare nell'utilizzo del sistema come

server, in quanto mette a disposizione una memoria aggiuntiva (massimo 2 GB) per contenere file da utilizzare in risposta a richieste di tipo GET o POST.

L'hardware completo utilizzato per interfacciare Arduino Uno al server SEPA è quindi composto dalla scheda base Arduino Uno Rev 3 corredata dell'Ethernet Shield montato su di esso, come si può vedere in Figura 2.5.

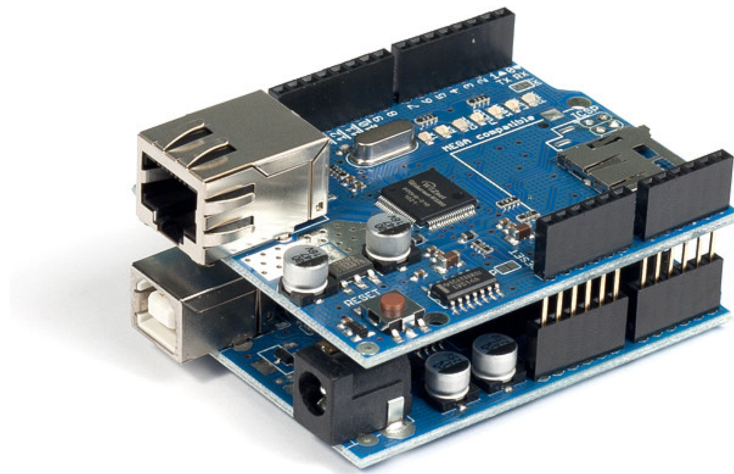

Figura 2.5: Hardware per interfacciamento Arduino Uno

Come detto nel capitolo precedente, le possibili operazioni che si possono effettuare sul server SEPA sono Update, Query e Subscribe. Durante i test effettuati su questo sistema, sono sorti alcuni problemi che hanno costretto l'abbandono di questo harware. In particolare, l'implementazione delle operazioni di subscribe risultano, al momento, troppo complesse per questo modello di Arduino, in particolare per due motivi specifici:

- La mancanza di librerie specifiche adatte all'implementazione dei protocolli di WebSocket su microcontrollore ATMega328P.
- Memoria Flash insufficiente per contenere programmi che riescano a gestire le notifiche di una Subscribe.

Le librerie messe a disposizione dagli utenti Arduino, per essere adattate all'hardware, implementano le Subscribe con un metodo a pollig, attraverso chiamate continue al server; in questo modo si perdeva il concetto fondamentale di un sistema *Publish-Subscribe*, nonchè il suo più grande potenziale. Inoltre scrivendo un programma che implementasse una Subscribe, la memoria programma raggiungeva livelli di occupazione del 90/95 %, il che portava ad un instabilità del sistema stesso. L'utilizzo di Arduino Uno Rev 3 può essere quindi limitato soltanto alle operazioni di Update e Query; al momento, quindi, questo modello non riesce a supportare l'operazione di Subscribe. Sarebbe necessario sviluppare una libreria ad hoc che implementi questa funzionalità, ma questo esula dall'obiettivo di questo lavoro di tesi.

Alla luce di queste considerazioni, si è deciso di passare all'utilizzo di dispositivi della famiglia ESP, in particolare dell'ESP8266, che mette a disposizione un hardware più prestante per le comunicazioni internet, una memoria più capiente ed un modulo WiFi integrato sulla scheda stessa, nonchè dimensioni molto ridotte. Questa scheda non appartiene alla famiglia Arduino, ma è stata scelta perchè permette la programmazione attraverso l'IDE Arduino, mantenendo quindi la facilità di utilizzo che si aveva con Arduino. Il modello Arduino Uno verrà comunque utilizzato come REST Server, introducendo all'interno del nostro sistema il concetto di sistema stand-alone.

2.0.3 ESP8266: Modulo ESP-01

L'ESP8266 è un chip con WiFi integrato prodotto dall'azienda cinese Espressif Systems e che si è immesso prepotentemente nel mercato intorno al 2014 per le sue alte prestazioni e per il suo costo molto basso. Ha un supporto completo al protocollo TCP/IP, nonchè alla stragrande maggioranza dei protocolli presenti su Internet. Per questi motivi è adatto alle applicazioni di tipo IoT. La casa costruttrice propone un SDK ufficiale per la programmazzione del chip, ma ci sono molte alternative che permettono una programmazzione attraverso altri SDK non ufficiali. Tra questi è presente proprio Arduino IDE. Questo è uno dei motivi per cui la scelta è ricaduta su questo modello EspressifSystem [2021].
Le principali caratteristiche del chip sono illustrate di seguito:

Figura 2.6: Chip ESP8266 della Espressif Systems

- Microprocessore RISC L106 a 32 bit, funzionante a 80 MHz.
- 64 kiB di RAM per la memoria programma.
- 96 kiB di RAM per la memoria dati
- IEEE 802.11 b/g/n WiFi, con autenticazione WEP, WPA/WPA2 e reti aperte.
- 16 pin GPIO.
- Supporto a protocolli SPI, e I^2C.
- Interfaccia I^2S con DMA, per la condivisione dei pin con GPIO.
- UART su pin dedicati.
- ADC ad approssimazioni successive a 10 bit.

In commercio, il chip si trova integrato in moduli che lo corredano di tutto l'hardware necessario per il corretto utilizzo; nell'attività di ricerca svolta è stato utilizzato il modulo ESP-01, mostrato in Figura 2.7. La scheda è fra le

più diffuse, perchè racchiude tutta la potenza del chip ESP8266 in un modulo di dimensioni molto ridotte, di circa 25mm x 15mm (altezza x larghezza). Esso è corredato di un antenna per la comunicazione WiFi e di un chip integrato per il controllo dell'alimentazione, oltre che dei pin di I/O.

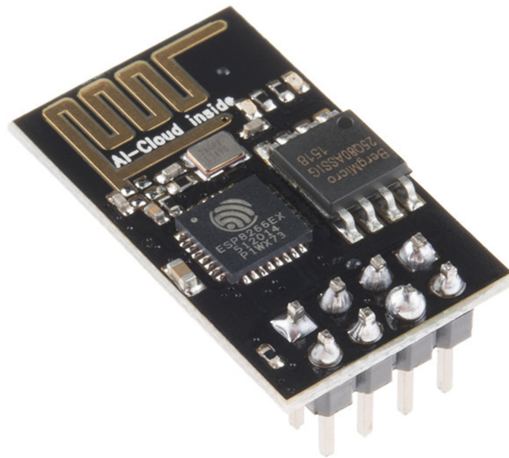

Figura 2.7: Modulo ESP-01

Uno dei limiti della scheda è proprio il numero di pin in uscita messi a disposizione dell'utente, che è molto ridotto rispetto ai pin di GPIO disponibili direttamente sul chip ESP8266. Questa scelta è legata direttamente alle dimensioni della scheda, che sarebbero molto più ingombranti nel caso di un maggiore numero di pin I/O disponibili.

I pin a disposizione sono in totale otto e si dividono in:

- Pin di Alimentazione a 3.3 V [8] e GND [1].
- RESET [6]: quando su questo pin viene rilevato un fronte di salita, manderà in reset il chip.
- Chip Enable [4], per abilitare la modalità di programmazzione.
- UART TX [2] ed RX [7], pin GPIO che possono essere sfruttati come Tx e Rx dell'UART e che ci permetteranno di visualizzare i dati a schermo
- 2 General Pourpouse I/O (GPIO)[3 e 5], utilizzabili per la lettura di sensori o per la gestione di attuatori

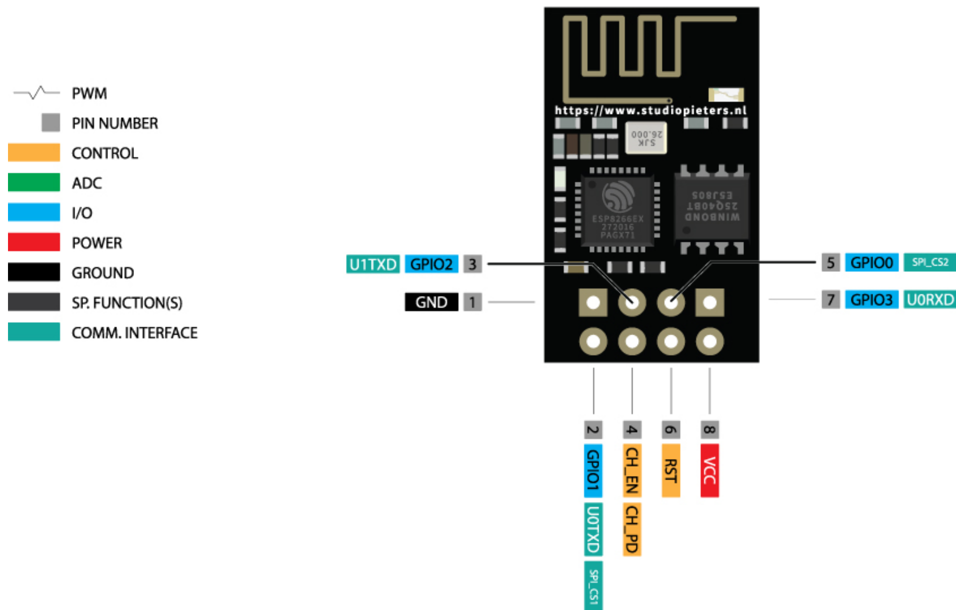

Figura 2.8: Schematico ESP-01

(una piccola overview dei pin è mostrata in Figura 2.8).

La particolarità del modulo è il metodo di programmazzione, che non è del tutto banale. Per la comunicazione tra software ed ESP8266 è necessario l'utilizzo di un interfaccia, in quanto la scheda ESP-01 non presenta una porta di tipo USB che ne consenta il collegamento diretto al PC. Ci sono diversi metodi di interfacciamento, ma quello più semplice è l'utilizzo di Arduino Uno, attraverso il collegamento mostrato in Figura 2.9:

- Rosso : Vcc connesso a 3.3V.
- Nero : GND connesso a GND.
- UART TX dell'ESP-01 connesso al TX di Arduino (Marrone) e UART RX dell'ESP-01 connesso all'RX di Arduino (Bianco).
- Arancione: RESET connesso al Reset di Arduino

La differenza tra le due configurazioni è solo nel collegamento tra Chip Enable e GND; quando questa connessione è presente, il chip ESP8266,

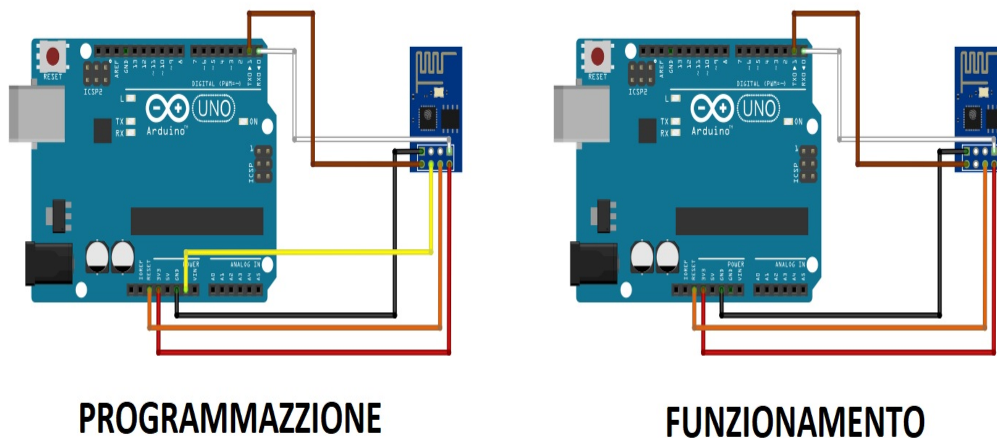

Figura 2.9: Collegamento tra Arduino Uno e ESP-01

trovando un valore basso sul pin, entra in modalità di programmazzione e sulla comunicazione UART viene trasferito il programma da eseguire. Quando invece la connessione è assente, il chip esegue il programma che è contenuto nella RAM e utilizzerà la comunicazione UART per trasferire ad Arduino, e quindi al PC, i dati da mostrare sul monitor seriale. Nel caso in cui non ci sia interesse di controllare i dati via seriale, la scheda può essere alimentata separatamente; una volta terminata la programmazzione, non sarà quindi più necessario l'ausilio di Arduino Uno. Inoltre, in questo modo avremo a disposizione due GPIO supplementari.

Un possibile sostituto del modulo ESP-01 è mostrato in figura Figura 2.10. Si tratta della NodeMCU Amica v. 2.0 e viene proposta come possibile soluzione al numero ridotto di GPIO e alla difficoltà di programmazzione dell'ESP-01.

I due moduli sono perfettamente intercambiali: montano entrambi lo stesso chip ESP8266, quindi un programma funziona ugualmente su entrambi. La scelta di uno o dell'altro e principalmente legata alla necessità di un certo numero di pin di GPIO: nel caso siano necessari meno di due pin, è consigliabile utilizzare il modulo ESP-01, visto il minor ingombro e prezzo, nonostante la difficoltà di programmazzione. Viceversa, si può passare al NodeMCU. Nelle varie demo che verranno analizzate nei capitoli precedenti, è stato

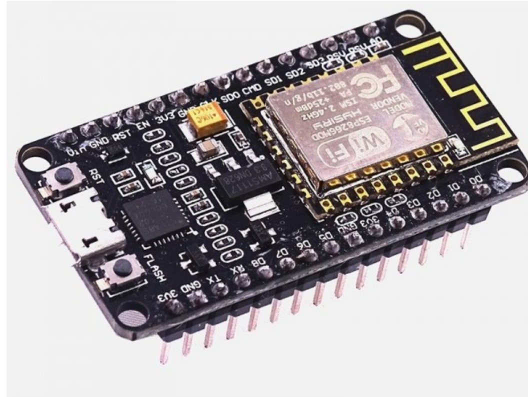

Figura 2.10: Modulo NodeMCU Amica v. 2.0

principalmente usato il modulo ESP-01, proprio per i motivi sopra citati, in quanto non c'era la necessità di un alto numero di pin GPIO.

2.0.4 Il Software: Arduino IDE

Arduino IDE è il software open-source messo a disposizione gratuitamente per la programmazione di tutte le schede della famiglia Arduino e, attraverso alcuni SDK esterni, anche di altri tipi di schede, come appunto l'ESP8266. La sigla IDE è acronimo di *Integrated Development Environment* ed indica appunto che è un ambiente di sviluppo per sistemi integrati. Il linguaggio di programmazzione è di tipo **C**, con alcuni accorgimenti per permettere il corretto funzionamento del programma in un microcontrollore.

Un grosso vantaggio fornito dall'Arduino IDE è la possibilità di utilizzare le *Librerie*: esse non sono nient'altro che parti di codice già pronte messe a disposizione degli utenti per utilizzare al meglio componenti o schede, come ad esempio sensori, schermi e anche l'ESP8266 stesso. Le librerie, oltre a rendere il programma più pulito e fluido, consentono di facilitare l'utilizzo di componenti esterni collegati ad un dispositivo Arduino, mettendo a disposizione classi, oggetti e metodi (caratteristiche importanti della programmazzione in C++) che consentono l'utilizzo di dispositivi anche

complicati senza conoscerne a fondo il funzionamento. Le librerie possono essere ufficiali e vengono quindi fornite direttamente a corredo dell'IDE stesso, oppure possono essere non ufficiali, quindi scritte da utenti che hanno deciso di condividere il loro lavoro con chiunque sia interessato. Le librerie non native possono essere importate attraverso una sezione specifica del software, il *Gestore Librerie*

In gergo, un programma generato con Arduino IDE prende il nome di *"Sketch"*. Un esempio è mostrato in Figura 2.11.

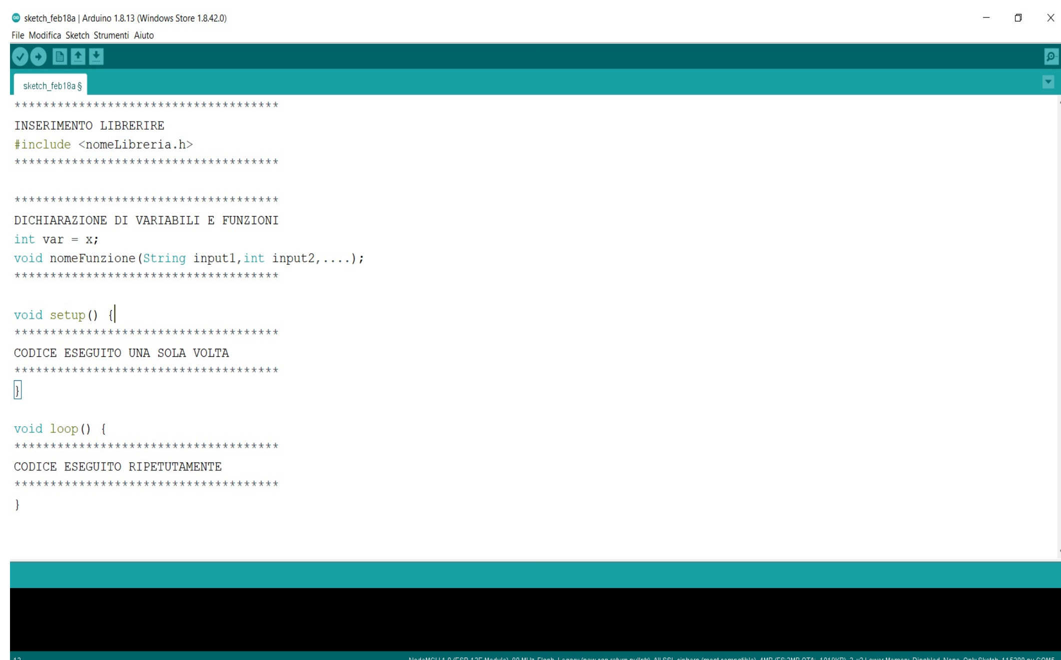

Figura 2.11: Schermata vuota di Arduino IDE

Come si può notare, uno sketch è suddiviso in 4 parti fondamentali:

- Dichiarazione delle librerie, dove è necessario includere le librerie che si vogliono utilizzare nel programma.
- Dichiarazione delle variabili e funzioni, caratteristica molto simile al linguaggio C, dove si dichiarano variabili globali e funzioni scritte dall'utente che si intende utilizzare

- **Void Setup**, che contiene la parte del programma che verrà eseguita una sola volta all'avvio del dispositivo. Generalmente viene utilizzato per inizializzare i componenti o per le configurazioni in generale (come ad esempio la connessione ad una rete WiFi)

- **Void Loop**, che contiene la parte del programma che verrà eseguita iterativamente fino alla disattivazione del sistema.

In particolare il Void Loop è una di quelle paticolarità introdotte necessariamente per permettere l'utilizzo del linguaggio C su un microcontrollore. Quest'ultimo infatti, non può eseguire un programma una sola volta, altrimenti perderemmo il concetto di microcontrollore stesso. In questo modo, possiamo inserire all'interno del loop una parte di programma che continuamente analizzerà lo stato del sistema e prenderà delle decisioni in output in base allo stato degli input attuale, che chiaramente può variare nel tempo.

Uno degli scopi dell'attività di tesi è quello di utilizzare l'Arduino IDE per creare una libreria Open-Source che renda le funzioni create per la realizzazione della demo fruibili da chiunque voglia utilizzare microcontrollori della famiglia Arduino per interfacciare i propri sistemi con il SEPA. Queste funzioni verranno illustrate nel capitolo 3 di questo elaborato.

Capitolo 3

Esempio di Funzionamento

Per dimostrare la possibilità di implementare un sistema di interfacciamento SEPA basato su microcontrollori della famiglia Arduino, viene proposta in questo capitolo una demo; in particolare saranno presenti due microcontrollori: ESP-01 che, assieme ad un sensore DHT11 costituisce il Nodo 1, e Arduino Yun che rappresenta il Nodo 2. Essi si scambieranno informazioni su Temperatura e Umidità nella stanza, interfacciandosi con il SEPA attraverso le funzioni di Update e Subscribe. Sarà inoltre presente un REST Server, implementato su Arduino UNO, che consentirà la configurazione autonoma del Nodo 1. I vari agenti sono connessi ad una rete locale LAN/WLAN; lo schema a blocchi dell'intero sistema è mostrato in Figura 3.1.

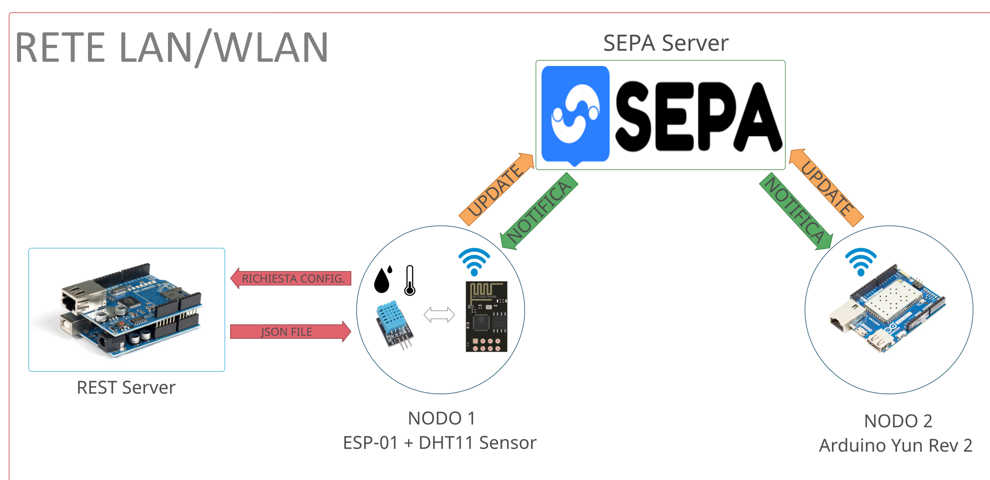

Figura 3.1: Schema a blocchi del sistema

Il funzionamento può essere suddiviso in quattro fasi fondamentali:

1. Inizialmente il Nodo 1 non ha nessuna informazione sul server SEPA (indirizzo IP, porte, ecc.). In memoria ha soltanto l'indirizzo IP del REST Server. Come primo passo si connetterà quindi al servizio REST e riceverà in risposta un file di configurazione JSON. In questo modo entrerà a conoscenza di tutte le informazioni necessarie per contattare il server.

2. A questo punto, entrambi i nodi client (Nodo 1 e Nodo 2) si sottoscriveranno al grafo; in particolare il Nodo 2 si sottoscriverà alla parte del grafo che contiene le informazioni su temperatura e umidità; il Nodo 1, invece, alla parte del grafo in cui vengono inserite le notifiche di ricezione.

3. Completata questa fase di configurazione, il Nodo 1 inizierà a misurare i valori di temperatura e umidità, ogni 30 secondi. Se questi hanno una variazione significativa, invierà un Update con i nuovi dati al grafo. In questo modo il SEPA invierà una notifica al Nodo 2, contenente i nuovi valori misurati.

4. Il Nodo 2 agirà sugli output in base ai valori ricevuti. Effettuerà quindi un Update per indicare che il dato è stato ricevuto correttamente. Il Nodo 1 riceverà la notifica, mostrerà a schermo un messaggio di conferma ed il ciclo sarà così completato.

Il grafo utilizzato per la realizzazione della demo è identificato dal graph name *demoSEPINO* e viene mostrato in Figura 3.2.

Nelle pagine successive verranno analizzati separatamente i vari blocchi, approfondendo in particolare il Nodo 1 ed il REST Server, su cui è incentrato questo elaborato. É importante ribadire che il sistema in questione funziona su una rete locale; il server SEPA viene infatti eseguito su un PC all'interno della rete e funziona attraverso due programmi Java messi a disposizione degli utenti dall'Arches Group: *blazegraph.jar* ed *engine-0-SNAPSHOT.jar*.

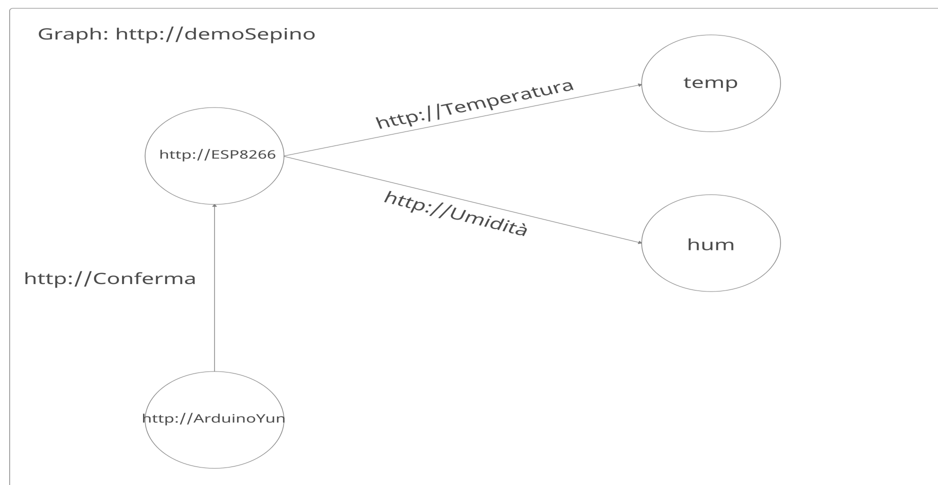

Figura 3.2: Grafo demoSepino

REST Service su Arduino UNO

Come sottolineato nei capitoli precedenti, il modello Arduino Uno Rev 3 supportato da Ethernet Shield, allo stato attuale, non è in grado di supportare tutte le possibili operazioni di interfacciamento con il SEPA. In particolare, trova parecchie difficoltà con l'operazione di Subscribe per 2 motivi principali, legati all'implementazione della WebSocket e a problemi di memoria. Nonostante questo, le operazioni di Update e Query funzionano correttamente.

Si è deciso quindi di utilizzare Arduino Uno per realizzare il REST Server. Sfruttando la connessione Ethernet e la scheda SD, andrà infatti a condividere con i vari nodi che lo richiedono un file di configurazione in JSON. In Figura 3.3 viene riportato l'hardware utilizzato per la realizzazione della demo.

Nella prima parte del codice, seguendo la sintassi e la forma illustrata nel capitolo 2, vengono dichiarate le librerie utilizzate; in particolare la libreria *SPI.h* implementa un protocollo di comunicazione necessario per l'utilizzo dell'Ethernet Shield; *Ethernet.h* e *SD.h* sono librerie native di Arduino rispettivamente per l'utilizzo della connessione Ethernet e della scheda SD. Successivamente sono dichiarate le variabili **mac** e *ip*, che contengono l'indirizzo mac e l'indirizzo IP su cui sarà inizializzato il server. A questo punto viene creato un oggetto Ethernet di nome **server**, inizializzato sulla

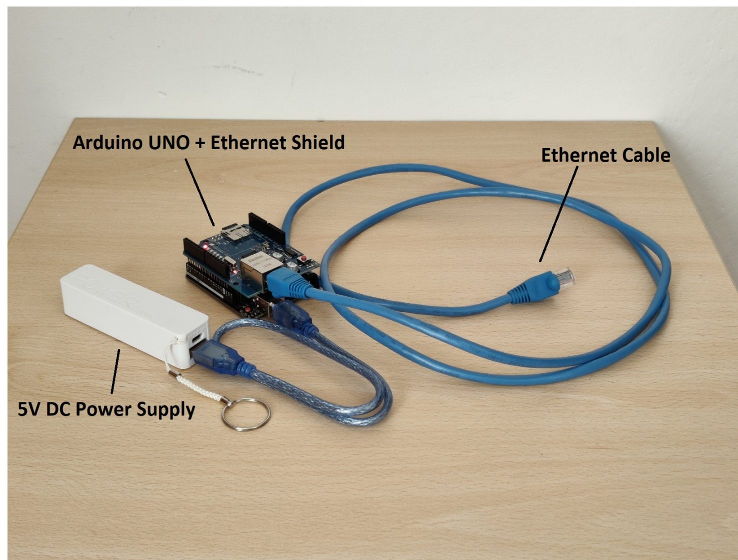

Figura 3.3: Hardware del REST Service su Arduino Uno

porta 80, in quanto viene utilizzato il protocollo HTTP. Infine si crea una variabile di tipo File, che conterrà il JSON presente in memoria.

```
/******DICHIARAZIONE LIBRERIE********/
#include <SPI.h>
#include <Ethernet.h>
#include <SD.h>
/************************************/


/*DICHIARARAZIONE VARIABILI E OGGETTI*/
//MAC e IPAddress che verranno assegnati al server in configurazione
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 30);

//porta su cui sarà attivo il server REST(HTTP Protocol:porta 80)
EthernetServer server(80);

//variabile di tipo FILE che conterrà il file JSON di configurazione
File myFile;
/************************************/
}
```

Dopo di che inizia il void Setup dello Sketch, che, come ricordiamo, verrà eseguito una sola volta all'avvio del dispositivo. Nel setup sono presenti:

- L'inizializzazione della porta seriale, necessaria per visualizzare i messaggi di sistema a schermo.
- L'inizializzazione della connessione Ethernet, attraverso i metodi forniti dalla libreria. Dato che occorre conoscere preventivamente l'indirizzo

del server, viene utilizzato un IP statico, univoco nella rete, ignorando l'assegnazione tramite DHCP.

- L'inizializzazione della scheda SD, in modo che sia pronta per l'apertura del file.

```
void setup() {
  //inizializzazione della porta seriale, per leggere i messaggi di
  ↪  avanzamento a schermo
  Serial.begin(9600);

  //inizializzo la connessione Ethernet
  Serial.println("Inizializzazione della connessione Ethernet...");
  Ethernet.begin(mac, ip); //metodo della libreria per l'inizializzazione

  //inizializzo il server
  server.begin();
  Serial.println("Server inizializzato all'indirizzo: ");
  Serial.println(Ethernet.localIP());
  delay(1000);

  //inizializzo la scheda sd
  Serial.println("Inizializzazione della scheda SD...");
  if (!SD.begin(4)) {
      Serial.println("Errore nell'apertura della scheda SD");
   }
  else{ Serial.println("SD inizializzata correttamente");}
  delay(1000);
}
```

Il comando *delay(n)* viene utilizzato per attendere n millisecondi, in modo da dare il tempo alle inizializzazioni effettuate di concludersi.

Come ultimo, viene illustarto il void Loop, eseguito iterativamente dal microcontrollore.

```
void loop() {

```

```
3    //creazione di un oggetto client, se un qualche client esterno contatta
     ↪  il server
4    EthernetClient client = server.available();
5    if (client) {//se abbiamo un client
6      //si avvia una routine per la gestione della richiesta e l'invio del
       ↪  File
7      boolean currentLineIsBlank = true;
8      while (client.connected()) {
9        if (client.available()) {
10         char c = client.read();
11         Serial.write(c); //mostra il messaggio di richiesta del client  a
           ↪  schermo
12         if (c == '\n' && currentLineIsBlank) { //quando la richiesta è
           ↪  terminata
13
14           /************INVIO DELL'HEADER************/
15           client.println("HTTP/1.1 200 OK");
16           client.println("Content-Type: text/plain");
17           client.println("Connection: close");
18
19           /***********APERTURA DEL FILE NELLA SD**********/
20           myFile = SD.open("test.txt");
21           int n = myFile.available(); //inserisco la lunghezza del file in
             ↪  una variabile
22           /********************************************/
23
24           //concludo l'header inviando la lunghezza del body che verrà
             ↪  inviato
25           client.println("Content-Length: " + String(n));
26           client.println();
27           /******************************************/
28
29           /********INVIO DEL BODY******************/
30           if (myFile) {
31             while (myFile.available()) { //finchè ci sono caratteri
               ↪  disponibili, li invia
32                 char tosend = myFile.read();
33                 client.write(tosend);
34             }
35             //una volta inviato tutto il file, viene chiuso
36             myFile.close();
37             Serial.println("File Inviato correttamente!");//messaggio di
               ↪  conclusione invio
```

```
        }

        break;
      }
      //parte di codice necessaria per indiviaduare la fine della
      ↪  richiesta del client
      if (c == '\n') {
        currentLineIsBlank = true;
      }
      else if (c != '\r') {
        currentLineIsBlank = false;
      }
    }
  }
  //concluso l'invio del file, si disconnette il client
  delay(1);
  client.stop();
  Serial.println("Client disconnesso");
 }
}
```

In questa parte dello sketch Arduino, il microcontrollore attende che un client si connetta al server. Una volta ottenuta una connessione, legge il testo della richiesta e la mostra sul monitor seriale. Attende che essa si concluda, controllando i caratteri speciali **\n, \r** e **carattere nullo**. Dopodichè invia l'header del messaggio di risposta, indicando la versione del protocollo e il metodo di connessione. Inoltre apre il file contenuto nella scheda SD e invia come conclusione dell'header la lunghezza del body, che sarà la stessa del file. A questo punto inizia a leggere il file, inviando carattere per carattere le informazioni di configurazione contenute nel JSON. Quando non ci sono più caratteri disponibili, chiude il file, mostra a schermo un messaggio di informazione e chiude la connessione con il client. In Figura 3.4 viene mostrato un esempio di file JSON di configurazione.

L'implementazione di questo nodo all'interno del sistema è molto importante per indirizzare il sistema verso lo stand-alone. Grazie a ciò, infatti, non è necessario riprogrammare i dispositivi ogni volta che cambia la configurazione del server centrale, ma basterà modificare il file JSON, che verrà autonomamente condiviso nella rete per permettere l'autoconfigurazione

```
{
    "host" : "192.168.1.30",

    "graph" : "<http://demoSepino>",

    "update" : {
          "port" : 8000,
          "path" : "/update",
          "header" : "application/sparql-update",
     },

     "query" : {
           "port" : 8000,
           "path" : "/query",
           "header" : "application/sparql-query",
      },

      "subscribe" : {
            "port" : 9000,
            "path" : "/subscribe",
       }
}
```

Figura 3.4: File di configurazione in sintassi JSON

di tutti i nodi interessati.
In figura Figura 3.5 viene mostrato l'output video del server REST a seguito di una connessione ad un client.

Nodo 1: ESP-01 e DHT11

Il nodo si occupa della rilevazione di temperatura e umidità, effettuando degli update al grafo demoSepino ogni volta che si misurano scostamenti considerevoli dai valori precedentemente inviati. Inoltre si occupa della gestione delle notifiche di ricezione, sottoscrivendosi all'apposita tripletta nel grafo. Ogni volta che viene ricevuta una notifica, mostrerà a schermo chi l'ha inviata e il contenuto di quest'ultima.

Per la realizzazione del programma viene utilizzato il modulo *Sepino.h*, che contiene le funzioni per implementare Update, Query e Subscribe su ESP8666.

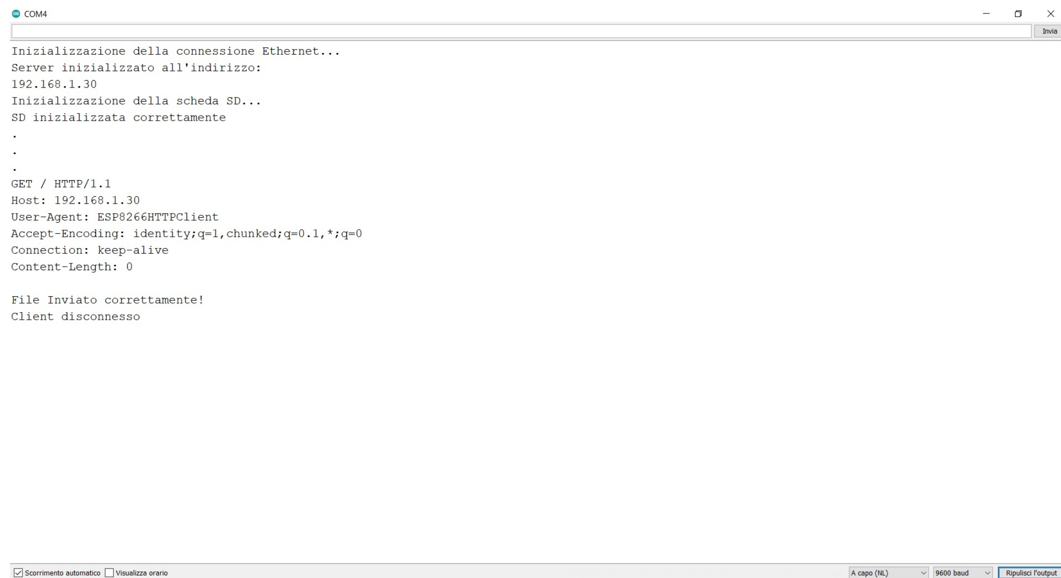

Figura 3.5: Output video del server REST

Inoltre si sfrutta la libreria *ArduinoJson*, per il parsing del file di configurazione e delle notifiche, e la libreria *DHT.h*, fornita da Adafruit Sensor per la gestione del sensore. Nella parte di dichiarazione vengono inoltre create diverse variabili, utilizzate successivamente nel programma. Alcune di esse sono sfruttate per la temporizzazione del sistema, altre per la lettura delle grandezze fisiche e altre ancora per la memorizzazione di stringhe, come ad esempio la password del WiFi e l'indirizzo IP del server REST. Inoltre sono presenti stringhe vuote che verranno successivamente riempite con le informazioni ricevute dal file JSON.

```
1  //inclusione delle librerie utilizzate dal sistema
2  #include "Sepino.h"
3  #include <Adafruit_Sensor.h> //librerie per la lettura del sensore
4  #include <DHT.h>
5  #include <ArduinoJson.h>  //libreria per la deserializzazione, utilizzata
   ↪  per parsare la notifica ricevuta
6
7  //_-_-_-DICHIARAZIONE DELLE VARIABILI_-_-_-//
8
9  //variabili per la connessione alla rete e al server REST
10 const char* ssid = "***************";
```

```
11  const char* password = "****************";
12  String REST_server = "192.168.1.30";
13
14  //variabili stringa e int che conterranno la configurazione, inizialmente
    ↪  sconosciuti
15  char host[30];
16  char up_path[30];
17  char up_header[30];
18  char sub_path[30];
19  char grafo[30];
20  int up_port, sub_port;
21
22
23  //variabili per la gestione del tempo
24  unsigned long int t=0,dt;
25
26  //variabili per la lettura delle temperature
27  int temp,hum; //conterranno la temperatura letta
28  int preTemp = 0; //contiene la temperatura della lettura precendente
29  int preHum = 0; //contiene l'umidità della lettura precendente
30
31  //variabili di controllo
32  int intervallo = 30; //ogni quanti secondi deve andare a leggere la
    ↪  temperatura
33  int scartoT = 2; //variazione della temperatura considerata significativa
34  int scartoH = 5; //varizione dell'unidità considerata significativa
35
36  //creazione di due oggetti DynamicDocument che consentiranno il parsing
    ↪  dei file JSON
37  DynamicJsonDocument doc(1024); //conterrà la stringa di notifica
38
39  //oggetto che rappresenta il sensore DHT11
40  DHT sensor(2, 11); //collegato al pin 2, modello 11 (DHT"11")
41
42  //dichiarazione funzione per il recupero configurazione
43  void getConfig(String ipRest, char *host[], char *u_path[], char
    ↪  *u_head[], char *s_path[], char *grafo[], int *u_port, int *s_port);
44
45  //_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-//
```

Viene anche dichiarato un oggetto DynamicJsonDocument, che consentirà il

parsing delle varie stringhe (dove il valore 1024 indica i byte) e un oggetto sensor, per utilizzare il sensore, che viene inizializzato indicando il pin di collegamento ed il modello. Si dichiara inoltre una funzione di comodo che eseguirà la routine di configurazione dal REST Server, con il seguente codice di implementazione:

```
1  //funzione di comodo che scarica e deserializza il file di configurazione
2  void getConfig(String ipRest, char host[], char u_path[], char u_head[],
   ↪  char s_path[], char grafo[], int *u_port, int *s_port){
3    //creo un oggetto HTTP per la connessione
4    WiFiClient client;
5    HTTPClient http;
6
7    //messaggio a video
8    Serial.println("Download del file di configurazione. Attendere");
9    Serial.println(".");
10   Serial.println(".");
11   Serial.println(".");
12
13   //connessione al REST Server
14   if(http.begin(client, "http://" + ipRest + "/")){ //se la connessione
     ↪  avviene con successo
15       int httpCode = http.GET();
16
17       if(httpCode > 0){
18         if(httpCode == HTTP_CODE_OK || httpCode ==
           ↪  HTTP_CODE_MOVED_PERMANENTLY){ //se la richiesta è corretta
19           String payload = http.getString(); //leggo il payload
20           int n = payload.length(); //ne leggo la lunghezza
21           char json[n]; //creo un array char per la deserializzazione
22           //converto la stringa in un char array, come richiesto dalla
             ↪  libreria ArduinoJson
23           payload.toCharArray(json, n);
24
25           //deserializzo il file ricevuto
26           deserializeJson(doc, json);
27
28          String ho = doc["host"];
29          String u_p = doc["update"]["path"];
30          String u_h = doc["update"]["header"];
31          String s_p = doc["subscribe"]["path"];
```

```
32          String grf = doc["graph"];
33
34
35          ho.toCharArray(host, 30);
36          u_p.toCharArray(u_path, 30);
37          u_h.toCharArray(u_head, 30);
38          s_p.toCharArray(s_path, 30);
39          grf.toCharArray(grafo, 30);
40          *u_port = doc["update"]["port"];
41          *s_port = doc["subscribe"]["port"];
42        }
43      }
44    }
45    http.end();//chiusura della connessione
46    Serial.println("Download effettuato. Connessione chiusa.");
47  }
```

Come si può notare, la funzione crea un oggetto HTTP, si connette al server REST, di cui è noto l'indirizzo IP, e scarica il file JSON, memorizzandolo localmente nella variabile stringa chiamata *payload*. A questo punto, dopo aver convertito la stringa in un array di char, esegue la deserializzazione e memorizza nelle variabili dichiarate nella prima parte del programma, i dati necessari per contattare il SEPA Server. In particolare, dell'intero file JSON mostrato in Figura 3.4, utilizza l'indirizzo IP dell'host, il graph name su cui deve operare, le porte ed i path per Update e Subscribe e l'header per le Update. Per concludere, chiude la connessione con il server e mostra in Serial Output che l'operazione è avvenuta con successo.

Passiamo quindi ad esaminare il Void Setup: dopo una prima parte in cui viene effettuata la connessione alla rete WiFi, viene richiamata la routine di configurazione sopra esaminata, mostrando a schermo i dati scaricati. Dopodichè, sfruttando la funzione *sepaSubscribe* del modulo Sepino.h, effettua la sottoscrizione al grafo; in particolare si sottoscrive a tutto ciò che ha come predicato l'URI "*Conferma*", a cui il nodo è interessato. Questo grazie alla stringa SPARQL di sottoscrizione che viene passata alla funzione, ottenuta seguendo le regole fornite dall SPARQL 1.1 SE Protocol. Viene poi dichiarata la funzione di callback, invocata ogni volta che si riceve una notifica.

All'interno di questa funzione, viene memorizzato localmente il testo della stringa ricevuta, viene effettuato un parsing e viene mostrato a video chi ha inviato la conferma e a chi essa è diretta. In particolare, il grafo è organizzato in modo che il soggetto è colui che effettua la notifica (nel caso in esame Arduino Yun) e l'oggetto è a chi viene confermata (nel nostro caso l'ESP8266), legati dal predicato "Conferma". Dopodichè viene effettuato un Update per eliminare quella tripletta dal grafo e la funzione di callback termina. É necessario effettuare l'operazione SPARQL di DELETE perchè, per il funzionamento del SEPA, se si riceve una tripletta uguale ad una già presente nel grafo, non viene effettuata nessuna notifica. In questo modo, invece, anche se le notifiche sono sempre uguali, verrà sempre ricevuta dal Nodo 1. Con questa funzione si conclude anche il Setup.

```
1   void setup(){
2     //inizializzazione della seriale con baud 115200
3     Serial.begin(115200);
4
5     //inizializzazione del sensore DHT11
6     sensor.begin();
7
8     //****************CONNESSIONE ALLA RETE WI-FI******************//
9         Per la connessione al WiFi può essere utilizzata una qualsiasi
10        procedura messa a disposizione dalle varie librerie
11    //**************************************************************//
12
13    if(WiFi.status() == WL_CONNECTED){//se la connessione WiFi è presente
14    //scarico il file di configurazione e lo deserializzo
15    getConfig(REST_server, &host[0], &up_path[0], &up_header[0],
       ↪   &sub_path[0], &grafo[0], &up_port, &sub_port);
16
17    //stampo a video i parametri di configurazione
18    Serial.println("**************CONFIGURAZIONE*************");
19    Serial.println("SEPA Server: " + String(host));
20    Serial.println("Graph: " + String(grafo));
21    Serial.println("Update Path: "+ String(up_path));
22    Serial.println("Update Header: " + String(up_header));
23    Serial.println("Subscribe Path: " + String(sub_path));
24    Serial.println("Update Port: " + String(up_port));
25    Serial.println("Subscribe Port: " + String(sub_port));
26    Serial.println("******************************************");
```

```
27    Serial.println("");
28   }
29
30   //******SOTTOSCRIZIONE AL SEPA, PREDICATO CONFERMA******//
31    String sub_msg = "{'subscribe' : {'sparql' : 'select * from "+
      ↪  String(grafo) +" where { ?s <http://Conferma> ?o }'}}";
32    sepaSubscribe(host, sub_port, sub_path, sub_msg);
33    Serial.println(".");Serial.println(".");
34   //*****************************************************//
35
36   //funzione di call-back
37   w_client.onMessage([&](WebsocketsMessage message) {
38      //leggo il testo della notifica inserendola in una stringa
39      String json = message.data();
40
41      //creo un array con la stringa per poterlo deserializzare
42      int len = json.length();
43      char dati[len];
44      json.toCharArray(dati,len);
45
46      //deserializzo la stringa letta
47      deserializeJson(doc, dati);
48
49      //Salvo in delle variabili i valori ricevuti di soggetto e oggeto
50      //il predicato è già noto, in quanto siamo sottoscritti solo agli
        ↪  elementi con predicato "Conferma"
51      String soggetto = doc["notification"]["addedResults"]["results"]
52                        ["bindings"][0]["s"]["value"];
53      String oggetto = doc["notification"]["addedResults"]["results"]
54                        ["bindings"][0]["o"]["value"];
55
56     if(soggetto != "null" && oggetto != "null"){
57        //mostro a schermo la notifica ricevuta
58        Serial.println("Ricevuta conferma di lettura da "+ soggetto +" con
          ↪  oggetto "+ oggetto);
59        //effettuo un UPDATE per eliminare la notifica appena ricevuta
60        String del_msg = "DELETE DATA { GRAPH "+ String(grafo) +" { <"+
          ↪  soggetto +"> <http://Conferma> <"+ oggetto +"> } }";
61        sepaUpdate(host, up_port, up_path, up_header, del_msg);
62     }
63   });
64  }
```

Per quanto riguarda il Void Loop, esso contiene il metodo che rimane in ascolto sulla WebSocket e in caso di notifca richiama la funzione di callback presente nel Setup. Inoltre, ogni trenta secondi, entra in un *if* in cui va a misurare il valore di temperatura e umidità e, se ci sono variazioni significative dalla lettura precedente, va ad aggiornare il grafo. Termina così lo sketch del nodo 1.

```
void loop(){
  //in questo if, il client rimane in ascolto delle possibili notifiche,
  ↪ se la webSocket è aperta
  //se riceve una notifica, invoca la funzione di callback presente nel
  ↪ setup
   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
      String t_msg = "INSERT DATA { GRAPH " + String(grafo) +" {
      ↪ <http://ESP8666> <http://Temperatura> "+ String(preTemp) +" }
      ↪ }";
      sepaUpdate(host, up_port, up_path, up_header, t_msg);
      Serial.println("Inviato un nuovo valore di Temperatura: "+
      ↪ String(preTemp));
      Serial.println(".");Serial.println(".");Serial.println(".");
```

```
26        }
27
28        if(hum > (preHum+scartoH) || hum < (preHum - scartoH)){
29          preHum = hum;
30          //invio l'update Umidità al grafo
31          String h_msg = "INSERT DATA { GRAPH " + String(grafo) +" {
          ↪   <http://ESP8666> <http://Umidita> "+ String(preHum) +" } }";
32          sepaUpdate(host, up_port, up_path, up_header, h_msg);
33          Serial.println("Inviato un nuovo valore di Umidità: "+
          ↪   String(preHum));
34          Serial.println(".");Serial.println(".");Serial.println(".");
35        }
36
37        //reset del valore di t
38        t = millis();
39      }
40    }
```

Per effettuare la temporizzazione si è deciso di non utilizzare la funzione delay per un motivo particolare; essa, infatti, va a bloccare il Loop per un certo lasso di tempo. Utilizzando un delay, quindi, se viene inviata una notifica mentre il sistema è in attesa, essa non verrà rilevata. Con il metodo utilizzato, invece, questo problema non si presenta.

Il serial Output del Nodo 1 è mostrato in Figura 3.6

Il modulo Sepino.h

Come precedentemente accennato, lo sketch sfrutta il modulo Sepino.h per utilizzare le funzioni di Update e Subscribe. A sua volta, il modulo utilizza le librerie per l'implementazione del protocollo HTTP e WebSocket su ESP8266.

```
1  #include <Arduino.h>
2  #include <ArduinoWebsockets.h> //libreria per la websocket
3  #include <ESP8266WiFi.h>  //libreria per la connessione al wifi
4  #include <ESP8266HTTPClient.h>  //libreria per le richieste HTTP
5
6  using namespace websockets;
7  WebsocketsClient w_client;
```

```
COM3
Connesso al WiFi!! Tentativo di connessione al server...
Download del file di configurazione. Attendere
.
.
.
Download effettuato. Connessione chiusa.
***************CONFIGURAZIONE*************
SEPA Server: 192.168.1.76
Graph: <http://demoSepino>
Update Path: /update
Update Header: application/sparql-update
Subscribe Path: /subscribe
Update Port: 8000
Subscribe Port: 9000
*******************************************

WebSocket aperta!!
.
.
Update effettuato!!
Inviato un nuovo valore di Temperatura: 26
.
.
.
Update effettuato!!
Inviato un nuovo valore di Umidità: 40
.
.
.
Ricevuta conferma di lettura da http://ArduinoYun con oggetto http://ESP8266
Update effettuato!!
```

Figura 3.6: Output video del Nodo 1

Saranno analizzate le due funzioni utilizzate nello sketch di implementazione del Nodo 1. La funzione di Update necessita di cinque parametri di ingresso: indirizzo IP, porta, path ed header per gli Update al SEPA ed il testo del body, che sarà la stringa SPARQL per effettuare l'operazione richiesta. La funzione sfrutta un oggetto della libreria HTTP ed effettua un Update con metodo POST, in cui il body è la stringa SPARQL contenuta nel parametro di ingresso.

```
1  void sepaUpdate(String host, int port, String path, String header, String
   ↪  msg){
2      //Creazione di un client HTTP WiFi
3      WiFiClient client;
4      HTTPClient http;
5
6      //connessione al server
7
8      http.begin(client, "http://"+ host +":"+ String(port) +"" + path);
       ↪  //connessione attraverso l'URL del server
9      //invio dell'header
10     http.addHeader("Content-Type", header);
11     //invio del body i POST
```

```
    http.POST(msg);
    //messaggio di conferma su seriale
    Serial.println("Update effettuato!!");
    //chiusura della connessione
    http.end();
}
```

La funzione di Subscribe, invece, ha solo quattro parametri di ingresso, che sono gli stessi dell'Update, ma senza l'header, non necessario per le operazioni di sottoscrizione. Il testo del body, in questo caso, conterrà la stringa per la richiesta di Subscribe, in sintassi SPARQL. Utilizza inoltre un oggetto della libreria ArduinoWebsocket, dichiarato nella prima parte del modulo, per gestire il protocollo.

```
void sepaSubscribe(String host, int port, String path, String msg){
↪  //tentativo di connessione al server SEPA)
  bool connected = w_client.connect(host, port, path); //tentativo di
  ↪  connessione al server SEPA
  if(connected){ //se la connessione è avvenuta con successo
    Serial.println("WebSocket aperta!!");
    w_client.send(msg); //sottoscrizione a ciò che siamo interessati
  }
  else{//altrimenti messaggio di errore
    Serial.println("Connessione al server fallita. Resettare per
    ↪  riprovare");
    while(1) ; //rimane fermo tutto fino ad un riavvio
 }
}
```

La realizzazione di questo modulo era uno degli obiettivi dell'attività di tesi e verrà messo a disposizione di chiunque sarà interessato ad interfacciare un microcontrollore ESP8266 con l'architettura SEPA.

Nodo 2: Arduino Yun

Il nodo 2 è costituito dal microcontrollore Arduino Yun. Questo modello sfrutta, oltre al chip ATMega32u4, un microrpocessore AR9331, su cui è possibile operare attraverso Linino, distribuzione di Linux realizzata per l'utilizzo su schede Arduino. In questo modo è possibile utilizzare diversi

linguaggi di programmazzione per la scrittura degli sketch. Attraverso l'apposita libreria *Bridge*, poi, il programma sul microprocessore comunicherà con il microcontrollore, consentendo la gestione dell'Output Seriale e dei vari pin di I/O. Per la realizzazione dei programmi è stato utilizzato Python 2.7. Il ruolo del nodo 2 è quello di ricevere le notifiche di temperatura e umidità ed attivare o disattivare dei led se queste superano determinate soglie. Ogni volta che riceverà una notifica, effettuerà un Update per eliminare la tripletta appena ricevuta, per lo stesso motivo precedentemente esposto. Infatti, se questa operazione non fosse effettuata, il SEPA non invierebbe le notifiche che contengono valori di temperatura o umidità uguali a quelli precedentemente ricevuti. Infine effettua un Update per notificare al Nodo 1 l'avvenuta ricezione del dato Raduzzi [2021].
Questo nodo è stato realizzato da un collaboratore dell'attività di ricerca e non verrà quindi approfindito in questo elaborato.

L'uscita video del Nodo 2 è mostrata in figura Figura 3.7.

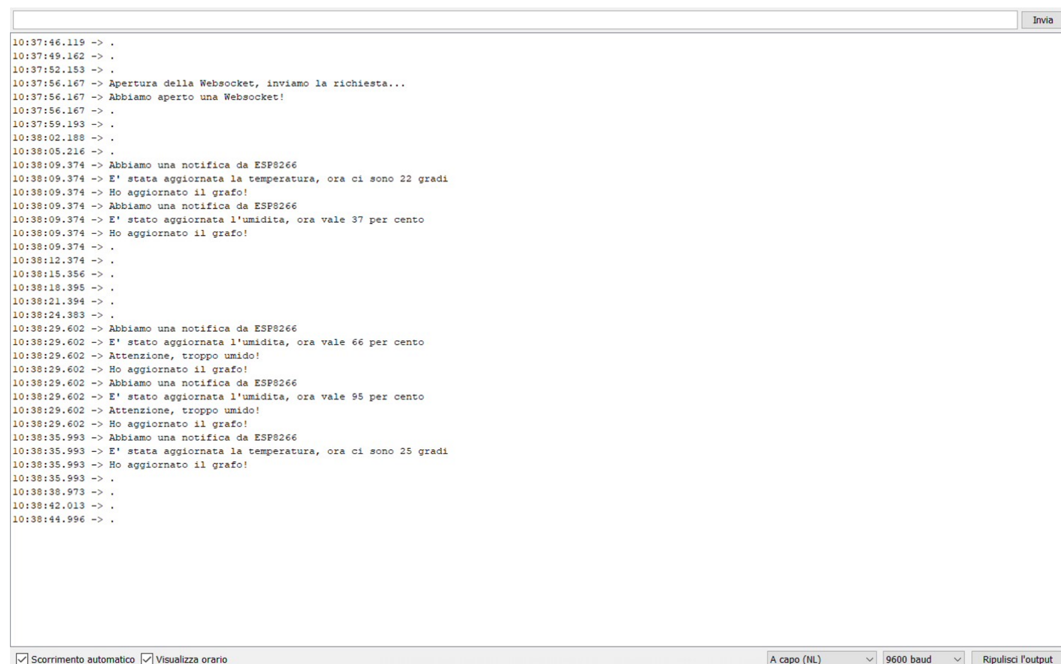

Figura 3.7: Output video del Nodo 2

Capitolo 4

Osservazioni e conclusioni

L'obiettivo del lavoro di tesi era quello di interfacciare il microcontollore Arduino Uno ed il chip ESP8266 con il server SEPA, sfruttando le operazioni di Update, Query e Subscribe necessarie per la manipolazione degli RDF Graph , andando inoltre a creare una libreria apposita dedicata a queste funzioni. Inoltre si voleva raggiungere l'autonomia del sistema con l'utilizzo di servizi REST, nel caso in esame introdotti attraverso Arduino Uno utilizzato in modalità server, per la configurazione automatica dei microcontrollori appartenenti al sistema. Tutti gli obbiettivi sono stati raggiunti grazie alle funzionalità messe a disposione dal software Arduino IDE.
In particolare, attraverso l'attività di ricerca svolta, si può affermare che il microcontrollore Arduino Uno, allo stato attuale, è in grado di interfacciarsi con la piattaforma semantica SEPA soltanto attraverso il protocollo HTTP, in quanto trova difficoltà nell'implementazione del protocollo WebSocket. Può quindi eseguire soltanto operazioni di Update e Query. Altrimenti, come nella demo illustrata, può essere utilizzato come server che implementi un servizio REST di configurazione del sistema, orientando il tutto verso il concetto di stand-alone.
Se invece si necessita del set completo di operazioni è possibile sfruttare i moduli hardware che utilizzano il chip ESP8266. Infatti, nonostante siano prodotti dalla Espressiff System e non da Arduino, essi consentono la programmazione attraverso Arduino IDE, mantenendo quindi le caratteristiche user-friendly illustrate. Forniscono, inoltre, un alto potenziale nell'utilizzo delle risorse di rete ad un prezzo di mercato alla portata di tutti e di facile reperibilità.

Inoltre è stata realizzata con successo una prima versione della libreria, ovvero il modulo Sepino.h. Esso permetterà di rendere disponibili le funzioni realizzate a chiunque voglia interfacciare i propri microcontrollori al SEPA.
Tra gli obbiettivi futuri dell'attività di ricerca, uno è quello di implementare con successo sui microcontrollori Arduino la teoria dei *Forced Bindings*. In questo modo, nel file di configurazione, possono essere inserite le stringhe SPARQL di Update, Query e Subscribe, consentendo all'utente finale l'utilizzo del sistema SEPA senza dover conoscere necessariamente il linguaggio SPARQL. Un'ulteriore obiettivo è quello di sfruttare il protocollo HTTPS, in modo da poter criptare le comunicazioni tra microcontrollori e server SEPA. Questo aprirebbe la strada a diverse possibilità, tra cui l'utilizzo di un server centrale su rete pubblica. Inoltre il file di configurazione potrebbe essere scaricato direttamente da una pagina Web, senza inserire un nodo apposito all'interno del sistema. Si sta inoltre pensando di inserire in ogni microcontrollore che si voglia connettere alla rete di monitoraggio, una routine di *"Ping"* che effettui un Update ad un determinato ramo del grafo, indicando lo stato della sua connessione al sistema. In questo modo possono essere sviluppate applicazioni Web che, sottoscrivendosi al grafo, possano monitorare lo stato della rete, mostrando i dispositivi connessi ed il loro ruolo nel sistema.
Il progetto SEPINO è ancora in fase di sviluppo, sostenuto da diversi collaboratori che studiano il comportamento di vari modelli di Arduino nell'interfacciamento con l'architettura SEPA. L'obbiettivo finale è quello di realizzare una libreria unica, che semplifichi notevolmente l'utilizzo dei microcontrollori Arduino nell'interfacciamento con il SEPA e di poter quindi creare una rete IoT basata su queste tecnologie.

Si ringrazia, oltre al Professore Luca Roffia e al correlatore Dott. Simone Sindaco per aver seguito lo sviluppo del progetto, il collaboratore Lucafrancesco Raduzzi, con cui è stata realizzata la demo presentata nell'elaborato.

Bibliografia

Luca Roffia, Cristiano Aguzzi, Francesco Antoniazzi, and Fabio Viola. Sparql event processing architecture (sepa). http://mml.arces.unibo.it/TR/sepa.html, 2018a.

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.

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

A. Melnikov. The websocket protocol. https://tools.ietf.org/html/rfc6455, 2011.

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

Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2000.

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

Arduino. Arduino uno rev 3. https://store.arduino.cc/arduino-uno-rev3, 2021.

EspressifSystem. Esp8266 documentation. https://www.espressif.com/en/support/documents/technical-documents, 2021.

Lucafrancesco Raduzzi. *Interfacciamento di un Sistema Embedded Arduino Yun alla piattaforma semantica SEPA*. 2021.

Elenco delle figure

1.1 Logo RDF Data Model 8
1.2 Esempio generale di Tripla 8
1.3 Esempio di dataset RDF 10
1.4 Sintassi N-Triplets per la rappresentazione del grafo in Figura 1.3 11
1.5 Metodi per Query 12
1.6 Metodi per Update 13
1.7 Esempio di Notifica 15
1.8 Architettura del SEPA 16
1.9 DashBoard di gestione del SEPA 17
1.10 Schema di Architettura REST in JSON 19

2.1 Logo ufficiale di Arduino 21
2.2 Arduino Uno Rev 3 23
2.3 Schematico di scheda Arduino Uno Rev 3 24
2.4 Ethernet Shield v 1.0 25
2.5 Hardware per interfacciamento Arduino Uno 26
2.6 Chip ESP8266 della Espressif Systems 28
2.7 Modulo ESP-01 29
2.8 Schematico ESP-01 30
2.9 Collegamento tra Arduino Uno e ESP-01 31
2.10 Modulo NodeMCU Amica v. 2.0 32
2.11 Schermata vuota di Arduino IDE 33

3.1 Schema a blocchi del sistema 35
3.2 Grafo demoSepino 37
3.3 Hardware del REST Service su Arduino Uno 38
3.4 File di configurazione in sintassi JSON 42

3.5 Output video del server REST 43
3.6 Output video del Nodo 1 51
3.7 Output video del Nodo 2 53