**Indice**

* Introduzione
  + Descrizione progetto
  + Principali tecnologie implementate
* (Capitolo 1) Panoramica tecnologie
  + 1.1 Event-driven architecture
    - Spiegazione
    - Confronto con modello request/reply
  + 1.2 Publish/subscribe
    - Spiegazione
    - Confronto con altri pattern
    - Motivazioni scelta di implementazione nel progetto
  + 1.3 Redis
    - Spiegazione
    - Confronto con database relazionali
    - Motivazioni scelta di implementazione
  + 1.4 MQTT
    - Spiegazione
    - Confronto con altri metodi di comunicazione votato all’IoT
    - Motivazioni scelta di implementazione nel progetto
  + 1.5 Server-sent events
    - spiegazione
    - Confronto con WebSocket e Long-polling
    - Motivazioni scelta di implementazione
* (Capitolo 2) Realizzazione progetto
  + [TODO]

# Sommario

[TODO]

# Introduzione

Con il seguente elaborato, descrivo l’utilizzo di pattern publish/subscribe usando il protocollo MQTT e Server Side Events per la comunicazione asincrona tra sistemi eterogenei all’interno della rete. In particolare vado a descriverne il funzionamento tramite un progetto realizzato con lo scopo di visualizzare le statistiche di vendita di un bar sia con un’ interfaccia web-based, che con un Arduino che comanda un device IoT.

**Capitolo 1 Panoramica tecnologie  
Event driven architecture**

**https://svn.wso2.org/repos/wso2/scratch/papers/mediation/ieee/papers/SOA&EDS-using-ESB.pdf**

Gli appicativi moderni si ritrovano sempre più spesso di fronte al problema di dover far comunicare più elementi eterogenei tra loro, come per esempio un applicativo web con sensori IoT e un’applicazione mobile; oppure anche più microservizi separati di una stessa applicazione. Per forza di cose questi elementi avranno caratteristiche e funzioni molto diverse tra loro, rendendo difficile la comunicazione.

Negli anni si è adottato sempre il pattern “request/response” che vede un elemento del sistema chiedere ad un secondo (o a più) i dati necessari, questo creerà una risposta e la invierà al primo che potrà riprendere con l’esecuzione normale. Questo pattern funziona bene ed è di semplice implementazione, ma ha alcuni punti in cui non risulta ottimale, come ad esempio la situazione in cui un elemento debba compiere delle operazioni solo quando una risorsa fornita da un secondo è pronta per essere elaborata.

Per risolvere questo problema è stato creato il pattern architetturale Event-driven.

L’architettura basata ad eventi è un modello che permette ad elementi differenti di un sistema di poter comunicare basandosi sulla produzione e il rilevamento di eventi. Per “evento” si intende un cambio significativo di stato che può essere di interesse all’interno del sistema, per cui valga la pena che gli altri componenti vengano notificati.

Questo modello va a sostituire, in determinate situazioni, il tradizionale modello “request/response”, in quanto non vi sarà più bisogno di richiedere le risorte per poter continuare l’esecuzione, ma ogni elemento verrà notificato e riceverà la risorsa automaticamente.

I vantaggi chiave dell’architettura Event-driven, sono diversi:

un vantaggio chiave sta nel fatto che ogni elemento del sistema è estraneo agli altri e si occuperà solo di inviare o ricevere eventi con un servizio dell’applicazione appositamente creato con lo scopo di gestire il traffico di questi messaggi chiamato “broker”;

questo porta ad un secondo vantaggio, per il quale un evento inviato, può essere ricevuto da diversi elementi, creando quindi una relazione 1 a N tra chi invia e chi riceve, non più come prima che era intrinsecamente 1 a 1;

altro punto importante, è (in generale) il minor uso di risorse in quanto non serve aspettare una risorsa in polling o effettuare richieste a vuoto sperando che sia disponibile ma sarà sufficiente reagire quando viene rilevato l’evento.

L’architettura Event-driven trova applicazione nel pattern Publish/Subscribe.

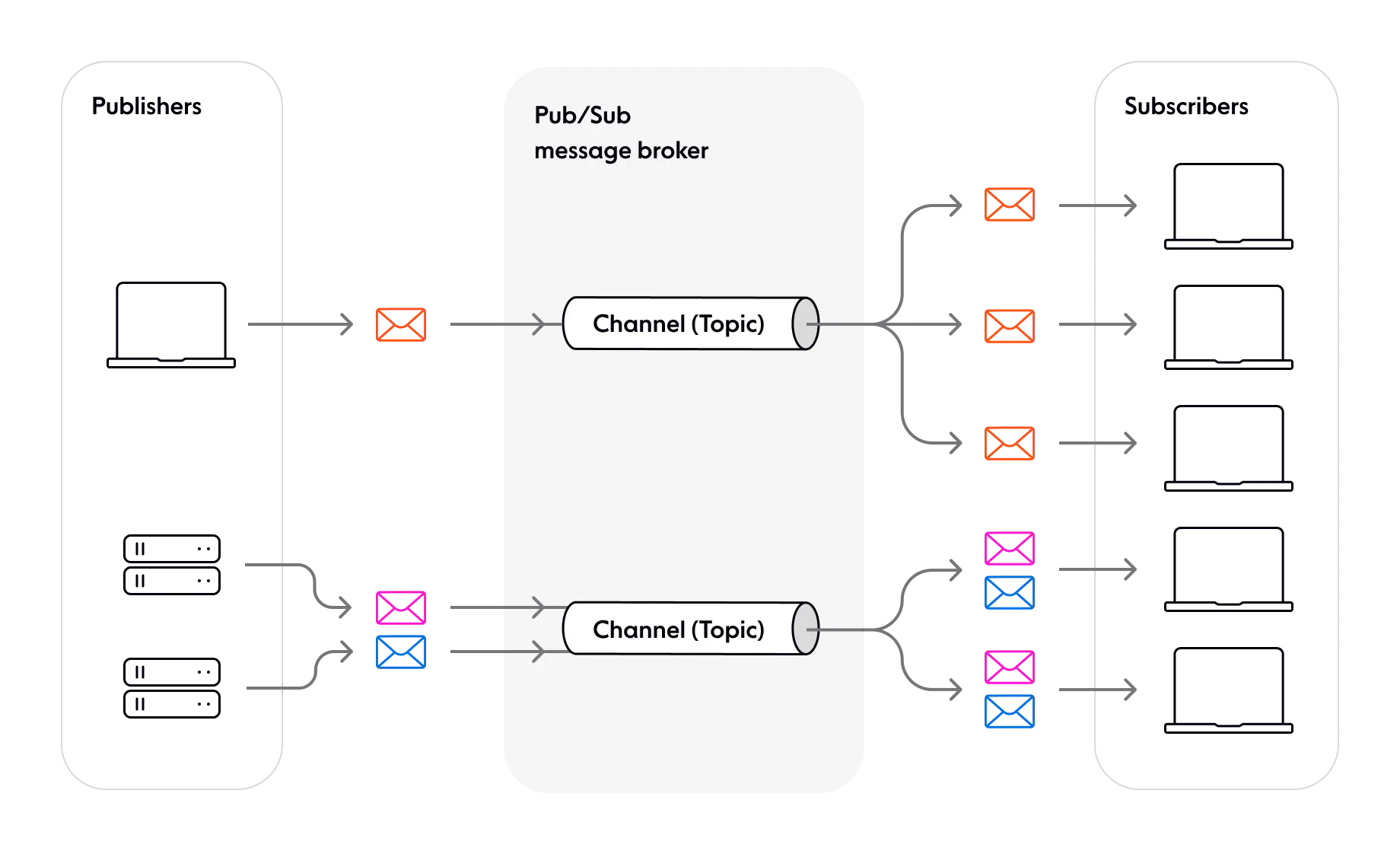
**Pattern Publish/Subscribe**

Abbreviato spesso in pub/sub, è un pattern usato per permettere una comunicazione asincrona tra componenti di un applicativo prevedendo la pubblicazione di un evento e la sua ricezione tra le componenti.

La particolarità importante di questo sistema sta nel fatto che i componenti hanno un compito ben definito ma non hanno coscenza di tutto il resto della rete, rendendo il tutto molto scalabile e adatto a elementi eterogenei tra loro.

Gli elementi principali di questo pattern sono:

* Publisher: l’elemento che invia il messaggio su un canale definito. Si occupa solo dell’invio e non si preoccupa se ci sono iscritti al canale su cui sta pubblicando o se qulacuno ha effettivmente letto i messaggi.
* Subscriber: viceversa è l’elemento che usufruisce dei messaggi forniti in un certo canale. Sempre per rispettare le regole del pattern, non ha idea di chi abbia pubblicato i messaggi, ma si limita a leggerli.
* Broker: è di fatto un intermediario che gestisce lo scambio di messaggi tra i vari publisher e subscribers nei vari canali
* Canali: Sono “topic” in cui i publishers inviano i messaggi che i subscribers leggono e servono per differenziare

https://ably.com/topic/pub-sub-benefits

#### Differenze con Observer pattern

Tra le varie possibilità di implementazione dell’architettura Event-driven, il pattern che più viene paragonato al pubsub è l’observer pattern, perché presentano caratteristiche comuni tra loro.

L’observer pattern è un modello di comunicazione che permette di inviare notifiche a più “observer” quando un soggetto cambia di stato. La principale differenza sta nel fatto che questo pattern crea una relazione 1 a N, con un oggetto osservato da più listererns, a differenza del pub/sub nel quale le relazioni possono essere anche N a M, in quanto N publishers possono inviare dati su diversi topic che verranno letti da M subscribers.

Nell’progetto realizzato, per i motivi sopra citati, ho deciso di adottare il pattern pub/sub piuttosto che l’observer perché la relazione 1 a N su determinati topic sarebbe potuta diventare una limitazione pensando in ottica di scalabilità, magari in uno scenario a più microservizi con necessità di trasmettere su più canali.

## Redis

Redis (acronimo di Remote Dictionary Server) è un software open-source uasto per salvare strutture dati in-memory ed è caratterizzato dalla velocità e la semplicità di utilizzo che lo rende molto versatile sia per applicazione web che sistemi IoT e dispositivi mobili.

Funziona salvando i dati in formato chiave-valore e risulta molto più veloce di database classici perché è interamente in-memory, abbattendo i tempi di accesso ai dati rispetto agli altri che salvano i dati su disco; questo introduce il problema della persistenza per cui Redis, dopo un riavvio, perde tutti i dati salvati. Per questo motivo non va a sostituire i database tradizionali, ma trova il suo utilizzo classico come sistema di caching, gestore di sessioni e broker di messaggi. Proprio questa sua ultima caratteristica lo rende un ottimo candidato per l’utilizzo in sistemi che usano il pattern Publish/Subscribe.

Altre caratteristiche fondamentali di Redis, sono

## MQTT

## Server Sent Events

Le applicazioni web utilizzano il protocolo HTTP per permettere la comunicazione tra client e server, usando metodi come GET o POST per accedere alle risorse.

Il funzionamento è tipicamente in forma “request/reply” e prevede l’inizio della comunicazione TCP da parte del client, l’accettazione da parte del server, lo scambio di messaggi testuali e infine la chiusura della connessione. Questo metodo funziona benissimo nella maggior parte dei casi.

Vi sono però situazioni in cui, ad esempio, il client necessita di una serie di dati continui da parte del server, oppure quando sarebbe richiesta una comunicazione ad eventi.

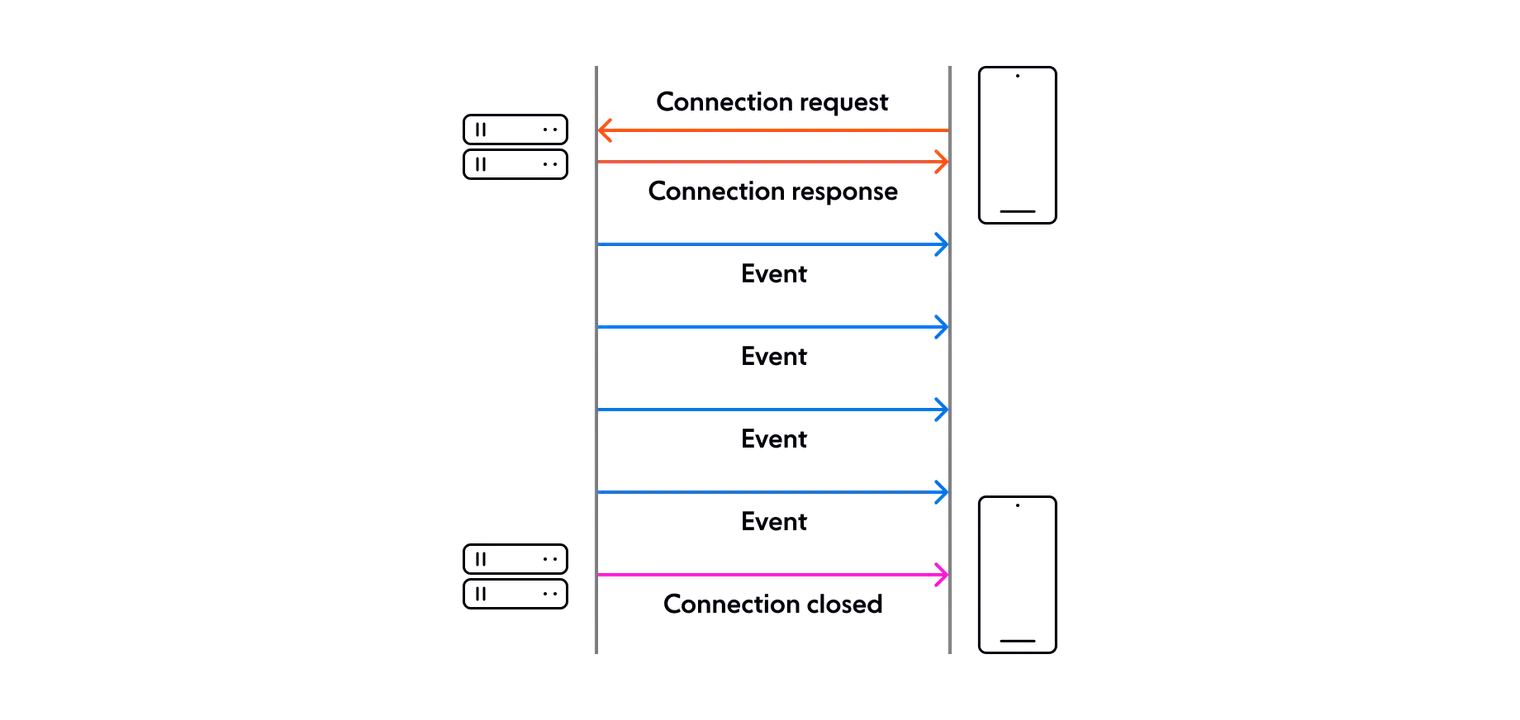
Usando semplicemente il protocollo HTTP, l’unico modo per ricevere continuamente degli update, sarebbe usare il polling, andando ad interrogare il server a ripetizione dopo un certo intervallo di tempo. Questo come già detto in precedenza, porterebbe inevitabilmente ad uno spreco di risorse e con potenziali chiamate inutili.

Per ovviare a questo problema, sono stati creati protocolli che permettono di implementare l’architettura Event-driven anche in una comunicazione web.

Quello scelto e implementato nell’applicazione in esame è il protocollo Server Sent Events.

Abbreviato in SSE, si tratta di un modello di comunicazione web ad eventi, che permette di aprire una comunicazione con cui i dati vengono inviati in modo asincrono a fronte di un’unica richiesta da parte del client e funzionano usando una connessione HTTP “di lunga durata” che rimarrà aperta finché uno tra client o server non la termina.

Il protocollo Server Sent Events ha la caratteristica di essere unidirezionale, che comporta l’invio di dati di tipo testuale solo da parte del server e in maniera asincrona e, anche grazie a questa caratteristica, l’implementazione risulta più semplice rispetto ad altri protocolli esistenti, in quanto sarà sufficiente predisporre il sistema a leggere i dati inviati dal server. Un’altra caratteristica importante, sta nella riconnessione automatica, per cui nel caso in cui vi sia una perdita di connessione il client tenterà di ristabilire la comunicazione automaticamete senza dover implementare la gestione dei codici di ritorno del protocollo HTTP.

[](https://ably.com/blog/websockets-vs-sse)https://ably.com/blog/websockets-vs-sse

Di seguito sono riportati i confronti con le soluzioni alternative più usate attualmente.

Il più simile è sicuramente il protocollo WebSocket, che consente di mantenere aperta una comunicazione bidirezionale client-server e implementa la gestione degli eventi. Da notare che la comunicazione per questo protocollo è full-duplex(1) attraverso una singola connessione TCP.

Le principali differenze con il SSE sono due: la prima è che non è previsto un meccanismo di riconnessione automatica; la seconda sta nel fatto che la comunicazione sia bidirezionale e questo, come già detto, comporta più complessità di implementazione perché ci sarà da gestire tutto il meccanismo di ricezione e risposta. Per l’applicazione in esame non ho usato WebSocket proprio perché non avevo necessità della bidirezionalità della comunicazione e, invece, mi ha fatto comodo il sistema di riconnessione automatica.

Altro modello molto usato è il Long Polling, che permette di avere una connessione aperta con il server senza usare nessun tipo di protocollo specifico. Il suo funzionamento è molto semplice: dopo aver inviato una richiesta al server, quest’ultimo non chiuderà la comunicazione finché non avrà un messaggio da inviare e, una volta che il messaggio è stato spedito e la comunicazione chiusa, il client si occuperà di riaprirne una nuova subito dopo.

Il Long Polling non rientra nella categoria dei protocolli Event-based, perchè il client deve occuparsi di aprire una connessione e di aspettare una risposta, senza però che vi sia un sistema di update o notifica e questo, come già detto, risulterà sempre in uno spreco di risorse, anche perché la connessione va riaperta ogni volta che arriva un nuovo dato dal server. Questo è il motivo principale per cui ho scelto di non implementarlo nell’applicazione proposta.

(1) Full-duplex: comunicazione biridezionale simultanea. Per fare un esempio pratico, è come la comunicazione del telefono, dove entrambi possono parlare e ricevere allo stesso momento

# Analisi iniziale

Il progetto è formato da più componenti hardware e software.

Partendo dal database usato dal software gestionale, che è un instanza di PostgreSQL 11 sulla quale, non avendo pieno accesso, mi sono dovuto limitare a leggere i dati in polling ogni tot secondi.

L’applicazione invece si divide in una parte di backend scritta in Python usando il framework Flask e una minima parte di visualizzazione frontend composta da semplici pagine HTML con aggiunta di bootstrap per curare la parte grafica.   
Il backend ha il compito di leggere i dati da database usando la libreria psycopg2, elaborarli e avviare una comunicazione SSE per renderli accessibili al frontend, che a sua volta, usando la libreria Chart.js li visualizza in grafici.  
Altro compito del backend è quello di rendere i dati accessibili anche tramite protocollo MQTT perché, in questo caso, c’è un Esp32 in ascolto che riceve i dati e li visualizza.

Poi l’applicazione è stata racchiusa in un container, usando docker, per permettere l’avvio combinato con l’istanza locale di Redis. L’uso di Redis è fondamentale per l’utilizzo dei Server Side Events che, essendo asincroni, hanno bisogno di un pool di dati sempre presente e pronto per essere letto (a differenza del polling su database). Inoltre mi ha permesso di rendere più flessibile l’accesso ai dati elaborati, permettendo di avere sempre disponibili i valori già formattati in base al tipo di visualizzazione scelta lato frontend.

### Scelta dei protocolli

Nel seguente paragrafo, andrò ad analizzare i protocolli utilizzati facendone il confronto con le loro controparti maggiormente in uso.

Partendo dal protocollo SSE

### Bibliografia