Laboratorio parte Software

Laboratorio 1)

1. Abbiamo eseguito la consegna sviluppando su cherrypy una metodo GET che, dopo aver fatto vari controlli sul dizionario restituito da **params, dividesse l'esecuzione con costrutti if-else in base al tipo di unità di misura immessa e richiesta.

2. Come sopra, ma questa volta utilizzando l'*uri si è snellito il numero di controlli da fare, diminuendo il numero di possibili errori immessi dall'utente.

3. Riprendendo l'esercizio 1 per il tipo di controllo errori (poiché si invia un dizionario sotto forma di json), si è aggiunta la lettura del body della PUT e un ciclo for per permettere la conversione di tutti i valori che poi abbiamo inserito nel dizionario come nuova entrata sotto forma di lista.

4. In questo esercizio abbiamo caricato i file statici mediante conf e, dopo aver analizzato la pagina web, abbiamo ricavato quale metodo e quali elementi dell'url (in questo caso l'uri) fossero preposti al salvataggio del layout e in quale file si andasse a scrivere questa informazione, quindi abbiamo scritto il metodo POST con queste caratteristiche.

Laboratorio 2)

1. Abbiamo svolto l'esercizio creando un catalog con gestione da file e uno con gestione di database interno: sebbene quello con database interno risulti più comodo e leggibile e si presti meglio a piccole modifiche del codice (abbiamo usato infatti questa versione nei laboratori e esercizi successivi), tuttavia presenta lo svantaggio di non poter tenere memorizzati i dati di dispositivi/servizi/utenti in seguito al riavvio del server.

Il catalog con gestione file è provvisto di qualche controllo aggiuntivo nel main con dei try-except per verificare che i file esistano o meno e in caso negativo per crearli. Dispone anche di alcuni dizionari interni di appoggio per semplificare diverse operazioni.
Il catalog con gestione di database ha avuto qualche problema in più nella gestione dei dizionari dopo l'aggiornamento per timeout, quindi si sono dovuti usare dizionari ausiliari per quella sezione di codice la quale è stata gestita come thread aggiuntivo. Infine, in entrambi abbiamo usato in lettura di dati il formato senML per i dispositivi, ma li abbiamo memorizzati in una notazione diversa che rendesse più semplice la ricerca per ID.

2. In questo esercizio abbiamo sviluppato un semplice menù che permette attraverso l'interfaccia del terminale di accedere alle funzionalità richieste utilizzando la libreria requests.

3. In questo esercizio, usufruendo della libreria requests, abbiamo fatto in modo che il main mandasse periodicamente delle richieste POST su */devices*, le quali, una volta intercettate dal server del catalog, creano la entry del dispositivo fittizio. Il lavoro di aggiornamento del timestamp era già presente nel catalog e quindi non l'abbiamo aggiunto.

4. Poiché il catalog gestisce già da sé le richieste POST su */devices* in formato senML abbiamo solo dovuto modificare l'url della curl rispetto al codice del laboratorio Hardware 3.2.

5. Partendo dal catalog gestito mediante database, abbiamo aggiunto la classe MySubscriber che permette l'aggiunta di dispositivi con un procedimento del tutto simile a quello relativo alla POST su */devices*; abbiamo usato come topic *iot/21/devices*.

6. In questo esercizio abbiamo ripreso il codice di SW2.3 e abbiamo aggiunto una classe publisher; abbiamo anche provato a inserire come messaggio un dispositivo che registra due eventi in una sola volta per controllare che venisse comunque registrato normalmente.

Laboratorio 3)

1. Abbiamo creato due script (uno contenente una classe publisher mentre l'altro subscriber) che si scambiano messaggi attraverso mqtt: abbiamo usato come topic in cui pubblicare *iot/21/prova* e dal subscriber ci siamo iscritti ai topic *iot/21/+*, *iot/#* e *iot/+/prova*, verificando di ricevere con tutti e tre i subscriber il messaggio.

2. In questo esercizio abbiamo prodotto tre script: uno è il catalog, essenzialmente uguale a quello dell'esercizio 2.5 di modo che potesse lavorare con richieste mqtt; un file .ino in cui pubblichiamo periodicamente sul topic *iot/21/devices* per la registrazione al catalog e poi sul topic *iot/21/temperature* per fornire i valori letti da sensore; infine un file .py in cui abbiamo creato un client che si iscrive al catalog come servizio, legge i dati di porta e broker dal catalog, fa richiesta di ricerca tramite ID mediante la libreria requests e memorizza l'endpoint di Arduino che permette di iscriversi al topic *iot/21/devices*.

3. Questo esercizio è molto simile al precedente ma cambia il comportamento del codice caricato su Arduino (oltre al publisher su *iot/21/devices* si ha un subscriber su *iot/21/led* al posto dell'altro publisher) e specularmente cambia quello del client che questa volta fungerà da publisher.

4. Anche in questo esercizio abbiamo i soliti tre file dei due esercizi precedenti, ma sia da lato Arduino, sia da lato python abbiamo publisher e/o subscriber multipli. In questa versione evoluta del laboratorio HW2 abbiamo riscontrato numerose criticità: la necessità di passare numerosi messaggi tra due hardware differenti che eseguono programmi con linguaggi di programmazione diversi crea abbastanza confusione e nell'insieme la leggibilità del progetto è bassa (anche se i segmenti di codice risultano più leggibili se presi singolarmente); i ritardi derivati dallo scambio di messaggi sono molto numerosi, quindi non si ha la velocità di azione del sistema del laboratorio HW2; se si vogliono usare librerie per la gestione di messaggi di formato .json, la memoria di Arduino risulta molto limitante (abbiamo infatti trovato preferibile non allocare memoria per il sender e crearlo direttamente come stringa perchè altrimenti sorgevano numerosi problemi anche seguendo le istruzioni del sito fornito nel laboratorio HW3). D'altra parte è innegabile che ci sono anche pregi relativi a questo approccio: la programmazione in python è molto più snella e chiara rispetto a quella su Arduino e consente operazioni con librerie più numerose e con funzioni più particolari di quelle che si potrebbero avere su Arduino; a questo proposito se si riesce a bypassare la limitazione inizialmente data da Arduino sulla memoria, una volta mandati i messaggi su pc la loro elaborazione può essere sia più veloce, sia più complessa (non avendo i limiti di cui sopra). In generale abbiamo concordato sul fatto che questo tipo di soluzione può essere vantaggiosa se si ha a che fare con elaborazioni di dati molto più complesse di quelle relative a questo esercizio.

Laboratorio 4)

1. In questo laboratorio abbiamo cercato di simulare un dispositivo per smart home che fungesse da antifurto. Il funzionamento è il seguente: abbiamo il catalog su cui si registra l'Arduino (questa volta tramite richiesta POST con una curl in cui il dato è formattato in senML gestito manualmente come stringa), un client che registra gli utenti al catalog, un subscriber che si iscrive come servizio al catalog e quando riceve un messaggio da Arduino tramite mqtt (che parte quando si verifica una segnalazione da parte del sensore di movimento e una del sensore di rumore), fa partire una funzione che scrive un file di testo personalizzato, formattato come se fosse una mail, per ogni utente che si è iscritto al catalog.

C'era la possibilità di utilizzare una vera mail per inviarle attraverso il codice, ma sarebbero sorti alcuni problemi di sicurezza e sembrava eccessivo creare un account e-mail solo per questo esercizio, quindi la scrittura su file ci è sembrata un buon compromesso.