Design and Implementation of Distributed Message Queue Systems with High Throughput and Availability

January 2017

Masafumi KINOSHITA

Design and Implementation of Distributed Message Queue Systems with High Throughput and Availability

Submitted to

Graduate School of Information Science and Technology

Osaka University

January 2017

Masafumi KINOSHITA

List of publications

A. Journal Papers

1. M. Kinoshita, G. Tsuchida, I. Mizutani and T. Koike, “High-throughput messaging system based on in-memory KVS for processing large traffic volume of short messages,” *IEICE Transactions on Communications*, vol. B96, no. 10, pp. 1206-1216, Oct. 2013 (in Japanese).

2. M. Kinoshita, O. Takada, I. Mizutani, T. Koike, K. Leibnitz, and M. Murata, “Improved resilience through extended KVS-based messaging system,” *IEICE Transactions on Information and Systems*, vol. E98-D, no. 3, pp. 578-587, Mar. 2015.

3. M. Kinoshita, H. Konoura, T. Koike, K. Leibnitz, and M. Murata, “High throughput dequeuing technique in distributed message queues for IoT,” *IPSJ Journal of Information Processing*, vol. 27, no. 2, to appear Feb. 2017.

B. Refereed Conference Papers

1. M. Kinoshita, G. Tsuchida, and T. Koike, “High-throughput message systems for mobile e-mail services based on in-memory KVS,” in *Proceedings of Wireless and Mobile Communications (ICWMC)*, pp. 146-153, June 2012.

2. H. Konoura, M. Kinoshita, T. Koike, K. Leibnitz, and M. Murata, “Efficient dequeuing technique for distributed messaging systems processing massive message volumes,” in *Proceedings of the 26th IEEE International Telecommunication Networks and Applications Conference (ITNAC)*, pp. 280-285,

Dec. 2016.

C. Non-Referred Conference Papers

1. M. Kinoshita, G. Tsuchida, and T. Koike, “Throughput improvement of message system in cooperation with distributed in-memory KVS,” in *Proceedings of IEICE Society Conference*, p. 412, Sep. 2011 (in Japanese).

2. I. Mizutani, K. Toumura, and M. Kinoshita, “Method of achieving N-to-N configuration for message system based on distributed KVS,” in *Proceedings of IEICE Society Conference*, p. 359, Sep. 2012 (in Japanese).

3. M. Kinoshita, O. Takada, I. Mizutani, T. Koike, K. Leibnitz, and M. Murata, “High availability method for extended KVS-based messaging system,” in *Proceedings of IEICE* (*IA2014-84*), vol. 114, no. 439, pp. 31-36, Jan. 2015 (in Japanese).

4. H. Konoura and M. Kinoshita, “Study of performance enhancing method on a message-oriented middleware,” in *Proceedings of IEICE Society Conference*, p. B-6-5, Sep. 2015 (in Japanese).

Preface

The innovation paradigm regarding smart phones, *Machine-to-Machine* (M2M) communication, or the *Internet of Things* (IoT) is currently causing an explosion in the number of devices connected to the network and thus requires changes to the service system. Smart phones have been gaining in popularity over the last seven years with about 2.5 billion connected devices in 2009. Then, M2M devices, such as smart meters and health equipment, accelerated this increase to 10 billion in 2014. IoT devices, such as sensors/actuators in factories, cars, home devices, etc., are expected to increase to 30 billion by 2020. In addition, the amount of digital data in the whole world created by connected devices is expected to reach 40 ZB by 2020.

In this thesis, we focus on these upcoming drastic changes of network systems providing services or applications to users, where we especially focus on message queue systems as frontend of these network systems. Furthermore, we discuss what features these message queue systems should provide for processing this unprecedented data volume created by IoT devices and how they should handle requirements on availability and scalability.

We begin this thesis with the discussion of high-throughput and scalable processing of huge volumes of messages in smart phone services. To solve this issue, we propose high-throughput queuing techniques and architectures for distributed message queue systems that can serve much larger message traffic than before. We designed a message queue system based on a distributed *in-memory key-value store* (KVS) to meet the requirements on throughput and scalability. We also propose an architecture for satisfying high throughput and high scalability in a message queue system for massive message traffic volumes through a distribution method of queue-type in-memory KVS and synchronized processing of distributed queues by single TCP connections. We embed the proposed architecture and strategies into a mail system for smart phones and perform evaluations of this system. The evaluation results reveal that the throughput of the proposed message queue system achieves 3,600

messages per second (msg/s) per server, which is about 5 times higher than that of the conventional method operating with RAID storages. Moreover, the throughput of the proposed KVS is 200,000 transactions per second for message size of 0.4 KB, which doubles the performance of the well-known KVS called *memcached*.

Our next concern is the resilience of the message queue system for M2M services. M2M services, such as metering and monitoring services, have enhanced the social infrastructure field. As social infrastructure, the service system, especially in our case the message queue system, is required to simultaneously satisfy both, high availability and high throughput. To solve this issue, we propose a resilient message queue system based on a distributed KVS. Its servers are interconnected among each other and messages are distributed to multiple servers in the normal processing state. Our proposed system can provide long-term availability and continue its service regardless where failures in the message queue server/process may occur by distributing messages to multiple servers. Furthermore, to achieve short-term availability, even during an underlying network failure and/or slowdown of servers, we propose message distribution by round-robin with slowdown KVS exclusion and two logical KVS counter-rotating rings. Evaluation results show that this system can continue service without the need for failover processing. Compared with the conventional method, our proposed distribution methods reduce 92% of service errors caused by server failures.

Finally, we discuss a method for increasing the dequeue throughput in message queue systems for the IoT era. IoT services require information extracted from historical or real-time data for specific objectives, such as optimization services or learning through trial-and-error pattern analysis of data. This approach requires collecting large volumes of messages that are periodically created by the devices. On other hand, the backend system retrieves messages from the message queue at its own non-periodic and process-dependent timings. Therefore, controlling the massive and heterogeneous traffic in the message system becomes a crucial issue. To solve this issue, we propose a dequeuing method called *Retry Dequeue-request Scheduling* (RDS), which can reduce unnecessary transmissions of dequeue requests to the message queues by waiting for messages to arrive at the message queues. RDS can better reduce throughput degradation than the conventional method by making use of *missed-dequeue* messages. By evaluation through simulations, we compare the throughputs achieved by the conventional method, RDS, and *Periodical Monitoring and Scheduling* (PMS), which is another dequeuing method proposed for reducing the number of *missed-dequeue*s by periodically monitoring each message queue to gather information on the message counters. Simulation evaluation results show that RDS maintains the highest throughput, regardless of an increased dequeue request rate. Further experimental evaluation results show that the RDS method achieves 80% higher throughput than the conventional method in real systems.

Through the following discussions, we conclude that high-throughput queuing techniques and a resilient message queue system are fundamental technologies to stably process large-volume messages created by IoT devices. Additionally, the increased throughput of the RDS method is essential in finding patterns within large data volumes from IoT services. These proposed technologies can make it much easier and faster than before to build complex IoT systems requiring high throughput, availability, and scalability. We believe that the following discussions will contribute to the better design and implementation of future IoT systems.

Acknowledgments

This thesis could not have been accomplished without the assistance of many people, and I would like to acknowledge all of them.

First and foremost, I would like to express my sincere appreciation to Professor Masayuki Murata of the Graduate School of Information Science and Technology, Osaka University, for his patient encouragement, insightful and comprehensive advice, and valuable discussions. He directed me to the appropriate perspective in this domain and inspired me to aim at higher goals.

I am also deeply grateful to the members of my PhD evaluation committee, Professor Teruo Higashino, Professor Toru Hasegawa, and Professor Takashi Watanabe of the Graduate School of Information Science and Technology, Osaka University, and Professor Morito Matsuoka of the Cybermedia Center, Osaka University, for their critical reviews and comments from various angles.

Furthermore, I am deeply grateful to Guest Associate Professor Kenji Leibnitz of Osaka University for his much-appreciated comments and support. His passionate and unerring guidance have been informative and helpful. His kindness on my behalf was invaluable, and I am forever in debt.

I am also grateful to Dr. Kenichi Sakamoto, Head of the Planning Office at the Yokohama Research Laboratory of Hitachi Ltd., Dr. Seishi Hanaoka, Department Manager of Hitachi Ltd., Center for Technology Innovation – Information and Telecommunication, Network Research Department, and Mr. Tatsuhiko Miyata, Unit Leader at the Center for Technology Innovation – Information and Telecommunication, Network Research Department, for giving me the opportunity to study for a doctorate at Osaka University.

My appreciation also goes to my colleagues Dr. Osamu Takada, Dr. Yasunori Kaneda, Dr. Ken Naono, Ms. Yukiko Takeda, Dr. Hiroaki Konoura, Ms. Izumi Mizutani, Dr. Yu Nakata, Dr. Kunihiko Toumura, Mr. Yoshiki Matsuura, Dr. Gen Tsuchida, Mr. Toshiyuki Kamiya, and Mr. Takafumi Koike of Hitachi Ltd. for their valuable discussions, advice, support, and encouragement during this study.

Additionally, I am thankful to all the members of the Network Research Department, Hitachi Ltd., Center for Technology Innovation – Information and Telecommunication, for their continuous support and friendship.

Finally, I deeply thank my wife, Maki, my daughter Hana, my son Sou, my parents, and my grandparents for their understanding and hearty support and encouragement in my daily life. This work would not have been achieved without them.

Contents

Chapter 1

Introduction

1.1. Background

Innovative network paradigms, such as mobile smart phones, *Machine-to-Machine* (M2M) communication, and the *Internet of Things* (IoT) have been gaining popularity with a billion devices already connected to the Internet today. Figure 1.1 outlines this growth in the number of connected devices over three phases of time based on [1]. The number of traditionally connected devices, like PCs, accounted for only 500 million in 2003. The number of connected smart phones was about 2.5 billion in 2009, and they have drastically gained in popularity since then. M2M devices, such as smart meters [2], health equipment [3], and vending machines [4] increased the number of connected devices even further to approximately 10 billion in 2014. In the future, IoT devices such as sensors/actuators in factories [5], used for transportation [6], or home devices [7] are expected to increase to about 30 billion by 2020.

In this thesis, we focus on traffic data of network systems providing services or applications, especially short-length data created by various devices, which we will refer to as *short messages* in the following. Reference [8] states that the amount of all digital data in the world created by various devices is predicted to reach 40 ZB by 2020 (Fig. 1.2).

To increase the future processing ability for short messages, we focus in this work on end-to-end communication. End-to-end protocols, such as the Hypertext Transfer Protocol (HTTP) [9], have been widely used for client-server communication in the Internet [10-12]. However, such protocols could also face problems if the number

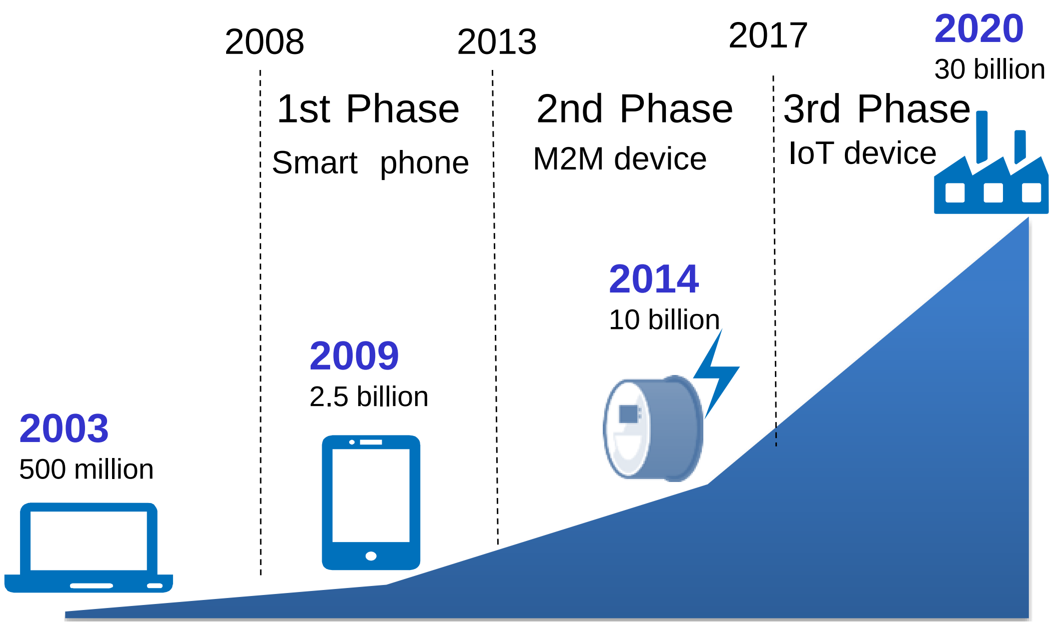


Figure 1.1 The growth in the number of connected devices

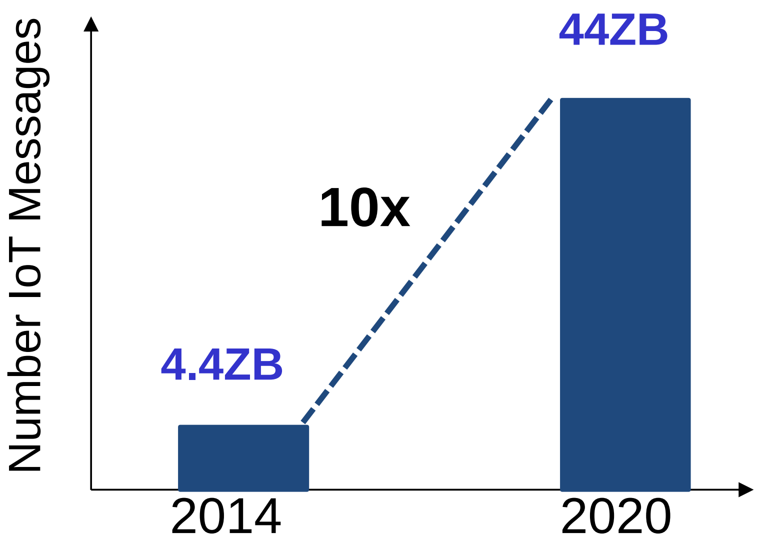


Figure 1.2 The growth in the volume of IoT messages

of end-to-end clients drastically increases due to server congestion by huge traffic volumes, inefficiency of one-to-many communication, or heavy loads for maintaining massive connections in server and network [13-15]. To solve these problems, messaging communication protocols, such as *MQ Telemetry Transport* (MQTT) [16] or *Advanced Message Queuing Protocol* (AMQP) [17] are expected to better handle the large volumes of short messages in M2M/IoT [18-21]. Messaging communication requires the network system to queue and relay messages between client and server and we will refer to such systems as *message queue systems* in the following. On the other hand, conventional message queue systems have often led to concerns regarding throughput and scalability [22-24]. Figure 1.3 outlines the drastic change of throughput requirements in a message queue system. Conventional message queue systems have been used for various services, such as e-mail messages on PCs or electronic data interchange between companies. These services don't have large numbers of devices or large message traffic volume. For example, the throughput of a single server for *sendmail* [25], which is widely used for e-mail message services in companies, is below 100 msg./s [26]. However, in the smart phone/M2M/IoT era, we estimate that a message queue system is required to process more than 10,000 msg./s of short messages due to the increase in number of devices and message traffic volume in service systems. For example, the number of smart phones of a carrier system increases approximately from 10 to 100 million [27, 28]. The number of smart meters of an electricity company increases approximately from 10 to 50 million [29, 30]. Furthermore, we estimate that the number of devices, such as sensors, actuators, and *radio frequency identifier* (RFID) for tracking products in a smart factory increases from 1,000 to 10,000 and all these messages are collected in real-time [31]. Therefore, message queue systems will be required to apply new methods and architectures to process these huge volumes of short messages. One main goal of this thesis is the discussion of the performance of message queue systems processing huge volumes of short messages.

From the viewpoint of message queue systems, we consider that these upcoming drastic changes in messaging traffic have progressed through roughly three phases as illustrated in Fig. 1.1. The initial phase consists of the increase of short messages used for e.g., mobile email services, *short message services* (SMS) [32], or *social networking services* (SNS) [33] between 2008 and 2013, which coincides with the spread of smart phones. The most important issue during this phase is the high-throughput and scalable processing of huge volumes of short messages in smart phone services. The second phase extends short messages to other M2M applications in social infrastructure fields beyond smart phones, such as smart meters and health equipment from 2013 to 2017. In this phase, high availability and resilience for

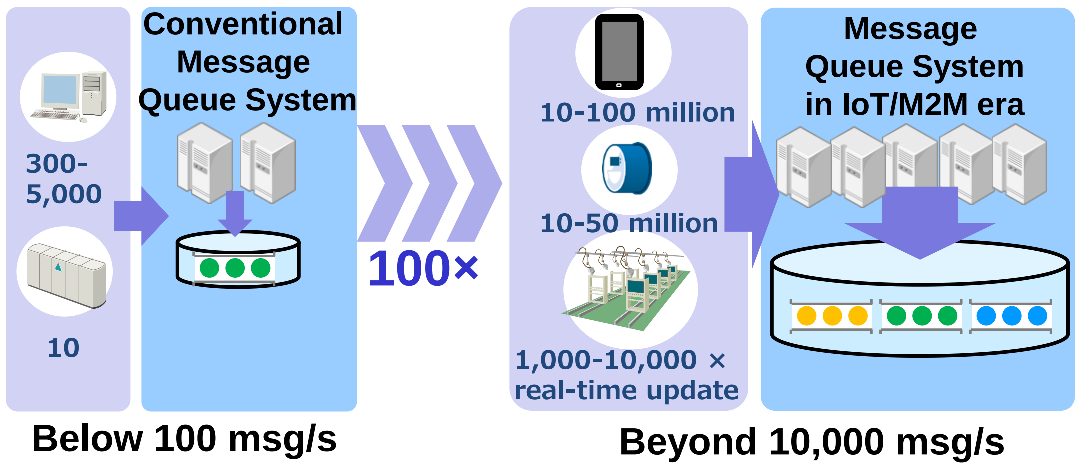

Conventional Message Queue System
300-5,000
10
100×
10-100 million
10-50 million
1,000-10,000 × real-time update
Message Queue System in IoT/M2M era
Below 100 msg/s
Beyond 10,000 msg/s

Figure 1.3 Drastic change of throughput requirements in message queue system

providing continuous and stable services becomes most important. The third phase is driven by the progress of IoT applications and its extension to the financial sector, industries and smart homes, which is currently ongoing and expected to continue until about 2020. The most important issue is how to control the massive heterogeneous traffic between devices and IoT service systems for achieving higher throughput, availability, and scalability than conventional systems.

In this thesis, we focus on the three phases mentioned above and discuss current and future challenges in message queue systems processing short messages from a realistic viewpoint. Furthermore, by the discussions in this thesis we intend to contribute to the better design and implementation of future IoT systems.

1.2. Overview of Message Queue Systems

Figure 1.4 outlines an example of the service system’s structure needed to process short messages. This system consists of four major components: connected devices, network, message queue system, and backend system. Connected devices include smart phones, smart meters, sensors, or other types of IoT devices. Network denotes the private or public network over which the connection takes place, e.g., Long Term Evolution (LTE) wireless network [34]. The message queue system is located as frontend system in the cloud and backend systems provide services and applications. The message queue system relays short messages between connected devices and the backend.

Generally, a message queue system has a messaging server relaying the message and message queues acting as a persistent (non-volatile) storage or data store. In this thesis, we define the process of *messaging* as receiving, handling, storing (queuing), and relaying (dequeuing) short messages. A conventional messaging server for enterprise service is in general a physical server running a messaging server program, while in this thesis we define *messaging server* only as a messaging server program (software) due to the consideration of server virtualization in cloud computing. Furthermore, a messaging server mainly consists of two programs: the *enqueue controller* (E-Ctrl), which processes the reception of messages and stores them in a queue (enqueue), and the *dequeue controller* (D-Ctrl), which processes the retrieval of messages from queues and their relaying.

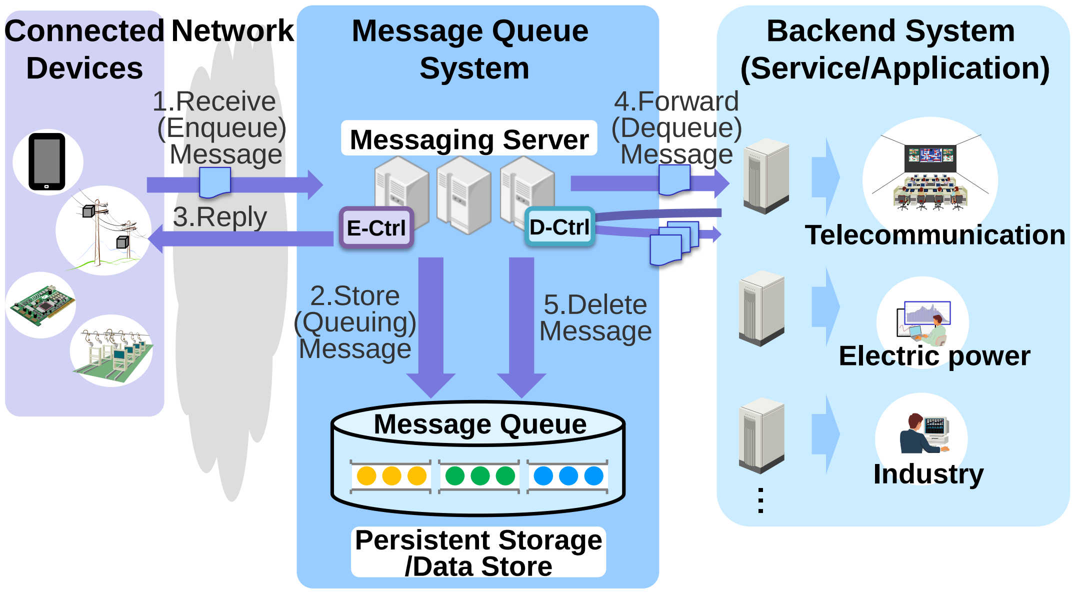


Figure 1.4 Outline of message queue system

A *message queue system* is also sometimes known under the terms of *message queue* (MQ) [35] or *message oriented middleware* (MOM) [36]. Message queue systems have several important tasks to perform, such as reliably relaying messages without loss, buffering of message traffic from devices, and providing interoperability between devices and backend systems. Furthermore, message queue systems can achieve reliable relaying of messages and buffering of message traffic from devices by using the *store-and-forward* method from source devices to backend systems (or to the next-hop message queue system). Processing of store-and-forward messages needs to be handled in the following order:

(1) receive (enqueue) messages from devices,
(2) store messages (queueing) into a queue in persistent storage,
(3) instantly reply to devices,
(4) forward (dequeue) messages to backend servers,
(5) delete stored messages from the message queue after successfully sending them.

If the message cannot be stored for any reason, e.g., due to a queue overflow, an error response is sent to the source device. The received message is normally sent instantaneously, but may also be delayed if the backend system is temporally unavailable. The message queue system keeps the message in its queue until the backend system becomes available again, which may take from several hours to a few days. Finally, the server deletes the message if the message was successfully sent or if a retransmission timeout occurred.

Message queue systems achieve interoperability by supporting various protocols. For example, MQTT, AMQP, or *Representational State Transfer* (REST) [37] are major protocols in the IoT era. This interoperability and absorption enables devices and the backend system to become *loosely coupled* and the message queue system enables the developer to rapidly interoperate between them. Under the condition that the message queue has both, sufficient performance to process the message traffic from devices and scalability in performance and storage, the message queue enables the developers of the backend system to design their system without considering the entire volume of the message traffic.

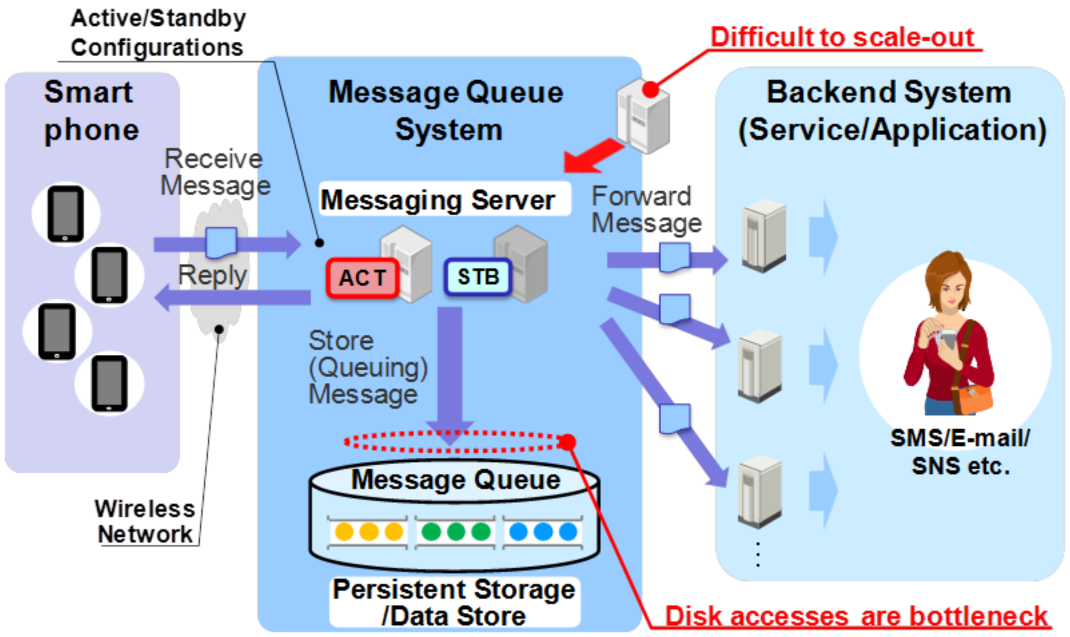


Figure 1.5 Issues in the first phase of message queue systems

1.3. Issues in Message Queue Systems

In this section, we provide an overview of the development of message queue systems and distinguish roughly in three different phases (see Fig. 1.1), which we now describe in more detail. Our focus lies on issues concerning the message queue system itself, and we do not elaborate on the other parts of the system, such as connected devices, network, or the backend system.

1.3.1. Issues in First Phase of Message Queue Systems

In the first phase, the enormous growth in the number of smart phones has led to an explosion in the volume of short message traffic encountered by telecommunication operators and other service providers. The most important issue of this phase is the high throughput and scalable processing of huge volumes of short messages in smart phone services.

Figure 1.5 outlines issues in the first phase of message queue systems. As mentioned above, message queue systems generally relay messages with the store-and-forward method such that incoming messages are first stored in a queue located within non-volatile storage and are then forwarded to the backend system server.

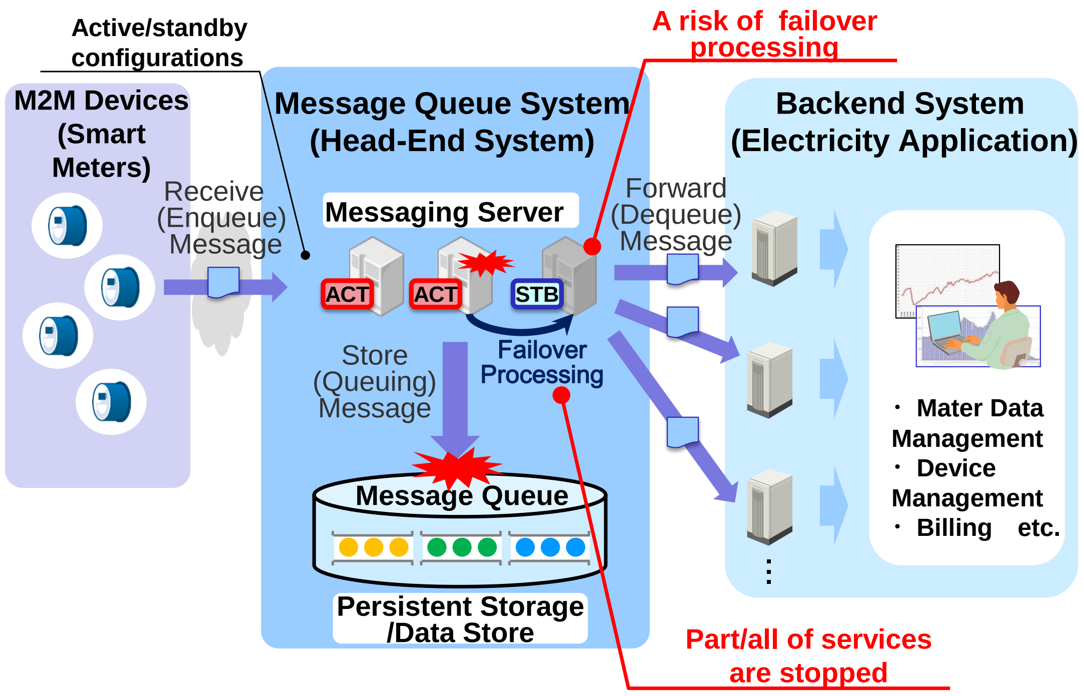

Active/standby configurations
A risk of failover processing
M2M Devices (Smart Meters)
Receive (Enqueue) Message
Message Queue System (Head-End System)
Messaging Server
ACT
ACT
STB
Forward (Dequeue) Message
Backend System (Electricity Application)
Store (Queuing) Message
Failover Processing
Message Queue
Persistent Storage /Data Store
・ Mater Data Management
・ Device Management
・ Billing etc.
Part/all of services are stopped

Figure 1.6 Issue in the second phase of message queue systems

Store-and-forward methods achieve reliable relaying and buffering of message traffic from devices, however, their main disadvantage is the low throughput due to non-volatile storage, such as when disks and storage systems are accessed, which turns out to be the bottleneck in relaying short messages. Additionally, conventional message queue systems generally have their message queues on RAID storage, which is difficult to scale-out.

Since these issues on high throughput and scalability for processing massive volumes of short messages are also fundamental for IoT applications, they will also be highly relevant to the IoT era.

1.3.2. Issues in Second Phase of Message Queue Systems

In the second phase, it has become common to connect M2M devices, such as smart meters or health equipment, to the network. For example, message queue systems are used in *Head-End Systems* (HES), which receives data through the network in a smart meter system [38]. The most important issue in this phase is the high availability and resilience for providing non-stop and stable services. Figure 1.6 outlines an example of such failover processing. Generally, mission-critical systems implement shared data and failover processing for providing *high-availability* (HA) services [39]. Failover processing includes application restart, process initialization, and recovery of data. These consist of special application-dependent processes as well as common processes, such as health check or error detection of hardware/software. However, catastrophic service failures of mission-critical systems with failover processing have frequently been reported [39-41]. Causes of these service failures are usually software or hardware defects and it is very difficult to exhaustively identify these defects during the system testing stage because all cases of failover processing, e.g., complex problems caused by only theoretically occurring defects, can hardly be tested. Therefore, a highly available message system without failover processing is needed.

1.3.3. Issues in Third Phase of Message Queue Systems

It is generally agreed that IoT services require information from historical or real-time data for their own objectives, such as optimization services. For example, message queue systems are expected to be applied to the Platform Tier, which receives device data through the network in the IoT reference architecture of the *Industrial Internet Consortium* (IIC) [42]. Figure 1.7 outlines issues in the third phase of message queue systems. In [43-45], IoT service systems are required to manage the massive volume of data generated by sensors from various fields, such as the financial sector, industries, smart homes, etc. In [46], optimization in smart manufacturing at enterprise level requires periodically collected data. In [47], general smart sensors may consist of single microchips and generate simple periodical data.

The general approach in IoT for finding patterns in data is to learn through trial-and-error data analysis. This approach requires collecting a large data volume for various analyses. Therefore, traffic volume from devices generating periodical message data has become enormous in IoT service systems.

On the other hand, the backend system collects data for various IoT objectives, such as monitoring and optimization, and retrieves messages from the queue at their own timing, which is non-periodic and process-dependent. These processing timings differ by context of message, message size, and other related data. To achieve higher throughput by fully utilizing computational resources, the backend system retrieves messages from the queue with a pull-based method [48]. In addition, progress in distribution platforms, such as Spark [49] or Storm [50], leads to a dramatic change in processing time of the

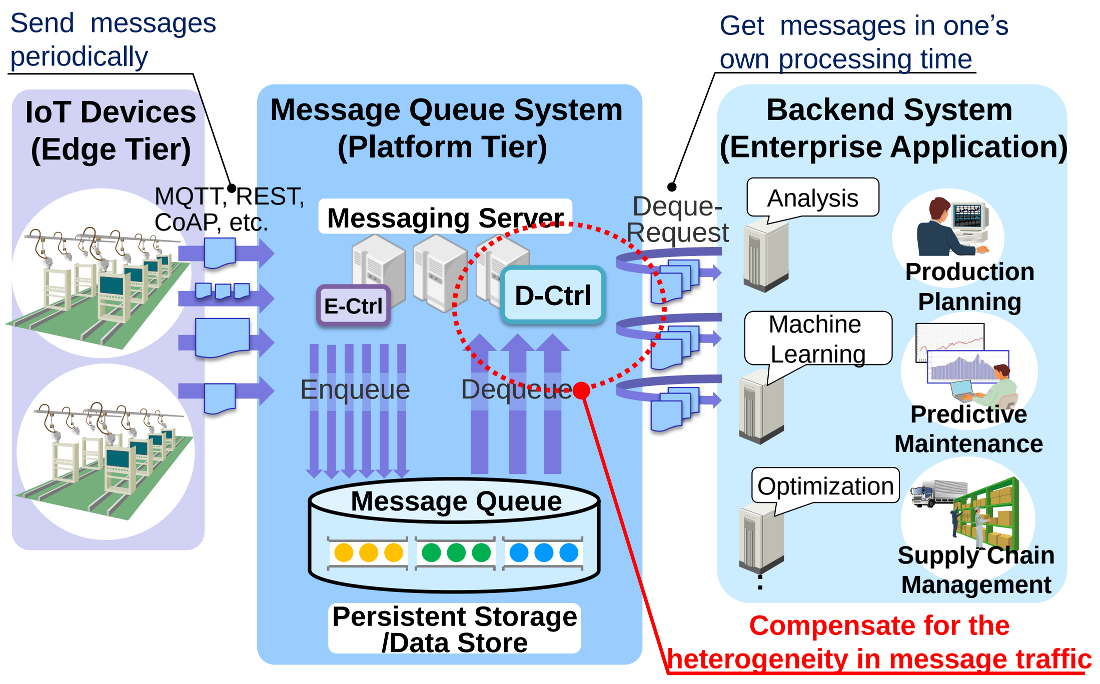

Send messages periodically
Get messages in one's own processing time
IoT Devices (Edge Tier)
Message Queue System (Platform Tier)
Backend System (Enterprise Application)
MQTT, REST, CoAP, etc.
Messaging Server
E-Ctrl
D-Ctrl
Enqueue
Dequeue
Deque-Request
Message Queue
Persistent Storage /Data Store
Analysis
Production Planning
Machine Learning
Predictive Maintenance
Optimization
Supply Chain Management
Compensate for the heterogeneity in message traffic

Figure 1.7 Issues in the third phase of message queue systems

backend system.

While devices send massive amounts of periodical messages, backend systems process IoT messages at their own timings. Therefore, the control function of the massive and heterogeneous message traffic in the message system becomes a crucial issue in Phase 3.

1.4. Outline of Thesis

In this thesis, we selected several important, but so far not well-discussed issues from those addressed in the previous section and studied solution approaches for message queue systems. In particular, this thesis focuses on the following main points in distributed message queue systems, spanning from short message services over M2M to the IoT era.

(1) Design of message queue systems with high-throughput queuing and scalability of short message services for smart phones
(2) Design and development of message queue systems with high availability through distribution methods for M2M services
(3) Design of message queue systems with increased throughput through dequeue scheduling in the IoT era

The contents of the chapters in this thesis are summarized in Fig. 1.8. and will be briefly summarized in the following subsections.

1.4.1. High-Throughput Message Queue System Based on Distributed In-memory KVS

In Chapter 2, we focus on high-throughput queuing techniques and architectures based on distributed message queue systems for smart phone services. We propose a message queue system for short messages based on a distributed in-memory key-value store (KVS) [51] to meet the requirements of high throughput and scalability, and to physically store messages in a queue structure while preserving the consistency of data in the respective queues. We present a method of high-throughput access to pipeline messages on an active TCP connection that is linked to a queue on message queue systems and its backup queue in KVS. We evaluate the performance of the proposed KVS and the message queue system corresponding to the KVS. The results show that both the KVS and message queue system achieve the required high throughput. Experimental evaluations further show that the throughput of our proposed method achieves 450% of that of the conventional method.

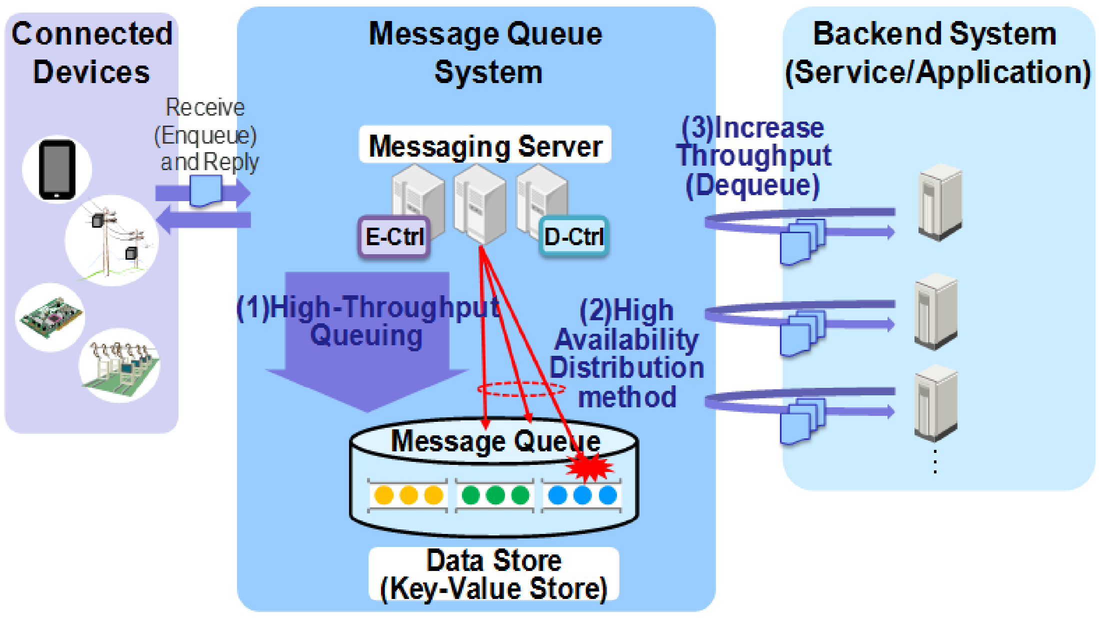

Connected Devices
Receive (Enqueue) and Reply
Message Queue System
Messaging Server
E-Ctrl
D-Ctrl
(1)High-Throughput Queuing
(2)High Availability Distribution method
Message Queue
Data Store (Key-Value Store)
Backend System (Service/Application)
(3)Increase Throughput (Dequeue)

Figure 1.8 Relationship between the 3 main topics of this thesis

1.4.2. Improved Resilience of Message Queue System through Server Distribution

In Chapter 3, we focus on a technique to achieve high availability for mission critical services using messages from M2M devices. We propose a resilient message queue system based on a distributed KVS. Its servers are interconnected among each other and messages are distributed to multiple servers in the normal processing state. This architecture can continue its messaging services regardless where any failures in the message queue server/process may occur without requiring any failover processing. We also propose further methods for improved resilience: the round-robin method with slowdown KVS exclusion and the two logical KVS counter-rotating rings to provide short-term availability in the message queue system. Evaluation results demonstrate that the proposed system can continue service without failover processing. Compared to the conventional method, our proposed distribution method reduces 92% of error responses caused by server failures.

1.4.3. Increased Throughput of Message Queue System through Dequeue Scheduling

In Chapter 4, we discuss a method for increasing dequeue throughput in message queue systems. In the IoT era, services require both information from historical or real-time data for their own objectives, such as optimization service, and learning through trial-and-error of data analysis for finding patterns in the data. This requires collecting large volumes of messages created by devices periodically. On the other hand, the backend system retrieves messages from the message queue at its own timing, which is non-periodic and process-dependent. Therefore, the control function of the massive and heterogeneous message traffic in the message system becomes a crucial issue, which can lead to dequeue throughput degradation. To solve this issue, we propose the dequeuing method called *Retry Dequeue-request Scheduling* (RDS) which can reduce the unnecessary transmission of dequeue requests to the message queues by waiting for messages to arrive at the message queues. In particular, RDS can better reduce throughput degradation due to *missed-dequeue* messages than the conventional method. By evaluations through simulation, we compare the throughputs achieved by the conventional method, RDS, and *Periodical Monitoring and Scheduling* (PMS), which is another dequeuing method proposed for reducing the number of *missed-dequeue*s by periodically monitoring each message queue to gather message counter information. Simulation results show that only RDS maintains highest throughput, regardless of an increase in the dequeue request rate. Experimental results further show that the RDS method achieves 80% higher throughput than the conventional method in real systems.

Chapter 2

High-Throughput Message Queue System Based on Distributed In-memory KVS

2.1. Introduction

The enormous growth in the number of smart phones has led to an explosion in the volume of short message traffic encountered by telecommunication operators and other service providers. Especially, short message communication services such as e-mail, *short message services* (SMS), and *social networking services* (SNS) have become essential for our life. For instance, message traffic at specific times, such as after the occurrence of disasters, the turn of New Year, and other popular events, may reach over 143,000 transactions per second [52]. This burst of transactions is beyond the capacity of conventional message queue systems and forces telecommunication operators to regulate the amount of transactions [53]. For processing the large and still growing amount of short messages traffic, much higher throughput has been required for message queue systems. To process this increasing traffic of short messages, high scalability is required for enabling the greater processing capacity and memory sizes. Furthermore, simultaneous availability of message queue systems is also required in the same way as for conventional systems.

As mentioned before, message queue systems conventionally relay messages with the store-and-forward method. Messaging servers of message queue systems receive messages once and store these received messages to persistent storage, such as disks, after which they successively relay the messages to the backend system. This enables an instant response to the devices and shortens the session activity time.

Here, let us focus on the function of message queue systems for smart phone services. In those systems, there are several important functions such as buffering and controlling traffic in the system, stabilization of the system, and avoidance of network contention between devices and message queue systems. However, when the message queue receives a lot of short messages, disk accesses for storing (queuing) these messages generally becomes the bottleneck for throughput. For example, if the throughput of a single server for *sendmail* [25] or *postfix* [54], which are both widely used for e-mail message services in companies, drops to below 100 msg./s, it will become too low to process a large amount of short messages [26].

Hence, to solve these concerns, we follow the approach of applying a distributed in-memory KVS instead of persistent storage to message queue systems. We aim at achieving high throughput and scalability of the message queue system and solve the following issues in this chapter. We propose an architecture for a message queue system with high throughput and scalability based on distributed in-memory KVS. We also design a high-throughput queuing (storage) method between message server and KVS with availability and process of KVS to achieve high-throughput queuing.

This chapter is organized as follows. First, we provide an overview of the message queue system for smart phone services. We then present the architecture and proposed methods. Next, we reveal the implementation and performance evaluation. Finally, we describe related work and give a conclusion.

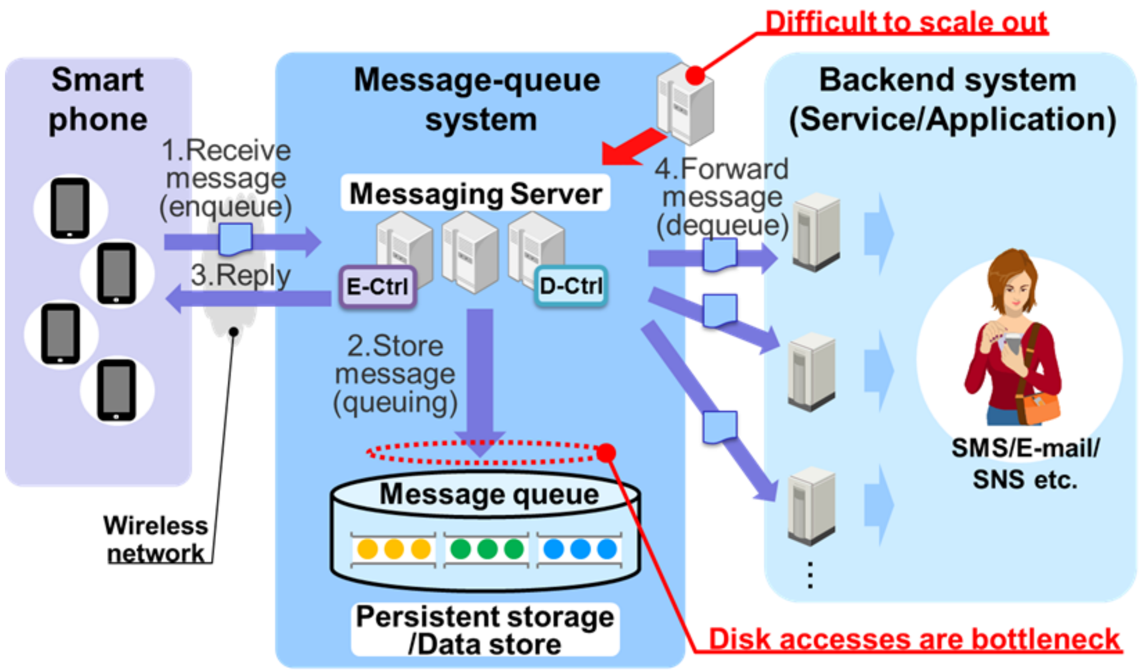

Difficult to scale out
Smart phone
Message-queue system
Backend system (Service/Application)
1.Receive message (enqueue)
3.Reply
Messaging Server
E-Ctrl
D-Ctrl
4.Forward message (dequeue)
2.Store message (queuing)
Message queue
Persistent storage /Data store
Wireless network
SMS/E-mail/ SNS etc.
Disk accesses are bottleneck

Figure 2.1 Example of message queue system for smart phone services

2.2. Overview of Message Queue Systems in Smart Phone Services

2.2.1. Components of Message Queue Systems

Figure 2.1 outlines an example of the system structure for short message service for smart phones. Message queue systems are widely used for a variety of services, such as e-mail, SNS, SMS, and other push notification services from data centers to smart phones.

Message queue systems receive messages from devices via the wireless network and relay them to the backend system or next-hop message queue system with the store-and-forward method. Message queue systems support various protocols for their own services. For example, *simple mail transfer protocol* (SMTP) and *multimedia messaging service* (MMS) are used in e-mail services, while *short message peer-to-peer* (SMPP) is used in SMS and other push notification services.

Main functions of message queue systems are reliable in relaying messages to the backend system without message loss and with congestion control of message traffic from devices to the backend system. In wireless networks, messaging servers decrease the failure rate of transmission and reduce the number of active sessions by quick responses with the store-and-forward method.

Messaging systems maintain several message queues for each backend system or next-hop message queue system. For each message queue, locks at the internal queues are required for relaying messages by the messaging server to provide exclusive access (these functions are denoted as *queue transactions*) as well as relay message priorities. Message queue systems enable congestion control of each backend system by regulating dequeue traffic from each message queue to the backend systems. Moreover, message queues store billing data or metadata depending on the situation and, therefore, their guarantee of data consistency is crucially important.

Conventional message queue systems have a message queue in RAID storage and *high-availability* (HA) cluster structure. As mentioned in Sect. 2.1, disk accesses at queues generally become the bottleneck for throughput. Furthermore, HA clusters generally have active/standby configurations making it difficult to scale-out for enhancing throughput.

2.2.2. Conventional Research on Distributed in-Memory KVS

Many efforts have been expended on distributed in-memory KVS for high throughput and scalability. In–memory KVS *memcached* [55], which is known as high throughput KVS, is currently used as a cache by many companies, such as Facebook [56]. *Memcached* runs on a single server and stores messages without any duplication and distribution, which is not utilized as persistent data store in general. On the other hand, in-memory KVS can be utilized as persistent data store by multiplication and distribution of data on memory. Nevertheless, they have two disadvantages compared with RAID storage used in conventional message queue systems. The first is that data is lost if all nodes having the same replicated data are down at the same time. However, as power supplies are duplicated at the data center and data is periodically backed up to disks, there is only a small probability that data will be lost. The second disadvantage is that storage capacity of KVS is usually not very large because memory is more expensive than disks.

In previous work related to messaging systems, Wang et al. [57] proposed a distributed message queue supporting *queue transaction* while relaying messages and guaranteeing the order of message processing by referring to additional metadata to control the message queue stored in typical in-memory KVS with simple key-value data

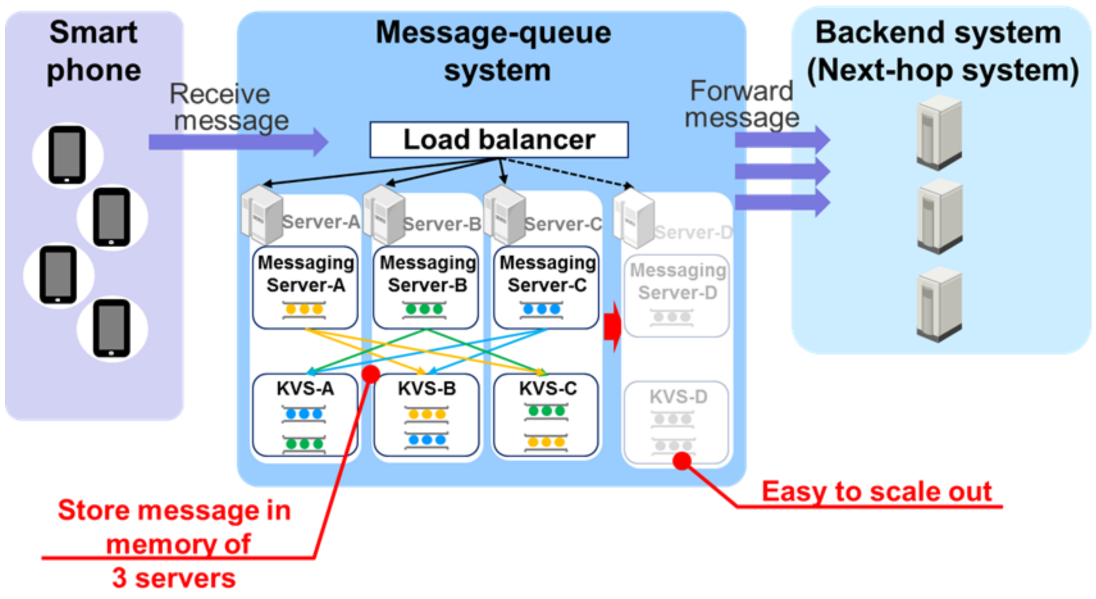


Figure 2.2 Proposed architecture of message queue system

structure. In the following, this is called *simple KVS-based method*. In general, KVS refers to a simple data model consisting of a pair of key and value, which is completely different from conventional queue data structures. Although the *simple KVS-based* method presents an implementation of message queues in distributed systems, it does not consider about high-throughput queuing, and data structure or processes of KVS.

2.3. Message Queue System-based on Distributed in-Memory KVS

This section presents the architecture and implementation method for message queue systems to address the issues mentioned in Sect. 2.1.

2.3.1. Architecture of Message Queue Systems

2.3.1.1. Implementation of High-Throughput Queuing and Scalability

Figure 2.2 shows the proposed architecture of a message queue system based on the distributed in-memory KVS. The proposed KVS differs from general KVS in having a queue structure on the server's physical memory. Each server has both, a messaging server program for relaying messages and an in-memory KVS program. As mentioned before, we refer to the message handling program as messaging server and refer to the

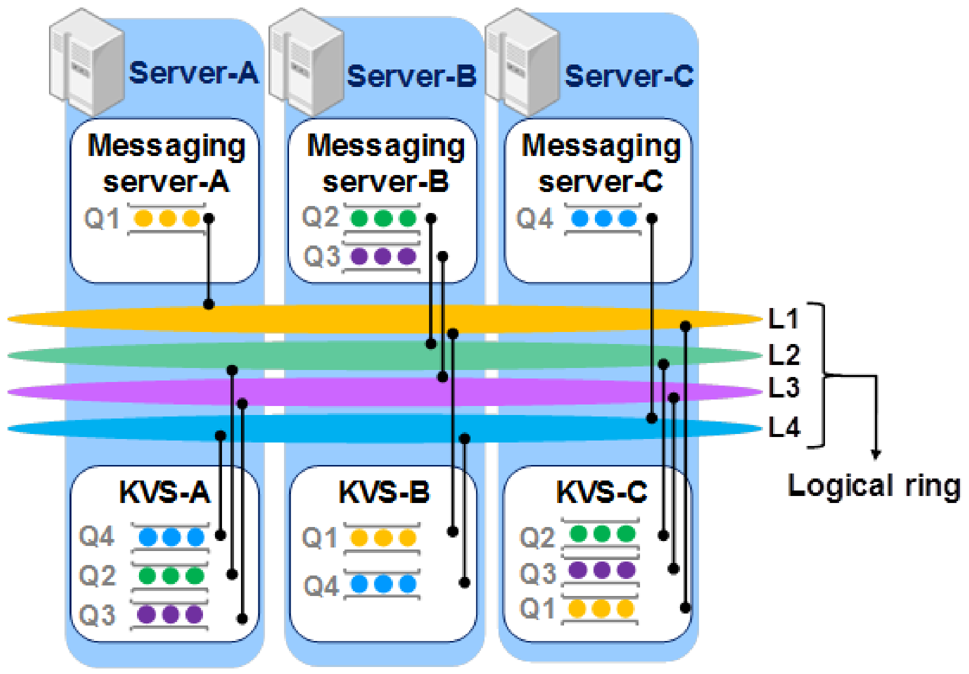


Figure 2.3 Example of logical rings of message queues

KVS program simply as KVS. Each server acts independently and communicates via an internal network among the servers.

To implement highly available and distributed message queues, they must be separately deployed into both, messaging servers and KVS as shown in Fig. 2.3. A load balancer allocated in front of each server distributes received messages to the messaging servers. After receiving messages from the load balancer, the messaging server stores these messages into the message queue and delivers them in accordance with the store-and-forward method. Messaging servers retain a message not only in a single KVS message queue, but also in other message queues of the KVS, as well as in a local message queue of the messaging server itself. This means that a single message is duplicated and exists on 3 servers simultaneously. Here, we decided the number of duplicates to achieve the same availability as RAID storage [58, 59]. Hence, messaging servers provide high availability (fault tolerance) to avoid message loss even in the case where up to two servers are broken down, achieving high-throughput access without the bottleneck of disk access.

Messaging server and KVS are homogeneously aligned in parallel on messaging servers, which is effective for balancing load, removing single points of failure, and flexible scaling of servers. In the following sections, this structure is referred to as *distributed message queue*.

2.3.1.2. Logical Ring Structure of Distributed Message Queues

A messaging server is composed of logical rings covering the message queues of two KVS servers and the local message queue. Figure 2.3 demonstrates the structure of logical rings over message queues.

The messaging server associates with the queues of two KVS having the same messages via logical ring and then synchronously processes messages (storing/enqueuing, dequeuing, or deleting). For instance, after receiving messages, the messaging servers distribute the same messages into both a local message queue and two message queues associated by a logical ring. In other words, the state of linked queues sharing the same logical ring is synchronized.

In each logical ring messaging servers can maintain multiple logical rings and synchronize. In Fig. 2.4, the messaging server *messaging-B* has two message queues, Q2 and Q3, where Q2 is associated with KVS-A and KVS-C via logical ring L2, and Q3 is associated with KVS-A and KVS-C via logical ring L3. From another point of view, KVS also retain multiple logical rings. In Fig. 2.4, KVS-A retains L2, L3, and L4 connected to Q2 of *messaging-B*, Q3 of *messaging-B*, and Q4 of *messaging-C*, respectively. For one process, KVS performs message processing of a single message queue, whereas for multiple processes, KVS can perform message processing of multiple message queues in parallel (details are explained in Sect. 2.3.3).

In CAP theory [60] the terms C, A, and P refer to *consistency*, *availability*, and *partition tolerance*, thus, the structure of message queues on a logical ring obtains C-P characteristic. This characteristic solves the issue of implementation of distributed message queues. Although availability is not originally satisfied in the C-P model, the proposed system also keeps availability on a certain level by allocating multiple logical rings connected to the messaging servers and the two KVS. In this structure, even though some of the logical rings may stop their function, the remaining logical rings can keep continuous services.

2.3.2. Proposed KVS with Queue Structure

As mentioned before, the KVS has a queue structure on the physical memory of the server, and the messaging server synchronizes message processing between the local message queue and message queues of the two KVSs on the logical ring. In terms of

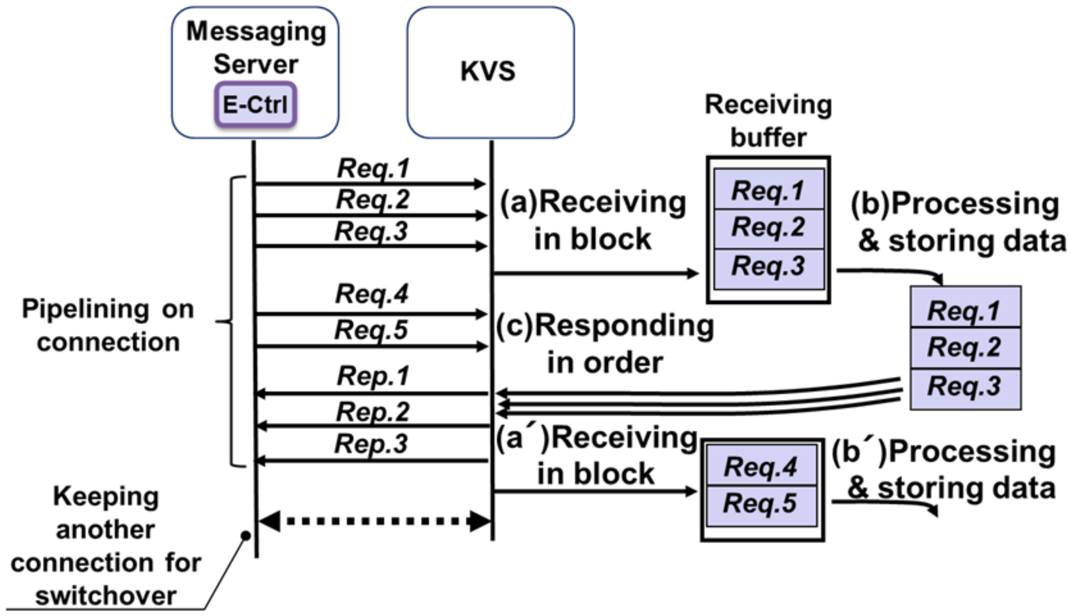


Figure 2.4 Method of communication to KVS for high-throughput queuing

fault tolerance and availability of the whole messaging system, the proposed method has two major features.

The first feature of the proposed method is the reduction of frequency and amount of communication required to synchronize the message queues for achieving high-throughput queuing. In the *simple KVS-based method*, extra processing of metadata is required for every access of queues. For example, in assuming a simple model for adopting the *simple KVS-based method*, messaging servers need to receive and update the metadata for each storing process. In this case, the communication frequency of the conventional method becomes more than 3 times larger than that of the proposed method. The behavior of messaging systems with the high-throughput queuing method is detailed in Sect. 2.3.3.

The second feature of the proposed method is the shortened downtime during server failures. When broken or stopped servers recover from failure, the messaging server gets all backup messages from the KVS message queues and then restarts services after synchronizing the message queues to guarantee data consistency. Due to the KVS retaining the physical queue structure, the messaging server efficiently obtains messages by just a single communication. The behavior of messaging systems for failure recovery is detailed in Sect. 2.3.4. On the other hand, in the *simple KVS-based method*, messages of queues are not accumulated in a specified server and, thus, the messaging server must access all servers repeatedly to resume each message one by one.

This restriction prolongs their recovery time.

2.3.3. High-Throughput Queuing Method with KVS

The communication method between the messaging server and KVS to achieve high-throughput queuing and guaranteed data consistency is described in Fig. 2.4. Communication between messaging server and KVS in Fig. 2.4 corresponds to the process between Q1 of messaging server A and Q1 of KVS B for synchronization in Fig. 2.3. Each queue is connected by an individual TCP connection. Although backup TCP connections are also prepared in practice to maintain availability, its explanation is omitted here for the sake of brevity.

As shown in Fig. 2.4, the messaging server sends multiple synchronization requests over a single TCP connection corresponding to a queue. KVS collectively receives the requests (Fig. 2.4 (a)) and processes theses requests successively (Fig. 2.4 (b)). While the KVS sends multiple replies (Fig. 2.4 (c)), processes described in Fig. 2.4 (a) and Fig. 2.4 (b) are also performed simultaneously.

By communicating the queue messages of each server through single TCP connections and processing requests in the order of their arrival sequence at the KVS, the order of message processing between messaging server and KVS is guaranteed. Furthermore, by issuing sequence numbers in every request and identifying the state of synchronization through this sequence number, data consistency is maintained.

To increase throughput of synchronized processing while guaranteeing data consistency of message queues, the conventional method increases multiplicity by increasing the number of TCP connections and the proposed method increases the data density (multiplicity) on TCP connections. The former access method is used in the *simple KVS-based method*. However, this method raises several concerns: the possibility of throughput degradation due to exclusive control among several TCP connections for strict guarantees of data consistency and the complexity of multiple design parameters of the network, such as the optimal number of TCP connections (or controls) to maximize message throughput [61].

On the other hand, the proposed method achieves an efficient internal queue lock of *queue transactions* by the simple design of using single TCP connections with every one-to-one message queue. However, for the proposed method, several demerits are considered. For example, due to a few TCP connections, influences of congestion control [61] and connection latency of application requests may become significant. In addition, the proposed method may not efficiently utilize CPUs on multi-cores for parallel processing. In Sect 2.4.4, we discuss and evaluate to what degree these influences are negligible.

2.3.4. Behavior of Failure and Recovery

We explain the behavior of our proposed method by referring to Figs. 2.3 and 2.4. When server B breaks down, both messaging server B and KVS B stop operating. The subsequent behavior is described as follows.

(1) In the wake of stopping messaging server B, the load balancer B isolates the stopped messaging server B and keeps distributing messages to the messaging servers A and C, thereby, steadily continuing message processing.
(2) Messages that were processed shortly before messaging server B broke down are also stored into KVS A and C, and are resumed at the time messaging server B recovers.
(3) Breakdown of KVS B affects messaging servers A and C, which share the logical ring connected to the queue of KVS B. Concretely, messaging servers A and C detect the breakdown of KVS B by reply timeout, isolate the KVS B, and continue service in duplication mode.

On the other hand, when server B recovers from failure, it tries to restart both messaging server B and KVS B. The subsequent behavior is described as follows.

(1) Messaging server B obtains messages that were partly processed before it broke down from KVS A or C sharing the same logical ring. After that, messaging server B restarts its service.
(2) Messaging servers A and C using KVS B detect its recovery. KVS B simultaneously obtains and synchronizes messages from the message queues of messaging servers A or C. After that, KVS B restarts service.

2.4. Evaluation of High Throughput Queuing

2.4.1. Implementation and Evaluation Environment

We developed a messaging server and KVS as an event driven architecture [62]. These server programs are implemented in the C programming language. For the environment of our evaluation, we assume an e-mail system for smart phones representing the message queue systems.

2.4.2. Throughput Evaluation of Message Delivery

We evaluate throughput of the message queue system proposed in Sect. 2.3.1 and compare it with that of a conventional message queue system using RAID storages. An overview of this evaluation architecture is depicted in Fig. 2.5.

The test program sends messages (e-mail data) to the messaging server by SMTP. The messaging server stores messages into KVS and RAID storage using proposed and conventional methods, respectively. After that, the messaging server forwards the messages to the message transfer agent (MTA), which is a typical backend system for the e-mail service, after which it deletes them from the message queue.

The test program sends messages to the messaging servers based on a predetermined transmission rate. We adopt the combination of different message lengths, consisting of 70% of 1 KB messages and 30% of 10 KB messages, used for the evaluation of the conventional method [26].

First, we evaluate the maximum throughput (msg./s) defined as the rate at which the messaging server can steadily process store-and-forward e-mails without overflowing the messaging queues. Figure 2.6 shows the result of this simulation. This result reveals that the proposed message queue system achieves 3,600 msg./s, which is 4.5 times larger than that of conventional message queue systems having the bottleneck of disk access (850 msg./s).

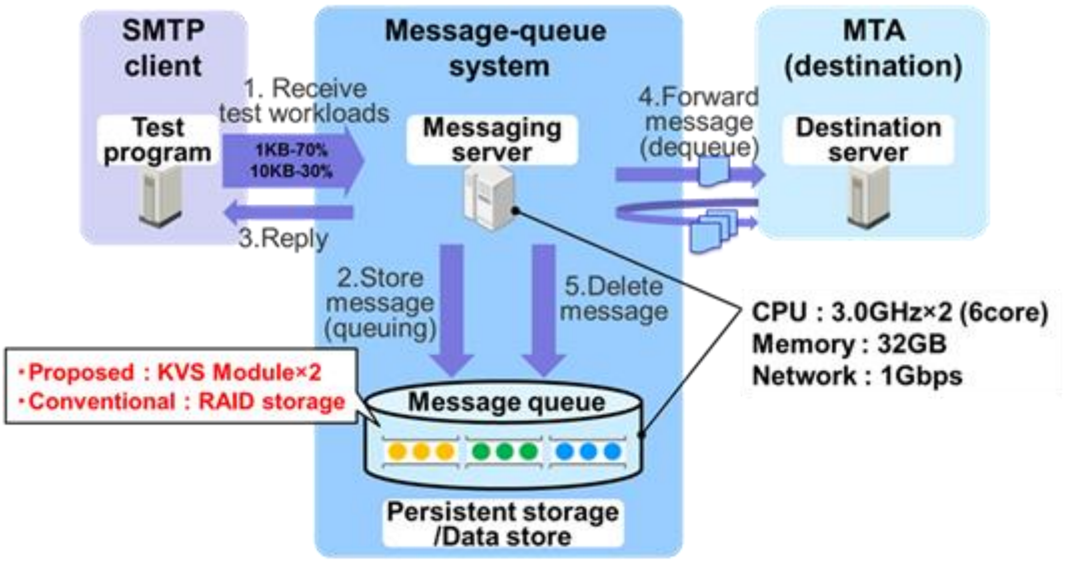


Figure 2.5 Method for the evaluation of message queue system.

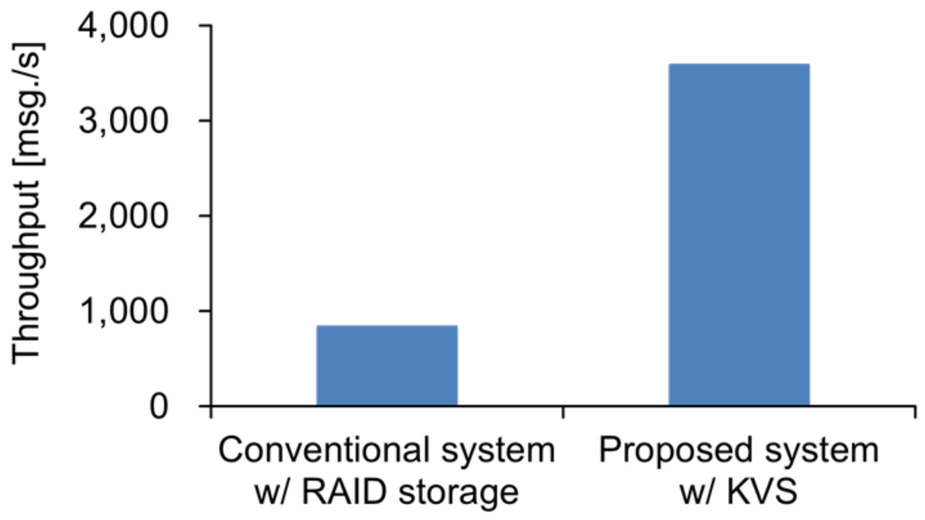


Figure 2.6 Throughputs of message queue systems

2.4.3. Performance of Proposed KVS

Next, we evaluate the method of communication processing to synchronize among distributed message queues as proposed in Sect. 2.3.3. First, we compare the throughput of the proposed method and the *simple KVS-based method*. Second, as a benchmark of KVS performance, we compare throughputs of the proposed queue-type KVS with *memcached* representing the in-memory KVS. Finally, we evaluate and discusse the dependence of throughput on the number of message queues of KVSs.

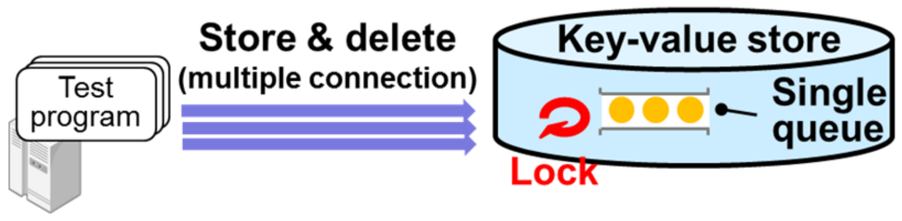
2.4.3.1. Throughput Comparison with *Simple KVS-based Method*

To compare the throughput of proposed method and *simple KVS-based method*, we experimentally produce results for KVS by simulating the *simple KVS-based method*. Figure 2.7 describes the overview of the evaluation for comparison of the throughputs. The test program sends a pair of enqueue request of 0.1KB fixed messages and delete request as one transaction to the KVS. Hence, we evaluate the maximum number of transactions that can be successfully processed by KVS.

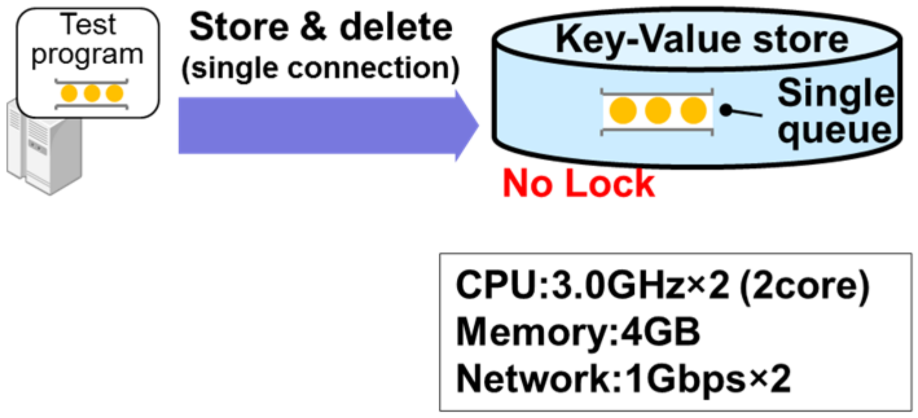
The prototype KVS based on the *simple KVS-based method* retains one message queue, receives transactions via multiple TCP connections from the test program, and performs message processing after setting an internal queue lock every time. In this research, we vary the number of TCP connections from 1 to 100.

On the other hand, we evaluate the maximum throughput for our proposed method while continuously connecting one test program and one message queue of a KVS by a single TCP connection.

Figure 2.8 shows the results of the throughput evaluation. Throughput of the proposed method is 91,000 msg./s, which is 3.8 times larger than that of the *simple KVS-based method* (24,000 msg./s with 100 connections). This results from the difference of the exclusive control methods and communication processing methods.



(a) The method based on standard KVS



(b) Proposed method

Figure 2.7 Evaluation of standard and proposed KVS methods

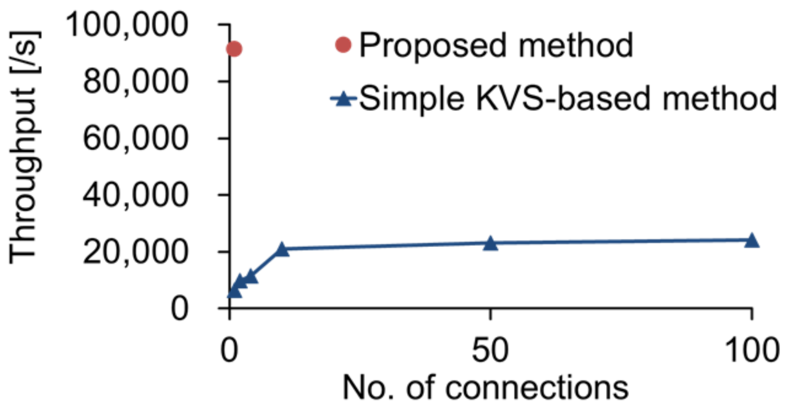


Figure 2.8 Transaction throughputs of KVS for different methods.

2.4.3.2. Throughput Comparison with *memcached*

We compare the throughput of proposed queue-type KVS with *memcached* as the benchmark of in-memory KVS. *memcached* is well known as simple and high-throughput KVS and is also an event-driven architecture and implemented in the C language. Figure 2.9 describes the evaluation to compare both KVSs. Note that the condition of traffic from the test program is same as that shown in Sect. 2.4.3.1. We evaluate six values of message lengths (0.4, 1, 2, 4, 10, and 20 KB).

In this evaluation, due to that the total number of cores being four (2 cores × 2 CPUs), the number of TCP connections used in *memcached* is also set to four. Meanwhile, between 1 and 4 TCP connections in proposed KVS are prepared for connection between one and four queues. We evaluated their maximum throughputs under these conditions.

In Fig. 2.10, the x-axis and y-axis show message length and corresponding throughputs, respectively. In addition to throughputs of KVS for the proposed method and *memcached*, the throughput between test program and KVS is also described as a reference value of the critical performance with message forwarding on a 1Gbps network.

From the results, the maximum throughput of the proposed KVS is 200,000 msg./s when the message size is 0.4KB and there are two queues. Moreover, the maximum throughput of the proposed KVS is 100,000 msg./s when the message length is 1KB and there is only a single queue. Furthermore, in the range where the message length is larger than 2KB, the maximum throughput of the proposed KVS reaches the critical performance value of the 1Gbps network when there is only a single queue. The results for message lengths 10KB and 20KB are omitted in Fig. 2.10. Here, we confirm that doubling the number of queues does not affect the throughput when the message length is 0.4KB. The relationship between throughput and the number of queues of KVS, i.e., the total number of TCP connections, is discussed in Sect. 2.4.4.3.

In comparison with *memcached*, the maximum throughput of KVS is approximately 1.4 times as large as that of *memcached* with 1 KB messages. With 0.4 KB messages, the maximum throughput of KVS is approximately 2 times as large as that of *memcached*. These results indicate that the proposed KVS achieves high throughput when short messages are smaller than 1KB.

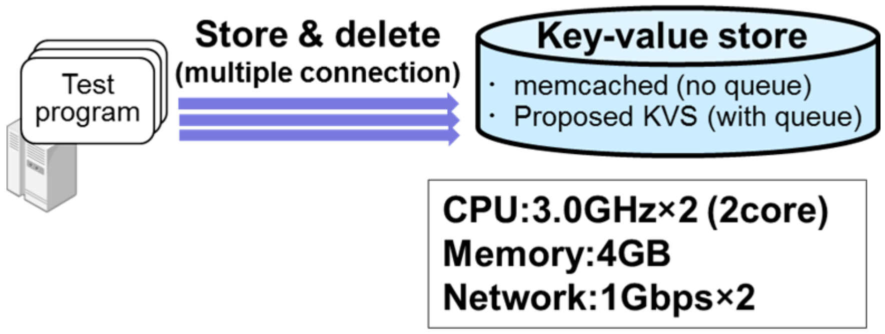


Figure 2.9 Evaluation method of KVS

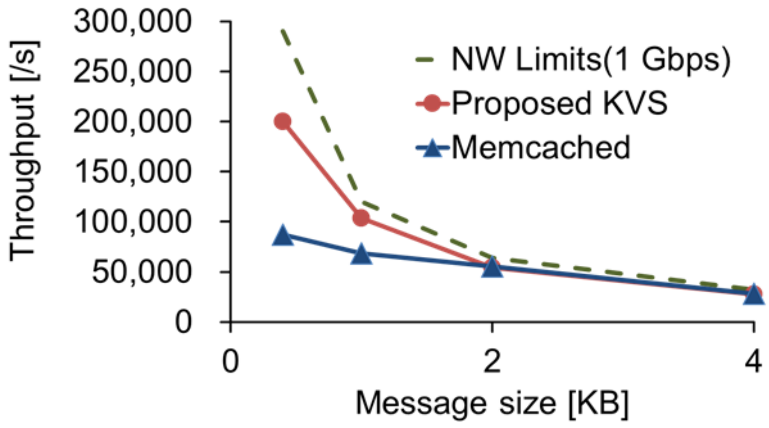


Figure 2.10 Transaction throughput of KVS for different messages sizes

Although proposed KVS performs more functions including processing for high availability and data consistency than *memcached*, the proposed communication processing method is superior to that of *memcached* in dealing with short-length messages. On the other hand, in the range where the message length is larger than 2KB, throughput of the proposed method reaches critical performance values and there is a margin to perform additional operations in the CPU usage (CPU usage is 3%/2.7% when dealing with 10KB/20KB messages, respectively). These facts indicate that the proposed method is always effective to enhance throughput even if more network capacity is available.

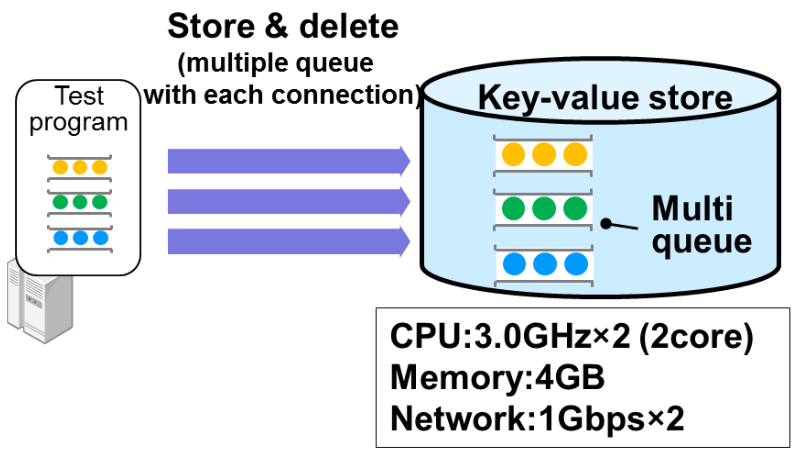


Figure 2.11 Evaluation method of KVS for different number of message queues

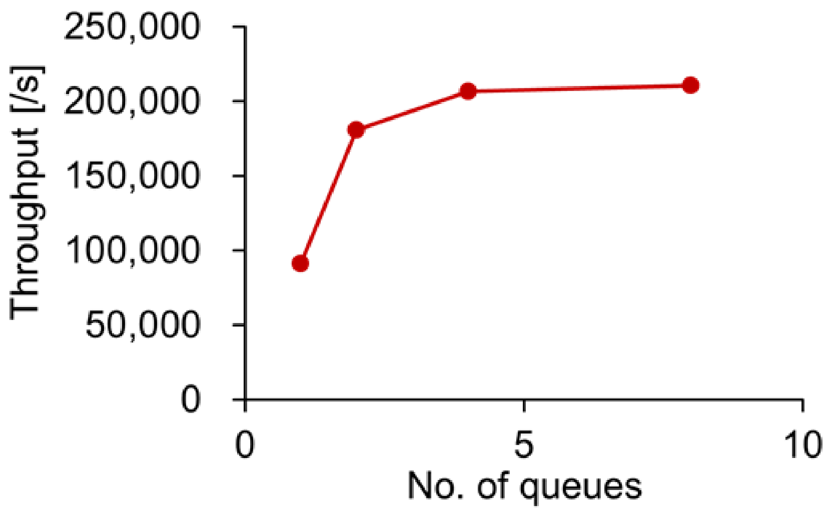


Figure 2.12 Transaction throughput of KVS for different number of message queues

2.4.3.3. Relationship between Number of Message Queues and Throughput

We evaluate the dependence of throughput on the number of logical rings (the number of queues) which KVS retains and processes. Figure 2.11 shows the overview of this evaluation.

Both, test program and KVS have multiple queues and each queue is connected by a single TCP connection. We evaluate throughput of the KVS when the number of queues (the total numbers of TCP connections) is 1, 2, 4, and 8. Referring to Sect. 2.4.4.2, maximum throughput is obtained when message length is 0.4KB for 2 message queues or 1KB for 1 message queue. To eliminate the limit of network margin and accentuate the effect of the different number of queues, we set the message length to 0.1KB.

Figure 2.12 shows the throughput of the KVS for different number of message queues. For the proposed method, throughput for two message queues is 180,000 msg./s, which is twice of that when using a single message queue. In this evaluation, even if the number of message queues is more than two, CPU usage is at most 7%, which shows that CPU is not a bottleneck. Meanwhile, for a message length of 0.1KB, the critical performance value with message forwarding on the 1Gbps network is 1,300,000 msg./s, which means that the network is also not the bottleneck.

There are three major tasks of the KVS: (a) receive requests, (b) store messages into memory, and (c) send reply, as shown in Fig. 2.4. Additionally, both (a’) waiting for requests after (c) and the communication time between the test program and KVS also affect the throughput. The KVS processes multiple queues in parallel and each message queue is handled by one process. Here, we consider that the throughput difference is not caused by processes (a), (b), and (c) due to the margin of the CPU. Besides, examination of the test program reveals that test program is no bottleneck for the CPU. Therefore, we presume that the communication time between the test program and KVS becomes the bottleneck. Concretely, the bottleneck originates from the window-based flow control of the TCP connection. When the number of message queues (the total number of TCP connections) increases, the bottleneck of communication between test program and KVS seems to mitigate and throughput is improved due to each queue being processed in parallel. Because KVS retains multiple message queues on logical rings as shown in Fig. 2.3, the proposed method is less subject to the influence of communication bottlenecks.

2.4.4. Experience in Real Message Queue Systems

Message queue systems as proposed in this chapter have been already applied to continuously support commercial services for more than five years without any service interruptions. This message queue system enables users to reduce the efforts for system construction without requiring RAID storages. Meanwhile, this message queue system also supports flexible system extension of the number of servers. From these features, the proposed message queue system satisfies both high availability and scalability. Moreover, the message queue system with distributed message queues is easy to interrupt and reboot, which can update software without stopping. Furthermore, the possibility that messaging server and KVS can coexist in one server contributes to a reduction in maintenance and monitoring workload compared with conventional message queue systems.

2.5. Related Work

For message queue systems achieving scalability and availability, several mail systems utilize distributed file systems based on hash tables [63-65] or distributed KVS Cassandra [66]. These proposals discuss scalability and availability by targeting the mailbox system, however, both approaches of distributed message queues and obtained throughputs are not mentioned in these works.

Moreover, in addition to the structure based on the pair of key and value in KVS, column-type KVS [66] is used for storing data into N-dimensional associative arrays. To our best knowledge, there has been no research on message queues utilizing such column-type KVS, however, it can be physically used as queue-type KVS by combining column-type KVS and *simple KVS-based method*. This KVS does not include the solution of high-throughput data synchronization in distributed systems. Hence, we conclude that the queue-type KVS proposed here is superior in performance.

2.6. Conclusion

In this chapter, we proposed an architecture for satisfying high throughput and high scalability in a message queue system for processing massive volumes of short-length messages through a distribution method of queue-type in-memory KVS and synchronized processing of distributed queues by single TCP connections.

We embedded the proposed architecture and method into a mail system for smart phones and performed evaluations of this system. The evaluation results revealed that throughput of the proposed message queue system achieves 3,600 msg./s per server, which is 4.5 times higher than that of the conventional method cooperating with RAID storages. Moreover, the throughput of the proposed KVS is 200,000 transactions/s with 0.4 KB messages, which is 2 times the performance of *memcached*.

Chapter 3

Improved Resilience of Message Queue System through Server Distribution

3.1. Introduction

Due to the progress of mobile network technology such as *Long Term Evolution* (LTE) it has become popular to connect *Machine-to-Machine* (M2M) devices, such as smart meters, health monitoring devices, or heavy equipment to the network. According to [1], the number of connected wireless devices reached 10 billion in 2014 and this number has been steadily increasing since then, especially with the continuous enhancements of the social infrastructure through M2M services. The service system, in particular its message queue system, is required to have high availability, which is considered among the most important features of mission-critical systems beside high-throughput processing of huge traffic volumes sent by devices. However, two issues need to be addressed to simultaneously satisfy high availability and high-throughput processing in a message queue system.

The first issue is that failover processing itself has a risk of failure. Generally, mission-critical systems implement shared data and failover processing for providing *high availability* (HA) services [39]. Failover processing includes application restart, process initialization, and recovery of data. These processes consist of special application-dependent processes, as well as common processes, such as health check and error detection of hardware/software. Their design and implementation become much more complex as the volume of messages to process becomes larger.

However, in recent years, catastrophic service failures of mission-critical systems processing large volumes of messages with failover processing [39-41] have frequently been reported. Causes of these service failures are usually software or hardware defects and it is very difficult to exhaustively identify these defects at the system testing stage because all possible cases of failover processing, e.g., complex problems caused by only theoretically occurring defects, can hardly be tested in advance. Therefore, a high-availability messaging system without failover processing is needed.

The second issue is to balance between consistency and availability. For message queue systems, a strong consistency of messages and message queues is the highest requisite to maintain reliable messaging. Furthermore, these message queue systems also require maintaining the state of the messaging process and the internal queue lock, which are denoted as queue transactions. To process large volumes of messages, the message queue system generally consists of multiple servers, however, maintaining consistency among these servers is a common issue for distributed processing [67]. Following Consistency Availability Partition (CAP) tolerance terminology [60], we can make a trade-off between consistency and availability at the KVS.

Here, an approach is required in which consistency in the message queue system can be guaranteed by the KVS functions and availability is improved by our proposal in this chapter. More specifically, not only 365 days of non-stop service is mandatory as long-term-availability, but also short-term-availability is required, e.g., even during a transient state when a failed server is being isolated or traffic congestion is being eliminated, the messaging service can be continuously provided without performance degradation. In this chapter, we propose a fabric message queue system without failover processing. *Fabric message queue* describes the distribution of messages to multiple servers in normal processing state to avoid failover processing. This system has the following two features and advantages.

(a) The message queue system architecture based on distributed in-memory KVS can provide long-term availability, i.e., it can continue its service wherever in the message queue system server/process failures may occur, by distributing messages to multiple servers, as well as by guaranteeing strong consistency of the messages and queues by using KVS functions and the Paxos protocol [68, 69].

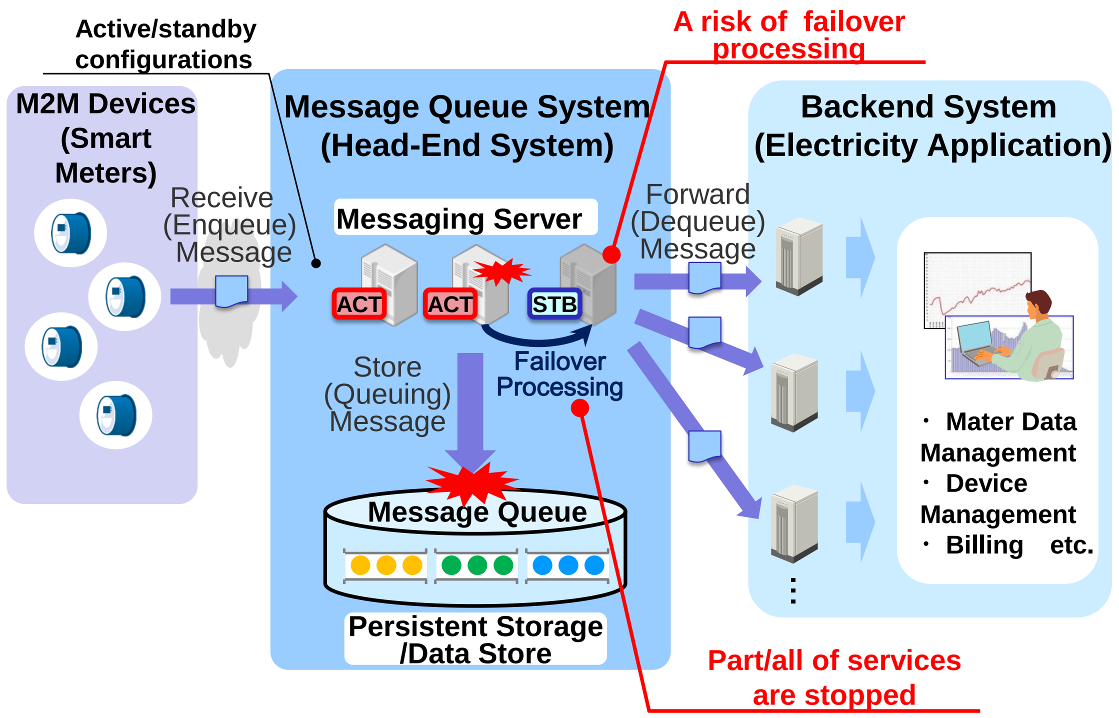

Active/standby configurations
A risk of failover processing
M2M Devices (Smart Meters)
Message Queue System (Head-End System)
Backend System (Electricity Application)
Receive (Enqueue) Message
Messaging Server
ACT
ACT
STB
Forward (Dequeue) Message
Store (Queuing) Message
Failover Processing
Message Queue
Persistent Storage /Data Store
・ Mater Data Management
・ Device Management
・ Billing etc.
Part/all of services are stopped

Figure 3.1 Example of message queue system for M2M

(b) The distribution method of messages to servers using round-robin with a slowdown KVS exclusion and two logical counter-rotating KVS rings can achieve short-term availability even during an underlying network failure and/or slowdown of servers.

This chapter is organized as follows. First, we explain the research background and issues related to message queue systems. We then present the system architecture and design. Next, we show the implementation and the performance evaluation results on availability of the system. Finally, we describe related work and conclusion.

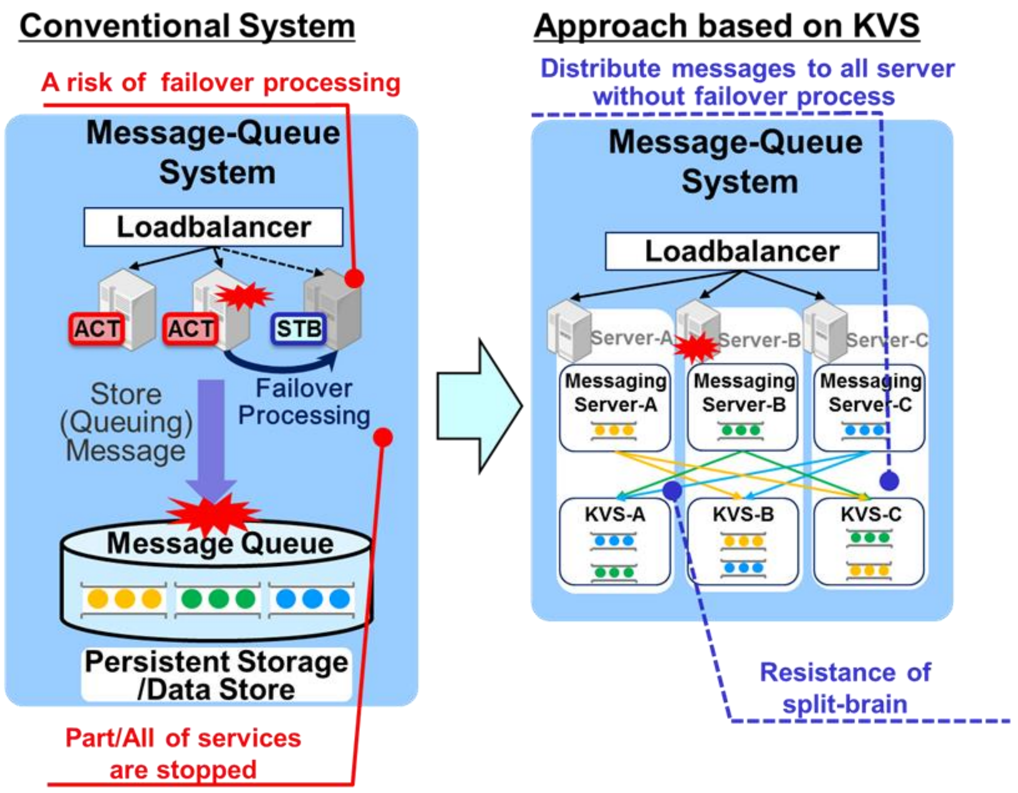


Figure 3.2 Risk of failover processing in conventional systems and our approach to achieve high-availability

3.2. Background and Issues

3.2.1. Outline of Message Queue System for M2M Devices

Figure 3.1 outlines an example of the system structure for services of M2M devices. Message queue systems are widely used for a large variety of services such as smart meter services in an electric power company, health equipment monitoring services, etc. Main functions of message queue systems are to reliably relay messages to the backend system without message loss and to buffer the message traffic of devices.

3.2.2. Risk of Failover Processing

Figure 3.2 outlines an example of failover processing and our approach to achieve high availability. Mission-critical systems usually have HA clusters for continuous service when their components fail. HA clusters detect hardware/software failures and immediately restart the application on another standby system, which is referred to as *failover*.

Conventional message queue systems have a risk of failover processing. Similarly, our previously proposed system in Chapter 2 also partly has this risk because it uses recovery processing in which another system (messaging server) on standby gets all messages stored before the failure of the KVS.

To solve this issue caused by failover processing in the message queue system, we take advantage of distributed in-memory KVS. Generally, distributed in-memory KVS is used for high-throughput and scalability. However, we use it here for improving availability of the message queue system. To remove failover processing, we follow the approach of fabric messaging that distributes messages to all servers during the normal processing state. A *fabric* is a topology in which nodes pass data to each other through interconnected nodes in a mesh fashion. In data center network research, switch fabrics are well known [70, 71]. In this chapter, we propose a *fabric architecture* on the application layer containing the data store for solving the above-mentioned failover processing issue of message queue systems (see Sect. 3.3.1).

3.2.3. Trade-off between Consistency and Availability

3.2.3.1. Queues on Distributed KVS Ring

For scalability of the data store, a general distributed KVS distributes data as (key, value) pair by consistent hashing [72] and a cluster of distributed KVS is configured by using *range partitioning* [72, 73] (the cluster of distributed KVS is denoted as KVS ring). In the KVS ring, each server (*coordinator* in [72]) is responsible for the region between itself and the previous server on the ring.

Our previously proposed message queue system in Chapter 2 simply applied basic KVS technology, therefore, the consistency of messages cannot be maintained when split-brain occurs as shown in Fig. 3.2. To maintain strong consistency, we use Paxos, a protocol for obtaining consensus in interconnected unreliable processors, which is widely used in many distributed processing systems [73].

However, even if both general distributed KVS technology and Paxos were simply applied to the message queue system as shown in Fig. 3.3, there would be new problems that are described in Sect. 3.2.3.2.

In Fig. 3.3, each KVS is assigned queues based on range partitioning. Each queue is stored in three KVS and can be in either master or non-master state. The master

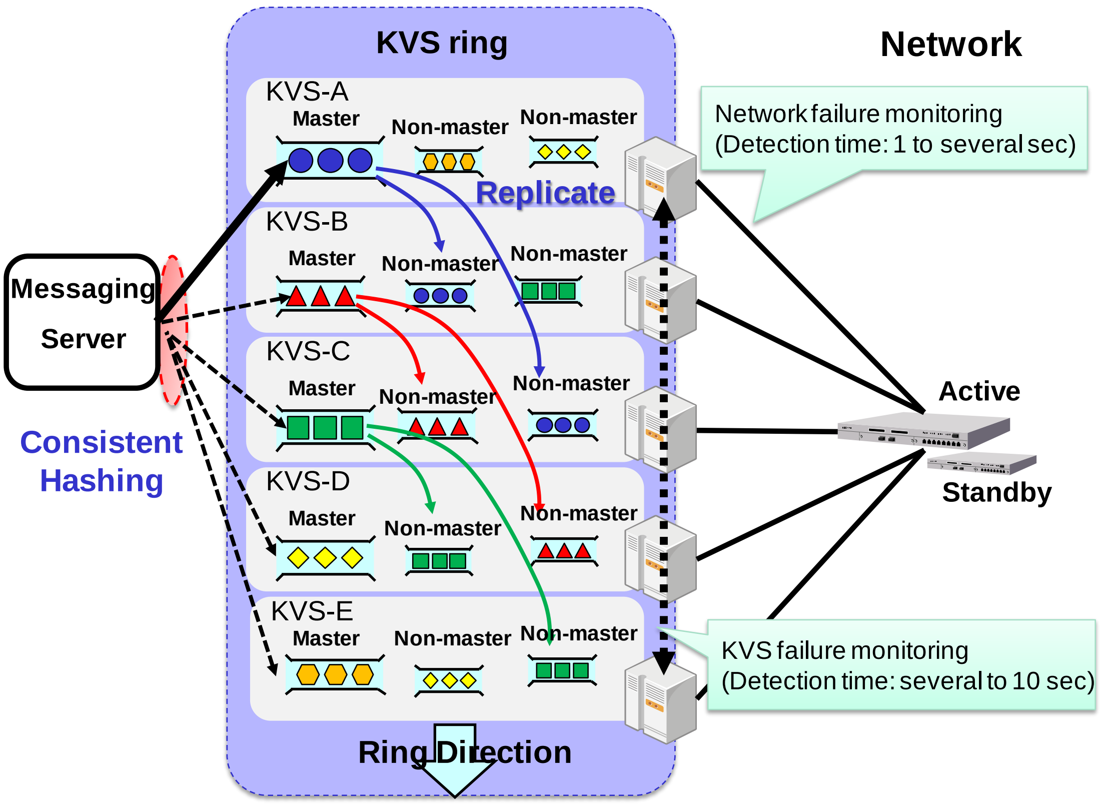


Figure 3.3 Applying conventional method in real system

queue is responsible for queue transactions, such as enqueuing or dequeuing of messages, and for message replication of the two non-master queues. If a KVS failure is detected in a KVS ring by Paxos, the faulty KVS is isolated from the ring and one of the non-master queues becomes the new master queue as alternative to the previous master queue on the faulty KVS. The messaging server selects the KVS with master queue by using consistent hashing and sends messages to this newly selected KVS.

3.2.3.2. Two Problems in Message Queue Systems

Figure 3.4 outlines the new problems that arise when applying conventional methods to this message queue system. When constructing the message queue system, as shown in Fig. 3.3, the KVS are connected to the underlying network, which consists of more than a single network device (each device has its own standby device in case of a failure). Considering a route change between switches in case of failure at a single or multiple switches, the route stabilization time takes several seconds or more than 10 seconds in either case.

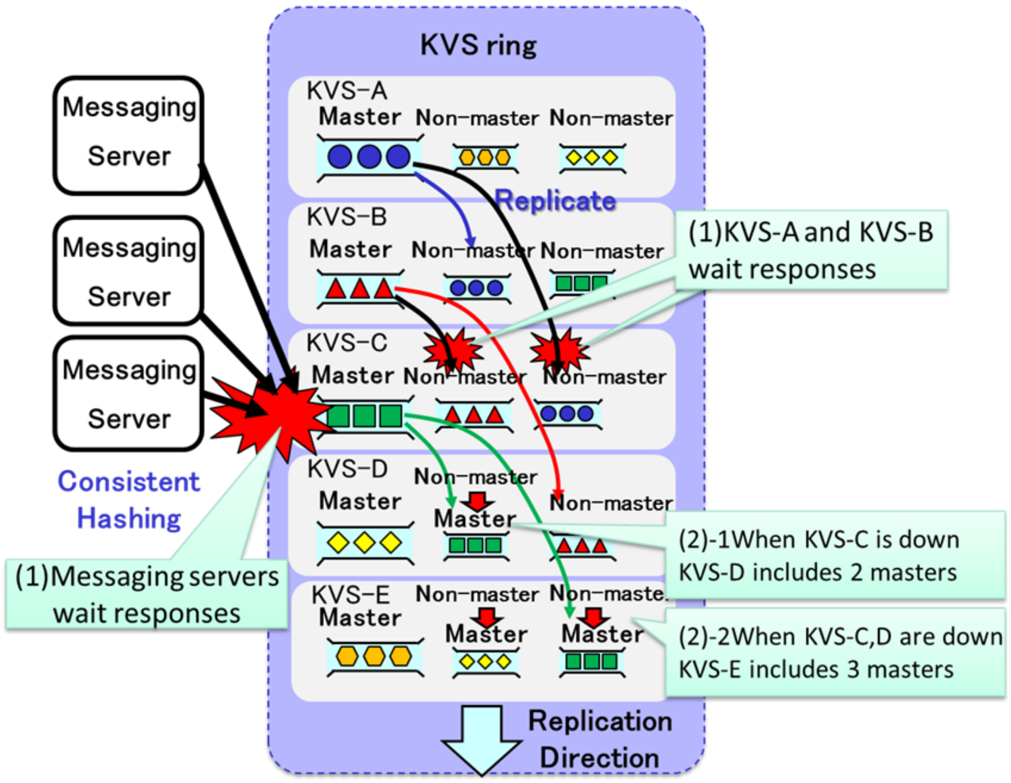

KVS ring
Messaging Server
Messaging Server
Messaging Server
Consistent Hashing
(1)Messaging servers wait responses
KVS-A
Master Non-master Non-master
Replicate
KVS-B
Master Non-master Non-master
(1)KVS-A and KVS-B wait responses
KVS-C
Master Non-master Non-master
KVS-D
Master Non-master Master Non-master
(2)-1When KVS-C is down KVS-D includes 2 masters
KVS-E
Master Non-master Master Non-master Master
(2)-2When KVS-C,D are down KVS-E includes 3 masters
Replication Direction

Figure 3.4 Problems of applying conventional methods.

The temporal performance degradation of a server is another problem. As shown in Fig. 3.4, when the server of KVS-C fails, the performance of KVS-A and KVS-B having the responsibility for non-master queues degrades due to multiple reply-timeouts of KVS-C until the detection and isolation of KVS-C's failure; Typically, more than 10 seconds are needed for a server failure detection, see Fig. 3.3. Furthermore, a heavy workload background job also temporally degrades server performance. This is denoted as *slowdown* of a server. Above-mentioned examples may lead to the following two problems of short-term availability in the message queue system as shown in Fig. 3.4.

(1) Large values of KVS failure detection timer

To avoid false detections of server-failures during the route stabilization of the underlying network, the detection time for KVS failures must be set to a value that is larger than the route stabilization time of the network, i.e., from several seconds to above 10 seconds. Consequently, all messaging servers must wait for the response from the faulty KVS until the KVS failure detection timer expires. The same problem also occurs for a slowdown. One example of this adverse effect is that the messaging server cannot reply to the mobile devices for over 10 seconds, while wasting wireless resources and degrading messaging service quality, see Fig. 3.4-(1).

(2) Concentration of message queue load after KVS failures

When KVS-C in Fig. 3.4 fails, KVS-D becomes the new master after the KVS failure detection time has passed and the non-master queue is designated as the new master queue as described in Sect. 3.2.3.1. The designation order of the new master queue depends on the KVS ring's direction. For example, if both KVS-C and KVS-D fail, KVS-E is designated as the new master, therefore, it must process three master queues of all three KVS (C+D+E). In this situation, the load of message traffic concentrates on KVS-E, which can lead to performance degradation (see Fig. 3.4-(2)).

The first problem described above is because the detection time for the KVS failure on the underlying network or a server slowdown can be relatively long compared to the messaging time itself (tens of milliseconds for messaging versus more than ten seconds for network stabilization or server slowdown). The second problem arises from the nature of the distributed KVS since it is important for mission-critical systems to continuously provide services even when multiple server failures occur [74, 75].

In this chapter, we propose distribution methods to solve these problems and provide short-term-availability while guaranteeing the consistency of the messages by the Paxos protocol used in the KVS.

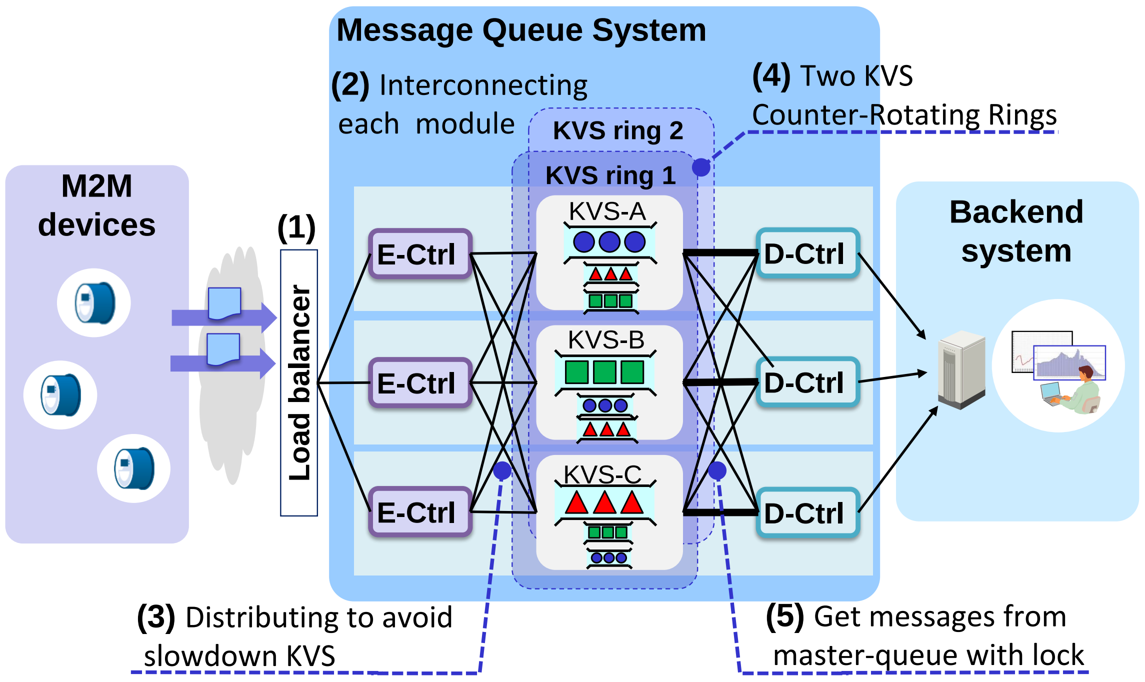

Message Queue System
(2) Interconnecting each module
(4) Two KVS Counter-Rotating Rings
KVS ring 2
KVS ring 1
M2M devices
(1)
Load balancer
E-Ctrl
E-Ctrl
E-Ctrl
KVS-A
KVS-B
KVS-C
D-Ctrl
D-Ctrl
D-Ctrl
Backend system
(3) Distributing to avoid slowdown KVS
(5) Get messages from master-queue with lock

Figure 3.5 Fabric message queue system architecture

3.3. Proposed Architecture and Distribution Methods

3.3.1. Architecture of Fabric Message Queue System

The architecture of the proposed fabric message queue system is shown in Fig. 3.5. Its logical structure and functions as well as its physical configuration are described below.

3.3.1.1. Logical Structure and Functions

As mentioned before, the messaging server consists of two programs: the *enqueue controller* (E-Ctrl), which receives messages and stores them to the queue (enqueue), and the *dequeue controller* (D-Ctrl), which retrieves messages from the queues and relays them. E-Ctrl distributes messages to all servers during the normal processing state to remove failover processing. This architecture has *fabric topology* in which nodes pass data to each other through interconnected nodes in a mesh fashion.

We describe the logical structure and function of proposed fabric message queue system where the following numbers correspond to those shown in Fig. 3.5.

(1) The load balancing module dispatches incoming messages from the source clients to the E-Ctrl in the same way as the conventional system. It monitors the TCP ports of the E-Ctrl to avoid dispatching to a faulty E-Ctrl.

(2) Multiple E-Ctrl and D-Ctrl are interconnected via multiple KVS. Both the E-Ctrl and the D-Ctrl are stateless and operate cooperatively and independently through the message queues in the KVS.

(3) The E-Ctrl selects a KVS by round-robin with a rule that excludes KVS in a faulty and/or slowdown state, then stores the message at the selected KVS (see Sect. 3.2.1). Therefore, the E-Ctrl can store messages, regardless if there are KVS failures and can continue services.

(4) The KVS on a server are logically linked to shape a directional ring that includes multiple KVS and provides distributed KVS. Message queues are deployed on the KVS ring as mentioned in Sect. 3.2.3.1. Messages and message queues are handled by the Paxos protocol as distributed KVS consisting of three KVS. Each KVS has multiple message queues. In Fig. 3.5, there are three queues, the topmost one is a master queue and the lower two are non-master queues. The E-Ctrl enqueues and the D-Ctrl dequeues messages via the master queue. Functions for high availability such as KVS failure detection, isolation of the faulty KVS, and the master/non-master KVS reassignment, are based on the basic distributed KVS described in Sect. 3.2.3. KVS can continue service such as enqueuing and dequeuing messages after failure detection regardless of which KVS has a failure (regarding availability within the failure detection, see Sect. 3.3.2.1).

(5) Multiple D-Ctrl get messages from multiple KVS and send them to the destination. Therefore, there is enough redundancy for the messaging service even if failure/slowdown of the D-Ctrl occurs. In detail, the D-Ctrl gets a message from one of the master message queues and sends it to the destination. If the message is successfully received by the destination, the D-Ctrl removes the message from the master messaging queue. The D-Ctrl sets an internal lock on the messaging queue while accessing it to arbitrate access conflicts. The D-Ctrl preferentially gets messages from a local KVS, i.e., located on the same physical server, rather than from non-local KVS to reduce processing overhead.

3.3.1.2. Physical Configuration and Features for High Availability and Scalability

The fabric message queue system consists of *N* units of load balancers and servers, as well as network devices (not shown explicitly in Fig. 3.5). All the servers have a homogeneous configuration where the E-Ctrl, distributed KVS, and the D-Ctrl are all located on one server. This configuration makes it easy to add/delete servers in this system.

KVS use Paxos for communication within their logical KVS ring for maintaining strong consistency, even in the case of network faults or split brain. We consider two KVS rings that are independent of each other in our architecture. If a server failure occurs in one KVS ring or its modules, this system can continue with the messaging service by using the other KVS ring. Both KVS rings are connected to different networks and therefore, this system can continue service even when one network becomes disconnected. Based on the proposed fabric architecture, high scalability and long-term-availability of the message queue system can be realized.

Note that regarding messages from a specific source to destination, the message delivery is guaranteed if the destination is ready to receive the message, but the order of message delivery is not necessarily guaranteed because multiple paths (KVS) between the E-Ctrl and D-Ctrl exist. We designed a fabric architecture and multiple paths to achieve higher availability. If the message queue system consists of N units of load balancers and servers having the same performance, we consider that availability is more important than message reordering for the majority of M2M services. If precise ordering is required, adding a KVS selection condition could prevent message reordering, e.g., a pair of source and destination client addresses is mapped to one specific KVS.

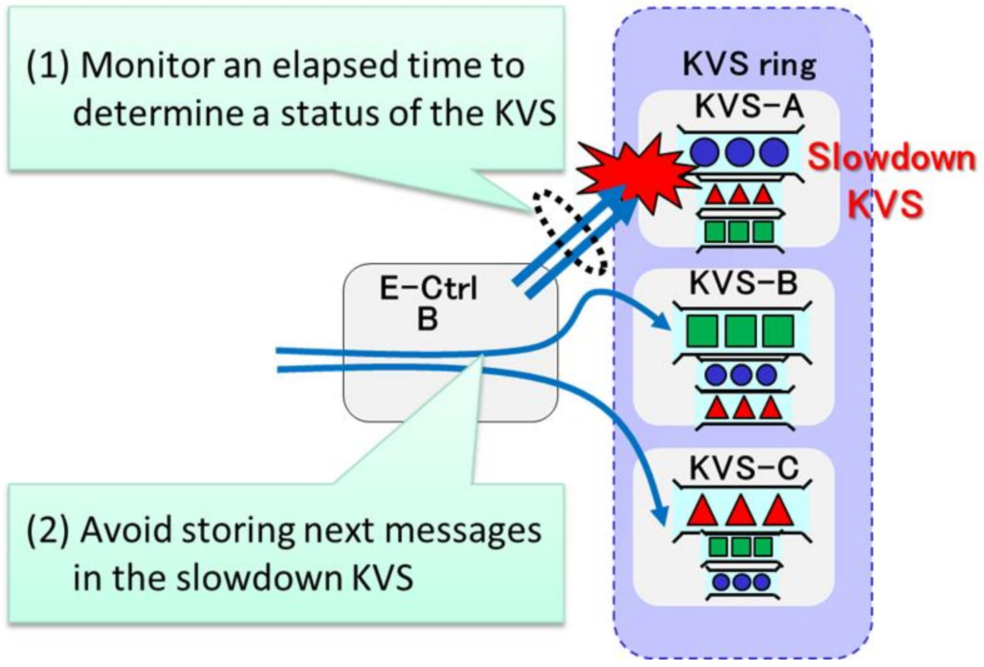

(1) Monitor an elapsed time to determine a status of the KVS
KVS ring
KVS−A
Slowdown KVS
E−Ctrl B
KVS−B
KVS−C
(2) Avoid storing next messages in the slowdown KVS

Figure 3.6 Distribution method avoiding slowdown KVS

3.3.2. Distribution Methods for Improving Short-Term Availability

3.3.2.1. Round-Robin Method with Slowdown KVS Exclusion

To solve the problem (1) described in Sect. 3.2.3.2, i.e., when the detection time for the KVS failure on the underlying network or a server slowdown is three orders of magnitude longer than the messaging service itself, we define the KVS status as being either in slowdown or no-slowdown. An E-Ctrl selects a KVS by the round-robin method with a slowdown KVS exclusion, instead of the consistent hashing method basically used in the conventional KVS.

In detail, the E-Ctrl monitors the elapsed time that starts at the time of transmitting messages to a KVS until reception by the KVS. The E-Ctrl has a threshold for the elapsed time of each KVS denoted as *slowdown detection time*. If the elapsed time exceeds the slowdown detection time, the E-Ctrl determines the KVS state as in slowdown, after which it avoids storing messages in that KVS and stores them instead on another KVS in non-slowdown state as shown in Fig. 3.6.

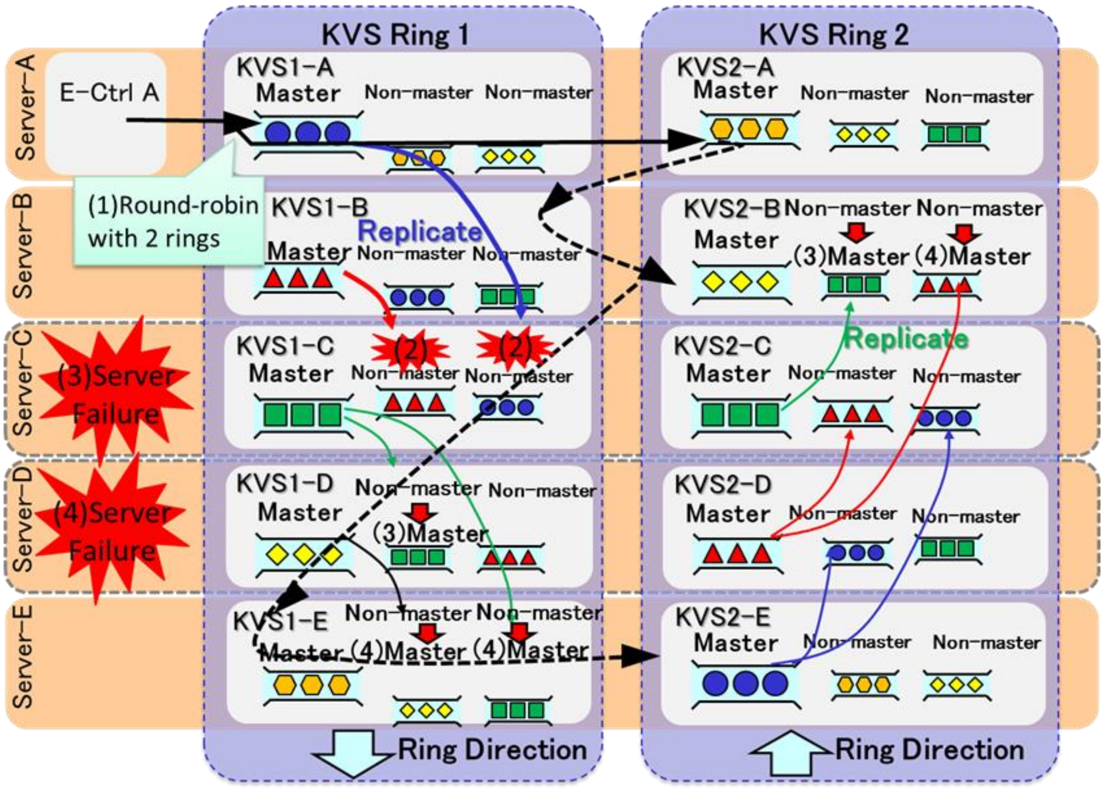

KVS Ring 1
KVS Ring 2
Server−A
Server−B
Server−C
Server−D
Server−E
E−Ctrl A
(1)Round-robin with 2 rings
KVS1−A Master Non−master Non−master
KVS1−B Master Non−master Non−master
Replicate
KVS1−C Master Non−master Non−master
(2)
(2)
(3)Server Failure
(4)Server Failure
KVS1−D Master Non−master Non−master
(3)Master
KVS1−E Master Non−master Non−master
(4)Master (4)Master
KVS2−A Master Non−master Non−master
KVS2−B Master Non−master Non−master
(3)Master (4)Master
KVS2−C Master Non−master Non−master
Replicate
KVS2−D Master Non−master Non−master
KVS2−E Master Non−master Non−master
Ring Direction
Ring Direction

Figure 3.7 Distribution method with 2 KVS rings

By monitoring each KVS with a slowdown detection time of several hundred milliseconds, we can avoid the longer detection time needed for the KVS failure or server-slowdown. An optimal value of the slowdown detection time will be evaluated in Sect. 3.4.3.

3.3.2.2. Two KVS Counter-Rotating Rings

To solve the problem of concentration of message queue load after KVS failures as described in Sect. 3.2.3.2, we propose the message distribution method with two counter-rotating KVS rings as shown in Fig. 3.7. This KVS has three queues, the leftmost is the master queue and the other two are non-master queues. Both KVS rings have opposite directions of processing order.

In normal state, an E-Ctrl distributes messages to the master queues by round-robin between both KVS rings, see (1) in Fig. 3.7. The master queues oversee the message replication for the two non-master queue. If a KVS failure/slowdown happens, it would impact the two KVS that have the master queue sending replicated messages to the non-master queues of the faulty KVS until the faulty KVS becomes isolated, see (2) in Fig. 3.7. For example, a slowdown of KVS1-C in Fig. 3.7 would influence KVS1-A and KVS1-B. At that time, if an E-Ctrl can determine KVS1-B slowdown as described in Sect. 3.3.2.1, it skips with the next message to KVS2-B in the other KVS ring, which is not influenced by the failure of KVS1-C.

If a server failure occurs, an E-Ctrl can also determine the KVS slowdown and it skips with the next message to the normal (non-failure) KVS. After that KVS detects the faulty KVS and changes one of the non-master queues to be the new master queue. For example, (3) in Fig. 3.7 shows that if a failure of Server-C occurs, the non-master queues of KVS1-D and KVS2-B become master queues. This divides the load onto both servers and is more effective when multiple server failures occur simultaneously, e.g., failures of Server-C and Server-D, see (4) in Fig. 3.7. For the conventional method that has only a single KVS ring, KVS-E must process the data of three KVS (C+D+E) when KVS-C and KVS-D fail. On the other hand, with our proposed method, KVS1-E and KVS2-B only need to process data of two KVS under the same situation. Thus, the proposed distribution method reduces the negative impacts on the service caused by server/KVS process failures.

3.4. Implementation and Evaluation

3.4.1. Implementation and Methodology for Evaluation

E-Ctrl and D-Ctrl were implemented based on an event-driven architecture [62] developed in the C language. We implemented KVS, which have a key-value data structure, in Java and added functions for queue transactions to the KVS. The message queue system for the evaluation consists of 5 E-Ctrl, 5 D-Ctrl, and 10 KVS. There are 2 logical KVS rings, each consisting of 5 KVS. Each KVS has 18 GB of memory for storing more than a million messages.

We evaluate the short-term-availability provided by proposed two methods described in Sect. 3.3.2 and the long-term-availability of the fabric message queue system described in Sect. 3.3.1. We intend to observe this behavior and evaluate availability under server failures, therefore, we assume message sizes as 30 KB, which is relatively large in our experience and it can therefore increase the server load of the messaging server/KVS. A test client program generates the workload to the E-Ctrl and the message queue system forwards the messages to a test destination server.

3.4.2. Verification of Detection of Slowdowns

To verify the effect of the round-robin method with slowdown KVS exclusion described in Sect. 3.3.2.1, we compared the throughput of two message queue systems, one with the proposed round-robin method with slowdown KVS exclusion and the other with conventional consistent hashing method. Figure 3.8 outlines the test environment of the evaluation. We let server-D fail while processing the workload and monitor the throughput and the error responses to the test client program from all the E-Ctrl. The test client program transmits the workload to E-Ctrl at a rate of 1200 msg./s that can be processed stably under one server failure in this evaluation environment.

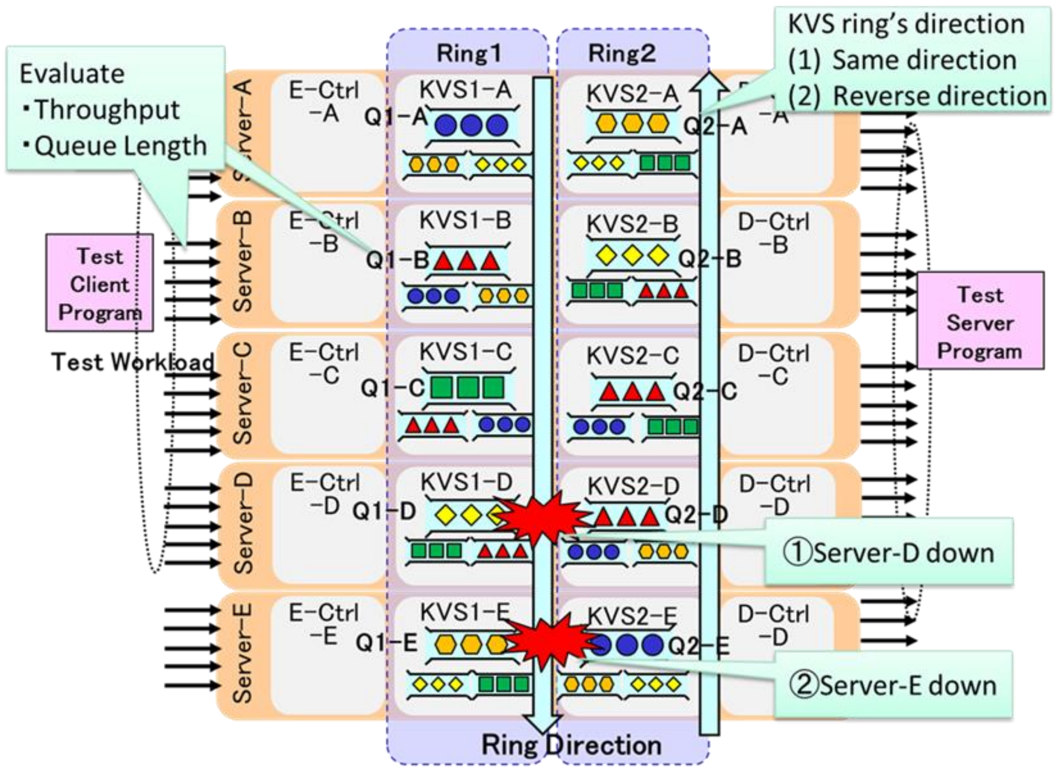


Figure 3.8 Method of evaluation of message queue system

Throughputs of the two message queue systems are shown in Fig. 3.9. For the proposed method, the throughput remains stable before and after the server failure. In contrast, for the conventional consistent hashing method, the throughput is temporally decreased for about 15 seconds after the server failure. The number of error responses is shown in Fig. 3.10. Compared to the conventional method (2379 error responses), the error responses decrease with the proposed method (214 error responses) by 92%. Thus, it is shown that the proposed method increases the short-term-availability of the message queue system.

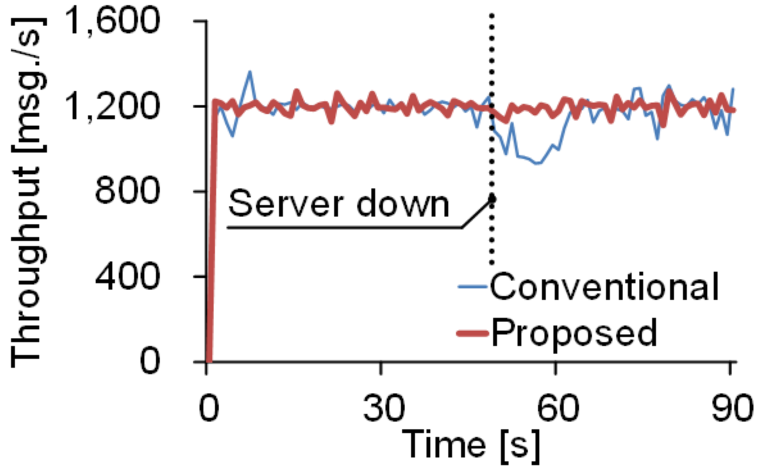


Figure 3.9 Throughput of message queue systems.

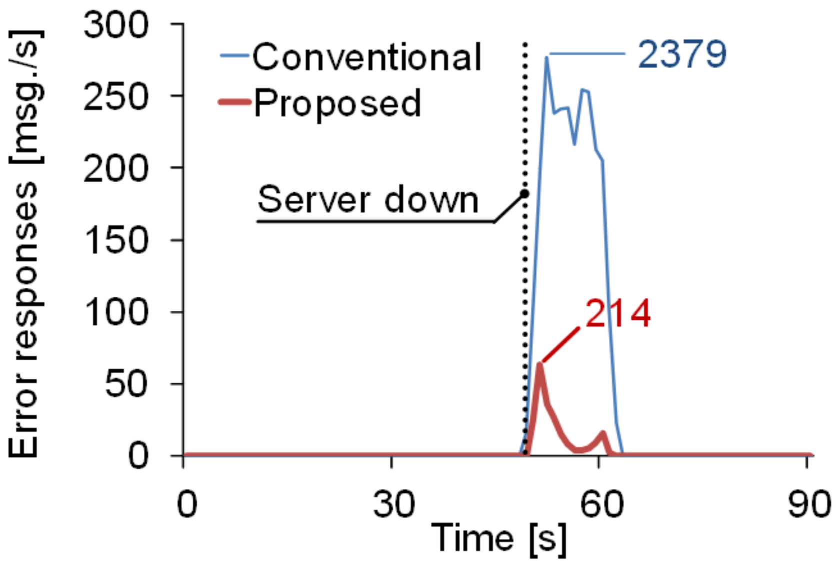


Figure 3.10 Number of error responses.

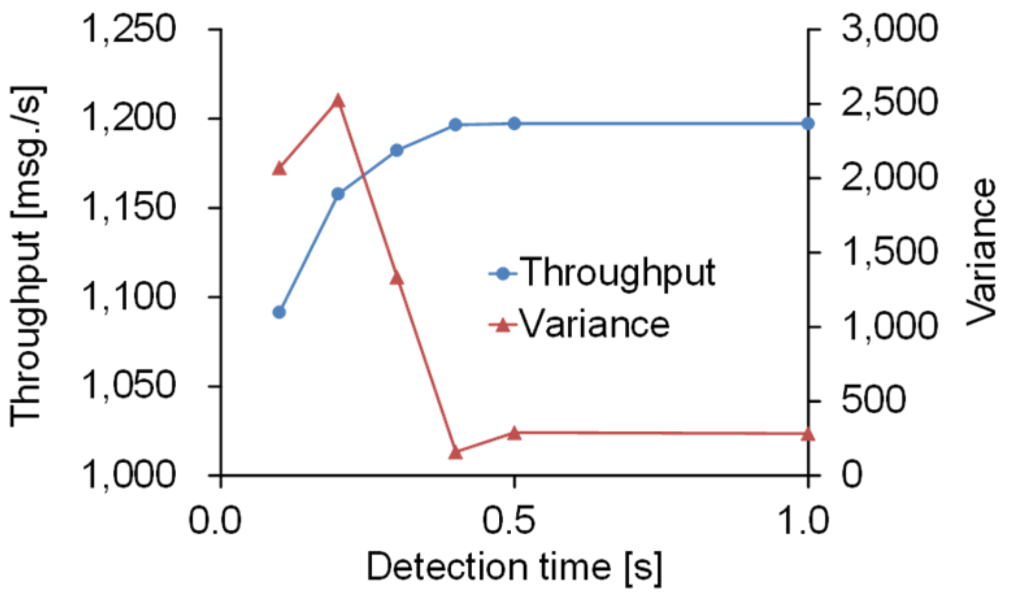


Figure 3.11 Average and variance of throughput for different slowdown detection time

3.4.3. Determining the Optimal Slowdown Detection Time

To find an optimum value of the slowdown detection time described in Sect. 3.3.2.1, we evaluate the performance of the proposed system for different parameter values. We use the same test environment as shown in Fig. 3.8 and evaluate the average and variance of the throughput for different slowdown detection time from 0.1 to 1 second.

The average and variance of the throughput are shown in Fig. 3.11. The average throughput increases in the range from 0.1 to 0.4 seconds for the slowdown detection time values, and flattens in the range larger than 0.4 seconds. On the other hand, the variance of throughput decreases in the range from 0.1 to 0.4 seconds. Figure 3.12 shows the behavior of the throughput for two slowdown detection time values, 0.1 and 0.4 seconds, before and after a server failure. The throughput for 0.1 seconds has a high fluctuation, while it is stable for 0.4 seconds. The average throughput for 0.1 seconds is 9% less than the throughput for 0.4 seconds.

In general, it is better to set smaller values for slowdown detection, because larger values impact the waiting time of the source clients (mobile devices) as described in Sect. 3.2.3.2. From the result in Fig. 3.11, an optimum value of slowdown detection time is 0.4 sec.

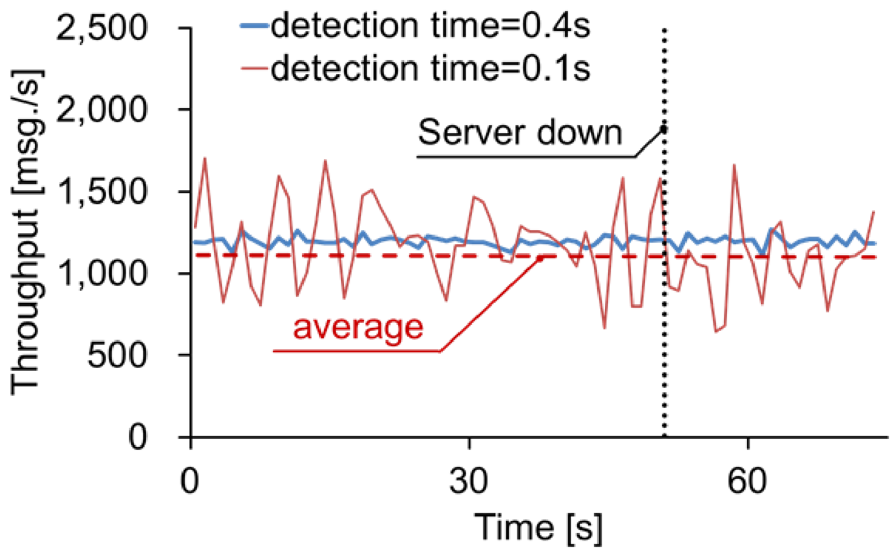


Figure 3.12 Throughput for different slowdown detection time values (0.1 and 0.4 sec) over time.

The reason why the throughput is not stable for 0.1 seconds is an effect of the copying garbage collection of Java. Copying garbage collection happened every second in the test and the process of KVS stopped operation when the detection time value is in the range from 0.1 to 0.3 seconds.

In conventional systems, the duration of copying garbage collection is negligible. Previous research on garbage collection of Java [76] revealed that the duration of copying garbage collection depends on the memory size and becomes non-negligible when the memory size is larger than 1 GB. We estimate that the duration time of copying garbage collection becomes longer, because each KVS has 18 GB memory and must store a lot of key-value data including the metadata to achieve queue transactions.

If a KVS halts due to copying garbage collection for more than slowdown detection time, the E-Ctrl stop transmitting messages to this KVS. Thus, the KVS has nothing to process, leading to a decrease in throughput of the whole message queue system. In addition, we presume this effect of copying garbage collection to be a common problem of KVS-based systems, because many KVS implementations such as Cassandra or Hbase [77] are implemented in Java and modern distributed systems are equipped with large memory.

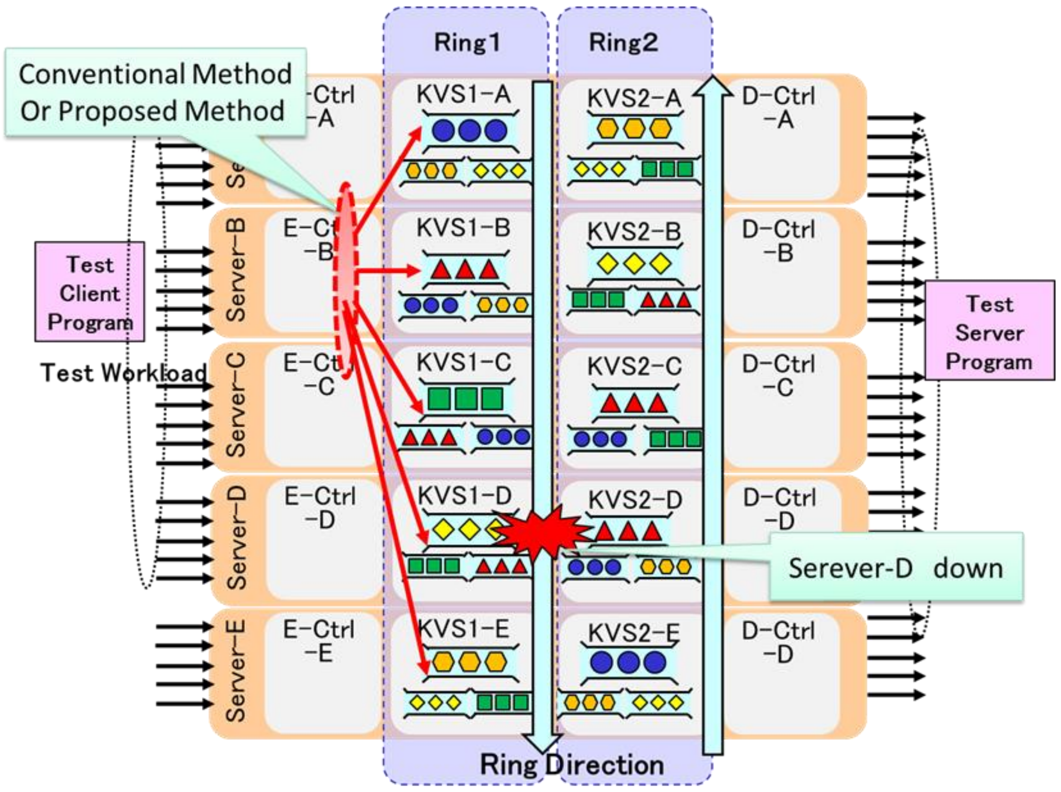


Figure 3.13 Performance evaluation of the proposed message queue system

3.4.4. Impact of Server Failures on Availability

We evaluate the performance of the proposed fabric message queue system from the long-term-availability point of view. Figure 3.13 outlines the test environment of the evaluation. We compare the performance behavior, throughput, and queue length of the message queue of the proposed system and the conventional system. The conventional system has two KVS same-direction rings because that is same as having a single KVS ring with one direction. The queue length of the message queue reflects the variance of load balancing in the whole system. We apply the optimum value of 0.4 seconds as the slowdown detection time to the system.

For mission-critical systems, e.g., carrier grade systems, operators expect the system to handle 2 simultaneous server failures and they construct the redundancy system for this worst-case scenario. A single server can stably process 300 msg./s as shown in Sect. 3.4.2, therefore, the test client program transmits the workload to the E-Ctrl at a rate of 900 msg./s that can be processed stably when 2 server failures occur. After 60 seconds of transmitting the workload, we first let server-D fail, followed by a

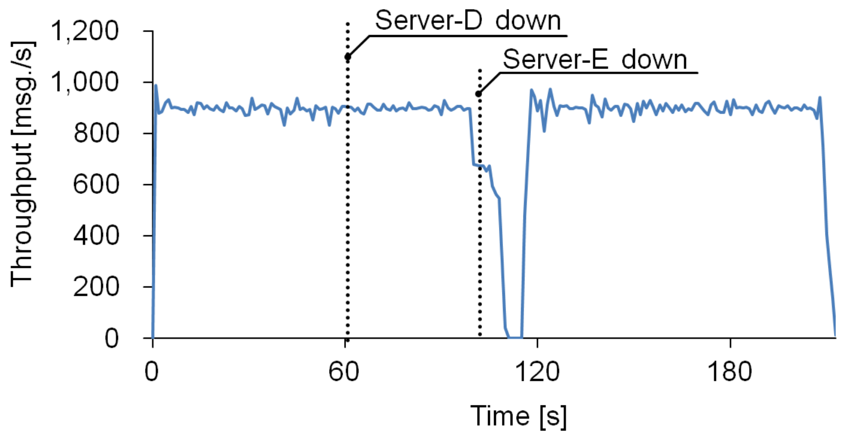


Figure 3.14 Throughput of message queue system (conventional method: same direction of both rings).

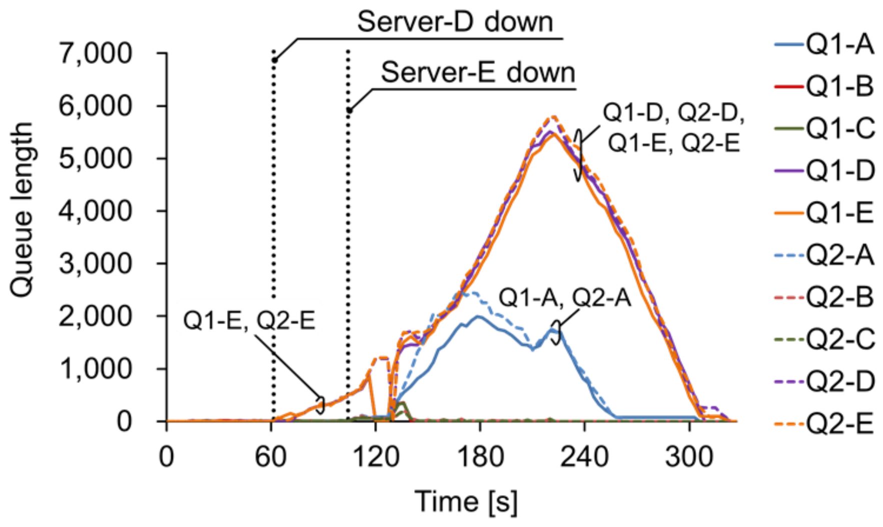


Figure 3.15 Queue lengths of each KVS (conventional method: same direction of rings).

failure of server-E.

(1) Conventional Message queue system

Throughput of the conventional message queue system is shown in Fig. 3.14. Throughput is stable when the first server failure happens, however, when the second server fails, it decreases to zero, meaning that the messaging service completely stops. After 5 seconds of stopping the messaging service, the service is recovered.

The queue length of the message queue in each KVS is shown in Fig. 3.15. For example, the queue name “Q1-A” shows the queue of KVS belonging to ring 1 and initially located in Server-A. When the Server-D failure occurs, the lengths of Q1-E and Q2-E increase. We consider that it is caused by excluding KVS-D. When the server-E failure occurs, the lengths of 6 queues (Q1-A，Q2-A, Q1-D，Q2-D，Q1-E，Q2-E) are increased. We attribute this to the problem in load balancing as described in Sect. 3.2.3.2(2). Server-A that includes KVS1-A, KVS1-B, and D-Ctrl-A must process the queues of 3 KVS and the failure of Server-E impacts the other KVS during the failure detection time (described in Sect. 3.3.2.2) leading to a messaging service stop for 5 seconds.

(2) Proposed Message Queue System

The throughput of the proposed message queue system having the two KVS counter-rotating rings is shown in Fig. 3.16. Compared to the conventional system in Fig. 3.14, throughput in Fig. 3.16 remains rather stable when the first and second servers fail. Throughput decreases about 20%, which corresponds to the workload of two messaging (master) queues temporally in an out-of-service state out of the initial five messaging (master) queues.

The queue lengths in each KVS are shown in Fig. 3.17 for the proposed system. Compared to the conventional system in Fig. 3.15, the lengths of queues in Fig. 3.17 are only slightly increased. That is caused by the proposed method reducing the impact of load balancing as described in Sect. 3.2.2. Thus, it is shown that the proposed architecture and two methods can continue the messaging service even if multiple server failures occur. Therefore, it can provide long-term-availability.

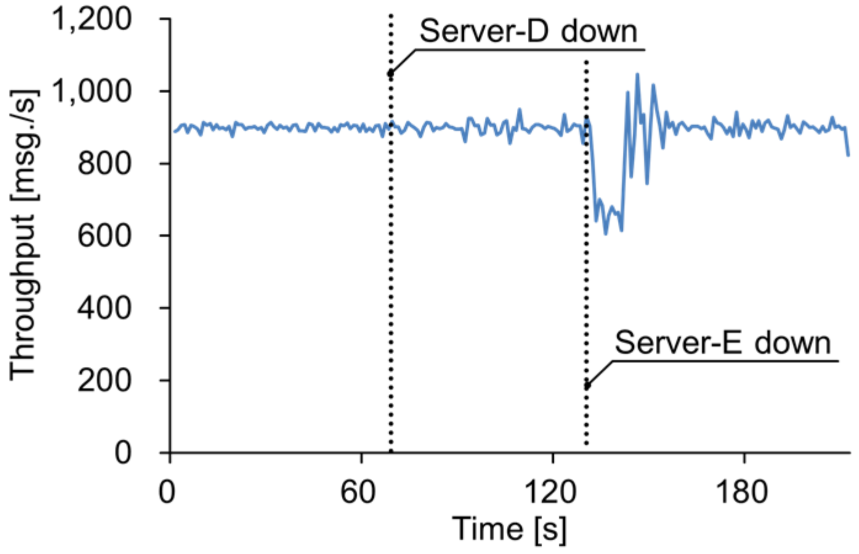


Figure 3.16 Throughput of message queue system (proposed method: opposite directions of rings).

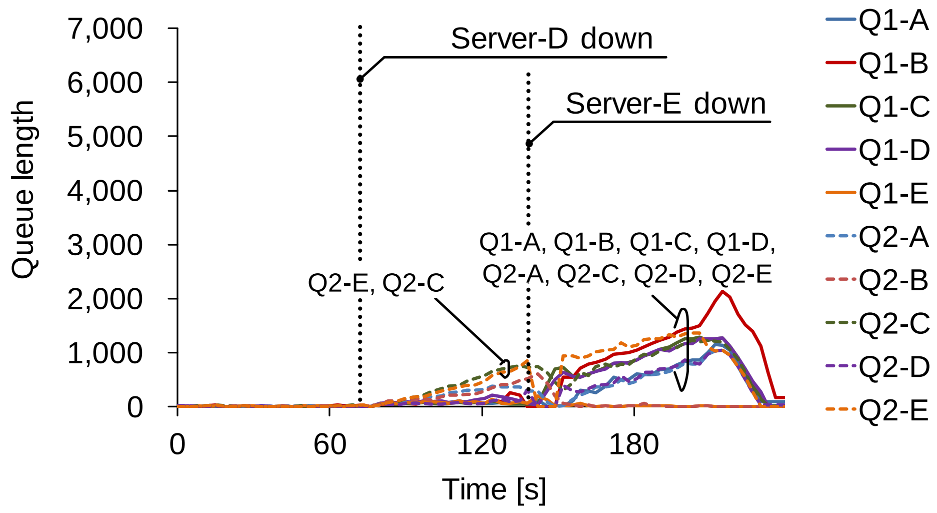


Figure 3.17 Queue lengths of each KVS

3.5. Related Work

We describe related work from three points of view: message queue systems, failover processing, and distribution methods.

Regarding message queue systems, a queuing system based on distributed in-memory KVS was proposed in [57]. Its queuing function deployed in the KVS is similar to the function of our proposed system. However, it focused on the queuing part only and it did not discuss about the availability of the whole system including the messaging process. In addition, our approach of focusing on availability of distributed in-memory KVS differs from conventional research.

Previous study in the risks of failover processing have been reported in [78, 79]. To avoid catastrophic service failures, they proposed management rules, e.g., monitoring system failures and verifying configurations of failover processing, and preparations, e.g., procedures when system failure happens. It was also mentioned in [78] that designing the system for a concentration of message load after failover processing was important.

The *shared nothing* architecture [79] is similar to ours when distributing messages in a normal processing state. However, the *shared nothing* system usually doesn't duplicate messages and needs failover processing or recovery processes to continue service. The significant difference between our proposed architecture and the *shared nothing* architecture is when it is executed. Our proposed system always executes the same process wherever a server failure happens, while application restart and recovery process in the *shared nothing* architecture are executed only when a server failure happens. Therefore, our proposed architecture can be more available than the *shared nothing* architecture.

Regarding the distribution method of KVS, consistent hashing is the standard distribution method of KVS such as in Cassandra. Our proposed method is optimized for queuing and high availability in the messaging service.

3.6. Conclusion

In this chapter, we proposed a fabric message queue system that has the following functions and advantages.

・ The message queue system architecture based on distributed in-memory KVS can provide long-term-availability and can continue its service wherever in the message queue system server/process failures may occur by distributing messages to multiple servers. Furthermore, it can guarantee strong consistency of the messages and message queues by using KVS functions and the Paxos protocol.

・ The distribution methods of messages to servers by using round-robin with a slowdown KVS exclusion and two logical KVS counter-rotating rings can achieve short-term-availability even during an underlying network failure and/or a slowdown of servers.

Evaluation results show that this system can continue service without failover processing. Compared with the conventional method, our proposed distribution methods reduced 92% of user errors caused by server failures. Furthermore, we determined the optimum value of slowdown detection time in our distribution method.

Chapter 4

Increased Throughput of Message Queue System through Dequeue Scheduling

4.1. Introduction

In the *Internet of Things* (IoT) era, the amount of all digital data in the world created by various devices and sensors is exponentially increasing and it is predicted to reach 40 ZB by 2020 [8]. IoT service systems utilizing data from devices typically consist of 3 groups: *field devices* which send and receive data, *backend systems* in a data center/cloud, and the *message queue systems* located between the devices and backend systems.

Message queue systems are widely used for interoperability and control of the huge message traffic between devices and backend systems [81, 82]. Especially, the control of message traffic has become an important requirement as the volume of IoT messages has increased dramatically over the past years. There are several solutions such as Kafka [48], Amazon Kinesis [83], Azure IoT Hub [84], etc., following different approaches depending on their respective objectives. In addition, to satisfy these requirements as well as obtaining a high availability, such as a short failover time of within one second for social infrastructure systems, we proposed in the previous chapters a high-throughput and reliable message queue system based on a distributed in-memory key-value store (KVS).

Here, we address another issue of traffic control between devices and backend systems for IoT services. Devices transmit messages periodically at their own intervals, such as the period of log collection for their service requirements. On the other hand, backend systems process messages at different rates to achieve maximum throughput for the individual objectives of the IoT services, such as analysis, management of devices, or data visualization. Therefore, to compensate for the heterogeneity in message traffic between devices and backend systems, message queue systems use buffering to handle message traffic from devices. This compensation is achieved through distributed message queue systems, which enables the distribution and load-balancing of message processing on multiple servers. In the past, the specifications of field devices and backend systems were defined in advance. However, today's IoT service systems in conjunction with *development and operations* (DevOps) trends require rapid implementation and continuous modification, additionally to the backend system also becoming adaptable [85-88]. Furthermore, progresses of distribution platforms such as Spark [49] or Storm [50], have dramatically improved the performance of backend system. In this background, updating the processes or parameter settings of backend system can impact the system's performance.

In fact, when the number of backend systems connecting to the message queue increases, we can observe that this situation impacts the performance of our proposed message queue and degrades the throughput for retrieving messages from the message queue (*dequeue*). By analyzing the factor of throughput degradation, we recognize a large number of *missed-dequeue*s, which means that the lack of messages in the selected queue wastes computational resources.

Therefore, in this chapter, we focus on the dequeue process of distributed message queue systems and we propose a method called *Retry Dequeue-request Scheduling* (RDS) to solve the throughput degradation problem. We evaluate the RDS method by simulation and also prove its advantage in experimental real servers.

This chapter is organized as follows. First, we explain the background and issues of message queue systems for IoT. We then present our proposed method and its design. Next, we show its performance evaluation by simulation, and the results from the experimental evaluation. Finally, we describe related work and provide a conclusion.

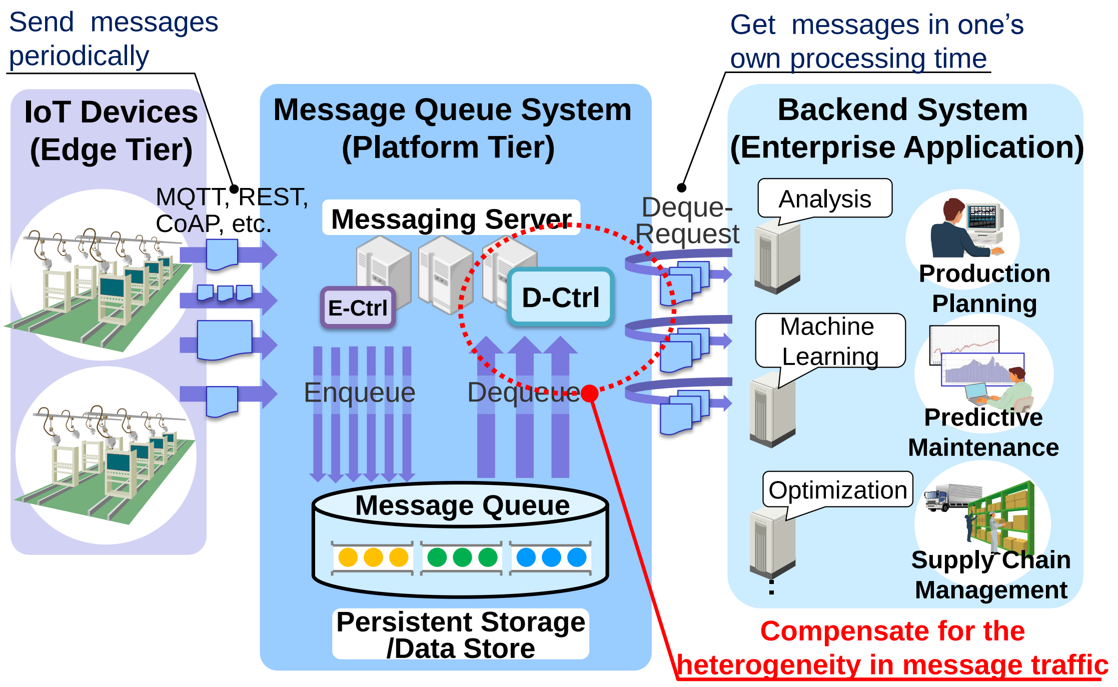

Send messages periodically
Get messages in one's own processing time
IoT Devices (Edge Tier)
Message Queue System (Platform Tier)
Backend System (Enterprise Application)
MQTT, REST, CoAP, etc.
Messaging Server
E-Ctrl
D-Ctrl
Deque-Request
Enqueue
Dequeue
Message Queue
Persistent Storage /Data Store
Analysis
Production Planning
Machine Learning
Predictive Maintenance
Optimization
Supply Chain Management
Compensate for the heterogeneity in message traffic

Figure 4.1 Structure of IoT service system

4.2. Background

4.2.1. Outline of IoT Service System

4.2.1.1. Message Queue System in IoT Service

Figure 4.1 outlines an example of the system structure in IoT services. Message queues are widely used for a large variety of services, e.g., monitoring/optimization of services in industry, smart meter services in electricity companies, connected vehicle services, or services of a telecom company collecting data from M2M devices. Message queues are required for the interoperability and abstraction (*absorption*) of message traffic of devices. By supporting IoT protocols, e.g., *MQ Telemetry Transport* (MQTT), *Representational State Transfer* (REST), or *Constrained Application Protocol* (CoAP) [89], and by making devices and backend system become loosely coupled (*independent*), message queues enable the developer to interoperate between them rapidly. Message queues buffer messages into a queue on a persistent storage (*enqueue*) and enable the backend system to retrieve the messages from the queue at their own timing. Under the condition that the message queue has both, sufficient performance to process messaging traffic from devices and scalability in performance and storage, the message queue enables the developers of the backend system to design their system without considering the entire volume of the messaging traffic.

4.2.1.2. Heterogeneity in IoT Message Traffic

It is generally agreed that IoT services require information from historical or real-time data for their own objectives, such as monitoring and optimization [43-47]. In [43-45], IoT service systems are required to manage the massive volume of data generated by sensors in various fields, such as smart grids, connected vehicles, and heavy equipment, etc. In [46], optimization at the enterprise level in smart manufacturing requires only periodically collected data. In [47], general smart sensors are organized in simple packages, i.e., they may consist of single chips and generate simple periodical data.

The general approach in IoT to find patterns in data is to collect much data from devices and learn through trial-and-error of data analysis. This approach requires a large data volume for various analyses. Therefore, traffic volume from devices generating periodical message data has become enormous in IoT service systems. The transmitted data size of sensors highly depends on their service requirements and protocols, such as MQTT, REST, and Transport Layer Security (TLS) [90], etc. From our past experiences with specific use cases, such as monitoring or optimization services, we assume in this chapter that the data size is 1 KB, which is widely applied to IoT services.

On the other hand, the backend system collects data for various IoT objectives, such as monitoring and optimization, for which it retrieves messages from the queue at its own non-periodic and process-dependent timing. The processing times differ by context of message, message size, and other related data. To achieve higher throughput by fully utilizing computational resources, the backend system retrieves messages from the queue with a pull-based method [48]. We describe further details in Sect. 4.6. In addition, progress in distribution platforms, such as Spark, leads to a dramatic change in processing time of the backend system.

Here it can be seen that while devices send massive amounts of periodical messages, backend systems process messages at their own timing in IoT. Therefore, the control function of the massive and heterogeneous message traffic in the message queue becomes a crucial issue in IoT. In this chapter, we are targeting these heterogeneous environments in the IoT service system.

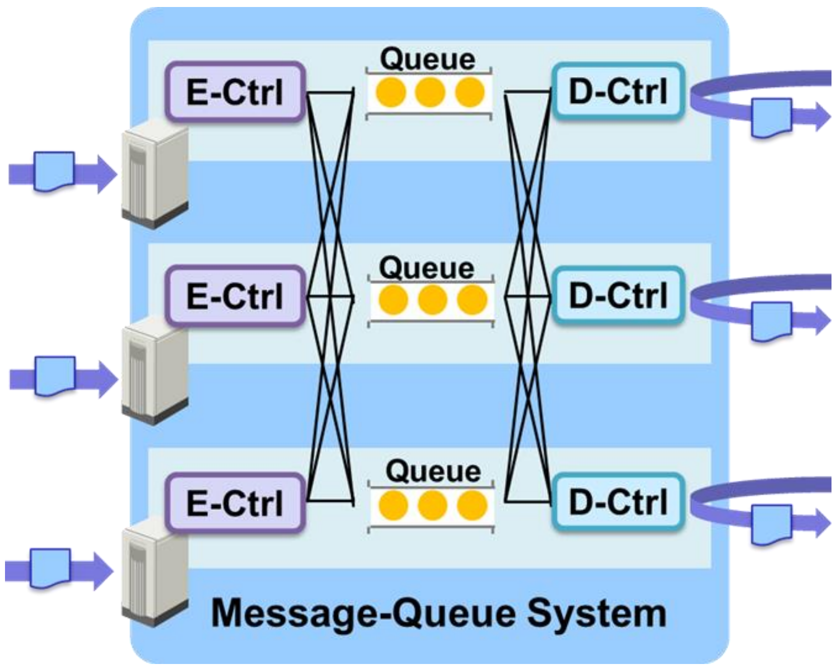


Figure 4.2 Overview of distributed message queue system. E-Ctrl and D-Ctrl denote enqueue controller and dequeue controller, respectively

4.2.2. Conventional Approach using Distributed Message Queues

4.2.2.1. Architecture for High Scalability and Availability

In Chapter 3, we proposed a high-throughput and reliable message queue system based on a distributed in-memory key-value store (KVS) for social infrastructure systems (Fig. 4.2). The proposed message queue system adopts a fabric architecture with connected full-meshed servers for high scalability and availability. The proposed message queue system consists of 3 parts: the *enqueue controller* (E-Ctrl) for receiving and storing messages in a queue, the *distributed queue* to the KVS server as persistent storage, and the *dequeue controller* (D-Ctrl) for receiving dequeue requests from the backend system and retrieving the messages from queues. This structure enables to eliminate a single point of failure and enhances the horizontal scalability of each part.

4.2.2.2. Transparency in Distributed Message Queues

In the proposed queue system as shown in Fig. 4.3, E-Ctrl and D-Ctrl provide access transparency and location transparency for devices and backend system. Let us detail

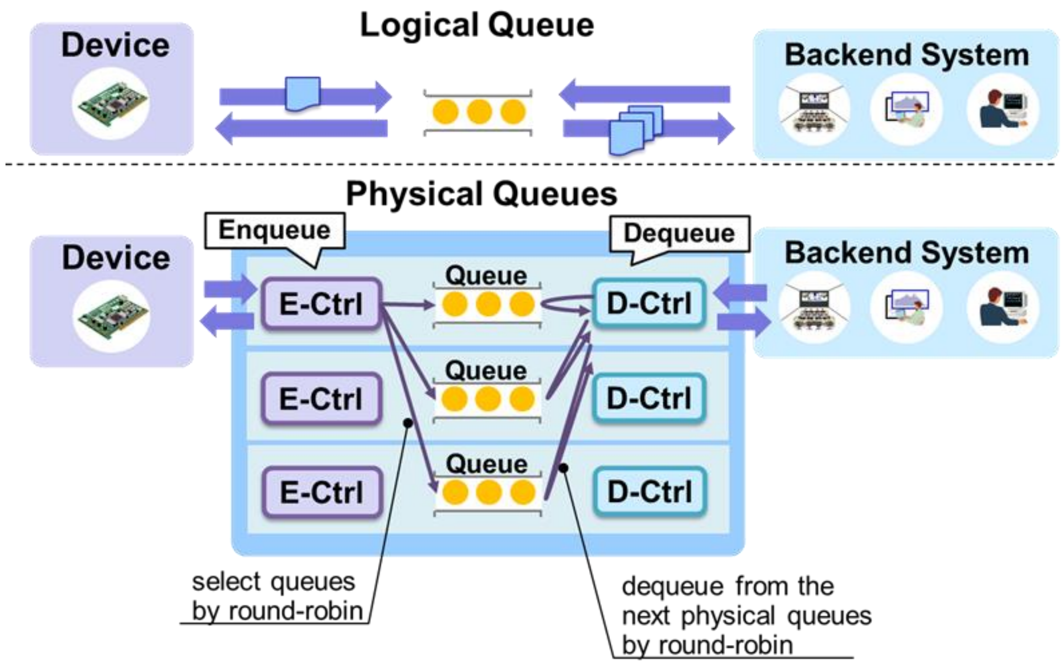


Figure 4.3 Transparency in distributed message queues

their transparency using Fig. 4.3. In the message queue system, a logical queue consists of multiple physical queues based on KVS. E-Ctrl and D-Ctrl share information of the logical queue, such as the location of physical queues, and enable devices/backend system to access logical queues as a single queue. When devices enqueue a new message into the logical queue, the E-Ctrl selects one of the physical queues by round-robin and physically enqueues it there.

On the other hand, when the backend system dequeues messages from the logical queue, the D-Ctrl searches for messages by round-robin in multiple message queues and dequeues them from those. The backend system can require how many messages are retrieved by a single dequeue-request and the D-Ctrl can dequeue messages from multiple queues. If the backend system requires the maximum number of messages and we define this number as N_{max}, there are two types of D-Ctrl dequeue procedures: (i) retrieving N_{max} messages or (ii) retrieving a number less than N_{max} messages from one of the physical queues. When the D-Ctrl gets N_{max} messages, it sends these messages to the backend system. On the other hand, when the D-Ctrl gets less than N_{max} messages from one physical queue, it continues with dequeuing from the next physical queues by round-robin until it has N_{max} messages in total or the counter for dequeue trials exceeds the setting of dequeue trials (retry out). Each D-Ctrl performs dequeues in parallel. This distribution of dequeue accesses enables the backend system to get the messages without considering the location where they were actually stored.

4.2.3. Outline of IoT Service System

As mentioned above, backend systems are required for rapid implementation and continuous modification due to DevOps trends in IoT services. Developers modify backend system parameters or data processing methods to adjust for variable requirements or objectives of the IoT service. For example, an interval of dequeue requests is required by the data processing time of backend systems for achieving IoT service requirements. The developers also determine the number of backend systems to ensure sufficient throughput.

However, as result of the real-world performance test in the case where a large number of backend systems is connected to our proposed message queue, the throughput is degraded by 20% from the expected message traffic volume. The reason for the degradation of throughput is that a large number of dequeue requests wastes computational resources of the message queue system. Especially missed-dequeues, which occur when there is a lack of messages in the selected queue, consume the computational resources for enqueue and dequeue operations (see Sect. 4.3.1).

We focus on the enqueue traffic in the conventional approach and extend the system based on the enqueue traffic. However, dequeuing (D-Ctrl) can become the bottleneck of the IoT service system in the above case. For IoT services, it is a fundamental issue for backend systems to modify data processing continually without the need for parameter tuning. To solve this issue, we propose novel dequeue methods in the distributed message queue in this chapter.

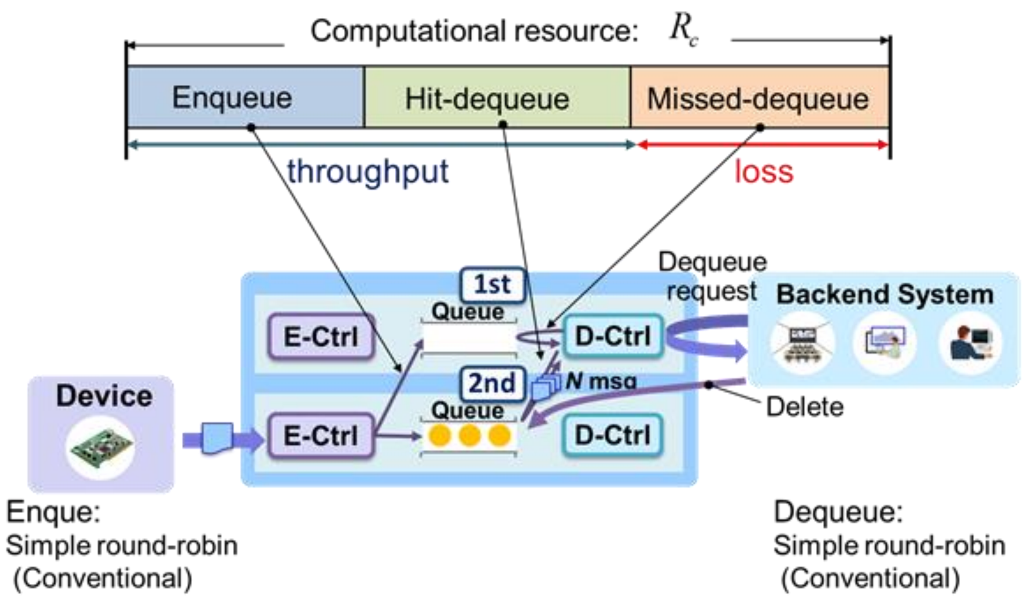


Figure 4.4 Process of distributed message queue

4.3. Analysis of Throughput Degradation and Proposal

In this section, we first analyze the processing of the message queue to solve the problem of throughput degradation. Next, we analyze the problem of throughput degradation based on computational resources. Finally, we propose two new dequeuing methods that decrease the number of dequeue requests from the backend system.

4.3.1. Process of Distributed Message Queue

Figure 4.4 shows a simplified view of each process of the message queue. There are three kinds of processes: *enqueue*, *dequeue*, and *delete*. Furthermore, we distinguish between two kinds of dequeues: *missed-dequeue* and *hit-dequeue*. Here, *hit-dequeue* describes the successful retrieval of messages from the selected queue. Note that *hit-dequeues* always include at least one message.

When D-Ctrl receives a dequeue request from the backend system, it selects one of the message queues and sends a dequeue request. Here, backend system sets the number of maximum messages and we define this number as N_{max}. If there are no messages in the selected queue, D-Ctrl gets a negative response that we refer to as *missed-dequeue* including no messages. If there are one or more messages in the selected queue, D-Ctrl gets a positive response that we denote as *hit-dequeue* regardless of whether D-Ctrl gets N_{max} messages or not. If D-Ctrl does not get N_{max} messages in total, D-Ctrl selects another message queue by round-robin and sends the dequeue request to it. D-Ctrl continues to select another message queue until it gets N_{max} messages in total or a *retry out* occurs.

After the backend system finishes processing data, it issues a delete request to D-Ctrl. A delete process corresponds to each message in the *hit-dequeue* process. Therefore, we define the computational cost of *hit-dequeues* including delete processes simply in the following consideration.

4.3.2. Analysis of Throughput Degradation

First, we consider the computational resources of data processing in a distributed messaging queue. For calculating the maximum throughput, if we define all the computational resources of the message queue system as R_c, the total cost of the enqueue process as C_e, the total cost of the hit-dequeue process as C_{dh}, and the total cost of the *missed-dequeue* process as C_{dm}, we obtain following expression.

$$R_c = C_e + C_{dh} + C_{dm} \qquad (1)$$

This expression means that the enqueue and dequeue processing share all computational resources. If we define the *enqueue message traffic* as E [msg./s], the cost of the enqueue process per message as c_{e0}, the *missed-dequeue* message traffic as D_m [msg./s], and the cost of the *missed-dequeue* process per message as c_{dm0}, we obtain the following expression.

$$R_c = E\, c_{e0} + C_{dh} + D_m\, c_{dm0} \qquad (2)$$

In Eq. (2), C_{dh} is a variable depending on how many messages are retrieved by D-Ctrl in a single dequeue request from a selected queue. On the other hand, c_{e0} and c_{dm0} are constant because enqueue and *missed-dequeue* are processed individually.

The total cost of the *hit-dequeue* process C_{dh} can be divided into two parts: the cost of constant processing and the cost of variable processing depending on the number of messages D-Ctrl retrieves by one dequeue. If we define *hit-dequeue* message traffic as D_h [msg./s], the cost of constant processing per message as c_{dh0}, the number of messages D-Ctrl obtained by one dequeue request as N_i, and the cost of variable processing when D-Ctrl gets N_i messages by one dequeue request as c_{dhNi}, we obtain the following expression in Eq. (3).

$$C_{dh} = D_h\, c_{dh0} + \sum_{i=1}^{D_h} N_i\, c_{dhN_i} \qquad (3)$$

Since the number of input messages to a message queue equals the number of output messages, enqueue message traffic *E* equals the *hit-dequeue* message traffic D_h. Hence, we obtain the following expression in Eq. (4).

$$R_c = E\,(c_{e0} + c_{dh0}) + \sum_{i=1}^{D_h} N_i\, c_{dhN_i} + D_m c_{dm0} \quad (4)$$

In this expression, the first term represents the cost depending on enqueue message traffic. The second term is the *hit-dequeue* cost depending on both how many messages D-Ctrl gets by one dequeue request and the *hit-dequeue* process. If D-Ctrl can get messages efficiently by a single dequeue request, the second term would decrease. The third term is the cost of *missed-dequeues* and it is in proportion to *missed-dequeue* message traffic D_m, which is independent of the enqueue message traffic *E*. This term represents the loss and is independent of the input message traffic.

Here, we consider the problem of throughput degradation described in Sect. 4.2.3, where the enqueue message traffic is not changed and the dequeue message traffic is changed. Therefore, we focus on the third term and take an approach to reduce the number of *missed-dequeue* requests.

4.3.3. Proposed Methods

In this section, we propose two dequeue methods to reduce *missed-dequeue* requests to avoid throughput degradation.

4.3.3.1. Periodical Monitoring Scheduling (PMS)

Figure 4.5 outlines a dequeue method we call *Periodical Monitoring and Scheduling* (PMS). PMS aims at reducing the number of *missed-dequeues* by periodically monitoring each message queue to gather the message counter information. D-Ctrl can access a message queue, which has a sufficient number of messages (Fig. 4.5 (a)) by status monitoring. If there are no queues which have enough messages, D-Ctrl regulates

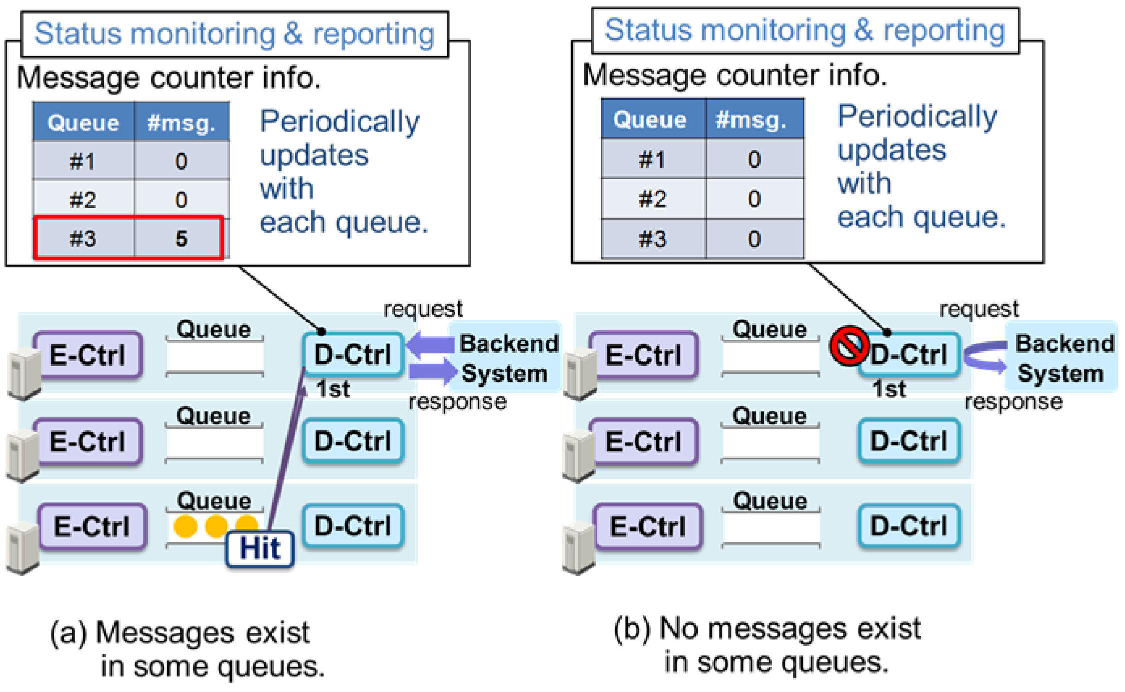

Status monitoring & reporting
Message counter info.
Queue | #msg.
#1 | 0
#2 | 0
#3 | 5
Periodically updates with each queue.
E-Ctrl
Queue
D-Ctrl
request
Backend System
1st
response
Hit
(a) Messages exist in some queues.
Status monitoring & reporting
Message counter info.
Queue | #msg.
#1 | 0
#2 | 0
#3 | 0
Periodically updates with each queue.
(b) No messages exist in some queues.

Figure 4.5 Periodical monitoring and scheduling (PMS) method

the access to the message queues (Fig. 4.5 (b)). PMS efficiently accesses the message queues to reduce the number of *missed-dequeues* trading off for the additional cost of periodical monitoring. If we define the monitoring traffic as M [msg./s] and the cost of monitoring one queue as c_M, we obtain the following expression.

$$R_c = E\,(c_{e0} + c_{dh0}) + \sum_{i=1}^{D_h} N_i\, c_{dhN_i} + D_m\, c_{dm0} + M\, c_M \quad (5)$$

In this expression, the *missed-dequeue* cost (third term) and the monitoring cost (fourth term) are in a trade-off relationship.

4.3.3.2. Retry Dequeue-Request Scheduling (RDS)

Figure 4.6 outlines a dequeue method we call *Retry Dequeue-Request Scheduling* (RDS). RDS aims at reducing the sending of dequeue requests to message queues by waiting until messages arrive at the message queues. When D-Ctrl receives a dequeue request from backend system, D-Ctrl accesses the selected message queue. If D-Ctrl cannot get any messages (i.e., *missed-dequeue* occurs), D-Ctrl holds responses to

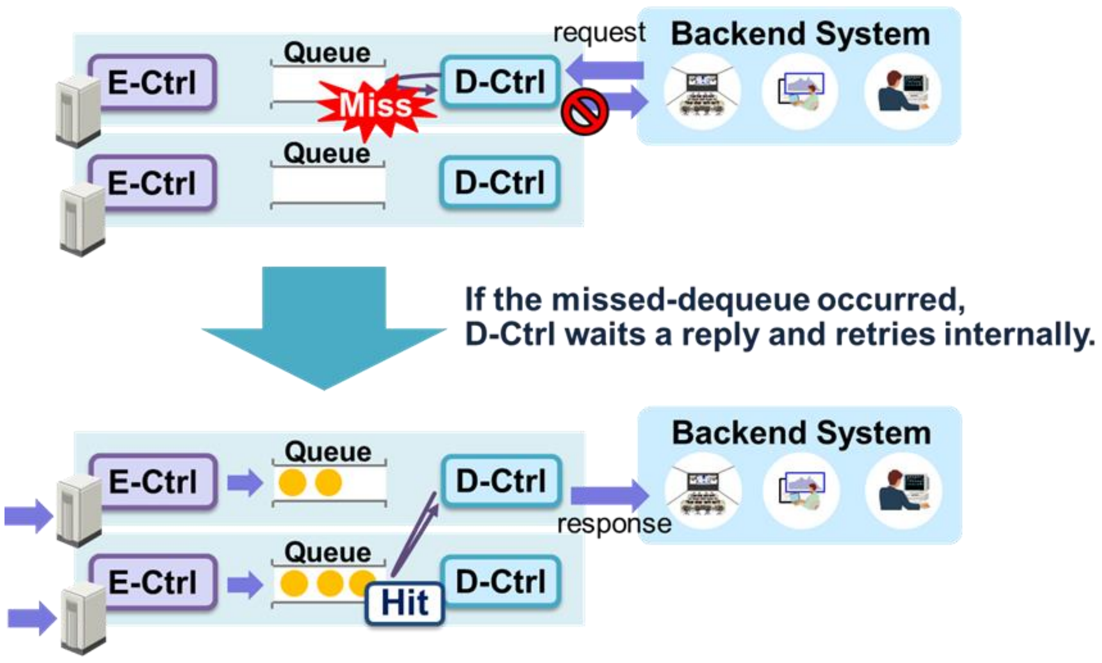


Figure 4.6 Retry Dequeue-request Scheduling (RDS) method

backend system and registers these requests to the distributed dequeue scheduler where each registered request waits for its next retrial after a certain interval. After this interval, the backend system sends the next dequeue request. RDS can reduce the third term *missed-dequeue* cost and the second term *hit-dequeue* cost of Eq. (4). Scheduling time (sleep time) of RDS is in a trade-off relationship with the latency of the message queue, which impacts the backend system's data processing time.

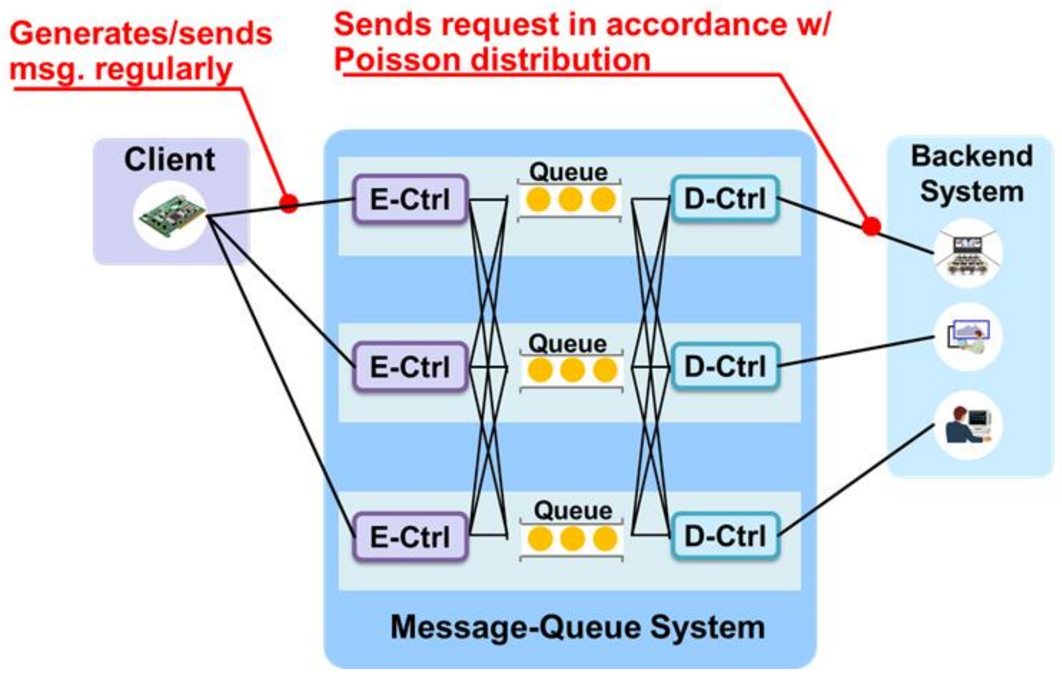


Figure 4.7 Simulation model of message queue systems

4.4. Simulation Evaluation

4.4.1. Description of the Simulation Model

To investigate the effectiveness of proposed PMS and RDS methods for maintaining high throughput in the heterogeneous environment as described in Sect.4.2.1.2, we calculate throughput of these methods in a simulation model as shown in Fig. 4.7. We set parameter values, such as enqueue/dequeue/monitoring cost, based on measured values from existing real-world message queue systems. In fact, in our message queue system, compared with the enqueue operation, the dequeue operation only includes dequeue lock (internal queue lock) and specific mutual exclusion. To emphasize this characteristic in this simulation, the cost of the dequeue operation is set to 20 times larger as that of the enqueue operation. Additionally, we set the maximum number of 3 single dequeues to meet the setting of the real message queue. This parameter contributes to keeping low latency of one dequeue by reducing access overhead of multiple servers.

Here, detailed views of E-Ctrl and D-Ctrl are also depicted in Fig. 4.8. In Fig. 4.8, the client application regularly generates messages and sends them to E-Ctrl of the message queue system, due to that most devices send messages periodically in IoT

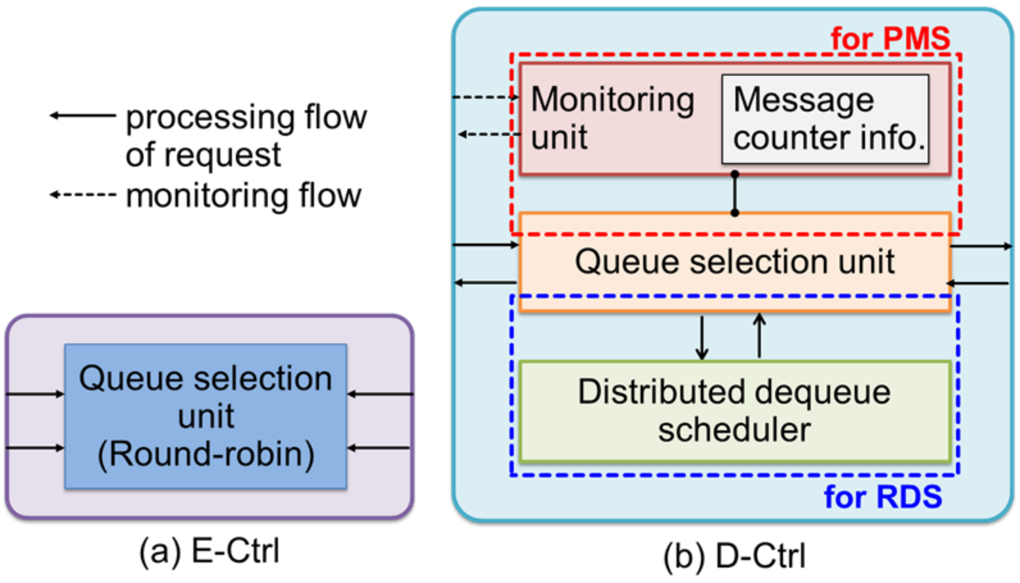


Figure 4.8 Structures of E-Ctrl and D-Ctrl

services. We assume that a client selects one of the E-Ctrl randomly each time. E-Ctrl receives this message and stores it into one of the queues selected by the queue selection unit in Fig. 4.8 (a). In this simulation, the queue selection unit selects the queue by round-robin ordering.

On the other hand, the backend system sends *dequeue requests* to D-Ctrl at random intervals following a Poisson process. Here, if we define the *dequeue request rate* as *D* [msg./s], the expected arrival rate of *dequeue requests* from one backend system used for definition of the Poisson process as λ, the maximum number of messages to collect at each dequeue request as N_{max}, and the number of backend systems as *B*, we obtain the following expression.

$$D = \lambda N_{max} B \quad (6)$$

In this expression, *dequeue request rate D* includes both, *hit-dequeue* and *missed-dequeue*. In other words, a part of λ is spent for *missed-dequeues* and λ itself depends on the processing time and settings of the backend system in the real system.

In this simulation, we set that one of backend system corresponds to one D-Ctrl without duplication. After D-Ctrl receives a dequeue request, D-Ctrl accesses queues selected by the queue selection unit in Fig. 4.8 (b). The function of this unit is different between PMS and RDS methods. In PMS, the queue selection unit selects the queues in descending order of the number of stored messages by referring to the message counter

Table I Simulation setup.

Description	**Value**
Number of E-Ctrl/queues/D-Ctrl/backend system *B*	10/10/10/10
Enqueue cost (time) c_{e0}	0.001 [s]
Dequeue cost w/o msg (time) c_{dh0}	0.02 [s]
Dequeue cost w/ msg (time) c_{dh}	0.001 [s]
Missed-dequeue cost (time) c_{dm0}	0.02 [s]
Monitoring cost (time) c_M	0.0001[s]
Max. number dequeued msg/request N_{max}	100
Arrival rate of dequeue requests from backend system λ	10-200 [/s]
Message size	1 KB
Max. number of dequeued messages for single dequeue	3

information of the monitoring unit, which is periodically updated by monitoring all queues.

In RDS, the queue selection unit selects the queue by round-robin ordering. In addition, when a *missed-dequeue* occurs, the dequeue request is registered to a distributed dequeue scheduler without responding to the backend system and retried after a certain interval.

Based on the above models for RDS and PMS methods, we computed throughput estimated by the number of received messages by the backend system. The simulation setup is listed in Table I. In this table, c_{e0}, c_{dh0}, c_{dh}, c_{dm0}, and c_M correspond to Eq. (4) and (5). For implementation, we used the library for the discrete event simulator NS3 [91] and implemented the simulation program in the C++ language. We decided data size by the reference from an equipment monitoring service.

4.4.2. Simulation Results and Discussion

Figure 4.9 shows the throughput comparison of message queue systems achieved by conventional, PMS, and RDS methods. Here, conventional method indicates the simple dequeuing based on round-robin without monitoring and scheduling. As mentioned before, *dequeue request rate* is obtained by Eq. (6) and includes both *hit-dequeue* and *missed-dequeue*. In this simulation, we set *arrival rate of dequeue requests from one of* *backend system* λ [/s] in the range from 10 to 200.

In Fig. 4.9, by applying the conventional method, throughput is gradually degraded as the arrival rate of dequeue requests λ increases. For PMS, when λ is in the range of 100 to 200, throughput of the PMS method is higher than that of the conventional method. Moreover, compared with conventional and PMS methods, especially the RDS method maintains the highest throughput, regardless of the increase in arrival rate of dequeue requests. Figure 4.10 shows the *hit-dequeue* rate comparison achieved by conventional, PMS, and RDS methods. The *hit-dequeue* rate represents the number of *hit-dequeue*s as a percentage of the number of all dequeue requests. Comparing Fig. 4.10 to Fig. 4.9, it is obvious that throughput of the message queue system has a strong relationship with the *hit-dequeue* rate. As mentioned in Sect. 4.3.2, the RDS method reduces the third term *missed-dequeue* cost of Eq. (4). On other hand, for PMS, when λ is in the range of 100 to 200, the *hit-dequeue* rate of the PMS method is lower than that of the conventional method. This result indicates that the PMS method cannot reduce the third term *missed-dequeue* cost of Eq. (4), however, the throughput in Fig. 4.9 is higher than the throughput of the conventional method when λ is in the range of 100 to 200.

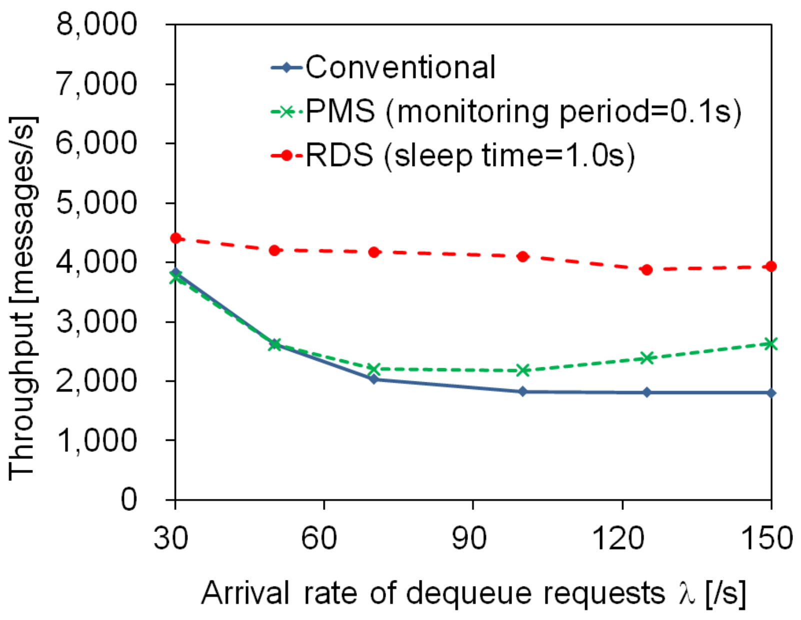


Figure 4.9 Throughput comparison between conventional method (solid line) and proposed PMS/RDS methods (dashed lines)

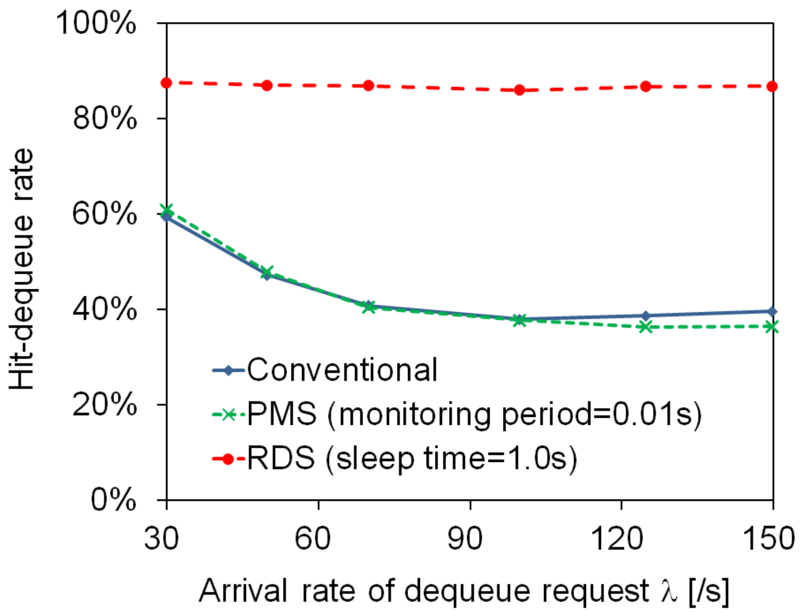


Figure 4.10 Hit-dequeue rate comparison between conventional method (solid line) and proposed PMS/RDS methods (dashed lines)

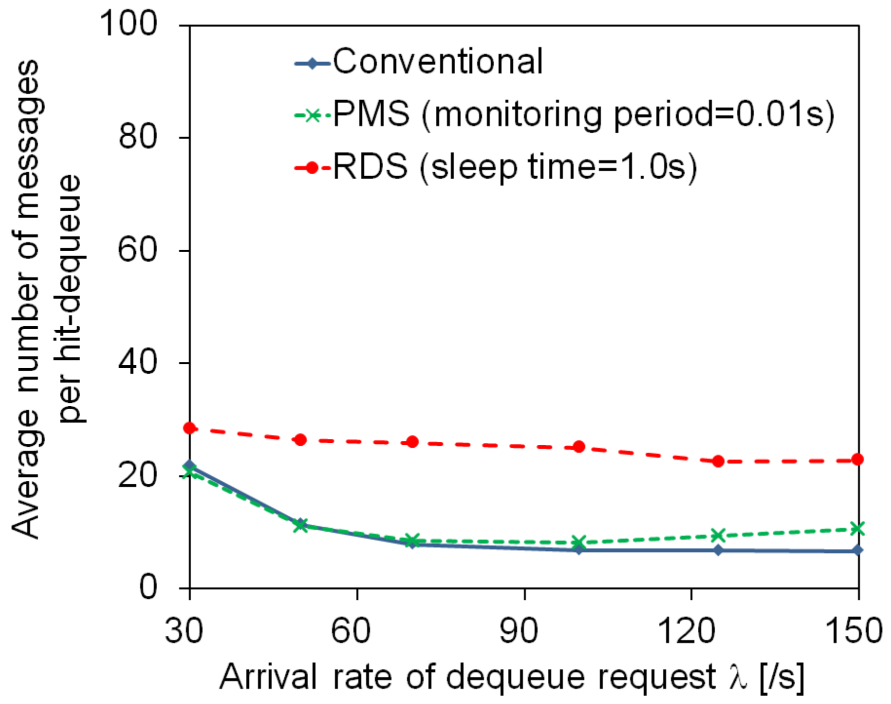


Figure 4.11 A comparison of average number of messages per *hit-dequeues* between conventional method (solid line) and proposed PMS/RDS methods (dashed lines)

Here, we consider the second term *hit-dequeue* cost of Eq. (4). *Hit-dequeue* cost depends on how many messages there are for one dequeue request. Unlike the *missed-dequeue* cost, *hit-dequeue* cost contributes to efficient dequeuing and throughput enhancement. Figure 4.11 shows the comparison of the average number of messages per *hit-dequeue* achieved by conventional, PMS, and RDS methods. The average number of messages per *hit-dequeue* describes the average number of messages D-Ctrl retrieves by one dequeue request. For the PMS method, when λ is in the range of 100 to 200, the average number of messages per *hit-dequeue* of the PMS method is higher than that of the conventional method. From this result, we proved that the PMS method dequeues more efficiently than the conventional method and the second term *hit-dequeue* cost of Eq. (4) enhances the throughput of the PMS method.

Here, we discuss why PMS does not contribute to maintaining the high throughput we expected and why RDS can maintain a high throughput. A conceivable explanation is as follows. In the PMS method, each D-Ctrl dequeues from a message queue by periodically monitoring each message queue to gather the message counter

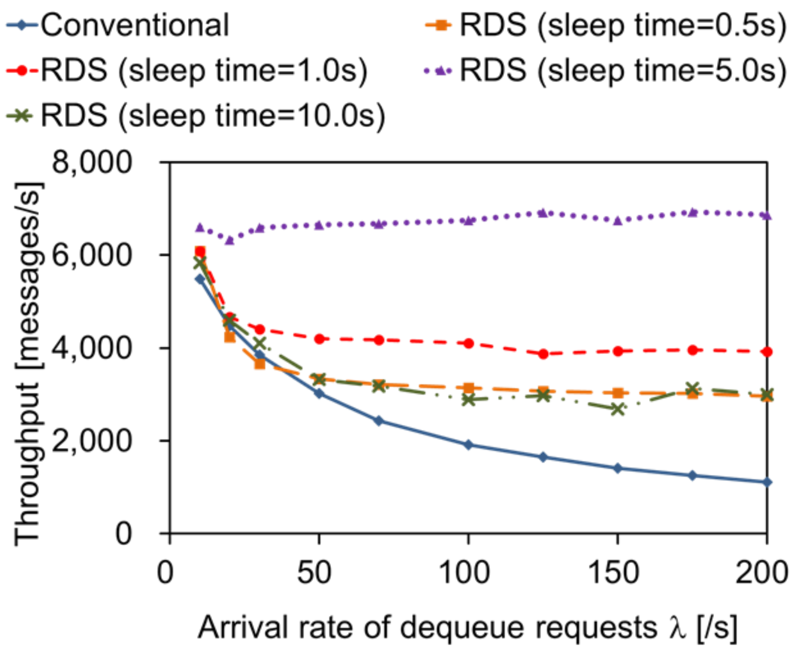


Figure 4.12 Relationship between sleep time and throughput for conventional method (solid line) and RDS (dashed lines)

information. At first, we predicted that D-Ctrl can successfully access the queue having the largest number of messages with high accuracy, which increases the *hit-dequeue* rate. However, *hit-dequeue* rate decreases in the PMS method as shown in Fig. 4.10. On the other hand, the PMS method increases the efficiency of dequeuing as shown in Fig. 4.11. These facts suggest that access contentions from D-Ctrl occur in the PMS method. We consider that multiple D-Ctrl dequeue from the same message queue which has the most messages at the same time. Although a D-Ctrl which accesses the queue first processes all messages from the queue as *hit-dequeue*, the others following it cannot dequeue any messages and generate *missed-dequeues*. In other words, PMS potentially gives D-Ctrl the access direction toward the same queues by referring to monitoring results, which increases the probability of access contention from D-Ctrl.

In contrast, in RDS each D-Ctrl independently selects queues to dequeue by round-robin, which is comparable to D-Ctrl randomly selecting queues. Therefore, the RDS method increases the *hit-dequeue* rate as shown in Fig. 4.10 and the average number of messages per *hit-dequeue* as shown in Fig. 4.11 by waiting for dequeue requests in order to increase the probability of messages arrivals at the message queues. In short, we show that our analysis of Eq. (4) is valid for the throughput degradation of message queues.

4.4.3. Evaluation and Discussion of Optimal Sleep Time for RDS

In Sect. 4.2, we showed the superiority of the RDS method. As mentioned in Sect. 4.3.2.2, sleep time of RDS determines the highest latency. Moreover, to increase the sleep time means limiting active connections between D-Ctrl and the backend system. This phenomenon may be either effective in enhancing throughput due to reducing excessive resource usage or ineffective due to limiting connections to dequeue excessively. Therefore, we assume the existence of an optimal sleep time to achieve maximum throughput without critical latency degradation. To investigate this assumption, we evaluate the relationship between sleep time and throughput for RDS method as described in Fig. 4.12. In this figure, when sleep time is 5.0 seconds, the message queue systems achieve the highest throughput. When the sleep time is longer or shorter than 5.0 seconds, we observe throughput degradation. These results indicate the existence of an optimal sleep time. The duration of sleep time strongly affects the obtained throughput.

Here, we discuss why throughput in the condition that sleep time is 5.0 seconds is highest. A conceivable explanation is as follows. As mentioned in Sect. 4.2, increasing *hit-dequeue* rate and the average number of messages per *hit-dequeue* improves dequeuing efficiency resulting in a higher message throughput. RDS enables efficiency of dequeuing by waiting for dequeue requests during sleep time in order to increase the probability of messages arriving at the message queues. However, the longer the sleep time is, the lower the throughput of the dequeue request becomes. If there are messages in a queue, decreasing throughput of the dequeue request means also decreasing the *hit-dequeue* throughput. Therefore, in RDS, there is trade-off between the efficiency of dequeue and throughput of the dequeue request. This trade-off is included in the second term *hit-dequeue* cost of Eq. (4), which depends on the queue status whether it has messages or not. We consider that a sleep time of 5.0s is well-balanced to obtain good values for both, efficiency of dequeuing and throughput of dequeue requests.

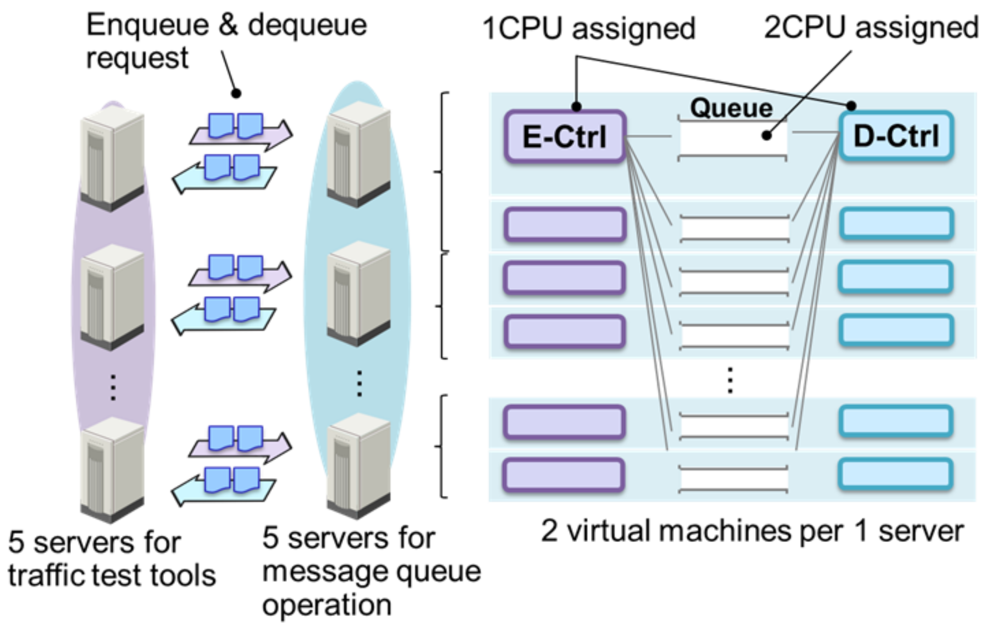

Enqueue & dequeue request
1CPU assigned
2CPU assigned
Queue
E-Ctrl
D-Ctrl
5 servers for traffic test tools
5 servers for message queue operation
2 virtual machines per 1 server

Figure 4.13 Environment of experimental evaluation of a message queue

4.5. Experimental Evaluation

4.5.1. Implementation and Methodology for Evaluation

From simulation results in Sect. 4, we revealed that high throughput of message queue systems is successfully maintained by applying the RDS method, even though the *dequeue request rate* is much higher. Actually, between the simulation and real environment, small differences of parameters/costs and deviation of processing timing are acknowledged. Therefore, to investigate the effectiveness of RDS in a real message queue, we implemented RDS for a message queue in real servers and evaluate its throughput. We evaluate the RDS method in the real heterogeneous environment described in Sect. 4.3.3.2. We designed enqueue/dequeue communication based on the *Representational State Transfer* (REST) protocol and prepared message data based on text log data of equipment monitoring services.

Figure 4.13 describes the environment of the experimental evaluation system with message queues. As shown in Fig. 4.13, we prepared a message queue having 10 sets of E/D-Ctrl and a queue with 10 virtual machines on 5 servers. 1 CPU is assigned to each set of E/D-Ctrl, and 2 CPUs are assigned to each queue. To evaluate this message queue, traffic test tools on other servers send enqueue and dequeue requests.

Table II Setup of experimental evaluation

Description	**Value**
Number of E-Ctrl/queues/D-Ctrl/backend system B	10/10/10/10
Max. number dequeued msg/request $\boldsymbol{N_{max}}$	100
Message size	1 KB
Max. number of message queues for single dequeue	3

Traffic test tools represent both field devices as message senders as well as the backend system as message receiver. The average number of received messages per second is estimated as throughput of the message queue systems.

Table 2 lists the parameter settings for the experimental evaluation setup. Note that we unified configurable design parameters of experimental evaluation with those of the simulation.

4.5.2. Results and Discussion

Figure 4.14 shows the throughput comparison of real message queue systems achieved by conventional and RDS methods with varying dequeue request rate from traffic test tools. As the dequeue request rate increases, throughput of the conventional method is degraded, however, throughput of RDS is maintained at a high level. When the arrival rate of dequeue requests reaches 200 and compared with the conventional method, the RDS method with sleep time of 0.1 s contributes to 80% improvement of throughput. Compared with simulation results, although the absolute throughput value is different, the tendency of the graph is relatively similar.

From the viewpoint of sleep time in RDS, high throughput is well maintained in the range of 0.1 s to 0.5 s. When the sleep time exceeds 1.0 s, we observe visible throughput degradation. This result strongly supports the assumption that excessive sleep time causes throughput degradation as explained in Sect. 4.2.

As a result, we reveal that the RDS method is effective for maintaining high throughput of message queue systems even if the amount of dequeue requests from the backend system greatly increases.

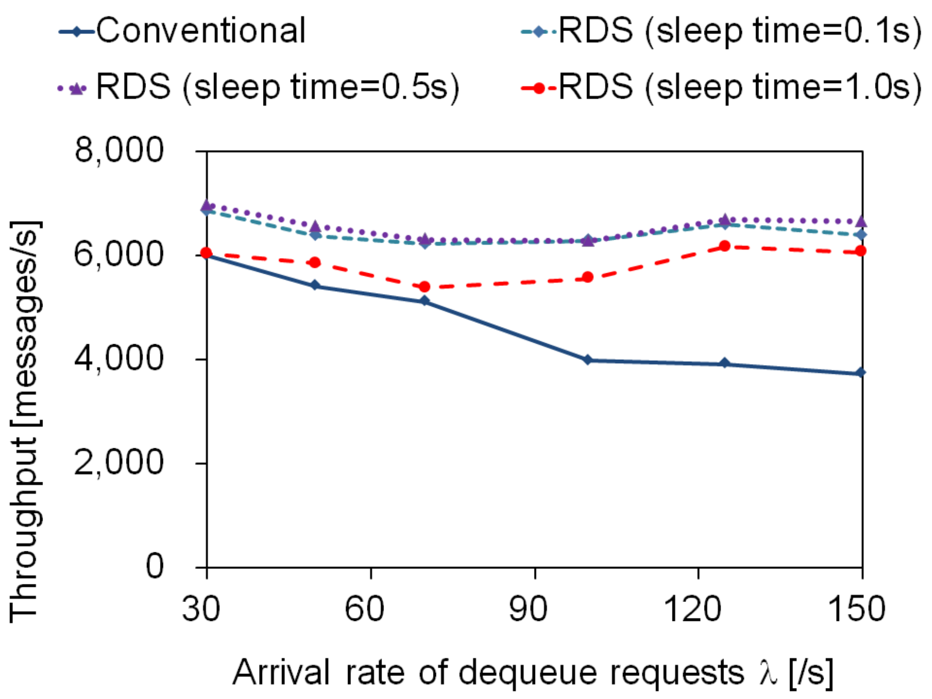


Figure 4.14 Throughput comparison between conventional and RDS method on experimental evaluation. Several patterns of are set for RDS to investigate the optimal sleep time.

4.6. Related Work

We describe related work on polling system models from two perspectives: queuing theory and IoT systems. Our proposed methods are based on research on polling system models. While a typical polling system consists of multiple queues accessed in cyclic order by a single server [92], our proposed system consists of multiple distributed message queues mesh-accessed by multiple servers.

There are many publications on polling systems that have been developed since the late 1950s [93]. In several surveys, the most notable ones written by Takagi [92], detailed and comprehensive descriptions of the mathematical analysis of polling systems are presented. Boon et al. [94] provided comprehensive descriptions of applications to polling systems, such as a production system, which consists of a single queue accessed by multiple processes.

However, to the best of our knowledge, there have been only few reports on polling methods, which have multiple queues with mesh-access from multiple servers as in our proposal. In this chapter, we simulated the polling model, which has a client application putting messages onto these queues at regular intervals and a backend application polling data at random intervals.

Regarding IoT systems, dequeuing methods follow not only the polling (“pull”) model, but also the “push” model. In the “push” model, the message queue system automatically sends messages to preliminarily registered backend systems at the timing when the message queue system receives messages from field devices. In the “pull” model, backend system send dequeue requests to the message queue system and retrieve messages.

Generally, the “push” model is effective in the case when backend systems have sufficient computing resources to process messages sent by the message queue system. Jiang et al. [95] indicate that “push” service can be faster and more energy-efficient for the backend system because in this approach the backend system does not need to look up a message queue or periodically synchronize.

On the other hand, the “pull” model is effective in the case when consumers make full use of computing resources to process messages and it is frequently used in cloud computing systems [48, 83, 84]. Kreps et al. [48] also mentioned that the “pull” model is more suitable for their applications since each client obtains some advantages: sustainability of retrieving the messages at the maximum rate and avoidance of message flooding by being pushed faster than the client can handle. Therefore, our pull-based proposal has advantages to achieve high throughput of message processing for fully utilizing computational resources of the backend system.

4.7. Conclusion

For the IoT era, message queue systems are required to have interoperability and the ability to control the huge message traffic between devices and the backend system. In this chapter, we proposed the dequeuing method called *Retry Dequeue-request Scheduling* (RDS) to solve the throughput degradation of distributed message queue systems.

RDS can reduce the unnecessary transmissions of dequeue requests to the message queues by waiting during the scheduling time for messages to arrive at the message queues. Especially, RDS can better reduce throughput degradation due to *missed-dequeue* messages than the conventional method.

By simulation evaluation, we compared throughputs achieved by the conventional method, RDS, and *Periodical Monitoring and Scheduling* (PMS), which is another dequeuing method proposed for reducing the number of *missed-dequeues* by periodically monitoring each message queue to gather message counter information. Simulation results show that RDS is able to maintain highest throughput, regardless of an increase in the dequeue request rate.

Experimental evaluation results also show that the RDS method achieves 80% higher throughput than the conventional method in real systems. Furthermore, we demonstrated that the setting of the optimal sleep time improves the efficiency of the proposed method even further.

Chapter 5

Conclusion and Future Work

The innovation of smart phones, Machine-to-Machine (M2M) communication, and the Internet of Things (IoT) is leading to an explosion in the number of devices connected to the network. In this thesis, we focused on the upcoming changes of network systems providing services or applications to a drastically increasing number of users. We discuss how message queue systems should be designed to process the significant increase in data volume created by existing and new devices and how to achieve other requirements such as availability and scalability.

We characterized these developments to proceed through roughly three phases. The first phase consists of the increase of short messages used for e.g. mobile email services, SMS, and SNS from 2008 to 2013, which started with the spread of smart phones. The second phase extended short messages to social infrastructure fields beyond smart phones, such as smart meters and health equipment from 2013 to 2017. The third phase is driven by the progress of IoT applications and its extension to industries, home, etc., which is expected to continue until about 2020.

For the first phase, the most important issue is the high-throughput and scalable processing of huge volumes of messages in smart phone services. To solve this issue, we proposed high-throughput queuing techniques and an architecture of distributed message queue systems to deliver much more messages than in the past. We designed a message queue system based on a distributed in-memory key-value store (KVS) to meet the requirements of throughput and scalability. We proposed an architecture for satisfying high throughput and high scalability in the message queue system for processing massive volumes of short-length messages through a distribution method of queue-type in-memory KVS and synchronized processing of distributed queues by single TCP connections. We embedded the proposed architecture and method into a mail system for smart phones and performed evaluations of this system. The evaluation results reveal that the throughput of the proposed message queue system achieves 3,600 msg/s per server, which is 5 times higher than that of the conventional method cooperating with RAID storages. Moreover, the throughput of the proposed KVS is 200,000 transactions/s for message sizes of 0.4 KB, which is double as fast as *memcached*.

For the second phase, M2M services such as metering and monitoring services have enhanced the social infrastructure field. As a social infrastructure, the service system, especially the message queue system, is required to satisfy both high availability and high throughput at the same time. To solve this issue, we proposed a resilient message queue system based on a distributed KVS. Its servers are interconnected among each other and messages are distributed to multiple servers during the normal processing state. Our proposed system can provide long-term availability, continuing its service regardless of where in the message queue system server/process failures may occur, by distributing messages to multiple servers as well as guaranteeing strong consistency of the messages/message queues by using KVS functions and the Paxos protocol. To achieve short-term availability even during an underlying network failure and/or slowdown of servers, we proposed message distribution methods using round-robin with a slowdown KVS exclusion and two logical KVS counter-rotating rings. Evaluation results show that this system can continue service without failover processing. Compared with the conventional method, our proposed distribution methods can reduce 92% of errors caused by server failures. Furthermore, we determined the optimum value of slowdown detection time in our distribution method.

In the third phase, IoT services require both information from historical and real-time data for their own objectives, such as optimization services or learning data analysis through trial-and-error for finding patterns in the data. This approach requires collecting large message volumes periodically created by devices. On other hand, the backend system retrieves messages from the message queue at its own non-periodic and process-dependent timing. Therefore, the control function of the massive and heterogeneous message traffic in the message system becomes a crucial issue, which can lead to dequeue throughput degradation. To solve this issue, we proposed a dequeuing method called Retry Dequeue-request Scheduling (RDS), which can reduce the unnecessary transmission of dequeue requests to the message queues by waiting during the scheduling time for new messages to arrive at the message queues. Especially, RDS can better reduce throughput degradation due to missed-dequeue messages than the conventional method. We used simulations to compare throughputs achieved by the conventional method, RDS, and Periodical Monitoring and Scheduling (PMS). Simulation results show that RDS can maintain the highest throughput, regardless of an increase in the dequeue request rate. Experimental evaluations also reveal that the RDS method achieves 80% higher throughput than the conventional method in real systems. Furthermore, we demonstrated that the setting of the optimal sleep time improves the efficiency of the proposed method even further.

We believe that in the IoT era the message queue system with high-throughput queuing proposed in the first phase and the resilient message queue system proposed in the second phase are fundamental technologies to stably process large volume messages created by IoT devices. Additionally, the increased throughput of the RDS method proposed in the third phase is essential in finding patterns in large volumes of data for various IoT services. These proposed technologies can make it much easier and faster than before to build complex IoT systems requiring high-throughput, availability, and scalability. We further believe that this can become a driving force in accelerating the innovation of IoT.

IoT will be drastically enhanced by three kinds of technological progress: *network technology*, *cloud computing*, and *sensing*. Network technology such as the 5th generation mobile networks (5G) will support IoT communication at higher capacity and lower latency [96-98]. The progress of cloud computing including machine learning/AI and *distributed computing* will enable greater variability and scalability in IoT applications [99-101]. We believe that message queue systems will be needed as frontend of IoT service systems to process much larger volume of messages than we are facing now. Messaging communication will be needed to efficiently communicate among the massive number of sensor devices. We also believe that the considerations and discussions regarding distributed message queue systems in this thesis will contribute to the better design and implementation of future IoT systems.

As our future work, we see the following challenges for IoT systems. First, research on system dimensioning and evaluation of the scalability of message queue systems needs to be continued. Previous studies on system dimensioning and scalability have proposed auto-scaling methods [102, 103], however, those studies did not consider message queue systems and heterogeneous traffic as described in this thesis. Therefore, we should consider how to determine both, the accurate message traffic and the performance of message queue systems to process this message traffic.

Second, it will also be necessary to consider the topology of the message queue system when it is extended from tens to hundreds of servers. We proposed two logical KVS counter-rotating rings in Chapter 3, and we should also consider various other topologies of KVS rings for improving availability and scalability.

Finally, we also need to research on functions of the message queue systems related to the specific requirements of IoT applications. In this thesis, we proposed fundamental methods that can be widely applied to IoT/M2M applications, and next we should consider issues of specific requirements of IoT applications. We will apply functions of priority queuing to IoT/M2M applications where messages are relayed based on the priority of the different queues. We also plan on investigating how to efficiently deliver messages from the message queue system to the devices for updating the configurations of a huge number of devices in IoT/M2M applications.

Bibliography

[1] J. Cohn, P. Finn, S. Nair, and S. Panikkar, “Device democracy: Saving the future of the Internet of Things,” *the IBM Institute for Business Value Executive Report*, July 2014.

[2] Z. M. Fadlullah, M. M. Fouda, N. Kato, A. Takeuchi, N. Iwasaki, and Y. Nozaki, “Toward intelligent machine-to-machine communications in smart grid,” *IEEE Communications Magazine*, vol. 49, no. 4, pp. 60-65, Apr. 2011.

[3] G. Wu, S. Talwar, K. Johnsson, N. Himayat, and K. D. Johnson, "M2M: From mobile to embedded internet," *IEEE Communications Magazine*, vol. 49, no. 4, pp. 36-43, Apr. 2011.

[4] J. Kim, J. Lee, J. Kim, and J. Yun, "M2M service platforms: survey, issues, and enabling technologies," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 61-76, Oct. 2013.

[5] M. Hermann, T. Pentek, and B. Otto, “Design principles for industrie 4.0 Scenarios,” in *Proceedings of IEEE Hawaii International Conference on System Sciences (HICSS)*, pp. 3928-3937, Jan. 2016.

[6] L. Atzori, A. Iera, and G. Morabito, “The internet of things: A survey,” *Computer networks*, vol. 54, no. 15, pp. 2787-2805, Oct. 2010.

[7] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, “Internet of Things (IoT): A vision, architectural elements, and future directions,” *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645-1660, Sep. 2013.

[8] J. Gantz and D. Reinsel, “The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east,” *IDC Analyze the Future*, pp. 1-16, Dec. 2012.

[9] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext transfer protocol -HTTP/1.0 ", *Internet RFC 1945*, May 1996.

[10] R. Fielding and R. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, pp. 115-150, May 2002.

[11] J. Jing, A. S. Helal, and A. Elmagarmid, “Client-server computing in mobile environments,” *ACM computing surveys (CSUR)*, vol. 31, no. 2, pp. 117-157, Jun. 1999.

[12] P. Fraternali, G. Rossi, and F. Sánchez-Figueroa, “Rich internet applications,” *IEEE Internet Computing*, vol. 14, no. 3, pp. 9-12, Jun. 2010.

[13] D. Liu and R. Deters, “The reverse C10K problem for server-side mashups,” in *Proceedings of International Conference on Service-Oriented Computing (ICSOC)*, pp. 166-177, Dec. 2008.

[14] M. Collina, G. E. Corazza, and A. Vanelli-Coralli, “Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST,” in *Proceedings of IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)* , pp. 36-41, Sep. 2012.

[15] N. De Caro, W. Colitti, K. Steenhaut, G. Mangino, and G. Reali, “Comparison of two lightweight protocols for smartphone-based sensing,” in *Proceedings of IEEE Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*, pp. 1-6, Nov. 2013.

[16] U. Hunkeler, H. L. Truong, and A. S-Clark, “MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks,” in *Proceedings of Communication Systems Software and Middleware and Workshops (COMSWARE)*, pp. 791-798, Jan. 2008.

[17] H. Subramoni, G. Marsh, S. Narravula, P. Lai, and D. K. Panda, "Design and evaluation of benchmarks for financial applications using Advanced Message Queuing Protocol (AMQP) over InfiniBand," in *Proceedings of IEEE Workshop on High Performance Computational Finance (WHPCF)*, pp. 1-8, Nov. 2008.

[18] A. A-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, “Internet of Things: A survey on enabling technologies, protocols, and applications,” *IEEE Communication Surveys & Tutorials*, vol. 17, no. 4, June 2015.

[19] C. Pereira and A. Aguiar, "*Towards efficient mobile M2M communications: Survey and open challenges*," Sensors, vol. 14, no. 10, pp. 19582-19608, Oct. 2014.

[20] J. S. Leu, C. F. Chen, and K. C. Hsu, “Improving heterogeneous SOA-based IoT message stability by shortest processing time scheduling,” *IEEE Transactions on Services Computing*, vol. 7, no. 4, pp. 575-585, May 2013.

[21] N. L. Tran, S. Skhiri, and E. Zim, “Eqs: An elastic and scalable message queue for

the cloud,” in *Proceedings* of *IEEE Cloud Computing Technology and Science (CloudCom)*, pp. 391-398, Nov. 2011.

[22] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann, "Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark," *Performance Evaluation*, vol. 66, no. 8, pp. 410-434, Aug. 2009.

[23] K. Sachs, S. Kounev, S. Appel, and A. Buchmann, “Benchmarking of message-oriented middleware,” in *Proceedings of ACM International Conference on Distributed Event-Based Systems (DEBS)*, p. 44, July 2009.

[24] K. Sachs, S. Appel, S. Kounev, and A. Buchmann, “Benchmarking publish/subscribe-based messaging systems,” in *Proceeding of International Conference on Database Systems for Advanced Applications (DASFAA)*, pp. 203-214, Apr. 2010.

[25] SendMail Inc. “Welcome to the Sendmail Community,” http://www.sendmail.com/sm/open_source/, accessed on Nov. 16, 2016.

[26] M. Kinoshita, M. Nakahara, and T. Sagara, “An implementation and evaluation of multiprotocol message gateway,” in *Proceedings of the 71th National Convention of IPSJ*, pp. 3.1-3.2, Mar. 2009.

[27] The Statistics Portal, “Number of subscribers to wireless carriers in the U.S. from 1st quarter 2013 to 3rd quarter 2016, by carrier,” https://www.statista.com/statistics/283507/subscribers-to-top-wireless-carriers-in-the-us/, accessed on Jan. 04, 2017.

[28] Telecommunication Carries Association, “Number of subscribers by carriers”, http://www.tca.or.jp/english/database/, accessed on Jan. 04, 2017.

[29] B. Manning, “1 billion-plus smart meters to be installed globally by 2022,” http://centricdigital.com/blog/internet-of-things/billion-smart-meters-installs/, accessed on Dec. 12, 2016.

[30] The Ministry of Economy, Trade and Industry, “A report on power companies and installation status of smart meters,” http://www.meti.go.jp/committee/sougouenergy/denryoku_gas/kihonseisaku/pdf/001_07_01.pdf, accessed on Dec. 12, 2016. (in Japanese)

[31] M. Brettel, , N. Friederichsen, M. Keller, and M. Rosenberg, “How virtualization, decentralization and network building change the manufacturing landscape: An industry 4.0 perspective,” *International Journal of Mechanical, Industrial Science and Engineering*, vol. 8, no. 1, pp. 37-44, 2014.

[32] C. Deglise, L. Suggs, and P. Odermatt, "Short message service (SMS) applications for disease prevention in developing countries," *Journal of medical Internet research*, vol. 14, No. 1, p. e3, Jan. 2012.

[33] I. Jung, H. Kim, D. Hong, and H. Ju, "Protocol reverse engineering to facebook messages," in *Proceedings of IEEE International Conference on Intelligent Systems, Modelling and Simulation (ISMS)*, pp. 539-542, Jan. 2013.

[34] B. Furht and S. A. Ahson, "Long Term Evolution: 3GPP LTE radio and cellular technology," *Crc Press*, Apr. 2016.

[35] J. Cownie and W. Gropp, "A standard interface for debugger access to message queue information in MPI," in *Proceeding of European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting(Euro PVM/MPI)*, pp. 51-58, Sep. 1999.

[36] G. Banavar, T. Chandra, R. Strom, and D. Sturman, "A case for message oriented middleware," in *Proceeding of International Symposium on Distributed Computing (DISC)*, pp. 1-17, Sep. 1999.

[37] R. T. Fielding, "Architectural styles and the design of network-based software architectures," *PhD Thesis*, University of California, Irvine. 2000.

[38] A. Erdeljan, F. Kuli, and S. Lukovi, "Software architecture for smart metering systems with virtual power plant," in *Proceedings of IEEE Mediterranean Electrotechnical Conference(MELECON)*, pp. 448-451, Apr. 2010.

[39] N. Owada, "How systems go down," *NIKKEI BP*, pp. 94-105, 2009 (in Japanese).

[40] H. Okabe, "Report of NIKKEI COMPUTER," *IT Pro*, http://itpro.nikkeibp.co.jp/article/COLUMN/20120824/417984/ accessed on Sep. 18, 2014 (in Japanese).

[41] Bank of Japan, "BOJ report and research papers," https://www.boj.or.jp/research/brp/ron_2010/data/ron1011a.pdf, accessed on Sep. 18, 2014 (in Japanese).

[42] Industrial Internet Consortium (IIC), "The Industrial Internet reference architecture technical report," http://www.iiconsortium.org/IIRA-1-7-ajs.pdf, accessed on Dec. 12, 2016.

[43] S. Huang, Y. Chen, X. Chen, K. Liu, X. Xu, C. Wang, K. Brown, and I. Halilovic, "The next generation operational data historian for IoT based on informix," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 169-176, June 2014.

[44] M. E. Porter and J. E. Heppelmann, “How smart, connected products are transforming competition,” *Harvard Business Review*, vol. 92, no. 11, pp. 64-88, Nov. 2014.

[45] D. Niyato, L. Xiao, and P. Wang, “Machine-to-machine communications for home energy management system in smart grid,” *IEEE Communications Magazine*, vol. 49, nol. 4, pp. 53-59, Apr. 2011.

[46] D. Kibara, K. C. Morris, and S. Kumaraguru, “Methods and tools for performance assurance of smart manufacturing systems,” *Journal of Research of the National Institute of Standards and Technology*, vol. 121, pp. 1-47, Dec. 2015.

[47] G. Meijer, K. Makinwa, and M. Pertijs, “Smart sensor systems: Emerging technologies and applications,” *John Wiley & Sons*, Apr. 2014.

[48] J. Kreps, N. Narkhede, and J. Rao, “Kafka: A distributed messaging system for log processing,” in *Proceedings of Networking Meets Databases (NetDB)*, pp. 1-7, June 2011.

[49] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *Proceedings of Symposium on Operating Systems Principles (SOSP)*, pp. 423-438, Nov. 2013.

[50] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni and N. Bhagat, “Storm@twitter,” in *Proceedings of the International Conference on ACM SIGMOD Management of Data*, pp. 147-156, June 2014.

[51] R. Cattell, “Scalable SQL and NoSQL data stores,” *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12-27, Dec. 2011.

[52] T. Sakaki, M. Okazaki, and Y. Matsuo, "Tweet analysis for real-time event detection and earthquake reporting system development," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 4, pp. 919-931, Apr. 2013..

[53] Ministry of Internal Affairs and Communications, “Ministry of Public Management,” http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h23/pdf/n0010000.pdf, accessed on Nov.16 2016. (in Japanese).

[54] Postfix, “The postfix home page,” http://www.postfix.org/, accessed on Nov. 16, 2016.

[55] Memcached, “What is memcached?,” http://memcached.org/, accessed on Nov. 16 2016.

[56] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, and D.

Stafford, “Scaling memcache at facebook,” in *Proceeding of Presented as part of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 385-398, Apr. 2013.

[57] Y. Wang, H. Chen, B. Wang, J. M. Xu, and H. Lei, “Scalable queuing service based on an in-memory data grid,” in *Proceedings of International Conference on IEEE e-Business Engineering (ICEBE)*, pp. 236-243, Nov. 2010.

[58] L. Xiang, Y. Xu, J. Lui, and Q. Chang, “Optimal recovery of single disk failure in RDP code storage systems,” *ACM SIGMETRICS Performance Evaluation Review*, Vol. 38, No. 1, pp. 119-130, June 2010.

[59] Y. Zhu, P. P. Lee, Y. Hu, L. Xiang, and Y. Xu, “On the speedup of single-disk failure recovery in XOR-coded storage systems: theory and practice,” *in Proceedings of IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1-12, Apr. 2012.

[60] E. A. Brewer, “Towards robust distributed systems,” in *Proceedings of the Annual Symposium on ACM Principles of Distributed Computing*, p. 7, July 2000.

[61] G. Hasegawa and M. Murata, “Transport-layer protocols for high-speed and long-delay networks,” in *Proceedings of IEICE technical report (IN2006-169)*, pp. 41-46, Feb. 2007 (in Japanese).

[62] M. Welsh, D. Culler, and E. Brewer, “SEDA: An architecture for well-conditioned, scalable Internet services,” in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pp. 230-243, Oct. 2001.

[63] N. Christenson, T. Bosserman, and D. Beckemeyer, “Highly scalable electronic mail service using open systems,” in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pp. 1-11, Dec. 1997.

[64] Y. Saito, B. N. Bershad, and H. M. Levy, “Manageability, availability and performance in Porcupine: A highly scalable, cluster-based mail service”, in *Proceedings of ACM Symposium on Operating Systems Principle (SOSP)*, pp. 1-15, Dec. 1999.

[65] J. R. von Behren, S. Czerwinski, A. D. Joseph, E. A. Brewer, and J. Kubiatowicz, “NinjaMail: The design of a high-performance clustered, distributed e-mail system,” in *Proceeding of International Workshop on Scalable Web Services (SWS)*, pp. 151-158, Aug. 2000.

[66] A. Lakshman and P. Malik, “Cassandra - A decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35-40, Apr.

2009.

[67] M. Burrows, “The chubby lock service for loosely-coupled distributed systems,” in *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 335-350, Nov. 2006.

[68] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, May 1998.

[69] L. Lamport, “Paxos made simple,” *ACM SIGACT News*, vol. 32, no. 4, pp. 18-25, Nov. 2001.

[70] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Randhakrishnan, V. Subramanya, and A. Vahdat, “PortLand: A scalable fault-tolerant layer 2 data center network fabric,” *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 39-50, Oct. 2009.

[71] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, “VL2: A scalable and flexible data center network,” in *Proceedings of ACM SIGCOMM*, pp. 51-62, Aug. 2009.

[72] G. DeCandia, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. “Dynamo: Amazon’s highly available key-value store,” *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205-220, Dec. 2007.

[73] J. Rao, E. J. Shekita, and S. Tata, “Using paxos to build a scalable, consistent, and highly available datastore,” in *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 243-254, 2011.

[74] J. Wang, B. V. Murciano, J. Bigham, and M. Q. Isrc, “Towards a resilient message oriented middleware for mission critical applications,” *in Proceedings of the International Conference on Adaptive and Self-adaptive Systems and Applications (ADAPTIVE)*, pp. 21-26, Nov. 2010.

[75] J. Mitsui, “Critical success factors of mission critical system based on windows server”, *UNISYS TECHNOLOGY REVIEW*, vol. 28, no. 1, pp. 29-43, May 2008. (in Japanese)

[76] Y. Miyata, M. Obata, T. Ohta, and H. Nishiyama, “Proposal of GC time reduction algorithm for large java object cache,” *IPSJ Transactions Programming*, vol. 5, no.3, pp. 29-39, Aug. 2012.

[77] Apache Hbase Project, “Apache Hbase,” http://hbase.apache.org/, accessed on Sep. 18, 2014.

[78] Information-technology Promotion Agency Japan, “High reliability lessons for IT systems,” http://www.ipa.go.jp/files/000038843.pdf, accessed on Sep. 18, 2014 (in Japanese).

[79] A. Egami, “The measures and background of catastrophic service failures,” http://e-public.nttdata.co.jp/topics_detail4/contents_type=20&id=653, accessed on May 18, 2014 (in Japanese).

[80] M. Stonebraker, “The case for shared nothing,” *IEEE Technical Committee on Database Engineering*, vol. 9, no. 1, pp. 4-9, Mar. 1986.

[81] M. Castro, A. J. Jara, and A. F. Skarmeta, “An analysis of M2M platforms: challenges and opportunities for the Internet of Things,” in *Proceedings of International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, pp. 757-762, July 2012.

[82] E. Adi, M. Loeis, M. Sunur, and K. Tjandrean, “Reliability implementation over message queue in the Internet of Things,” in *Proceedings of Mobile Communications, Networking and Applications (MobiCONA)*, p. 1, 2012.

[83] Amazon, “Amazon Kinesis”, http://aws.amazon.com/kinesis/, accessed on May 1, 2016.

[84] B. Familiar, “Microservices, IoT and Azure: Leveraging DevOps and Microservice Architecture to deliver SaaS Solutions,” *Apress*, pp. 133-163. Oct. 2015.

[85] Z. Yang, Y. Peng, Y. Yue, X. Wang, Y. Yang, and W. Liu, “Study and application on the architecture and key technologies for IoT,” in *Proceedings of Multimedia Technology (ICMT)*, pp. 747-751, July 2011.

[86] A. Azzarà, D. Alessandrelli, S. Bocchino, P. Pagano, and M. Petracca, “Architecture, functional requirements, and early implementation of an instrumentation grid for the IoT,” in *Proceedings of High Performance Computing and Communication & Embedded Software and Systems (HPCC-ICESS)*, pp. 320-327, June 2012.

[87] J. Zhao, X. Zheng, R. Dong, and G. Shao, “The planning, construction, and management toward sustainable cities in China needs the environmental Internet of Things,” *International Journal of Sustainable Development & World Ecology*, vol. 20, no. 3, pp. 195-198, May 2013.

[88] J. Humble and J. Molesky, “Why enterprises must adopt devops to enable continuous delivery,” *Cutter IT Journal*, vol. 24, no. 8, p. 6, Aug. 2011.

[89] C. Bormann, A. P. Castellani, and Z. Shelby, "CoAP: An application protocol for billions of tiny internet nodes", *IEEE Internet Computing*, vol. 16, no. 2, p. 62, 2012.

[90] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.2," *IETF RFC 5246*, pp. 1-104, Aug. 2008.

[91] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," *Modeling and Tools for Network Simulation*, pp. 15-34. Sep. 2010.

[92] H. Takagi, "Queuing analysis of polling models," *ACM Computing Surveys (CSUR)*, vol. 20, no. 1, pp. 5-28, Mar. 1988.

[93] C. Mack, "The efficiency of N machines uni-directionally patrolled by one operative when walking time is constant and repair times are variable," *Journal of the Royal Statistical Society Series B*, vol. 19, no. 1, pp. 173-178, Oct. 1957.

[94] M. A. A. Boon, R. D. van der Mei, and E. M. M. Winands, "Applications of polling systems," *Surveys in Operations Research and Management Science*, vol. 16, no. 2, pp. 67-82, Feb. 2011.

[95] P. Jiang, J. Bigham, E. Bodanese, and E. Claudel, "Publish/subscribe delay-tolerant message-oriented middleware for resilient communication," *IEEE Communications Magazine*, vol. 49, no. 9, pp. 124-130, Sep. 2011.

[96] C. X. Wang, F. Haider, X. Gao, X. H. You, Y. Yang, D. Yuan, and E. Hepsaydir, "Cellular architecture and key technologies for 5G wireless communication networks," *IEEE Communications Magazine*, vol. 52, no. 2, pp. 122-130, Feb. 2014.

[97] P. Demestichas, A. Georgakopoulos, D. Karvounas, K. Tsagkaris, V. Stavroulaki, J. Lu, and J. Yao, "5G on the horizon: key challenges for the radio-access network," *IEEE Vehicular Technology Magazine*, vol. 8, no. 3, pp. 47-53, July 2013.

[98] A. Osseiran, F. Boccardi, V. Braun, K. Kusume, P. Marsch, M. Maternia, and H. Tullberg, "Scenarios for 5G mobile and wireless communications: the vision of the METIS project," *IEEE Communications Magazine*, vol. 52, no. 5, pp. 26-35, May 2014.

[99] M. A. Alsheikh, S. Lin, D. Niyato and H. P. Tan, "Machine learning in wireless sensor networks: Algorithms, strategies, and applications," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 1996-2018, Apr. 2014.

[100] K. Hwang, J. Dongarra, and G. C. Fox, "Distributed and cloud computing: from parallel processing to the internet of things," *Morgan Kaufmann*, Dec. 2013.

[101] M. N. Sadiku, S. M. Musa, and O. D. Momoh, “Cloud computing: opportunities and challenges,” *IEEE potentials*, vol. 33, no. 1, pp. 34-36, Jan. 2014.

[102] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, “Autoscale: Dynamic, robust capacity management for multi-tier data centers,” *ACM Transactions on Computer Systems (TOCS* , vol. 30, no. 4, pp. 14. Nov. 2012.

[103] M. Mao, J. Li, and M. Humphrey, “Cloud auto-scaling with deadline and budget constraints,” *In Proceedings of the IEEE/ACM International Conference on Grid Computing*, pp. 41-48, Oct. 2010.