* 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.

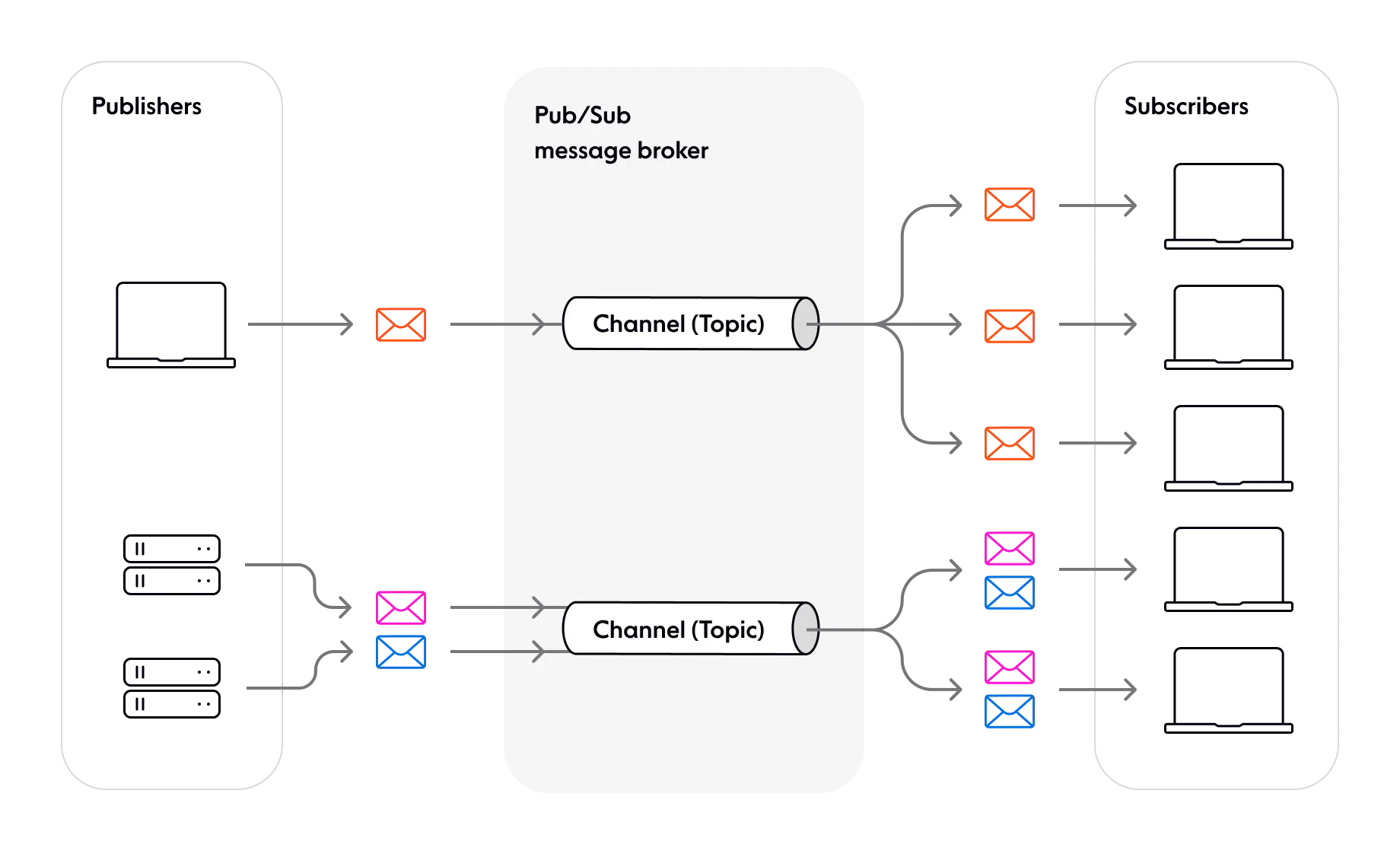
**Pattern Publish/Subscribe**

Il publish/subscribe, che viene spesso abbreviato in pub/sub, è un pattern architetturale usato per permettere una comunicazione asincrona tra componenti diversi di una rete, senza che questi siano effettivamente legati tra loro.

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

Con Server-Sent Events (abbreviato in SSE) si intende un meccanismo di comunicazione tra backend e frontend, con cui il server invia update automatici al cliente via HTTP e viene usato per mandare continui update asincroni dopo una sola richiesta.  
I SSE sono di particolare interesse per questo elaborato perché di fatto rientrano tra i modelli publish/subscribe, in quanto una volta avviata la comunicazione, il canale tra client e server rimarrà aperto e, quando ci saranno dei messaggi, il server si occuperà di inviarli automaticamente.

Le alternative più in uso attualmente sono:

- polling continuo del server, che è sconsigliato perché comporta uno spreco di risorse e di chiamate HTTP potenzialmente inutili

- WebSocket

# 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