REST+MQTT - Riassunto REST e MQTT

Ingegneria del Software (Università degli Studi del Piemonte Orientale Amedeo Avogadro)

REST

REST = definisce un insieme di principi architetturali per sviluppare software. Non c'è nessun organismo che regola i principi fondamentali di questa architettura né si tratta di un sistema concreto.

REST non è un protocollo (perché un protocollo descrive come gli attori comunicano tra loro), non descrive messaggi ma specifica requisiti architetturali da soddisfare. Non è una specifica e non è uno standard. Non si adatta solo al web ma può essere generalizzato per qualsiasi altra applicazione.

Cos'è REST:
Dice come dev'essere architettato un sistema distribuito ma ignora i dettagli implementativi (no protocolli specifici, no pacchetti o standard sulle strutture del messaggio o contenuto), permette allo sviluppatore di concentrarsi sulle componenti e le loro interazioni.

Architettura vs Protocollo:
• Architettura = dice che i client possono contattare i server ma non il contrario, base delle strutture client-server. Pattern di comportamento, ad esempio se il traffico sul server diventa troppo elevato si può smistarlo su altri server.
• Protocollo = specifica anche i linguaggi di programmazione, framework.. tutto ciò che si può utilizzare a livello implementativo.

REST = astrazione di un'architettura, indica dei pattern comportamentali
(API) RESTful = costrutti software, programmi/script.. sviluppati seguendo i principi REST

Le API devino rispettare **6 principi fondamentali** per essere catalogate come RESTful:

- Architettura client-server : per poter essere implementata dev'esserci un client (attore che chiede le risorse) e un server (attore che ha al suo interno o è grado di interrogare un database che contiene le info che serve al client).

- stateless: richieste indipendenti tra loro.

- cache: supporto a caching, tra client e server implementate strutture di mezzo destinate al caching (software o altri server) che non sovraccaricano la rete per chiedere risorse. Queste strutture intermedie salvano le richieste più recenti e vengono interrogate dal client prima di interrogare il server, limitando lunghi tempi di attesa per ricevere la risorsa.

- sistema a livelli: client e server non comunicano quasi mai direttamente tra loro ma ci sono sempre componenti di mezzo che forniscono al client un'astrazione della rete a cui fa richiesta, alleggerendolo come incarichi perché non ha logica al suo interno ma si sviluppa come un presentation layer che serve alla rete in cui il server è contenuto una risorsa (load balancer = instradatori / router / server di instradamento posti come entry point della rete e in base al tipo di richiesta capiscono su quale server instradare la richiesta).

- codice on-demand (opzionale): aspetto poco utilizzato, consiste nel dare facoltà al server di ampliare le funzionalità del client trasferendo del codice eseguibile (pericoloso)

- interfaccia uniforme: i componenti devono comunicare attraverso un'interfaccia sviluppata nello stesso modo per permettere una comunicazione più efficiente e più facile da sviluppare. E' divisa in quattro aspetti: identificazione delle risorse (risorse che devono essere facilmente identificabili dal

client quando le riceve in risposta a una sua request), manipolazione delle risorse tramite rappresentazioni (i client ricevono informazioni che rappresentano tutto ciò di cui hanno bisogno per una modellazione, ossia consultazione della risorsa o modifica), messaggi autodescrittivi (il client deve ricevere tutto quello di cui ha bisogno per modellarla), ipermedia come motore dello stato dell'applicazione (il client deve poter individuare tutte le altre azioni disponibili su determinate risorse, si fa con i metodi crud (??)).

Vantaggi di REST:
Nonostante questi 6 vincoli rendano apparentemente lo sviluppo di questa architettura più complicato, presenta diversi vantaggi:

- separazione client-server: hanno uno sviluppo indipendente, non devono avere parti in comune. Permette di avere uno sviluppo dei componenti indipendente

- visibilità, affidabilità, scalabilità perché client e server risiedono su macchine differenti, utile perché se dovessimo aumentare la dimensione del server potremmo scalare solo quel componente senza scalare anche il resto del sistema

- indipendenza: da linguaggi di programmazione, framework, standard.. che permette una completa libertà nella scelta di questi aspetti per sviluppare i componenti del sistema. L'importante è mantenere uno standard su come vengono scambiati i messaggi (struttura condivisa). Il formato più utilizzato in un'architettura REST è il JSON.

Risorse:
Possono essere viste come delle istanze degli oggetti nella programmazione a oggetti, sono delle entità del mondo reale mappate su una struttura sola.

Vengono quasi sempre interrogate tramite il pattern URI: all'interno della richiesta http ci sono tutte le informazioni per ottenere una risorsa.
Es: http://www.myapp.com/client/1234 richiede la risorsa di un cliente identificato con ID 1234.

Per convenzione si tende a dare a ogni entità come chiave primaria un ID numerico. Oppure si può utilizzare una richiesta più descrittiva (es. http://www.myapp.com/prodotti?colore=rosso).

Come si richiedono le risorse:

Uno dei modi più utilizzati è HTTP con i suoi metodi che permette una mappatura 1:1 con le operazioni CRUD (Create Read Update Delete).

4 metodi principali:
- POST: crea una risorsa
- GET: richiede una risorsa
- UPDATE: aggiorna lo stato di una risorsa (es. aggiornamento dello stato di un utente)
- DELETE: elimina una risorsa

Verranno poi utilizzate le API Rest per comunicare con il server che avrà al suo interno un database in cui risiedono tutte le risorse di cui i client hanno bisogno.

Come funziona lo standard di trasferimento **JSON**:

JSON non è un linguaggio di programmazione, è uno standard per la formattazione dei messaggi tra vari attori all'interno del web. Formato testuale, molto leggero, facilmente parsabile da qualunque linguaggio di programmazione.

Come creare un oggetto JSON: parentesi iniziali, coppie di chiave-valore.

JSON nasce per evitare di perdere troppo tempo nell'interpretare i dati che arrivano quindi si tende a non utilizzare troppi livelli di annidamento.

MQTT

Protocollo pensato per la comunicazione tra macchine, è molto semplice e leggero, basato sul modello di comunicazione publish-subscribe.

Progettato per i dispositivi con tante capacità, es. durata batteria o banda di trasmissione.. e per reti che possono avere alta latenza o poco affidabili.

Caratteristiche che lo rendono ben adatto ai sistemi di IoT.

Esistono 2 diversi modelli di comunicazione:
1. classico, request/response: 1 richiesta a cui segue 1 risposta, tipico modello client-server (es. HTTP)

2. publish/subscribe, utilizzato da MQTT, ci sono dei publisher che pubblicano delle informazioni e le inviano a un intermediario che si occupa di inviare le informazioni ai subscriber. I subscriber per ricevere queste informazioni devono prima registrarsi e riceveranno le notifiche di avvenuta pubblicazione. Comunicazione mediata dal dispatching service e non più diretta come client server.

Es: i publisher sono i docenti che inseriscono i video e i subscriber sono gli studenti che ricevono la notifica. Il dispatching service è il server di YouTube.

Architettura
Nell'ambito dei sistemi IoT abbiamo il message broker (servizio di dispatching) e 2 tipi di client: uno che pubblica le informazioni (es. sensore) che periodicamente invia le sue informazioni e uno client che devono iscriversi e mostrare l'interesse nel ricevere le info del sensore.

Questo tipo di architettura funziona bene per comunicazioni a molti, i publisher potrebbero essere molti, anche i subscribers. Un publisher pubblica le info ma non è a conoscenza del numero dei subscribers, potrebbero anche non essercene.

In questo protocollo c'è un modello di comunicazione basato su protocollo TCP a livello di trasporto. Il broker è il punto centrale di comunicazione, quello che riceve tutti i messaggi prodotti dai vari publisher e si occupa di inoltrare ai corretti subscribers. Ogni messaggio ha un topic associato, cioè una classe di informazioni. Il subscriber per ricevere i messaggi deve prima sottoscriversi su uno o più topic.

Il subscriber decide poi cosa farne di questi dati: può leggerli e basta oppure agire di conseguenza.

Proprietà
Il publisher e il subscribers non si conoscono e non sanno nemmeno della loro esistenza perché interagiscono solo tramite il message broker. Devono però conoscere come raggiungere / inviare informazioni al broker. E' quest'ultimo che filtra e indirizza i messaggi distribuendoli ai subscriber corretti.

In questo modo abbiamo diversi tipi di disaccopiamento tra publisher e subscriver:

• di tipo spaziale: non si conoscono e non è necessario che conoscano i loro rispettivi indirizzi IP, porte, ecc..

• temporale: P e S non è necessario che siano connessi nello stesso istante di tempo perché i messaggi inviati sono ricevuti dal broker che si preoccuperà di spedirli quando i S sono connessi.

• sincronizzazione: non ci sono operazioni in cui bisogna attendere le risposte dell'altro, il P può pubblicare e disconnettersi subito dopo.

Topics
Meccanismo che permette al broker di indirizzare correttamente le informazioni. IL topic è implementato come una semplice stringa codificata in UTF-8. Questa stringa può avere più livelli gerarchici che sono separati da uno slash.

Es. "home/livingroom/temperature" —> Temperature rilevate dal sensore che si trova nel livingroom

La suddivisone gerarchica del topic è stabilita chi usa il protocollo. Il protocollo dice solo che i topic vanno gestiti in modo gerarchico, sta all'utilizzatore gestire questa gerarchia, decidendo quale è più adatta al suo sistema.
Può essere una suddivisione spaziale (come quella dell'esempio) per identificare le stanze della casa, oppure in base alla tipologia di dato.

I S possono anche iscriversi a un sottoinsieme di topic, definito usando delle wildcard. Disponibili 2 tipi:

• + : rappresenta un singolo livello della gerarchia
home/+/temperature —> ricevi le informazioni di tutte le stanze

• # : wildcard multilivello, sottoscrizione a tutti i livelli di una particolare gerarchia.
home/# —> tutti i topic definiti sotto home (es. tutte le stanze e tutti i parametri, non solo temperatura)

Caratteristiche offerte - QoS
Permette di pubblicare un determinato messaggio con una specifica qualità del servizio. Ne definisce 3 e sono associate all'affidabilità nella spedizione del messaggio. Sia i client che il broker possono fornire ulteriori meccanismi per incrementare l'affidabilità in caso di fallimenti della rete ecc.

Il protocollo definisce 3 livelli QoS:

• 0: at most once delivery. Assicura che venga inviato al più un messaggio. Il sender usa un meccanismo di best effort per il messaggio e si basa del tutto sull'affidabilità di TCP. Il messaggio può andar perso e il protocollo MQTT non utilizza nessun meccanismo ulteriore di ritrasmissione. Potranno esserci delle perdite e al massimo riceverò 1 copia di quel messaggio.

• 1: at least once delivery. SI garantisce che il ricevente otterrà quel messaggio almeno 1 volta. Se per caso il ricevente non dà l'hack del messaggio, questo verrà rispedito finché non ottengo un hack. Possibilità di avere delle duplicazioni dei messaggi.

• 2: exactly once delivery. Viene garantito che il messaggio venga spedito esattamente una volta sola utilizzando un meccanismo di hand shake a 4 step. Incrementa l'overhead di comunicazione però è la soluzione migliore se non accetto duplicazioni o perdite dei messaggi. Dipende dal tipo di applicazione.

Altre opzioni:

• LWT: è un messaggio che viene specificat dal P nel momento in cui si connette con i broker. Questo messaggio viene spedito ne momento in cui il P può disconnettersi dal broker in modo inaspettato ed è quello che viene spedito dal broker ai vari S nel momento in cui il P muore o cade in modo inaspettato. A seconda dell'applicazione il S potrà decidere di ignorare il fatto o di prendere dei provvedimenti.

• Retained messages: si può stabilire che un messaggio venga inviato come un retained message, ossia un messaggio persistente. Per uno specifico topic il broker mantiene l'ultimo valore conosciuto su quel particolare messaggio se inviato come messaggio retained. Ogni volta che un nuovo client si sottoscriver a quell specifico topic, riceverà l'ultimo messaggio retained valido su quel topic. Si assicura almeno 1 messaggio anche se la sottoscrizione è avvenuta dopo.

• clean / durable session: come sono le sessioni, nel momento in cui ci si connette al broker il client può dire come è la connessione: tramite un clean session flag che può essere settato a true (= tutte le sottoscrizioni che il client/S aveva fatto vengono cancellate quando ci si disconnette dal broker. Se si riconnette dovrà iscriversi di nuovo) o false (= la connessione con il broker è persistente, il client può disconnettersi ma il broker mantiene le informazioni delle sottoscrizioni. Quando si riconnette riceve tutti i messaggi con QoS 1 o 2.

Brokers
Quali tipi di broker sono disponibili all'uso:

• eclipse mosquitto: broker open source, in C, supporta MQTT 3.1 e 3.1.1, è molto leggero
• HiveMQ: molto affidabile
• IBM WebSoereMQ: commerciale

Client Library
Librerie che permettono di sviluppare applicazioni che utilizzano questo protocollo. Diverse per differenti linguaggi:

• eclipse Paho: implementazione open source del protocollo MQTT. Noi ci concentreremo sulla libreria Java.
Installazione: https://youtu.be/h_WJcWOhCSA?list=PL-mYjmFx7mFlt_3pKpmxCWYab2Jhqpek_&t=2862
• M2MQTT
• Fusesource MQTT Client

Client Tools

Client per debug, forniscono un tool per fare delle sottoscrizioni a topic di un particolare server/broker MQTT