Regular Paper

High Throughput Dequeuing Technique in Distributed Message Queues for IoT

Masafumi Kinoshita[1,2,a) Hiroaki Konoura[1] Takafumi Koike[1] Kenji Leibnitz[2,3]
Masayuki Murata[2]

Received: May 23, 2016, Accepted: November 1, 2016

Abstract: With the dramatic increase in Internet of Things (IoT) related messaging volume, message queue systems are highly required for both interoperability among devices, as well as for control message traffic between devices and heterogeneous back-end systems (BES). When connected BES issue several dequeue requests to the message queue and no message is available, the frequency of *missed-dequeues* increases, which causes a degradation of the maximum throughput. Therefore, we propose the retry dequeue-request scheduling (RDS) method that decreases the number of dequeue requests from the BES by delaying the replies to the BES when *missed-dequeues* occur. Simulation and experimental evaluations show that the throughput of the RDS method achieves 180% of that of the conventional dequeue method.

Keywords: Internet of Things (IoT), distributed messaging, message queue, key-value store (KVS)

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 [1]. IoT service systems utilizing data from devices typically consist of 3 groups: *field devices* which send and receive data, *back-end application systems* (BES) in a data center/cloud, and the *message queue* (or message hub/message bus) systems located between the devices and back-end systems.

Message queue systems are widely used for interoperability and control of the huge message traffic between devices and BES [2], [3]. 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 [4], Amazon Kinesis [5], Azure IoT Hub [6], 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 a high-throughput and reliable message queue system based on a distributed in-memory key-value store (KVS) in [7], [8], [9].

Here, we will address another issue of traffic control between devices and BES 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, BES 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 BES, 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 [9]. In the past, the specifications of field devices and BES were defined in advance. However, today's IoT service systems as well as development and operations (DevOps) trends require rapid implementation and continuous modification, additionally to the BES also becoming adaptable [10], [11], [12], [13]. Furthermore, progress of distribution platforms such as Spark [14] or Storm [15], have dramatically improved the performance of BES. In this background, updating the processes or parameter settings of BES can impact the system's performance.

Actually, when the number of BES 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 paper, 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.

[1] Hitachi Ltd., Chiyoda, Tokyo 100–8280, Japan
[2] Osaka University, Suita, Osaka 565–0871, Japan
[3] National Institute of Information and Communications Technology, Suita, Osaka 565–0871, Japan
[a)] masafumi.kinoshita.rt@hitachi.com

The rest of this paper is organized as follows. The background and issues of message queues are introduced in Section 2. Section 3 presents our proposed method and its design. Section 4 shows its performance evaluation by simulation, followed by Section 5 where we present the experimental evaluation. Section 6 describes related work and Section 7 concludes this paper.

2. Background

2.1 Outline of IoT Service System

2.1.1 Message Queue in IoT Service System

Figure 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 smart 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) [16], *Representational State Transfer* (REST) [17], or *Constrained Application Protocol* (CoAP) [18], and by making devices and BES 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 BES 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 BES to design their system without considering the entire volume of the messaging traffic.

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 [19], [20], [21], [22], [23]. In Refs. [19], [20], [22], 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 Ref. [21], optimization at the enterprise level in smart manufacturing results in requiring only periodically collected data. In Ref. [23], 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 values 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 sizes of sensors highly depend on their service requirements and protocols, such as MQTT, REST, and Transport Layer Security (TLS) [24], etc. From our past experiences with specific use cases, such as monitoring or optimization services, we assume in this paper that the data size is 1 KB, which can be widely applied to IoT services.

On the other hand, BES collect data for various IoT objectives, such as monitoring and optimization and retrieve messages from the queue at their own timing, which is non-periodic and process-dependent. These processing times differ by context of message, message size, and other related data. To achieve higher throughput by fully utilizing computational resources, the BES retrieves messages from the queue with a pull-based method [4]. We describe further details in Section 6. In addition, progress in distribution platforms, such as Spark, leads to a dramatic change in processing time of the BES.

Here it can be seen that while devices send massive amounts of periodical messages, BES 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 paper, we are targeting these heterogeneous environments in the IoT service system.

2.2 Conventional Approach using Distributed Message Queues

2.2.1 Architecture for High Scalability and Availability

In our previous work, we proposed a high-throughput and reliable message queue system based on a distributed in-memory key-value store (KVS) for social infrastructure systems [7], [8], [9] (**Fig. 2**). The proposed messaging system adopts a fabric architecture with connected full-meshed servers for high scalability and availability. The proposed messaging 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 BES and retrieving the messages from queues. This structure enables eliminating a single point of failure and enhances the horizontal scalability of each part.

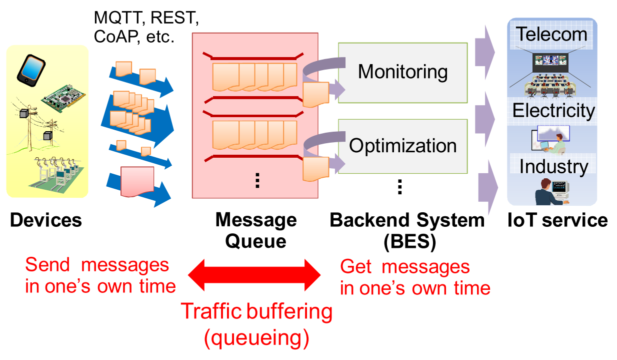


Fig. 1 Structure of IoT service system.

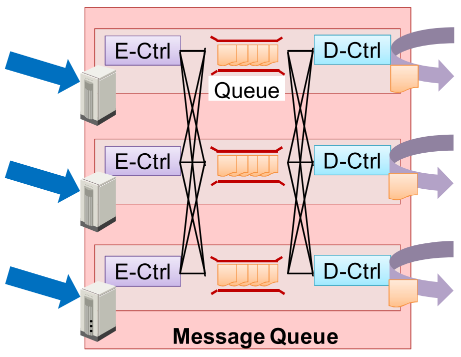


Fig. 2 Overview of distributed message queue system. E-Ctrl and D-Ctrl denote *enqueue controller* and *dequeue controller*, respectively.

2.2.2 Transparency in Distributed Message Queues

In the proposed queue system as shown in **Fig. 3**, E-Ctrl and D-Ctrl provide access transparency and location transparency for devices and BES. Let us detail their transparency using Fig. 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/BES 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 BES 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 BES retrieves a fixed number of messages by a single dequeue-request and the D-Ctrl can dequeue messages from multiple queues. If the BES 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 BES. 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 BES to get the messages without considering the location where they were physically stored.

2.3 Throughput Degradation Problem of Message Queue

As mentioned above, BES are required for rapid implementation and continuous modification due to DevOps trends in IoT services. Developers modify BES 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 BES for achieving the IoT service requirement. The developers also determine the number of BES to ensure sufficient throughput.

However, as a result of the real-world performance test in the case where a large number of BES is connected to our proposed message queue, the throughput is degraded by 20 percent 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-dequeue*s, which occur when there are no messages to retrieve from the selected queue, consume the computational resources for enqueue and dequeue operations (see Section 3.1).

We focused on the enqueue traffic in the conventional approach and extended 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 BES 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 paper.

3. Analysis of Throughput Degradation and Proposal

In this section, we first analyze the processing of the message queue in order 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 BES.

3.1 Process of Distributed Message Queue

Figure 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 BES, it selects one of the message queues and sends a dequeue request. Here, BES 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

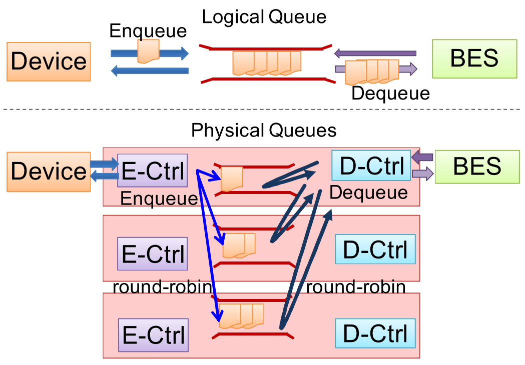


Fig. 3 Transparency in distributed message queues.

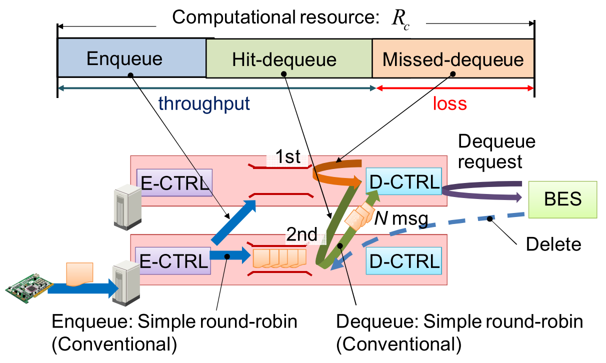

Computational resource: R_c
Enqueue
Hit-dequeue
Missed-dequeue
throughput
loss
Dequeue request
1st
E-CTRL
D-CTRL
BES
2nd
N msg
Delete
E-CTRL
D-CTRL
Enqueue: Simple round-robin (Conventional)
Dequeue: Simple round-robin (Conventional)

Fig. 4 Process of distributed message queue.

queue until it gets $N_{\max}$ messages in total or a *retry out* occurs.

After the BES 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.

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 the following expression.

$$R_c = C_e + C_{dh} + C_{dm} \tag{1}$$

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.

$$R_c = Ec_{e0} + C_{dh} + D_m c_{dm0} \tag{2}$$

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).

$$C_{dh} = D_h c_{dh0} + \sum_{i=1}^{D_h} N_i c_{dhN_i} \tag{3}$$

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).

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

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.

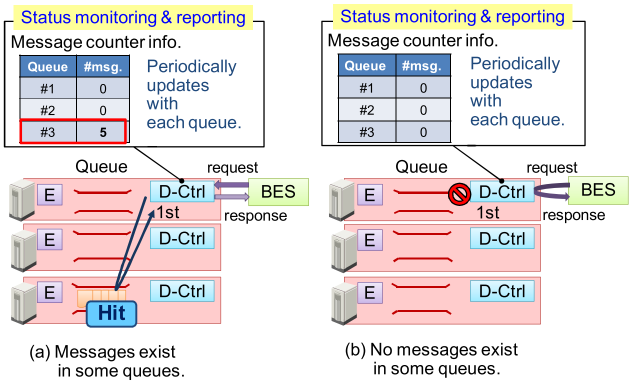

Status monitoring & reporting
Message counter info.
Queue
#msg.
#1
0
#2
0
#3
5
Periodically updates with each queue.
Queue
request
response
1st
E
D-Ctrl
BES
Hit
(a) Messages exist in some queues.
Status monitoring & reporting
Message counter info.
Queue
#msg.
#1
0
#2
0
#3
0
Periodically updates with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

Fig. 5 *Periodical monitoring and scheduling* (PMS) method.

Here, we consider the problem of throughput degradation described in Section 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.

3.3 Proposed Methods

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

3.3.1 Periodical Monitoring Scheduling (PMS)

Figure 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 sufficient messages (Fig. 5 (a)) by status monitoring. If there are no queues which have enough messages, D-Ctrl regulates the access to the message queues (Fig. 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.

$$R_c = E(c_{e0} + c_{dh0}) + \sum_{i=1}^{D_h} N_i c_{dhN_i} + D_m c_{dm0} + Mc_M \tag{5}$$

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

3.3.2 Retry Dequeue-Request Scheduling (RDS)

Figure 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 BES, D-Ctrl accesses the selected message queue. If D-Ctrl cannot get any messages (i.e., *missed-dequeue* occurs), D-Ctrl holds responses to BES and registers this request to the distributed dequeue scheduler where each registered request waits for its next retrial after a certain interval. After this interval, BES send 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

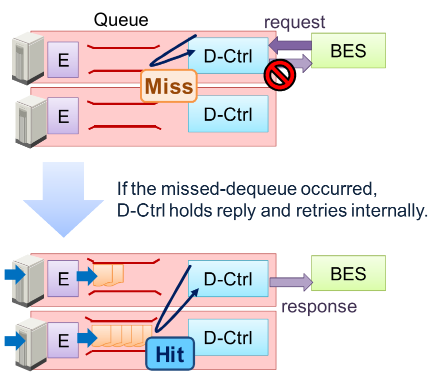

Queue
request
E
D-Ctrl
BES
Miss
E
D-Ctrl
If the missed-dequeue occurred,
D-Ctrl holds reply and retries internally.
E
D-Ctrl
BES
response
E
D-Ctrl
Hit

Fig. 6 *Retry Dequeue-request Scheduling* (RDS) method.

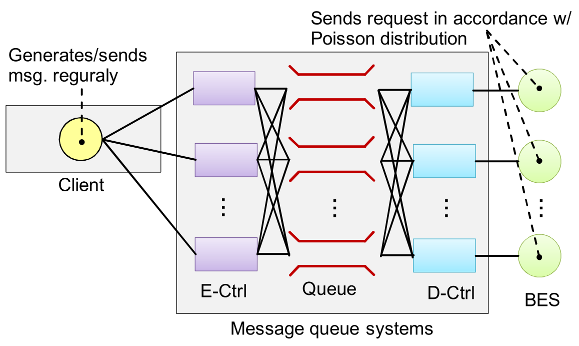


Fig. 7 Simulation model of message queue systems.

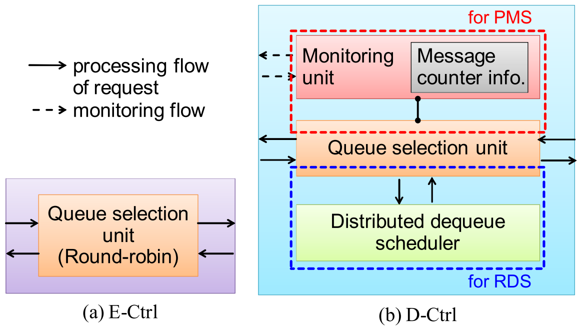


Fig. 8 Structures of E-Ctrl and D-Ctrl.

Table 1 Simulation setup.

Description	**Value**
Number of E-Ctrls/queues/D-Ctrls/BES 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 one of BES λ	10-200 [/s]
Message size	1 KB
Max. number of dequeued message queues for single dequeue	3

BES's data processing time.

4. Simulation Evaluation

4.1 Description of the Simulation Model

To investigate the effectiveness of the proposed PMS and RDS methods for maintaining high-throughput in the heterogeneous environment described in Section 2.1.2, we calculate throughput of these methods in a simulation model as shown in **Fig. 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 and specific mutual exclusion [9]. 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 3 as the maximum number of dequeued message queues for single dequeue 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. 8**. In Fig. 8, the client application regularly generates messages and sends them to E-Ctrls of the message queue system, since most devices send periodic messages in IoT services. We assume that a client selects one of the E-Ctrls randomly each time. E-Ctrl receives this message and stores it into one of the queues selected by the queue selection unit in Fig. 8 (a). In this simulation, the queue selection unit selects the queue by round-robin ordering.

On the other hand, BES send *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 of the BES used for the definition of a Poisson process as λ, the maximum number of messages to collect in each dequeue request as $N_{\max}$, and the number of BES as B, we obtain the following expression.

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

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*. λ depends on the processing time and settings of BES in real systems.

In this simulation, we set that one of the BES 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. 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 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 BES 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 BES. The simulation setup is listed in **Table 1**. In this table, c_{e0}, c_{dh0}, c_{dh}, c_{dm0}, and c_M correspond to Eqs. (4) and (5). For implementation, we used the library for the discrete event simulator NS3 [25] and implemented the simulation program in C++ language. We used the data size from the recommendation

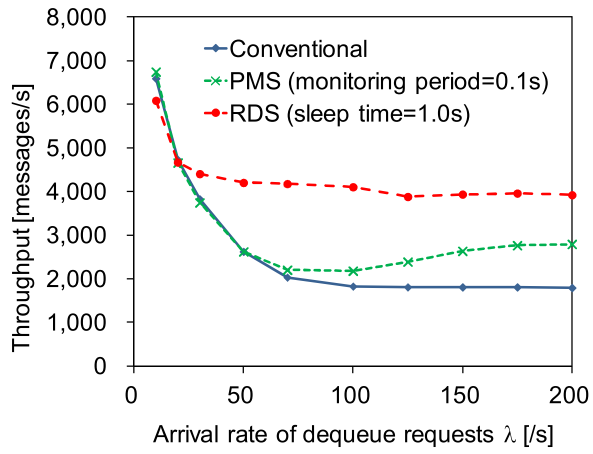


Fig. 9 Throughput comparison between conventional method (solid line) and proposed PMS/RDS methods (dashed lines).

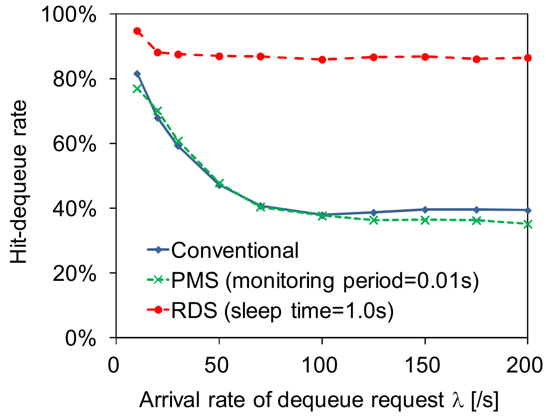


Fig. 10 *Hit-dequeue* rate comparison between conventional method (solid line) and proposed PMS/RDS methods (dashed lines).

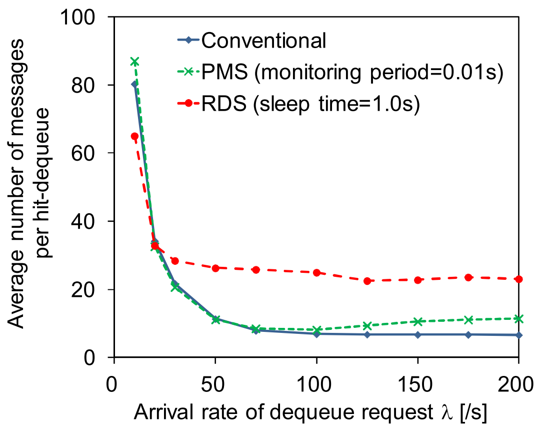


Fig. 11 A comparison of average number of messages per *hit-dequeues* between conventional method (solid line) and proposed PMS/RDS methods (dashed lines).

of the equipment monitoring service.

4.2 Simulation Results and Discussion

Figure 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 BES* λ [/s] in the range from 10 to 200.

In Fig. 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 throughput of the conventional method. Moreover, compared with conventional and PMS methods, the RDS method maintains the highest throughput, regardless of the increase in arrival rate of dequeue requests.

Figure 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. 10 to Fig. 9, it is obvious that throughput of the message queue system has a strong relationship with the *hit-dequeue* rate. As mentioned in Section 3.2, the RDS method reduced the third term *missed-dequeue* cost of Eq. (4). On the 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. 9 is higher than the throughput of the conventional method when λ is in the range of 100 to 200.

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 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* means 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 contributes to maintaining 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 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. 10. On the other hand, the PMS method increases the efficiency of dequeue as shown in Fig. 11. These facts suggested that access contentions from D-Ctrls occur in the PMS method. We consider that multiple D-Ctrls 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-Ctrls the access direction toward the same queues by referring to monitoring results, which increases the probability of access contention from D-Ctrls.

In contrast, in RDS, each D-Ctrl independently selects queues to dequeue by round-robin, which is comparable to D-Ctrls randomly selecting queues. Therefore, the RDS method increases the *hit-dequeue* rate as shown in Fig. 10 and the average number of messages per *hit-dequeue* as shown in Fig. 11 by waiting for dequeue requests in order to increase the probability of messages arrival at the message queues. In short, we show that our analysis of Eq. (4) is valid for the throughput degradation of message queues.

4.3 Evaluation and Discussion of Optimal Sleep Time for RDS

In Section 4.2, we showed the superiority of the RDS method. As mentioned in Section 3.2.2, sleep time of RDS determines the highest latency. Moreover, to increase the sleep time means limiting active connections between D-Ctrls and BES. 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 assumed the existence of an optimal sleep time to achieve maximum throughput without critical latency degradation. To investigate this assumption, we evaluated the relationship between sleep time and throughput for RDS method as described in **Fig. 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 observed a 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 Section 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 dequeue by waiting for dequeue request for sleep time in order to increase the probability of messages arrival at the message queues. However, the higher 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 is dependent on the queue status whether a queue has messages or not. We consider that sleep time of 5.0 s is a well-balanced value to obtain good values for both, efficiency of dequeue and throughput of dequeue request.

5. Experimental Evaluation

5.1 Implementation and Methodology for Evaluation

From simulation results in Section 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 evaluated its throughput. We evaluate RDS method in the real heterogeneous environment described in Section 2.1.2. We designed enqueue/dequeue communication based on *Representational State Transfer* (REST) [17] protocol and prepared message data based on text log data of an equipment monitoring services.

Figure 13 describes the environment of the experimental evaluation system with message queues. As shown in Fig. 13, we prepared a message queue having 10 sets of E-/D-Ctrls 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. Traffic test tools represent both field devices as message senders and BES as message receivers. 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.

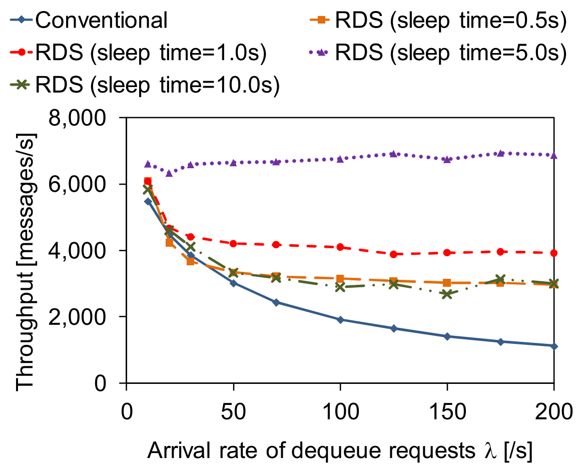


Fig. 12 Relationship between sleep time and throughput for conventional method (solid line) and RDS (dashed lines).

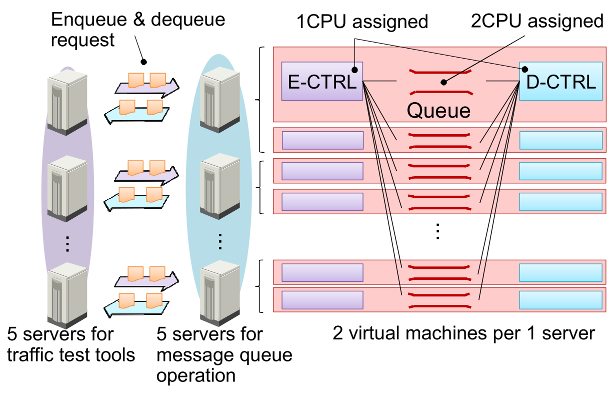

Enqueue & dequeue request
1CPU assigned
2CPU assigned
E-CTRL
Queue
D-CTRL
5 servers for traffic test tools
5 servers for message queue operation
2 virtual machines per 1 server

Fig. 13 Environment of experimental evaluation of a message queue.

Table 2 Setup of experimental evaluation.

Description	Value
Number of E-Ctrls/queues/D-Ctrls/BES B	10/10/10/10
Max. number dequeued msg/request N_{max}	100
Message size	1 KB
Max. number of dequeued message queues for single dequeue	3

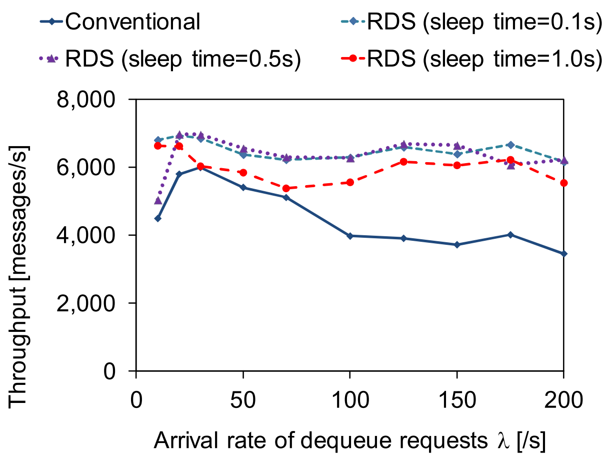


Fig. 14 Throughput comparison between conventional and RDS method on experimental evaluation. Several patterns of are set for RDS to investigate the optimal sleep time.

5.2 Result and Discussion

Figure 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 seconds contributes to 80 percent 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 seconds to 0.5 seconds. When the sleep time exceeds 1.0 seconds, we observe visible throughput degradation. This result strongly supports the assumption that excessive sleep time causes throughput degradation as explained in Section 4.2.

As a result, we reveal that the RDS method is effective to maintain high throughput of message queue systems even if the amount of dequeue requests from BES increases largely.

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 [26], our proposed system consists of multiple distributed message queues mesh-accessed by multiple servers.

There is a large number of publications on polling systems that have been developed since the late 1950s [27]. In several surveys, the most notable ones written by Takagi [26], detailed comprehensive descriptions of the mathematical analysis of polling systems are presented. Boon et al. [28] 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 few reports on polling methods, which have multiple queues with mesh-access from multiple servers as in our proposal. In this paper, we simulated the polling model, which has a client application putting messages onto these queues at regular intervals and back-end 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 BES at the timing when the message queue system receives messages from field devices. In the "pull" model, BES send dequeue requests to the message queue system and retrieve messages.

Generally, the "push" model is effective in the case when BES have sufficient computing resources to process messages sent by the message queue system. Jiang et al. [29] indicate that "push" service can be faster and more energy efficient for BES because in this approach BES do not need to look up a message queue or synchronize periodically.

On the other hand, the "pull" model is effective in the case when consumers make full use of its computing resources to process messages and it is frequently used in cloud computing systems [4], [5], [6]. Kreps et al. [4] also mentioned that the "pull" model is more suitable for their applications since each consumer obtains some advantages: sustainability of retrieving the messages at the maximum rate and avoidance of message flooding by being pushed faster than it can handle. Therefore, our pull-based proposal has advantages to achieve high throughput of message processing for fully utilizing computational resources of BES.

7. Conclusion

For the IoT era, message queue systems are required for interoperability and control of the huge message traffic between devices and BES. In this paper, 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 transmission 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 evaluation results show that only RDS maintains highest throughput, regardless of an increase in the dequeue request rate.

Experimental evaluation results show that the RDS method achieves 80 percent 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.

References

[1] Gantz, J. and Reinsel, D.: The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east, *IDC iView: IDC Analyze the Future* 2007, pp.1–16 (2012).

[2] Castro, M., Jara, A.J. and Skarmeta, A.F.: An analysis of M2M platforms: challenges and opportunities for the Internet of Things, *Proc. Innovative Mobile and Internet Services in Ubiquitous Computing* (*IMIS*), pp.757–762 (2012).

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

[4] Kreps, J., Narkhede, N. and Rao, J.: Kafka: A distributed messaging system for log processing, *Proc. Networking Meets Databases* (*NetDB*) (2011).

[5] Amazon Kinesis, available from ⟨http://aws.amazon.com/kinesis/⟩ (accessed 2016-05).

[6] Familiar, B.: IoT and Microservices, *Microservices, IoT, and Azure*, pp.133–163 (2015).

[7] Kinoshita, M., Tsuchida, G., Mizutani, I. and Koike, T.: High-throughput messaging system based on in-memory KVS for processing large traffic volume of short message, *IEICE Trans. Commun.*, Vol.B96, No.10, pp.1206–1216 (2013).

[8] Kinoshita, M., Tsuchida, G. and Koike, T.: High-throughput mail gateways for mobile e-mail services based on in-memory KVS, *Proc. Wireless and Mobile Communications* (*ICWMC*), pp.146–153 (2012).

[9] Kinoshita, M., Takada, O., Mizutani, I., Koike, T., Leibnitz, K. and Murata, M.: Improved resilience through extended KVS-based messaging system, *IEICE Trans. Inf. Syst.*, Vol.98, No.3, pp.578–587 (2015).

[10] Yang, Z., Peng, Y., Yue, Y., Wang, X., Yang, Y. and Liu, W.: Study and application on the architecture and key technologies for IOT, *Proc. Multimedia Technology* (*ICMT*), pp.747–751 (2011).

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

[12] Zhao, J., Zheng, X., Dong, R. and Shao, G.: 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 (2013).

[13] Humble, J. and Molesky, J.: Why enterprises must adopt devops to enable continuous delivery, *Cutter IT Journal*, Vol.24, No.8, p.6 (2011).

[14] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S. and Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale, *Proc. Symposium on Operating Systems Principles* (*SOSP*), pp.423–438 (2013).

[15] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S. and Bhagat, N.: Storm@ twitter, *Proc. 2014 ACM SIGMOD International Conference on Management of Data*, pp.147–156, ACM (2014).

[16] Hunkeler, U., Truong, H.L. and Stanford-Clark, A.: MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks, *Proc. Communication Systems Software and Middleware and Workshops* (*COMSWARE*), pp.791–798 (2008).

[17] Fielding, R.T.: Architectural styles and the design of network-based software architectures, PhD Thesis. University of California, Irvine (2000).

[18] Bormann, C., Castellani, A.P. and Shelby, Z.: Coap: An application protocol for billions of tiny internet nodes, *IEEE Internet Computing*, Vol.16, No.2, p.62 (2012).

[19] Huang, S., Chen, Y., Chen, X., Liu, K., Xu, X., Wang, C., Brown, K. and Halilovic, I.: The next generation operational data historian for IoT based on Informix, *Proc. ACM SIGMOD International Conference on Management of Data* (*ICMD*), pp.169–176 (2014).

[20] Porter, M.E. and Heppelmann, J.E.: How smart, connected products are transforming competition, *Harvard Business Review*, Vol.92, No.11, pp.64–88 (2014).

[21] Kibara, D., Morris, K.C. and Kumaraguru, S.: Methods and Tools for Performance Assurance of Smart Manufacturing Systems, NIST.IR.8099 (2015).

[22] Niyato, D., Xiao, L. and Wang, P.: Machine-to-machine communications for home energy management system in smart grid, *IEEE Communications Magazine*, Vol.49, No.4, pp.53–59 (2011).

[23] Meijer, G., Makinwa, K. and Pertijs, M.: *Smart sensor systems: Emerging technologies and applications*, John Wiley & Sons (2014).

[24] Dierks, T. and Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246 (2008).

[25] Riley, G.F. and Henderson, T.R.: The ns-3 network simulator, *Modeling and Tools for Network Simulation*, pp.15–34, Springer Berlin Heidelberg (2010).

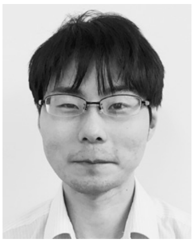
[26] Takagi, H.: Queuing analysis of polling models, *ACM Computing Surveys*, Vol.20, No.1, pp.5–28 (1988).

[27] Mack, C.: 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 (1957).

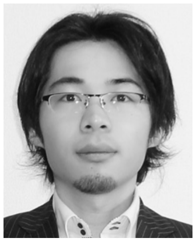
[28] Boon, M.A.A., van der Mei, R.D. and Winands, E.M.M.: Applications of polling systems, *Surveys in Operations Research and Management Science*, Vol.16, No.2, pp.67–82 (2011).

[29] Jiang, P., Bigham, J., Bodanese, E. and Claudel, E.: Publish/subscribe delay-tolerant message-oriented middleware for resilient communication, *IEEE Communications Magazine*, Vol.49, No.9, pp.124–130 (2011).

Masafumi Kinoshita received his M.E. degree in mechanical engineering from Nagoya University, Japan in 2002. He joined the System Development Laboratory, Hitachi, Ltd., Japan in 2002. He is currently a Senior Researcher at the Center for Technology Innovation, Hitachi, Ltd. His research interests are network architectures for enterprise and carrier systems, especially messaging systems. He is a member of IPSJ and IEICE.



Hiroaki Konoura received his M.E. and Ph.D. degrees in information systems engineering from Osaka University, Japan, in 2011 and 2014, respectively. He is currently a Researcher at the Center for Technology Innovation, Hitachi, Ltd. His research interests include high-reliable and high-throughput messaging systems supporting social infrastructure and high-reliable circuit design. He is a member of IEEE and IEICE.



Takafumi Koike received his B.S degree in computer science and engineering from the University of Texas at Arlington, U.S. in 2005. He joined the Network Solution Department, Hitachi, Ltd., Japan in 2005. He is currently an Engineer of an Innovation Solution Development Department, Hitachi, Ltd. His fields of work are on network and communication software, especially for messaging systems.

Kenji Leibnitz received his master and Ph.D. degrees in information science from the University of Würzburg in Germany. In May 2004, he joined Osaka University, Japan, as a Postdoctoral Research Fellow and from July 2006 he was a Specially Appointed Associate Professor at the Graduate School of Information Science and Technology. Since April 2010 he is a Senior Researcher at the National Institute of Information and Communications Technology (NICT), as well as a Guest Associate Professor at Osaka University and from April 2013 he is with the Center of Information and Neural Networks (CiNet) of NICT and Osaka University. His research interests are in modeling and performance analysis of communication networks, especially the application of biologically and brain inspired mechanisms to self-organization in future networks.

Masayuki Murata received his M.E. and D.E. degrees in information and computer science from Osaka University, Japan, in 1984 and 1988, respectively. In April 1984, he joined Tokyo Research Laboratory, IBM Japan, as a Researcher. From September 1987 to January 1989, he was an Assistant Professor with Computation Center, Osaka University. In February 1989, he moved to the Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University. In April 1999, he became a Professor of Cyber media Center, Osaka University, and he is now with the Graduate School of Information Science and Technology, Osaka University, since April 2004. He has more than five hundred papers of international and domestic journals and conferences. His research interests include computer communication network architecture, performance modeling and evaluation. He is a member of IEEE, ACM and IEICE. He is a chair of IEEE COMSOC Japan Chapter since 2009. Also, he is now partly working at NICT (National Institute of Information and Communications Technology) as Deputy of New-Generation Network R&D Strategic Headquarters.