* Introduzione:
  + Descrizione progetto per visualizzazione statistiche bar
* Pattern Pub/sub
  + Spiegazione generale
  + Differenze con altri pattern
  + Server Side Events
  + MQTT
  + Redis
* Analisi iniziale progetto:
  + Analisi dei requisiti
  + Descrizione principali componenti e linguaggi utilizzati (PC, Raspberry/Arduino e Python)
  + Descrizione e scelta protocolli: Server-sent events, MQTT, Redis
* Implementazione:
  + Spiegazione fasi di sviluppo e relativo codice documentato
  + Test effettuati
* Risultato e conclusioni:
  + Riflessioni sul risultato ottenuto
  + Eventuali migliorie ed implementazioni future

# Introduzione

Con il seguente elaborato, descrivo l’utilizzo di pattern architetturale 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.

**Event driven architecture**

**Pattern Publish/Subscribe**

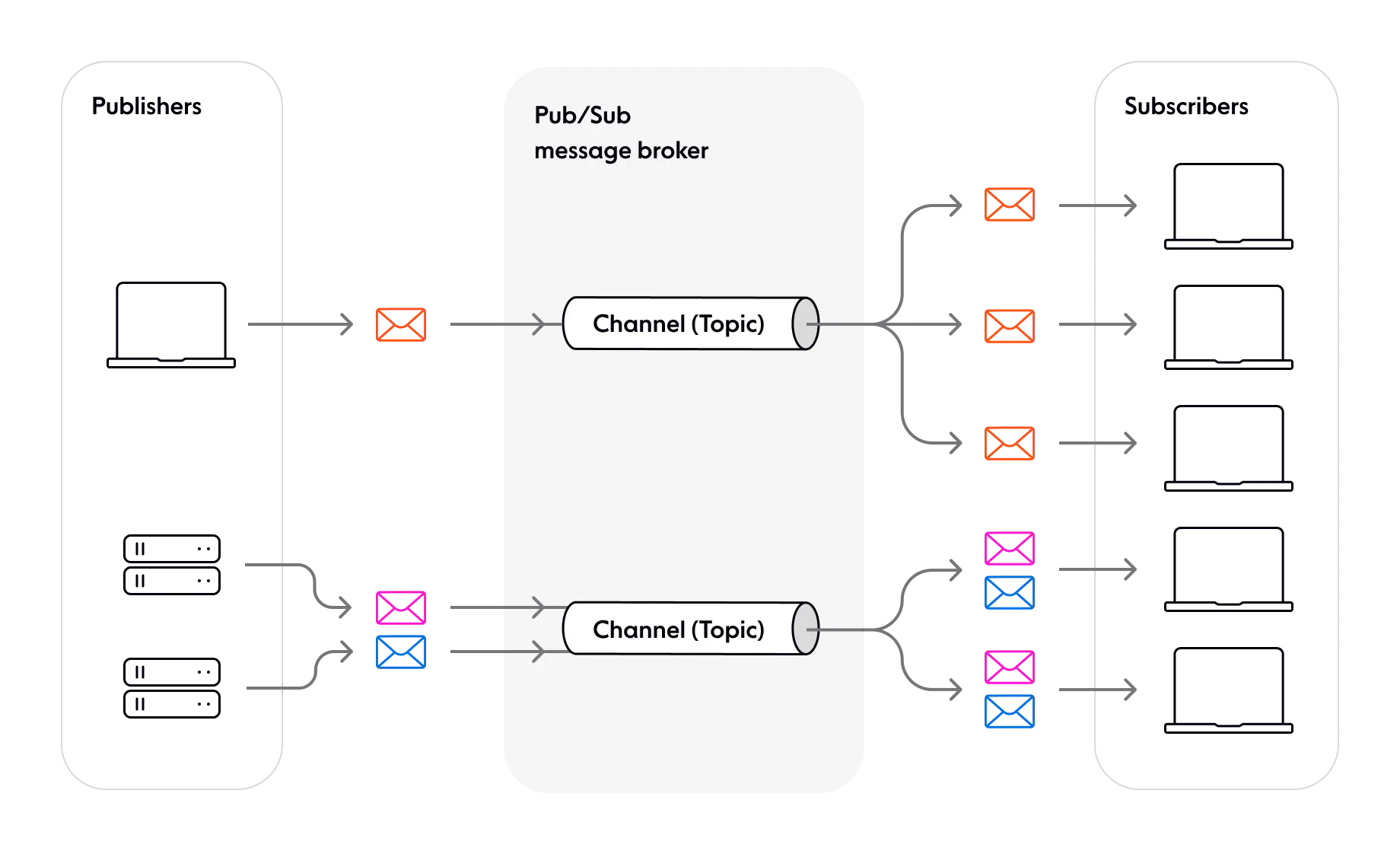
Il publish/subscribe, che viene spesso abbreviato in pub/sub, è un pattern usato per permettere una comunicazione asincrona tra componenti diversi di una rete, senza che questi siano effettivamente legati tra loro, implementando un sistema di update o notifica per la trasmissione di nuovi dati.

Infatti questo pattern è ampiamente utilizzato nelle architetture Event-Driven, ovvero quei sistemi dove la comunicazione tra più elementi distinti avviene tramite invio di eventi e notifiche.

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

Ci sono particolari caratteristiche che portano il modello publisher/subscribe ad essere un’ottima scelta architetturale quando si cerca un sistema di comunicazione con più componenti differenti. Prima di tutte il “decoupling” tra publisher e subscriber, permettendo di avere elementi eterogenei in comunicazione tra loro senza che questi siano a conoscenza degli altri. Altra caratteristica fondamentale sta nell’asincronicità degli eventi, per cui i publisher possono scrivere messaggi in qualsiasi momento non dovendosi preoccupare di altro, mentre i subscribers analogamente li andranno a leggere in modo del tutto asincrono; questo aumenta ancora di più il decoupling, in quanto ogni elemento del sistema, può avere i suoi tempi di lettura/scrittura dei messaggi, senza che vi siano problemi di sincronismo. Infine una caratteristica che differenzia questo modello da altri, come ad esempio il “request/response” o il “polling”, sta nella gestione dei messaggi, per cui il broker si occupa di indirizzare i messaggi a tutti i subscriber, e non con relazione 1 a 1 tra chi invia e chi riceve; ogni canale può essere acceduto da più subscribers contemporaneamente e tutti leggeranno lo stesso messaggio

#### Differenze con altri pattern

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.

Nell’progetto realizzato, ho deciso di non adottare l’observer pattern perché la relazione 1 a N su determinati topic sarebbe potuta diventare una limitazione pensando in ottica di scalabilità, magari avendo più thread (o più microservizi) per scrivere messaggi sugli stessi canali.

Altro pattern ampiamente usato per la comunicazione è il Request/response. Siamo abituati a vederlo e usarlo tutti i giorni perché è il meccanismo principale usato per comunicare con i servizi internet. Il funzionamento è semplice: un client fa una richiesta ad un determinato servizio e questo da una risposta in ritorno.

Come principio di funzionamento, si allontana dal pubsub, in quanto vi è una relazione 1 a 1 tra client e server; inoltre è un protocollo (generalmente) sincrono, infatti dopo che il client ha inoltrato la richiesta HTTP, tiene la comunicazione aperta finché non arriva una risposta (o scade un timeout). I vantaggi maggiori sono la semplicità di implementazione e la sicurezza di ricezione corretta del messaggio.

Non ho usato questo pattern perché non prevede un sistema di notifica, che mi avrebbe costretto all’utilizzo di un polling continuo che sarebbe uno spreco di risorse.

## 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.

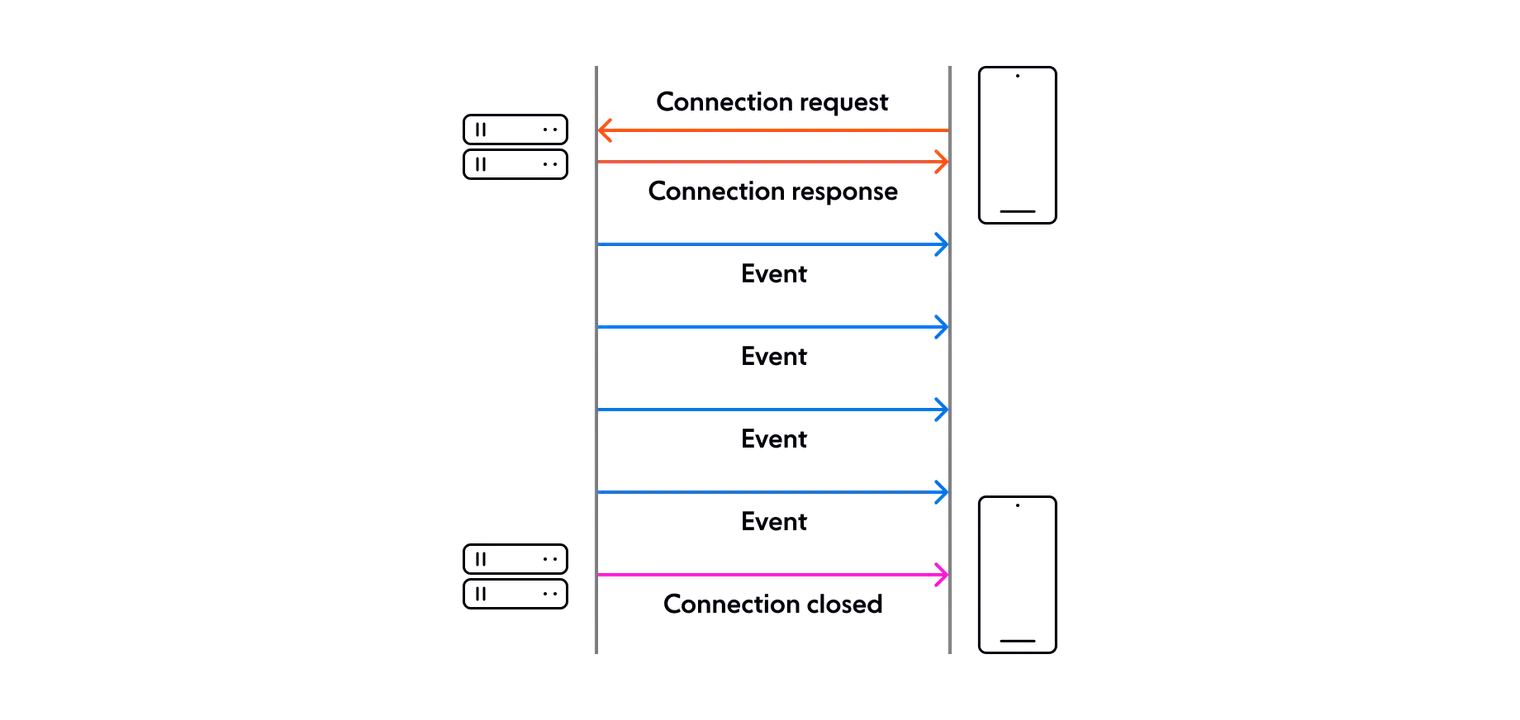
Usando semplicemente il protocollo HTTP, l’unico modo per ricevere continuamente degli update, sarebbe usare il polling, andando ad interrogare il server a ripetizione dopo un certo intervallo di tempo. Questo come già detto in precedenza, porterebbe inevitabilmente ad uno spreco di risorse e con potenziali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

Per ovviare a questo problema, sono stati creati protocolli che permettono di implementare il pattern publish/subscribe 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