AstraMQ: Distributed MQTT Broker

Rohan Doshi
Department of Computer Engineering
Pune Institute Of Computer Technology
Pune, India
rohan.doshi02@gmail.com

Sanika Inamdar
Department of Computer Engineering
Pune Institute Of Computer Technology
Pune, India
sanika.inamdar.2002@gmail.com

Tanmay Karmarkar
Department of Computer Engineering
Pune Institute Of Computer Technology
Pune, India
tanmaykarmarkar49@gmail.com

Madhuri Wakode
Department of Computer Engineering
Pune Institute Of Computer Technology
Pune, India
mswakode@pict.edu

***Abstract*—Distributed MQTT Broker offers a scalable, fault-tolerant solution for efficient message brokering in IoT deployments. This architecture integrates a custom load balancer, MQTT broker nodes, and Redis Streams. The load balancer ensures a single point of entry and load distribution among broker nodes, enhancing system reliability. MQTT broker nodes handle standard operations and utilize Redis Streams for asynchronous message passing and state synchronization, ensuring decoupled communication and fan-out delivery. This system also explores alternative approaches using etcd with Raft for client state synchronization and gRPC with protocol buffers for message passing, as well as Kafka for message storage and inter-broker communication. AstraMQ addresses the limitations of existing distributed brokers by leveraging open-source components and the efficiency of Golang, providing a robust, cost-effective solution for large-scale IoT applications.**

***Index Terms*—MQTT, Redis, Redis Streams, Load-Balancer, gRPC, Kafka, Raft, etcd, Distributed Systems, High Availability**

I. Introduction

MQTT is a lightweight messaging protocol based on a publish-subscribe model. It is most commonly used in the Internet of Things (IoT) since it is efficient and requires minimal resources. It is an ideal protocol for connecting remote devices with minimal network bandwidth [5], [6].

Unlike traditional network communication between clients and servers directly, the publish-subscribe architecture decouples the sender and receiver of a message by using a central entity called the MQTT message broker. This broker handles the communication between multiple publishers (senders) and subscribers (receivers) of a message based on topics. Topics are keywords to which messages are published. A subscriber will receive all messages published on a particular topic by subscribing to that topic. Topics are arranged hierarchically.

The conventional MQTT Broker deployment typically consists of a single-node setup. This single node manages all the MQTT operations. Failure of this node results in total disruption of the communication. Due to the limitations of Network i/o calls of a single broker, it can only serve a limited number of clients at a time or has limited message throughput, resulting in limited performance.

In contrast, a distributed MQTT Broker network comprises multiple brokers that collaborate in synchronization. These interconnected brokers facilitate more scalable MQTT implementations. Distributed brokers offer several advantages, including enhanced performance, scalability, load distribution, and fault tolerance. Such capabilities are crucial for large-scale IoT deployments where reliability and performance are extremely crucial.

Current distributed MQTT brokers offer a range of functionalities but often come with some limitations that restrict their accessibility or are highly priced [15]. Our system incorporates open-source components like Mochi-MQTT broker and Redis. Our system is developed using Golang, which in turn addresses these shortcomings to provide a compelling alternative. Distributed brokers like HiveMQ use closed-source for their distributed features. HiveMQ written in Java, requires more system resources compared to Golang. EMQX and VerneMQ developed in Erlang, is difficult to setup, manage and configure [12], [16]. Golang programs typically require less memory and startup time compared to their Java or Erlang counterparts, making them ideal for these scenarios [1].

II. Architecture

Our architecture consists of a load-balancer, MQTT broker nodes, and Redis (Streams) at the abstract level. The load-balancer acts as a reverse proxy, that intercepts MQTT *Connect* requests from the clients and forwards them to different broker nodes in the system. The load balancer also acts as a single point of entry to our system, protecting the direct identities of the server nodes, ensuring that there is only one single IP address for clients to connect. A private network of load-balancer and broker nodes can be created to only expose the default MQTT port 1883. MQTT broker nodes are standard broker nodes that support all the default MQTT operations along with certain capabilities to send and receive data between the brokers. Redis a freely available open-source system, popularly used as a cache mechanism, is a distributed key-value store. Redis also provides us with the functionality of Redis Streams, which is like an in-memory append-only log.

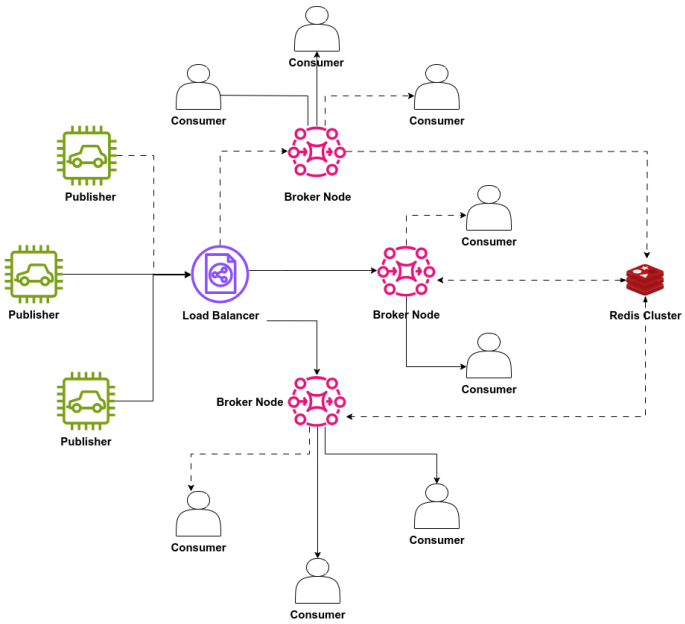

Fig. 1. System Architecture

We use Redis Streams to decouple the connections between the brokers, i.e. each broker doesn't need to be connected with every other broker in the cluster but rather has a single connection with the Redis Stream. Redis Streams also ensure a fan-out delivery to other brokers, meaning a guaranteed delivery. We use Redis Streams to forward MQTT *Publish* messages to other brokers who might have intended subscribers connected.

To better understand the architecture, consider a system consisting of 3 broker nodes. A "Client A" sends an MQTT *Connect* request to the system. The load-balancer receives the *Connect* packet and forwards it to one of the broker nodes (N1). Now, the broker node completes the TCP handshake and all communication occurs directly between the broker and the client without involving the load-balancer. Similarly "Client B" connects to node (N2), and "Client C" to node (N3). Consider that "Clients A, B, and C" are subscribed to a common topic. "When 'Client A' publishes a message on this topic, it will be handled by the Node N1. N1 will then insert this message into the Redis Stream. Nodes N2 and N3 will asynchronously consume this message from the Redis Stream and subsequently process it by sending it out to "B" and "C" respectively."

Components:

1) Load-balancer: The load-balancer acts as a single point of entry to the system and is responsible for 3 things. 1) To forward the client MQTT *Connect* message to any one of the brokers. 2) To discover broker nodes in the system by using the GOSSIP protocol (Node Discovery). 3) Health check and remove dead nodes from the system using the GOSSIP protocol again. We are using a custom load-balancer for this purpose but can be easily replaced by other load-balancing solutions. The load-balancer provides a single IP Address for clients to connect to, which in turn helps in hiding the IP Addresses of our broker nodes. The load-balancer uses a simple Round-Robin algorithm to distribute clients equally across each node. Once a load-balancer has forwarded the incoming MQTT Connect packet (TCP packet) from the client to the broker node, the load-balancer has no further interaction between the client and broker communication.
2) MQTT Broker: We are using a modified Mochi-MQTT broker [9] to support data exchange between other broker nodes. The modified MQTT Broker adds any Publish Packet that it receives from the client to the Redis Stream. To avoid a cyclic loop the broker does not add Publish packets that other Brokers have added. The broker node also listens for every message in the Redis Stream to process it and deliver messages to the intended subscriber connected to it.
3) Redis-Streams: They are used to decouple the connections between the broker nodes and also for a fan-out (guaranteed) message delivery to other broker nodes. As messages live in-memory, we are running a service to periodically clean this Redis Stream. A cron job runs every few seconds which checks the memory usage of the server where Redis Stream is deployed. Depending on the usage, the process prunes the Redis Stream, keeping only the top messages, and freeing up space for new messages to be stored.

It is important to note that we are using the Mochi-MQTT broker because of its exceptional raw performance. The Mochi broker, written in Golang, is simply unmatched in its efficiency and speed.

III. Node Discovery and Health Monitoring

We will now dive deep into the Node Discovery component of the load balancer. Our custom Load Balancer uses *Hashicorp Memberlist* [7] to discover new brokers in the system. The Memberlist is the backbone of many distributed systems as these systems are required to maintain the nodes present in a system. The Hashicorp Memberlist is based on the *SWIM* protocol [2]. SWIM protocol is a peer-to-peer membership protocol. The protocol takes care of health checks for the system and removes stopped nodes from the system.

The Memberlist improves in this area by implementing a process called *Lifeguard* because the node may be healthy but due to slow message processing or network latency, the protocol marks it to be down. The Lifeguard process works on the principle of local health which drastically reduces the number of false positives in the system.

The Load Balancer is configured to listen for incoming gossip requests on port 7946. The Load Balancer maintains a list of broker nodes connected to it in a simple Array data structure. Configuring the Memberlist depends on whether the network is a LAN or a WAN. There are various factors like *ProbeInterval, and SuspicionMult* that need to be configured based on the network for efficient performance. The *SuspicionMult* should be set by figuring out what the packet round trip

time looks like. Higher latency should have a higher multiplier. Setting this to very low-value marks that the node is down even though it is healthy. This results in a large number of false positives.

v1: The Load Balancer scans through the Memberlist. Every 5 seconds the Memberlist checks for new devices to be added to the list of connected brokers to the Load Balancer. On testing, we found that this was a bottleneck and hampering the performance of the load balancer.

v2: Harnessing the beauty of Golang we have implemented a channel-based design pattern, to add and remove nodes from the Memberlist to the list of the Load Balancer. There are 2 events of interest that the brokers emit to the Load Balancer *1) NodeJoin & 2) NodeLeave*.

When any broker emits a NodeJoin or NodeLeave event the Node is added or removed respectively from the broker's list.

IV. Detailed Design and Implementation

A. The Load Balancer

The main entry point in any distributed system is a Load Balancer or a Reverse Proxy. The same is the case with our system. When it comes to using an out-of-the-box Load Balancer, it is a complete black box. No one knows what actually goes on in the back. To have complete transparency and control on our system we decided to go with a custom Load Balancer.

We needed to make sure that the load balancer was not a bottleneck in our system. For this, we compared the throughput of the system with some reputed Load Balancers like HAProxy and NginX.

When a publisher or subscriber wants to enter our system only the connect packet is intercepted by the Load Balancer. This Load Balancer forwards this connect packet to the brokers in the system using the round-robin algorithm.

As the load balancer only intercepts the connect packet after which the connection is maintained by the broker the load balancer does not prove to be a bottleneck. Which claim was also proved when stress tested against HAProxy and NginX.

B. Redis

In our architecture, every message is stored and passed through Redis [8]. One may assume that Redis acts as a major bottleneck in our system, but this is not the case. Redis is a widely adapted distributed system and is a major contributor to modern architectural patterns. Let's see how Redis scales to handle the large message throughput.

1) Redis in-memory database: Redis is an in-memory database, which is persistent (using Snapshotting (RDB) / Append-only File (AOF)). To ensure fault tolerance, Redis allows us to create multiple replicas of the Master for both Redis Cluster and Redis Sentinel. However, it's important to note that Redis does not ensure high synchronization between the replicas. Let us understand how exactly Redis Replication works:

1) Client writes transfer to Replicas: Master sends a stream of client operations to the replica asynchronously. Replicas asynchronously acknowledge these operations to the master.
2) Partial Resynchronization: Due to any failure (network, hardware, system crash, etc) of replicas, the master only rolls out the missed operations during which the replica was not connected.
3) Initial Synchronization: When a new replica is added or partial resynchronization is impossible, the master creates a snapshot of its entire state and then transfers it to the replica. This can be a huge amount of data and is a very costly operation [18].

2) Redis Sentinel: Another thing to mention is that Redis Replication is used for High Availability / Data Safety and Scalability (read-only Replicas). Redis also supports Cascading Replicas: Replicas can be connected to other Replicas. Redis is primarily a multi-master architecture.

But what happens when one of the masters fails? The key to addressing this lies in Redis Sentinel. Redis Sentinel is a standalone process independent of the Redis database, ideally running on separate machines.

Redis Sentinel Use Cases:

1) Monitoring: Continuously checks the status of the Master and Replicas.
2) Notification: Alerts system administrators if the master or replicas are offline.
3) Automatic Failover: Promotes a Replica to Master in the event of a Master failure.
4) Configuration Provider: Stores the latest Master's address to provide to clients.

3) Automatic Failover System: For robustness, a minimum of three Sentinels operate simultaneously in the system, eliminating single points of failure. These Sentinels regularly assess the health of all nodes in the system. In the event of failure or other critical events, they notify clients through Pub/Sub Messages.

In the case of a Master failure, a Sentinel triggers the failover process. To execute a failover, a majority of Sentinels must reach a consensus, as determined by the Quorum number. This number represents the count of Sentinels required to agree on the master failover, preventing false failover scenarios. During the failover process, one of the Redis replicas is promoted to the new master. The remaining replicas are promptly informed of their new master nodes to ensure the continuity of write operations.

Using these approaches, we can understand that Redis is a highly available (HA) system and can scale well for a large number of operations, hence it is not a bottleneck.

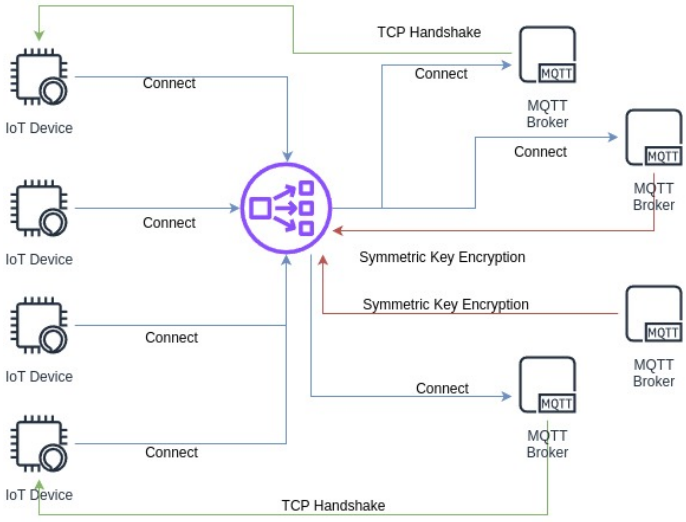

Fig. 2. Authentication Mechanisms

V. Security Considerations

A. Node Discovery in Memberlist

There are two main authentication mechanisms that we need to configure in order to completely secure the system. The first is to secure the broker load balancer authentication. Only the brokers having a shared secret key can join the Memberlist. For this purpose, we use an AES-256 standard Key. AES-256 provides a symmetric key encryption [19]. If and only if the broker has the correct key it can connect to the load balancer.

B. Client Broker Communication

The second mechanism is between the client and the broker. Only clients authenticated by us should be able to connect to the brokers. For this, we have configured the server and client to use TLS Certificates. These certificates are signed by a common Certificate Authority. The clients will be able to connect to the broker if and only if they are able to present the broker with valid certificates signed by the Common Certificate Authority. This ensures a secured channel for message transmission between the client and the broker. The client here can be anyone a publisher as well as the subscriber. The clients can connect only to a *"ssl://x.x.x.x:port"* and not a plain TCP connection. The broker will not accept the TCP connection in such cases.

VI. Explored Approaches

A. ETCD coupled with GRPC

1) etcd: etcd-io is a distributed key-value store used for shared configuration and service discovery in distributed systems. It provides a reliable way to store data across a cluster of machines. It gracefully handles leader elections during network partitions and can tolerate machine failure, even in the leader node. Being built on Raft, etcd ensures strong consistency. Raft is a protocol with which a cluster of nodes can maintain a replicated state machine. The state machine is kept in sync through the use of a replicated log. The state machine takes a Message as input. A message can either be a local timer update or a network message sent from a remote peer. The state machine's output is a 3-tuple []Messages, []LogEntries, NextState consisting of an array of Messages, log entries, and Raft state changes. For state machines with the same state, the same state machine input should always generate the same state machine output.

2) Raft Consensus Algorithm: Raft is a simpler and improved version of Paxos.Raft is based on a leader-follower algorithm. Any changes made by clients are forwarded to the Raft leader, which then replicates them to other nodes in the cluster. Raft takes care of leader election, which ensures fault tolerance in the system [3].

1) Raft Log Entries:
   Each node in the Raft cluster maintains a replicated log. Each log entry should represent a single client state change operation (e.g., connect, disconnect, subscribe, unsubscribe).
2) Raft Leader:
   In Raft, one node is elected as the leader.When a client state change occurs on the leader, it creates a new log entry and replicates it to a majority of nodes in the cluster.
3) Raft Replication: Raft ensures that the log entries are replicated to a majority of nodes before they are considered committed. Once a log entry is committed all nodes have the same client state.
4) Handling failures in Raft:
   a) Leader Failure:
      If the leader fails, a new leader is elected from the remaining nodes. The new leader continues to replicate client state changes and ensure consistency across the cluster.
   b) Node Failure:
      If a follower node fails, the leader continues to replicate log entries to other nodes, ensuring fault tolerance.

3) Configuring etcd: On system startup, the brokers will initially connect to the Load Balancer using the gossip protocol. The first broker that joins the system will start the etcd cluster. Any subsequent broker that joins the system, will join this pre-existing cluster.

There must be a reliable and continuous communication channel (stream)/connection between all the brokers in the cluster. Etcd internally uses gRPC to maintain this connection. gRPC, which stands for Google Remote Procedure Call [11].

Whenever a broker joins the cluster it needs to form a gRPC connection with each of the remaining broker nodes in the system. When it leaves the cluster all the connections must be gracefully handled and closed to mitigate any security threats to the system.

1) The Client State

a) Connection properties: It contains information like the will properties, which broker it is connected to in the system, QoS levels, etc.
b) Subscriptions: It contains the list of topics the client is subscribed to.
c) Client sessions: It contains information about the active client sessions.

2) When to sync the data?
The client state of the broker will be updated when the broker receives either a subscribe, unsubscribe, connect or disconnect packet. The state across the brokers should be synced when the broker receives any one of these packets.

3) Packet forwarding to other brokers
We have tackled the client state synchronisation problem with the above-mentioned approach. Now comes another challenge that we need to tackle. We have to find some way to transfer packets from one broker node to the other.
The question arises why do we need to do this? Let's say there are 2 subscribers subscribed to the same topic connected to 2 different brokers. Now some publisher 'X' publishes a message on this topic. It may happen that this publisher 'X' is connected to some other broker in the system. The broker to whom the publisher is connected needs to process this packet. There will be two cases the broker needs to handle:

a) There will be subscribers connected to the same broker, subscribed to the topic the publisher has published to.
b) There will be subscribers connected to some other broker in the cluster, subscribed to the topic the publisher has published to.

The first case is fairly simple to handle; the broker just sends out the message to the intended subscriber. The client state sync that we mentioned above comes into the picture here. The broker figures out which brokers have subscribers connected to them with the same topic. Using the same gRPC connection that Raft uses, the broker will forward the message to these brokers so that they can further send these messages to the intended subscribers.

4) One last thing left to do!
Now, there is only one thing left to do. How do we send these packets to the brokers? One might think as it is a gRPC connection we ought to use protocol buffers. He may not be completely wrong about this one. Here's where Cap'n Proto comes into the picture. Cap'n Proto is an insanely fast data interchange format and capability-based RPC system. Cap'n Proto is highly efficient due to the absence of encoding and decoding procedures. Its encoding method is well-suited for both data exchange and in-memory representation. This means that after constructing your structure, you can save the bytes directly to disk without any additional steps. As there are no encoding and decoding processes involved, Cap'n Proto is significantly faster than protocol buffers.

B. Kafka

Kafka is an open-source distributed event streaming platform developed by LinkedIn and later open-sourced as a part of the Apache project. It is designed to handle high-throughput, fault-tolerant, and scalable event streaming in real-time [4]. Kafka follows a distributed architecture model, consisting of the following core components:

1) Architecture :

- Broker: Kafka brokers are the fundamental building blocks of a Kafka cluster. They are responsible for storing and managing the streams of records. Each broker can be thought of as a single Kafka server instance. Brokers are deployed as a cluster to provide fault tolerance and scalability.
- Topic: Kafka organizes data into topics. A topic is a particular stream of records, similar to a database table or a message queue. Producers publish records to one or more topics, while consumers subscribe to topics to consume records.
- Partition: Each topic is divided into one or more partitions. Partitions allow Kafka to parallelize data and write and read across multiple brokers. Each partition is an ordered, immutable sequence of records.
- Replication: Kafka provides built-in replication of partitions across multiple brokers. Replication ensures fault tolerance and high availability. Each partition has one leader and one or more follower replicas. If a broker fails, one of the replicas can be promoted to leader to continue serving requests.
- Producer: Producers are applications that publish records to Kafka topics. They send records to one or more brokers, which then distribute the records to the appropriate partitions.
- Consumer: Consumers are applications that subscribe to Kafka topics to consume records. They read records from one or more partitions in the topics they subscribe to.
- Consumer Group: Consumers can be organized into consumer groups. Each consumer group consists of one or more consumers that collectively consume all the records in a topic. Kafka ensures that each record is consumed by only one consumer within a consumer group, enabling parallel processing of records.

2) How Kafka can be used in our architecture: In the current system, Redis Streams is used to decouple connections between MQTT brokers. Similarly, Kafka can be used to achieve this decoupling by acting as the intermediary message queue between brokers.

Each broker in the cluster acts as a producer and subscriber to the same broadcast topic for the Kafka cluster. Every publish packet is received by Kafka, consumed by all the brokers in the cluster and processed by only those brokers which has clients (consumers) connected to it for the topic of the published message.

Test	Broker	publish fastest	median	slowest	receive fastest	median	slowest
2 Clients 10000 Messages	AstraMQ	124,772	125,456	124,614	314,461	313,186	311,910
	Mosquitto v2.0.15	155,920	155,919	155,918	185,485	185,097	184,709
	EMQX v5.0.11	156,945	156,257	155,568	17,918	17,783	17,649
	Rumqtt v0.21.0	112,208	108,480	104,753	135,784	126,446	117,108
10 Clients 10000 Messages	AstraMQ	45,615	30,129	21,138	232,717	86,323	50,402
	Mosquitto v2.0.15	42,729	38,633	29,879	23,241	19,714	18,806
	EMQX v5.0.11	21,553	17,418	14,356	4,257	3,980	3,756
	Rumqtt v0.21.0	42,213	23,153	20,814	49,465	36,626	19,283
100 Clients 10000 Messages	AstraMQ	51,044	4,682	2,345	72,634	7,645	2,464
	Mosquitto v2.0.15	3,826	3,395	3,032	1,200	1,150	1,118
	EMQX v5.0.11	4,086	2,432	2,274	434	333	311
	Rumqtt v0.21.0	78,972	5,047	3,804	4,286	3,249	2,027

TABLE I
PERFORMANCE COMPARISON OF DIFFERENT BROKERS UNDER VARIOUS CONDITIONS.

Kafka achieves "at-least-once" delivery semantics by default. This means a published message is replicated across multiple brokers in the cluster and delivered to at least one subscribed consumer. Since each broker acts as a consumer, all brokers will receive a copy of the message, guaranteeing delivery to all.

3) Redis vs. Kafka: Redis Streams are faster with in-memory operations, while Kafka relies on disk I/O, which can introduce more latency. Kafka needs a complex setup with multiple components, whereas Redis is simpler and easier to manage. Kafka uses topic partitions for load balancing and efficient offset management; Redis Streams do not have partitions. Kafka automatically manages consumer groups and rebalances them when consumers join or leave, offering robust at-least-once and exactly-once processing.

VII. PERFORMANCE METRICS

Our test bench consisted of a Macbook Air M2 with 8GB RAM. We had a cluster of 3 nodes running on the same system along with Redis. We tested our system with multiple test scenarios and found out that the system was quite resilient to broker failures. The clients can reconnect to other broker nodes in case the one where a client was connected fails.

Key tests included ensuring no packets are sent once a node disconnects, and resuming packet transmission upon node reconnection. Discovery time, reaction to burst messages, and auto-scaling capabilities were assessed, alongside the master-slave load balancer's performance during master node failure [13], [14]. Essential tests covered Redis failure and bandwidth overload. The system's convergence time and resilience against DDOS attacks were also tested. Edge cases included the system's behaviour when only the load balancer is present, fan-in scenarios, message sequencing under different QoS levels, and handling broker failures before acknowledgement [17].

The above performance metrics show the results of mqtt-stresser for the configuration of 2 clients, 10 clients, and 100 clients where each client is publishing 10000 messages each second, which results in about 1 million messages for 100 clients.

For the following test we used the mqttloader tool for measuring the latencies and average throughput. Configuration:

1) No of Publishers: 10
2) No of Subscribers: 10
3) No of messages published per publisher: 10000

Maximum latency [ms]: 42.063
Average latency [ms]: 37.380

Type	**Maximum Throughput**	Average Throughput
Publisher	30335	25000.000
Subscriber	229165	204081.633

VIII. CONCLUSION

The solution thus, provides increased scalability and performance by the use of a Load-Balancer that is responsible for distributing every Connect packet between broker nodes, to ensure equal load among all the brokers. Redis is responsible for storing inflight (messages that are not processed) messages and the client state (including subscription tables, will messages, etc). Redis streams provide us with a fan-out delivery of messages. This packet-level distribution approach proves efficient in implementing the MQTT broker. This approach with enhanced client connectivity, improves scalability. Distributing broker nodes ensures fault tolerance of the system and makes it robust. Using a load-balancer ensures equal load distribution across each node. Redis acts as a global shared memory maintaining consistency. Redis-Streams are used to decouple connections between client nodes and for guaranteed message delivery.

ACKNOWLEDGMENT

We would like to express our heartfelt gratitude to Mr. Ishan Daga, the founding engineer at Golain, for sponsoring the essential research work. We are also deeply thankful to Dr. Geetanjali Kale for her invaluable guidance throughout the project.

REFERENCES

[1] R. Doshi, S. Inamdar, T. Karmarkar and M. Wakode, "Distributed MQTT Broker: A Load-Balanced Redis-Based Architecture," 2024 International Conference on Emerging Smart Computing and Informatics (ESCI), Pune, India, 2024

[2] A. Das, I. Gupta and A. Motivala, "SWIM: scalable weakly-consistent infection-style process group membership protocol," Proceedings International Conference on Dependable Systems and Networks, Washington, DC, USA, 2002

[3] Ongaro, D. and Ousterhout, J., 2014. In search of an understandable consensus algorithm. In 2014 USENIX annual technical conference (USENIX ATC 14) (pp. 305-319).

[4] Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." Proceedings of the NetDB. Vol. 11. No. 2011. 2011.

[5] [mqtt-v3.1.1] MQTT Version 3.1.1. Edited by Andrew Banks and Rahul Gupta. 29 October 2014. OASIS Standard. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. (last visited: 1st May 2024)

[6] [mqtt-v5.0] MQTT Version 5.0. Edited by Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. 07. March 2019. OASIS Standard. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html. (last visited: 1st May 2024)

[7] HashiCorp. "Memberlist: Golang Package for Gossip-Based Membership Protocols." GitHub, https://github.com/hashicorp/memberlist. (last visited: 1st May 2024)

[8] Redis. "Redis: Open source in-memory data structure store." GitHub, https://github.com/redis/redis. (last visited: 1st May 2024)

[9] Mochi-MQTT Server - The fully compliant, embeddable high-performance Go MQTT v5 server for IoT, smarthome, and pubsub. https://github.com/mochi-mqtt/server (last visited: 1st May 2024)

[10] Comqtt - A lightweight, high-performance go mqtt server(v3.0—v3.1.1—v5.0) supporting distributed cluster. https://github.com/wind-c/comqtt (last visited: 1st May 2024)

[11] gRPC - https://grpc.io/ (last visited: 1st May 2024)

[12] Mishra, B.; Mishra, B.; Kertesz, A. Stress-Testing MQTT Brokers: A Comparative Analysis of Performance Measurements. Energies 2021, 14, 5817.

[13] R. Banno, K. Ohsawa, Y. Kitagawa, T. Takada and T. Yoshizawa, "Measuring Performance of MQTT v5.0 Brokers with MQTTLoader," 2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 2021, pp. 1-2.

[14] Load testing tool for MQTT, capable of benchmark test for both MQTT v5.0 and v3.1.1 brokers. https://github.com/dist-sys/mqttloader. (last visited: 1st May 2024)

[15] Koziolek, H., Grüner, S., Rückert, J. (2020). A Comparison of MQTT Brokers for Distributed IoT Edge Computing. In: Jansen, A., Malavolta, I., Muccini, H., Ozkaya, I., Zimmermann, O. (eds) Software Architecture. ECSA 2020. Lecture Notes in Computer Science(), vol 12292. Springer, Cham.

[16] Mishra, B. (2018). Performance Evaluation of MQTT Broker Servers. In: Gervasi, O., et al. Computational Science and Its Applications – ICCSA 2018. ICCSA 2018. Lecture Notes in Computer Science(), vol 10963. Springer, Cham.

[17] I. -D. Gheorghe-Pop, A. Kaiser, A. Rennoch and S. Hackel, "A Performance Benchmarking Methodology for MQTT Broker Implementations," 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), Macau, China, 2020, pp. 506-513, doi: 10.1109/QRS-C51114.2020.00090. keywords: Transport protocols;Performance evaluation;Standardization;Software quality;Benchmark testing;Software reliability;Security;Performance testing;Benchmarking methodology;MQTT;IoT;Open Source;TTCN-3,

[18] Chen, Shanshan & Tang, Xiaoxin & Wang, Hongwei & Zhao, Han & Guo, Minyi. (2016). Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis. 1660-1667. 10.1109/TrustCom.2016.0255.

[19] National Institute of Standards and Technology (2001) Advanced Encryption Standard (AES). (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 197-upd1, updated May 9, 2023. https://doi.org/10.6028/NIST.FIPS.197-upd1