Matteo Quarta,

Riccardo Sepe,

Andrea Toscano

Relazione di Laboratorio – Tecnologie per IoT

Descrizione Laboratorio hardware parti 1 e 3:

Parte 1:

Esercizio 1)

- Inclusione della libreria `TimerOne.h` per gestire l'attach degli interrupt con timer ai due led
- Il led Verde cambia stato con l'interrupt in base a `G_HALF_PERIOD * 1e06`
- Il led Rosso cambia stato nel loop con un delay di `R_HALF_PERIOD * 1e03`

Esercizio 2)

- Estensione del primo esercizio con l'aggiunta di un input per la stampa dello stato dei led

Esercizio 3)

- Verifica di un movimento mediante sensore PIR, collegato al PIN 7, mediante Interrupt
- Attach del pin del PIR ad una funzione di callback di cambio di stato in base alla verifica del cambio di valore dal pin.

Esercizio 4)

- Comando del Fan fornito mediante PWM
- Gestione della velocità del Fan mediante input e gestione di velocità minima/massima

Esercizio 5)

- Calcolo della temperatura mediante lettura di un valore analogico in input con funzione di mapping predefinita, propria del sensore.

Esercizio 6)

- Estensione dell'Esercizio 5 con stampa sul LCD della temperatura rilevata

Matteo Quarta,

Riccardo Sepe,

Andrea Toscano

Parte 3:

Esercizio 1)

- La scheda deve predisporre un server HTTP in grado di rispondere alle richieste GET provenienti dalla LAN:
- I DynamicJsonDocument hanno capacità pari a: const int capacity = JSON_OBJECT_SIZE(2) + JSON_ARRAY_SIZE(1) + JSON_OBJECT_SIZE(4) + 40;
- Una volta inizializzato il BridgeServer si gestiscono le richieste nel loop accettandole mediante BridgeClient
- Ogni volta che viene rilevata una richiesta viene processata mediante la funzione process(client) che elabora la stringa ottenuta dalla richiesta.
- Si analizza la stringa con un parse per riconoscerne i campi e si stampa la risposta a seconda dei casi, gestendo eventuali errori 400 e 404
- I valori sono formattati in formato JSON

Esercizio 2)

- A differenza del primo esercizio, questa volta la scheda deve fare da Client HTTP
- Abbiamo scelto di utilizzare la stessa capacità dei DynamicJsonDocument dell'esercizio precedente poiché il formato è lo stesso
- All'interno del loop, una volta ogni secondo, viene effettuata una richiesta mediante il processo curl, gestito dalla libreria Process.h, al termine della quale viene stampato il valore di ritorno.
- I valori inviati sono anche qui formattati in JSON

Modifica al Lab Software 1 parte 2:

Come richiesto dalla traccia si è modificato lo script in Python in modo tale da esporre un server con funzionalità REST (GET e POST)

- POST: Il server deve accettare un documento JSON come input, che deve essere salvato in una lista
- GET: Il server deve restituire la lista delle stringhe JSON serializzate ottenute mediante POST

Esercizio 3)

Il terzo esercizio prevede la sottoscrizione e la pubblicazione con protocollo MQTT su un server di test, nel nostro caso "test.mosquitto.org", e come topic "/tiot/18/"

Matteo Quarta,

Riccardo Sepe,

Andrea Toscano

La subscribe viene effettuata nel setup con il subtopic "led", collegandogli la funzione di callback onMessage() che gestisce il payload ottenuto per accendere e spegnere il led

La publish invece viene effettuata una volta ogni 10s (10* 1000ms nel delay alla riga 86) sul subtopic "temperature", nel quale viene inviato un payload chiamato data, ovvero una stringa JSON serializzata contenente la temperatura

Smart Home

Il lavoro da svolgere consisteva nella realizzazione di un sistema IoT in grado di emulare le funzionalità principali di una Smart Home, dall'impianto hardware al sistema software, passando per i servizi middleware.

Hardware:

Il progetto della smart home si basa sull'esposizione di vari servizi quali: riscaldamento, ventilazione e segnalazione di presenze, tutti reperibili mediante un output sul display LCD. Il tutto viene effettuato mediante l'ausilio del materiale fornito. La configurazione disponibile nel repository è stata testata e verificata funzionante dopo aver settato correttamente i vari potenziometri sui vari sensori/dispositivi.

DESCRIZIONE DEI PIN:

2,3: Utilizzati per il display LCD, mediante I2C cosi da dover usare solo 2 pin.

7: Abbiamo scelto il Pin 7 per connettere il sensore PIR, in modo tale da poter usare la funzionalità di interrupt fornita dalla scheda sul pin sopracitato.

9: Per il sensore di rumore

10: Per il LED

11: Per il Fan di ventilazione operante in PWM

A1: Per il rilevamento della temperatura

Matteo Quarta,

Riccardo Sepe,

Andrea Toscano

CONVENZIONE ADOTTATA PER I SetPoint:

Dal momento che le configurazioni disponibili potevano essere solo due, ognuna composta da 4 SetPoint per i rispettivi valori di riscaldamento e di condizionamento (minimi e massimi), abbiamo scelto di utilizzare un totale di 12 variabili short.

La convenzione utilizzata è la seguente:

- C: Condizionamento
- R: Riscaldamento
- P: Presenza
- A: Assenza

Così denominando i setPoint come "cpMinSetPoint","raMaxSetPoint" ecc...

Di queste 8 variabili di base abbiamo scelto di aggiungerne altre 4 per salvare i setPoint in utilizzo corrente.

GESTIONE DELLE PRESENZE:

Per verificare che effettivamente siano presenti delle persone, basta verificare che il sensore di movimento, oppure il sensore di rumore, abbiano rilevato movimento.

Dichiarati i valori booleani PIRPresence e soundPres, inizializzati a zero, la reale presenza delle persone è data dalla variabile PPres = (soundPres || PIRPresence), verificato ogni ciclo del loop mediante la funzione switchP().

La presenza delle persone viene aggiornata mediante i due valori booleani rispettivi dei due sensori.

Nel momento in cui non viene rilevato più alcun suono o movimento per una data finestra temporale si ritorna in una condizione di assenza.

GESTIONE DELLA (RI)CONNESSIONE:

La (ri)connessione al catalogo viene gestita nel file Software/Part2/4/3_4.ino, che non è altro che il file precedentemente sviluppato, esteso con le funzionalità necessarie ad interagire con il catalogo descritto nella sezione *Middleware.* Per quanto riguarda le interazioni con il catalogo, è richiesta la costruzione di strutture dati JSON simili a quelle realizzate esaustivamente in altre parti del codice sviluppato e del tutto analoghe per quanto riguarda l'implementazione, la serializzazione delle quali produrrebbe come risultato delle stringhe contenenti le informazioni da inviare al catalogo; per brevità, dopo aver realizzato in dettaglio la stringa per la prima operazione (registrazione), abbiamo,

Matteo Quarta,

Riccardo Sepe,

Andrea Toscano

nelle successive operazioni di questo tipo, assunto di aver ottenuto tali stringhe come descritto, privilegiando l'implementazione dettagliata delle operazioni significative a valle del procedimento descritto.

TEMPLATE DEL PROGETTO:

- Inizializzazione dei Pin e dell'interfaccia Seriale
- Attesa che la seriale venga aperta
- Inizializzazione del LCD
- Attach dell'interrupt al pin Digitale del PIR, assegnandogli la funzione di callback di verifica della presenza
- Stampa del SetPoint iniziale e di un menu base per poter cambiare i SetPoint
- Inizializzazione dei Timer

Nel Loop:

Se non sono stati cambiati i valori di SetPoint:

- Prendo un altro timestamp
- Controllo la presenza di persone
- Gestisco il condizionamento mediante varie funzioni di mapping, in base alla temperatura rilevata dalla funzione `tempCalc()`
- Verifico la presenza di eventi sonori
- Stampo sul lettore LCD
- Verifico se è disponibile un input sull'interfaccia seriale

GESTIONE DEGLI EVENTI SONORI:

Per evitare che la presenza di una persona possa passare inosservata a causa di un forte raggruppamento di eventi sonori, seguito da una assenza degli stessi, abbiamo scelto di utilizzare una finestra a scorrimento per la gestione degli eventi.

La finestra è realizzata mediante buffer circolare nel quale vengono memorizzati i timestamp degli eventi sonori. Se hanno luogo più di 49 eventi sonori, l'algoritmo controlla la distanza tra l'ultimo evento sonoro e il primo avvenuto. Nel caso in cui il tempo fosse minore della finestra di rilevamento richiesta, allora la presenza di una persona verrebbe rilevata. In caso contrario si continua a scrivere nel buffer avanzando di posizione e riscalando gli indici di coda e testa del buffer.

Matteo Quarta,

Riccardo Sepe,

Andrea Toscano

REGOLAZIONE DEL CONDIZIONAMENTO E DEL RISCALDAMENTO:

Una volta calcolata la temperatura mediante la funzione tempCalc(), che fa utilizzo dei valori del sensore per mappare la temperatura in base al valore di tensione rilevato, è possibile utilizzare il suddetto valore per stampare la temperatura corrente e di conseguenza mappare questo valore sull'intervallo di SetPoint corrente.

Il riscaldamento deve essere inversamente proporzionale al condizionamento, per cui abbiamo deciso di mappare la temperatura tra [cpMin, cpMax] per il Fan, e tra [rpMax, rpMin] per il LED, entrambi in un range tra 0 e 255. In tal modo otteniamo un valore utilizzabile da scrivere in maniera analogica sui rispettivi pin.

TEMPORIZZAZIONE DEL DISPLAY LCD:

Per evitare di dover aggiungere delay nel loop e quindi di aggiungere ritardo ad ogni passaggio, si è deciso di utilizzare dei timer ad intervalli costanti per temporizzare l'output sul display

In questo modo, prendendo un timestamp ad ogni loop, possiamo decidere di stampare sull'LCD la prima pagina ad intervalli di tempo compresi tra 0 e 1000, la seconda tra 1001 e 2000, effettuando sempre una divisione in modulo 2000.

Middleware:

Per quanto riguarda gli aspetti middleware, il Catalogo è composto da cinque file scritti in Python, di cui quattro classi.

Il primo file, main.py, è il server basato sul modulo CherryPy e utilizza le quattro classi.

Cleaner.py è la classe che si occupa di cancellare periodicamente i dispositivi e servizi che non hanno rinnovato la loro iscrizione, come da richiesta.

DeviceManager.py, UserManager.py e ServiceManager.py si occupano di soddisfare le richieste, memorizzare e concordare l'accesso ai dati rispettivamente per dispositivi, utenti e servizi, montati sugli endpoints '/devices', '/users' e '/services'.

Il server può essere visto come un catalogo REST improprio poiché non implementa tutti i metodi CRUD, per due motivi:

- Non c'è un metodo GET (se non all'index del catalog, per ottenere gli endpoints) in quanto si è deciso di non utilizzare queries per l'ottenimento dei dati, scelta che ha creato un piccolo

Matteo Quarta,

Riccardo Sepe,

Andrea Toscano

problema dovuto a CherryPy, che ignora il payload delle richieste GET, anche quando presente. Abbiamo quindi implementato le funzionalità richieste tramite richieste POST.

- Non c'è un metodo DELETE in quanto non è stato richiesto un metodo specifico per la cancellazione di utenti, servizi o dispositivi.

Le classi, molto simili nella struttura, hanno tutte un meccanismo di accesso concordato ai dati in scrittura, che si tratti di un file locale o in memoria. Per mancanza di tempo non abbiamo ottimizzato l'arbitraggio, che sfrutta un paradigma a semaforo, che causa busy-waiting in attesa che la risorsa si liberi. Questo è tuttavia abbastanza ininfluente dato che le risorse vengono comunque occupate per il tempo minimo indispensabile alla riscrittura dei dati.

Un ampliamento del catalogo è disponibile in (5) dove è disponibile un ulteriore endpoint '/mqtt/devices' impiegato dalla classe MqttDeviceManager.py, che rende il catalogo anche un subscriber a questo topic. La classe è soggetta comunque a quanto descritto sopra. Per far funzionare correttamente il catalog con MQTT, abbiamo aggiunto una funzione ad-hoc in DeviceManager.py, che elabora le richieste dal broker, tuttavia non troppo diversamente da quanto farebbe con delle normali richieste PUT.

Software:

Per quanto riguarda gli aspetti software, oltre al codice Arduino (C++), già descritto insieme agli aspetti hardware, abbiamo scritto del codice Python, necessario sia per realizzare dei servizi MQTT da esporre alla smart home, sia per realizzare i servizi aggiuntivi richiesti dal Laboratorio SW - Part 4.

I servizi MQTT richiesti sono relativi alla ricezione di informazioni di sensing provenienti dalla scheda e all'invio di comandi di attuazione ad essa; essi sono implementati nella cartella IoT_Labs/LabSW3.

- Il file Exercise_2.py contiene una classe MQTTSubscriber (che implementa un subscriber MQTT) che, dopo essersi registrato al Service Catalog, è in grado di ricevere i dati raccolti dal sensore di temperatura installato sulla Yùn, nel formato JSON, tramite la ridefinizione della callback on_message().
- Il file Exercise_3.py contiene una classe MQTTPublisher (che implementa un publisher MQTT) che, dopo essersi registrato al Service Catalog, espone il metodo switch_LED(), il quale riceve un booleano in ingresso (True=ON, False=OFF) e costruisce un messaggio formattato in JSON da pubblicare al topic corrispondente al LED, per accenderlo o spegnerlo. In caso si richieda di accendere o spegnere due volte consecutive il LED, le richieste vengono ignorate dalla seconda in poi
- Nella cartella '…/3' sono presenti due file Python: in MQTT.py è presente una implementazione di un client MQTT generico, in grado da comportarsi sia come publisher,

Matteo Quarta,

Riccardo Sepe,

Andrea Toscano

sia come subscriber, mentre in Exercise_4.py è presente una classe di front-end che espone i metodi necessari per inviare comandi di attuazione e per ricevere i dati del sensing.

I servizi aggiuntivi richiesti dal Laboratorio SW - Part 4 sono raccolti nella cartella LabSW4: si tratta di un bot Telegram per consultare in sola lettura il Catalog e di un servizio basato su REST per mostrare sul display dell'Arduino il meteo relativo alla località in cui si trova la smart home stessa.

Bot Telegram

Abbiamo scritto il bot in Python utilizzando le API non ufficiali (ma ben testate ed affidabili) Python-Telegram-Api. Esso può essere visto come uno strumento rapido per eseguire richieste POST (ma non PUT) al catalogo, come fosse una linea di comando, lasciando a lui il compito di formattare la richiesta e interrogare il catalogo come segue.

I comandi prendono il nome dai rispettivi endpoints, eccezione fatta per /mqtt/devices, le cui informazioni vengono unite a quelle di /devices. È possibile scrivere solo il nome dell'endpoint, per ottenere informazioni sull'integrità dei dati attualmente in possesso dal catalogo, oppure indicare gli ID di interesse, divisi da uno spazio.

Si noti che non si possono eseguire richieste eterogenee, vale a dire che se, ad esempio, si vogliono conoscere le informazioni su un dispositivo ed un utente, sarà necessario eseguire due comandi distinti.

Il bot può operare indistintamente sia sulla versione del catalog dell'esercizio (5) che su quella dell'esercizio (1).

Servizio Meteo

Nella cartella Weather sono presenti 5 file, necessari per implementare il servizio di meteo.

Il file 4_1.ino non è altro che una copia della versione della smart home implementata nel laboratorio hardware parte 2, con aggiunta la gestione di un pushbutton che, se premuto, consente di mostrare sul display LCD le informazioni relative al meteo, in due schermate consecutive.

Il file RESTservice.py contiene la classe WeatherService, che è l'interfaccia HTTP RESTful (impropria) esposta verso l'esterno. L'unico metodo consentito è GET, che, quando invocato da un client, risponde con il meteo relativo all'orario corrente. In particolare, questo metodo crea una sessione, nella quale salva l'indirizzo IP del client che effettua la richiesta e i dati relativi al meteo delle 24h successive, ricevuti dal modulo get_weather.py.

Nel metodo GET ci sono tre possibili scenari (più un quarto corrispondente ad una richiesta mal posta).

Matteo Quarta,

Riccardo Sepe,

Andrea Toscano

- La prima volta che viene effettuata una richiesta su <baseurl> o se viene effettuata la richiesta su <baseurl>/new, viene creata una nuova sessione (ed eventualmente terminata quella precedente), il server ottiene il meteo tramite la get_weather(), salva nella sessione l'indirizzo IP del richiedente e i dati relativi al meteo e infine risponde con il meteo relativo all'orario in cui è stata effettuata la richiesta. Come orario viene utilizzato quello relativo al server, nell'ipotesi semplificativa che il server e la smart home abbiano lo stesso fuso orario.
- Se viene effettuata la richiesta su <baseurl> e l'indirizzo IP del richiedente corrisponde a quello salvato nella sessione, il server risponde con il meteo relativo all'orario corrente, che aveva precedentemente salvato.
- Se viene effettuata la richiesta su <baseurl>, ma l'indirizzo IP del richiedente è cambiato, viene creata una nuova sessione.

Si noti che questa applicazione è pensata per servire un solo client per volta, e che essa cerca di massimizzare la computazione lato server, per poter alleggerire il carico computazionale del client, e quindi il suo consumo di energia. Poiché è pensata per servire un solo client per volta, abbiamo ritenuto non necessario iscrivere questo servizio al Service Catalog, nell'ipotesi che il suo indirizzo sia noto a priori e che la macchina che esegue questo server non cambi indirizzo IP nel tempo. Un modo per aprire l'utilizzo di questo servizio a più client contemporaneamente potrebbe essere quello di salvare i dati di ogni client in una diversa entry del dizionario cherrypy.session, e gestire a lato client lo scenario in cui quest'ultimo cambia indirizzo IP rispetto ad una richiesta precedente.

La classe WeatherService prevede inoltre l'utilizzo del file weatherconversion.json, il cui contenuto viene inviato alla funzione get_weather(), insieme all'indirizzo IP del client, per consentire la traduzione dei dati grezzi ricevuti dalla API del meteo in dati human-readable. I dati ricevuti sono contenuti in una struttura JSON, che contiene i campi "product", "init" e "dataseries": il primo è riferito al tipo di servizio richiesto al server, in quanto esso può rispondere anche con altri set di informazioni meteo, utili ad altri scopi; il secondo è un'informazione riguardo anno, mese, giorno e ora di riferimento per i dati meteo, che infatti sono memorizzati in base alla differenza di ore rispetto al campo "init"; il campo "dataseries" è una lista di dizionari JSON che contengono i dati meteo relativi alla settimana successiva al campo "init", riferiti a intervalli equispaziati di 3 ore. La funzione get_weather() prende solo i primi 8 campi, relativi alle 24h successive al campo "init", e li consegna alla funzione convert_data(), che si occupa di creare dati human-readable.

Il file weatherconversion.json non è altro che un dizionario organizzato secondo le regole JSON.

Il file get_weather.py contiene le funzioni necessarie per ricevere i dati grezzi dalla API del meteo utilizzata (www.7timer.info) e per convertirli in dati human-readable, grazie al dizionario ricevuto come parametro dalla WeatherService.

Inoltre, la funzione get_weather() contenuta nel file omonimo chiama l'unico metodo presente nel file position.py, cioè get_position(), che, dato un indirizzo IP in ingresso, restituisce le coordinate geografiche associate a quell'indirizzo IP, sfruttando la API 'ip-api.com'.

Matteo Quarta,

Riccardo Sepe,

Andrea Toscano

Per la rilevazione della posizione abbiamo preferito sfruttare l'indirizzo IP invece di un modulo hardware GPS, perché, presumibilmente, la posizione di una smart home non dovrebbe cambiare molto frequentemente, per cui rilevarla una volta al giorno è sufficiente, quindi per motivi di cost effectiveness abbiamo preferito adottare una soluzione software.

Analisi comparativa delle due versioni di smart home

Volendo confrontare le due versioni di smart home, quella controllata attraverso tecnologia embedded e quella controllata in remoto sfruttando i protocolli che si appoggiano su IP, bisogna effettuare una distinzione.

In un primo scenario, in cui è presente un solo dispositivo IoT all'interno di un ambiente, qualitativamente le due versioni del sistema sviluppato funzionano in maniera simile.

Infatti, in entrambi i casi esse devono gestire dati prodotti localmente sul dispositivo, anche se la versione embedded necessita di un'interfaccia locale per gestire input e output, che noi abbiamo potuto solamente emulare tramite ambiente di sviluppo, mentre per la versione 'remote' è sufficiente utilizzare un qualunque terminale connesso a Internet.

Da un punto di vista più tecnico, la versione 'remote' consuma generalmente meno energia di quella embedded. Infatti, mentre quest'ultima deve svolgere tutte le proprie operazioni localmente, sfruttando la (bassa) potenza di calcolo della MCU, la versione 'remote' deve solamente effettuare alcune comunicazioni sfruttando i protocolli di rete, mentre tutte le operazioni computazionali (che in questo caso non sono particolarmente gravose, ma in dispositivi del genere presenti sul mercato possono esserlo, si pensi già soltanto al riconoscimento vocale) sono demandate ad un calcolatore remoto, che comunica i propri risultati.

Ovviamente, dal punto di vista della praticità, la versione 'remote' è più comoda per l'utente, in quanto non richiede di interagire direttamente con il dispositivo, bensì consente di controllarlo da qualunque ambiente della casa, anche se nella versione embedded abbiamo implementato delle features che consentono lo svolgimento di determinate operazioni e routine in automatico,

In un altro scenario, invece, in cui vi siano due o più dispositivi IoT nello stesso ambiente, la versione embedded è fortemente penalizzata, in quanto ogni dispositivo va controllato individualmente tramite interfaccia I/O locale, prestando attenzione a non creare situazioni conflittuali tra i vari device IoT, a meno che non si vogliano disporre interconnessioni cablate tra i dispositivi, più eventuali protocolli, per consentirne la comunicazione via cavo: in tal caso, si potrebbe automatizzare la prevenzione delle situazioni di conflitto. La versione remote, invece, offre un'interfaccia presumibilmente centralizzata, o comunque consente di centralizzare in maniera relativamente più semplice il controllo della rete di dispositivi, quindi evitando a priori, in maniera

Matteo Quarta,

Riccardo Sepe,

Andrea Toscano

automatica, eventuali situazioni di conflitto tra i dispositivi, e inoltre consente l'utilizzo di tecnologie e protocolli standard delle telecomunicazioni (già esistenti) per implementare la comunicazione tra essi.

In definitiva, nonostante la versione 'remote' presenti numerosi vantaggi, una soluzione di trade off, che implementi tutte le features della comunicazione via rete, ma anche le caratteristiche di automatizzazione delle operazioni della versione embedded, risulterebbe probabilmente la più completa sotto ogni punto di vista, in quanto offrirebbe un'interfaccia wireless semplice, comoda per l'utente e facilmente scalabile, ma anche la capacità di automatizzare l'esecuzione di servizi e routine, senza richiedere l'intervento diretto dell'utente.