Benchmarking di implementazioni di MQTT-Brokers open-source

Pasquale Caramante
Department of Computer Science
University of Salerno
Italy

***Abstract*—Negli ultimi decenni l'Internet of Things (IoT) si è affermato come un campo di ricerca in rapida crescita, i cui risvolti offrono un enorme potenziale per migliorare e ottimizzare processi che spaziano dalla sfera quotidiana fino all'ambito industriale e agricolo. Questo paradigma consente di connettere rapidamente dispositivi intelligenti, sensori e oggetti di uso quotidiano distribuiti geograficamente e di monitorarli da remoto, aumentando la loro utilità e produttività. Una delle principali sfide dell'Internet of Things (IoT) rispetto all'Internet classico, è rappresentata proprio dall'utilizzo di dispositivi con risorse computazionali limitate, che spesso operano anche in contesti con connettività di rete limitata.**

Per questo motivo diventa cruciale lo sviluppo e l'utilizzo di protocolli di rete adatti ad operare in tali contesti come MQTT (Message Queuing Telemetry Transport), un protocollo basato su un modello di comunicazione publish/subscribe, che permette di distribuire messaggi in maniera efficiente tra dispositivi e applicazioni anche in presenza di reti instabili o limitate. Questo lavoro propone quindi un confronto sperimentale tra diversi broker di messaggi MQTT valutandone le prestazioni in termini di throughput, latenza nella consegna dei messaggi, consumo di risorse CPU e RAM.

I. Introduzione

Motivazioni. Negli ultimi anni, la crescente diffusione di dispositivi IoT e applicazioni distribuite ha reso fondamentale l'adozione di protocolli di comunicazione leggeri ed efficienti. Il protocollo Message Queuing Telemetry Transport (MQTT) (Message Queuing Telemetry Transport) si è affermato come uno degli standard più utilizzati in scenari di pubblicazione/-sottoscrizione grazie alla sua semplicità, al basso overhead e al supporto a connessioni persistenti [1], [2]. MQTT è particolarmente adatto per ambienti con risorse limitate e reti instabili, come nel caso di dispositivi embedded e sistemi edge[2].

Il protocollo MQTT trova applicazione in numerosi ambiti, tra cui smart home, monitoraggio industriale, automazione dei veicoli, sensor networks e telemetria cloud, dove requisiti come bassa latenza, elevata affidabilità e ottimizzazione delle risorse sono fondamentali. In questi contesti, la scelta dell'implementazione del broker MQTT diventa critica, poiché influisce direttamente sulle prestazioni complessive del sistema e sulla scalabilità in presenza di numerosi client [3].

Nonostante la disponibilità di numerose implementazioni open source e commerciali, le differenze di performance, gestione delle risorse e scalabilità tra i vari broker non sono sempre documentate in maniera sistematica, rendendo complessa la selezione del broker più adatto a uno scenario specifico. Per colmare questa lacuna, questo lavoro propone un benchmark comparativo di diverse implementazioni di broker, includendo sia implementazioni più affermate e stabili, come **NanoMQ** [5], **EMQX** [6], **Mosquitto** [8] sia progetti più sperimentali tra cui **Rumqttd** [9], **HMQ** [10] e **RMQTT** [11].

Il benchmark è progettato per valutare due aspetti principali:

1) **Performance**: misurata in termini di throughput (**msg/sec**) e latenza (**ms**) nella pubblicazione dei messaggi .
2) **Scalabilità**: analizzata attraverso il consumo di risorse di sistema, in particolare **CPU** e **memoria**.

L'obiettivo è fornire un resoconto comparativo di queste diverse soluzioni, evidenziando punti di forza e limiti di ciascun broker in diversi scenari tipici di messaging Pub/Sub.

I test effettuati riguardano scenari di Connessione, Fan-Out (rapporto 1:N fra publisher e subscribers), Fan-In (N:1), e Point to Point (N:N). Le metriche di performance (throughput e latenza) sono state raccolte utilizzando il tool di benchmarking **emqtt-bench** [15]. Sia i broker che il tool di benchmarking sono stati eseguiti tramite Docker container, una scelta che ci permette di velocizzare la fase di installazione e configurazione dei vari broker, senza incorrere nell'overhead di una macchina virtuale tradizionale. Inoltre, l'utilizzo di Docker permette di raccogliere metriche riguardo l'utilizzo di CPU (%) e di memoria RAM (MB) tramite il comando `docker stats`.

Dopo una breve presentazione di alcuni lavori rilevanti in letteratura a conclusione dell'introduzione, la sezione 2 introdurrà i concetti fondamentali di funzionamento e architetturali di MQTT. La sezione 3 presenta la metodologia adottata nel dettaglio, per poi passare alla presentazione e discussione dei risultati nella sezione 4.

Related work. Il lavoro di Mishra [12] analizza le prestazioni di Mosquitto, HiveMQ, ActiveMQ, Bevywise MQTT [13], VerneMQ [14] ed EMQX, concentrandosi sul throughput di sottoscrizione dei client e misurando il tempo impiegato dal broker per consegnare messaggi ai client sottoscritti, utilizzando strumenti di stress test come mqtt-stresser e mqtt-bench. I risultati hanno evidenziato che, in condizioni di carico elevato, ActiveMQ ha mostrato la migliore capacità di elaborazione dei messaggi grazie alla sua implementazione multi-threaded, superando gli altri broker in termini di throughput. Tuttavia, Mosquitto ha ottenuto buone prestazioni

anche in scenari di stress, risultando una scelta valida per ambienti con risorse limitate.

Saha [16] si concentra invece sull'analisi delle prestazioni delle librerie client MQTT, piuttosto che sui broker. Questo lavoro propone un framework di valutazione delle prestazioni per le librerie client MQTT utilizzate in applicazioni IoT nel settore manifatturiero. L'analisi si concentra su diversi aspetti delle librerie client, come la latenza, la gestione delle connessioni e l'efficienza energetica, al fine di ottimizzare le prestazioni complessive dei sistemi IoT.

Lo studio di Longo et al. [17] propone BORDER, un framework di benchmarking per broker MQTT distribuiti, che consente la creazione di topologie di cluster flessibili e la valutazione delle prestazioni in scenari di carico elevato. Il framework è progettato per ambienti distribuiti, utilizzando container Docker e componenti di rete emulati per simulare architetture realistiche. In contrasto, il nostro lavoro si concentra sull'analisi delle prestazioni di broker MQTT in scenari con deployment su singolo nodo

II. Background

Il protocollo MQTT (Message Queuing Telemetry Transport) è un protocollo di messaggistica leggero basato sul paradigma *publish/subscribe*, progettato per scenari con vincoli di risorse, come i dispositivi IoT. MQTT consente la comunicazione tra dispositivi (client) e un broker centrale che gestisce la distribuzione dei messaggi ai destinatari sottoscritti a specifici topic.

A. Architettura

A livello protocollare, MQTT si basa su un modello client-broker. I client si connettono al broker tramite TCP/IP, spesso utilizzando TLS/SSL per garantire la sicurezza della connessione. La comunicazione è strutturata in messaggi definiti dal protocollo, ciascuno con un *fixed header* e, opzionalmente, un *variable header* e un *payload*, secondo lo standard MQTT 3.1.1 o 5.0.

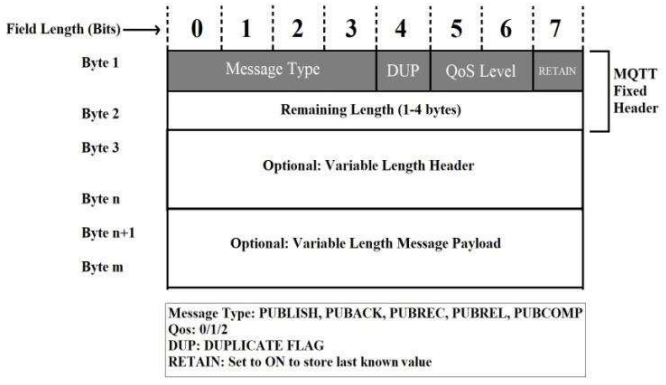


Fig. 1: Formato dei messaggi MQTT: fixed header, variable header e payload.

I principali messaggi del protocollo includono:

- **CONNECT:** inizializza la connessione tra client e broker, stabilendo eventuali parametri come QoS, clean session e keep-alive.
- **PUBLISH:** invia un messaggio su un topic specifico.
- **SUBSCRIBE:** richiede la ricezione di messaggi da uno o più topic.
- **UNSUBSCRIBE:** interrompe la ricezione dei messaggi da uno o più topic.
- **PINGREQ / PINGRESP:** mantiene attiva la connessione durante periodi di inattività.
- **DISCONNECT:** termina la connessione tra client e broker.

Il ciclo base di comunicazione di MQTT prevede:

1) Connessione del client al broker tramite messaggio CONNECT.
2) Pubblicazione di messaggi su topic tramite PUBLISH.
3) Inoltro dei messaggi dal broker a tutti i client sottoscritti al topic.
4) Gestione delle conferme in base al livello di Quality of Service (QoS):
   - QoS 0: consegna al massimo una volta, senza conferma.
   - QoS 1: consegna almeno una volta, con acknowledgment.
   - QoS 2: consegna esattamente una volta, con handshake completo a quattro fasi.
5) Manutenzione della connessione tramite PINGREQ/PINGRESP e gestione della disconnessione tramite DISCONNECT.

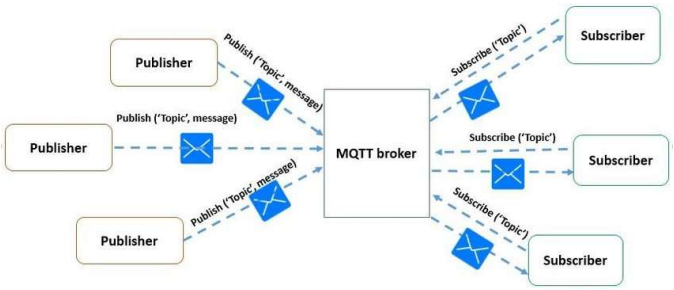


Fig. 2: Architettura publish/subscribe di MQTT: client pubblicano messaggi su topic gestiti dal broker, che li inoltra ai client sottoscritti.

III. Metodologia

In questa sezione andremo a presentare gli strumenti, la metodologia e il setup adottati per la procedura di benchmarking.

Benchmarking Tool. Per la generazione del workload è stato utilizzato lo strumento ***emqtt-bench***, un tool di benchmarking MQTT open source che consente di simulare un numero qualsiasi di client MQTT concorrenti, permettendo così di testare la capacità di un broker di gestire flussi di messaggi in maniera controllata.

Broker	Linguaggio	Multithread	Versioni MQTT supportate	QoS supportate	Transport Layer
Mosquitto	C	No	MQTT 3.1, 3.1.1, 5.0	0, 1, 2	TCP, TLS, WebSocket
EMQX	Erlang/OTP	Sì	MQTT 3.1, 3.1.1, 5.0 (e MQTT-SN)	0,1,2	TCP, TLS, WebSocket, MQTT-SN, QUIC
NanoMQ	C (basato su NNG)	Sì	MQTT 3.1.1, 5.0	0,1,2	TCP, TLS, WebSocket, QUIC
rumqttd	Rust	Sì	MQTT 3.1.1, 5.0	0,1,2	TCP, TLS, WebSocket
RMQTT	Rust	Sì	MQTT 3.1.1, 5.0	0,1,2	TCP, TLS

TABLE I: Confronto tra broker MQTT, includendo i protocolli di trasporto supportati

emqtt-bench offre la possibilità di configurare vari parametri, quali il numero di client, la frequenza e la dimensione dei messaggi, il livello QoS e gli intervalli di pubblicazione, rendendo possibile la riproduzione di scenari realistici.

Per generare il carico sul broker MQTT è stato utilizzato il comando `emqtt-bench pub`, che permette di simulare la pubblicazione di messaggi da parte di un numero configurabile di client. Un esempio tipico di esecuzione è riportato di seguito:

```
emqtt-bench pub -c 100 -t t/1 -s 256 -q 1 -I 10
```

Listing 1: Esempio di esecuzione di `emqtt-bench` in modalità publisher

Dove:
`-c 100` specifica il numero di publishers (in questo caso 100);
`-t t/%i` indica il topic sul quale i messaggi vengono pubblicati. Il placeholder %i indica un numero intero sequenziale che identifica univocamente il client;
`-s 256` definisce la dimensione del payload del messaggio in byte (qui 256 byte);
`-q 1` imposta il livello di QoS (Quality of Service) su 1, garantendo che ciascun messaggio sia recapitato almeno una volta;
`-I 10` stabilisce l'intervallo in millisecondi tra due pubblicazioni consecutive (per ciascun client).

Analogamente, il comando `emqtt-bench sub` permette di lanciare un numero arbitrario di client subscriber, specificando la sottoscrizione a determinati topic.

```
emqtt-bench sub -c 100 -t t/%i -q 1
```

Listing 2: Esempio di esecuzione di `emqtt-bench` in modalità subscriber

Dove:
`-c 100` specifica il numero di subscribers;
`-t t/%i` indica il topic di sottoscrizione. In questo esempio, ciascun subscriber si mette in ascolto su un topic diverso tramite il placeholder %i ;
`-q 1` imposta il livello di QoS (Quality of Service) su 1, garantendo che ciascun messaggio sia recapitato almeno una volta;

Una volta lanciati i publisher e i subscriber, possiamo controllare lo **stdout** di `emqtt-bench sub` per recuperare il throughput medio calcolato sull'ultimo secondo di esecuzione:

```
recv(28006): total=2102563, rate=99725(msg/sec)
```

Listing 3: Output del comando `emqtt-bench sub`

Inoltre, *emqtt-bench* fornisce statistiche dettagliate sulla **latenza** dei messaggi, calcolata come la differenza dei timestamp di invio e ricezione dei messaggi.

Ambiente di test. I test sono stati condotti in un ambiente locale, utilizzando una macchina desktop con Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz, 6 core / 12 threads e un laptop con Intel(R) Core(TM) i7-3610QM CPU @ 2.3 GHz, 4 core / 8 threads per eseguire i broker MQTT. La tabella II riassume le specifiche delle due macchine.

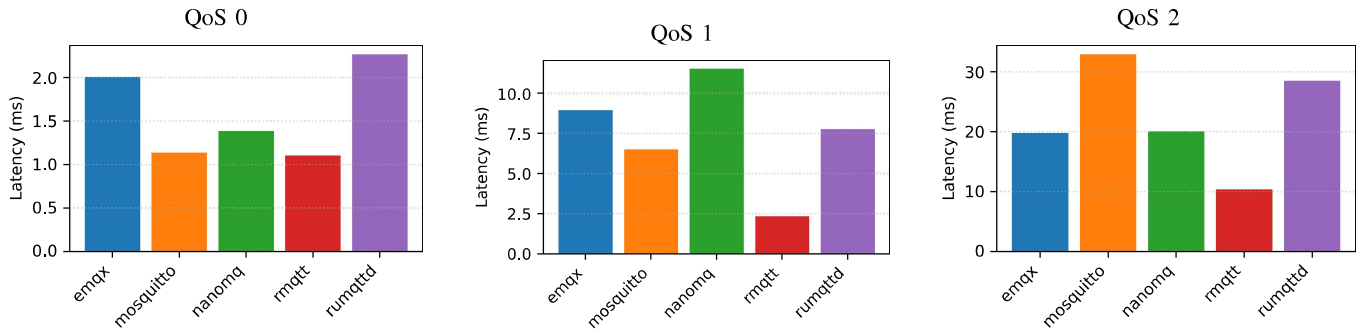
HW/SW Details	Client	MQTT Broker
CPU	Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz, 6 core / 12 threads	Intel(R) Core(TM) i7-3610QM CPU @ 2.3 GHz, 4 core / 8 threads
Memory	16 GB DDR4	16 GB DDR3-SDRAM (1600 MHz, 4x4 GB)
Storage	1.82 TB HDD WDC WD20EZRZ-00Z5HB0 + 238 GB SSD ADATA SX6000NP	750 GB HDD
Network	Intel(R) Ethernet Connection (2) I219-V, 1 Gbit/s	Gigabit Ethernet LAN 10/100/1000 Mbit/s
OS, Kernel	Ubuntu 24.04 LTS, 64 bit	Ubuntu 24.04 LTS, 64 bit

TABLE II: Configurazione hardware e software delle macchine usate per il benchmark

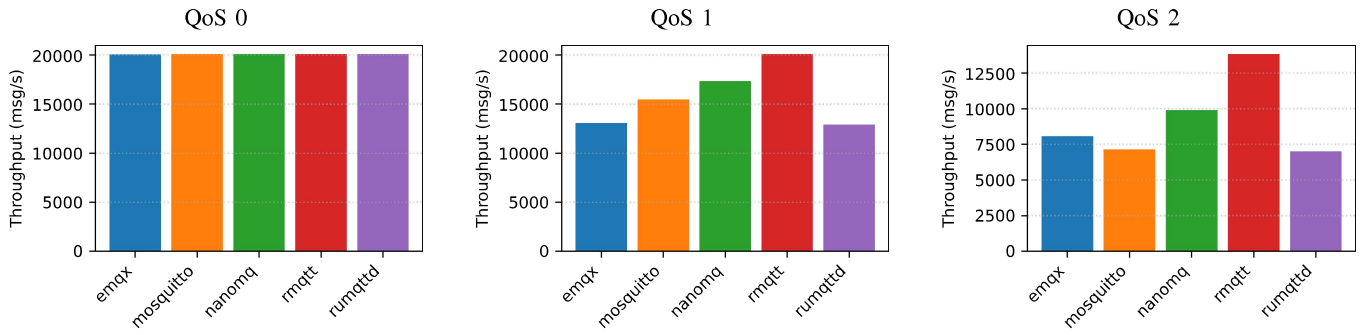
Ciascuno dei broker MQTT valutati è stato eseguito tramite Docker sul laptop, mentre la macchina desktop è stata utilizzata per la generazione dei workload e la raccolta delle metriche. La scelta di utilizzare la macchina desktop (con caratteristiche superiori) per simulare i client ha lo scopo di eliminare il bottleneck rappresentato dalla capacità di elaborazione dei client, influenzando negativamente le prestazioni dei broker e i risultati finali.

La lista di broker MQTT valutati in questo lavoro e le loro caratteristiche sono riassunte nella tabella I e comprende **EMQX** [6] (Erlang), **NanoMQ** [5] (C) , **Mosquitto** [8] (C) , **Rumqttd** [9] (Rust) e **RMQTT** [11] (Rust).
Tutti i broker sono stati valutati con le configurazioni di default, ad eccezione di due parametri:

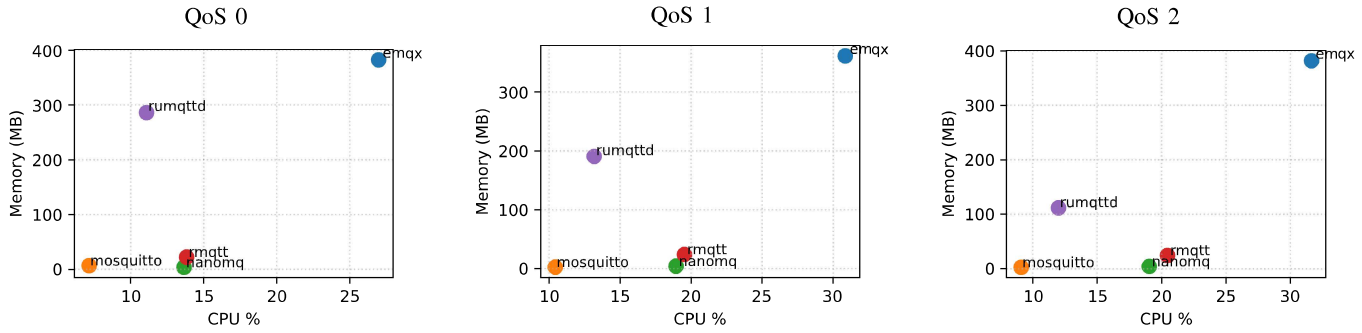
- **Message Queue Length**: questo parametro determina la dimensione massima del buffer contenente i messaggi QoS 1 e QoS 2 in attesa di riscontro, ossia il numero di messaggi QoS 1 e QoS 2 che il broker può inoltrare



(a) Latenza media dei broker per ciascun livello di QoS



(b) Throughput medio dei broker per ciascun livello di QoS



(c) Utilizzo di CPU e memoria dei broker per ciascun livello di QoS

Fig. 3: Performance dei broker MQTT su tre livelli di QoS, con latenze medie (a), throughput (b) e utilizzo di CPU/memoria (c).

contemporaneamente. In tutti casi è stato impostato il massimo valore possibile di 65535;

- **Rate-Limiting Off**: il limite sul rate di messaggi pubblicati al secondo è stato disattivato per far operare i broker al massimo della capacità;

IV. Risultati

A. *Performance con diversi livelli di QoS*

Il primo test valuta le performance dei broker con diverse impostazioni di QoS (0, 1, 2) in un un'architettura Point to Point standard con 100 publisher che inviano messaggi a 100 subscriber simmetrici tramite topic dedicati. Dopo aver stabilito la connessione al broker, ciascun publisher invia 20 messaggi al secondo, per un workload totale W di

$$20msg/sec * 100 = 20000msg/sec$$

La dimensione del payload dei messaggi è fissata a 64 byte.

Per ciascun livello di QoS eseguiamo un test della durata di 5 minuti, raccogliendo le medie della latenza dei messaggi, il throughput (msg/sec) e l'utilizzo di CPU e memoria (MB). Il test è stato ripetuto per 5 volte riportando per ciascun broker i risultati migliori rappresentati nei grafici in figura 4.

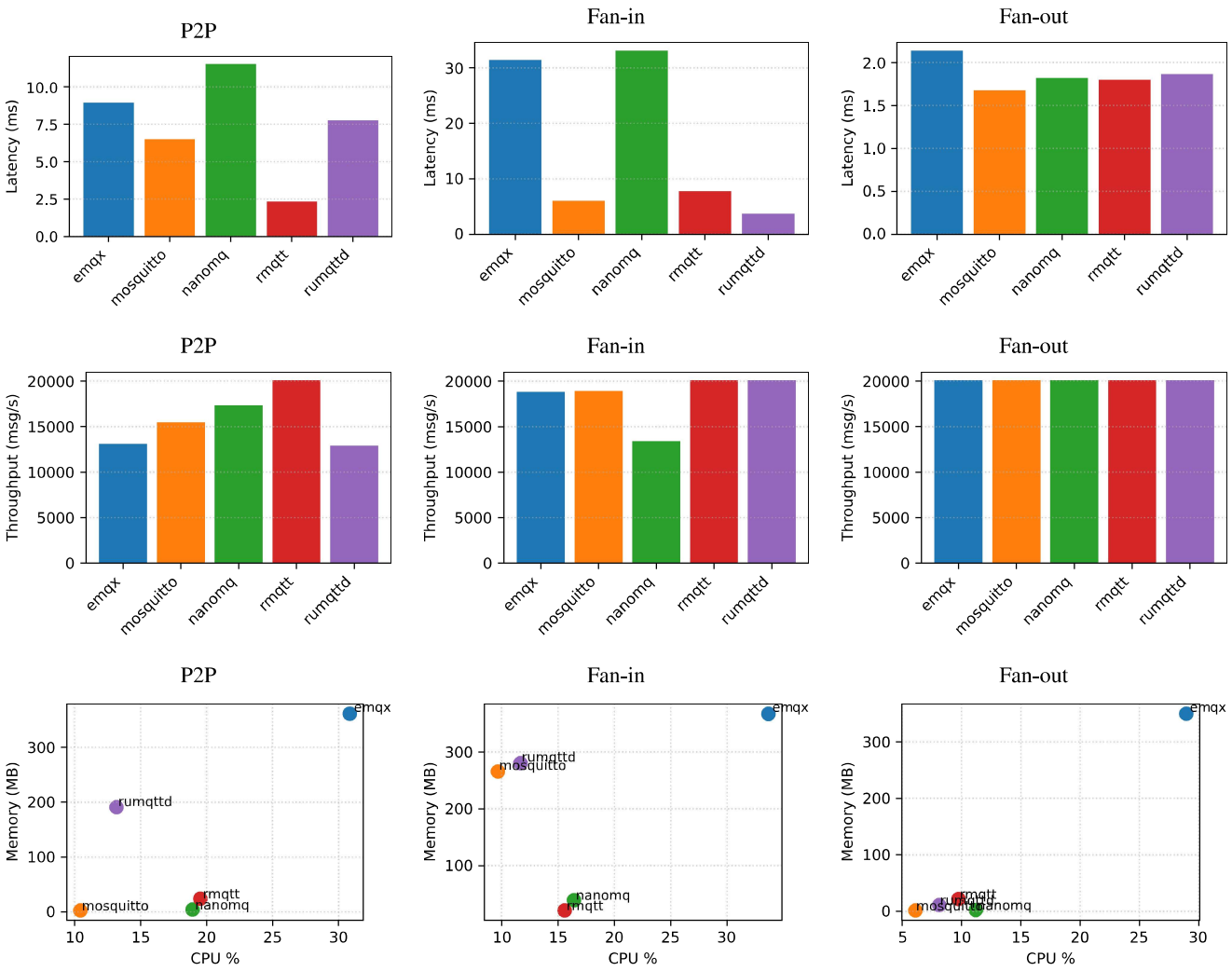


Fig. 4: Confronto tra P2P, Fan-in e Fan-out con QoS 1: Latency, Throughput e CPU/Memory.

Latenza. I risultati ottenuti mostrano differenze significative tra i broker al crescere del livello di QoS. Con `QoS 0`, tutti i broker mantengono latenze medie molto basse (circa 1–2 ms per **Mosquitto**, **NanoMQ** e **RMQTT**, leggermente superiori per **Rumqttd** ed **EMQX**). Con `QoS 1`, la latenza aumenta in maniera più marcata per **EMQX** (circa 9 ms) e **NanoMQ** (11 ms), mentre **RMQTT** e **Mosquitto** restano rispettivamente intorno a 2 ms e 6 ms. L'utilizzo di `QoS 2`, impatta significamente le performance di **Mosquitto** e **Rumqttd**, con latenze che superano i 28–32 ms di media, mentre **RMQTT** e **NanoMQ** si assestano tra 10 e 20 ms. In tutti i casi, **RMQTT** risulta il broker più efficiente, mantenendo la latenza media inferiore rispetto a tutti gli altri.

Throughput. Per `QoS 0`, tutti i broker riescono a gestire pienamente il carico di 20000 msg/sec, senza cali significativi. Passando a `QoS 1`, si osserva un leggero decremento per alcuni broker: **EMQX** scende a 13k msg/s, **Mosquitto** a 15k msg/s, **NanoMQ** a 17k msg/s, mentre **RMQTT** rimane vicino al massimo teorico (20k msg/s) e **Rumqttd** scende a 12.8k msg/s.

Con `QoS 2`, il throughput cala ulteriormente, in modo più marcato per broker come **EMQX** (8k msg/s), **Mosquitto** (7k msg/s) e **Rumqttd** (7k msg/s). **RMQTT** e **NanoMQ** gestiscono ancora un carico significativo (13.7k e 9.8k msg/s rispettivamente).

Questi dati indicano che l'aumento del livello di QoS, che garantisce consegne più affidabili e gestione dei messaggi duplicati, introduce overhead che riduce il throughput, ma broker come **RMQTT** e **NanoMQ** riescono a mitigare parzialmente l'impatto.

Consumo di CPU e memoria. Le differenze più marcate emergono sul fronte del consumo delle risorse. **Mosquitto** e **NanoMQ** risultano estremamente leggeri in termini di memoria (pochi MB) e con un utilizzo CPU variabile ma contenuto. **RMQTT** si colloca a metà strada, con consumi moderati sia in CPU che memoria. Al contrario, **EMQX** e soprattutto **Rumqttd** presentano un footprint molto più elevato: EMQX con oltre 350 MB di memoria e un utilizzo CPU tra il 27% e

il 33%, Rumqttd con picchi superiori a 280 MB di memoria e valori CPU attorno al 12%. Questi risultati evidenziano una chiara distinzione tra broker minimali, ottimizzati per ambienti a risorse limitate, e soluzioni più pesanti ma con feature avanzate, come EMQX e Rumqttd.

B. Performance su diverse architetture Pub/Sub

Nel secondo test, le prestazioni dei broker sono state valutate e confrontate in diverse architetture Pub/Sub tipiche degli scenari IoT: Point to Point, Fan-In, Fan-out.
L'architettura **Point to Point** è totalmente analoga all'architettura utilizzata nel test sulla QoS, con 100 publisher e 100 subscriber che comunicano simmetricamente.
Nell'architettura **Fan-In** andiamo a simulare 100 publisher che pubblicano messaggi su 100 topic, e un unico subscriber che si iscrive a tutti i topic, testando le capacità del broker in uno scenario di aggregazione dei messaggi.
L'architettura **Fan-Out** è diametralmente opposta, con 1 publisher, 1 topic, e 100 subscriber iscritti all'unico topic, simulando l'invio di messaggi broadcast.

In tutti i test è stato utilizzata come QoS 1 e un publishing rate di 20 msg/sec per ciascun client, con un payload fissato di 64 byte.

Latenza. In `fan-out` la latenza resta molto bassa (1–2 ms medi), mentre in `fan-in` si osservano valori molto più elevati per broker come **EMQX** e **NanoMQ** (oltre 30 ms). Lo scenario `p2p` mostra invece un livello intermedio, con latenze variabili tra 2 ms (**RMQTT**) e 11 ms (**NanoMQ**).

Throughput. Nel caso `fan-out`, tutti i broker raggiungono o si avvicinano al limite massimo (20k msg/s). In `fan-in` le prestazioni restano elevate (18–20k msg/s) per la maggior parte dei broker, ad eccezione di **NanoMQ** che scende a circa 13k msg/s. In `p2p`, invece, emergono differenze più marcate: **RMQTT** mantiene throughput pieno, mentre **EMQX** e **Rumqttd** si fermano intorno a 13k msg/s.

CPU e Memoria. I risultati ottenuti confermano quanto riportato in precedenza per il test di QoS, con **EMQX** e **Rumqttd** che mostrano un utilizzo di memoria più alto rispetto agli altri broker, mentre **Mosquitto** e **NanoMQ** hanno footprint molto ridotti. In termini di utilizzo di CPU, **EMQX** e **NanoMQ** mostrano valori più alti in scenari fan-in, mentre **Mosquitto** e **RMQTT** restano più stabili.

V. Conclusioni

Dai test condotti emerge chiaramente come le prestazioni dei broker MQTT varino in modo significativo in funzione del livello di QoS e dell'architettura Pub/Sub adottata. In generale, i broker più leggeri come **Mosquitto** e **NanoMQ** si distinguono per il ridotto consumo di memoria e CPU, risultando particolarmente adatti a scenari con risorse limitate, nonostante cali di prestazione evidenti con `QoS 2` e in architetture fan-in. Al contrario, soluzioni più complete come **EMQX** e **Rumqttd** offrono feature più complesse ma al prezzo di un footprint di risorse e latenze medie superiori. Tra tutti i broker valutati, **RMQTT** risulta essere la soluzione più equilibrata, mantenendo latenze contenute, throughput vicino al massimo teorico anche con QoS elevati e consumi moderati di risorse. Complessivamente, i risultati confermano che la scelta del broker dipende fortemente dallo scenario applicativo: per ambienti IoT a basso consumo sono preferibili soluzioni leggere

REFERENCES

[1] Wikipedia contributors, *MQTT*, 2023. https://it.wikipedia.org/wiki/MQTT. Accessed: 2025-09-11.

[2] Jasenka Dizdarevic, Marc Michalke, Admela Jukan, *Engineering and Experimentally Benchmarking Open Source MQTT Broker Implementations*, 2023. https://arxiv.org/abs/2305.13893. Accessed: 2025-09-11.

[3] HiveMQ, *Overcoming MQTT Cluster Sharding Challenges for IoT Scalability*, 2023. https://www.hivemq.com/blog/overcoming-mqtt-cluster-sharding-challenges-iot-scalability/. Accessed: 2025-09-11.

[4] Apache ActiveMQ. https://activemq.apache.org/

[5] NanoMQ. https://nanomq.io/

[6] EMQX. https://www.emqx.com/en

[7] HiveMQ. https://www.hivemq.com/products/mqtt-broker/

[8] Eclipse Mosquitto. https://mosquitto.org/

[9] Rumqttd. https://rumqtt.bytebeam.io/docs/rumqttd/Introduction/

[10] HMQ (Go MQTT Broker). https://github.com/hmq-project/hmq

[11] RMQTT. https://github.com/rmqtt/rmqtt

[12] B. Mishra, *Performance Evaluation of MQTT Broker Servers*, 2018. https://www.researchgate.net/publication/326167412$_{P}erformance_{E}valuation_{o}f_{M}QTT_{B}roker_{S}ervers$

[13] Bevywise MQTT Broker. https://www.bevywise.com/mqtt-broker/

[14] VerneMQ. https://vernemq.com/

[15] emqtt-bench. https://github.com/emqx/emqtt-bench

[16] N. Saha, *Performance evaluation framework of MQTT client libraries for IoT applications in the manufacturing industry*, 2024. https://www.sciencedirect.com/science/article/pii/S221384632400213X

[17] E. Longo, A.E.C. Redondi, M. Cesana, P. Manzoni, *BORDER: a Benchmarking Framework for Distributed MQTT Brokers*, IEEE Internet of Things Journal, vol. 9, no. 17, pp. 17272–17285, Sept. 2022. https://re.public.polimi.it/retrieve/059afd8c-20fb-4752-adbe-96b514e16b8e/longo2022$_{b}order.pdf$