

FAULT TOLERANT CONTAINER-BASED
MESSAGE QUEUING TELEMETRY TRANSPORT
(MQTT) EMBEDDED CLUSTER SYSTEM

THEAN ZHONG YING

MASTER OF ENGINEERING SCIENCE

FACULTY OF ENGINEERING AND GREEN
TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

September 2020

**FAULT TOLERANT CONTAINER-BASED MESSAGE QUEUING
TELEMETRY TRANSPORT (MQTT) EMBEDDED CLUSTER
SYSTEM**

By

THEAN ZHONG YING

A dissertation submitted to
Faculty of Engineering and Green Technology,
Universiti Tunku Abdul Rahman,
in partial fulfillment of the requirements for the degree of
Master of Engineering Science
September 2020

ABSTRACT

FAULT TOLERANT CONTAINER-BASED MESSAGE QUEUING TELEMETRY TRANSPORT (MQTT) EMBEDDED CLUSTER SYSTEM

THEAN ZHONG YING

This dissertation work presents implementations of a distributed MQTT broker cluster in an edge-based environment. Since a single node broker can lose messages to the clients or the cloud when the node crashes. Hence, the purpose of this work to implement local fault tolerance to preserve the distributed system locally at the edge of network. Many previous studies have focused on distributed publish/subscribe systems but very few of them addressed the issue of local fault tolerance and the MQTT standard. Due to the recent popularity of the MQTT protocol, the MQTT middleware layer is developed to facilitate the cooperation of MQTT brokers without modifying the MQTT broker software. Also, the use of single-board computers as an edge-based hosting infrastructure keeps the cost low and can be flexibly sized according to workload demand and location of deployment. The purpose of the edge provisioning of the broker cluster is to reduce end-to-end latency for IoT and M2M streaming applications.

The proposed system uses two approaches to realize fault tolerance. First, the proposed system tolerates node crashes by maintaining consistency of state information using time-to-live (TTL) subscription routing entries. Next, message loss is corrected through retransmission at the broker nodes to the subscribers. The evaluations demonstrated improved scalability for the horizontal scaling approach and successful recovery of failed publication during failover. The worst-case end-to-end latency of the proposed system is at a maximum of 42 milliseconds. All missed publications are redelivered to the subscriber during failover without significant delay between the retransmitted messages. The jitter values between recovered messages during the recovery period range from 10 to 20 milliseconds. The maximum recovery time of the proposed broker cluster is at least 256.33 milliseconds, which is within hundreds of milliseconds difference, compared to 50 milliseconds of the primary-backup broker approach. The fail-test confirms the reliability of the MQTT cluster, as failed publications can be redelivered during broker failure. The evaluations demonstrated the feasibility of the proposed broker cluster to maintain consistent latencies and support reliable MQTT services despite server failures.

ACKNOWLEDGEMENT

This work would not have possible without the support of my supervisor, Dr. Yap Vooi Voon, and co-supervisor, Dr. Teh Peh Chiong, who worked actively to provide me patient guidance and encouragement. The advice I received on this dissertation work has been invaluable. Also, I am grateful to all those with whom I had the pleasure to work during this project. Finally, special thanks to all my family members.

APPROVAL SHEET

This thesis/dissertation entitled “**FAULT TOLERANT CONTAINER-BASED MESSAGE QUEUING TELEMETRY TRANSPORT (MQTT) EMBEDDED CLUSTER SYSTEM**” was prepared by THEAN ZHONG YING and submitted as partial fulfillment of the requirements for the degree of Master of Engineering Science at Universiti Tunku Abdul Rahman.

Approved by:



(Dr. Yap Vooi Voon)

Date: 06/07/2020

Professor/Supervisor

Department of Electronics Engineering

Faculty of Engineering and Green Technology

Universiti Tunku Abdul Rahman



(Ir. Dr. Teh Peh Chiong)

Date: 06/07/2020

Professor/Co-Supervisor

Department of Electronics Engineering

Faculty of Engineering and Green Technology

Universiti Tunku Abdul Rahman

FACULTY OF ENGINEERING AND GREEN TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

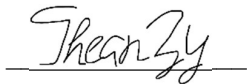
Date:

SUBMISSION OF DISSERTATION

It is hereby certified that **Thean Zhong Ying** (ID No: **18AGM05810**) has completed this dissertation entitled “**FAULT TOLERANT CONTAINER-BASED MESSAGE QUEUING TELEMETRY TRANSPORT (MQTT) EMBEDDED CLUSTER SYSTEM**” under the supervision of **Dr. Yap Vooi Voon** (Supervisor) from the Department of **Electronics Engineering**, Faculty of **Engineering and Green Technology**, and **Dr. Teh Peh Chiong** (Co-Supervisor) from the Department of **Electronics Engineering**, Faculty of **Engineering and Green Technology**.

I understand that the University will upload a softcopy of my dissertation in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

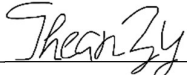
Yours truly,



(Thean Zhong Ying)

DECLARATION

I, Thean Zhong Ying hereby declare that the thesis/dissertation is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not previously or concurrently submitted for any other degree at UTAR or other institutions.



(Thean Zhong Ying)

Date: 06/07/2020

TABLE OF CONTENTS

ABSTRACT	ii
ACKNOWLEDGEMENT	iv
APPROVAL SHEET	v
SUBMISSION SHEET	vi
DECLARATION	vii
TABLE OF CONTENTS	viii
LIST OF TABLES	xi
LIST OF FIGURES	xii
LIST OF ABBREVIATIONS	xv
LIST OF PUBLICATIONS	Err

or! Bookmark not defined.

CHAPTER

1 INTRODUCTION	1
1.1 Scope and Goals	5
1.2 Application Scenario	6
1.3 Contribution of This Research Work	7
1.4 Structure of Dissertation	8
2 EDGE CLOUD SYSTEMS	9
2.1 Edge-cloud Internet of Things	9
2.1.1 Fog Computing	14
2.1.2 Edge Computing	15
2.1.3 Mist Computing	15
2.1.4 Dew Computing	16
2.2 Edge-based Container Orchestration	16
2.2.1 Container Orchestration with Single-board Computers	19
2.2.2 Microservice Architecture	20
2.2.3 Docker Distributed Networking	22
2.3 Fault Tolerance in IoT systems	26
2.3.1 Redundancy	26
2.3.2 Checkpointing	28
2.3.3 Container Service Migration	30
2.3.4 Load Balancing	31
2.4 Summary	31

3	PUBLISH-SUBSCRIBE SYSTEM	33
3.1	MQTT	35
3.2	Commercial and Open Source MQTT Brokers	38
3.3	Distributed Publish-subscribe System	44
3.3.1	Distributed Routing Mechanism	45
3.3.2	Overlay Infrastructure	50
3.4	Publish-subscribe Fault Tolerance	56
3.4.1	Distributed State Recovery	57
3.4.2	Periodic Subscription	58
3.4.3	Self-stabilization	59
3.4.4	Event Retransmission	61
3.4.5	Redundant Paths	63
3.4.6	Consensus-based Publish-subscribe System	63
3.4.7	Availability of Distributed Publish-subscribe Systems	64
4	DESIGN AND IMPLEMENTATION	66
4.1	Broker Cluster Architecture	66
4.2	Software Application Stack	71
4.3	Components Relationship	75
4.4	Broker Cluster Topology	76
4.5	Subscription Routing Management	77
4.6	Publication Message Forwarding	82
4.6.1	Normal Condition	82
4.6.2	Message Forwarding with Failed Brokers	84
4.7	Recovery of Routing State	86
4.7.1	Monitoring and Failure Detection	86
4.7.2	System State Reconfiguration	89
4.8	Implementation	92
5	RESULTS AND EVALUATIONS	95
5.1	Experiment Setup	95
5.2	Throughput	99
5.3	Latency	102
5.4	Microservice and Monolithic Broker Comparison	110
5.5	Inter-message Jitter	114

5.6	Evaluating Publication Retransmission	116
5.6.1	Throughput	117
5.6.2	Inter-message Jitter	118
5.6.3	Latency	121
5.7	Resource Usage	129
5.7.1	CPU Utilization	129
5.7.2	RAM Usage	132
5.8	Discussion	137
5.8.1	Throughput	137
5.8.2	Latency	139
5.8.3	Performance overhead of the microservice-based broker cluster	140
5.8.4	Inter-message Jitter	141
5.8.5	Impact of Message Publication Rate	142
5.8.6	Failure recovery and comparison to primary/backup broker	143
5.8.7	Resource usage	146
6	DISCUSSION AND CONCLUSION	148
6.1	Introduction	148
6.2	Methodology Used	149
6.3	Summary of Results	150
6.4	Future Work	151
	REFERENCES	153
	APPENDICES	
A	LIST OF PUBLICATION	163

LIST OF TABLES

Table		Page
2.1	Comparison of Edge, Cloud, Fog, and Mist (Dogo et al., 2019)	13
3.1	MQTT broker cluster implementations (Mishra, 2019)	39
3.2	Summary of generic publish-subscribe systems (<i>Setty et al., 2012</i>)	45
5.1	Experimental setup configurations	98
5.2	Summary of latency and throughput results	141

LIST OF FIGURES

Figure		Page
2.1	Edge Cloud Paradigm	14
2.2	Monolithic and microservice implementation (Cicizz, 2019)	21
2.3	Docker Network Configuration (To et al., 2015)	23
2.4	Docker Swarm Network Overlay (Church, 2019)	24
2.5	Docker Swarm Ingress Network (Church, 2019)	25
3.1	Publish-subscribe sequence	34
3.2	MQTT publish-subscribe sequence	36
3.3	MQTT message format (Tang et al., 2013)	38
3.4	Distributed and local area broker network	43
4.1	Edge cloud MQTT broker cluster architecture	68
4.2	Data Pipeline	71
4.3	Cluster container application stack	72
4.4	Software component interaction	75
4.5	Topic Trie for Forwarding Table	79
4.6	Broker publication message forwarding	83
4.7	Message routing sequence	84
4.8	Message retransmission process	85
4.9	Cluster membership heartbeat detection	87
4.10	Load balancer TCP health checks	88
4.11	MQTT keep-alive	89
4.12	MQTT broker cluster implementation	94
5.1	System metrics monitor	98
5.2	Average publish throughput with increasing clients (QoS 2)	101
5.3	Throughput variations with 50 and 2000 clients	102

5.4	Message forwarding between backend servers	104
5.5	Average end-to-end latencies with increasing clients	106
5.6	Worst case end-to-end latencies with increasing clients	107
5.7	Latency histogram with a normal density curve	108
5.8	Latency quantile plot against normal probability distribution	109
5.9	Latency cumulative distribution function with the normal probability curve	109
5.10	Latency probability plot against normal probability distribution	110
5.11	End-to-end latencies performance comparison	111
5.12	Average publish throughput (QoS 2)	113
5.13	Throughput degradation (message per second)	113
5.14	Maximum jitter of the broker cluster	115
5.15	Minimum jitter of the broker cluster	115
5.16	Percentile 90 jitter of the broker cluster	116
5.17	Fail test for the broker cluster	116
5.18	Message throughput for fail test	117
5.19	Jitter under normal condition	119
5.20	Jitter under one fail node	120
5.21	Jitter under two fail nodes (at same time)	120
5.22	Latency histogram with a normal density curve	121
5.23	Latency quantile plot against normal	122
5.24	Latency CDF plot with a normal probability distribution curve	122
5.25	Latency probability plot against normal	123
5.26	Latency histogram with a normal density curve	124
5.27	Latency quantile plot against normal	125
5.28	Latency CDF plot with a normal probability distribution curve	125
5.29	Latency probability plot against normal	126
5.30	Latency histogram with a normal curve	127

5.31	Latency quantile plot against normal	127
5.32	Latency CDF with a normal probability distribution	128
5.33	Latency probability plot against normal	128
5.34	Time series of CPU time utilization for broker cluster	130
5.35	Time series of CPU time utilization for single node broker	131
5.36	Time series of CPU time utilization for two failed brokers	132
5.37	Time series of total memory usage for broker cluster	133
5.38	Time series of total memory usage for single-node broker	135
5.39	Time series of total memory usage under two failed brokers	136

LIST OF ABBREVIATIONS

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CAP	Consistency, availability, and partition tolerance
COAP	Constrained Application Protocol
CPU	Central processing unit
DHCP	Dynamic Host Configuration Protocol
DHT	Distributed hash table
DNS	Domain name system
EDF	Earliest deadline first
I/O	Input/Output
IoT	Internet of Things
IP	Internet Protocol
IPVS	Internet Protocol Virtual Server
LXC	Linux userspace interface for container virtualization
M2M	Machine to machine communication
MAC	Media Access Control
MQTT	Message Queuing Telemetry Transport

NAT	Network address translation
NFC	Near field communication
P2P	Peer-to-peer
PaaS	Platform as a Service
QoS	Quality of Service
RAM	Random-access memory
RFID	Radio frequency identification
RPC	Remote procedure call
SBC(s)	Single-board computer(s)
SDN	Software-Defined Network
SF algorithm	Subscription flooding algorithm
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol and the Internet Protocol
TTL	Time-to-live period
UDP	User Datagram Protocol
VXLAN	Virtual Extensible Local Area Network
Wi-fi	A family wireless network protocols based on IEEE 802.11 standards
WSN	Wireless Sensor Network
XMPP	Extensible Messaging and Presence Protocol

CHAPTER 1

INTRODUCTION

Internet of Things (IoT) technology usage is increasing extensively as a result of the exponential growth of sensor device usage over the past decade (Robert, 2014). This development will generate huge amounts of traffic on the internet. Traditional approaches to the Internet of Things (IoT) send data to the cloud for processing and then transmit the responses back to the end devices. These approaches are not viable anymore due to the rapid growth of IoT devices, as cloud facilities will have a difficult time managing the huge amount of information flow. Even though the size of each sensor data is small, a substantial amount of data produced by IoT devices could congest the flow of network traffic and causes delays to the data transfers to and from cloud data centers. Besides, many IoT applications with timing constraints do not work well with cloud-based data processing and dissemination, because of the high bandwidth requirements and unpredictable latency (Lopez et al., 2015).

Recently, sensor-based IoT applications have been using edge and fog technologies to integrate latency-sensitive edge-based environments with the cloud (Lopez et al., 2015). Edge computing environment is facilitated by small, heterogeneous embedded devices that spread across multiple edge networks. With local data processing, edge computing can reduce end-to-end latency by immediately updating the results over the network. Nevertheless, deployed devices on edge infrastructures are often constrained by resource, limited

processing power, power supply, and storage capacity. Due to resource limitations, these smaller edge devices will need a software framework that is lightweight to deploy distributed applications. Containerization technology establishes a lightweight middleware platform that provides scalability, service orchestration, and flexibility for edge-based service deployment (Pahl and Lee, 2015; Pahl, 2015). Containers are software packages like virtual machines but they are lightweight and smaller in size. Several previous works have demonstrated the feasibility of using container orchestration technology for edge-based applications (Roberto Morabito, 2017; Bellavista and Zanni, 2017). This research work uses the Docker Swarm orchestration tool to deploy distributed applications for IoT edge devices.

The publish-subscribe paradigm is a communication paradigm, which provides a loosely coupled form of data dissemination between producers and consumers. Loose coupling and lightweight properties of the publish-subscribe paradigm is well suited for two-way communication in between IoT endpoint devices. Several IoT standards have been adopting the publish-subscribe interface over recent years. Some examples of them are AMQP, MQTT, XMPP, ZeroMQ, and COAP. The Message Queuing Telemetry Transport (MQTT) is a publish-subscribe protocol that uses a central broker to mediate messages between endpoint devices through the cloud (Popov et al., 2017). The centralized paradigm of MQTT is suitable for cloud computing as it can effectively collect and distribute data using a central broker. However, a cloud-based MQTT broker is very ineffective (Happ and Wolisz, 2016) for data dissemination between edge devices because MQTT communications usually

involve edge-heavy environments (Banno et al., 2017). In edge-heavy environments, data producers and consumers are located close to each other in a certain geographical location. Edge-based dissemination with MQTT protocol provides many benefits particularly in latency reduction (Happ and Wolisz, 2016), as well as additional data analytics near the edge (Cheng et al., 2016). Edge-based deployment of MQTT moves the centralized single-broker topology to a distributed, multi-broker topology to serve communication between devices in multiple edge network bases.

Potential large-scale deployment of IoT creates huge traffic to the MQTT broker server which causes congestion and reduced throughput. The MQTT service needs to deal with large volumes of periodical short messages in M2M and IoT applications (Pereira and Aguiar, 2014). When the volume of traffic exceeds the capability of the MQTT broker, the MQTT service must scale horizontally to deal with the traffic. The publish/subscribe-based MQTT protocol can support horizontal scalability because its message communications are asynchronous (Happ et al., 2017). With horizontal scaling, the system can preserve the availability of resources despite server failures.

For IoT and M2M applications, the MQTT server needs to maintain high availability and resilience. The broker system may lose its messages to its clients or the cloud when the broker node crashes. Thus, local fault tolerance needs to be implemented to preserve the system locally at the edge of network. Existing studies on topic-based publish-subscribe systems address the problem of scalability and fault tolerance by utilizing distributed brokers on top of various

kinds of overlay infrastructures. Examples of used overlay infrastructures for distributed publish-subscribe systems are DHT (Castro et al., 2002), hybrid overlay (Rahimian et al., 2011), Skip-Graph overlay (Banno et al., 2015), and broker-based overlay (Carzaniga et al., 2003). However, very few of the studies address the issue of local fault tolerance in distributed MQTT system.

The objective of this research work is to develop a lightweight distributed middleware layer, based on the edge-cloud computing model to support collective message delivery with a cluster of MQTT brokers and to improve its fault tolerance. Edge cloud devices often require cost-efficiency, low power consumption, and robustness (Novo et al., 2015). Edge servers can use single-board computers (SBCs) with networking capabilities such as Raspberry Pi as their hardware infrastructure. The broker cluster is deployed on a single cluster of Raspberry Pi SBCs. Docker Swarm is used for orchestrating service containers at the edge of the cloud. The proposed broker cluster uses a single load balancer node to distribute incoming MQTT data to multiple backend MQTT brokers. The implementation assumes that the load balancer is reliable so that the focus is on the fault tolerance against the crashing of backend servers. Experiments are carried out to evaluate the latency, throughput, computational load, and fault tolerance of the broker system. The evaluation also compares the performance of the microservice-based broker cluster with cloud-based, single-broker, and monolithic implementations of the broker-cluster. The resiliency of the broker cluster is also compared to a primary-backup approach, in terms of its maximum recovery time. Through combining multiple MQTT brokers in a single and collective cluster, the system continues

to provide MQTT broker service regardless of node failures in the broker cluster. The clustered broker can resend lost publications and perform failover and among MQTT brokers in the network.

1.1 Scope and Goals

The main goal of this research work is to develop a distributed MQTT messaging framework on a set of SBC edge servers. The messaging service should be scalable and have local fault tolerance capability. This work establishes the feasibility of the microservice-based MQTT cluster implementation and Docker Swarm orchestration as a suitable MQTT service for edge cloud settings. The scope of this research work consists of four goals detailed below.

- Scalability. The process of matching subscribers and disseminating publications to subscribers incurs high processing costs. The proposed broker system must reduce the effect of service degradation under high loads or sudden load spikes.
- Application transparency. The microservice-based distributed application layer service offers application transparency to clients. MQTT service is delivered to clients without their need to know underlying software and hardware. The application service layer is also transparent to the broker. The application broker itself does not require recompilation or relinking.
- Low and predictable latency. Low latency is the response time as perceived by the client. Unstable latency is inappropriate for delay-

sensitive applications that require timely event delivery. Edge-based IoT applications should ideally provide low and predictable latency for end-to-end communication between IoT client devices.

- Availability and fault tolerance. Components of a distributed system are prone to unexpected failures that could potentially bring down an entire distributed application. For a publish-subscribe system, in-transition messages may be dropped, which leads to inconsistency among routing tables. Incorrect routing table state disrupts the entire message delivery traffic. This invalidates the purpose of the publish-subscribe system to reliably deliver messages. Therefore, a publish-subscribe service requires additional redundancy to increase availability and resiliency. The proposed broker system uses two approaches to realize fault tolerance. First, it handles node crashes by maintaining consistency of state information through a leasing approach with the use of time-to-live (TTL) countdown entries. Next, it uses message redundancy to realize fault tolerance. Message loss is corrected through the retransmission of buffered messages from broker nodes to the disconnected subscribers.

1.2 Application Scenario

In centralized cloud computing settings, IoT applications collect and send data to the centralized cloud data center, where storage and processing take place. This research work implements a low-cost and lightweight solution to disseminate MQTT messages and provide local fault tolerance for the edge-based MQTT system. The main application scenarios are data streaming and analytics, real-time monitoring, and automation systems. Instead of using a

traditional compute cluster, the SBC-based cluster keeps the cost of incrementing a cluster node low. Also, the flexibility of single-board computers (SBC) and software implementation can support use cases in secluded locations with limited connectivity and computational resources. The edge servers help to provide a subset of resilient online services for communication between edge devices in the vicinity, by providing redundancy and local fault tolerance. Examples of applications are local smart grids (Viswanath et al., 2016), autonomous farm monitoring (Zyrianoff et al., 2018; Zamora-Izquierdo et al., 2019), and distributed systems for homes and buildings automation (Babou et al., 2018).

1.3 Contribution of This Research Work

The contributions of this work are the following.

- Develop a microservice-based cluster component to automate the initial formation of the MQTT cluster and facilitate routing of MQTT messages.
- Develop an edge to cloud IoT integration framework to stream edge data to the cloud servers.
- Establish the feasibility of using Docker Swarm as an orchestration framework for edge-based deployment on Raspberry Pi single-board computers.
- Establish the feasibility of the software implementation as a scalable, lightweight solution for the deployment of MQTT service at the edge network.

- Provide resiliency and high availability for the MQTT messaging application.

1.4 Structure of Dissertation

The rest of this dissertation is presented as follows. Chapter 2 presents various enabling technologies related to the scope of this research work. The topics discussed in this chapter are background information about IoT, edge and cloud computing technologies, container virtualization and orchestration, and fault tolerance approaches to IoT systems. Chapter 3 discusses fault tolerance and scalability in publish-subscribe systems, and the MQTT protocol. Chapter 4 presents the architecture and components of the system. Chapter 5 presents the evaluations on the performance and fault testing of the proposed broker cluster system. The last chapter presents the conclusions. Chapter 6 concludes this research work.

CHAPTER 2

EDGE CLOUD SYSTEMS

This chapter describes the technologies and related works in the context of fault tolerance concerning IoT systems, and container orchestration in the edge-cloud paradigm. The literature review describes various practical and technology integrated approaches to develop a scalable and fault-tolerant, edge-based messaging system. This chapter consists of four sections. Section 2.1 compares various edge and cloud technologies in the context of IoT. This section also outlines the use of container technology in the deployment of IoT systems. Section 2.2 introduces Docker and the Docker Swarm container virtualization framework, which will be the software deployment tools used for MQTT cluster servers. Section 2.3 presents the fault tolerance approaches to the IoT edge-cloud systems.

2.1 Edge-cloud Internet of Things

In recent years, advancement in cloud computing introduces many growing progressions in cellular internet and cloud-based Internet of things through a wide range of software and services. Due to the convergence of IoT enabling technologies such as ubiquitous computing, communication protocols, embedded systems, sensors, and wireless communications, all physical objects can interconnect and exchange data with each other with minimum human intervention. The sensor devices collect and produce information about internal states of devices or external environment such as temperature, humidity, light, soil conditions, pressure, and radiation. These resource-constrained devices are

typically are usually equipped with wireless communication technologies such as Wi-Fi, RFID, Near Field Communication (NFC), and Bluetooth Low Energy. Each sensor device has a microcontroller, that is either used to manage communications or to process incoming information. In general, cloud-based IoT systems involve three major technologies which are embedded systems, middleware, and cloud computing. Embedded systems process information in the front-end devices. The middleware interconnects heterogeneous front-end embedded devices to the cloud. The cloud computing platform provides storage, processing, and management services.

Recent emerging applications surrounding IoT have increased the number of connected devices that will require real-time processing (Adjih et al., 2015). Although IoT commonly uses cloud computing, it becomes a bottleneck for sensor-based IoT applications due to bandwidth requirements, latency, fault tolerance, and security issues (Shi and Dustdar, 2016). Applications such as data streaming will presumably serve devices with a high data rate. Data generated by sensor devices is transmitted to the distant cloud network for processing, storage, and analytics. Eventually, the frequent rate of data production will exceed the bandwidth availability. This results in a long delay in transmission requests, more network congestion, and reduced network connectivity. Therefore, it is not feasible to use the cloud as an intermediary transfer medium between local endpoint devices. Also, waiting for a request from a cloud server can be disastrous as delay-sensitive applications expect an immediate response, often within tens of milliseconds. In a safety-critical control system, a delay in response time may cause severe injury to humans or damage to the machine.

Typical cloud computing arrangements fail to achieve the bandwidth and latency requirements. Consequently, the IoT paradigm is moving away from the centralized structure as computations are moved closer to the edge of network.

Edge and fog computing are intermediate layers of network infrastructure which integrate the cloud with sensor-based IoT environments. The edge computing paradigm complements cloud computing by moving substantial compute, storage resources, and existing cloud computing services closer to the edge of a network, usually one hop away from IoT devices (Lopez et al., 2015). This reduces unnecessary transfer latency as compared to cloud computing. Local information caching and selective information processing network edge avoids high data volume transferring to the central cloud, as data sources and actuation will be processed within the same location. Thus, user experience and quality of service are improved, since the latency in edge computing is typically lower than cloud computing.

Edge computing can fulfill the requirements such as improved bandwidth, low latency, and low power in many IoT applications (Shi and Dustdar, 2016). According to various studies (Dastjerdi and Buyya, 2016; Morabito et al., 2016), edge technologies such as IoT gateways, local data management, data aggregation, and data filtering are becoming the main preference for integration between cloud and heterogeneous IoT devices. Morabito et al. (2018) propose LEGIoT, an edge gateway that uses container virtualization technologies to manage information exchanges between cloud and sensors. IoT gateways bridge cloud services and IoT sensors and provide

them connectivity and data routing functionalities. Lertsinsruttavee et al. (2017) present the PiCasso framework for service deployment based on a fog computing model. The PiCasso framework uses technologies such as SBC, container service orchestration, and software-defined networks (SDN).

In publish-subscribe scenarios, a centralized cloud model works poorly for sensor-based IoT environments. This is because most data exchanges that usually happen locally around the edge network need to travel across the central cloud (Banno et al., 2017). In an edge broker approach, message brokers are placed in front of the edge network and directly interact with endpoint devices. Publications and subscriptions are collected at the broker closest to the devices. This improves the capability of brokers to send an immediate response without waiting for responses from the cloud.

Table 2.1 Comparison of Edge, Cloud, Fog, and Mist (Dogo et al., 2019)

	Cloud	Edge	Fog	Mist
Deployment model	Centralized	Centralized with distribution	Decentralized/ distributed gateways	Centralized or distributed with microcontroller network
IoT support	Yes	Yes	Yes	Yes
Latency	High	Low	Low within 100ms	Very low
Bandwidth	Very High	Low	Low	Very Low
Power consumption	Very High	Moderate	Moderate	Very Low
Computational power	Very high	Moderate	Moderate	Low
Service Coverage	Global	Limited Spread	Wide Spread	Very limited
Hardware	High specifications	Limited	Limited	Very Limited

The edge cloud architecture is facilitated by smaller devices that spread across the network consists of edge nodes, fog cloudlets, and cloud servers. Figure 2.1 shows a variety of edge computing technologies which include fog, edge, mist, and dew computing (Naha et al., 2018). In the cloud hierarchy, dew computing has the lowest latency and processing power, while cloud computing has the highest latency and processing power. This section gives an outline of the edge computing architecture compares it to cloud and fog computing. Table 2.1 compares different aspects of various edge and cloud technologies. The following subsections discuss various forms of edge cloud technologies.

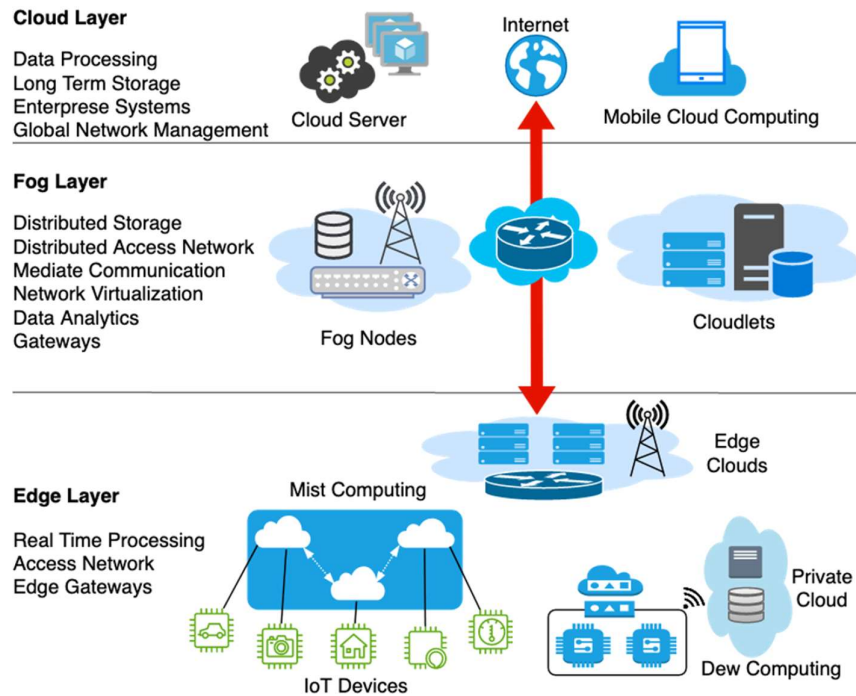


Figure 2.1 Edge Cloud Paradigm

2.1.1 Fog Computing

Fog computing is a highly virtualized platform service consisting of computing elements such as fog nodes and cloudlets. These fog computing elements provide extended services of the central cloud, at the level of routers and gateway. The fog elements process and store some of the data between endpoint devices and cloud data centers. Fog computing systems are integrated into a cellular network for mobile carrier usage which utilizes devices like M2M communication and wireless routers (Tandon and Simeone, 2016). Fog computing reaches from IoT devices to the edge and core of the network, while edge computing is only limited to computing at the edge (Chiang et al., 2017). Cloudlet is a fog layer mobile computing platform which stores and process data without going through the remote cloud, to reduce response time (Ahmed and Ahmed, 2016). Cloudlet uses virtualized technology that sits between mobile

devices and remote cloud. Cloudlets can improve processing speed and save energy, by providing internet services to endpoint mobile devices that offload resource-intensive tasks and data caching (Pang et al., 2015).

2.1.2 Edge Computing

Edge computing systems deploy edge servers into edge networks where endpoint devices are connected to a base station. Edge computing reduces the amount of data being to the cloud by processing latency constrained data locally at the edge and sends filtered data to the cloud for storage or further processing. Thus, avoiding network traffic and allows faster response time. However, edge computing may not assure ultra-low latency for some real-time applications. Heavy information traffic will overload the edge server. This can cause many issues such as missing latency deadlines, scalability problems, and server failures (Satria et al., 2017). Therefore, edge computing facilities need to provide high availability and fault tolerance to guarantee uptime and performance.

2.1.3 Mist Computing

Mist computing offloads tasks to the outermost edge of the IoT network at the layer of microcontrollers and embedded devices, without burdening the communication network on the Internet (Preden et al., 2015). This results in reduced latency and bandwidth usage. The main applications of mist computing are machine-to-machine (M2M) communication services, where edge devices can access communicative resources in their vicinity (Liyanage et al., 2016; Tammemäe et al., 2018). Mist computing is used to preserve the privacy of

internet applications via local processing in the early days (Campbell et al., 2003). Recently, mist computing is extended to collaborate with fog and cloud computing and has self-awareness about its local situation and physical environment (Tammemäe et al., 2018). The infrastructure of mist computing consists of heterogeneous devices located in front of the network edge, to provide various IoT services and improve computational processing power (Liyanage et al., 2016).

2.1.4 Dew Computing

Dew computing is a local client/server-based paradigm, whereby on-site local servers provide cloud-independent services, while collaborative to the cloud services. With dew computing, users can access cloud services even when there is no internet connection. Babou et al. (2018) propose a framework similar to dew computing called home-edge computing. The home-edge computing approach offloads certain latency constrained tasks to the edge compute server located nearer to end-users. Also, the home-edge server synchronizes with the edge and central cloud servers. If the local resource is unavailable, the client requests are delegated in a hierarchical way towards the edge and central cloud servers. Dew computing is also used to offload cloud computing tasks and stream data between IoT sensors and devices (Gusev, 2017).

2.2 Edge-based Container Orchestration

Edge devices are constrained in terms of computational power, power consumption, and connectivity. The hardware of edge computing devices usually has processing power, memory, and storage capabilities lower than that

of typical server machines (R Morabito, 2017). Embedded computers and micro-servers are widely used in edge-based IoT application domains as they facilitate lower cost, lower energy consumption, considerable computational power. Edge-based deployment on constrained devices will require lightweight software solutions. Container virtualization technology is capable to facilitate edge because it is lightweight, portable, easy to deploy, and has near-native performance (Felter et al., 2015; Morabito and Bejar, 2016). One drawback is the extra User Datagram Protocol (UDP) traffic between containers such as the Network Address Translation (NAT) feature of Docker.

Container technologies such as Docker provide process isolation, less virtualization overhead, and local fault-tolerant for edge computing (Ismail et al., 2015). The size of each container is minimal because kernel resources and standard libraries are shared between containers. However, a container cannot run on a different platform within itself. Containers are also easy to deploy because of continuous integration tools and development environments. Containers solve the dependency tree issue of the development and deployment environment as application packages are wrapped in container images along with utilities and shared libraries. Many container platforms are built on top of Linux LXC techniques, by using kernel mechanisms such as *cgroups* and *namespaces* to isolate process environments from system resources. Linux *cgroups* limits container process to system resources. Linux *namespace* isolates the container view of the runtime environment. Processes within different containers all share the same kernel, but each has different views of isolated system resources such as process management interfaces, network stack, and

filesystem namespace. Popular projects based on container virtualization are Docker, LXC, CoreOS, and Apache Mesos.

Docker Swarm is a container orchestration tool that enables users to deploy Docker containers on multiple Docker hosts within the distributed edge-cloud environment. All host nodes of Docker Swarm run the Docker engine, and co-operate together as a tightly-coupled unit, to deploy containerized workloads. Docker Swarm orchestration also helps to improve availability and fault tolerance by spreading application services across redundant micro-service nodes (Alam et al., 2018). With microservice, different components of the same application can be implemented as several small services to achieve a collective target.

In a cluster of Docker Swarm hosts, a *service* is an abstraction of Docker containers. The *service* is based on specific Docker images, and is comprised of a set of different tasks, and are implemented as individual containers. Docker Swarm makes use of the Raft Consensus algorithm to maintain the cluster state (Vohra, 2017) and preserve fault tolerance of the Swarm cluster. For practical use, the cluster requires multiple manager nodes to ensure the maintenance of the cluster state. If there are n manager nodes, the cluster requires a minimum quorum of $\frac{n}{2+1}$ manager nodes to operate normally, and can tolerate up to $\frac{n-1}{2}$ manager node failures.

The combination of edge computing and orchestration enables a highly dynamic and distributed deployment of services. Containerization is a viable lightweight virtualization option for single-board computers, as demonstrated in various studies. Pahl et al. (2016) present an architecture to support fault-tolerant edge-cloud service orchestration through containers, which consequently avoids the high volume of data flooding into the cloud network. All communication, decision making, and analytics services are packaged into different containers and deployed on selected edge nodes. Alam et al. (2018) have used Docker as a distributed service platform for fault tolerance, via microservice-based edge deployment. For IoT applications, Docker-based IoT gateways are also implemented. Morabito and Beijar (2016) propose an IoT gateway framework and evaluated the performance of containerized IoT gateways running on single-board computers. Their evaluations show that there is minimal overhead on containerized applications over native host applications for the Raspberry Pi 2 SBC in terms of CPU time (2.67%), memory (6.04%), and disk access speeds (10%), and power consumption (10%).

2.2.1 Container Orchestration with Single-board Computers

Low-cost single-board computers provide low energy and reduced infrastructure cost while still capable of running complex software services. Studies have shown the feasibility and limitations of Raspberry Pi boards in Docker-based edge-cloud deployments regarding their performance, energy, and cost-effectiveness (Pahl and Lee, 2015; Bellavista and Zanni, 2017). A Raspberry Pi single-board computer offers lower energy consumption and considerable computational power. Since the Raspberry Pi SBCs lack

computing power, it cannot run computationally intensive software. Nevertheless, this downside of the SBCs can be remedied by combining a larger number of these devices into an affordable and energy-efficient cluster. Thus, edge-based deployment can be flexibly configured and customized to accommodate demanding workload and scarce environments, where limited resources are available.

2.2.2 Microservice Architecture

This research work implements a microservice-based MQTT broker cluster within the Docker Swarm cluster. The microservice architecture splits the application into a suite of smaller, interconnected services as opposed to a single monolithic application. Figure 2.2 depicts the difference between monolithic and microservice-based MQTT cluster implementation. This implementation follows the adapter pattern (Burns and Oppenheimer, 2016). The adapter pattern enables the use of any MQTT broker implementations as long as the middleware present a uniform interface that implements the MQTT standard. MQTT clustering implementations similar to the microservice architecture are presented in ILDM and Nucleus (Banno et al., 2017; Sen and Balasubramanian, 2018).

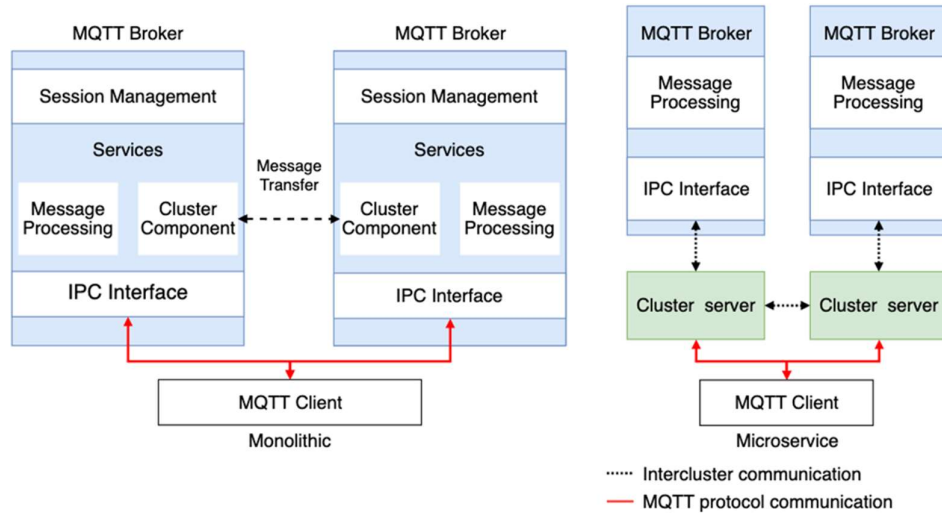


Figure 2.2 Monolithic and microservice implementation (Cicizz, 2019)

The combination of modularity of microservice architecture and Docker orchestration increases the dynamicity of deployment for distributed systems (Alam et al., 2018). The services are loosely coupled and have their application logic combined with various adapters. Some microservices expose an API to communicate with other services in the application. Some microservices communicate with established communication protocols such as AMQP, MQTT, and RPC. The microservice architecture divides a single application into a discrete set of smaller services, each with its application logic and communication mechanisms. Microservices are well supported by container virtualization technologies. Microservices can be independently deployed and automatically scheduled with an orchestration framework.

2.2.2.1 Benefits of Microservices Architecture

Microservice architecture decomposes a complex application into a set of manageable services that are easier to understand and maintain. Each service can be independently maintainable and deployable. This reduces the barrier of

adopting new technologies because of the flexibility of technology stacks. Services can be implemented in different programming languages, which fits best for their applications. Each microservice can be granularly scaled within the cluster computer. As demonstrated in (Sen and Balasubramanian, 2018), microservice can increase fault isolation as the application logic is distributed across different layers. Failures of devices and microservices can be masked by the inherently redundant architecture of the system.

2.2.2.2 Drawbacks of microservice architecture

Microservices architecture adds complexity because of additional inter-process communication mechanisms, such as message passing or RPC, between components. Developers also have to handle partial failure components in the application. A monolithic application simply deploys a set of identical servers behind a load balancer. In contrast, each service in a microservice architecture will have multiple runtime instances and each instance needs to be independently managed. A microservice application needs to implement a service discovery mechanism to support communication between services.

2.2.3 Docker Distributed Networking

Each Docker container is a virtual network host within the host machine. Each container has an internal network attached to the virtual ethernet interface. The internal network also provides a private network address. The container uses the host machine network address as the default gateway for external communication. As shown in Figure 2.3, the local bridge network of the Docker

host connects to containers through a virtual ethernet bridge on the same Docker host. Docker containers can operate in two network modes. Bridge mode isolates the network environment for each container residing on the host. The bridge forwards packets between networks based on each MAC address of the virtual network interfaces. On the other hand, the host mode directly links containers to the network interface of the host. The proposed broker system in this work uses host mode because the host mode performs better in a multi-container environment (Lee et al., 2018).

The container attached to the default bridge network does not involve in intra-cluster service communication, because the scope of the bridge network is limited to the local cluster nodes. Services running on other nodes will be unable to consume the local service. Docker Swarm overlay is the network solution used in Swarm to manage communication between containers (Merkel, 2014). The container clusters in Docker Swarm support network connection of containers using network virtualization features. Docker Swarm uses overlay networks, based on VXLAN networking protocol, to enable virtual networks to span multiple Docker hosts as shown in Figure 2.4.

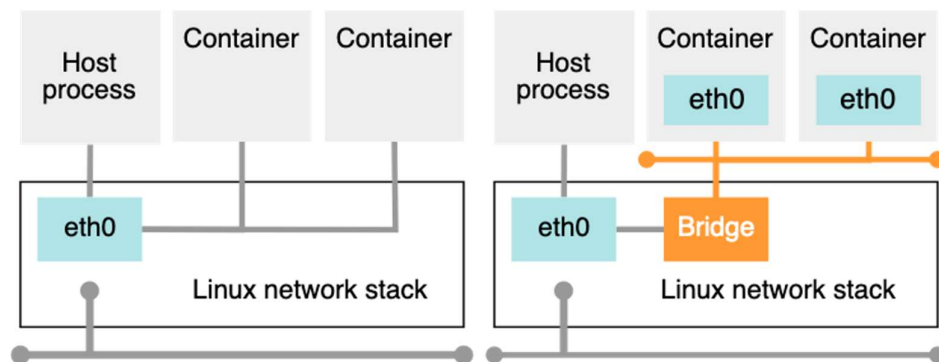


Figure 2.3 Docker Network Configuration (To et al., 2015)

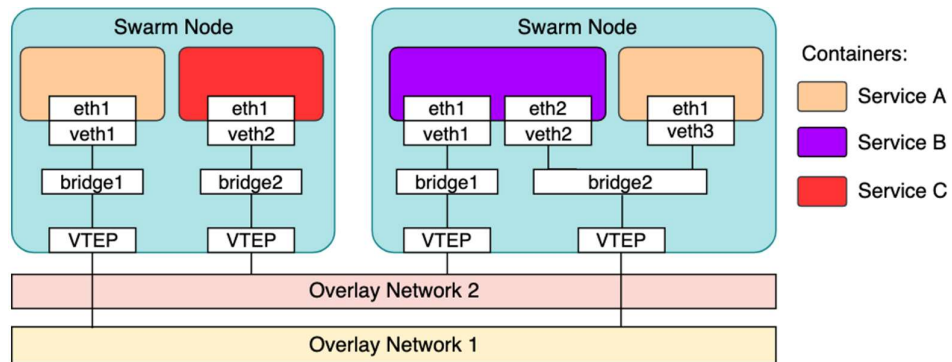


Figure 2.4 Docker Swarm Network Overlay (Church, 2019)

The overlay network forms a single Ethernet network across all machines within the Swarm cluster. Containers, and services on different Docker hosts to communicate with each other through dedicated overlay networks they are associated with. Overlay networks construct virtual links between machines using a packet tunneling protocol. The tunneling protocol encapsulates the container packets using a host network address and forwards the packets to the destination machine. The global scope of the Swarm network is referred to as *swarm*. The global *swarm* network spans across the entire Swarm cluster. A container consumes a discrete service by querying the embedded DNS server to resolving its virtual IP address. A prerequisite for this is the query must be from a container attached to the same network as the container or service being looked up. Overlay networks can be isolated from each other by specifying the name resolution in the network scope. However, Docker containers can use more than one bridge and overlay network at one time.

Then, IP Virtual Server (IPVS) performs load balancing and forwards traffic to each container of the service. Swarm uses service routing mesh to route external incoming requests to the published tasks that constitute the service, as shown in Figure 2.5. The routing mesh of Docker Swarm comprises an ingress overlay network, *netfilter* rules, and IPVS. A service running in a Swarm cluster makes itself available to consumers external to the cluster through the publication of a port. Docker Swarm routes requests internally from one service to another via a virtual IP address, resolvable by the service name. Swarm uses layer 4 (transport layer) load balancing, by making use of the IPVS, which is a built-in feature of the Linux kernel. Packets destined for the virtual IP address of a Swarm service, are marked and forwarded according to the kernel *netfilter* rules. Through a combination of the *netfilter* rules and IPVS load balancing via the service's virtual IP, traffic is routed through the ingress network to all Docker containers. The configuration of Docker Swarm overlay is relatively simple but has a performance lower than that of native host networking (Zeng et al., 2017).

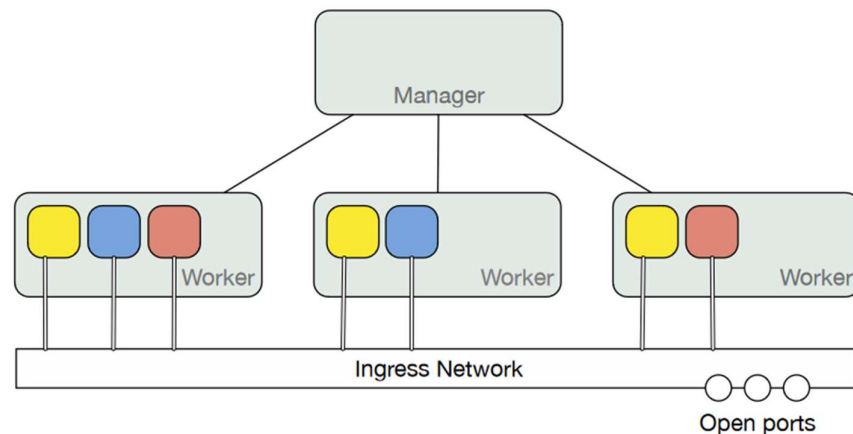


Figure 2.5 Docker Swarm Ingress Network (Church, 2019)

2.3 Fault Tolerance in IoT systems

2.3.1 Redundancy

Failures in edge cloud environments occur regularly as edge devices are sometimes very unstable. These devices are vulnerable to many types of failures such as power and hardware failures, which cause instability to the network and IoT services. Edge cloud systems for IoT data streaming applications require reliable and timely message delivery. Fault tolerance measures such as detection of failures, redundancy, and consistent recovery are required to accommodate reliable IoT services in edge cloud environments. The most common way of achieving fault tolerance is to incorporate redundancy in the system. In the event of faults, the system leverages redundancy to mask and tolerate faults, thus maintaining desired functionality and performance. Redundancy techniques involve replicating components in the application through redundancy to eliminate single points of failure.

Typical redundancy techniques are redundant hardware, software replicas, and distributed networks. A redundant hardware solution consists of a set of loosely coupled servers that are self-contained and able to detect faults on other nodes. For example, instead of having a single processor, two or more processors are used to perform the same function. Static hardware redundancy is mainly used to immediately mask failure. Dynamic hardware redundancy involves adaptively activating redundant components when the primary component fails. Karthikeya et al. (2016) propose a fault-tolerant algorithm, which determines the minimum number of necessary gateways to provide redundancy for edge servers in a smart city application. The proposed algorithm

detects gateways and link failures to create extra redundant routes between gateways. Hardware redundancy also facilitates failover processing to take over the operation of the failed servers. Failover processing involves restarting the application, initialization of the restarted process, and rollback recovery (Su et al., 2014). Health checks and error detections of hardware and software is required to perform failover.

Software redundancy involves replicating software components and splitting the application into independent components, to tolerate crash failures (Birman, 2012). For data streaming applications, it is important to have reliable local fault-tolerant solutions at the edge of network through edge computing. Through edge computing, centralized computing elements and application logic are offloaded from centralized nodes to edge servers. Javed et al. (2018) address local fault tolerance and data resiliency by implementing a fault-tolerant data pipeline via Kafka, to provide extra redundancy to the edge-cloud environment. Sen and Balasubramanian (2018) propose the use of data redundancy to resolve the issue of resiliency in the MQTT broker system. All of the important states such as client sessions and routing information are maintained in a separate shared cache. The use of a shared cache improves the fault tolerance of the system because the failure of broker components will not affect the data.

For networking redundancy, Gia et al. (2015) address network-level fault tolerance using backup routes to maintain connectivity between all the nodes in a wireless sensor network (WSN). Network virtualization via Software Defined Network (SDN) approach is another used to improve resiliency in the

distributed network (Gonzalez et al., 2016). This approach focuses on a global view of the network infrastructure spanning from edge IoT devices to the central cloud. SDN can provide independent forwarding and routing decisions at various points of SDN nodes in the fog and edge layer of the network. When one of the SDN nodes fails, the redundant SDN nodes in the network will build up alternative network paths for data flow.

Su et al. (2014) present the WuKong framework, which is a decentralized ring topology where services are delegated from a failed device to a redundant device in the ring, to recover the lost service. The WuKong middleware facilitates the failover of identical services among heterogeneous devices. The proposed framework is designed for sensor networks. One master device manages and handles failures of all worker devices and their services. However, the proposed fault tolerance mechanism does not consider the failure of the master device and the gateway, only the failure of the worker devices.

2.3.2 Checkpointing

It is important to capture runtime progress so they are not lost in the event of failure. Checkpointing is a recovery-based fault tolerance approach used for recovery after a system failure. Checkpointing is used for storing the state of periodically advancing applications such as file systems and databases. A checkpoint is done after every critical change made to the system. When a process fails, rather than restarts from the beginning, it starts the task from the most recent checkpoint. The restarted process starts the recovery process and replays all of the checkpoints done between the previous state and the time of

failure. Most checkpointing techniques are applied in stateful applications to ensure reliability and continuity of services (Khunteta and Praveen, 2010). Checkpointing technique with rollback recovery is used to recover from transient faults in real-time embedded systems (Saraswat et al., 2010). A transient fault is a type of fault that happens once during runtime and will never happen again afterward.

Coordinated checkpointing is a centralized checkpointing technique that uses an atomic commitment protocol to ensure that the global checkpoint is consistent. The atomic commitment protocol performs a distributed atomic transaction for every turn of checkpointing. A distributed atomic transaction involves a querying phase, an agreement phase, and a commit phase. Uncoordinated checkpointing is a decentralized fault-tolerant technique where nodes construct their checkpoint individually (Guermouche et al., 2011). However, this causes the system to have a low level of global consistency. A global consistent checkpoint must be computed from available checkpoints to prevent a domino effect. Checkpointing techniques depend on global rollback for consistency restoration. Global rollbacks lead to systemwide aftermaths since all entities have to rollback after each failure (Guermouche et al., 2011). Ozeer et al. (2018) present an uncoordinated checkpoint with message logging to save the state of IoT devices in the fog environment. In the framework, a set of data representing the application state is periodically saved in each node and restored during an occurrence of failure.

2.3.3 Container Service Migration

For many virtualized applications, job migration is a strategy to keep alive important services even in a faulty scenario. If a machine fails, the job can be migrated to a different machine in the network. Job migration can also be used pre-emptively in which the application is constantly monitored on a feedback-loop mechanism (Devi and Saikia, 2014). Container migration via Docker is being used to preserve system liveliness but it is only for stateless applications (Ismail et al., 2015). In a distributed edge deployment, live container migration is a solution to reduce service downtime by moving containers between different physical machines without restarting the container. A container cluster requires a shared network infrastructure to support the reattachment of the container's network interface to a different location in the cluster network. Container live migration has been demonstrated working for stateless applications. However, the live migration technique is still unstable for stateful application as it is erroneous and will slow down the entire service on edge infrastructure (Kakakhel et al., 2018).

Deshpande and Liu (2017) proposed a Docker container service migration framework to transfer service containers in an edge cloud platform between edge nodes that consist of embedded single-board computers. The proposed framework prevents service from stopping by check-pointing live containers and restoring failed containers in redundant nodes. Container migration mechanism migrates containers across different hosts without disconnecting the clients. The memory file system and live network connections that are running on the hardware are transferred to another machine while

preserving the state of the container. When a host fails, the process freezes a container, saves the state of the container, and migrates it to a destined node, and resumes the state.

2.3.4 Load Balancing

Additionally, fault tolerance can be achieved through load balancing techniques. Applications can be deployed behind load balancers to mask failures. Load balancing can be implemented as hardware, software, or network. A load balancer distributes workload to server nodes to improve response time and throughput of a system (Rao et al., 2003). A load-balancing algorithm should have fault tolerance and fault detection ability. This means that it should perform load balancing accordingly despite node failures and redirects traffic to healthy nodes.

2.4 Summary

In this chapter, the concepts related to edge cloud systems and container virtualization are discussed. Section 2.1 presents the drawback of cloud-based IoT and the comparison of various edge and cloud technologies in the IoT paradigm. Cloud computing is inappropriate for latency-sensitive IoT applications due to its high and unstable latency, and large bandwidth demands. Edge computing moves some of the computing resources closer to end-users to reduce end-to-end transfer latency and avoid a large volume of information flow to the central cloud. The hardware of edge computing device has the processing power, memory, and storage capabilities lower than that of typical mainframe servers in the cloud. To compensate for the lack of computing power, smaller

edge devices such as the Raspberry Pi SBCs are combined into a distributed computer cluster.

Section 2.2 presents an overview of edge-based orchestration using Docker containers and SBCs. Docker container is suitable for edge computing applications because of its lightweight-ness and ease of deployment. This proposed broker system in this work implements a microservice-based MQTT broker cluster within the Docker Swarm cluster. The application services can be automatically deployed and scheduled using Docker Swarm.

Section 2.3 presents various fault-tolerant approaches used in IoT systems. Hardware and software redundancy involved replicating hardware and software components and splitting the system into independent components, to tolerate crash failures. Network redundancy focuses on maintaining routing and stable connectivity between nodes via backup routes. Checkpointing techniques restore the system to the correct state after a system failure. Container migration mechanism migrates containers between different nodes in a distributed network to provide availability for IoT services.

CHAPTER 3

PUBLISH-SUBSCRIBE SYSTEM

This chapter presents an overview of the publish-subscribe communication paradigm. Section 3.1 presents the MQTT protocol and its communication patterns. Section 3.2 reviews various distributed MQTT systems present in the industry and the literature. Section 3.3 presents an overview of scalability and fault tolerance on generic distributed publish-subscribe systems. Section 3.4 outlines the fault-tolerant approaches used in distributed publish-subscribe systems.

The publish-subscribe system is a key technology for information dissemination in the IoT paradigm. The publish-subscribe paradigm differs from conventional client-server architecture, in a way that both endpoints do not communicate directly with each other. The publish-subscribe model describes a loosely coupled information dissemination middleware for message exchanges. Each participant in a publish-subscribe system can be either a publisher or a subscriber of information. Client endpoints do not need information about each other to work correctly as communications between clients are managed by a message broker. Figure 3.1 illustrates the publish-subscribe pattern with a central broker.

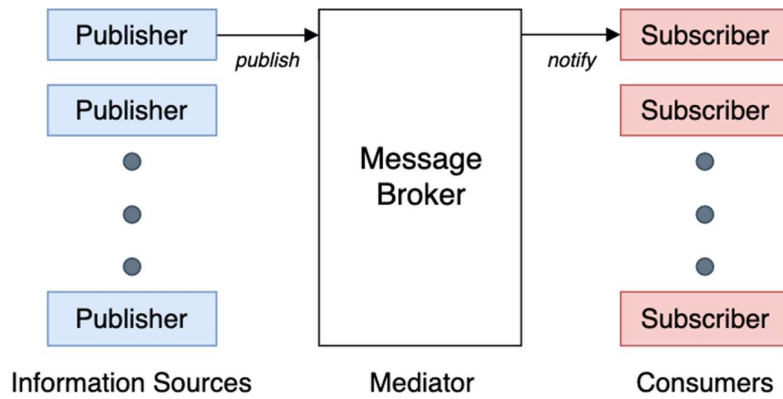


Figure 3.1 Publish-subscribe sequence

To perform publication or subscription, clients use a publish-subscribe API to connect to the message broker. Publish operation is invoked by publishers to produce messages and inject publications into the server. Subscribers register their interest to the broker to receive messages according to the topic of interest or based on constraints over the publishing content. Each subscription is considered a filter on a set of published events. Subscribe and unsubscribe operations are invoked by subscribers to respectively declare or remove their interest in certain types of content. The broker accepts connections, stores subscriptions, and forwards matched publications to subscriptions.

Examples of broker-based protocols include MQTT, Advanced Message Queuing Protocol (AMQP), CoAP, and Java Message Service API. The many-to-many communication model of the publish-subscribe paradigm has the advantage of space and time decoupling. However, the central broker becomes a single point of failure and bottleneck for performance. Network concentration and load peaks can potentially slow down message delivery. The broker server must scale horizontally to avoid these issues.

This research work mainly focuses on topic-based publish-subscribe systems. Topic-based publish-subscribe systems are widely popular in the industry and open source communities. A topic-based publish-subscribe system partitions event-space into separate channels, known as topics. A subscriber uses a topic string as a predicate to register its interest in a topic. Publishers tag their payload with a topic as the metadata of the message. Subscriptions are stored as a set of subscribers for every topic entry. When a publication needs to be matched, one simply needs to find its topic and obtain the corresponding set of subscribers. Some topic-based publish-subscribe systems such as MQTT support the organization of topics in a hierarchy. In that case, publications are matched against subscriptions with an equal topic or with equal upper-level hierarchies.

3.1 MQTT

Message Queue Telemetry Transport (MQTT) is a topic-based messaging protocol built on top of TCP/IP protocol. The MQTT protocol is useful for IoT data exchange in constrained environments due to its lightweight design and minimal overhead. An MQTT client interacts with the MQTT broker through the MQTT interface. The MQTT broker mediates data between publishers and subscribers. The broker filters all incoming messages and correctly distributes them to all subscribers.

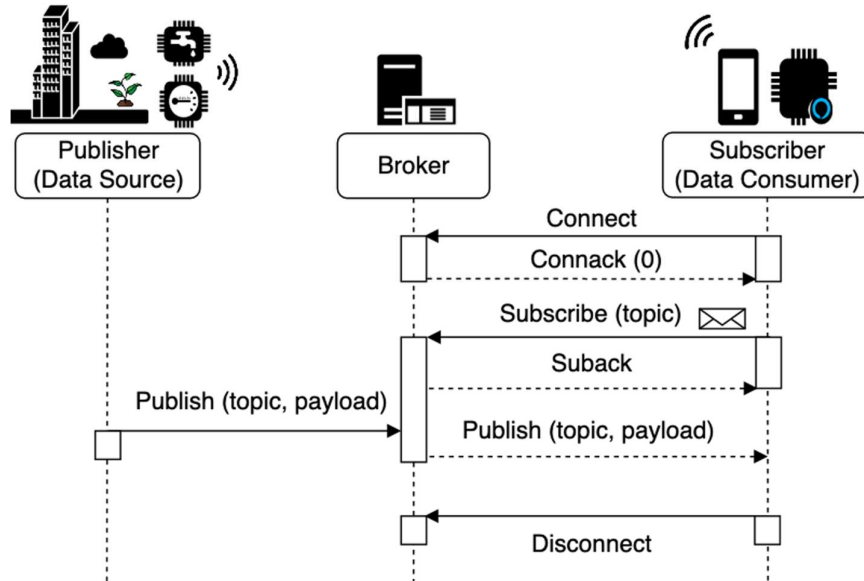


Figure 3.2 MQTT publish-subscribe sequence

Figure 3.2 illustrates an example of the publish and subscribe operation of the MQTT protocol. First, the subscriber sends a *CONNECT* message to a broker. The broker replies with a *CONNACK* message to establish a connection between the subscriber and the broker. The subscriber then sends a *SUBSCRIBE* message to the broker to register its topics of interest to the broker. The publisher then sends a *PUBLISH* message, with a specific topic, to the broker. If the topic is included in the registered topics of interest, this message is forwarded to the subscriber by the broker. The subscriber sends a *DISCONNECT* message to the broker to terminate the connection. MQTT supports hierarchical topics in the form of *topic/sub-topic/sub-sub-topic* path. In MQTT client and server maintain a connection during communication. However, the central broker configuration presents a bottleneck that results in a broker queuing delay when a large scale of IoT devices are connected to the MQTT broker (Xu et al., 2016). This could also potentially result in a single

point of failure if the broker crashes. Broker crashes can cause the loss of all MQTT states maintained by the broker.

The MQTT protocol has three levels of quality of service (QoS) to serve the reliability of message delivery. QoS 0 is an “*at-most-once*” message delivery that delivers on a best-effort basis, without confirmation on message reception. In some applications where sensor value does not change significantly over a long period, this QoS can be used because losing data occasionally is not critical for overall sensor value is still understandable. The reliability of QoS 0 is dependent on the TCP/IP protocol where messages will be lost if a TCP session is broken.

QoS 1 is an “*at-least-once*” message delivery that guarantees message arrival to the receiver. The receiver must send an acknowledgment to confirm its message reception. If the connection between the client and the broker is broken, the client stores a few messages in the buffer and resend them when the session comes back. This quality of service guarantees the delivery of sent messages but messages can be delivered more than once. QoS 2 ensures that the message will be delivered exactly once without duplication. The client and broker perform a four-way handshake to confirm the message reception on both sides. This QoS level has the most overhead because it requires 4 hops of transmission to complete a message transaction.

0	1	2	3	4	5	6	7
Message Type				DUP	QoS Level		Retain
Remaining Length (1-4 bytes)							
Variable Length Header (Optional)							
Variable Length Message Payload (Optional)							

Figure 3.3 MQTT message format (Tang et al., 2013)

Figure 3.3 shows the MQTT message format. Message Type refers to the type of message. These message types are *CONNECT*, *CONNACK*, *PUBLISH*, *SUBSCRIBE*, etc. DUP indicates that the message duplication flag is used when the broker processes the message. *QoS* field specifies the quality of service. *Retain* field stands for message retention. This means that any message can be retained and published as the first message to a new subscriber. *Remaining Length* field indicates the remaining length of the message. The rest of the message field is associated with the message payload.

3.2 Commercial and Open Source MQTT Brokers

MQTT is very popular in M2M and IoT applications. Facebook has been using MQTT as the communication protocol for its messenger application (Zhang, 2011). IoT platforms such as Amazon IoT and Microsoft Azure also provide services that use MQTT as their communication interface. AWS IoT, HiveMQ, and IBM Bluemix are examples of enterprise-ready, cloud-based MQTT brokers. However, implementations of these cloud-based brokers are limited for edge servers because they are closed systems. Table 3.1 lists available commercial and open source MQTT broker servers and their cluster support.

Table 3.1 MQTT broker cluster implementations (Mishra, 2019)

Broker	Clustering support	MQTT bridge	IoT bridge support
jmqttd	Multi-host clustering	✓	✓
ActiveMQ	Inter-broker bridges	✓	✓
emqttd	Multi-host clustering	✓	✓
flespi	Inter-broker MQ/job queues	✓	✓
HiveMQ	Multi-host clustering	✓	✓
JoramMQ	Distributed hierarchical brokers	✓	✓
mosquitto	No	✓	✓
RabbitMQ	Queue mirroring	✓	✓
VerneMQ	Multi-host clustering	✓	✓

The scale of the local sensor environment in edge-based IoT is expected to serve up to tens of thousands of concurrent clients while providing adequate latency quality. Unfortunately, standard MQTT brokers have poor scalability when the network load is heavy. Open-source MQTT brokers such as Mosquitto (A Light, 2017) suffer from single-point-of-failure that results in a complete breakdown of the system if the broker fails. Data exchanges by an overwhelming amount of IoT devices can cause large queuing delays for a single broker (Xu et al., 2016). Overhead of queuing delay may be negligible for a powerful machine, but for resource-constrained systems, it is very difficult to handle the significant portion of message overhead. Instead of improving a single server, the broker can be horizontally scaled adding extra brokers to the work pool. The distributed system can use a load balancer as a single-entry point for all client communications. This allows the clients to perceive the system as a single logical broker, thus providing user transparency.

Standard MQTT brokers do not offer a way to group a cluster of MQTT brokers with similar topics. To operate with any existing MQTT brokers, this work uses an adapted middleware layer that sits between the MQTT brokers and MQTT clients so that the clients can use the existing MQTT infrastructure while obtaining a highly available service. The dissertation work addresses the fault-tolerance and performance aspects of a topic-based distribution system that is composed of several MQTT brokers. Standard MQTT brokers like Mosquitto provide a basic mechanism for edge computing setups. The proposed broker system extends the MQTT broker using a clustering approach, in which a set of MQTT brokers are grouped into a cluster, and a load balancer is used to distribute incoming requests.

MQTT clustering can be achieved by using broker bridging. Many brokers can be configured at deployment time to bridge their topics tree structure to a centralized bridge broker (Schmitt et al., 2018). Thus, the bridged brokers can exchange messages from the bridged remote broker through the bridge configuration. However, this approach is very static and has limited scalability. The communication overhead between bridges is significant because bridging involves propagating messages between all MQTT clients that are connected to different bridges. If a broker bridge crashes, the MQTT service stops functioning correctly as a distributed system, and some messages are lost because there is no failover mechanism for bridging. Besides that, bridging brokers can potentially trap messages in an endless loop of transmission between broker nodes (Redondi et al., 2019). This can deplete the bridged system as the brokers repeatedly send shared messages over the bridge links.

Multiple works suggest the deployment of brokers in multiple distributed edge networks, across several geographical locations (Banno et al., 2017; Rausch et al., 2018; Park et al., 2018). However, this work will implement MQTT servers on a single local cluster.

Rausch et al. (2018) present EMMA, which uses bridging tables to dynamic link MQTT brokers in an MQTT bridge. The EMMA framework reduces end-to-end latency from client to server by dynamically rerouting the connections based on proximity and QoS index of the connection. EMMA changes the connection between clients and MQTT brokers through buffering gateways that reside in multiple locations. These gateways allow MQTT clients to transparently connect to the system. The gateways tunnel MQTT traffic and provide a buffering mechanism during a reconnection process to a different broker.

JoramMQ introduces two types of distributed brokers, clustered and tree-based brokers (Scalagent, 2014). Clustered brokers broadcast all publications messages to all other brokers when they received a publication. Tree-based brokers are distributed in the network based on the hierarchy of subscriptions topic. They transfer messages between the upper and lower tier of brokers. HiveMQ implements subscription topic sharing among clustered brokers, forwarding publication messages only to brokers that have the same subscriptions (Hivemq, 2019).

DM-MQTT (Park et al., 2018) is an edge-based MQTT broker system that uses the multicast mechanism of Software Defined Network (SDN) to transfer data as MQTT packets between MQTT brokers distributed across different edge networks. SDN switches are distributed across the edge cloud network to interconnect all edge and cloud brokers. DM-MQTT reduces latency delay by using a bi-directional SDN approach. The SDN controller forms multicast groups classified by topics and QoS levels collected in the central broker. The SDN controller uses this information to form a multicast path along with the SDN switches, that interconnect edge brokers residing in different edge networks. The multicast paths are formed between MQTT brokers with the same subscribed topic, which entirely bypasses the centralized broker.

The ILDM (Banno et al., 2017) framework introduces the use of an intermediary relay node to support distributed MQTT messaging between edge servers. For each edge server, the ILDM node is placed between the MQTT broker and the clients. The ILDM node connects heterogeneous brokers that are located on multiple edge networks. The routing algorithm used in ILDM is like a tree-based routing approach that its publication forwarding process routes along a fixed propagation path towards interested subscribers. However, the message forwarding process requires multiple forwarding hops between disperse edge servers and also indirect data flow between the local ILDM and broker node. This increases forwarding delay and causes network congestion. The authors showed that in the absence of failure, their system can increase the throughput by approximately 2 to 4 times without any loss of message. This research work uses a similar concept of deploying an intermediate layer of

software to coordinate between multiple MQTT brokers. However, this research work addresses local fault tolerance for the MQTT broker cluster where all MQTT client devices are concentrated around a single access edge node. Figure 3.4 depicts the differences between a distributed broker network and a broker cluster in a local area network. The target environment depicted in Figure 3.4(b) consists of only edge servers in a single local cluster, which permits more reliability in terms of message delivery, and less delay between cluster nodes (Rooney et al., 2005). In ILDM data is transferred as a unicast MQTT-like operation and may travel across multiple relay brokers which increases transmission delay. On the other hand, DM-MQTT delegates the data transmission function to the SDN networking module which enables multicast functionality between brokers.

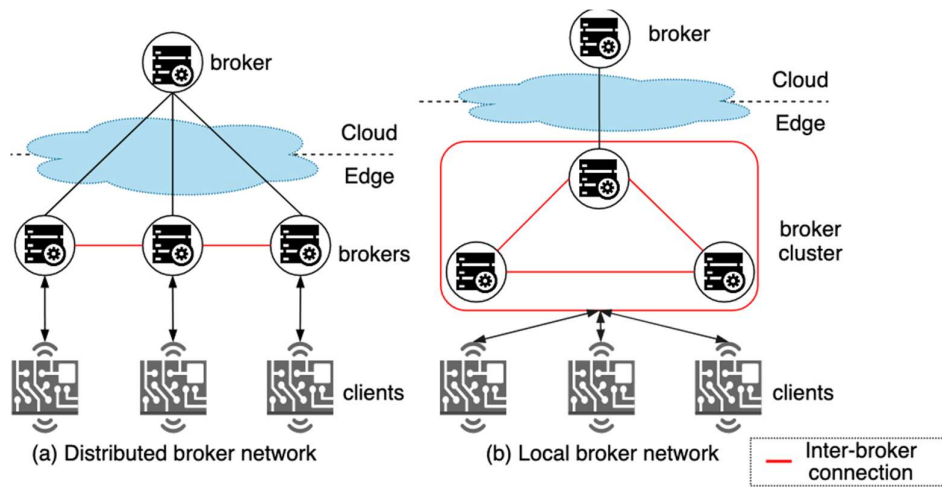


Figure 3.4 Distributed and local area broker network

Nucleus (Sen and Balasubramanian, 2018) improves fault tolerance by splitting the system into stateless MQTT brokers and a separate shared cache to maintain its routing state. This way the broker can fail anytime without affecting the state information. The broker component uses the elasticity mechanism of

Kubernetes to dynamically scale the number of stateless brokers according to the CPU utilization index. However, the distributed data store introduces additional access delay, which increases overall latency for message delivery. A similar architecture is also presented in MigratoryData, where worker I/O threads and state information are separated into discrete components (Rotaru et al., 2017). MigratoryData partition incoming subscriptions by splitting the subscribers among all servers, irrespective of their subscribed topic.

3.3 Distributed Publish-subscribe System

Distributed event notification services are implemented through a set of event brokers that forms a message broker overlay. For a generic publish-subscribe system, the system stack generally consists of three layers: the overlay infrastructure, event routing, and subscription matching algorithm. The overlay infrastructure maintains the routing information and arrangement of member nodes. Event routing utilizes the underlying overlay infrastructure to perform message delivery. Event routing will require a subscription matching process, which evaluates the matching function of a subscription to an event. In the past years, several research contributions have focused on reliability and scalability in internet-scale, distributed publish-subscribe systems. These systems use an application-level overlay network to preserve scalability and fault tolerance. However, most of these schemes employ their own standards instead of implementing established IoT protocols like MQTT. Table 3.2 shows a comparison of various generic internet-scale publish-subscribe systems. The following subsections present various scalability and fault-tolerant approaches to publish-subscribe systems found in the literature.

3.3.1 Distributed Routing Mechanism

In a distributed event notification network, event routing involves the global aspect of traversing messages to relevant brokers before reaching interested subscribers. This subsection discusses available forms of event routing.

Table 3.2 Summary of generic publish-subscribe systems (*Setty et al., 2012*)

	Architecture	Overlay Structure	Subscription Management	Event Dissemination	Fault tolerance
Siena	Brokers on top of Mesh	None / mesh	Subscription state at each node	Broadcasting states	Best Effort
Scribe	Decentralized	Pastry DHT	Rendezvous node	Multicast Tree	No subscription persistence
Tera	Decentralized	Gossip Based Overlay	Overlay Per Topic	Random Walks & Flooding	Best Effort
Poldercast	Decentralized	Ring based DHT, Vicity, Cyclon	Ring Per Topic	Ring Overlay Routing	High Churn resistant
Vitis	Decentralized	Hybrid Overlay	Rendezvous & Overlay Per Topic & Inter-topic Gateway	Scoped flooding	Best Effort

3.3.1.1 Event Flooding

A straightforward event routing approach is to broadcast publications to all nodes across the publish-subscribe network. This means that each subscribing node in the system will match against all publications. This approach is straightforward and has no memory overhead. However, it does not scale well in terms of the number of message transfers (Eugster et al., 2003), since publications are always sent to all brokers regardless of subscriptions they hold.

Subscription flooding involves diffusing each subscription into each broker to build a locally complete subscription table. Each node knows all the subscriptions of the entire system so that the event can reach subscribing nodes in a single hop. This approach incurs large memory overhead when the total number of subscribers is high, but message overhead is optimal (Eugster et al., 2003). Subscription flooding is impractical for applications where the subscriber fluctuation rate is high because each node must propagate changes to all other nodes in a completely connected overlay. Subscription flooding also needs a convergence period to stabilize the routing state after subscriptions and un-subscriptions. The system can only guarantee the delivery of matching publications to all registered subscriptions after this period. Event propagation based on an incomplete routing state before state convergence will inevitably lead to publication loss. In an asynchronous model, the publish-subscribe system must deliver publications issued within a certain period following the time of subscription arrival.

The message overhead of publication flooding depends on the network size because all publications are forwarded to reach every other broker node in the network. Subscription-based flooding builds up a routing pattern in which publications are only forwarded to interest subscribers, thus do not depend on the network size of the group. A recent work that presents flooding techniques is ILDM (Banno et al., 2017).

3.3.1.2 Selective Routing

Selective routing presents a middle ground in terms of memory and message overhead as compared to event flooding. Selective routing reduces the message overhead by only allowing a certain subset of the nodes to involve in routing a specific topic. Selective routing algorithms save more network resources because an event must be transmitted only to a restricted portion of subscribers. In filter-based routing (Baltoni et al., 2009), subscriptions are partially diffused in the system and used to build routing paths. This approach maintains routing information to construct routing paths that connect publishers to all interested subscribers. The routing information of a node is associated with each of its neighbors in the overlay and the set of subscribing nodes that are reachable through the neighbor. In contrast to subscription flooding, each node only communicates with its neighbor nodes, thus reducing memory overhead. The disadvantage of a filter-based routing scheme is that nodes arranged in the edge of the tree structure are not used for forwarding. This results in unnecessary long forwarding paths, which causes a large delivery delay (Siegemund et al., 2015).

Rendezvous routing is based on rendezvous relay brokers in the overlay network and two sets of functions, to associate respectively subscriptions and publications to broker nodes. The subscription function $\mathcal{S}(N)$ returns a set of nodes that are responsible for storing the subscription s as well as forwards events to all matching subscribers. The event function $EN(e)$ returns a set of nodes that must receive the event e to match it against the subscription they store. This approach uses a controlled subscription distribution to better load

balance the subscriptions for storage and management. All subscriptions that match the same events will be hosted by the same node to avoid a redundant matching in several different nodes. One drawback of the rendezvous-based solution is the reliance on central rendezvous nodes for the correct operation of distributed event routing. The rendezvous node can quickly become a hot spot for popular topics. Moreover, rendezvous-based routing does not handle well dynamicity. When a new node joins the system, the whole routing structure and rendezvous points must be rearranged among nodes. If many nodes join at nearly the same time, rendezvous routing will miss some of the message deliveries because the routing structure takes a long time to converge the topic paths. Rendezvous-based routing is commonly used on top of DHT overlays such as in Scribe (Castro et al., 2002), Bayeux (Zhuang et al., 2001).

3.3.1.3 Gossip-based Routing

Gossip-based event propagation is used to cope with dynamism and to reduce the effect of node churn within a publish-subscribe environment (Baldoni et al., 2009). In the gossip protocol, each node maintains only a partial view of the subscriptions of its neighbor in the group and propagates event hop-by-hop through gossiping to its view. Eugster and Guerraoui (2002) propose an informed gossip protocol that avoids gossiping a message to uninterested subscribers. The proposed gossip protocol organizes groups in hierarchies according to the physical proximity of nodes. PopSub (Salehi et al., 2017) reduces the message overhead of the publish-subscribe system by propagating less popular publications through gossiping.

Deterministic technique over multicast trees is fast and has minimal duplication during the stable period but it is fragile to node churn, whereas gossip-based dissemination is more robust but does not guarantee deterministic delivery (Baldoni et al., 2009). There is also a probability that uninterested nodes will receive duplicated messages during event propagation, thus creating unnecessary network traffic.

3.3.1.4 Clustering Topics

The correctness of the state of multicast trees is easily disrupted when the subscription fluctuation rate and dynamism of overlay topology are high. The routing tree needs to be continuously updated to guarantee correct message delivery. Hence, the dissemination of events incurs additional overhead in maintaining the routing tree. The topic clustering approach is introduced to achieve space and time efficiency along routing paths (Milo et al., 2007). Milo et al. (2007) present a dynamic topic clustering mechanism that groups per-topic peers to reduce maintenance and relay overhead. The authors adopt a cost-benefit analysis to dynamically merge or split two topic clusters. The authors define the overall cost as the sum of maintenance cost and dissemination cost for the publish-subscribe system. The topic clustering mechanism only takes place if the overall cost is reduced when merging two clusters. The clustering algorithm groups topics together into topic clusters. Each topic cluster represents a single multicast group. When the event reaches one member of the topic cluster, event forwarding only involves nodes correspond to the cluster to confine the traffic (Baldoni et al., 2009).

3.3.2 Overlay Infrastructure

Publish-subscribe brokers sit on top of an application-level overlay that characterizes the organization of nodes and the functionality of each node. Data dissemination and subscription matching happen over an overlay network of publish-subscribe nodes. Overlay infrastructure provides useful communication and data structure primitives on the application layer. Examples of overlay infrastructures are structured and unstructured peer-to-peer overlay, broker-based overlay, skip-graph overlay, and cloud-based overlay.

3.3.2.1 Broker-based Overlay

In a broker-based overlay, each broker forms an application-level overlay and communicates through an underlying transport protocol. Unlike peer-to-peer overlays, broker-based overlays are typically deployed with dedicated brokers in the network as intermediaries between publishers and subscribers. Clients can access the publish-subscribe system through any broker. In general, each broker stores only a part of all the subscriptions in the system. Connections between broker nodes are application layer links. The server network can be organized in many types of overlays that are typical in peer-to-peer networks, such as ring, hierarchical, etc. Network topology changes are rare. A broker-based overlay is mainly used to facilitate failed brokers and the addition of a new broker. Examples of the broker overlay publish-subscribe systems used in past works are TIB/RV (Oki et al., 1993), and Siena (Carzaniga et al., 2003), and Hermes (Pietzuch and Bacon, 2002). Siena is content-type broker architecture that uses subscription advertisements to aggregate events for each subscriber group. Event brokers keep track of routing

information to efficiently match publications with brokers with similar subscriptions. Siena has limited scalability because it relies on a global broadcast mechanism to disseminate publications. Hermes (Pietzuch and Bacon, 2002) is a type-based publish-subscribe system implemented using middleware to route events across the broker network. For fault tolerance, Hermes uses replicated rendezvous broker nodes, as meeting points, for other event brokers to advertise their subscriptions and publications. Also, Hermes periodically refreshes the state of event brokers to adapt to broker failures.

3.3.2.2 Peer-to-peer Overlay

Peer-to-peer (P2P) approaches do not require centralized architecture. In peer-to-peer publish-subscribe systems, the message dissemination mechanism is implemented on top of an overlay network that joins all messaging nodes together. Peer-to-peer based publish-subscribe systems are generally classified as a structured overlay and unstructured overlay.

3.3.2.3 Structured Peer-to-peer Overlay

Structured P2P overlay infrastructure is based on per-topic multicast trees on top of P2P DHT overlays. A P2P node exploits P2P communication primitives of underlying DHT to implement its event routing algorithm. DHT based publish-subscribe systems are commonly used in conjunction with rendezvous-based routing schemes.

Scribe (Castro et al., 2002) is a topic-based publish-subscribe system that uses P2P communication primitives of Pastry DHT to implement rendezvous routing. Scribe uses Pastry DHT to locate an active rendezvous node that holds nodes subscribe to a topic, in order to build a multicast tree that connects from publishers to subscribers. Publications sent to the rendezvous node are then forwarded in a reversed manner along the multicast tree to corresponding subscribers. Scribe uses the heartbeat detection method to detect faults in DHT nodes and to maintain the completeness of the multicast trees. Each parent node periodically sends a heartbeat message to its children. If an active child node found out that its parent is faulty, the children node will get a new active parent by issuing a re-subscribe message for the topic to repair the broken multicast tree.

Dynatops (Zhao et al., 2013) is a topic-based publish-subscribe system that builds its overlay on top of Chord DHT (Stoica et al., 2003). Dynatops uses a similarity grouping algorithm to map subscribers into multiple groups of brokers, based on similar interest to nearby brokers. This reduces the overlay management overhead of brokers in the network. The overlay of Dynatops forms multiple per-topic multicast trees to propagate publications to all interested nodes. For each subscription whose topic matches the publication, the publication is then propagated hop-by-hop along the topic tree in the reverse direction of the subscription until it reaches the subscribers. For consistency and fault tolerance, Dynatops uses a centralized reconfiguration mechanism to monitor the rate of outgoing publications and change of subscriptions and restructure the overlay according to these changes. Dynamic reconfiguration of

overlay also minimizes the number of unrelated relay brokers along the publication routing path.

Structured publish-subscribe overlays are scalable and robust with respect to node degrees. However, this approach incurs more propagation overhead because event forwarding also traverses through uninterested nodes (Rahimian et al., 2011). Structured overlays can be well suited for sensor networks with occasional node churn, frequent failures, and limited node reliability. One drawback of DHT based publish-subscribe systems is the dependence on rendezvous points which lead to multiple single points of failure. Rendezvous points also can become a hotspot for concurrent events on the same topic.

In most IoT applications, many types of exhaust data have been occupying most of the IoT data distribution trends (Banno et al., 2015). According to these trends, most topics in IoT topic-based publish-subscribe usually have very few subscribers and these data have low value most of the time. Routing schemes presented in most publish-subscribe systems such as Scribe and Bayeux waste network resources by excessively circulating a large amount of low-value data around routing paths. Banno et al. (2015) present a design of a relay-free overlay network for topic-based publish-subscribe systems. The proposed framework detects low-value data among the subscriptions on top of a DHT overlay to minimize unnecessary forwarding of published messages. The authors use Multi-key skip graph to construct the application layer overlay. A multi-key skip graph is a distributed data structure

that is used to quickly insert and search resources among peer nodes based on provided keys. The proposed framework uses peer information in the multi-key skip graph to determine the shortest routing path in order to reduce the length of routing path during message propagation. The proposed framework is resistant to failure and subscription fluctuation because the multi-key skip graph can quickly detect the presence of subscribers by querying into subgraphs in the overlay. The skip graph overlay maintains information on neighbors' state and builds up redundant links to bypass a failed node. In an application of distributed edge-based publish-subscribe, the authors show that the multi-key skip graph approach can reduce propagation path but requires larger routing tables.

3.3.2.4 Unstructured Peer-to-peer Overlay

A tree-based structure can be weak against node failure. Unstructured overlay topology is used to organize nodes in one flat or hierarchical small diameter network to minimize the effect of node failures (Baldoni et al., 2007). The publish-subscribe system can continuously repair the overlay topology even if node failure occurs. The unstructured overlay uses a gossip-based and uniform sampling protocol to periodically update each local view, about participant interests at each node and swaps random view entries between randomly chosen nodes.

Tera (Baldoni et al., 2007) uses a hierarchical structure to implement a topic-based publish-subscribe based on a uniform peer sampling service. Tera uses interest clustering by constructing topic overlay networks for each topic which includes all nodes subscribed to that topic. The inter-cluster routing

mechanism performs a random walk that stops at first a broker that holds an access point for the target topic. Once a subscriber received an event, it broadcasts the event to all nodes in the sub-cluster. The probabilistic gossip algorithms used in the system are resistant to both subscription fluctuations and node failures.

Poldercast (Setty et al., 2012) uses probabilistic event propagation over a gossip-based dissemination overlay to maintain application-level topic rings. Each peer node uses gossip to connect to other peers with the same topic to form an interconnected ring overlay between all subscribing peers. Poldercast exploits correlation within subscriptions by reusing the same shortcut links between multiple rings to minimize the number of links maintained in each node and hence reduces average event propagation paths.

3.3.2.5 Hybrid Overlay

Vitis (Rahimian et al., 2011) is a hybrid approach that extends the rendezvous routing on top of an unstructured overlay of peers. Vitis uses gossip-based peer sampling to build sub-clusters covering all subscribing nodes with the same topic to reduce the number of intermediary nodes along the routing path. However, due to the bounded node degree implemented in Vitis, disjoint overlays of similar topics can be formed. Therefore, gateway nodes are selected by nodes in each sub-cluster to connect to other sub-clusters for the same topic. One drawback of the Vitis framework is additional relay latency because sub-clusters need to be connected by additional gateway nodes, rendezvous nodes,

and relay nodes. The reliance on central nodes also makes it weak to dynamic node joining and failures.

3.3.2.6 Cloud-based Overlay

Cloud-based broker systems are single, flat-layer brokers generally deployed in the cloud. MQTT, ActiveMQ, and Amazon IoT are examples of single-hop cloud publish-subscribe systems used in IoT applications. BlueDove (Li et al., 2011) is an elastic cloud-based publish-subscribe system that can dynamically scale-out according to workload demands. BlueDove utilizes an attribute-based filtering model for subscriptions. Dynamoth (Gascon-Samson et al., 2015) is a cloud-based publish-subscribe system with single-hop routing. Dynamoth uses a hierarchical load balancer to dynamically redistributes topic channels among elastically replicated publish-subscribe brokers. The cloud-based publish-subscribe overlay has lower routing latency between brokers but presents higher end-to-end latency for many IoT applications.

3.4 Publish-subscribe Fault Tolerance

The occurrence of frequent faults is inevitable due to the distributed nature of the network. To tolerate failures, the distributed publish-subscribe system must have a built-in fault-tolerant mechanism to ensure that disruptions do not affect the operation of the message delivery in the long run. This section presents various approaches to tolerant faults in the publish-subscribe system.

3.4.1 Distributed State Recovery

Distributed checkpointing rolls back the entire state of a globally consistent state after failure (Carbone, Katsifodimos, et al., 2015). Lineage-based recovery is a logging technique used to recover lost states from the partial result in a distributed system. Lineage information such as the source of data and intermediate data flow paths of a message is sent along with the message payload to other nodes in the distributed system (Zaharia et al., 2013). During failure, the lineage information is used to recover from failure. However, both checkpointing and lineage-based recovery can take a longer time to recover if downtime is long, which is not suitable for latency-sensitive applications. Causal logging protocols send lineage information with each message in the data plane. On failure, the information on surviving nodes can be used to restore the system to a globally consistent state. Depending on the size of lineage information, causal logging may incur high runtime overheads. The size of lineage can be decreased to reduce runtime overhead. Lineage stash removes the overhead from the data path by asynchronously logging lineage information to a decentralized store (S. Wang et al., 2019).

Kazemzadeh and Jacobsen (2009) propose a recovery procedure that the publish-subscribe system must execute when a new or failed broker enters the system. Existing broker nodes in the system form a set of synchronization points to help the recovering broker to synchronize its routing state. Each synchronization point computes its local topology and subscription information and sends them over to the recovering broker. Each synchronization point uses guided messages to improve the message forwarding process that happens

concurrently with the recovery process. Each synchronization points attach a special header to every message sent to the recovering broker. The special header contains information about the destination of the message. This way the recovering broker can use the header information to determine its routing path without relying on the complete routing information of the system. The broker recovery approach presented in this dissertation work is based on the recovery procedure proposed by Kazemzadeh and Jacobsen (2009). However, this work uses a local network with a fully connected topology to implement the broker network. This eliminates the need to maintain intermediate neighbor nodes, which is suggested in the former approach.

3.4.2 Periodic Subscription

Every broker in a distributed publish-subscribe system stores a certain amount of soft state information required to facilitate event routing (Jerzak and Fetzer, 2009). This soft state information can be permanently lost in the event of failures. Jerzak and Fetzer (2009) address this problem in a periodic subscription approach, in which subscribers actively reissue their subscriptions towards the broker to maintain its soft-state information about client subscriptions. Each broker maintains a timestamp, which is refreshed every time, for each received subscription in its routing table. Scribe (Castro et al., 2002) and Bayeux (Zhuang et al., 2001) use subscription refresh to handle subscription fluctuations in which subscription enters are refreshed periodically before each expiry time. Subscriptions entries that are not renewed on-time are removed from the routing table. This approach ensures any incorrect routing state is discarded and reinstated by correct routing information which is

propagated following new subscription refreshes. The automatic expiration of subscriptions helps to eliminate the need for unsubscription advertisements since a subscribing node can unsubscribe just by stopping the subscription advertisement. By periodically flooding the subscriptions, this approach can prevent message loss and ensures that subscribers will eventually receive all the publications to their subscriptions. It also limits the effect of node crashes by periodically refreshing the subscription message, which guarantees subscription delivery when the system recovers. However, periodic subscriptions incur large bandwidth costs and do not consider publication loss.

3.4.3 Self-stabilization

The publish-subscribe systems must ensure that the shared state, which includes all registered subscriptions, is always consistent with the actual population of clients. Self-stabilization aims to eventually reach a stable and correct global state. Self-stabilization uses an approach similar to the periodic subscription method. Zhenhui Shen and Srikanta Tirthapura (2004) present a self-stabilization algorithm to maintain the consistent routing state in the distributed publish-subscribe system. Each node periodically swaps its local routing state with neighboring nodes and independently updates itself when the local routing state with neighboring nodes and independently updates itself when the local routing information is inconsistent with neighbor nodes. A mismatch between the local routing table and neighbor's subscriptions indicates new subscriptions or a potential loss of subscriptions at either of the neighbors. Inconsistencies between the tables lead to corrective actions at each local node. Each node makes corrections to its local routing state by adding missing

subscriptions and discarding stale subscriptions. The series of local corrections eventually restores the consistency among the distributed routing tables.

For wireless sensor networks, Siegemund et al. (2015) use the self-stabilization technique to handle subscriptions and un-subscriptions. The authors use a leasing approach with a time-to-live (TTL) countdown value to periodically refresh subscriptions and discard un-subscriptions due to faults. This is similar to a watchdog timer where the subscription routing table entry must rest a timer before it expires. Otherwise, it is assumed to have failed. The proposed system can deliver all messages correctly, without receiving duplicates on the client-side.

Also, DHT infrastructures such as Pastry and Tapestry implements self-configuration to adjust routing paths according to subscription and overlay information. Scribe and Bayeux are examples of publish-subscribe systems that are built on top of Pastry and Tapestry DHT respectively (Zhuang et al., 2001; Castro et al., 2002).

However, this approach has a large message overhead and lacks scalability mainly due to the periodic exchange of complete routing tables. Self-stabilization needs a convergence period to stabilize the routing state after node failures, subscriptions, and un-subscriptions. The system can only guarantee the delivery of matching publications to all registered subscriptions after this period. Updates to routing tables during recovery may take a certain period to converge the routing state. As a result of recovery action, the subscribers may

experience a short disruption in the system. Event propagation based on an incomplete routing state before state convergence will cause publication loss. Thus, the recovery period needs to be bounded and fast enough to prevent message losses.

3.4.4 Event Retransmission

Modern messaging solutions such as Kafka (Kreps et al., 2011), Flink (Carbone, Fóra, et al., 2015), and Spark Streaming (Zaharia et al., 2013) implement event retransmission to tolerate publication loss. Through the event retransmission approach, a broker resends an event whenever the event is discovered lost. One way of detecting publication loss is to exchange acknowledgment messages between each broker in the network (Kazemzadeh and Jacobsen, 2009; Salehi et al., 2016). When a broker receives a publication, the broker stores a buffer of the message and forwards it to neighbor nodes in forwarding paths. After receiving this publication message, nodes send an acknowledgment (ACK) message back to their upstream neighbors. Each broker will discard the event once it receives all ACK messages from the forwarding set. If the broker does not receive one or more ACK messages in its forwarding set, it retransmits the event to the forwarding path until all ACK messages are collected or until the buffered event becomes expired. A prerequisite of this approach is the fast recovery of failed brokers. An event may not be delivered to subscribers if a routing path is disconnected for too long.

Sourlas et al. (2009) recover publication lost using in-network caches that reside on message brokers along the propagation path of publication and show a high publication rate in presence of broker failure. Publications are cached at broker devices in the overlay so that multiple concurrent failures can be tolerated. The availability of multiple replicated copies of the publication cache eliminates the need for persistent storage and allows each cache to keep the data in memory. In-network caching reduces access delays as compared to caching in persistent storage. In-network caching also improves retransmission time because a lost publication is resent from the closest available cache node in the overlay.

C. Wang et al. (2019) propose FRAME to tolerate message loss in a real-time messaging system with the use of a primary-backup broker approach. The proposed framework presents a configurable scheduling and recovery method that handles different messages according to their fault-tolerance and latency requirements in an edge computing environment. Each subscriber is tagged with a latency deadline and a fault-tolerance level, for each of its topics. FRAME also uses earliest deadline first (EDF) scheduling to dispatch and deliver messages according to their deadlines. For message loss tolerance, the primary broker replicates a copy of the received published message to the backup broker. The backup broker uses periodic pooling to check the status of the primary broker. When a failure happens, the backup broker becomes the new primary broker. The system retains publisher messages during failure and prunes the backup messages during fault-free operation. When the primary broker fails, the publisher sends its message to the backup broker, and the subscriber recovers

the message from the backup broker. The evaluations confirm that the FRAME system can improve loss-tolerance performance and reduce the effect of latency during the fault recovery process. The peak latency due to fault recovery is 50ms for the FRAME system.

3.4.5 Redundant Paths

Redundant paths are used in the publish-subscribe overlay to guarantee correct message delivery in the presence of broker/link failures (Sherafat Kazemzadeh and Jacobsen, 2012). In this approach, the overlay topology adaptively creates disjoint routing paths to ensure that there is at least one correct path between the corresponding publisher and subscriber. A broker sends an update about the neighbor state to every other reachable broker whenever it detects a change in the live status of any broker. This way broker will be able to identify and build alternative routing paths towards all the subscribers. Thus, the redundant path can exclude paths that have failed brokers or broken links. Extra paths can be created based on the similarity of interest or vicinity between brokers (Setty et al., 2012). The downside of a redundant forwarding path is that it may consume high bandwidth and become very inefficient in the presence of node churn and frequent subscription changes. Also, this approach incurs additional notification latency under failure.

3.4.6 Consensus-based Publish-subscribe System

Consensus-based techniques are centralized approaches that focus on the consistency of distributed systems. Paxos-based pub/sub middleware (P2S) (Chang et al., 2014) is a crash tolerant Paxos-based publish-subscribe

middleware based on the Goxos Replication State Machine (RSM), which extends the original Paxos framework. Paxos is a consensus protocol that involves a set of processes that are trying to agree on a value. The implementation of P2S uses multiple instances of Paxos in the Goxos RSM to execute the publish-subscribe broker. A leader node is responsible to handle client requests and disseminate them to all replicas to reach a consensus. Each replica computes the ordering of competing requests and executes them in order. Upon receiving a subscription or un-subscription, the broker replicas query for consensus and update their local subscription table. The replicas deliver messages to each of the subscribed clients. The Paxos implementation tradeoffs system liveness for stronger consistency. If more than the maximum number of replicas fail, the system cannot make progress. This approach uses a centralized broker-based architecture and replication of a publish-subscribe broker to achieve resiliency. The fault-tolerant mechanism of P2S shows a tradeoff between performance overhead and reliability. The replicated approach has a maximum of 1.25% throughput reduction and 0.58 milliseconds end-to-end latency compared to its non-replicated counterpart.

3.4.7 Availability of Distributed Publish-subscribe Systems

The CAP theorem (Brewer, 2000) refers to consistency, availability, and partition tolerance. Many publish-subscribe systems satisfy the AP characteristic. The availability characteristics keep the continuation of the messaging service on a high level by using multiple messaging servers on top of the distributed application-layer overlay. Even though some of the message servers may fail, the remaining servers can keep the service going. Nevertheless,

consistency is also important to maintain reliable messaging. The distributed publish-subscribe system requires a consistent routing state to correctly process and deliver messages. Inconsistent routing states can lead to wrong behavior in message delivery. According to the CAP theorem, a distributed system can make a trade-off between consistency and availability characteristics. It is important to balance between consistency and availability for the distributed broker system. In this research work, the proposed broker cluster will continuously provide the MQTT service while gracefully handle broker failures. Even after a broker failure, the broker cluster continues to operate as normal, to avoid recovery or failover processing. This is because recovery processing incurs a delay to restart and initialize to a previous correct state (Egwutuoha et al., 2013), which can temporally stop the clients from using the service. To increase availability, the system discards the failed broker, and the load balancer routes the reconnected requests are to the remaining online brokers. The system achieves eventual consistency by periodically exchanging routing state with neighbor nodes.

CHAPTER 4

DESIGN AND IMPLEMENTATION

This chapter presents the design and implementation of the MQTT broker cluster system. In a typical cloud-based IoT application, data streaming, and communication between edge devices suffer from latency problems, which will create network congestion to the cloud. To resolve prior issues, the proposed broker system uses an SBC-based broker cluster and deploy it at the network edge. The broker cluster extends the cloud computing platform to the edge of the network. An edge-cloud integration layer is deployed to combine both the cloud and IoT environments. The MQTT edge server provides communication services and locally processes selected information before transmitting the information to the cloud, thus reducing overall network traffic. The implementation targets a clustered set of Raspberry Pi SBCs to deploy MQTT services in an edge-cloud architecture. The SBC cluster runs Docker Swarm to orchestrate Docker-based containers and services. This chapter presents the core components of the middleware platform for the edge cloud MQTT application.

4.1 Broker Cluster Architecture

This section presents the system design and features of the MQTT broker cluster. The distributed system comprises a collection of independent devices that communicate through message passing. The MQTT broker system consists of several application services distributed across the entire cluster. Each

application service is isolated into a set of containers, that are encapsulated with specific functions. These containers work collectively to execute the distributed MQTT application. The system appears to its application user as a single coherent system. The messaging service interacts with the external environment by sending and receiving messages through the MQTT communication protocol. The system uses a variation of the broker-based publish-subscribe overlay, which consists of a set of MQTT brokers.

Each online node in the broker cluster manages its own messages and client sessions. The cluster nodes communicate with each other via message passing. The system resembles a single logical unit from outside which is much like a centralized architecture. Nodes can join the cluster at any moment, by transmitting join requests to the other active nodes in the cluster. Node crashes are detected by the membership protocol. As the MQTT broker is a stateful application, a node that leaves and rejoins after a while is assumed to forget all its prior state. Therefore, the state information is maintained and is refreshed periodically to maintain state consistency and correctness. Upon failure of one node, the remaining brokers can take over and continue to provide the MQTT service.

As depicted in Figure 4.1(a), there are a few different components in the system. The entire MQTT functionality of the system is provided by the MQTT broker. Mosquitto broker is used for the MQTT broker service. The MQTT broker is the core communication component of MQTT devices that implements standard MQTT server protocol. The broker accepts subscriptions and

disseminates information between subscribers and publishers. The MQTT broker does all the topic matching, subscriptions management, filtering and matching, and publication delivery. The broker communicates with its local clients while the cluster server acts as a message forwarder to the broker. Both the MQTT broker and the cluster server are stateful applications. Each broker node shares information about the state of connected sessions and also meta-information about the cluster itself. If a cluster node is stopped, the other cluster member will know that there is a node missing. The cluster meta-information and MQTT subscription state need to be updated to deal with network partition and possible message loss. By using periodic updates with time-to-live (TTL) intervals to discard outdated entries, the broker cluster ensures that the convergence regarding subscription data within the distributed system is eventually reached.

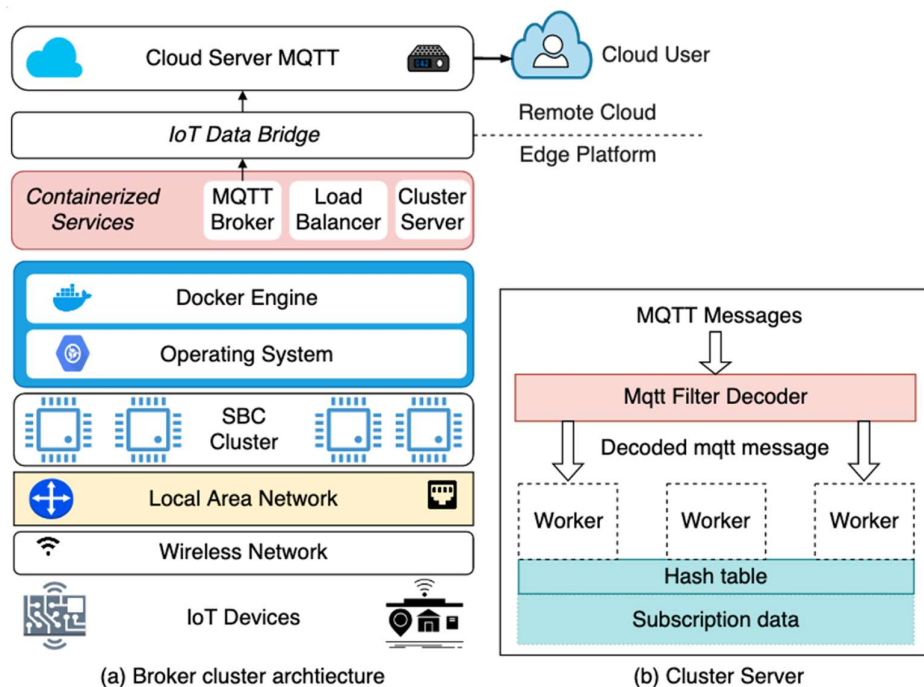


Figure 4.1 Edge cloud MQTT broker cluster architecture

The broker cluster system uses a load balancer to share the load among multiple brokers. The performance testbed uses a TCP load balancer as a single endpoint for the communication with publishers and subscribers. The TCP load balancer can be implemented using software components such as HAProxy (Tarreau and others, 2012). The load balancer equally spreads incoming workload across backend MQTT broker nodes, so that neither broker node is overloaded with too many clients. Since this implementation deals with the reliability of backend servers and only uses one load-balancing node, the load balancing server is assumed to be reliable. The focus of the design and implementation shall be resisting against crash failures among the backend MQTT brokers.

The design focuses on building routing states among brokers to efficiently route messages towards interested subscribers. In an edge-based deployment, the topology of the local network enables higher routing efficiency because routing only needs one hop, as all the brokers are fully connected in a local area network. The idea of adding a middle-layer cluster server is to decouple the MQTT broker's operations from clients. The MQTT service is provided to the clients without their need to know the details about underlying software and hardware. As depicted in Figure 4.1(a), the cluster server separates the distributed routing mechanism from publish-subscribe operations of individual MQTT brokers. The cluster server is responsible for coordinating publish-subscribe and routing operations among all MQTT brokers. It sits on top of the MQTT broker, and use message passing to communicate with other cluster nodes. The cluster server is also transparent to the broker.

The cluster server is implemented using Golang and deployed using Docker containers. The MQTT broker application does not need recompilation or relinking on its software. The cluster server consists of two components as depicted in Figure 4.1(b). The first component manages and intercepts incoming MQTT packets asynchronously via pcap-filter. An MQTT Deserializer module converts input streams of MQTT messages into meaningful application-level messages. Packets are deserialized according to the MQTT version 3.1.1 standard. The MQTT packets are captured, decoded, and tunneled into the message receiving buffer for processing.

The second component uses a configurable worker thread pool and a thread-safe hash table to match publish messages, forward publications, detect duplication, synchronize subscription state, and retransmit failed publications. Each worker thread encodes and forwards received publication messages to matching remote brokers within the cluster. The hash table is a temporary in-memory storage component used to maintain subscription data necessary for routing, synchronization, and publication retransmission. The hash table is updated as part of the synchronization protocol in the cluster. The broker cluster uses decentralized routing and event dissemination. The system also uses a gossip-based protocol to maintain node membership, by performing periodic updates to give a more consistent view of the cluster environment.

Figure 4.2 depicts the data pipeline of MQTT from edge to cloud through the internet and data bridge. The cloud integration module provides a publish-subscribe messaging framework to stream edge data to the cloud. The

cloud module provides an interface between the MQTT edge services with end-user requests at the cloud. It also forwards aggregated data from the edge server to the cloud through an MQTT data streaming bridge. The IoT end-to-end data pipeline ingests messages from IoT devices to different data stores in the cloud. In this way, the edge brokers are ubiquitous and do not depend on connectivity to the cloud broker. The functionality of this module is achieved using Apache Kafka with MQTT bridge. Kafka is an open-source stream processing data pipeline developed by LinkedIn (Kreps et al., 2011).

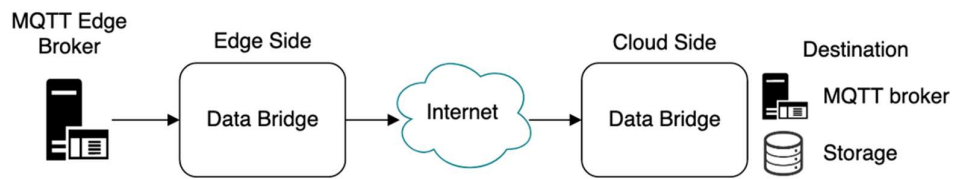


Figure 4.2 Data Pipeline

To integrate MQTT messages to Kafka, the implementation uses an MQTT extension framework called Kafka connect to ingest data from the edge MQTT brokers and streams the data to the cloud system. Kafka pushes selected MQTT messages received in the edge brokers to the cloud for storage and further processing. Data pipelines of Kafka also maintain network and data fault-tolerance by allowing data buffer at the edge side when the Internet is inaccessible.

4.2 Software Application Stack

Docker Swarm orchestration increases the ease of deployment in the edge cloud settings. Container services scheduling can be set up easily with Docker compose scripts and Docker Swarm API. The edge to cloud integration

and orchestration module is developed based on the architecture presented in (Alam et al., 2018). Figure 4.3 shows the software components in different layers of the cloud. The application services are logical groups of containers based on a set of Docker images. Containers can be easily removed and updated without impacting overall cluster services. Docker services can be orchestrated locally, or centrally in the cloud through the middle layer gateway. The edge broker offers a local connection point to edge devices and MQTT services with reduced response times.

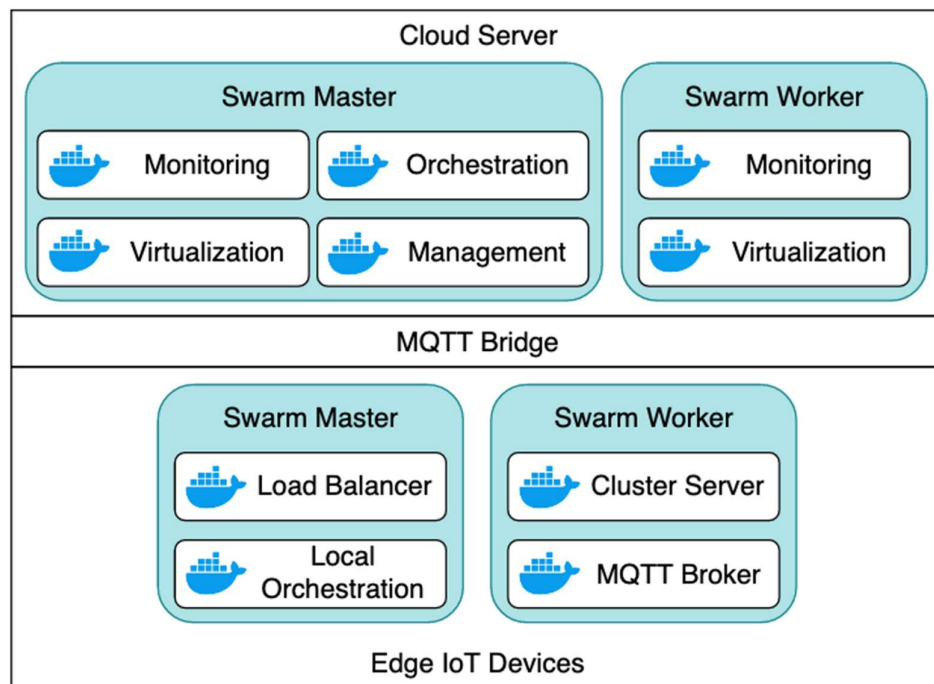


Figure 4.3 Cluster container application stack

The application in the edge layer has three pieces of services: a cluster server node, an MQTT broker, and a load balancer. The MQTT brokers are edge brokers that are responsible for connectivity between sensors. The cluster server is an interconnect layer that works on top of the distributed MQTT brokers. The

cluster service is built to fit between MQTT server protocol and MQTT client so that existing open-source MQTT broker and client implementations can be seamlessly integrated. The cluster server makes routing decisions based on the type of MQTT messages from clients. Critical client information is stored locally in each node. Distributed state information is maintained using periodic updates realized by communication primitives in the membership protocol. Each cluster server node cooperates and communicates with each other within the network to make routing messages and share routing information. This removes the necessity for individual brokers to communicate directly with each other. Each broker node may not be aware of the presence of a proxy or any other broker nodes in the network.

The MQTT broker is only responsible for its own designated functions. The MQTT broker manages and delivers messages from publishers to subscribers local to its own scope. The broker also maintains a session with connected clients and sends an acknowledgment (*ACK*) and ping response (*PINGRESP*) message to the clients. All incoming MQTT messages are seen by the broker as MQTT client requests. The cluster server is, therefore, a network element that enables coordination between physically distributed MQTT brokers, which themselves do not have any clustering capabilities. Another purpose of the cluster server is to redeliver failed messages by caching failed publications.

The Docker Swarm master node can scale and update Docker containers in a cluster of SBC nodes. It ensures efficient allocation and scheduling of Docker containers by keeping track of the deployed services. The load balancer sits in the manager node and offers a single-entry point to clients. The clients connect to the frontline load balancer to receive MQTT services. By using a load balancer, the backend IP address of each cluster node is not published to the clients. Each broker node receives a request distributed by the load balancer, processes the request, and responds to the balancer. The balancer, in turn, changes the response IP back to the IP of the balancer and forwards the response to the client. Clients can connect to any node.

A round-robin algorithm usually works fine for short-lived and stateless connections. However, the application workload for the broker cluster requires long-lived and bidirectional communication over the load balancer. If an SBC node crashes, a potentially large number of subscribers will initiate new connections to other online brokers. A round-robin algorithm for this workload causes the workload to spread unevenly in the backend. Some backend servers will have too many connections compared to the others. The load balancing algorithm should assign new connections to the least loaded backend server based on the number of active connections on each of the backend servers. Therefore, the load balancer uses the least connected algorithm so that new clients will be allocated to the least loaded servers when any backend node fails. Apart from a better balance in the number of connections to each backend server, the reconnected clients from a dead backend node will be spread evenly over the remaining backend servers.

4.3 Components Relationship

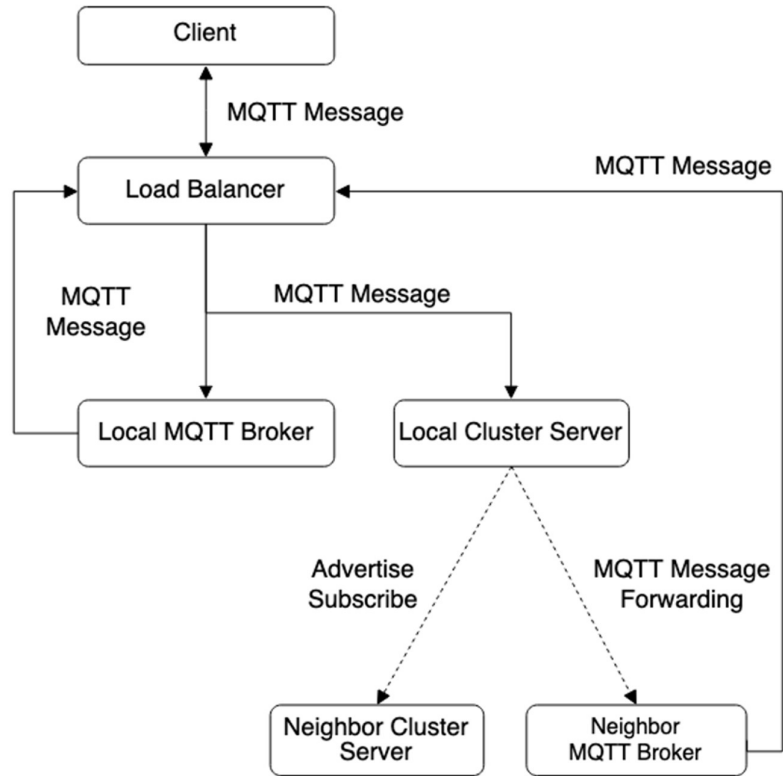


Figure 4.4 Software component interaction

Figure 4.4 depicts the interaction between each component of the MQTT broker cluster. Each component of the broker cluster is implemented via Docker container using the microservice architecture. In the context of the broker system, an MQTT client that is connected directly to a broker is called a local client. To the clients, the broker that connects to them is called the local broker. The MQTT broker directly sends MQTT responses to the load balancer, which in turn redirects the responses to the clients. A local broker is a broker that sits within the same host machine as the local cluster node. The local broker delivers messages to local subscribers while the local cluster node routes messages to

remote neighbor brokers inside the cluster. The neighbor cluster server is a remote cluster node that maintains a connection with the local cluster node. A neighbor client is a client that is connected to a neighbor broker. The cluster node works together to facilitate message delivery to all subscribers, regardless of which node they are connected to.

4.4 Broker Cluster Topology

The broker cluster is configured in a fully connected mesh where every node in the cluster is connected to every other node. Each peer node is only one hop away from each other. With this configuration, each node knows about all the state information and client connections from each neighbor node. Each cluster server node maintains the routing state information within its local data store. The routing state information contains a forward table, a recovery table, and a subscription routing table. The forward table maps each topic to the IP address of the neighbor brokers that subscribed to the topic. The subscription routing table maps the IP address of each neighbor broker to a set of its subscribed topics. The recovery table caches lost publication to facilitate message retransmission.

The membership framework used here implements the SWIM protocol (Das et al., 2002). SWIM protocol is a gossip-based membership protocol that detects node failures and maintains membership information of the cluster nodes. One drawback of this protocol is weak consistency. The membership protocol maintains the broker nodes in a mesh topology. Each member node has a complete view of the topology. At any time, nodes can have a different view

of the global overlay and will eventually converge to the same state. A node disseminates its message in an epidemic protocol by sharing information only with a random subset of its peers. Subsequent nodes then share this information with a random subset of its neighbor peers, until the entire cluster receives that information.

SWIM has separate layers of failure detection and message dissemination module. Each node in the cluster probes a node at random and expect an acknowledge message in return within a timeout. If the acknowledgment message is not received from a node during probing, the node will try to probe it through other nodes to prevent a false-positive state. A node is marked as dead when the node cannot be accessed by any of the members. The event of node failure is propagated across the cluster. The membership protocol will notify other broker nodes when a new node joins in. Each online node sends a set of its subscribed topics to that newly joined member so that the routing information is kept synchronized. The newly joined member node adds the topic list received to its routing structure.

4.5 Subscription Routing Management

The system uses subscription flooding (Eugster et al., 2003) to propagate subscription state information across the broker cluster. Each node maintains a local subscription table, a subscription routing table, and a forward table.

The local subscription table contains entries for each MQTT subscription received by the MQTT broker. The entry for the local subscription table is in the form of $\langle ClientID, topicSet \rangle$, where *ClientID* is the identifier of

the MQTT client, and *topicSet* is a set of MQTT topics subscribed by the client. The broker node uses the local subscription table for the recovery and synchronization process, which will be described in Section 4.7.2.

For each received MQTT subscription, the broker node sends a *subscribe* message along with the MQTT client identifier, and lists of subscribed topics. An Unsubscription advertisement is like a subscription advertisement except that it marks the message with an *unsubscribe* tag. The subscription routing table contains entries for each subscription advertisement received by the system.

Each subscription routing entry contains a pair of tuples in the form of $\langle sNode, ClientID, topicSet \rangle$ where *sNode* is the identifier of the broker node that sends the subscription advertisement, *ClientID* is the identifier of the subscribing MQTT client and *topicSet* is the list of topics specified by the client. Each broker node that receives the subscription advertisement stores it in a subscription routing table and a forward table.

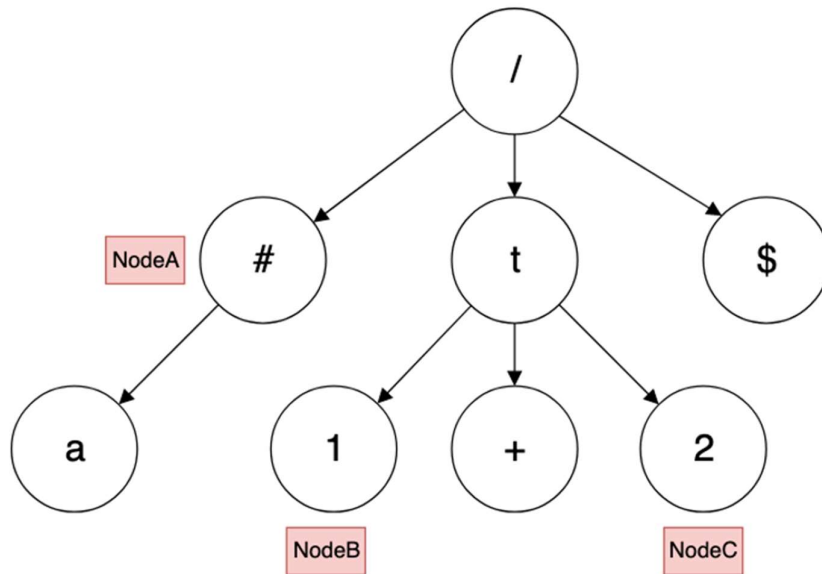


Figure 4.5 Topic Trie for Forwarding Table

Each broker node uses the subscription routing information to construct publication forwarding paths. The forward table stores all the routing paths of each topic with the identity of the neighboring node from which the subscription advertisement was received. The forward table uses the topic as keys and destination brokers as values. The forward table takes the form of $\langle \text{topic}, \text{nodeSet} \rangle$, where *topic* is the topic of subscription, and *nodeSet* is a set of broker nodes that have registered to the given topic. The purpose of the routing table is to provide fast topic matching for received publication messages. The forward table implements the prefix trie data structure (Datta et al., 2005) as depicted in Figure 4.5. A topic trie is a data structure used for fast searching operations on a given topic. Each broker uses the topic matching routine $\langle \text{MatchTopic} \rangle$ given in Listing 4.1 to search for broker nodes, whose subscription matches with the given publication topic. $\langle \text{AddBroker} \rangle$ is a helper function to append all brokers in the current nodes to the result. $\langle \text{Recursive_Match} \rangle$ is a recursive function

that matches the subscriptions for every level of a given *pubTopic* separated by ‘/’ and appends the brokers for every matching topic level to the search results. Upon querying the forward table with a topic that comes with the publication message, it returns a set of forwarding paths consisting of a set of brokers nodes *<brokerSet>* that matches the publication topic. This reduces redundancy and network overhead of message forwarding as all received messages are filtered before they are forwarded. The system then forwards this message to the list of destination brokers using the message delivery process described in Section 4.6.

The system implements the subscription routing table using a thread-safe concurrent hash table, with each neighbor id mapping to the corresponding *clientID* and the topics that they subscribed to. The list of topics within the hash table is implemented as a set so that it is easier to update the structure, during the synchronization period. A set can be updated through combination set operations such as union, intersect, and complement. Each subscription entry has a time-to-live (TTL) period and expects updates from the neighbor nodes to refresh the TTL period. A subscription entry is discarded when it expires or the broker receives an *unsubscribe* advertisement from a neighbor node.

```

function MatchTopic(topicName) {
    // Split string with topic level separator '/'
    topicSlice := splitString(topicName, "/")
    brokers := Array(string)
    root := topicTrie.Root
    Recursive_Match(root, topicSlice, brokers)
    return brokers
}

function AddBroker(node, brokers) {
    for each broker in node.Brokers {
        Add broker to brokers
    }
}

function Recursive_Match(node, topicSlice, brokers) {
    endFlag := length_of (topicSlice) == 1           // one token left
    // find for key '#' in set of children Nodes
    if childNode := node.children["#"] {             // multi-level wild card
        AddBroker (cnode, brokers)
    }
    if childNode := node.children["+"] {             // single-level wild card
        if endFlag == TRUE {
            AddBroker (childNode, brokers)
            if n := childNode.children["#"] {        // multi-level wild card
                AddBroker (n, brokers)
            }
        } else {
            Recursive_Match (childNode, topicSlice[1:], brokers)
        }
    }
    if childNode := node.children[topicSlice[0]] {
        if endFlag == TRUE {
            AddBroker (cnode, rs)
            if n := childNode.children["#"] {        // multi-level wild card
                AddBroker (n, brokers)
            }
        } else {
            Recursive_Match (childNode, topicSlice[1:], brokers)
        }
    }
}

```

Listing 4.1 Topic matching routine

4.6 Publication Message Forwarding

4.6.1 Normal Condition

The broker node uses the forward table to build forwarding paths. For each publication message, the broker node generates a unique sequence number $\langle pubMsgSeq \rangle$ for the reception of this message for the first time. This sequence number is unique and ascending. Each broker node uses the matching routine $\langle MatchTopic \rangle$ to find neighbor nodes with matching MQTT subscription.

After retrieving the $\langle brokerSet \rangle$ containing the matching destination brokers, the broker node stores the entry of publication confirmation message in the form of $\langle pubMsgSeq, ackSet \rangle$ in the publication acknowledgment table. $ackSet$ represents the acknowledgment messages received from the destination brokers, $brokerSet$. The entries of $ackSet$ are in the form of $\langle broker, ackReceived \rangle$, where $ackReceived$ is a Boolean value that indicates the reception of acknowledgment message from a destination broker. The value of $ackReceived$ is *false* by default and is set to *true* whenever the forwarding broker receives a publication acknowledgment (*PUBACK*) message from a destination broker. The broker node removes the copy of the publication message after it receives acknowledgment messages from all destination brokers.

The broker node filters destination brokers in a $destinationBrokers$ by the $brokerSet$ and Boolean values in the $ackSet$, given in Equation 4-1.

$$destinationBrokers = \{brokerSet \wedge ackSet\} \quad (4-1)$$

The broker node only forwards the publication message to a neighbor broker that (i) has subscriptions that match the publication topic and (ii) does

not confirm the reception of the publication message. The forwarding broker sends a copy of the publication message to all the brokers in $\langle destinationBrokers \rangle$ using QoS 1 of the MQTT protocol, as depicted in Figure 4.6(a). The neighbor broker sends an acknowledgment message (*PUBACK*) back to the forwarding broker to confirm the reception of the publication message. If *PUBACK* is not received from the destination broker, the publication message is stored in a buffered queue for message retransmission, as depicted in Figure 4.6(b). The message retransmission process is described in Section 4.6.2. Figure 4.7 shows the message forwarding sequence for the broker cluster.

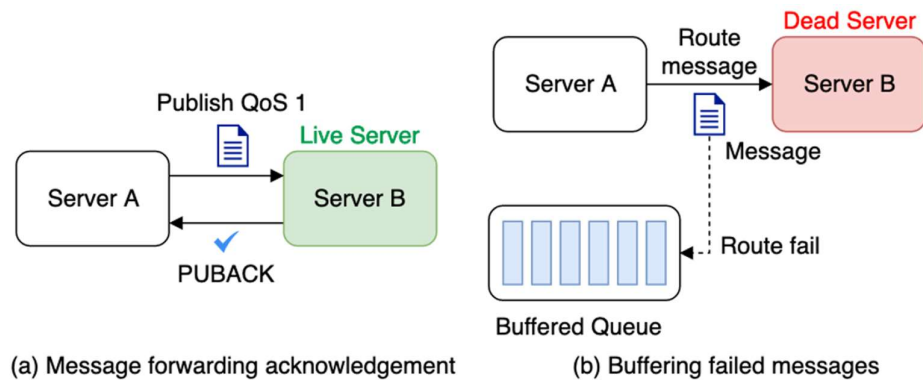


Figure 4.6 Broker publication message forwarding

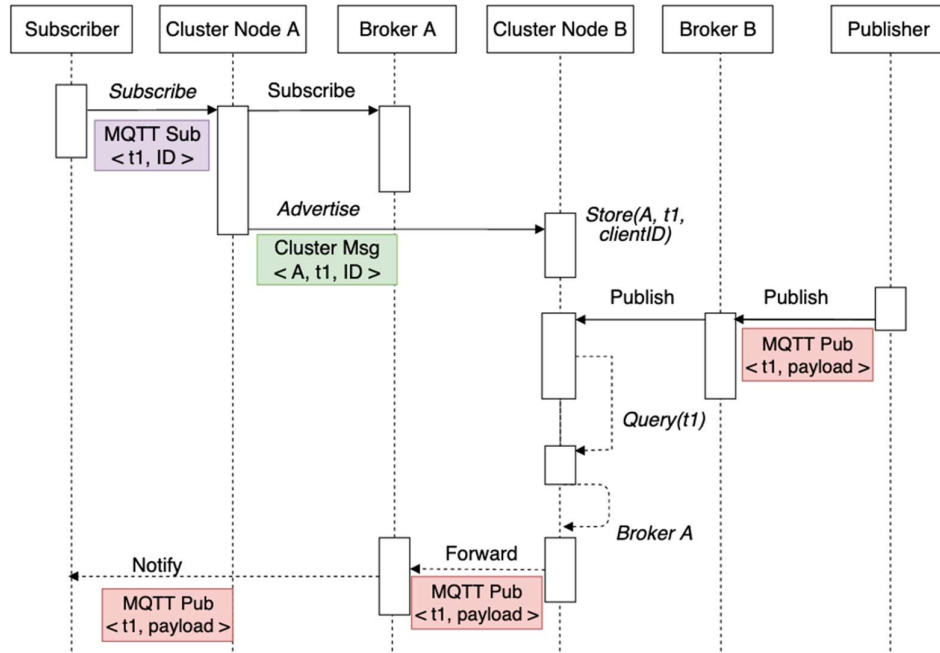


Figure 4.7 Message routing sequence

4.6.2 Message Forwarding with Failed Brokers

Message forwarding implements message acknowledgment to confirm the successful forwarding of a message to a neighbor MQTT broker. If the *PUBACK* message is not received from any one of the destination brokers, the message forwarding process fails. The failed publication message is pushed into a recovery table that maintains a queue of backup messages. The broker node redelivers these messages as soon as client connections come back online. The message retransmission approach realizes fault tolerance in the system by a compensation approach (Avizienis et al., 2004). Figure 4.6(b) depicts the message buffering process for if the failed messages when *PUBACK* is not received. For each incoming subscription, the message recovery module checks the client ID and the subscription topic by referring to the recovery table. Each entry of the recover table is in the form of `<ClientID, topic, msgQueue>`, where *msgQueue* refers to a first-in, first-out (FIFO) buffer of failed messages. Each

entry in *msgQueue* is in the form of $\langle pubMsgSeq, msg \rangle$, where *pubMsgSeq* refers to the publication message sequence described in Section 4.6.1 and *msg* is the copy publication message. If both the *ClientID* and *topic* for a subscription match any entries in the recover table, it means that the subscription is a disconnected client and not a resubscription by the client itself. The recover table returns the message queue that corresponds to the *ClientID* and *topic*.

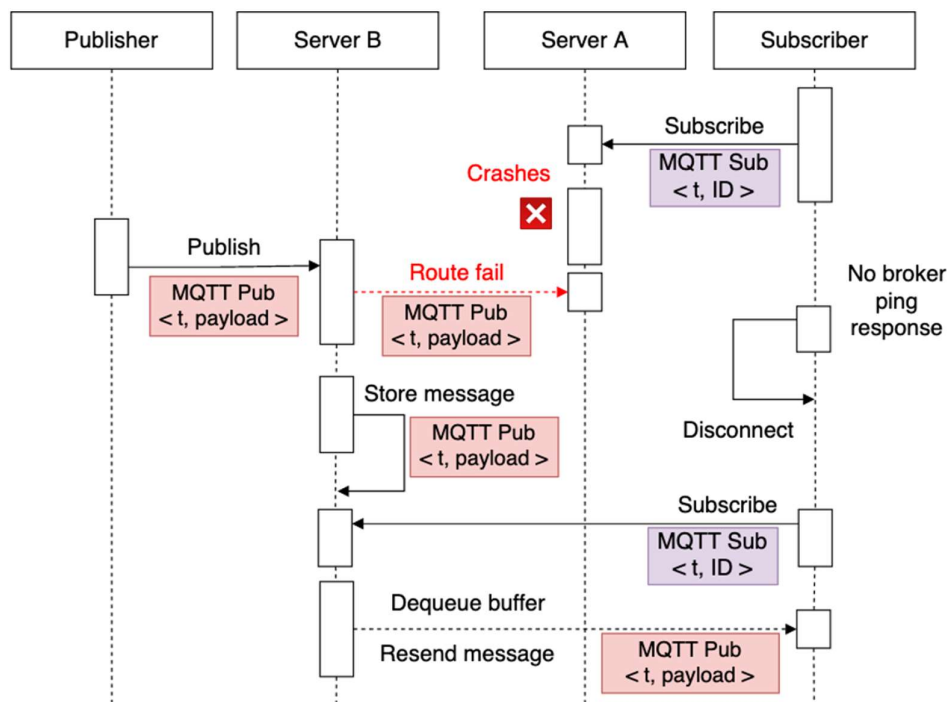


Figure 4.8 Message retransmission process

Figure 4.8 depicts the sequence of the message resending process. The system will retransmit all the missed publications that are stored in the message queues. The cluster server records the message IDs of forwarded publications to prevent forwarding the same message multiple times. If the retransmission process receives and a publication acknowledgment *PUBACK* message in return, the corresponding entry in the publication acknowledgment table is set

to *true*. The publication message is discarded when all of the values in the *ackSet* entries are *true*.

4.7 Recovery of Routing State

The proposed broker cluster only tolerates node crashes, where a process stops reacting to incoming messages due to software or hardware faults. Node crashes can lead to inconsistency in state information within the broker overlay within the cluster. However, the system does not handle Byzantine faults. The reason for this is to reduce the cost of replication. This section describes the fault monitoring and recovery process for the broker cluster.

4.7.1 Monitoring and Failure Detection

Without fault tolerance, the failure of a single component can disrupt the normal operation of the system. The broker cluster provides client connection failover so that clients can their connection as quickly as possible. Any cluster nodes can fail independently without affecting other nodes.

4.7.1.1 Cluster Node Failure Detection

Each cluster peer node implements a heartbeat mechanism for monitoring neighbor peers. It emits heartbeat signals to neighbor servers at regular intervals. For every heartbeat received, the peer node resets the timeout for heartbeat reception. Figure 4.9 depicts the UDP heartbeat mechanism among cluster nodes. When the server failure is detected through a heartbeat timeout, the client should continue receiving a stream of MQTT service within the broker cluster. In every broker node, a local failure detector is implemented by the

membership protocol to monitor the reachability of a neighboring node, via periodic pings with timeouts. A live node that detects a failed node will update the subscription routing table associated with the failed node. If node crashes occur, the fault mechanism ensures that a live broker node will replace the client and operations of the failed node.

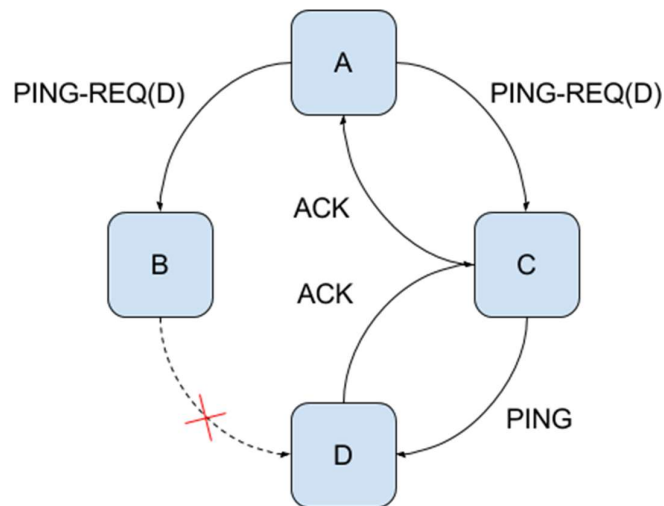


Figure 4.9 Cluster membership heartbeat detection

4.7.1.2 Load Balancer Health Checks

The load balancer implements layer 4 TCP health checks to prevent sending packets to an offline server. The load balancer periodically sends health check probes and attempts to connect to the TCP port of the backend servers. As depicted in Figure 4.10, a *TCP SYN* request to a backend port expects a *TCP SYNC ACK* response in return. If the response is not received within a predefined timeout, the server is marked *DOWN* by the load balancer. The load balancer isolates the *DOWN* server and routes packets to the remaining online servers based on the least connected algorithm. The load balancer then continues to distribute MQTT requests to the remaining online broker nodes.

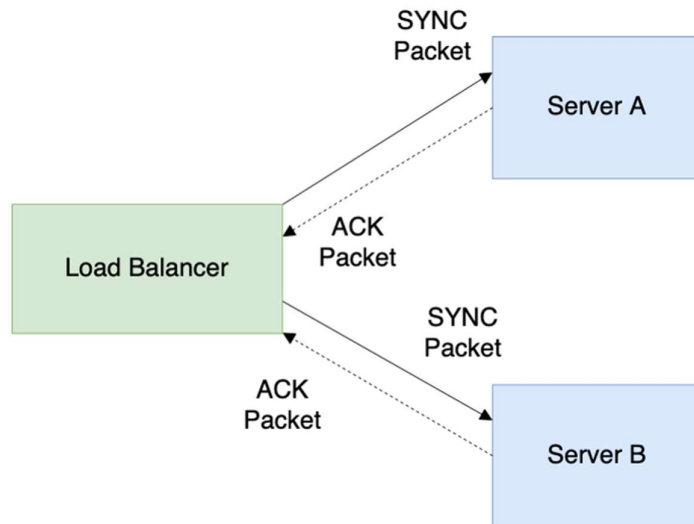


Figure 4.10 Load balancer TCP health checks

4.7.1.3 MQTT Client Keepalive

For monitoring MQTT server health, the MQTT client uses ping requests and expects a ping response from the server. The MQTT client starts a timeout for every ping acknowledgment sent. If the server has not replied within the timeout period, the client assumes the server to have failed. The client then reconnects and sends a connect message to the remaining online servers routed by the load balancer. Figure 4.11 depicts the MQTT client keepalive and reconnection process.

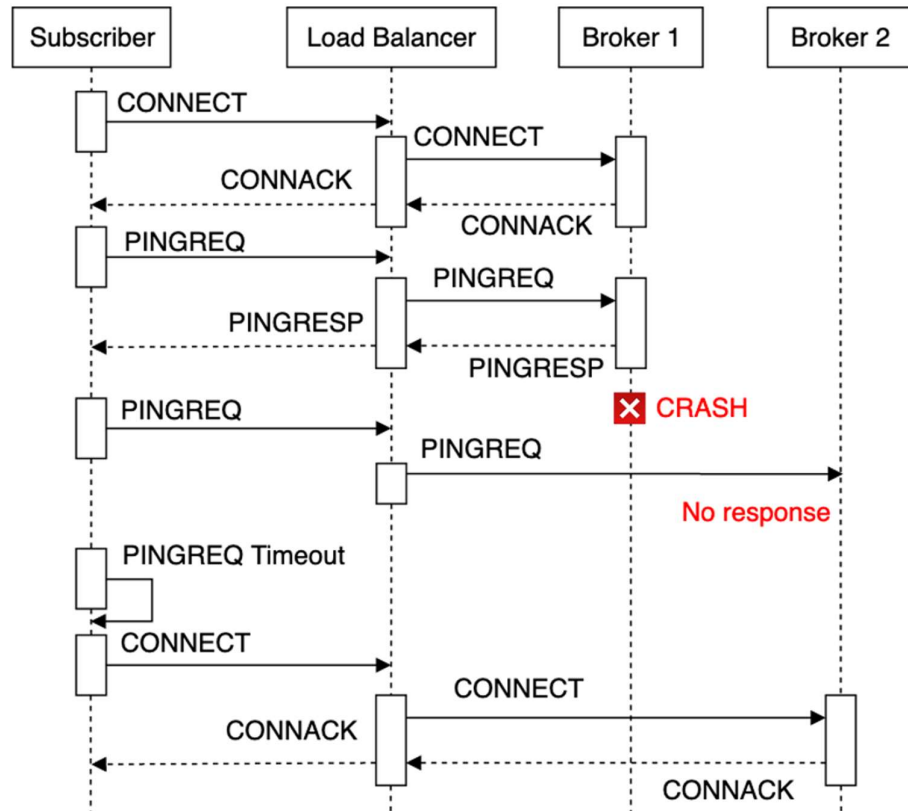


Figure 4.11 MQTT keep-alive

4.7.2 System State Reconfiguration

The system assumes the crash-recovery failure model for broker node failures. This means that each broker node is assumed to be either online or offline at any point in time. When a broker node fails, it stops operating until it comes back online. When a failed broker comes back online, it loses all of its soft-state subscriptions and routing information that it had before it crashes. To handle crash failures, the system recovers the subscription routing state into a consistent state and provides redundancy through message retransmission. When a broker fails, clients connected to the failed broker reconnect to one of the online brokers in the cluster. The rest of the cluster stores the publication messages until the clients restore their connections to the system. If state

information in the failed broker cannot be recovered or the recovery process is slow, the publication messages will be lost. The system updates the routing table of remaining brokers so that new publication messages are properly routed. The system will deliver the publication received during the recovery period after the joining broker is fully recovered.

4.7.2.1 Node Failure and Routing State Synchronization

The state information in each cluster node will become inconsistent when a broker node fails. To maintain consistency of the routing state after a failure, the system uses the periodic resubscription approach (Jerzak and Fetzer, 2009; Siegemund et al., 2015). Each broker node maintains a timestamp for every subscription entry in the subscription routing table. The broker node refreshes the subscription entry every time it receives a subscription advertisement from a neighbor broker node. A subscription entry will timeout when the broker node does not receive a refresh subscription advertisement within a time-to-live (TTL) period. The broker node discards any outdated entries in the subscription routing table and the forward table. For fast updates in the case of failure, the event of failure is propagated across the cluster. Each broker node immediately updates and deletes the corresponding entry in its routing tables relative to the failed broker node.

4.7.2.2 Node Initialization and Recovery Operation

A failing or new broker node that joins the cluster topology is treated as a recovering node by the system. The new node enters a *recovery* state. When a node joins the cluster topology, the joining node sends a *JOIN* message along

with a TCP push/pull request to every other live broker node to initialize a recovery operation. Upon receiving a TCP push/pull request with a *JOIN* message from a broker node, each existing broker node computes a set of its local subscription information from its local subscription table and sends it to the joining broker over the TCP network. The recovery message is in the form of $\langle originNode, subscriptionSet \rangle$, where *originNode* is the synchronizing broker node and *subscriptionSet* is a set of entries in the local subscription table. After sending the recovery message, the existing broker node sends a *syncOK* message to end the recovery process for the joining broker node. After receiving a *syncOK* message, the broker node enters a *normal* operation state. Each existing broker uses a *syncFlag* to indicate the normal operation of a neighbor broker node. Section 4.7.2.3 discusses the uses of *syncFlag*.

4.7.2.3 Recovery State

During the *recovery* state, publication may reach the newly joined broker node that does not have the complete subscription routing information of its neighbor broker nodes. This causes missing publication because the joining broker cannot compute any forwarding paths for the received publication messages. To prevent this, the joining broker node enters a recovery state and temporarily stores all publication messages into an initialization message queue.

The broker node maintains a synchronization set $\langle neighborBroker, syncFlag \rangle$, where *neighborBroker* is the identifier of a neighbor broker node and *syncFlag* is a Boolean value that indicates the completion of the

synchronization period. A broker node first enters a recovery state when it joins the cluster topology. During the recovery state, the joining broker node stores all received publication messages in the initialization message queue. The joining broker node updates its subscription routing table from the information received from neighbor broker nodes during the recovery operation described in Section 4.7.2.2.

Each neighbor broker node sends a *syncOK* message to the joining broker to mark the completion of the recovery operation. After receiving a *syncOK* message from a neighbor broker node, the joining broker node sets the *syncFlag* for the corresponding neighbor broker node to *True*. The joining broker node goes out of the recovery state after all of the values in the synchronization set entries are *True*. After going out of the recovery state, the joining broker retrieves the publication messages from the initialization message queue and performs the message forwarding process described in Section 4.6.

4.8 Implementation

Edge computing platform orchestrates services and resources on edge nodes in a distributed way, similar to typical PaaS functions in the cloud. A Raspberry Pi board has low power consumption, which makes it possible to create an affordable and energy-efficient cluster for environments for which high-tech installations are not possible. A cluster consisting of five Raspberry Pi 3B boards is used as the hardware infrastructure to deploy the Docker containers. The clustered configuration also allows better robustness against

failure. Each board runs Hypriot OS, a customized version of Raspbian integrated with Docker engine to deploy container services such as storage and cluster management. The operating system is a headless version of the Raspbian system, without a desktop environment, thus freeing up a large amount of memory and storage. The least amount of RAM is allocated for the board GPU to fully utilize the CPU in the platform. The Raspberry Pi SBC cluster uses a star network topology. One switch act as the core of the star and the other switches then links the core to the Raspberry Pi SBCs. The switch is connected to a router that supplies the DHCP server to distribute network configuration parameters.

The implementation of the MQTT broker cluster is illustrated in Figure 4.12. One node is used as the Docker Swarm manager that hosts the load balancer and manages the scheduling of services. The Docker Swarm manager node is connected to the internet and serves as a gateway for development and deployment interface for the cluster. User applications and IoT devices can connect the broker service through the wireless network. The load balancer is implemented using HAProxy which routes all requests at the transport layer. The other four SBCs are worker nodes that run the MQTT broker cluster. The broker component uses Mosquitto MQTT broker v-1.6.4 and a customized middleware cluster server for distributed coordination among brokers.

All application services are modular and independent microservices. This research work uses Docker Swarm for cluster management. One node is used as the swarm manager that runs a dedicated container for the load balancer

and the management interface. For fault tolerance, Docker Swarm supports redundant Swarm managers and can replicate information in the distributed key-value store. In the implementation, Docker Swarm orchestrates and distributes application services into a set of clustered edge nodes. A Docker-compose file is used to define the orchestration of the microservices and replicas for each cluster node.

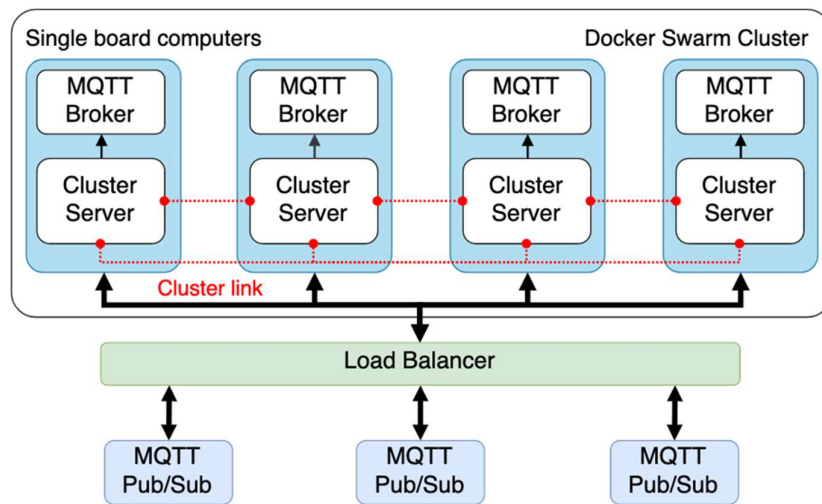


Figure 4.12 MQTT broker cluster implementation

The cloud integration module uses Kafka APIs to implement an intermediary MQTT bridge for both producer and consumer in the data pipeline. The Kafka cluster is deployed as Docker containers on two Raspberry Pi boards. The cloud users can also receive MQTT messages from the IoT devices residing at the network edge through the cloud MQTT broker. The cloud MQTT broker, central cloud orchestration, and monitoring are implemented in a generic laptop running Ubuntu OS.

CHAPTER 5

RESULTS AND EVALUATIONS

This chapter presents the evaluations of the microservice-based broker cluster to support communication between MQTT publishers and subscribers. To evaluate the distributed broker cluster system and its performance and fault tolerance, a testbed of an MQTT communication scenario is implemented. As many as two thousand MQTT devices are emulated as the application load. The evaluations assume that the load balancer is reliable and the system is secure.

5.1 Experiment Setup

The experiment setup uses a cluster of four broker nodes and one load balancer node. The results evaluated are end-to-end latency (millisecond), throughput (messages/second), inter-message jitter (milliseconds), latency after broker failure, CPU time utilization percentage, and RAM usage. The performance test is conducted by scaling the clients from 50 to 2000 pairs of publishers and subscribers. Each pair of clients corresponds to one unique topic. The performance of the broker cluster system is measured in terms of latency and message throughput to understand the effect of the number of clients on the system performance. The MQTT clients establish TCP connections over the wireless network to the MQTT broker. The testbed is implemented using a modified version of `mqtt-bm-latency` (hui6075, 2018), an MQTT load-test simulation written in Go. The benchmark uses multiple threads to asynchronously imitates a large number of devices as the publisher (QoS 2) and subscribers (QoS 0) via MQTT client APIs. The relationship between publishers

and subscribers is one-to-one, which represents a maximum of 2000 channels of MQTT communication.

The script is configured to publish 100 messages per second to the MQTT cluster, such that each publication is sent between an interval of 10 milliseconds. The message payload is fixed at 32 bytes, where it contains 32 bytes of timestamp value. The message payload is chosen to be small so that it does not overwhelm the Raspberry Pi 3 SBC that has only 957 MiB of usable RAM. The throughput is measured in terms of the average number of messages sent per second for each publisher. The benchmarks were conducted on a local router network, without any external traffic.

Throughput and latency evaluations are presented in Section 5.2 and 5.3 respectively. Due to variations in the network processing time, the latency and throughput measurements are slightly different each time the benchmark is performed. Therefore, the benchmark is repeated five times, and the average value of the measurements are shown in the result. Section 5.4 presents the jitter values the message delivery across the broker cluster. Timestamps of message production in the publisher and message reception in subscribers are both recorded as a pair to compute the end-to-end delivery latency for each message. The end-to-end delivery latencies are then measured to obtain their maximum, minimum, and mean values. Due to time synchronization errors between different machines, all clients are simulated on the same machine to obtain more accurate results.

Section 5.5 presents the fault tolerance evaluations of the proposed broker cluster. One of the brokers is deliberately turned off after all clients have subscribed to the MQTT broker. The end-to-end latencies and successful delivery rate of missed publications, before and after the failure of one broker, are evaluated.

Resource usage evaluations are presented in Section 5.6. To obtain runtime system metrics, the metric data is extracted from a metric server on each Raspberry Pi board and accumulated to a data aggregator. Prometheus is used to collect the metrics from the Docker hosts. Both services can be easily integrated into Docker Swarm. Node Exporter is a server provided by Prometheus to collect and expose metrics such as CPU utilization, RAM usage from a Docker host. The Node Exporter acts as a server that periodically sends out system metrics of the host to the Prometheus server. Prometheus then stores the metrics data with a timestamp in a database. Grafana is an open analytics and monitoring platform that is used to visualize the time series data collected by Prometheus. Instances of Node Exporter are deployed on each Raspberry Pi SBC host that runs the MQTT broker, while Prometheus and Grafana are deployed on a generic Ubuntu-based laptop. Figure 5.1 depicts the Prometheus-based monitoring stack to gather the system metrics, which include CPU utilization percentage and total RAM usage.

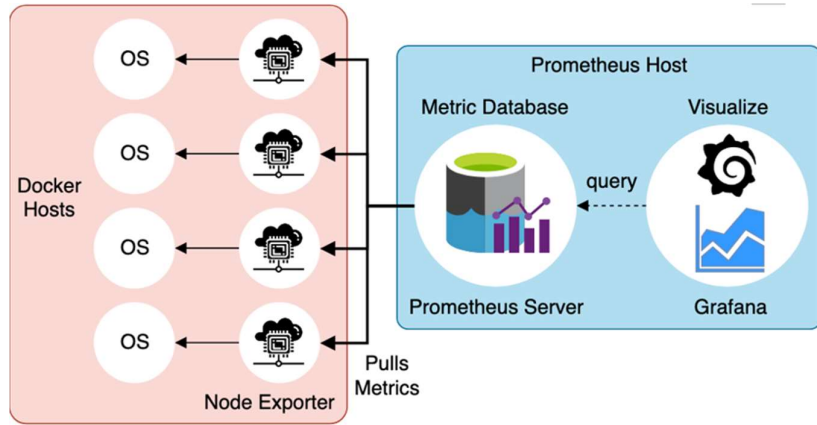


Figure 5.1 System metrics monitor

The broker cluster consists of four Raspberry Pi 3 Model B SBCs, each of which features 1GB of RAM, and 10/100 Mbps Ethernet network interface card (NIC). Each broker node runs HypriotOS v1.8.0 with Kernel v4.14 and Docker v18.06. The load balancer node uses Raspberry Pi 3 Model B and runs HAProxy v1.7 as the load balancer. The network switch used is a 100Mbps per second switch.

Table 5.1 gives the details of hardware and software configurations for the experimental setup.

Table 5.1 Experimental setup configurations

Processor	Quad-Core 1.2GHz Broadcom BCM2837 64bit
Motherboard	Raspberry Pi 3 Model B
RAM	1GB DDR3
Network Interface	10/100 Mbps Ethernet
Storage	16 GB 80MB/s microSD
Operating System	Hypriot OS v1.8.0 Linux Kernel 4.14
Docker Version	Docker 18.06.3-ce
Network	100Mb/s network switch, 100Mb/s wireless router
Load Balancer	HAProxy v1.7-stable
MQTT Broker	Eclipse Mosquitto v-1.6.4

Payload size	32 bytes (Unix timestamps)
Publish QoS	2
Publish message rate	100 message/sec

5.2 Throughput

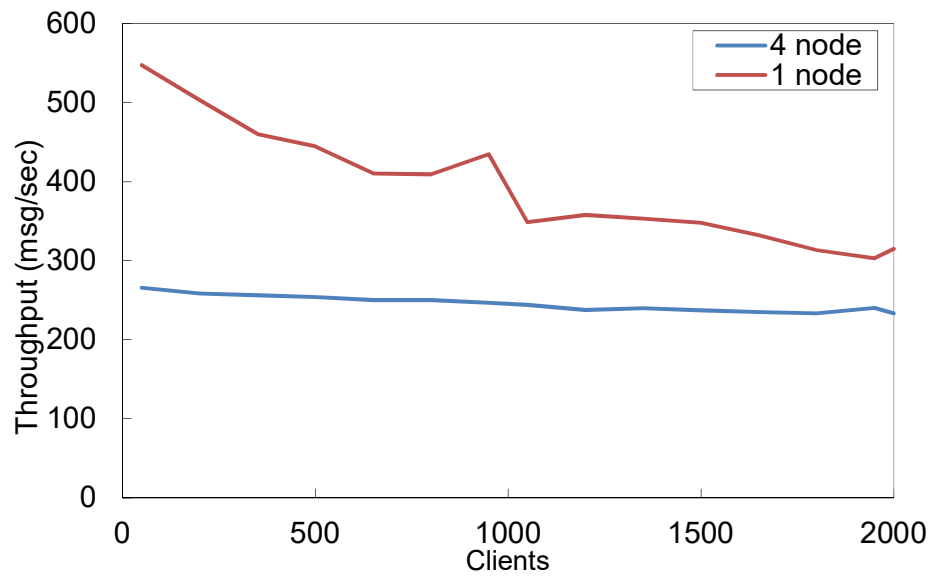
The throughput evaluation shows the speed at which data can be transmitted between devices. The publish throughput evaluated in this section is the average number of publication messages pushed to the broker per second. The average throughput is obtained by dividing the total throughput by the number of messages published. The goal of this benchmark is to evaluate how the MQTT broker cluster scales with the number of publishers.

As depicted in Figure 5.2(a), the average throughput delivered by each client decreases slightly when the number of clients increased. The single-node broker outperforms the broker cluster in terms of average throughput per client. The decline in overall throughput for the proposed broker cluster is not desirable, considering the cost of horizontal scaling. This happens as a result of relay elements presented in the load balancer. Since QoS 2 of the MQTT protocol involves a 4-way handshake, the total publication runtime increases. This reduces the overall message throughput since throughput is inversely proportional to their total runtime.

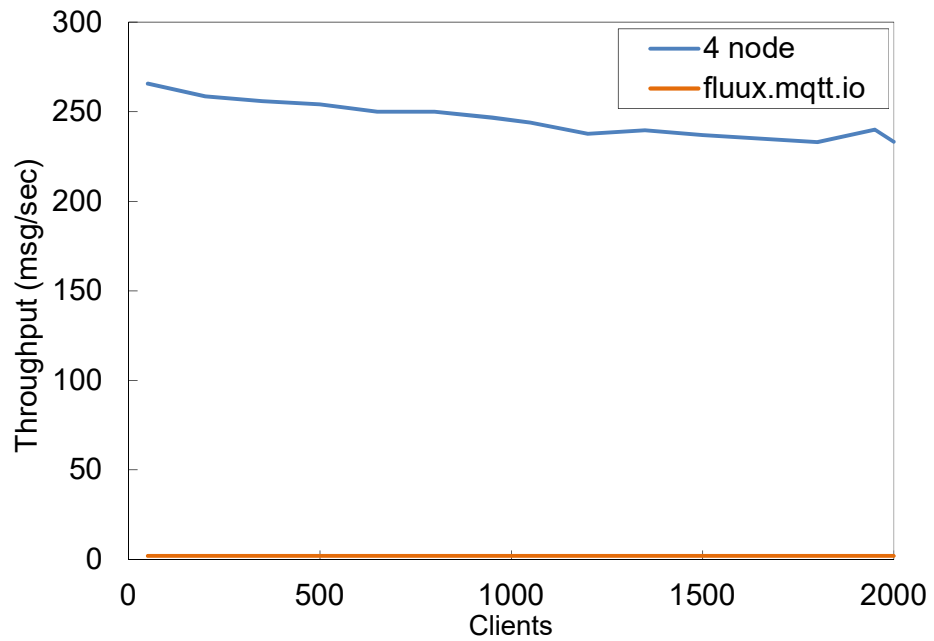
The average throughput of the clustered system is also compared to that of a cloud broker. As depicted in Figure 5.2(b), the edge-based broker cluster has a significantly higher throughput compared to the cloud broker. The average throughput per client remains below 1.5 messages per second for the cloud broker. It appears clear that how edge-based MQTT provisioning brings

significant advantages when compared to the cloud-based approach. It is also observed that the connections between the cloud broker and endpoint MQTT clients are not stable, due to network congestion on the internet. Some of the clients fail to establish a connection with the broker, while some clients are disconnected half-way when the broker fails to respond with a ping response (*PINGRESP*) message back to the clients. Extra delays are incurred when the disconnected clients reconnect to the cloud broker, which causes a reduction in message throughput.

The variations in average throughput per client for 50 and 2000 clients are provided in Figure 5.3(a). The average throughput values for fifty and two thousand clients are almost comparable for the broker cluster. The performance degradation of the broker cluster 12.98% when the number of clients increases. For the single broker setup, the average throughput decreases by 54%. Despite having lower average throughput measurements, the 4-node broker cluster shows better scalability compared to the single node broker. This suggests that the broker cluster can handle load increase better than the single broker setup. The type of service provisioning, either edge-based or cloud-based MQTT broker, does not impact performance degradation. The evaluations in Figure 5.3(a) show that the average throughput of the cloud broker is decreased by 1.18%.



(a) 4 node broker cluster and 1 node broker



(b) 4 node broker cluster and cloud MQTT broker

Figure 5.2 Average publish throughput with increasing clients (QoS 2)

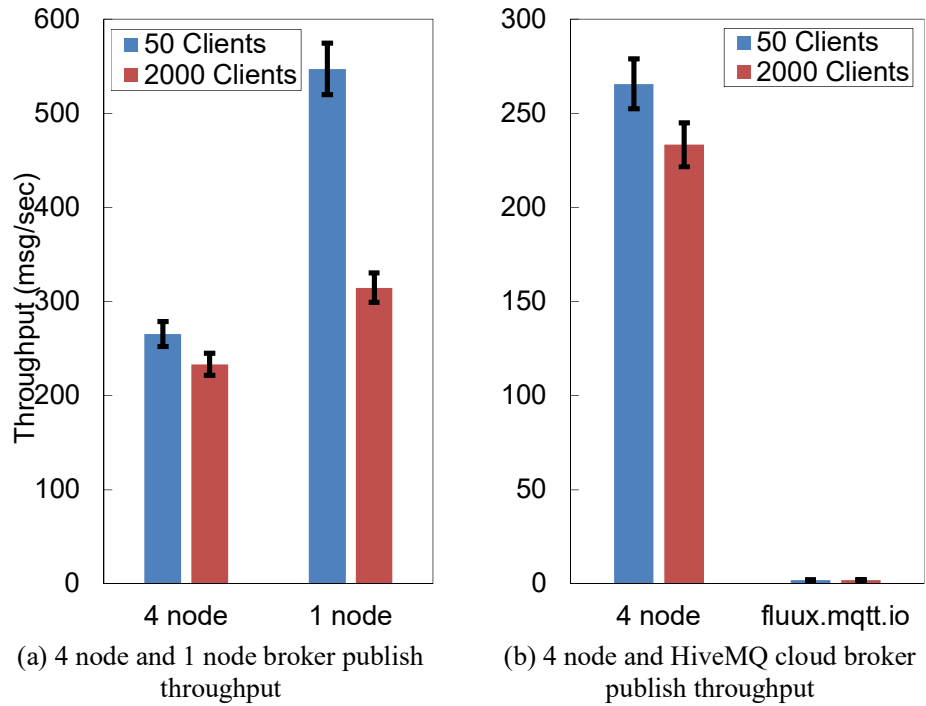


Figure 5.3 Throughput variations with 50 and 2000 clients

5.3 Latency

This section compares the latency performance between the proposed broker cluster, the cloud broker, and the single node Mosquitto broker. The measured end-to-end latency consists of transmission delay, the broker processing delay, and the message forwarding delay within the cluster. For latency constraints applications, the end-to-end latency should be bounded to an acceptable threshold depending on the users of the system. The latency requirement of a typical IoT data streaming application usually ranges between 10 milliseconds to 100 milliseconds, depending on the users of the application (Nikaein and Krea, 2011). The test script records a timestamp when publications are sent by a publisher within the MQTT payload. The testbed varies the number of loads to measure its effect on the end-to-end latency.

The end-to-end latency of the received messages is computed by subtracting the timestamp of the published message ($sent_t$) from the timestamp of the received message (rcv_t). The same machine is used for both the receiving and sending side, so clock drifts do not affect the accuracy of the measured latency. For every test, the difference in the arrival values is computed using the following calculation.

$$L_{i,rcv_ts} = (msg_{i,rcv_ts} - msg_{i,sent_ts}) \quad (5-1)$$

where msg corresponds to the received messages, rcv_ts represents the reception timestamps, $sent_ts$ represents the sending timestamps, and i is the current message.

The values of latency are stored in a vector for each test, building a new data matrix of values whose ith row has the following elements.

$$[Data]_{i,k} = \left(\sum_{i=1}^n L_{k1}, \sum_{i=2}^n L_{k2}, \dots, \sum_{i=n}^n L_{km} \right) \quad (5-2)$$

where n is the total number of messages ($1 \leq n \leq 2000$). L_k corresponds to the latencies vector of the current test ($1 \leq k \leq m$), and m is the number of tests. These latency values are used for descriptive and probabilistic statistical analysis.

Figure 5.5(a) shows the average end-to-end latencies of the broker cluster and the single node broker setup. The average latency values of the broker cluster are inconsistent with increasing the number of clients. The reason behind this is due to the data locality of the backend MQTT broker being routed by the load balancer. The least connected algorithm of the load balancer routes

client requests to backend brokers that were holding the least number of clients. The results in Figure 5.5(a) show that, with N number of brokers, the non-deterministic routing happens when the total number of publisher/subscriber pairs are not divisible by N . This is because of the uneven distribution during the subscribing process that causes some backend brokers serving more subscribers than the others. During the publishing process, each publisher is routed to the broker with the least clients at the time which causes some mismatches. Hence, more publishers could not reach the backend brokers with the same registered subscription. As depicted in Figure 5.4, an extra forwarding delay is incurred when a published message must be forwarded to the broker that is holding a registered subscriber of the same topic.

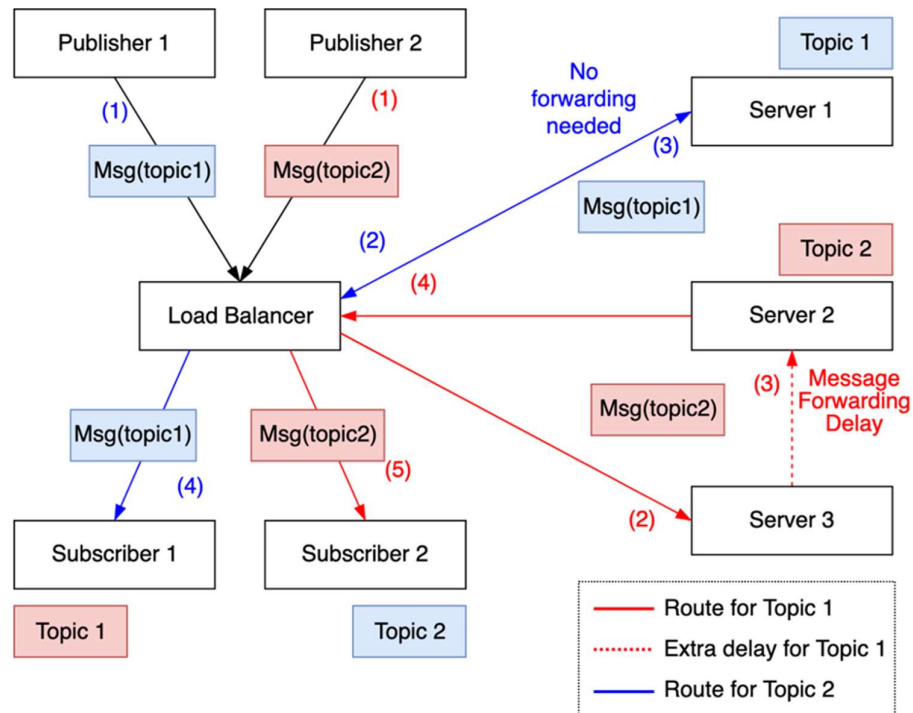
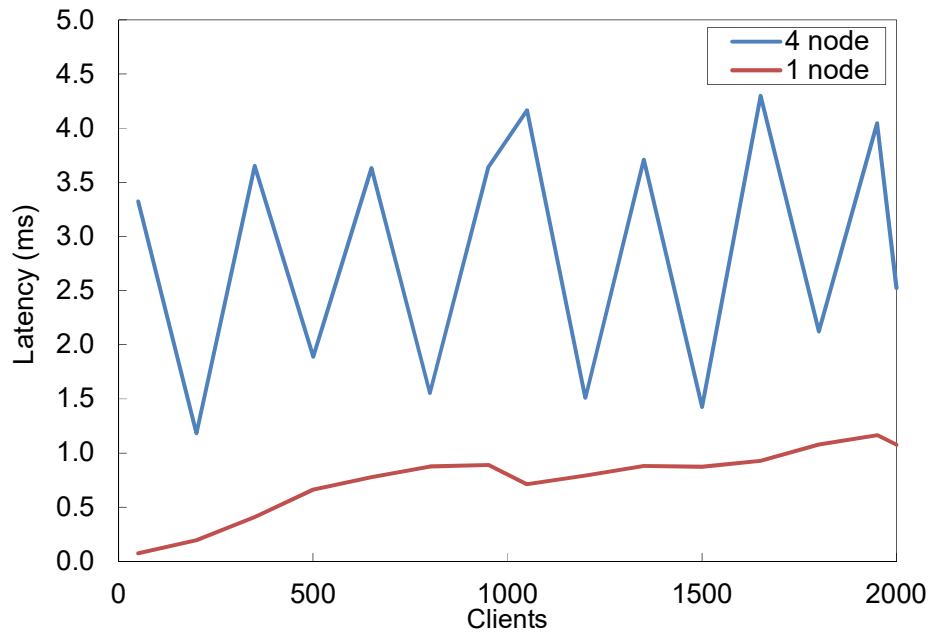


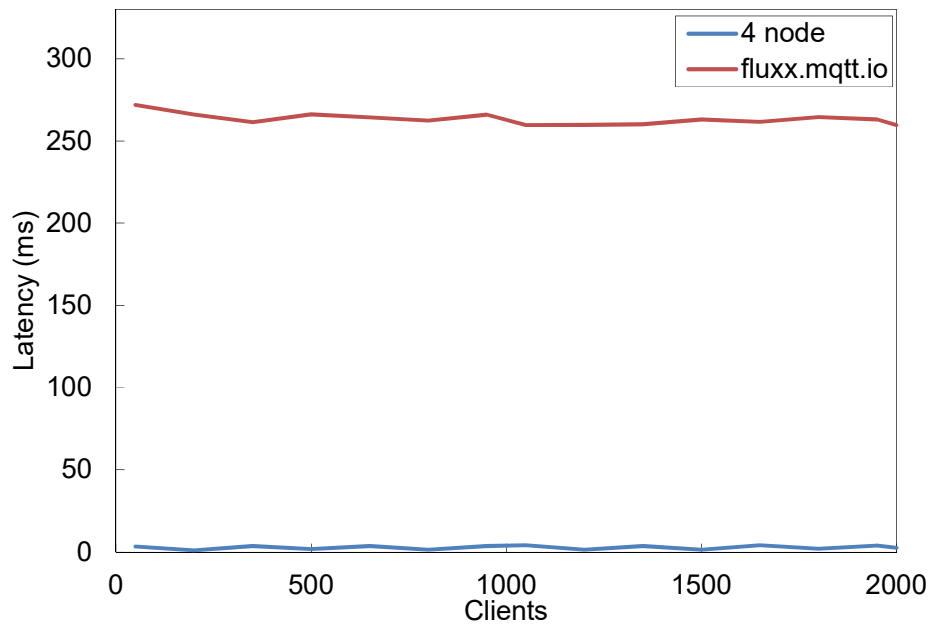
Figure 5.4 Message forwarding between backend servers

As depicted in Figure 5.6(a), the maximum value of worst-case latency of the 4-node broker cluster is 42 milliseconds, which is 32.2 milliseconds higher than that of a single node broker. The maximum worst-case latency of the single node broker is 9.8 milliseconds. It is observed that the volume of the load has no direct impact on the worst-case latency.

From the observation in Figure 5.6(b), the broker cluster has an overall higher worst-case latencies compared to the single node broker. The end-to-end latency values of the HiveMQ cloud broker are continuously above the latency of 300 milliseconds, surges to 900 milliseconds when 1800 clients are connected, and drops to 477 milliseconds afterward. The cloud broker has a worst-case end-to-end latency of 900 milliseconds when 1800 clients are connected.

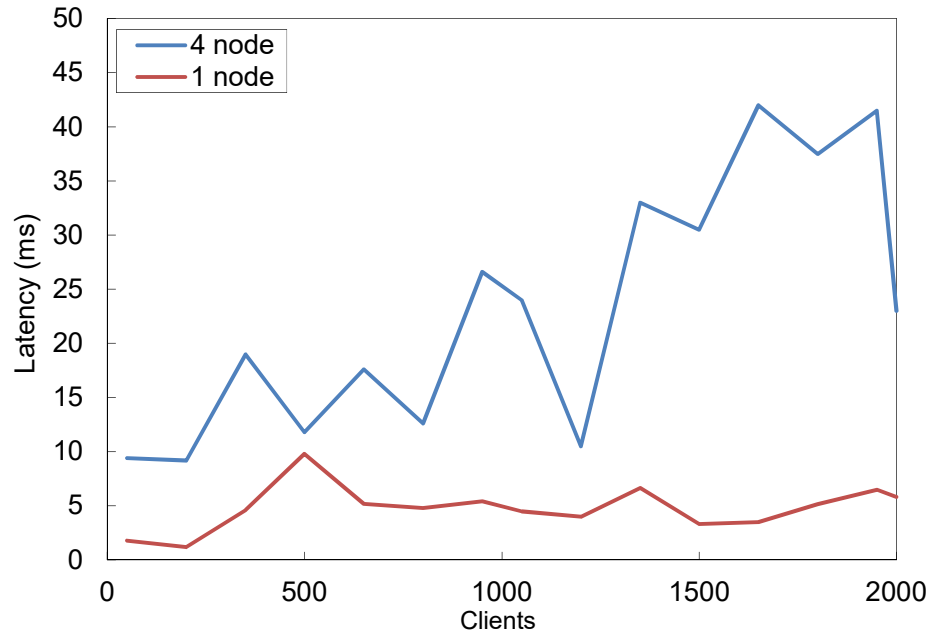


(a) 4 node and single node broker

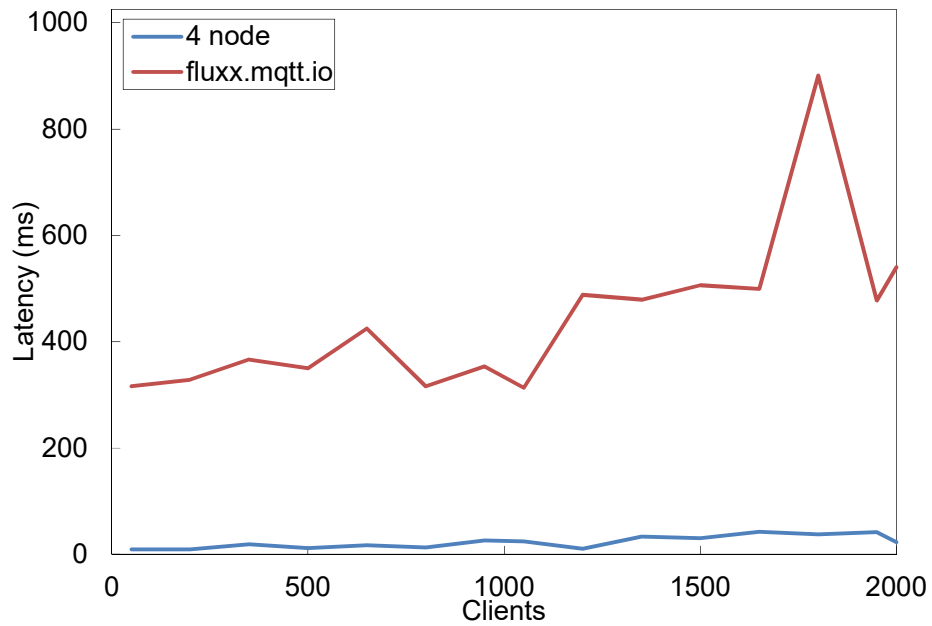


(b) 4 node and HiveMQ cloud broker

Figure 5.5 Average end-to-end latencies with increasing clients



(a) 4 node and 1 node broker



(b) 4 node and HiveMQ cloud broker

Figure 5.6 Worst case end-to-end latencies with increasing clients

As depicted in Figure 5.7, the data distribution of the latency values does not follow a normal distribution. The experimental data distribution skews to the right. This is because most of the measurements (80% according to the CDF

plot in Figure 5.9) are less than or equal to 3 milliseconds, while the rest of the data are relay latencies between broker nodes. The histogram data within the lower half hump of the density curve is close to zero, with values between 1 and 8 milliseconds. Based on quartile information in Figure 5.8, only a few outlier values are greater than 15 milliseconds. The Q-Q plot shows fat tails in small latency values to the left of the regression line. As depicted in the CDF plot in Figure 5.9, 95% of values fall between the range of 1 and 8 milliseconds, confirming that the routing delays between the broker cluster nodes are small. Also, a few sporadic values of 30 to 44 milliseconds appear. The P-P plot in Figure 5.10 presents a behavior quite similar to the CDF plot. The measured values are not aligned along a regression line, with a huge concentration on the smaller values. The probability statistics demonstrate high data locality for the test as only less than 5% of messages needs to be forwarded between backend servers.

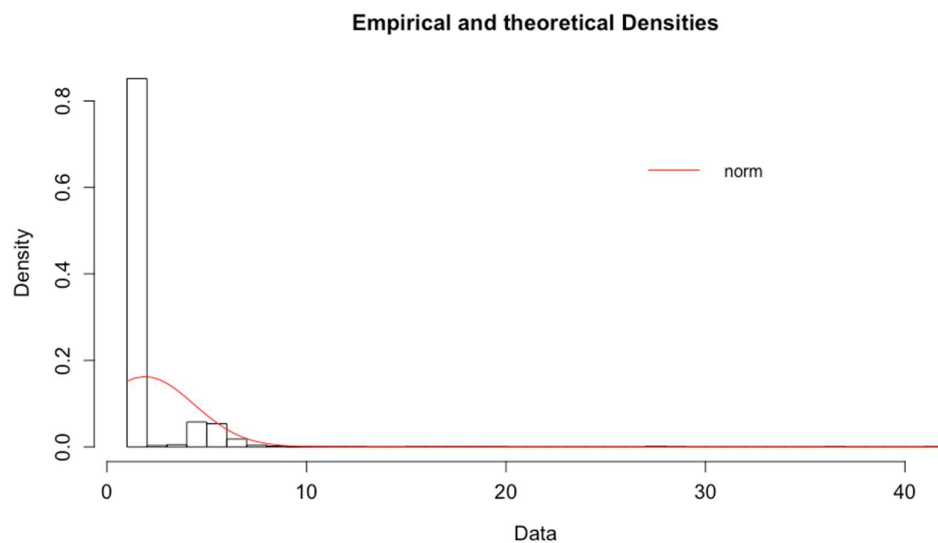


Figure 5.7 Latency histogram with a normal density curve

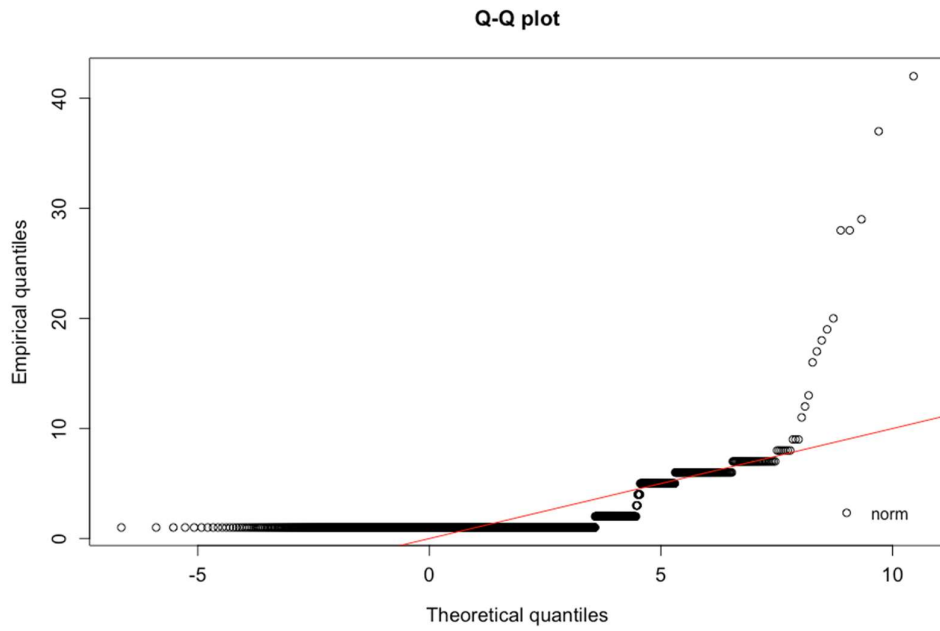


Figure 5.8 Latency quantile plot against normal probability distribution

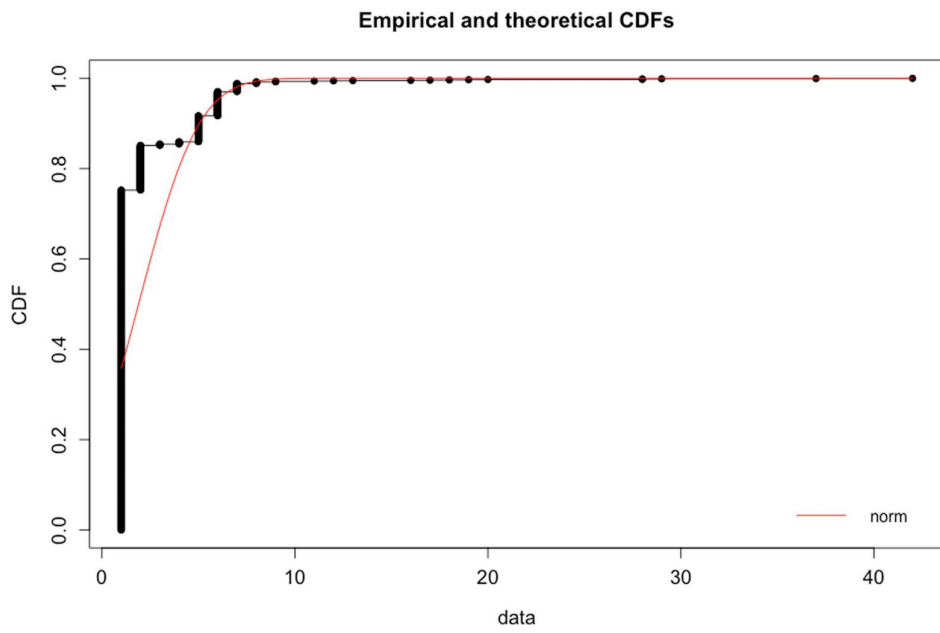


Figure 5.9 Latency cumulative distribution function with the normal probability curve

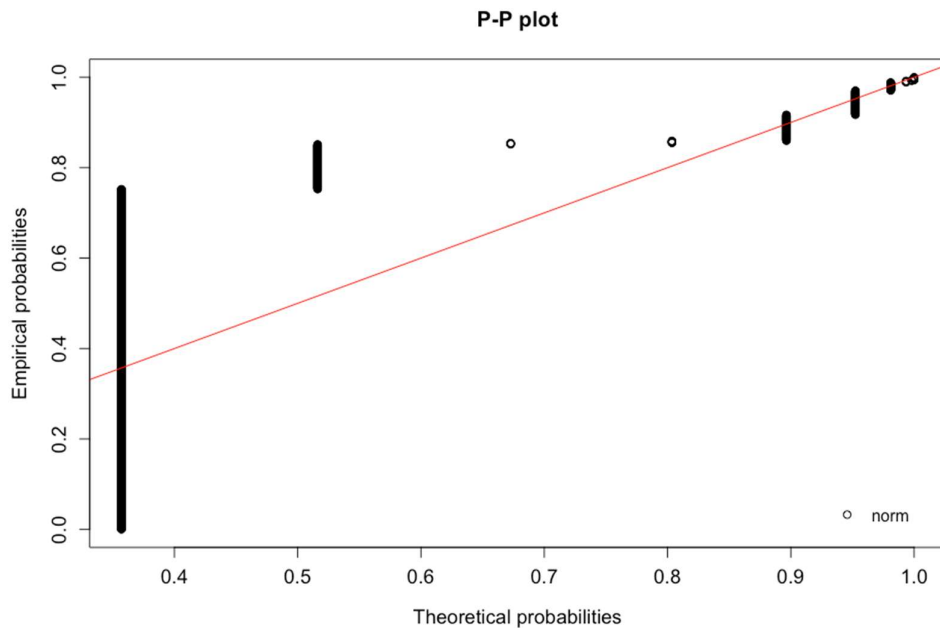
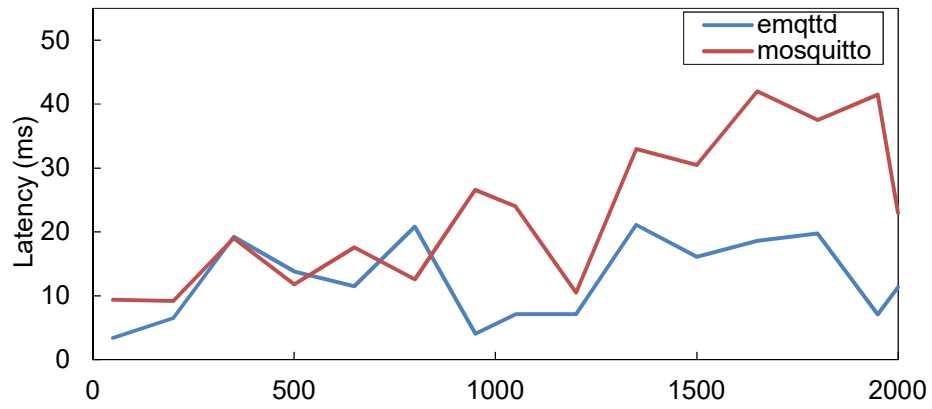


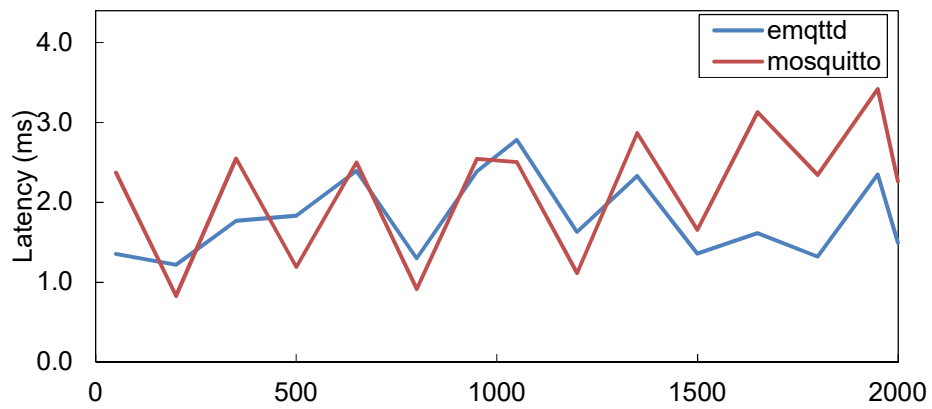
Figure 5.10 Latency probability plot against normal probability distribution

5.4 Microservice and Monolithic Broker Comparison

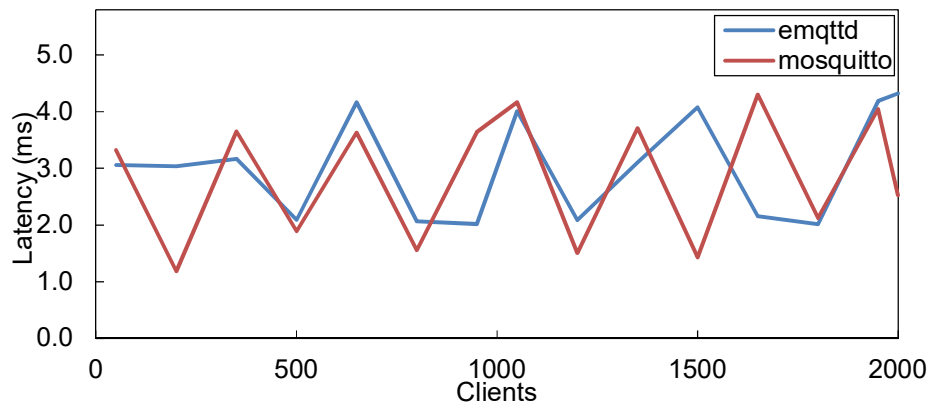
This section compares the performance of the microservice-based Mosquito broker cluster to the monolithic emqttd broker cluster (Emqtt.io, 2018). Emqtt broker implements the mnesia database (Mattsson et al., 1998) to store and replicate routing tables and uses the SF algorithm (Banno et al., 2017) to route messages between emqtt broker nodes. The monolithic implementation of emqttd integrates the broker and cluster component into a single software package. However, this research work implements the cluster component as a different set of microservice, which loosely couples with the mosquito MQTT broker. Figure 2.2 Monolithic and microservice implementation (Cicizz, 2019) the architectural differences between both implementations.



(a) Worst-case latencies



(b) Mean latencies



(c) Standard deviation latencies

Figure 5.11 End-to-end latencies performance comparison

The evaluations in Figure 5.11 show that the performance of the microservice broker cluster is almost comparable to its monolithic counterpart. However, the microservice-based Mosquitto broker has lesser performance than

the emqttd broker cluster, in terms of end-to-end transmission latency. The average latency of the Mosquitto broker cluster is 3.42 milliseconds, while the average latency of the emqttd broker cluster is 2.78 milliseconds. The worst-case end-to-end latency for the Mosquitto broker cluster is 42 milliseconds, which is higher compared to the 20.8 milliseconds latency of the emqttd broker cluster. The lower standard deviation of the emqttd broker cluster suggests that each latency values among the received messages are less dispersed. The results confirm that the implementation of the proposed broker cluster presents an overhead within the microservice layers.

The results in Figure 5.12 show the performance difference between both the monolithic and microservice implementations. The microservice broker cluster has a lower average throughput in terms of messages per second compared to the monolithic emqttd broker cluster. The average performance difference between both implementations is 3.7%. As depicted in Figure 5.13, the values of throughput degradation between both implementations are comparable. When the number of clients increases from 50 to 2000, the publish throughput of the microservice broker cluster drops by 8.47%, while the throughput of the emqtt broker cluster drops by 7.37%.

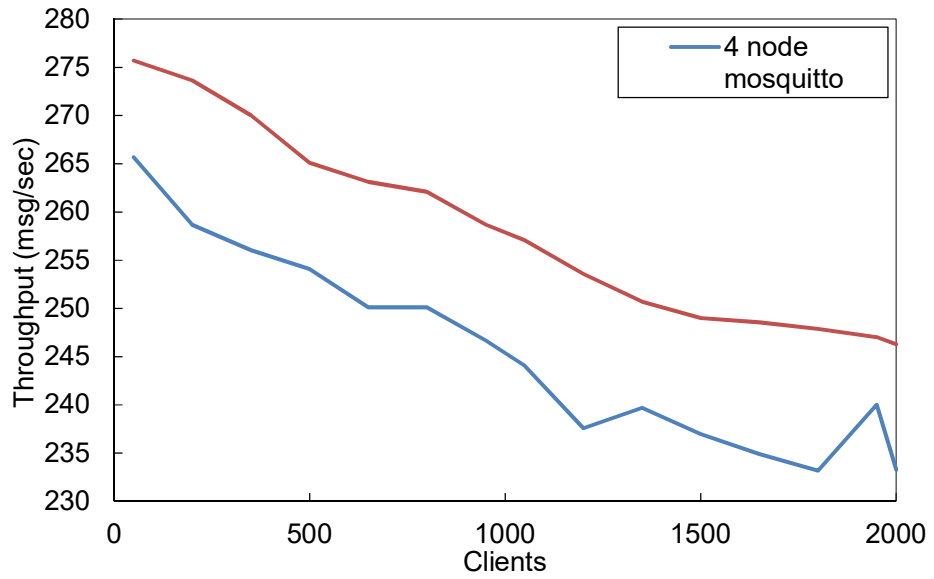
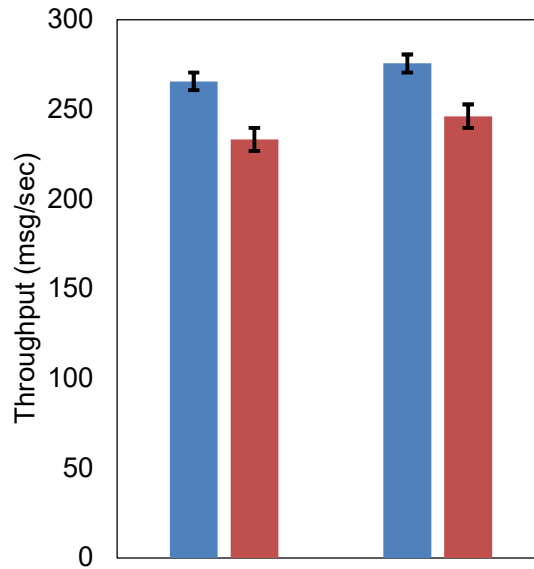


Figure 5.12 Average publish throughput (QoS 2)



Clients	4-node mosquitto cluster	4-node emqttd cluster
50	265.68 msg/sec	275.68 msg/sec
2000	233.28 msg/sec	246.28 msg/sec
Percentage drop	8.47 %	7.37 %

Figure 5.13 Throughput degradation (message per second)

5.5 Inter-message Jitter

Periodically and continuously transmitted messages will have different delays because consistent delay pacing cannot be guaranteed due to the asynchronous nature of distributed systems. This experiment measures the jitter value between consecutively received messages. The inter-message jitter is the discrepancy between the delivery times of two consecutively received messages (Luzuriaga et al., 2014). Inter-message jitter is computed through the following formula.

$$Jitter_{i,rcv_ts} = (msg_{i+1,rcv_ts} - msg_{i,rcv_ts}) - T \quad (5-3)$$

where msg represents a received message for one iteration, i is the message for the current iteration ($1 \leq i \leq N$), N is the total number of clients, rcv_ts is the reception timestamps, and T is the constant value of the inter-message production period. T is 10 milliseconds in the experiment. Clock drift is not an issue for Equation (5-3) because both sending and receiving time stamps are generated on the same machine.

The evaluations compare the maximum, minimum, and 90th percentile values of the inter-message jitter against the increasing number of clients on multiple test runs. As depicted in Figure 5.15, the minimum jitter values go from 0.04 to 6.16 milliseconds. The results show that minimum jitter values for each test run are unaffected by the number of clients. As depicted in Figure 5.14, the maximum jitter values range from 14.7 to 33.8 milliseconds. The results show increasing trends in maximum jitter values when the number of clients increases. Instead of using the average values of each test run, which are heavily

affected by outliers, the 90th percentile draws a better representation for most of the jitter values. The results in Figure 5.16 show that the 90th percentile jitter for each test run falls between 11 to 15 milliseconds. The results suggest that the discrepancies between delivery times of two consecutively received messages are under 15 milliseconds.

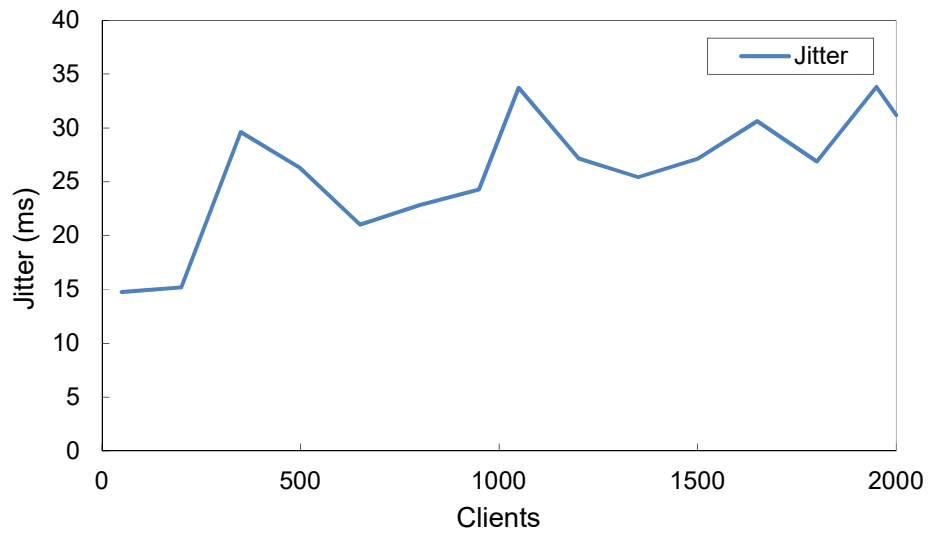


Figure 5.14 Maximum jitter of the broker cluster

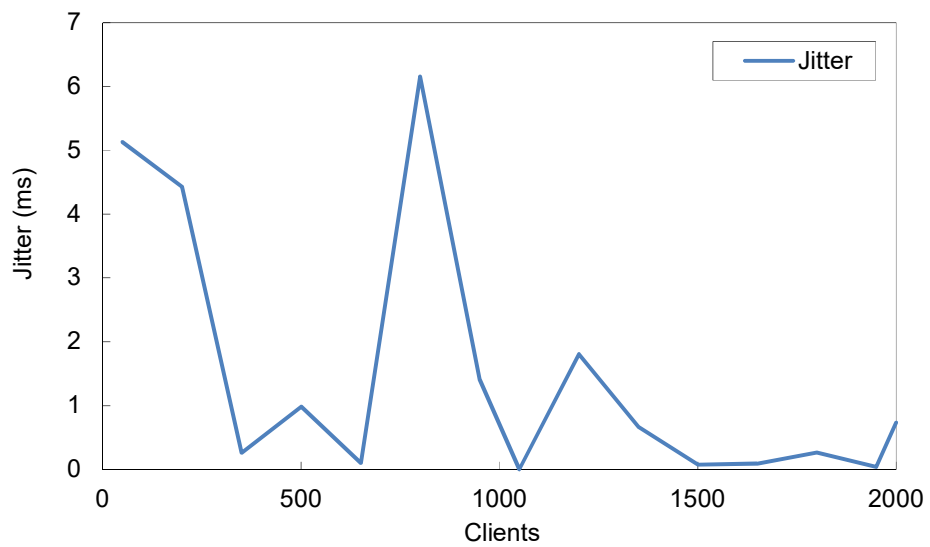


Figure 5.15 Minimum jitter of the broker cluster

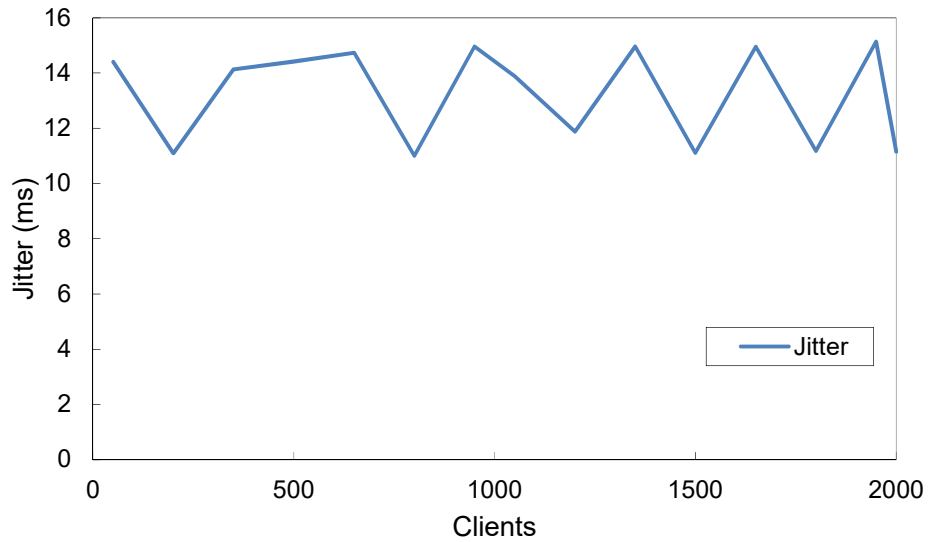


Figure 5.16 Percentile 90 jitter of the broker cluster

5.6 Evaluating Publication Retransmission

To verify the effect of publication retransmission, the throughput measurements before and after broker failure are evaluated. Figure 5.17 depicts the fail test experiment for the broker cluster. Server C and D are killed while after 30 seconds of processing the workload. The client test program monitors the throughput and end-to-end latency before and after the failure of one broker.

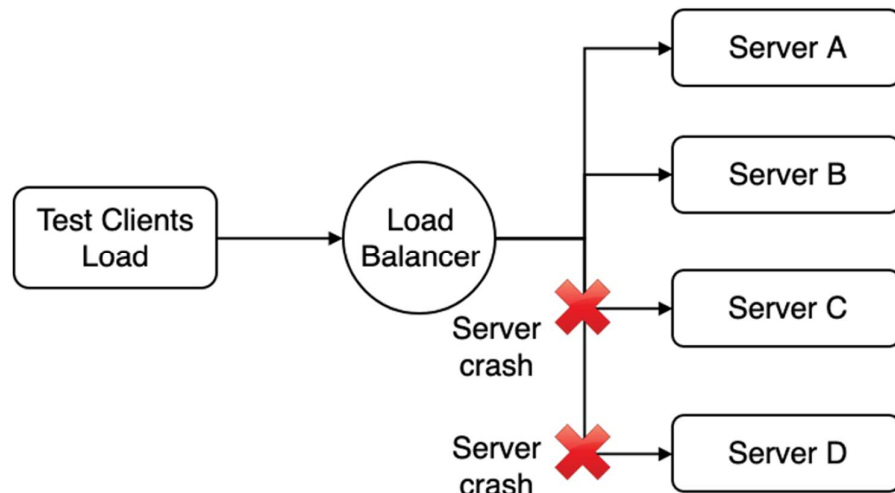


Figure 5.17 Fail test for the broker cluster

5.6.1 Throughput

Figure 5.18 depicts the throughput in terms of the number of messages per second for the test environment. The throughput of the broker cluster is stable when the first server failure happens. Throughput decreases by 7.3% when a server crash occurs. For the single node broker, the throughput decreases to zero, showing that the messaging service completely stops. After 5 seconds of stopping the messaging service, the single-node service is recovered. The throughput of the broker cluster remains stable before and after the server failure. This suggests that the broker cluster can maintain the availability of the MQTT service and gracefully handle the degradation of performance, during broker failure.

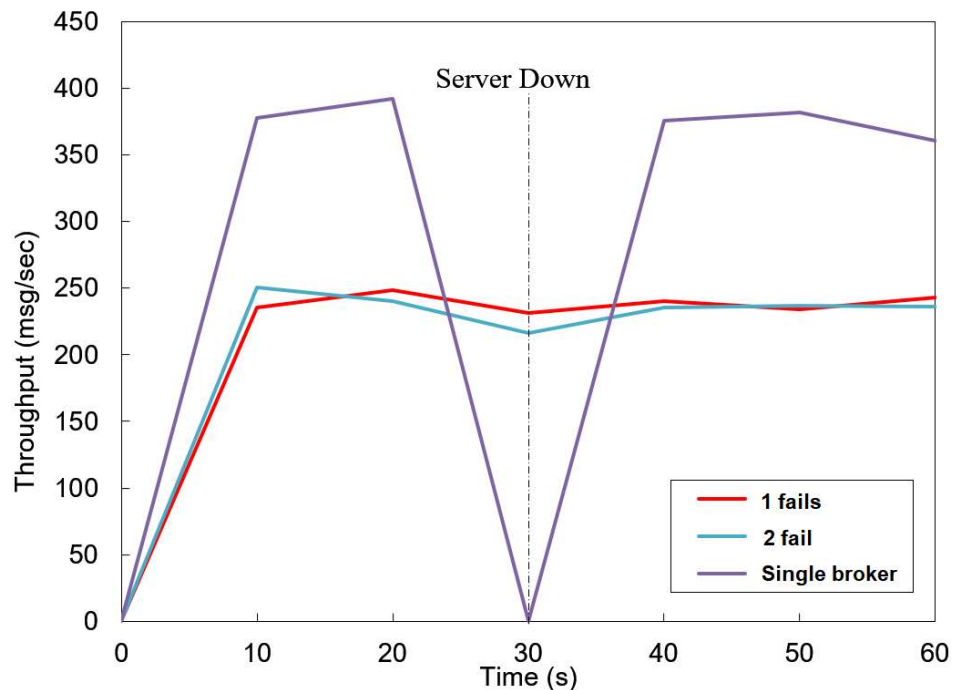


Figure 5.18 Message throughput for fail test

5.6.2 Inter-message Jitter

The test script runs with 200 subscribers for 1 minute so that the routing state has enough time to converge. The test clients are set to have a keepalive interval of 200 milliseconds. After that, one node and two nodes are deliberately made to fail respectively in two different tests when the publishers start to send the messages. The average measurements of 5 test runs are presented. The results show the jitter measurements between each consecutively received message.

The broker cluster can provide message loss tolerance for MQTT clients. The clients previously held by the failed broker automatically reconnect to other brokers. After one node fails, three remaining broker nodes take over the subscriptions of the reconnected clients. The broker cluster can recover all the messages published, during the period of failover, to the reconnected subscribers.

From the observations, the subscribers do not receive any duplicate messages during failover. This is because the retransmission process will filter and send the failed message to the relevant online brokers. Therefore, a subscriber will not receive the same message again. Also, typical IoT deployments expect about the same order of magnitude for the number of publishers, the number of topics, and the number of subscribers (Happ et al., 2017; Rotaru et al., 2017). There will be no duplicate during failure retransmission if each subscriber subscribes to a different partition of the topic, as described in the scenario in (Scalagent, 2014).

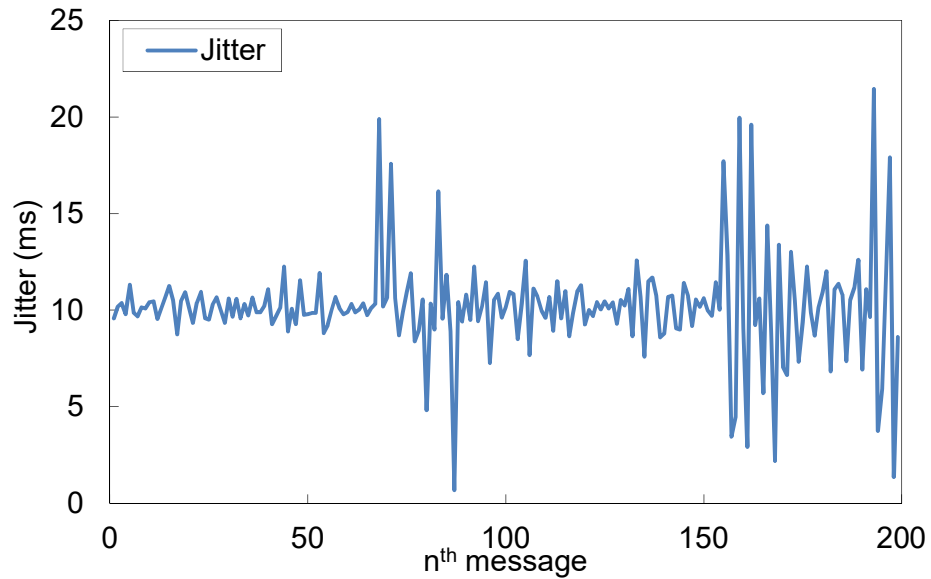


Figure 5.19 Jitter under normal condition

The inter-message jitter values are plotted against the message received in ascending order. The jitter values range between 0.67 milliseconds and 21.45 milliseconds when no failure occurs, as depicted in Figure 5.19. Some spikes in jitter values are because of the inter-cluster message routing delay. These spikes appear to be large because the jitter values of the other messages without delay are small.

As depicted in Figure 5.20, a large jitter spike occurs because of message redelivery by the broker cluster. The maximum jitter value for one failed node is 210.69 milliseconds. The maximum jitter value for two simultaneous failed nodes is 215.44 milliseconds, as depicted in Figure 5.21. The differences between the maximum jitter values for both tests with one and two failed nodes are almost comparable. This shows that the broker cluster can maintain almost the same degree of retransmission performance regardless of the number of node crashes. The results show no significant jitter spikes after the occurrence

of the first large spike. The maximum jitter value for the retransmitted message is 23.58 milliseconds. This suggests that the buffered publication messages are delivered successfully without significant inconsistencies in message delay.

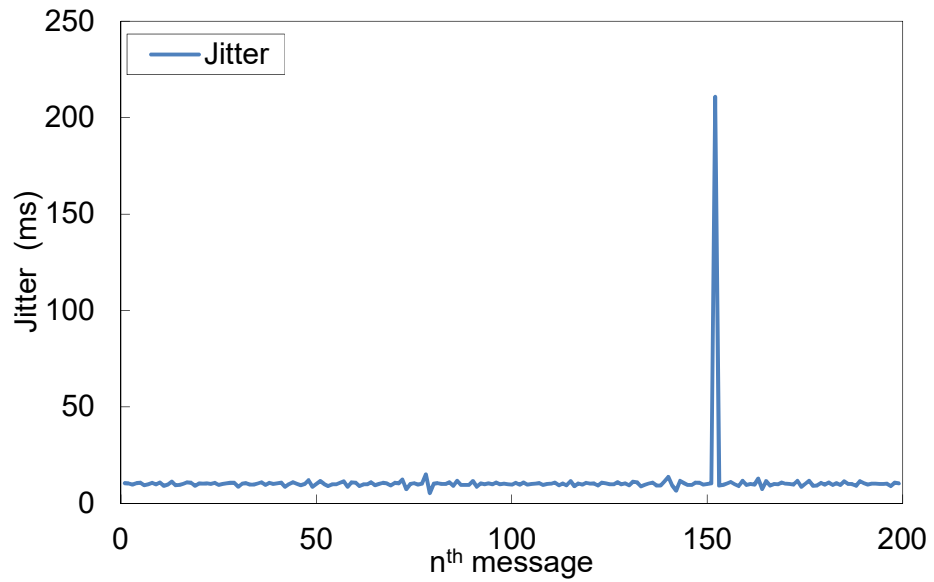


Figure 5.20 Jitter under one fail node

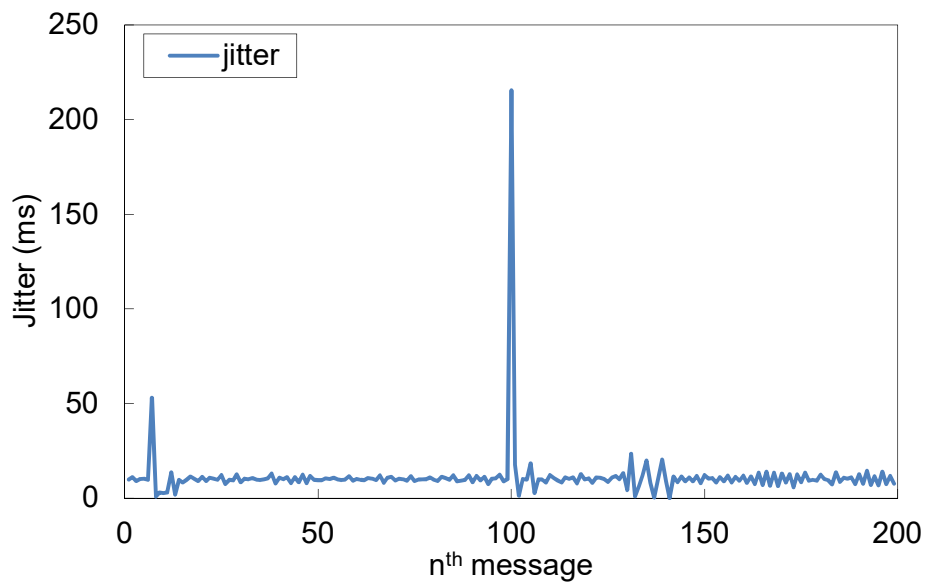


Figure 5.21 Jitter under two fail nodes (at same time)

5.6.3 Latency

As depicted in the histogram in Figure 5.22, major portion latency values are larger than 5 milliseconds. As compared to the latency values during normal operation shown in Figure 5.7, there are a greater number of high latency values due to message recovery. The data distribution is concentrated between 0 to 9 milliseconds with a few outliers of 18 to 44 milliseconds. Based on quartile information in Figure 5.23 only a few outlier values are greater than 15 milliseconds. Also, the majority of smaller values fall within 3 milliseconds when compared to the regression line. The distribution plot depicted in Figure 5.24 shows that 95% of values are between 1 and 10 milliseconds, which confirms that most of the latency values are below 10 milliseconds. Also, a few unusually high of 32 to 40 milliseconds appear. The P-P plot in Figure 5.25 presents a behavior similar to the CDF plot. The measured data is concentrated on a smaller range of values.

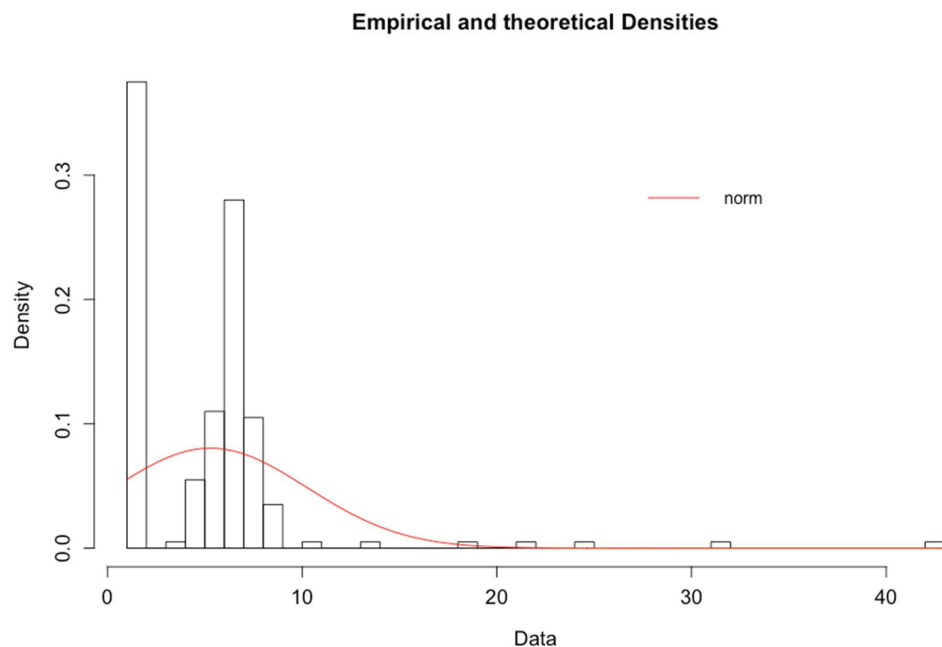


Figure 5.22 Latency histogram with a normal density curve

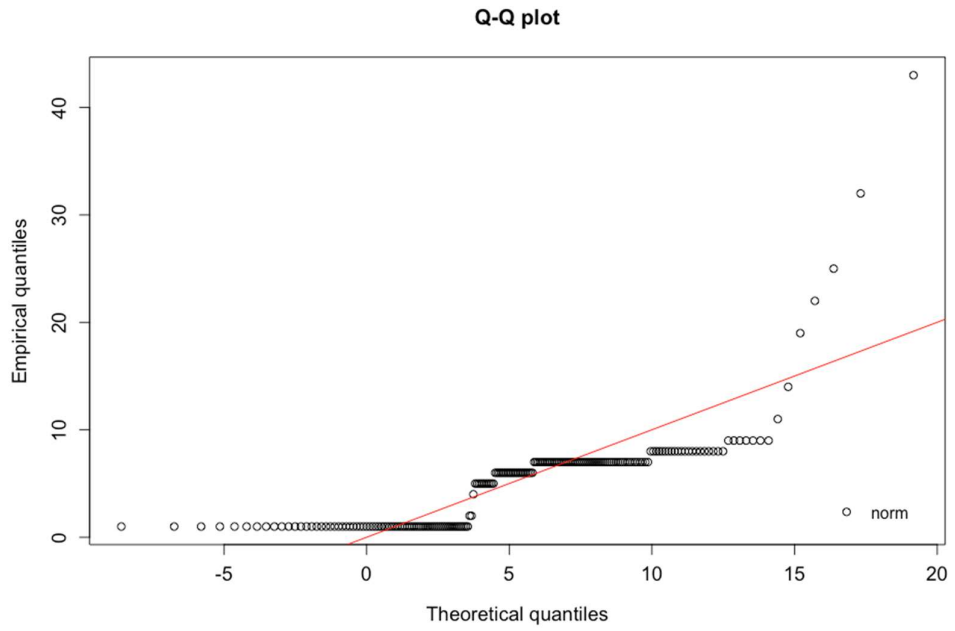


Figure 5.23 Latency quantile plot against normal

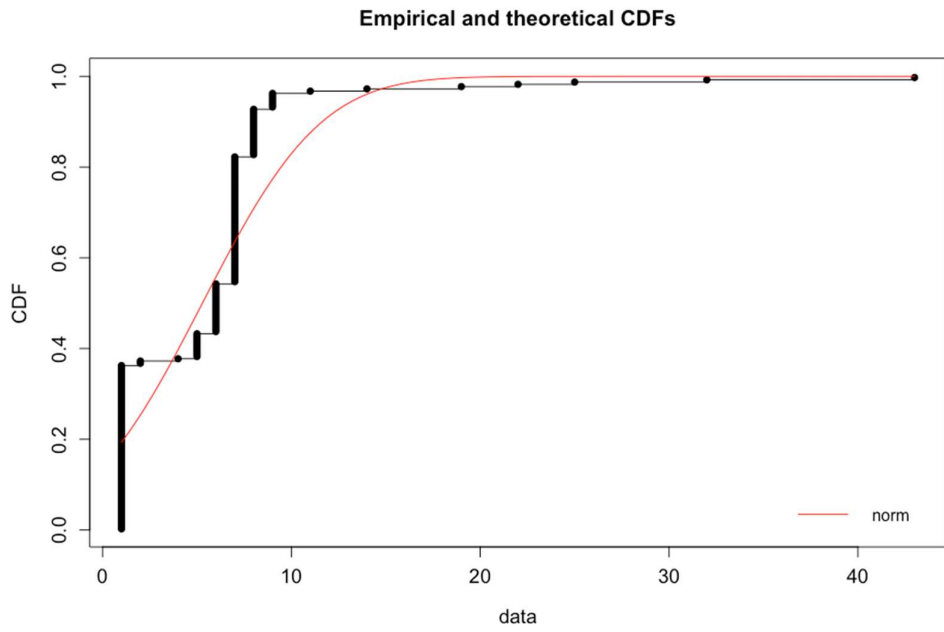


Figure 5.24 Latency CDF plot with a normal probability distribution curve

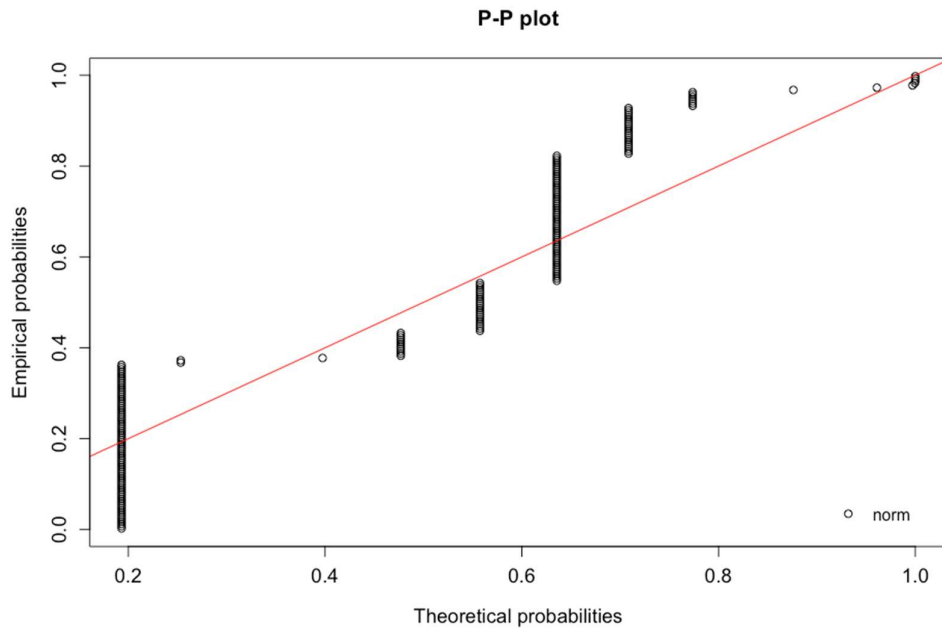


Figure 5.25 Latency probability plot against normal

Next, the extra recovery latencies, during failover of one broker node, are evaluated. The latency incurred in recovering the missed messages is dependent on the keepalive interval of the client to the broker. The keepalive interval is the time between each MQTT live-check message sent from a client to the broker when a connection is established. When the keepalive interval increases, the latency for getting missing messages increases.

As depicted in Figure 5.26, the data distribution of the latency values does not follow a normal distribution. The experimental data distribution is bimodal with values concentrated on the left and right. The histogram data within the middle of the density curve is zero. Based on quartile information of the Q-Q plot in Figure 5.27, there are larger values than expected when compared to the regression line. The regression line intercepts values between 0 and 250 milliseconds. As depicted in the CDF plot in Figure 5.28, 76% of

values are between 0 and 3 milliseconds. Also, 24% of the latency values are larger than 200 milliseconds as a result of message retransmission. The P-P plot in Figure 5.29 presents a behavior quite similar to the CDF plot, with more concentration of data distribution on the left and less concentration of data on the right.

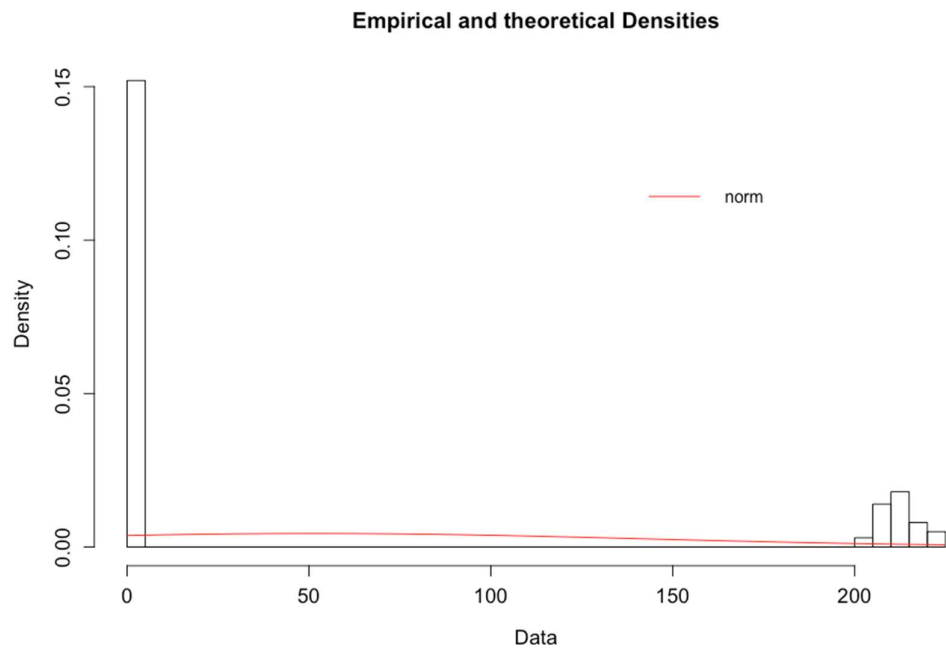


Figure 5.26 Latency histogram with a normal density curve

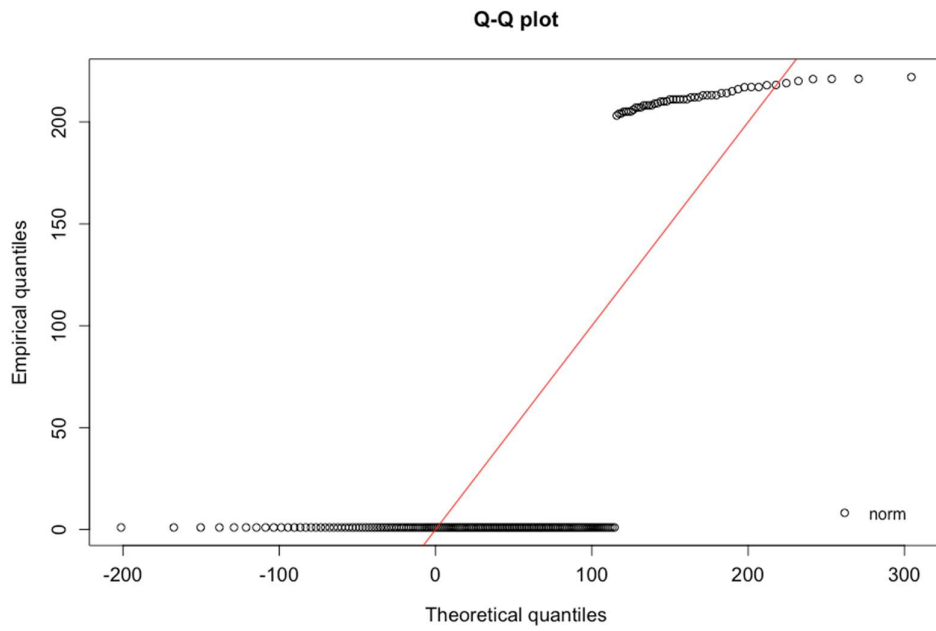


Figure 5.27 Latency quantile plot against normal

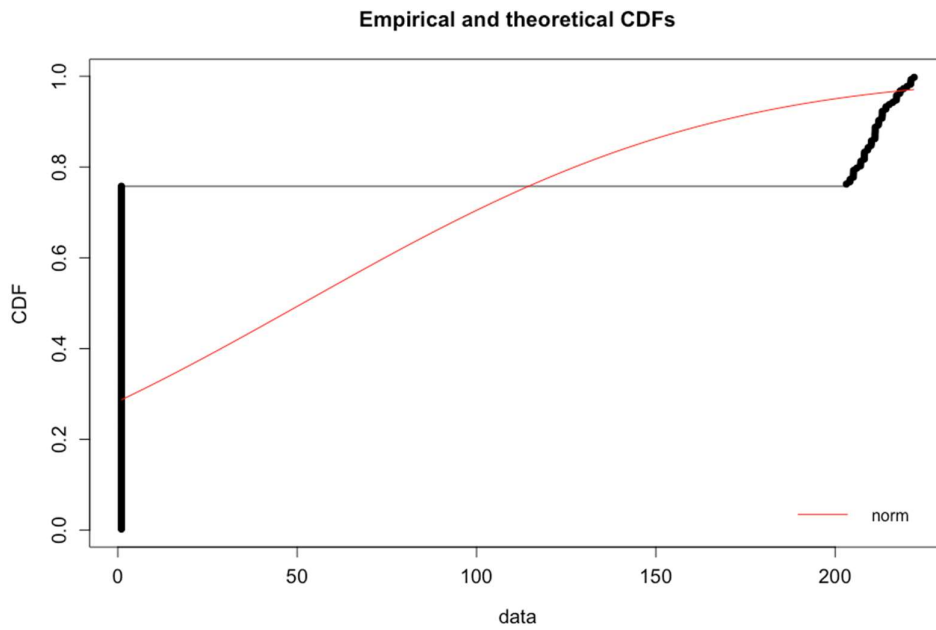


Figure 5.28 Latency CDF plot with a normal probability distribution curve

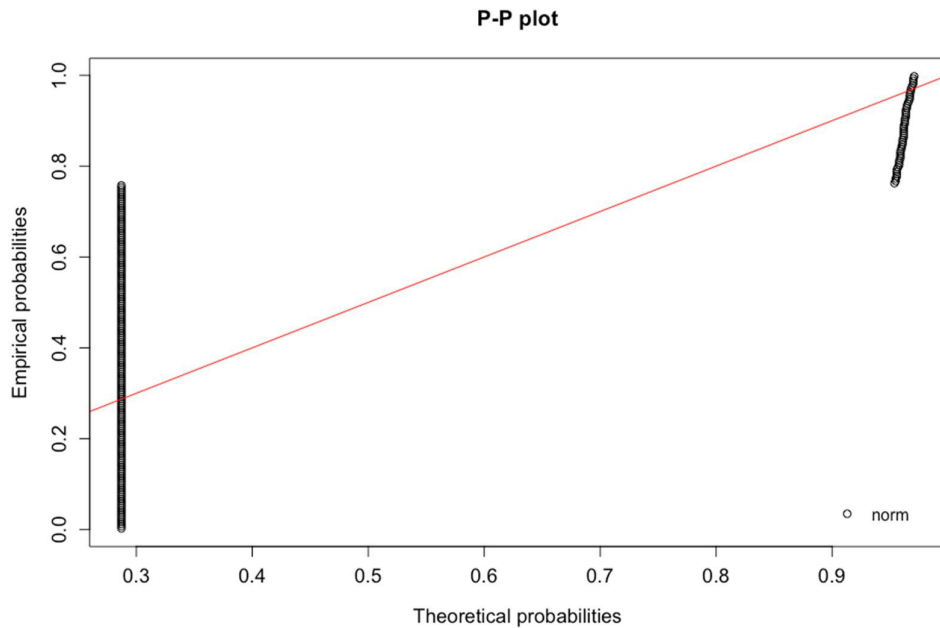


Figure 5.29 Latency probability plot against normal

Next, the extra recovery latencies, during failover of two broker nodes, are evaluated. As depicted in Figure 5.30, as much as half of the experimental measurements are larger than 200 milliseconds. This shows that almost half of the messages have failed to be delivered at first and are redelivered by the system with added delay. On the left of the histogram, the values range from 0 to 25 milliseconds. On the right, the values are larger than 200 milliseconds. Based on quartile information in Figure 5.31, the data distribution is bimodal. Also, there are more data with smaller values than expected when compared to the regression line. Based on the CDF plot in Figure 5.32 50% of values are between 1 and 25 milliseconds and 50% of values are larger than 210 milliseconds, confirming that the data distribution is bimodal. Also, a few unusually high of 30 to 44 milliseconds appear. As depicted in Figure 5.33 the measured values are not aligned along a regression line. The data distribution was split into half. Half as much of the latency values are above 210

milliseconds, indicating the split of the failed messages that are recovered after the failure of two broker nodes.

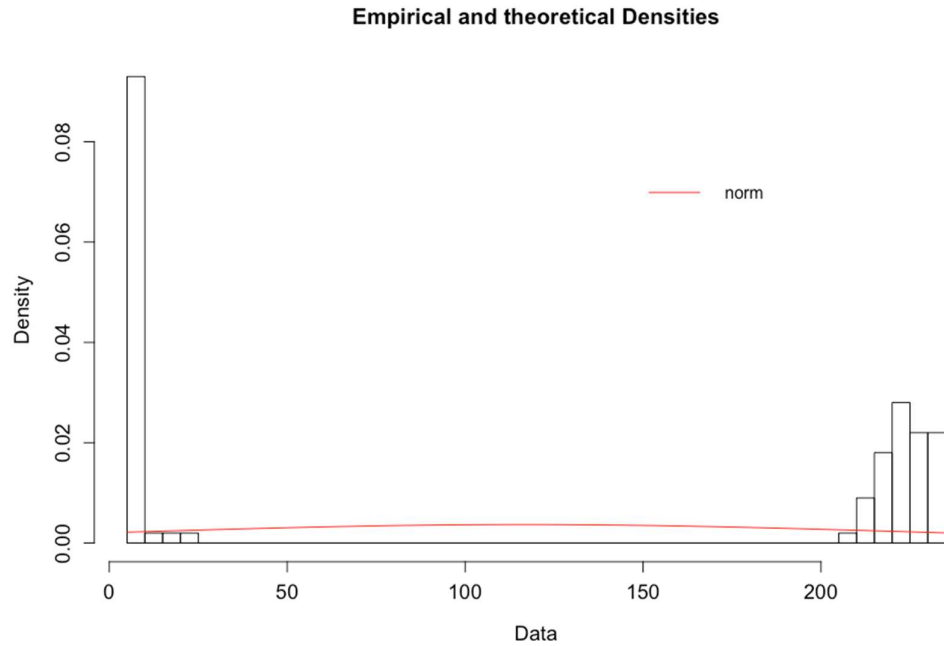


Figure 5.30 Latency histogram with a normal curve

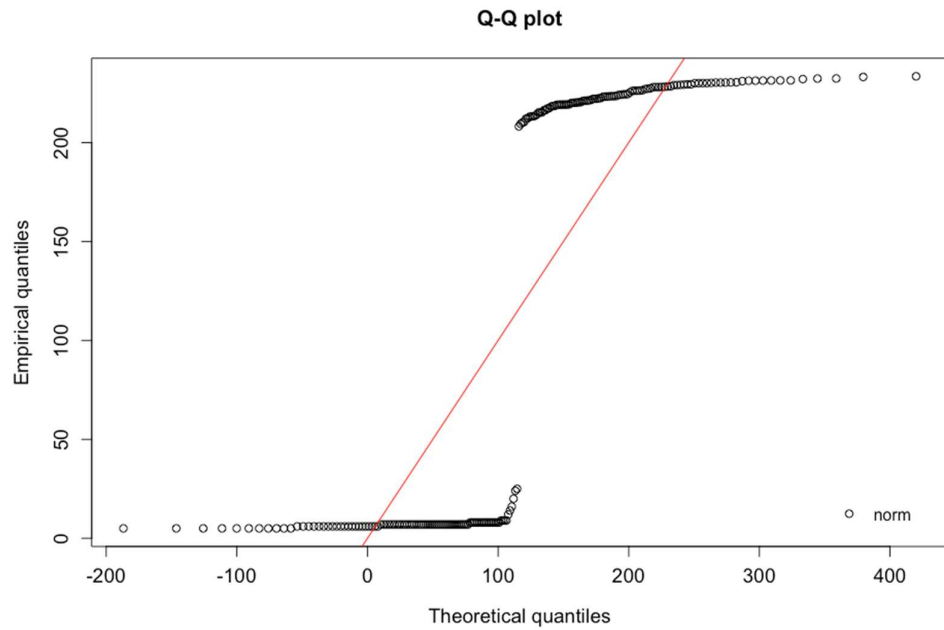


Figure 5.31 Latency quantile plot against normal

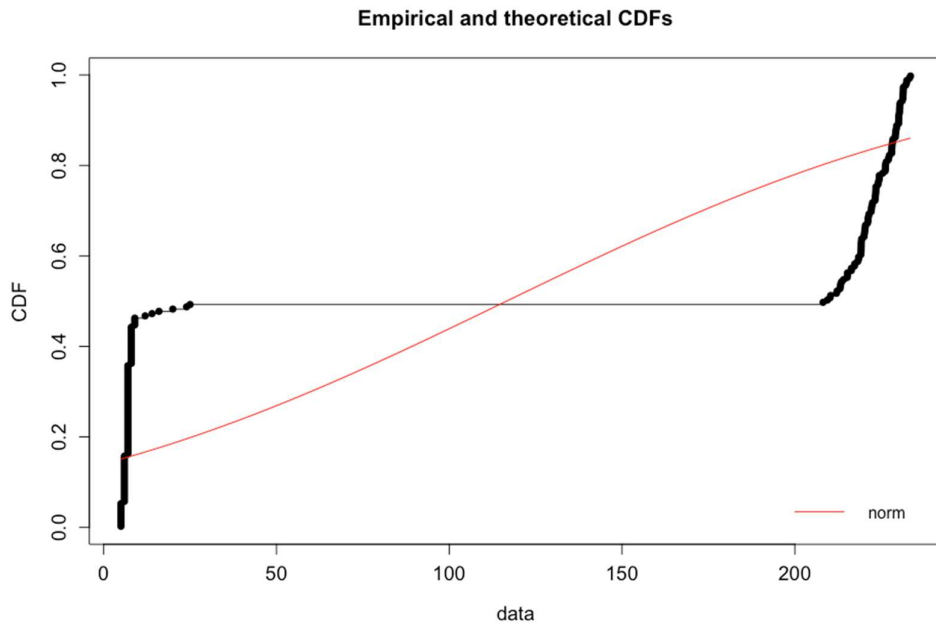


Figure 5.32 Latency CDF with a normal probability distribution

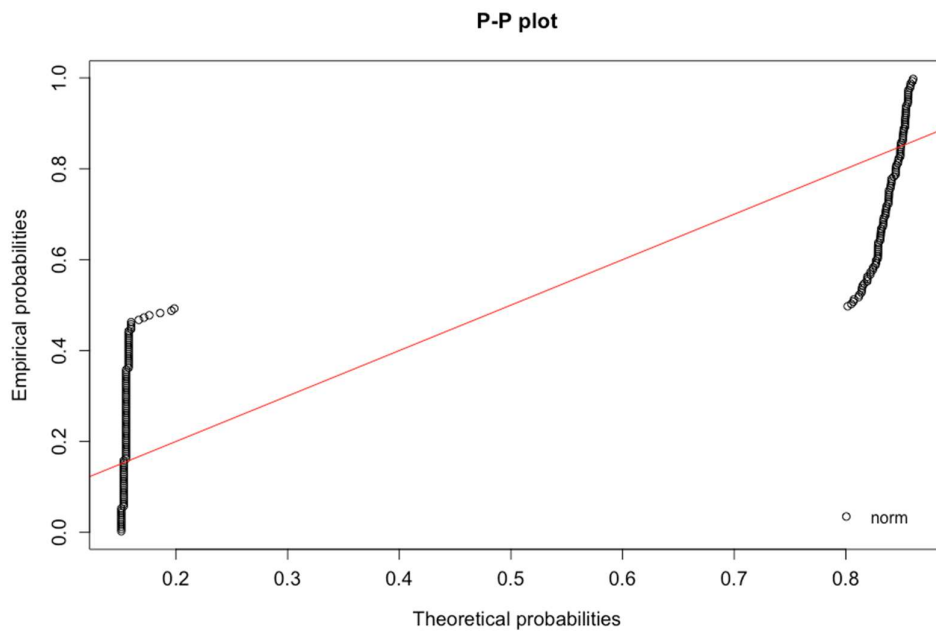


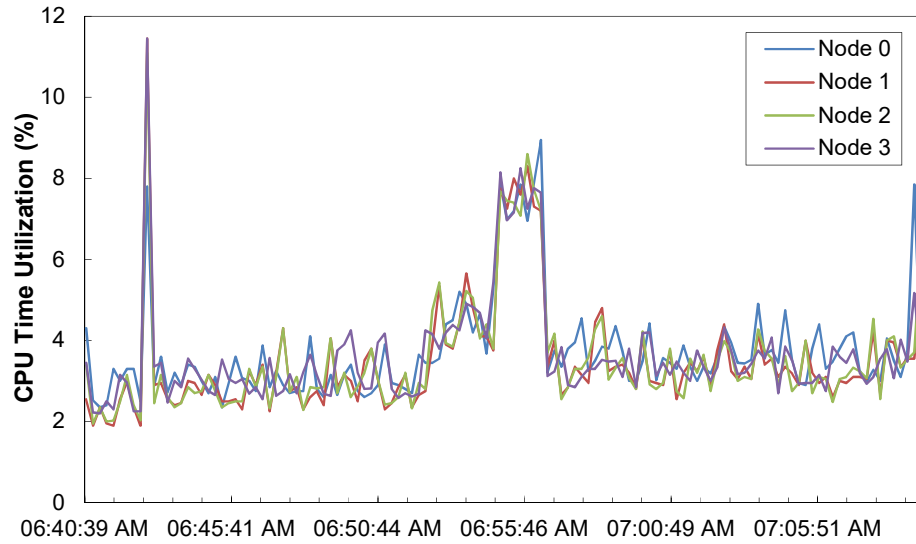
Figure 5.33 Latency probability plot against normal

5.7 Resource Usage

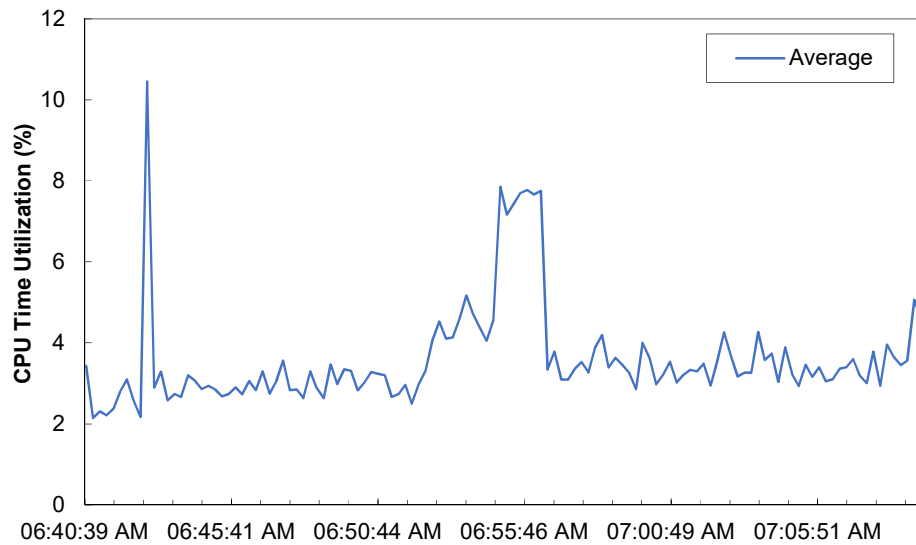
To evaluate the system metrics of the broker cluster, Node Exporter and Prometheus are used to obtain the host's system metrics such as CPU usage percentage and RAM usage, when the workload is inserted. The evaluations shown here are tested using 2000 pairs of MQTT clients, with 32 bytes payload for each message, and a sending interval of 10ms between each publication.

5.7.1 CPU Utilization

Figure 5.34 depicts the time series of CPU utilization percentage when the number of connected client devices increases for the clustered broker. Figure 5.34(a) shows the results for four different broker nodes, while Figure 5.34(b) shows the moving average of four nodes. As depicted in Figure 5.34(b), the CPU utilization percentage peaks at 10.45% during the initialization of the docker container and the broker cluster. The CPU percentage is at an average of 3% when the broker cluster is idle. Next, the CPU percentage rises to 4.7% when clients are subscribing, rises and oscillates around 8% when clients are publishing, and drops back to 3% afterward.



(a) Series of four individual nodes



(b) Series of average of four nodes

Figure 5.34 Time series of CPU time utilization for broker cluster

As depicted in Figure 5.35, the CPU percentage of the single node broker spikes at 2.8% before dropping to idle at around 1.45%. The CPU utilizes a maximum of 3.4% when the broker is handling the MQTT workload. The results show that the extra cluster server component costs about 2% of CPU overhead.

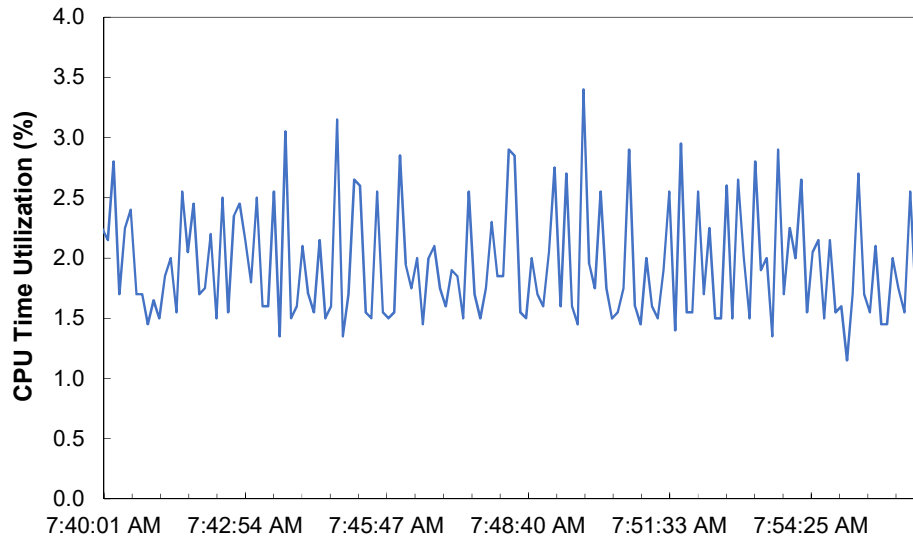


Figure 5.35 Time series of CPU time utilization for single node broker

The CPU utilization percentage is irregular when clients are subscribing and publishing. The reason behind this is due to idle times of software processes, which are blocked while waiting for input-output (I/O) operations to complete (Bovet and Cesati, 2005). The CPU does not spend clock cycles during I/O wait times. It can be observed that the CPU has spent a lot of time being idle because of the intensive number of I/O operations of the network sockets.

Figure 5.36 depicts the CPU time utilization percentage for the MQTT load test under the condition of two simultaneous failed nodes. The CPU percentage is at an average of 3% when the broker cluster is idle. Next, the CPU percentage rises to 4.6% when clients are subscribing. After all of the subscribers are registered in the system, the CPU percentages of two failed broker nodes (Node 2 and Node 3) drop to 0. CPU percentage of Node 0 and Node 1 rise and oscillate around 7.6% when the clients started publishing. The CPU percentage rises to 4.7% when the disconnected clients came back

subscribing to two remaining broker nodes. Retransmission of failed message happens then the CPU percentage rises and oscillates around 7.6%. With comparison to the CPU utilization under normal conditions in Figure 5.34(a), the system encountered a CPU utilization overhead right after the published messages are delivered for the first time. The results also show that both first-time publishing and the retransmission process have about the same CPU utilization peaks at around 7.6%.

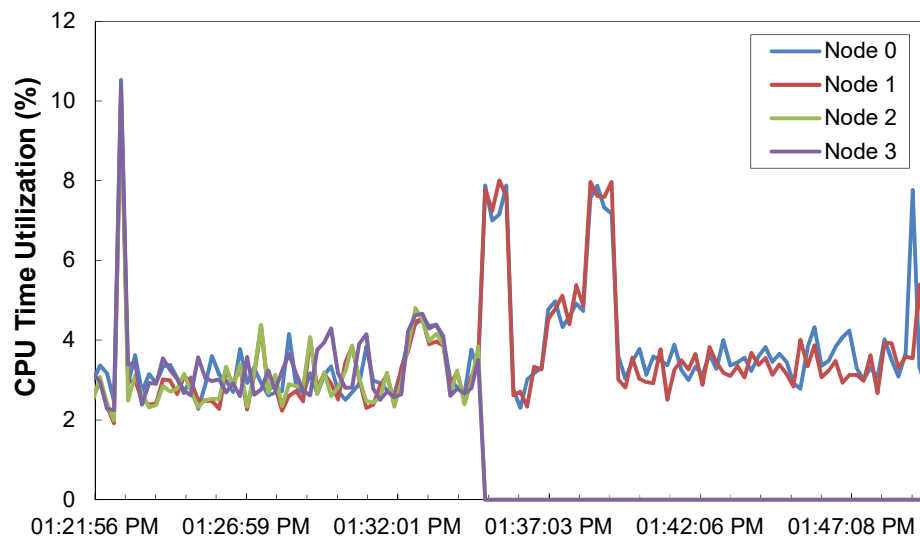
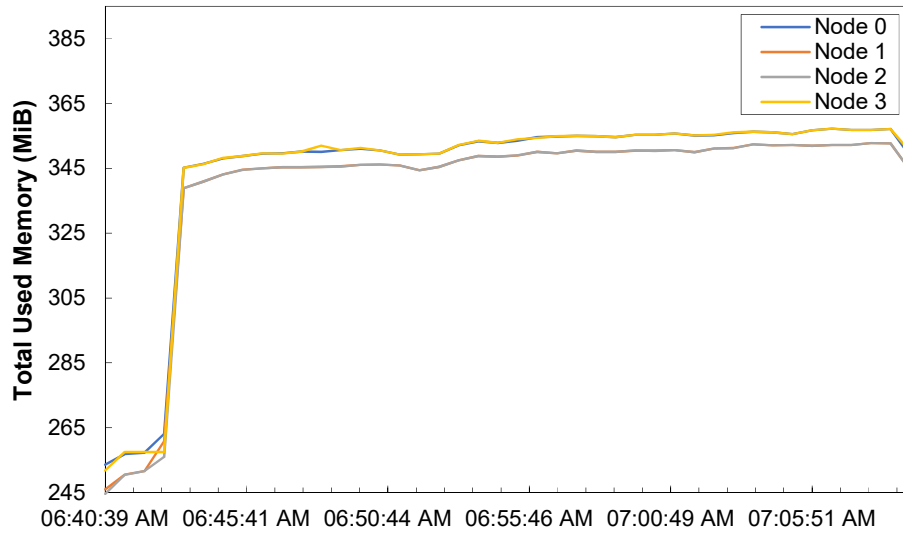


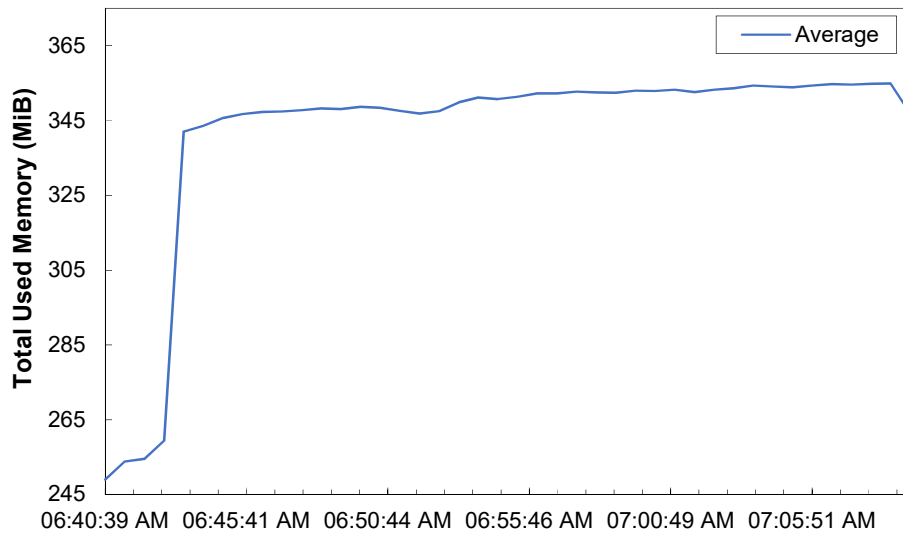
Figure 5.36 Time series of CPU time utilization for two failed brokers

5.7.2 RAM Usage

The Raspberry Pi 3 Model B board has 957 Mebibyte (MiB) of total usable memory. This evaluation demonstrates the impact of increased workload on the memory usage and the memory overhead incurred within the software implementation of the broker cluster over the single node broker.



(a) Series of four individual nodes



(b) Series of average of four nodes

Figure 5.37 Time series of total memory usage for broker cluster

Figure 5.37 depicts the total memory usage during the entire test run. The average memory used rises from 249 MiB to 347 MiB when initializing the broker cluster. Figure 5.37(b) demonstrates that the base memory footprint of the Docker-based broker cluster is 98 MiB on average. As the clients are subscribing, the memory slowly rises to 352 MiB and remains steadily below 355 MiB during the test run. The results in Figure 5.37(b) show that the average

memory is larger than 8 MiB (2.28% increase) when the number of connected MQTT client increases.

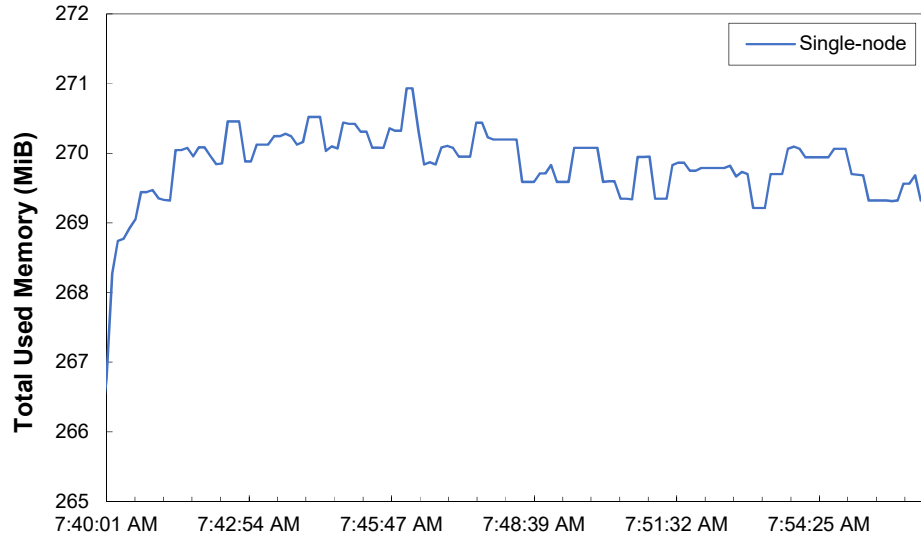


Figure 5.38 shows that the RAM usage for the single broker increases from idle memory of 268.74 MiB to 270.46 MiB (an increase of 1.72 MiB) as the clients are subscribing. The memory usage falls back to 269.32 MiB when the MQTT workloads are completed. As compared to the memory overhead of the single broker setup, the memory overhead of the broker cluster is higher due to the redundancy of the cluster component.

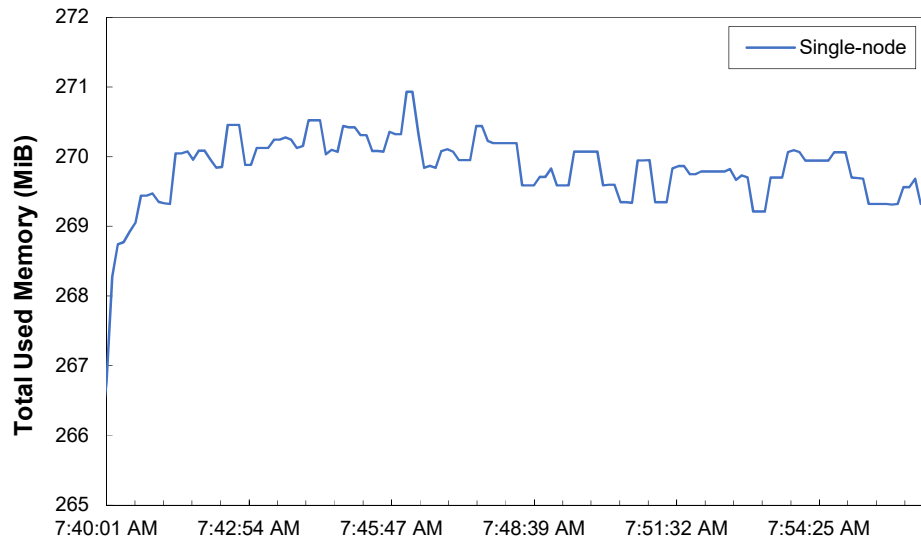


Figure 5.38 also shows the fluctuation in memory usage for the single node broker. This can be related to its CPU utilization shown in Figure 5.35, where the broker frequently went idle while waiting for the data stream in the I/O operation. After the broker completes some of the message deliveries and the CPU goes to idle, small portions of memory get to free up, which describes the continuous rise and fall of memory usage during the entire test run.

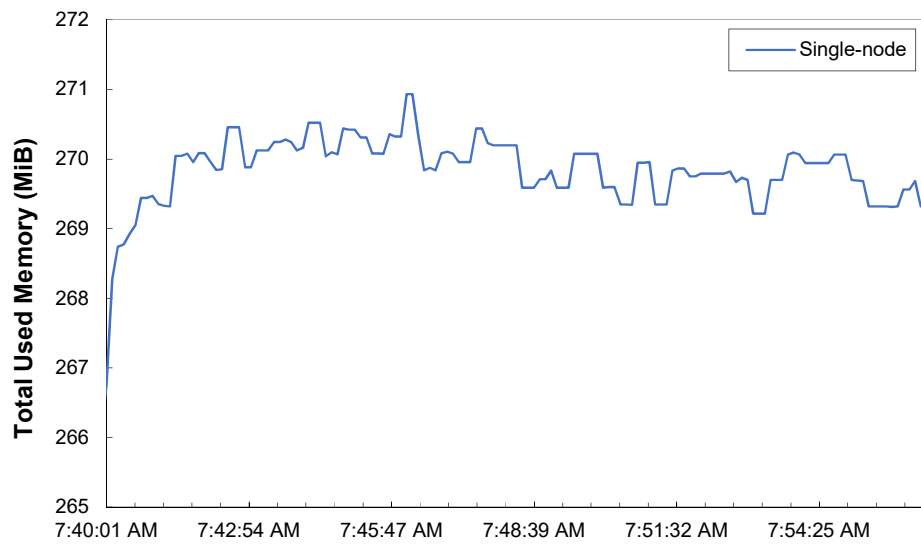


Figure 5.38 Time series of total memory usage for single-node broker

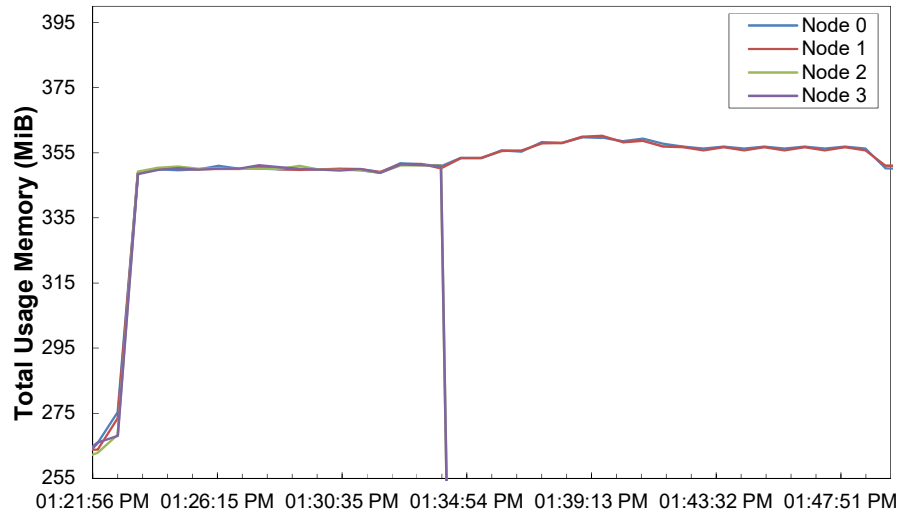


Figure 5.39 Time series of total memory usage under two failed brokers

Figure 5.39 depicts the total memory usage during the experiment for two simultaneously failed broker nodes. The memory usage rises from 260 MiB to 349 MiB when initializing the broker cluster. When the system is registering subscribers, the memory usage rises from 349 MiB to 351 MiB. Before receiving the publish messages, the memory usage of two broker nodes dropped because of node failure. When the system is receiving publish messages, the memory rises from 351 MiB to 353 MiB. There is an overhead as compared to the test under a normal condition in Figure 5.37 when the memory usage slowly rises to 358 MiB after two broker nodes have completed the delivery of publication messages. The results show that some of the publication messages that are failed to be delivered are stored in the memory in the form of a temporary queue. The system delivers all the message left in the temporary queue and delete each message in the queue after it is confirmed to be delivered. The total memory usage drops back to 351 MiB when all of the subscribers are disconnected.

5.8 Discussion

This section presents the discussion of the results evaluated in previous sections with regard to the performance, fault tolerance, and limitations of the proposed implementation. Since the dissertation focuses on crash failures of the backend brokers, the discussions of the results assume the load balancer is reliable and the system is secure.

5.8.1 Throughput

To support interaction between the large volume of devices, the broker needs to have horizontal scalability. The broker must minimize the degradation of performance with an increasing number of clients. The evaluations in Section 5.2 show that the broker cluster setup has a higher value of average throughput. However, the decline in overall throughput is not desirable, considering the cost of horizontal scaling and replication. The reason behind this is related to the relay elements present in the microservices layers and in between broker nodes. The relay element incurs an extra delay in publication time which reduces the average throughput. Therefore, the evaluations reveal that implementing load balancing and horizontally scaling MQTT brokers does not directly improve the performance of the system in terms of message throughput.

The evaluations in Section 5.2 also demonstrates the improvement in terms of the average publishing throughput of the edge-based MQTT broker over the cloud-based approach. The edge-based broker significantly outperforms the cloud broker in terms of average throughput. This occurs due to the high volume of traffic flow to the cloud broker. The average throughput

per client remains below 1.5 messages per second for the cloud broker. The benefits of an edge-based over a cloud-based MQTT broker are also demonstrated in (Laaroussi et al., 2018) (Zyrianoff et al., 2018) where the average throughput is higher for edge-based provisioning of MQTT. Therefore, it is clear that the edge node is beneficial for many data streaming-based IoT applications.

Even though the broker cluster has lesser performance in terms of average throughput, the evaluations demonstrate better scalability than the single broker. The evaluations in Figure 5.3(a) show that the performance degradation of the broker cluster is lesser (1.18%) as compared to the single broker (21.23%). This also shows the bottleneck when a large number of clients must transmit data to a single broker. The implementation of single-threaded implementation Mosquitto broker prevents multi-core utilization of the Raspberry Pi 3 Model B SBC, which features a quad-core processor. As the load increases, the single node Mosquitto broker presents a bottleneck that limits its scalability when the entire capacity of the SBC is under-utilized.

In summary, it can be concluded that the broker cluster scales better in terms of throughput when the workload increases. This shows that the broker cluster can handle workload increases better than a single node MQTT broker. A single node MQTT broker will perform more efficiently when the number of clients is small but scales rather poorly with increasing clients. The single-node broker shows more severe performance degradation, which explains the reason for the implementation of the MQTT broker cluster through horizontal scaling.

Also, the scalability regarding throughput degradation is independent of the service provisioning method.

5.8.2 Latency

For the edge brokers, average latencies increase with the number of client connections, as depicted in Figure 5.5(a). The average latencies of the cloud broker, as depicted in Figure 5.5(b), are more than 100 milliseconds and are very unpredictable. Servers with unpredictable latencies are generally avoided as it can affect IoT applications that are sensitive to delays.

The evaluations in Figure 5.6 show that worst-case latencies are independent of the number of client connections, but rather the degree of data locality. The worst-case latencies are higher when data locality is low. Using the broker cluster for transmission results in larger average and worst-case latencies, as depicted in Figure 5.5(a) and Figure 5.6(a). This is because the workload for the broker cluster has less data locality than a single node broker. Data locality is present when a publication reaches a broker that also happens to subscribe to a similar topic.

This is especially true for the subscription diffusion method implemented for event routing, as discussed in Section 4.4. The subscription diffusion routing incurs extra software latency to match publications to subscription and network latency to relay the publication messages to neighbor nodes. Similar observations are also shown in the work of ILDM (Banno et al.,

2017), where the authors tested different patterns of data locality workloads and confirm that higher data locality results in lower latency.

The setup does not have fine-grain control for data locality as all requests are routed based on the least connected algorithm in the load balancer. This is because publication messages need to be forwarded with one extra hop to the neighbor nodes when subscriptions are scattered across multiple brokers. Lower overall data locality in the workload causes messages to relay to other nodes which increases the end-to-end latency. The high transmission latency for some messages also suggests that publications are buffered in the Mosquitto broker before being processed because the Mosquitto broker only uses a single thread for message forwarding (Scalagent, 2014).

The worst-case end-to-end latency for the Mosquitto broker cluster is 40 milliseconds, which is not ideal for real-time messaging but is adequate for less demanding applications. Also, the Mosquitto cluster will provide more stable latency as variations between each latency measurement are minor and the standard deviation is lower.

5.8.3 Performance overhead of the microservice-based broker cluster

Section 5.4 presents the latency evaluations of the microservice-based Mosquitto cluster and the monolithic emqttd cluster, with 2000 clients connected. Under normal conditions, without any broker node failure, the Mosquitto cluster delivers messages with comparable average latency. This suggests that the delay associated with microservice layers has a low impact on

performance in both latency and throughput. The scalability in terms of throughput degradation is also comparable for both implementations. Table 5.2 gives a summary of the performance comparison in latency and throughput

Table 5.2 Summary of latency and throughput results

	Throughput (msg/sec)			End-to-end latency (ms)		
	50	2000	% difference	Average	Worst-case	Std
Microservice broker cluster	265.68	233.28	8.47	3.42	42	3.42
Emqttd broker cluster	275.73	246.32	7.36	2.78	20.8	2.78

5.8.4 Inter-message Jitter

The jitter value measured here is the variation between response times. Jitter is a function of queuing, network buffering, processing, and competing for traffic for the network link. Good connections will have reliable and consistent response time, which is represented by a lower jitter value. High jitter means inconsistent response time, which will result in degraded quality of experience on the end-user for data streaming applications. The lower processing speed of the broker will tend to produce higher jitter because the delay in processing speed will cause jitter to increase. The experiment measures jitter values for periodic messages sent between an interval of 10 milliseconds.

From the evaluations in Section 5.5, the jitter values remain stable under increasing load. The sender sends messages at a constant rate of 100 messages per second, but the messages reach the receiver at a variable rate. If every message takes the same amount of time to travel from publisher to subscriber,

there is no jitter. While latency can be improved with optimized software processing, the jitter value is more random due to the asynchronous nature of the distributed system. The maximum jitter 35 milliseconds for the experiment. The maximum jitter value in the tests is the consequence of message forwarding delay between cluster nodes. Over 90% of jitter values are less than 15 milliseconds. This confirms that the jitter values remain consistent for the load testing. The receiver will not have significant perceivable abnormalities on the received messages, as large jitter will affect the quality of user experience.

5.8.5 Impact of Message Publication Rate

It is also noticed that when the fixed publication rate is increased to 1,000 message/sec, which means that each message is sent asynchronously between the interval of 1 millisecond. Transmission loss is observed for the broker cluster when more than 200 clients are connected. It is observed that some of the publications are not received by the cluster server. This can be related to message loss within the TCP and pcap capture buffers in the Linux network stack. The packet capture software (*tcpdump*) which is used in the pcap API captures and filters raw packets through the ethernet. Delays are incurred for the process of parsing and filtering of MQTT packets in the user application. Incoming packets are first queued in a buffer before processing. When the rate of packet pushed into the buffer exceeds the rate of packets consumed by *tcpdump* (Tcpdump&Libpcap, 2019), the buffer becomes full, which forces the kernel to drop all further packets until there is enough space available in the buffer.

This effect can be reduced by increasing the size of the buffer in the pcap stack and implementing a ring buffer for the consumer thread as suggested in (Albin and Rowe, 2012). Also, it could be possible that the Linux kernel has missed some of the network I/O interrupt due to high message and interrupt rate (Wu et al., 2007). The received packet is transferred into the main memory and the I/O receive interrupt is raised only when the packet is accessible in the ring buffer. The packet receiving process of network interface card (NIC) and network device driver interrupt is discussed in detail in (Wu and Crawford, 2007).

5.8.6 Failure recovery and comparison to primary/backup broker

To ensure the continuous availability of the MQTT service, the system has to preserve its message delivery operation in the event of unusual circumstances and server failures. This is achieved by using redundancy, and backup brokers, to resist server fault, either broker crashes or network partition of one server from other servers. The MQTT notification service is a critical component of the MQTT communication protocol and must not be a single point of failure. Subscribers will not receive messages if the entire system collapses.

Resiliency is achieved in the broker cluster by the means of message retransmissions. The broker cluster can provide continuous service even after the failure of brokers. The evaluations in Section 5.6 show that the broker cluster can tolerate and quickly recover from server crashes. Any one of the broker nodes will take over the subscribers when the subscribers reconnect.

Subscribers that are connected to a failed server can receive previously missed messages when they reconnect. The MQTT service runs asynchronously by all redundant MQTT brokers to ensure normal operation under failures.

The connection between the MQTT subscriber and the broker is maintained through an end-to-end TCP connection through a session. When a broker fails, the packets are routed to an offline IP node by the load balancer, and the end-to-end TCP connection with the client breaks. The broker cluster has to retransmit the MQTT messages received to the disconnected subscribers if the topic of the message matches.

The evaluations in Figure 5.36 and Figure 5.39 also reveal the overhead incurred during the failed test as the system uses more memory and CPU time to store missed messages and to perform the message retransmission process.

The broker stores each received publication as soon as the connections of previously disconnected subscribers come back online. When the subscriber reconnects, it receives a burst of messages that it missed when it was disconnected. With two out of four failed brokers, the retransmission process takes a maximum of 256.33 milliseconds to recover missed MQTT messages. This delay includes a 200 milliseconds live-check time client to broker, the reconnection time, the broker processing delay, and the message forwarding delay. The minimal jitter values between the retransmitted messages indicate that the remaining broker nodes can quickly retransmit the failed messages as soon as the client reconnects.

However, FRAME is able to achieve 50 milliseconds of worst-case recovery latency for its edge-based broker (C. Wang et al., 2019). Compared to this research work, FRAME uses a two-node primary-backup broker with TAO publish-subscribe interface. This research work implements a four-node cluster with a fully connected mesh topology based and uses a backup message queue to store the failed messages in each cluster node. Both approaches locally deploy the edge-based publish-subscribe broker system.

Since the proposed method needs to maintain more cluster nodes, their corresponding routing state is bigger and this leads to a higher recovery delay. Also, the MQTT protocol includes a keep-alive interval that is used by the client to detect broker failure. High failure detection timeout increases the time for broker nodes to recover messages, as confirmed by (Kazemzadeh and Jacobsen, 2009). In the experiment, the keep-alive interval used is 200 milliseconds. As a result, the client needs to wait for at most 200 milliseconds to detect a broker failure and reconnects a different broker node, which yields a latency of more than 200 milliseconds.

The broker cluster will perform additional client ID and message ID filtering, to prevent forwarding duplicate messages within the cluster, as described in Section 4.6.1. As a result, the MQTT broker cluster will not produce duplicate messages during normal operation and recovery process. However, depending on each MQTT broker, QoS 1 does allow for message duplicates, as described in the specification of MQTT protocol v3.1.1 (OASIS,

2014). A message with a duplicate may also be sent from the broker after the delivery of the first message instance.

5.8.7 Resource usage

Compared to the CPU time percentage of the single broker setup, the broker cluster setup utilizes more CPU time for the implementation of an additional cluster component. The results in Section 5.7 show that the system resources are not optimally used by the single node Mosquitto broker for both CPU and memory. The CPU utilization percentage is irregular throughout the whole test run. This is due to the high frequency of network I/O operations, which starved the running processes.

Since the Mosquitto broker is I/O bounded and makes use of only one processor core, the progress of MQTT message transmission is limited by the speed of network I/O and communication delays. The broker cluster implementation partly addresses this problem by making use of multiple processors in a distributed system. The MQTT broker can be faster and more efficient if the I/O subsystem is made faster. This approach is presented in (Pipatsakulroj et al., 2017), where the authors make use of the parallelization of both the MQTT broker components and TCP threads to minimize the effect of I/O bound. The proposed broker scheme is able to outperform the Mosquitto broker by 5.38% in terms of message throughput.

Unlike the CPU usage, the RAM usage for the clustered broker remains stable below 240 MiB despite the increase in workload. The evaluations

demonstrate that an increasing number of clients produce a negligible amount of RAM increases. RAM usage of the single MQTT broker increases by 21 MiB when the system is loaded with clients. The observations suggest that replication of the MQTT broker helps reduce the memory footprint for MQTT message processing.

CPU usage grows fast when handling client subscriptions, while memory only grows steadily and remains stable afterward. CPU and memory utilization of the broker cluster is relatively low, with a maximum of 7.8% CPU utilization and RAM usage below 355 MiB. Hence, the implementation of the Docker-based MQTT cluster can be considered lightweight.

The results in Section 5.7.2 suggest that the implementation of the Mosquitto broker cluster is sufficient to support as many as two thousand pairs of MQTT transmission despite the constraints imposed by the hardware of the single-board computers.

CHAPTER 6

DISCUSSION AND CONCLUSION

6.1 Introduction

This dissertation work presents implementations and evaluations of an MQTT broker cluster in edge setup and an IoT edge-cloud integration module. The motivation for the implementation of the proposed broker cluster is to improve local fault tolerance. The reason for this is because it is also possible to lose messages to the clients or the cloud due to server failure. Thus, local fault tolerance needs to be implemented to preserve the system locally at the edge of network. Many previous studies have focused on distributed publish-subscribe systems but few of them addressed the issue of local fault tolerance and the MQTT standard. Due to the recent popularity of the MQTT protocol, the MQTT middleware layer is developed to facilitate the cooperation of MQTT brokers without modifying the MQTT broker software. Thus, different broker implementations can fit together into the local cluster as long as they implement the MQTT protocol standard. Also, the use of a single-board computer as an edge-based hosting infrastructure keeps the cost low and can be flexibly sized according to workload demand and location of deployment. This provides the means to set up a local MQTT broker cluster in an edge-based environment such as in rural and remote areas where internet connectivity is limited. Besides, cloud-based messaging brokers have high and unreliable latency. The purpose of the edge provisioning of the broker cluster is to reduce end-to-end latency for IoT and M2M streaming applications. The cluster configuration also helps to

horizontally scale to MQTT broker to deal with increasing workload, and to prevent overloading of a single MQTT broker.

6.2 Methodology Used

This research work implements a microservice-based MQTT broker cluster where the cluster server is implemented as a separate software application. The cluster server communicates with its local broker via inter-process communication and its neighbor cluster server through a gossip-based membership protocol. The cluster server uses a topic-trie as a routing table to store neighbor subscription advertised by each neighbor node upon receiving a subscription locally. The load balancer that is used to distribute MQTT clients to the backend servers is assumed to be reliable. Each cluster node can fail independently without affecting other nodes. Each broker node immediately updates and deletes the corresponding entry in its routing tables when a failed broker node is detected. A buffered queue is used to store the failed message so that the message can be retrieved and sent to the affected clients when they reconnect to the broker cluster. To prevent message loss for recovering brokers that have previously failed, the joining broker node enters a recovery/synchronization state and temporarily stores all publication messages into an initialization message queue. After the recovery state, the broker node redelivers the message from the message queue to all corresponding neighbor nodes and clients.

The cluster is built using five Raspberry Pi boards that are connected to a network switch. In the edge layer, four boards are used for MQTT services, and one board is used as the Swarm manager node for Swarm orchestration and TCP load balancing. One Mosquitto MQTT broker container and one cluster server container are deployed into each Docker host.

6.3 Summary of Results

The evaluations demonstrated improved scalability for the broker cluster and successful recovery of failed publication during failover. The evaluations in Section 5.2 show that horizontal scaling does not always improve the overall system performance. Nevertheless, the broker cluster has other benefits to compensate for the lack of performance. The broker cluster can handle as many as 2000 connections without any major performance issues. Also, the method of service provisioning does not always impact the scalability of the system. The evaluations in Section 5.3 show that the worst-case end-to-end latency is at a maximum of 42 milliseconds. Although the broker cluster has increased latency values compared to the single broker, it is still adequate for many IoT applications, including building automation, smart grids, and smart farms. This is a tradeoff for reliability as there is an extra message forwarding delay in the clustered setup when the degree of data locality is low (Banno et al., 2017). Section 5.4 presents the performance overhead of the microservice-based Mosquitto cluster over the monolithic emqttd cluster. The proposed broker cluster delivers messages with higher average latency (3.42ms) compared to the monolithic-based emqttd cluster (2.78ms). The evaluations in Section 5.6 show that all missed publications are redelivered to the subscriber during failover

without significant delay between the retransmitted messages. This is because the jitter values after the recovery period are small. With two out of four failed brokers, the retransmission process takes a maximum of 256.33 milliseconds to recover missed MQTT messages. The overall system is efficient and overhead is minimal with Docker container. The observations from load testing show that the software implementation is lightweight with a CPU usage overhead of less than 5% and RAM usage overhead of about 8 MiB.

6.4 Future Work

For future work, the broker cluster can be improved by adding a Byzantine fault tolerance framework, with consideration to secure transactions. Trinity is a decentralized publish-subscribe broker that integrates the MQTT broker with Tendermint blockchain (Ramachandran et al., 2019). The Trinity brokers verify all published data, by executing smart contracts and consensus algorithm, before distributing the verified data to the other brokers in the blockchain network. However, the delay takes 1.5 to 4 seconds to deliver an MQTT message due to the duration of the consensus and transaction validation process within the Tendermint blockchain. Blockchain-based publish-subscribe communication can be useful for secure transactions and to provide assurance to a multiple stakeholder environment. This work can also be extended with the integration of a real-time latency deadline policy as suggested in (C. Wang et al., 2019). For the MQTT protocol, this requires the MQTT client to add a deadline tag on the message payload to indicate the minimum latency requirement for each message.

Taking robustness into account, the latency performance of the broker cluster is acceptable. The fault tolerance test confirms the reliability of the MQTT cluster, as failed publications can be redelivered during broker failure. In conclusion, this research work demonstrated that it is feasible to utilize a broker cluster to maintain consistent latencies and support reliable MQTT services despite server failures.

REFERENCES

- A Light, R., 2017. Mosquitto: server and client implementation of the MQTT protocol. *The Journal of Open Source Software*, 2(13), p.265. Available at: <http://dx.doi.org/10.21105/joss.00265>.
- Adjih, C. et al., 2015. FIT IoT-LAB: A large scale open experimental IoT testbed. *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. December 2015 IEEE.
- Ahmed, A. and Ahmed, E., 2016. A survey on mobile edge computing. *2016 10th International Conference on Intelligent Systems and Control (ISCO)*. 2016 pp. 1–8.
- Alam, M. et al., 2018. Orchestration of Microservices for IoT Using Docker and Edge Computing. *IEEE Communications Magazine*, 56(9), pp.118–123.
- Albin, E. and Rowe, N.C., 2012. A Realistic Experimental Comparison of the Suricata and Snort Intrusion-Detection Systems. *2012 26th International Conference on Advanced Information Networking and Applications Workshops*. 2012 pp. 122–127.
- Avizienis, A., Laprie, J.-., Randell, B. and Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1), pp.11–33.
- Babou, C.S.M. et al., 2018. Home Edge Computing (HEC): Design of a New Edge Computing Technology for Achieving Ultra-Low Latency BT - Edge Computing – EDGE 2018. 2018 Springer International Publishing, Cham, pp. 3–17.
- Baldoni, R. et al., 2007. TERA: Topic-based Event Routing for Peer-to-peer Architectures. *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems. DEBS '07*. 2007 ACM, New York, NY, USA, pp. 2–13.
- Baldoni, R., Querzoni, L., Tarkoma, S. and Virgillito, A., 2009. Distributed Event Routing in Publish-subscribe Systems BT - Middleware for Network Eccentric and Mobile Applications. In: Garbinato, B., Miranda, H. and Rodrigues, L., (eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 219–244.

- Banno, R. et al., 2015. Designing Overlay Networks for Handling Exhaust Data in a Distributed Topic-based Pub/Sub Architecture. *Journal of Information Processing*, 23(2), pp.105–116. Available at: <https://doi.org/10.2197/ipsjjip.23.105>.
- Banno, R. et al., 2017. Dissemination of edge-heavy data on heterogeneous MQTT brokers. *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. September 2017 IEEE.
- Bellavista, P. and Zanni, A., 2017. Feasibility of Fog Computing Deployment Based on Docker Containerization over RaspberryPi. *Proceedings of the 18th International Conference on Distributed Computing and Networking. ICDCN '17*. 2017 ACM, New York, NY, USA, pp. 16:1--16:10.
- Birman, K.P., 2012. *Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services (Texts in Computer Science)*, Springer.
- Bovet, D. and Cesati, M., 2005. Chapter 7 Process Scheduling. In: *Understanding the Linux Kernel: From I/O Ports to Process Management*. O'Reilly Media, pp. 258–290.
- Brewer, E.A., 2000. Towards Robust Distributed Systems (Abstract). *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. 2000 ACM, New York, NY, USA, pp. 7--.
- Burns, B. and Oppenheimer, D., 2016. Design Patterns for Container-based Distributed Systems. *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. June 2016 USENIX Association, Denver, CO.
- Campbell, R. et al., 2003. Towards Security and Privacy for Pervasive Computing BT - Software Security — Theories and Systems. 2003 Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–15.
- Carbone, P., Katsifodimos, A., et al., 2015. Apache FlinkTM: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin*, 38.
- Carbone, P., Fóra, G., et al., 2015. Lightweight Asynchronous Snapshots for Distributed Dataflows. *CoRR*, abs/1506.0. Available at: <http://arxiv.org/abs/1506.08603>.
- Carzaniga, A., Rosenblum, D.S. and Wolf, A.L., 2003. Design and evaluation of a wide-area event notification service. *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*. 2003 pp. 283–334.
- Castro, M., Druschel, P., Kermarrec, A.-M. and Rowstron, A.I.T., 2002. Scribe: a large-scale and decentralized application-level multicast infrastructure. *{IEEE} Journal on Selected Areas in Communications*, 20(8), pp.1489–1499. Available at: <https://doi.org/10.1109/jsac.2002.803069>.

- Chang, T. et al., 2014. P2S: A fault-tolerant publish-subscribe infrastructure. *DEBS 2014 - Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*.
- Cheng, B., Papageorgiou, A. and Bauer, M., 2016. Geelytics: Enabling On-Demand Edge Analytics over Scoped Data Sources. *2016 IEEE International Congress on Big Data (BigData Congress)*. June 2016 IEEE.
- Chiang, M. et al., 2017. Clarifying Fog Computing and Networking: 10 Questions and Answers. *IEEE Communications Magazine*, 55(4), pp.18–20.
- Church, M., 2019, *Docker Reference Architecture: Designing Scalable, Portable Docker Container Networks* [Online]. Available at: <https://success.docker.com/article/networking> [Accessed: 19 July 2019].
- Cicizz, 2019. Cicizz/jmqtt. *GitHub*. Available at: <https://github.com/Cicizz/jmqtt>.
- Das, A., Gupta, I. and Motivala, A., 2002. SWIM: scalable weakly-consistent infection-style process group membership protocol. *Proceedings International Conference on Dependable Systems and Networks*. 2002 pp. 303–312.
- Dastjerdi, A.V. and Buyya, R., 2016. Fog Computing: Helping the Internet of Things Realize Its Potential. *Computer*, 49(8), pp.112–116. Available at: <https://doi.org/10.1109/mc.2016.245>.
- Datta, A. et al., 2005. Range queries in trie-structured overlays. *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*. 2005 pp. 57–66.
- Deshpande, L. and Liu, K., 2017. Edge computing embedded platform with container migration. *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. 2017 pp. 1–6.
- Devi, Y.L. and Saikia, L.P., 2014. *Fault tolerance techniques and algorithms in cloud system*,
- Dogo, E.M., Salami, A.F., Aigbavboa, C.O. and Nkonyana, T., 2019. Taking Cloud Computing to the Extreme Edge: A Review of Mist Computing for Smart Cities and Industry 4.0 in Africa BT - Edge Computing: From Hype to Reality. In: Al-Turjman, F., (ed.) Springer International Publishing, Cham, pp. 107–132.
- Egwutuoha, I.P., Levy, D., Selic, B. and Chen, S., 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3), pp.1302–1326. Available at: <https://doi.org/10.1007/s11227-013-0884-0>.
- Emqtt.io, 2018, *EMQ* [Online]. Available at: <http://emqtt.io/> [Accessed: 18 June 2019].

- Eugster, P.T., Felber, P.A., Guerraoui, R. and Kermarrec, A.-M., 2003. The Many Faces of Publish-subscribe. *ACM Comput. Surv.*, 35(2), pp.114–131. Available at: <http://doi.acm.org/10.1145/857076.857078>.
- Eugster, P.T. and Guerraoui, R., 2002. Probabilistic multicast. *Proceedings International Conference on Dependable Systems and Networks*. 2002 pp. 313–322.
- Felter, W., Ferreira, A., Rajamony, R. and Rubio, J., 2015. An updated performance comparison of virtual machines and Linux containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015 pp. 171–172.
- Gascon-Samson, J., Garcia, F., Kemme, B. and Kienzle, J., 2015. Dynamoth: A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud. *2015 IEEE 35th International Conference on Distributed Computing Systems*. 2015 pp. 486–496.
- Gia, T.N. et al., 2015. Fault tolerant and scalable IoT-based architecture for health monitoring. *2015 IEEE Sensors Applications Symposium (SAS)*. 2015 pp. 1–6.
- Gonzalez, A.J., Nencioni, G., Helvik, B.E. and Kamisinski, A., 2016. A Fault-Tolerant and Consistent SDN Controller. *2016 IEEE Global Communications Conference (GLOBECOM)*. 2016 pp. 1–6.
- Guermouche, A. et al., 2011. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications. *2011 IEEE International Parallel & Distributed Processing Symposium*. 2011 pp. 989–1000.
- Gusev, M., 2017. A dew computing solution for IoT streaming devices. *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2017 pp. 387–392.
- Happ, D. et al., 2017. Meeting IoT platform requirements with open pub/sub solutions. *Annals of Telecommunications*, 72(1–2), pp.41–52. Available at: <https://doi.org/10.1007/s12243-016-0537-4>.
- Happ, D. and Wolisz, A., 2016. Limitations of the Pub/Sub pattern for cloud based IoT and their implications. *2016 Cloudification of the Internet of Things (CIoT)*. November 2016 IEEE.
- Hivemq, 2019, *HiveMQ | Reliable Data Movement for Connected Devices* [Online]. Available at: <https://www.hivemq.com/>.
- hui6075, 2018. hui6075/mqtt-bm-latency. *GitHub*. Available at: <https://github.com/hui6075/mqtt-bm-latency>.
- Ismail, B.I. et al., 2015. Evaluation of Docker as Edge computing platform. *2015 IEEE Conference on Open Systems (ICOS)*. 2015 pp. 130–135.

- Javed, A., Heljanko, K., Buda, A. and Främling, K., 2018. CEFIoT: A fault-tolerant IoT architecture for edge and cloud. *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. 2018 pp. 813–818.
- Jerzak, Z. and Fetzer, C., 2009. Soft State in Publish-subscribe. *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. DEBS '09. 2009 ACM, New York, NY, USA, pp. 17:1--17:12.
- Kakakhel, S.R.U., Mukkala, L., Westerlund, T. and Plosila, J., 2018. Virtualization at the network edge: A technology perspective. *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*. 2018 pp. 87–92.
- Karthikeya, S.A., Vijeth, J.K. and Murthy, C.S.R., 2016. Leveraging Solution-Specific Gateways for cost-effective and fault-tolerant IoT networking. *2016 IEEE Wireless Communications and Networking Conference*. 2016 pp. 1–6.
- Kazemzadeh, R.S. and Jacobsen, H., 2009. Reliable and Highly Available Distributed Publish-subscribe Service. *2009 28th IEEE International Symposium on Reliable Distributed Systems*. 2009 pp. 41–50.
- Khunteta, A. and Praveen, K., 2010. *An Analysis of Checkpointing Algorithms for Distributed Mobile Systems*,
- Kreps, J., Narkhede, N. and Rao, J., 2011. Kafka : a Distributed Messaging System for Log Processing. 2011
- Laaroussi, Z., Morabito, R. and Taleb, T., 2018. Service Provisioning in Vehicular Networks Through Edge and Cloud: An Empirical Analysis. *2018 IEEE Conference on Standards for Communications and Networking (CSCN)*. 2018 pp. 1–6.
- Lee, K., Kim, Y. and Yoo, C., 2018. The Impact of Container Virtualization on Network Performance of IoT Devices. *Mobile Information Systems*, 2018, pp.1–6. Available at: <https://doi.org/10.1155/2018/9570506>.
- Lertsinsruttavee, A. et al., 2017. PiCasso: A lightweight edge computing platform. *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. September 2017 IEEE.
- Li, M. et al., 2011. A Scalable and Elastic Publish-subscribe Service. *2011 IEEE International Parallel & Distributed Processing Symposium*. 2011 pp. 1254–1265.
- Liyanage, M., Chang, C. and Srirama, S.N., 2016. mePaaS: Mobile-Embedded Platform as a Service for Distributing Fog Computing to Edge Nodes. *2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. 2016 pp. 73–80.
- Lopez, P.G. et al., 2015. Edge-centric Computing. *ACM SIGCOMM Computer Communication Review*, 45(5), pp.37–42. Available at: <https://doi.org/10.1145/2831347.2831354>.

- Luzuriaga, J.E. et al., 2014. Testing AMQP Protocol on Unstable and Mobile Networks BT - Internet and Distributed Computing Systems. 2014 Springer International Publishing, Cham, pp. 250–260.
- Mattsson, H., Nilsson, H. and Wikström, C., 1998. Mnesia — A Distributed Robust DBMS for Telecommunications Applications BT - Practical Aspects of Declarative Languages. 1998 Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 152–163.
- Merkel, D., 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239). Available at: <http://dl.acm.org/citation.cfm?id=2600239.2600241>.
- Milo, T., Zur, T. and Verbin, E., 2007. Boosting Topic-based Publish-subscribe Systems with Dynamic Clustering. *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. 2007 ACM, New York, NY, USA, pp. 749–760.
- Mishra, H., 2019. VerneMQ - Clustering MQTT for high availability and scalability. *IoTbyHVM*. Available at: <https://iotbyhvm.ooo/vernemq/>.
- Morabito, R., 2017. Inspecting the performance of low-power nodes during the execution of edge computing tasks. *2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. 2017 pp. 148–153.
- Morabito, Roberto, 2017. Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation. *{IEEE} Access*, 5, pp.8835–8850. Available at: <https://doi.org/10.1109/access.2017.2704444>.
- Morabito, R. and Beijar, N., 2016. Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies. *2016 IEEE International Conference on Sensing, Communication and Networking (SECON Workshops)*. 2016 pp. 1–6.
- Morabito, R., Petrolo, R., Loscrì, V. and Mitton, N., 2018. LEGIoT: A Lightweight Edge Gateway for the Internet of Things. *Future Generation Computer Systems*, 81, pp.1–15. Available at: <https://doi.org/10.1016/j.future.2017.10.011>.
- Morabito, R., Petrolo, R., Loscri, V. and Mitton, N., 2016. Enabling a lightweight Edge Gateway-as-a-Service for the Internet of Things. *2016 7th International Conference on the Network of the Future (NOF)*. November 2016 IEEE.
- Naha, R.K. et al., 2018. Fog Computing: Survey of Trends, Architectures, Requirements, and Research Directions. *IEEE Access*, 6, pp.47980–48009. Available at: <https://doi.org/10.1109/access.2018.2866491>.
- Nikaein, N. and Krea, S., 2011. Latency for Real-Time Machine-to-Machine Communication in LTE-Based System Architecture. *17th European Wireless 2011 - Sustainable Wireless Technologies*. 2011 pp. 1–6.

- Novo, O. et al., 2015. Capillary networks - bridging the cellular and IoT worlds. *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. 2015 pp. 571–578.
- OASIS, 2014, *MQTT Version v3.1.1* [Online]. Available at: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html> [Accessed: 9 September 2019].
- Oki, B., Pfluegl, M., Siegel, A. and Skeen, D., 1993. The Information Bus: An Architecture for Extensible Distributed Systems. *SIGOPS Oper. Syst. Rev.*, 27(5), pp.58–68. Available at: <http://doi.acm.org/10.1145/173668.168624>.
- Ozeer, U. et al., 2018. *Resilience of Stateful IoT Applications in a Dynamic Fog Environment*,
- Pahl, C. et al., 2016. A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters. *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*. 2016 pp. 117–124.
- Pahl, C., 2015. Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3), pp.24–31. Available at: <https://doi.org/10.1109/mcc.2015.51>.
- Pahl, C. and Lee, B., 2015. Containers and Clusters for Edge Cloud Architectures -- A Technology Review. *2015 3rd International Conference on Future Internet of Things and Cloud*. August 2015 IEEE.
- Pang, Z. et al., 2015. A Survey of Cloudlet Based Mobile Computing. *2015 International Conference on Cloud Computing and Big Data (CCBD)*. 2015 pp. 268–275.
- Park, J.-H., Kim, H.-S. and Kim, W.-T., 2018. DM-MQTT: An Efficient MQTT Based on SDN Multicast for Massive IoT Communications. *Sensors*, 18(9).
- Pereira, C. and Aguiar, A., 2014. Towards Efficient Mobile M2M Communications: Survey and Open Challenges. *Sensors*, 14(10), pp.19582–19608. Available at: <https://www.mdpi.com/1424-8220/14/10/19582>.
- Pietzuch, P.R. and Bacon, J.M., 2002. Hermes: a distributed event-based middleware architecture. *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*. 2002 pp. 611–618.
- Pipatsakulroj, W., Visoottiviseth, V. and Takano, R., 2017. muMQ: A lightweight and scalable MQTT broker. *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. 2017 pp. 1–6.
- Popov, A., Proletarsky, A., Belov, S. and Sorokin, A., 2017. Fast Prototyping of the Internet of Things solutions with IBM Bluemix. *Proceedings of the 50th Hawaii International Conference on System Sciences (2017)*. 2017 Hawaii International Conference on System Sciences.
- Prenden, J.S. et al., 2015. The Benefits of Self-Awareness and Attention in Fog and Mist Computing. *Computer*, 48(7), pp.37–45.

- Rahimian, F., Girdzijauskas, S., Payberah, A.H. and Haridi, S., 2011. Vitis: A Gossip-based Hybrid Overlay for Internet-scale Publish-subscribe Enabling Rendezvous Routing in Unstructured Overlay Networks. *2011 {IEEE} International Parallel & Distributed Processing Symposium*. May 2011 IEEE.
- Ramachandran, G.S. et al., 2019. Trinity: A Byzantine Fault-Tolerant Distributed Publish-Subscribe System with Immutable Blockchain-based Persistence. *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2019 pp. 227–235.
- Rao, A. et al., 2003. Load Balancing in Structured P2P Systems BT - Peer-to-Peer Systems II. 2003 Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 68–79.
- Rausch, T., Nastic, S. and Dustdar, S., 2018. EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications. *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 2018 pp. 191–197.
- Redondi, A.E.C., Arcia-Moret, A. and Manzoni, P., 2019. Towards a Scaled IoT Pub/Sub Architecture for 5G Networks: the Case of Multiaccess Edge Computing. *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*. 2019 pp. 436–441.
- Robert, B., 2014. Towards the trillion sensors market. *Sensor Review*, 34(2), pp.137–142. Available at: <https://doi.org/10.1108/SR-12-2013-755>.
- Rooney, S., Bauer, D. and Scotton, P., 2005. Edge server software architecture for sensor applications. *The 2005 Symposium on Applications and the Internet*. 2005 pp. 64–71.
- Rotaru, M., Olariu, F., Onica, E. and Rivière, E., 2017. Reliable Messaging to Millions of Users with Migratorydata. *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*. Middleware '17. 2017 ACM, New York, NY, USA, pp. 1–7.
- Salehi, P., Doblender, C. and Jacobsen, H.-A., 2016. Highly-available Content-based Publish-subscribe via Gossiping. *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. 2016 ACM, New York, NY, USA, pp. 93–104.
- Salehi, P., Zhang, K. and Jacobsen, H.-A., 2017. PopSub: Improving Resource Utilization in Distributed Content-based Publish-subscribe Systems. *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS '17. 2017 ACM, New York, NY, USA, pp. 88–99.
- Saraswat, P.K., Pop, P. and Madsen, J., 2010. Task Mapping and Bandwidth Reservation for Mixed Hard/Soft Fault-Tolerant Embedded Systems. *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. 2010 pp. 89–98.

- Satria, D., Park, D. and Jo, M., 2017. Recovery for overloaded mobile edge computing. *Future Generation Computer Systems*, 70, pp.138–147. Available at: <http://www.sciencedirect.com/science/article/pii/S0167739X16302096>.
- Scalagent, 2014, *JoramMQ, a distributed MQTT broker for the Internet of Things. White paper and performance evaluation* [Online]. Available at: http://www.scalagent.com/IMG/pdf/JoramMQ_MQTT_white_paper-v1-2.pdf.
- Schmitt, A., Carlier, F. and Renault, V., 2018. Dynamic bridge generation for IoT data exchange via the MQTT protocol. *Procedia Computer Science*, 130, pp.90–97. Available at: <http://www.sciencedirect.com/science/article/pii/S1877050918303661>.
- Sen, S. and Balasubramanian, A., 2018. A highly resilient and scalable broker architecture for IoT applications. *2018 10th International Conference on Communication Systems & Networks (COMSNETS)*. 2018 pp. 336–341.
- Setty, V., van Steen, M., Vitenberg, R. and Voulgaris, S., 2012. PolderCast: Fast, Robust, and Scalable Architecture for P2P Topic-based Pub/Sub. *Proceedings of the 13th International Middleware Conference*. Middleware '12. 2012 Springer-Verlag New York, Inc., New York, NY, USA, pp. 271–291.
- Sherafat Kazemzadeh, R. and Jacobsen, H.-A., 2012. Opportunistic Multipath Forwarding in Content-Based Publish-subscribe Overlays BT - *Middleware 2012*. 2012 Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 249–270.
- Shi, W. and Dustdar, S., 2016. The Promise of Edge Computing. *Computer*, 49(5), pp.78–81. Available at: <https://doi.org/10.1109/mc.2016.145>.
- Siegemund, G., Turau, V. and Maâmra, K., 2015. A self-stabilizing publish-subscribe middleware for wireless sensor networks. *2015 International Conference and Workshops on Networked Systems (NetSys)*. 2015 pp. 1–8.
- Sourlas, V., Paschos, G.S., Flegkas, P. and Tassiulas, L., 2009. Caching in Content-Based Publish-subscribe Systems. *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*. 2009 pp. 1–6.
- Stoica, I. et al., 2003. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1), pp.17–32.
- Su, P.H. et al., 2014. Decentralized fault tolerance mechanism for intelligent IoT/M2M middleware. *2014 IEEE World Forum on Internet of Things (WF-IoT)*. 2014 pp. 45–50.
- Tammemäe, K. et al., 2018. Self-Aware Fog Computing in Private and Secure Spheres BT - *Fog Computing in the Internet of Things: Intelligence at the Edge*. In: Rahmani, A.M., Liljeberg, P., Preden, J.-S. and Jantsch, A., (eds.) Springer International Publishing, Cham, pp. 71–99.
- Tandon, R. and Simeone, O., 2016. Harnessing cloud and edge synergies: toward an information theory of fog radio access networks. *IEEE Communications Magazine*, 54(8), pp.44–50.

Tang, K. et al., 2013. Design and Implementation of Push Notification System Based on the MQTT Protocol BT - 2013 International Conference on Information Science and Computer Applications (ISCA 2013). October 2013 Atlantis Press.

Tarreau, W. and others, 2012. HAProxy-the reliable, high-performance TCP/HTTP load balancer. 2011-8)[2013-4]. <http://haproxy.lwt.eu>.

Tcpdump&Libpcap, 2019, *Manpage of TCPDUMP* [Online]. Available at: <https://www.tcpdump.org/manpages/tcpdump.1.html>.

To, M.A., Cano, M. and Biba, P., 2015. DOCKEMU -- A Network Emulation Tool. *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*. 2015 pp. 593–598.

Viswanath, S.K. et al., 2016. System design of the internet of things for residential smart grid. *IEEE Wireless Communications*, 23(5), pp.90–98.

Vohra, D., 2017. Using Docker in Swarm Mode BT - Docker Management Design Patterns: Swarm Mode on Amazon Web Services. In: Vohra, D., (ed.) Apress, Berkeley, CA, pp. 9–30.

Wang, C., Gill, C. and Lu, C., 2019. FRAME: Fault Tolerant and Real-Time Messaging for Edge Computing. *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2019 pp. 976–985.

Wang, S. et al., 2019. Lineage Stash: Fault Tolerance off the Critical Path. *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. 2019 ACM, New York, NY, USA, pp. 338–352.

Wu, W. and Crawford, M., 2007. Potential performance bottleneck in Linux TCP. *International Journal of Communication Systems*, 20(11), pp.1263–1283. Available at: <https://doi.org/10.1002/dac.872>.

Wu, W., Crawford, M. and Bowden, M., 2007. The performance analysis of linux networking – Packet receiving. *Computer Communications*, 30(5), pp.1044–1057. Available at: <http://www.sciencedirect.com/science/article/pii/S0140366406004221>.

Xu, Y., Mahendran, V. and Radhakrishnan, S., 2016. Towards SDN-based fog computing: MQTT broker virtualization for effective and reliable delivery. *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*. 2016 pp. 1–6.

Zaharia, M. et al., 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. 2013 ACM, New York, NY, USA, pp. 423–438.

Zamora-Izquierdo, M.A. et al., 2019. Smart farming IoT platform based on edge and cloud computing. *Biosystems Engineering*, 177, pp.4–17. Available at: <https://doi.org/10.1016/j.biosystemseng.2018.10.014>.

Zeng, H., Wang, B., Deng, W. and Zhang, W., 2017. Measurement and Evaluation for Docker Container Networking. *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. 2017 pp. 105–108.

Zhang, L., 2011, *Building Facebook Messenger* [Online]. Available at: <https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920/>.

Zhao, Y., Kim, K. and Venkatasubramanian, N., 2013. DYNATOPS: A Dynamic Topic-based Publish-subscribe Architecture. *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS '13. 2013 ACM, New York, NY, USA, pp. 75–86.

Zhenhui Shen and Srikanta Tirthapura, 2004. Self-stabilizing routing in publish-subscribe systems. *IET Conference Proceedings*, pp.92-97(5). Available at: https://digital-library.theiet.org/content/conferences/10.1049/ic_20040389.

Zhuang, S. et al., 2001. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. *Proceedings of the IEEE International Workshop on Network and Operating System Support for Digital Audio and Video*. 2001 pp. 11–20.

Zyrianoff, I., Heideker, A., Silva, D. and Kamienski, C., 2018. Scalability of an Internet of Things Platform for Smart Water Management for Agriculture. *Proceedings of the 23rd Conference of Open Innovations Association FRUCT*. FRUCT'23. 2018 FRUCT Oy, Helsinki, Finland, Finland, pp. 58:432--58:439.

APPENDIX A

LIST OF PUBLICATION

Thean, Z.Y., Yap, V.V. and Teh, P.C., 2019. Container-based MQTT Broker Cluster for Edge Computing. 2019 4th International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE). 2019 pp. 1–6.