Simone Sasso, Roon Mullaaliu

Report IoT – gruppo 8

Il report è suddiviso per laboratori, e ogni punto nell'elenco rappresenta un esercizio. Per i laboratori 'guidati' mi sono limitato a fare una breve descrizione degli esercizi, non ritenendo interessante l'approfondimento.

HW LAB 1

1) Pilotaggio di due led con periodo differente. Il led rosso viene modificato nel loop e il periodo è dato dalla funzione delay. Per il led verde si utilizza la libreria *TimerOne* che permette di lanciare una procedura di interrupt alla scadenza di un timer. Lo stato del led viene cambiato dentro la procedura.
2) L'esercizio precedente è modificato aggiungendo la comunicazione tramite seriale. Quando si inserisce 'R' viene stampato lo stato del led rosso, quando si inserisce 'G' lo stato di quello verde. Questo avviene nella funzione *serialPrintStatus* presente nel loop.
3) Conteggio dei passaggi davanti al sensore PIR. La funzione *attachInterrupt* permette di far partire la procedura di interrupt *checkPresence* quando cambia lo stato del sensore. Dentro la procedura, se lo stato del sensore è alto, si incrementa il contatore di persone.
4) Pilotaggio della ventola tramite seriale. La costante *STEP* indica il numero di velocità selezionabili. Quando inserisco un + o un – sulla seriale, per cambiare la velocità basta aggiungere o sottrarre *255/STEP* alla velocità corrente (*current_speed*).
5) Stampa i valori di temperatura letti dal sensore sulla seriale.
6) Stampa i valori di temperatura letti dal sensore sul monitor LCD. Nel loop uso la funzione *setCursor* per scrivere il dato dopo "Temperature:".

HW LAB 2

1) Il primo punto è realizzato dalla funzione *airConditioner*, che accetta come parametri la temperatura e i valori dei 2 set-points (*tmin* e *tmax*). Se la temperatura è inferiore o superiore ai set-points, la velocità della ventola è 0 o a 255. Altrimenti, per trovare la velocità utilizzo l'equazione della retta passante per i punti (tmin,0) e (tmax,255), ovvero (temperature-tmin)/(tmax-tmin)= (speed-0)/(255-0).
2) Procedimento analogo al caso precedente. Implementato nella funzione *heater*.
3) Quando lo stato del PIR diventa alto, viene attivata la procedura di interrupt *checkPir*, che porta a 1 la variabile *pres* (se uguale a 1 indica la presenza di persone nella stanza) e azzera *seconds*. Per misurare il tempo, inizializzo *Timer1* a un secondo. Ogni secondo viene chiamata la funzione *resetPres*. Nella funzione incremento *seconds* e controllo se sia maggiore o minore di *timeout*_pir. Se è maggiore, azzero *pres*.

4) La funzione *checkSound* è chiamata nel loop. Quando il sensore rileva un rumore incremento *n_sound_events*, azzero *seconds* e inserisco un delay di un secondo per evitare di rilevare più eventi dello stesso suono. Per controllare se vengono rilevati *n_sound_events* in un intervallo di tempo pari a *sound_interval*, uso la variabile *sound_seconds*. Questa viene incrementata ogni secondo (funzione *resetPres*), ma viene azzerata quando diventa uguale a *sound_interval* e *n_sound_events* è almeno 50.
5) Per come è strutturato il codice, se almeno uno dei due sensori rileva una presenza si assume che ci siano persone nella stanza (*pres*=1).
6) Questo punto è realizzato nella loop. Se ci sono persone nella stanza (*pres*=1), alle funzioni *airConditioner* e *heater* passo i set-points associati alla presenza (quelli che terminano con la 'P'). Altrimenti, se *pres*=0, vengono passati gli altri set-points.
7) Dovendo alternare tra due schermate ogni 5 secondi, ho creato due funzioni diverse per stampare i messaggi. Per effettuare il controllo dei secondi utilizzo la variabile *lcd_seconds*, che viene gestita nella funzione *resetPres*. Nella loop, se sono tra 0 e 5 secondi, chiamo *lcd_print1*. Questa stampa sulla prima riga la temperatura e il valore di *pres*, poi si sposta sulla seconda riga e stampa le percentuali di attivazione della ventola e del riscaldatore. Sempre nella loop chiamo *lcd_print2* quando *lcd_seconds* è compresa tra 5 e 10. Questa funzione accetta come parametri i 4 set-points, che cambiano in base alla presenza o meno di persone nella stanza. Sulla prima riga stampa i set-points del condizionatore, sulla seconda quelli del riscaldatore.
8) Il formato dei comandi è indicato nel codice, commentato di fianco all'inizializzazione delle variabili. Nella loop chiamo la funzione *checkInput*, che si aspetta un input del tipo "fm0 25" traducibile come: "temperatura minima per attivare la ventola è 25°". Per dividere il comando dal valore di temperatura utilizzo la funzione *substring*. Successivamente, la funzione legge il comando e modifica il set-point relativo.

SW LAB 1

1) Creazione di un server con *cherrypy* per convertire valori di temperatura ricevuti tramite GET. Nel metodo GET controllo che i parametri inseriti siano nel formato corretto, altrimenti stampo un messaggio di errore. La conversione avviene nel metodo *convert*.
   L'esercizio 1 presente nella cartella SW_1 è stato modificato durante il lab HW 3. Le modifiche le tratterò nella sezione relativa a quel laboratorio.
2) Analogo al precedente, con i parametri passati separati da "/".
3) Simile ai due esercizi precedenti, qui si usa una PUT che accetta un JSON contenente una lista di valori da convertire e le unità di misura. Con un ciclo for converto i singoli valori e poi li aggiungo alla lista *results*.
4) Creazione di un server per ospitare la dashboard "freeboard". Nella GET fornisco la pagina "index.html". Nella POST salvo la configurazione.

Simone Sasso, Roon Mullaaliu

HW LAB 3

1) Nel primo esercizio si utilizza la Yùn come server http, tramite la libreria *Bridge*. La Yùn accetta richieste GET per pilotare i led o per fornire valori di temperatura.
2) Lo sketch Arduino legge la temperatura dal sensore e invia una POST utilizzando *curl*. Lo sketch di questo esercizio contiene già le modifiche richieste durante il lab SW 2, ma le tratterò più avanti. Gli esercizi 1 e 2 del lab SW_1 sono stati modificati in modo da accettare la richiesta POST proveniente dalla Yùn, caricando i valori di temperatura nella lista *logs*. Anche la GET è stata modificata, in modo che stampi la lista *logs* ogni volta che venga effettuata una richiesta verso la risorsa "/log".
3) La Yùn viene utilizzata come client MQTT che pubblica i valori di temperatura letti dal sensore, e riceve comandi per pilotare i led tramite *subscribe* al relativo topic.

SW LAB 2

1) Ogni punto dell'elenco corrisponde a una richiesta.
   1.1) La prima richiesta è stata resa accessibile sul path "/broker". La classe *Broker* inizializza l'IP del broker e la porta nel metodo *__init__*. La GET restituisce IP e porta quando si accede a "/broker".
   1.2) La classe *Devices* si occupa della seconda richiesta. Come struttura dati per memorizzare i devices ho scelto un dizionario, che viene inizializzato nell'*__init__*. Inoltre, sempre nel costruttore, viene lanciato il metodo *readFile*, che si occupa di caricare sul dizionario eventuali devices presenti nel file "data.json". Il costruttore si occupa anche di creare e lanciare il thread *Controller* che implementa la funzione di eliminare tutti i devices e i servizi con insert-timestamp maggiore di due minuti. *Controller* accetta come parametro, oltre al dizionario dei devices, anche il metodo *updateFile*, che quando chiamato sovrascrive il file "data.json" con i valori attualmente presenti nel dizionario *devices*. Infine, il costruttore avvia un subscriber iscritto al topic "/tiot/8/devices", in modo da permettere la registrazione dei devices via MQTT, come richiesto dall'esercizio 5. Per quanto riguarda la registrazione via REST, la classe *Devices* implementa la POST, che accetta solo i devices che mandano la richiesta con un determinato formato, mentre stampa un messaggio di errore in caso contrario. In fase di registrazione, il catalog si occupa di inserire il timestamp corrente tramite il metodo *time*().
   1.3) La GET si occupa di restituire il dizionario dei devices connessi al catalog quando si accede al path "/devices/registered". Se la lunghezza del dizionario è 0, viene stampato un messaggio di errore in formato JSON.
   1.4) Sempre nella GET, se viene richiesto come parametro il "deviceID" e questo è presente nel catalog, viene restituito il device corrispondente. Altrimenti, se il device non è nel dizionario oppure se viene richiesto un parametro diverso da "deviceID"

viene stampato un messaggio di errore.

I punti successivi, da 1.5 a 1.10, seguono un procedimento perfettamente analogo a quello descritto nei punti precedenti. La classe *Users* si occupa degli utenti, la classe *Services* si occupa dei servizi.

2) Un semplice client per verificare il funzionamento del catalog.
3) Un client che simula un sensore IoT per verificare che i devices vengano registrati correttamente nel catalog.
4) L'esercizio 2 del lab HW 3 viene modificato in modo che si registri al catalog. L'esercizio si trova nella cartella del lab HW 3.
5) Già trattato nella spiegazione del funzionamento del catalog.
6) Un publisher che simula un sensore IoT per verificare che la registrazione dei devices tramite MQTT funzioni correttamente.

SW LAB 3

Dato che gli esercizi 2 e 3 sono contenuti nell'esercizio 4, passerò direttamente a spiegare il funzionamento di quest'ultimo.

Sketch Arduino

La funzione *loop* dello sketch del lab HW 2 è stata modificata in modo da permettere la registrazione dei vari sensori al catalog, tramite delle publish sul topic "/tiot/8/devices". Registro anche i set-points, perché poi verranno modificati da remoto invece che dalla seriale. Ogni device che viene registrato ha la sua funzione *senMlEncode*, per rispettare il formato richiesto dal catalog. Tutti i devices che si registrano forniscono i propri end-points MQTT per permettere la comunicazione con essi. Nella funzione *setup* vengono effettuate due *subscribe* per i set-points e per il monitor lcd. L'invio delle informazioni di temperatura, presenza e rumore è effettuato nella *loop* tramite *publish*. Per quanto riguarda i comandi di attuazione, mi sono trovato di fronte a due possibili scelte. O permettevo di controllare la ventola e l'intensità del led (il riscaldatore) direttamente da remoto, e in questo caso i set-points non avrebbero più avuto senso, oppure permettevo solamente la modifica dei set-points. Ho scelto la seconda opzione perché mi è sembrata più sensata, dato che con la modifica in remoto dei set-points si controllano indirettamente sia la velocità della ventola sia il riscaldatore. Quando viene inviato un messaggio sul topic "tiot/8/setpoints", parte la procedura *modifySetPoints*. Questa chiama la funzione *checkInput* che si occupa di modificare il set-point richiesto, ricevendo come parametri la stringa che identifica il set-point e il suo valore.
Per mostrare i messaggi inviati da remoto sul monitor LCD seguo un procedimento simile a quello per la modifica dei set-points. Nel *setup* viene fatta una *subscribe* al topic "tiot/8/lcd". Quando viene rilevato un messaggio pubblicato su quel topic, parte la funzione *displayMessage*. Questa si occupa di stampare il messaggio richiesto. Se il messaggio supera la lunghezza del monitor, allora viene fatto traslare a sinistra per permetterne la lettura.

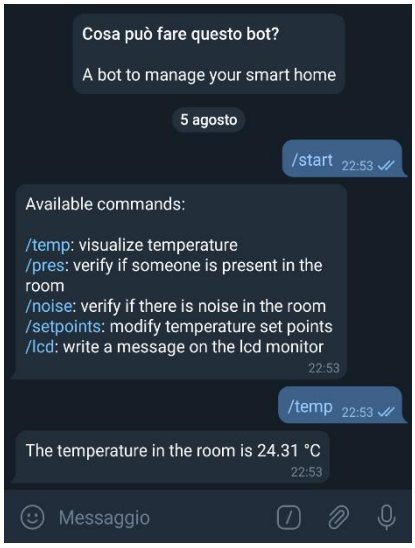
Python

Nello script python ho creato la classe *Mqtt* che implementa sia la publish che la subscribe. Nel costruttore vengono inizializzate le variabili *temp*, *pres* e *noise* e i relativi topic. In questo modo, quando viene ricevuto un messaggio su un determinato topic, la *myReceived* registra il dato nella variabile corrisponente.
Per registrare il Remote Controller al catalog, utilizzo un thread (classe *Register*) che manda l'iscrizione ogni minuto, dato che il catalog rimuove automaticamente tutti i servizi che non hanno aggiornato l'iscrizione negli ultimi due minuti.
Nel main, dopo aver avviato il thread per la registrazione, viene effettuata una richiesta GET al catalog per ottenere l'IP del broker e la porta. Successivamente, ci sono una serie di richieste GET per ottenere i topic MQTT dei vari devices registrati al catalog.
Una volta ottenuti tutti i topic, il programma visualizza il menu con i comandi per permettere all'utente di interagire con la smart home. In particolare, i servizi forniti sono la visualizzazione della temperatura, la verifica della presenza di persone o rumori nella stanza, la modifica dei set-points di temperatura e la possibilità di scrivere un messaggio sul monitor LCD.

La versione locale dello smart home controller, realizzata nel HW Lab 2, è più semplice da progettare e occupa meno memoria, tuttavia può essere utile solamente in piccoli ambienti, dove è presente solamente una scheda che gestisce i vari sensori. Nel caso di ambienti di grandi dimensioni, dove è richiesta la gestione di diverse schede, la versione remota è sicuramente più efficiente. Se dovessi ad esempio cambiare i set-points di 10 schede, con la versione locale dovrei spostarmi fisicamente andando a cambiare le impostazioni di ogni singola scheda, mentre con la versione remota sarebbe molto più immediato e non richiederebbe spostamenti. Inoltre, posso visualizzare le informazioni relative alla temperature e alla presenza di persone nella stanza anche se sono lontano da casa.

SW LAB 4

Per questo laboratorio ho scelto di modificare il lab SW 3, implementando la comunicazione con la smart home tramite bot di telegram. Per farlo, ho utilizzato la libreria "telepot".
Prima di tutto, ho creato un nuovo bot usando il *BotFather* di telegram. Il *BotFather* mi ha poi fornito il token, che mi ha permesso di accedere al bot e modificarlo. Il bot è gestito dalla classe *MyBot*. Il metodo *on_chat_message* gestisce i messaggi che arrivano dall'utente. Quando viene avviato ("/start") il bot visualizza un messaggio con tutti i comandi disponibili. Quando viene inserito il comando "/setpoints", il bot mostra i set-points modificabili sotto forma di pulsanti. Per fare questo ho usato l'oggetto *InlineKeyboardMarkup.*



Il metodo che gestisce le risposte dei pulsanti è *on_callback_query*, che si salva il tipo di set-point da modificare e chiede all'utente di inserirne il valore, oltre a settare il flag *sp* a 1. Il flag serve al metodo *on_chat_message* per sapere che il messaggio successivo inviato dall'utente è il valore del set-point. Uso il metodo del flag anche per i messaggi da mostrare sul monitor LCD.

In aggiunta ai servizi sviluppati nel lab SW 3, ne ho implementato uno nuovo che permette il controllo della concentrazione di gas metano all'interno di una stanza. Il file *gasSensor.py* simula un sensore di gas metano. Si registra al catalog come device, fornendo il topic MQTT su cui pubblica i dati sulla concentrazione del gas. Quando quest'ultima diventa superiore al 4% (da Wikipedia, il limite inferiore di esplosività è del 4.4%) il bot invia automaticamente un messaggio di allarme all'utente, informandolo di una possibile fuga di gas. Nel programma principale il thread *GasControl* si occupa di fare questo controllo ogni minuto.