Regular Paper

Comparative Evaluation of Dataflow Component Selection Methods in Distributed MQTT Broker Environment

Shintaro Ishihara[1,a)] Kazuma Yasuda[1] Kota Abe[2] Yuuichi Teranishi[3]
Toyokazu Akiyama[1,b)]

Received: March 10, 2021, Accepted: September 9, 2021

Abstract: Internet of Things applications often require reducing the communication delay and the traffic between sensors and actuators. In addition, research and development of dataflow platforms is ongoing. In these platforms, to meet the aforementioned requirements, geographically distributed dataflow components should be connected appropriately using edge computing environments. Existing approaches provide efficient communication considering the geographical distance using a distributed publish/subscribe broker that uses the peer-to-peer overlay; however, they do not consider resource information. In this paper, we propose two component selection methods – Multicast and Anycast – for inter-component communication considering resource information. Multicast selects a component by collecting resource information before selection, while Anycast selects a component using the aggregated resource information together with the overlay maintenance. We evaluated the hop count and amount of traffic using each method. As a result, we clarified that Anycast provides a smaller number of hops than Multicast when the aggregated values are sufficiently updated or there are sufficient available components. Furthermore, we examined how to use Anycast and Multicast considering the traffic volume against the sending interval of the component reservation request and the interval between sending the update query for maintaining the overlay. The sender node can choose the component selection method based on the number of hops and the traffic volume.

Keywords: edge computing, Pub/Sub, structured overlay network

1. Introduction

To process massive amounts of data generated by Internet of Things (IoT) devices, various dataflow applications have been developed. However, there are several challenges in the operation phase. One challenge is increased running costs including the cost of traffic to and from the cloud, while another challenge is unsatisfied application delay requirements due to large communication delays between sensors and actuators. To solve these problems, leveraging edge computing environments has become important. For example, the amount of traffic and delay can be reduced by processing part of data generated from IoT devices in an intermediate node before passing the data through the cloud (**Fig. 1**).

Research and development of a dataflow platform that supports dataflow application development is ongoing [1], [2], [3]. These projects exploit edge computing environments and have the potential to solve the above-mentioned challenges. However, most are strongly coupled with a specific cloud service, and it is difficult to construct a large-scale platform across multi-domain networks. In contrast, in Ref. [4], the authors utilized open source software to construct a platform while avoiding vendor lock-in. By adopting this approach, a platform can be constructed over multiple edge/cloud environments. To solve the dataflow application challenges in this platform, a component placement method considering delay and traffic volume should be provided.

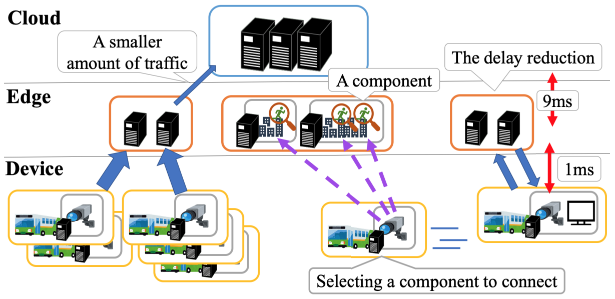


Fig. 1 Example of inter-component selection on a dataflow platform in edge computing.

The authors of Refs. [5], [6], [7] proposed component placement methods considering application requirements, such as the delay and the amount of traffic. All of these methods require resource information (e.g., computational load of a computing node) of target clusters where the application components are located. In Ref. [5], the authors assumed that the platform consists of an edge server cluster in radio access networks. They achieved appropriate component placement by real-time monitoring in each edge server. In their target cluster, the number

1 Graduate School of Frontier Informatics, Kyoto Sangyo University, Kyoto 603–8047, Japan
2 Graduate School of Engineering, Osaka City University, Osaka 558–8585, Japan
3 National Institute of Information and Communications Technology, Koganei, Tokyo 184–8795, Japan
a) shintaro.stonefield@gmail.com
b) akiyama@cc.kyoto-su.ac.jp

of edge servers was limited, and the resource information of all nodes could be obtained. In Ref. [6], the authors proposed a service deployment method targeting a specific private edge/cloud. Although their target cluster was larger than that in Ref. [5], they assumed centralized management and required resource information of the entire network. The authors of Ref. [7] targeted geographically spread applications, such as a passenger counting application for a bus service [8]. The target clusters of the component placement are clusters located around the bus route; however, the range of clusters changes for each route. To collect resource information in an appropriate range, flexible resource collection is necessary.

PIQT [9], which uses peer-to-peer (P2P) overlay technology, has two functions that can be used to efficiently collect resource information. The first is a function that aggregates multiple response messages of a multicast message for efficient response collection, which we refer to as a *resource information request*. The second is the ability based on Ref. [10] to use aggregated resource information collected by a resource information request for efficient message routing. Based on these functions, we propose two component selection methods: Multicast and Anycast. The Multicast method simply collects information of all available components by response aggregation, and then selects one component. Although the delay is increased by sending requests to all nodes, reliable delivery can be guaranteed. In the Anycast method, because the aggregation messaging involves the aggregated values updated with maintenance messages, the method can reach the appropriate component with few hops. However, in this method, selection failure may occur depending on the accuracy of the aggregated values. Furthermore, since the maintenance messages for aggregation messaging are larger than the original overlay messaging, increased traffic becomes a concern. To apply the two methods to component selection, the messaging delay and amount of traffic must be investigated considering the characteristics of the aggregated values.

The contributions of this paper are as follows:

- We investigate the influence of the accuracy of the aggregated values on component selection.
- We measure the message size to obtain aggregated values.
- We investigate the applicability of the Anycast and Multicast methods.

2. Dataflow Platform

In this section, we describe the assumed dataflow platform considering edge computing environments. In Section 2.1, we present an overview of the dataflow platform based on Ref. [7], and then clarify the functions required for inter-component communication. In Section 2.2, we describe two methods to realize the required functions.

2.1 Inter-component Communication Requirements

We first present a dataflow application based on Ref. [7]. The dataflow application can be represented as a dataflow graph, and **Fig. 2** presents an example of such a graph. Each c_i ($i \in N$) represents the type of component that constitutes the dataflow application. The dataflow graph may contain several dataflow applications. Figure 2 presents two examples of dataflow applications for improving a bus service using drive recorder data. The first application detects the number of people getting on and off the bus. In this application, each person is detected from the video data captured by the drive recorder, and the number of people is counted and stored in the database. The second application detects dangerous driving and notifies it to the driver. These sample applications can be represented as the set of c_0 to c_5 in Fig. 2. Hereafter, we assume the former application and describe the deployment and connections of components constructing that application in the dataflow platform considering edge computing environments.

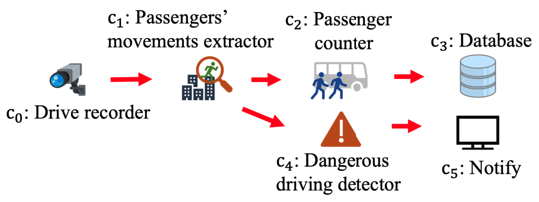


Fig. 2 Example of dataflow.

In this paper, a three-layer network is assumed as the edge computing environment, and the three network layers are the cloud network, edge network, and device network. The cloud network connects servers in a cloud computing environment located at a distant data center. The edge network connects servers deployed in a point of presence (POP), while the device network connects IoT devices. The device and edge networks have a lower network delay than cloud networks; however, devices have limited computational resources, and the components are thus distributed geographically considering the network delay, limited computational resources, and other factors.

A method to deploy the components in an appropriate network layer considering the required response time and resources was proposed in Ref. [7]. The authors of Ref. [7] also proposed an inter-component communication method. In this method, the components are connected via the distributed Message Queuing Telemetry Transport (MQTT) broker PIQT [9], which supports the standardized publish/subscribe (pub/sub) protocol MQTT [11]. PIQT uses a P2P structured overlay called PIAX [12] for communication between brokers deployed in each network layer. Hereafter, one PIQT broker is represented as a broker, and its broker ID is represented as B_i ($i \in N$). An example of component deployment based on the proposed method for the assumed dataflow application is illustrated in **Fig. 3**. Brokers (B_i) are placed into the computing resources of all network layers, and the processes of the components (P_c_i) are then placed into each computing resource located at the appropriate network layer. P_c_i connects with B_i deployed in the same computing resource or located in the same network layer, and communicates with other P_c_i via B_i using a topic-based pub/sub. To connect all components deployed in the various network layers, at least one broker must be deployed in each network layer. However, if brokers connect with each other without considering the geographical distance, they may route the message through a broker deployed at a distant location.

To limit the local communication inside each cluster [13], proposes a clustering method that groups the nodes of an overlay

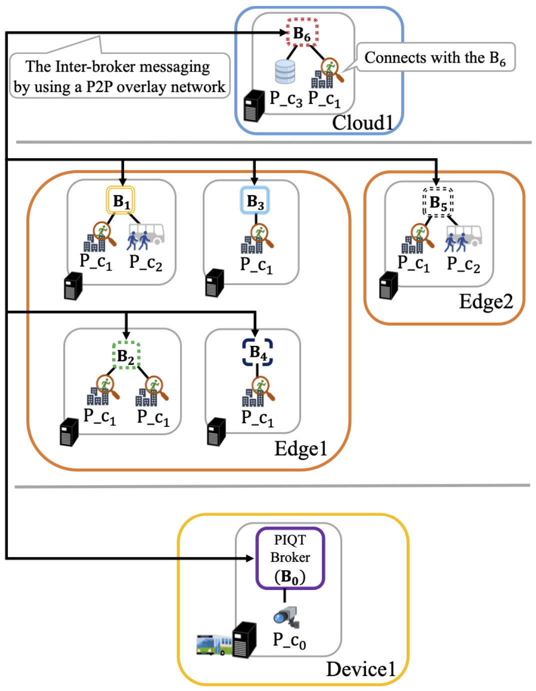


Fig. 3 Inter-component communication utilizing PIQT brokers.

by utilizing geographical information. For example, brokers located in the same POP or nearby POPs are grouped and labeled as Edge1, Edge2, and so on, and the topics in the overlay are sorted by the labels in terms of adjacency. Here, a multi-level hierarchy can also be defined as Edge1/EdgeA and Edge1/EdgeB, and the target range of the cluster can be defined flexibly for deployed brokers [13]. However, in this paper, a single hierarchy, such as Edge1 and Edge2, is used for simplicity. Communication considering such labels is also supported in PIQT and is assumed in the following discussion.

As shown in Fig. 3, we assume that multiple components of the same type are deployed in the proposed system. It enables resource adjustment considering dynamic changes of required resources. For example, in the bus application, component c_1 in Fig. 2 should be allocated to each bus, however, preparing one component per bus causes inefficient resource allocation. For resource efficiency, we would like to deploy several c_1 (the number is the maximum number of busses servicing simultaneously) and dynamically allocate one of them to a bus in service. The allocation should be done considering the bus route and its current location. For example, a component in Edge1 should be allocated to a bus close to Edge1. Furthermore, if we assume an on-demand bus service, the demands related to Edge1 change more dynamically depending on the conditions (e.g., weather and bargain sale in a supermarket), and therefore, the above-mentioned deployment and allocation strategy becomes important.

Another problem arises in the proposed inter-component communication method when a dataflow application is deployed for multiple devices and the same type of component is used in multiple dataflow applications. In the proposed method, components communicate with each other using a topic, which is the component name. Since the topic-based pub/sub transmits a message to all nodes subscribing to one topic, when multiple components subscribe to the same topic, the pub/sub cannot transmit the message to a single component. This can be resolved by attaching a unique ID, such as a source device ID, or a data stream ID, to each component when the device chooses one from candidate components. Then, the components can communicate with each other publishing/subscribing to the unique topic (e.g., "the component name + the source device ID"). However, since the chosen components are monopolized by the source device ID when the component does not output data frequently, always reserving a component wastes resources, and this application should instead use a shared component. To consider such resource sharing, applications should be categorized by the processing style of the data stream. One type of application handles data frames using the previous data frames (e.g., comparison of captured video data for detecting a movement of persons). For this type of application, because the data stream must be delivered to the same component, the application should perform reservation-based processing, in which a data stream is delivered after reserving the component by a component reservation request. Another type of application handles each data frame independently. In this case, each data frame is delivered to any shared components. Both types of applications require a method to select one component. If a message can be delivered to a component considering the resource status, the delivery method can be used for both application types. Thus, in the following discussion, we assume the case of reservation-based processing.

2.2 Proposed Methods to Select One Component

As mentioned in Section 1, we propose two component selection methods: Multicast and Anycast. PIAX provides the Suzaku overlay proposed in Ref. [14]. As the Suzaku overlay supports aggregating values, such as CPU usage, and provides message routing based on the aggregated values, a message can be routed to the component with minimum CPU usage. The Anycast method has the potential to cause fewer message hops than the Multicast method. However, in this method, sending a component reservation request before the update query causes component selection errors due to insufficient updates of the aggregated values. In other words, the performance of the Anycast method depends on the accuracy of the aggregated values and the interval of the component reservation request. In this paper, we investigate the characteristics of the Multicast and Anycast methods.

3. Component Selection Procedure

In this section, we provide a detailed description of the procedure of the Multicast and Anycast methods. Section 3.1 describes the relationship between the PIQT brokers and the overlay, while Section 3.2 explains how PIQT Brokers store the node information on the P2P overlay Suzaku, and then describes the multicast query utilizing the overlay and its efficiency. Section 3.3 describes the mechanism of the aggregation messaging provided by Suzaku, while Section 3.4 describes the component selection methods, Multicast and Anycast.

3.1 Assumed Environments

Figure 4 displays part of the overlay network created by

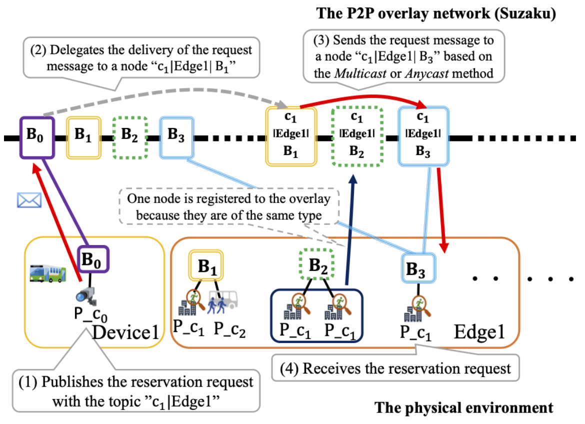

The P2P overlay network (Suzaku)
(2) Delegates the delivery of the request message to a node "c1|Edge1| B1"
(3) Sends the request message to a node "c1|Edge1| B3" based on the Multicast or Anycast method
B0
B1
B2
B3
c1 |Edge1| B1
c1 |Edge1| B2
c1 |Edge1| B3
One node is registered to the overlay because they are of the same type
B0
P_c0
Device1
B1
P_c1 P_c2
B2
P_c1 P_c1
B3
P_c1
Edge1
(1) Publishes the reservation request with the topic "c1|Edge1"
(4) Receives the reservation request
The physical environment

Fig. 4 Relationship between PIQT brokers and an overlay network.

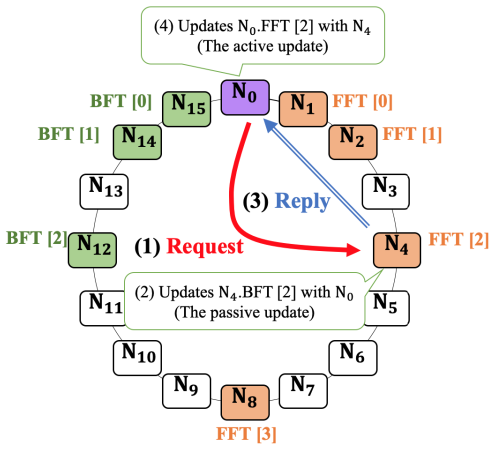


Fig. 5 Active and passive updates in Suzaku.

Suzaku in the physical environment of Fig. 3. Each broker has multiple nodes in the overlay. For example, the overlay presented in Fig. 4 consists of nodes indicating brokers, such as B_3. Broker nodes with subscribers, such as c_1|Edge1|B_3, are represented as {topic}|{cluster name}|{broker ID}. The cluster name and broker ID are automatically added by a broker at the subscription of the components. In Fig. 4, two P_c_1 are connected with B_2; however, because they are the same type of component, only one c_1|Edge1|B_2 is registered to the overlay. Here, B_2 can correctly obtain the status of P_c_1 because they are directly connected. When B_2 receives a message, B_2 delivers it to an appropriate P_c_1 considering the status. In the following, we focus on component selection in the overlay. It should be noted that c_0 is not registered in the overlay because it only publishes and does not subscribe.

The inter-component messages in Fig. 3 are actually delivered via the overlay network, as illustrated in Fig. 4. In Ref. [13], the authors proposed the *delegate* method, in which message delivery is delegated to a node located in the target cluster, and messages are then propagated from the delegated node. This method can be used to prevent an increase in inter-cluster traffic. Because the PIQT broker supports the delegate method, we also deliver the component reservation request using this method. Figure 4 presents an example in which P_c_0 publishes the component reservation request to a node c_1|Edge1|B_1 with the topic c_1|Edge1 via a broker B_0. First, node B_0 in the overlay owned by the broker B_0 delegates the processing of the request to c_1 in Edge1 using the delegate method. The delegated node delivers the request to an appropriate node in the overlay with the Multicast or Anycast method. Then, the node receiving the request delivers it to the component via the MQTT broker embedded within the PIQT broker. In the following, we describe how to manage nodes in the Suzaku overlay and how to deliver messages between nodes.

3.2 Inter-broker Messaging in the Suzaku Overlay

In this subsection, we describe how to store a component name as a topic name and how to deliver messages to the components subscribing to the topic. We first explain the node management of the Suzaku overlay and then describe the message delivery utilizing the query provided by the overlay.

Suzaku is a key-order preserving structured overlay network that is an extended version of Chord# [15]. Each node has a key, which is an element of a totally ordered set, and the nodes are arranged as a ring structure preserving the order of the keys. Each node also has a successor, which is a pointer to the node with the next largest key, and a predecessor, which is a pointer to the node with the next smallest key. To preserve the ring structure, the successor of the node with the largest key points to the node with the smallest key, and the predecessor of the node with the smallest key points to the node with the largest key.

Furthermore, to improve the efficiency of key search, each node has two finger tables (FTs). A FT is an array of multiple pointers to nodes located at a distance of a power of two away. One FT is a forward finger table (FFT), which stores node pointers in the clockwise direction, while the other is a backward finger table (BFT), which stores node pointers in the counterclockwise direction. In the following, the key, FFT, and BFT owned by node u are represented as u.key, u.FFT, and u.BFT, respectively. Here, let u.FFT[i], u.FFT[i].node, and $u.getFFT(i)$ be the i-th element of u.FFT, the node pointed to by u.FFT[i], and the function to obtain the i-th FFT of a remote node u, respectively. Then, as indicated in Eq. (1), u.FFT[i] is defined as in Ref. [10], which extends Chord#. Letting $k, n \in \mathbb{N}$ be the maximum of i and the total number of nodes, k becomes $\lceil \log_2 n \rceil - 1$.

$$u.\mathrm{FFT}[i] = \begin{cases} u.\mathrm{node} & (i = -1) \\ \mathrm{Successor} & (i = 0) \\ u.\mathrm{FFT}[i-1].\mathrm{node}.getFFT(i-1) & (i > 0) \end{cases} \quad (1)$$

To maintain the FTs, the node sends update queries to the nodes in the FTs when joining/leaving the ring. In addition, each node regularly sends update queries to detect node failures and fix the FT. When a node joins the ring, it sends update queries in the following order: FFT[0], BFT[0], FFT[1], BFT[1], . . . , FFT[k], BFT[k]. When a node leaves the ring, it sends update queries to nodes with a pointer to itself. In the case of regular updates, each node sends update queries to FFT nodes in the following order: FFT[0], FFT[1], . . . , FFT[k]. There are two types of updates: active and passive updates. In an active update, the node updates its own FT utilizing the reply to the update query. In a passive update, the FT of the node receiving the update query is updated. Two examples of active and passive updates in Suzaku are presented in **Fig. 5**. Figure 5 presents the ring when n is set to 16.

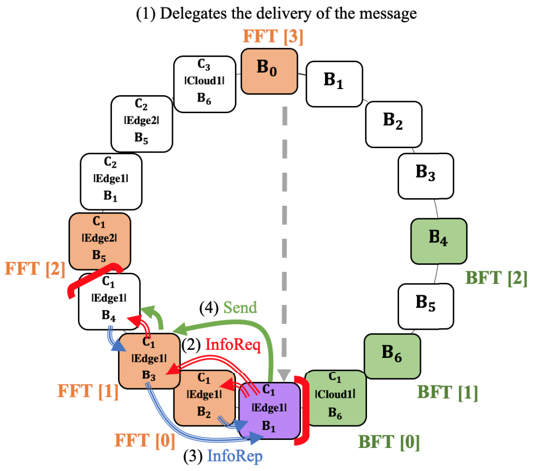


Fig. 6 Example of Multicast method.

Each node has N_i as a key and is connected in dictionary order of the key. For example, when N_0 updates FFT[2], N_0 sends an update query to N_4, and then uses the reply to the query to update N_0.FFT[2]. This is an active update. In contrast, a passive update is performed when the node receives the update. For example, N_4 updates N_4.BFT[2], which points to the sender node N_0, using the information included in the received update query.

In Suzaku, messages are delivered by utilizing the range query, as in Chord$^\#$. An example of delivering a message from N_0 to the nodes $N_2, N_3, \ldots, N_8$ in Fig. 5 is explained as follows. First, N_0 generates a range of [N_2.key, N_9.key), which signifies $N_2.\text{key} \leq \text{key} < N_9.\text{key}$. N_0 divides the generated range based on the keys of the nodes included in N_0.FT, and then sends a range query to each node specifying the divided range. In this case, the ranges provided to N_2, N_4, and N_8 are [N_2.key, N_4.key), [N_4.key, N_8.key), and [N_8.key, N_9.key), respectively. These are performed recursively, and the message is finally delivered to all nodes in the search range. The reply to the query traces back the delivery path of the query to N_0 while aggregating replies in each range.

Now, we describe message delivery between PIQT brokers using the range query of Suzaku. If brokers and components are deployed as illustrated in Fig. 3, the overlay presented in **Fig. 6** is constructed. As described in Section 3.1, a broker is registered as a key B_i, where $B_{\min} < B_i < B_{\max}$. c_1|Cloud1|B_6, c_1|Edge1|B_1, c_1|Edge1|B_2, and so on presented in Fig. 6 indicate the keys in the overlay. The nodes in the overlay are arranged with the keys in dictionary order arranged by topic, cluster name, and broker ID.

An example of PIQT messaging is provided below, where a message is delivered from B_0 to the nodes with c_1 located at Edge1. First, B_0 generates [{c_1|Edge1|$B_{\min}$}.key, {c_1|Edge1|$B_{\max}$}.key) as a search range. Then, it delegates the range to c_1|Edge1|B_1. The delegated node divides the range to [{c_1|Edge1|B_2}.key, {c_1|Edge1|B_3}.key) and [{c_1|Edge1|B_3}.key, {c_1|Edge1|$B_{\max}$}.key), and then sends range queries to c_1|Edge1|B_2 and c_1|Edge1|B_3. The node c_1|Edge1|B_3 recursively sends a range query to c_1|Edge1|B_4 with a range of [{c_1|Edge1|B_4}.key, {c_1|Edge1|$B_{\max}$}.key). Then, the receiving brokers recursively deliver the message to the directly connected components.

Table 1 FT of c_1|Edge1|B_1.

FT[i]	Aggregation range	Aggregation labels: Aggregated values
FFT[−1]	[{c_1|Edge1|B_1}.key, {c_1|Edge1|B_2}.key)	c_1|Edge1 : 1
FFT[0]	[{c_1|Edge1|B_2}.key, {c_1|Edge1|B_3}.key)	c_1|Edge1 : 1
FFT[1]	[{c_1|Edge1|B_3}.key, {c_1|Edge2|B_5}.key)	c_1|Edge1 : 2
FFT[2]	[{c_1|Edge2|B_5}.key, B_0)	c_1|Edge2 : 1 c_2|Edge1 : 1 c_2|Edge2 : 1 c_3|Cloud1 : 1
FFT[3]	[B_0, {c_1|Edge1|B_1}.key)	c_1|Cloud1 : 1
BFT[0]	[{c_1|Cloud1|B_6}.key, {c_1|Edge1|B_1}.key)	c_1|Cloud1 : 1
BFT[1]	[B_6, {c_1|Cloud1|B_6}.key)	null
BFT[2]	[B_4, B_6)	null

3.3 Aggregated Values Maintained in Each Finger Table (FT) of Suzaku Nodes

Aggregated values are the values in each FT[i], which is maintained by Suzaku nodes. A specified attribute value is aggregated with a specified function. The node range to be aggregated by each FT[i] is [FT[i].node.key, FT[$i+1$].node.key). The overhead to aggregate values is smaller than querying because the aggregation is performed along with overlay ring maintenance. In other words, after a FT update, the aggregation results are cached in the FT. In the following, we describe the mechanism for maintaining aggregated values.

In the following notation, u.value represents the specified attribute value of node u, FFT[i].range represents the aggregation range of FFT[i], and FFT[i].value is a cached value of the aggregated values. FFT[i].range is [FFT[i].node.key, FFT[$i+1$].node.key), and contains 2^i nodes clockwise from FFT[i].node.key. BFT[i].range is [BFT[i].node.key, BFT[$i-1$].node.key), and the number of nodes is 2^{i-1} (1 if $i = 0$). As an example, **Table 1** presents the aggregation ranges and aggregated values in the FT of node c_1|Edge1|B_1 in Fig. 6. In Table 1, the number of components connected to the broker is a target value for the aggregation, and each node aggregates the number of components labeled {topic}|{cluster name}. FFT[-1] points to itself, and the aggregation range only contains its own node. Therefore, u.FFT[-1].value = u.value. Here, the accuracy of the aggregated values depends on the frequency of FT updates. For example, when an active update of {c_1|Edge1|B_1}.FFT[2] is executed, the node c_1|Edge1|B_1 (requester) generates the aggregation range [FFT[0].node.key, FFT[1].node.key), and then prepares aggregated values of FFT[-1], FFT[0], and FFT[1] for a passive update of the node c_1|Edge2|B_5 (responder). Then, the requester sends an update query including the range and the aggregated values to the responder. The responder receives the query and sends an update query reply including the aggregated values to the requester. At the same time, the responder c_1|Edge2|B_5 updates the aggregated values of {c_1|Edge2|B_5}.BFT[2] based on the update query.

The accuracy of the aggregated values should be discussed to evaluate their effectiveness. When an update query of FFT[k] is sent, the responder must prepare aggregated values FFT[0], FFT[1], . . . , FFT[$k-1$]. Whereas FFT[-1] always contains the latest value, the others, such as FFT[0], FFT[1], . . . , depend on the status of the FT of the node pointed to by FFT[i]. Here, let T seconds be the interval between sending update queries. One node takes $(k+1)T$ seconds to update all aggregated values after a target value change at a certain node since all the aggregated values of FFT must be updated. Furthermore, to update the values correctly, the aggregated values of the other nodes to be used must also be correctly updated. For example, if {c_3|Cloud1|B_6}.value is changed, to update {c_1|Edge1|B_1}.FFT[2] correctly, the aggregated values {c_1|Edge2|B_5}.FFT[0] and {c_1|Edge2|B_5}.FFT[1] must be updated. From the above, at worst, $(k+1)^2 T$ seconds are required to correctly update all aggregated values of all nodes after a target value change at a given node. For example, if T is set to 60, which is the default value of PIAX, and the number of nodes is 100, it will take more than 1 hour to correctly update all aggregated values. T must be set appropriately depending on the accuracy requirements of the use case.

3.4 Component Selection Methods

In this subsection, we present two methods to select a component: the Multicast method, which aggregates the component information by utilizing a range query described in Section 3.2, and the Anycast method, which utilizes aggregated values described in Section 3.3. Here, to reduce the number of wasted hops, we also use the delegate method proposed in Ref. [13]. In both methods, a sender first delegates the selection to the delegated node in the search range, and the delegated node then selects a component based on each component selection method. Hereafter, the delegated node is called a *sender*.

3.4.1 *Multicast* Method

In the Multicast method, the sender collects the resource information affecting the components using a range query, and then delivers the request to the appropriate component considering the corrected information. Figure 6 presents the procedure in which c_1|Edge1|B_1, which is delegated the component reservation request from B_0, collects the candidate list of c_1 located at Edge1, and transmits the request to the selected node. First, c_1|Edge1|B_1 sends a resource information request with the range [{c_1|Edge1|B_{min}}.key, {c_1|Edge1|B_{max}}.key) and receives an aggregated reply. Then, it sends a component reservation request. In the following, the resource information request and its reply are called *InfoReq* and *InfoRep*, respectively. In the Multicast method, although the sender must send *InfoReq* to all candidate nodes once, the sender can select the appropriate component reliably.

3.4.2 *Anycast* Method

In Ref. [10], the authors proposed a conditional multicast delivering the message to nodes that match conditions utilizing aggregated values. By extending the conditional multicast, we propose the Anycast method, which sends the message to only one node selected from the matching nodes.

Algorithm 1 presents the procedure of the Anycast method. Variables r, $r_{\min}$, and e represent the search range, the minimum key of the search range, and an element of FT, respectively. The MATCH function returns a Boolean value based on whether *e.value* meets the conditions. SELECTONE, without definition here, has the selection logic to select an element from multiple candidate elements, and can be switched for different purposes. Simple strategies to select a proper node based on the number of components are described in Section 4.1.

Algorithm 1 Anycast method using aggregated values

```
▷ $r$: a search range
▷ $r_{min}$: minimum key in $r$
▷ MATCH: function to return Boolean true if value meets the conditions
▷ SELECTONE: function to select one from candidate nodes
▷ $e$: one element of FT
function CONDANYCAST($r$, MATCH, SELECTONE)
    candidateElems ← [$e$|($e \in$ FT)∧ MATCH(e.value) ∧ (e.range ∧ $r \neq \emptyset$)]
    if candidateElems is not empty then
        selectedElem ← SELECTONE(candidateElems)
        if selectedElem.node is FFT[−1].node then
            ▷ Finish selection
        else
            selectedElem.node. CONDANYCAST((selectedElem.range ∧ $r$), MATCH, SELECTONE)
        end if
    else
        $r$ ← initializes a range
        FFT[−1].node. CONDANYCAST($r$, MATCH, SELECTONE)
    end if
end function
```

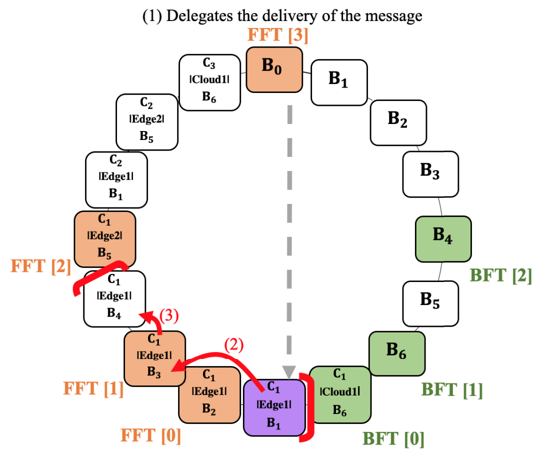


Fig. 7 Example of Anycast method.

First, the sender node s calls s.CONDANYCAST(r, MATCH, SELECTONE). *candidateElems* is the list of elements in FT satisfying MATCH, and ($e.range \wedge r \neq \emptyset$). *selectedElem* is an element selected from *candidateElems* by SELECTONE. If the selected node is the current node *self* (*self*.FFT[-1]), it finishes the selection. Otherwise, the current node lets the *selectedElem.node* call the CONDANYCAST() with a narrower r, which is an intersection of *selectedElem.range* and r. The message is delivered recursively and is finally received by a single node that satisfies the conditions. If there is no candidate element with a candidate component, the selection is considered a failure, and the current node retransmits from itself by initializing the range.

Figure 7 presents an example of delivery with the Anycast method in the same environment as in Section 3.2. The conditions specified by the MATCH function here are that the aggregation labels contain the group c_1|Edge1|. In this example, SELECTONE is defined to select a node with larger aggregated values for the label (i.e., with a large number of candidate nodes). From Table 1, which presents the FTs of the delegated node c_1|Edge1|B_1, FFT[0].value has one candidate, and FFT[1].value has two candidates. In this case, the FFT[1].node is selected ((1) in Fig. 7). Then, the selected node FFT[1].node also calls CONDANYCAST(), and then, c_1|Edge1|B_3 or c_1|Edge1|B_4 is selected ((2) in Fig. 7). Here, if FT elements have the same number of candidate components, one of them is randomly selected. Furthermore, if multiple nodes call condAnycast() simultaneously, there is a possibility that the delivery may be biased to a certain node, which may cause a reservation conflict. To avoid this, we adopt a weighted random algorithm in selectOne. The algorithms are described in detail in Section 4.1.

Figure 7 demonstrates that the message can be delivered in two hops. This signifies that the Anycast method can reduce the number of hops to half of that of the Multicast case. However, the performance of the Anycast method depends on the accuracy of the aggregated values. When the values are insufficiently updated or the selection logic is not suitable for the aggregated values, the performance degrades. To verify the effectiveness of the proposed method, it is necessary to clarify the influence of the accuracy of the aggregated values and the selection logic on the performance. Here, CPU usage and memory usage should be used as aggregated values to select the appropriate node based on the resource usage. However, in this paper, we focus on the influence of the accuracy of the aggregated values on the performance and evaluate the Anycast method by simply using the component availability information as the aggregated values.

There is a case in which the performance of Anycast decreases when the aggregated values are not sufficiently updated. Initially, all components are available; however, as the reservation proceeds, the number of remaining candidate nodes decreases. If the aggregated values are not sufficiently updated, the actual number of remaining candidate nodes may differ from the aggregated values. In this case, the selection is more likely to fail. In Section 4, we investigate the change in the number of hops for the number of nodes contained in the search range. The total number of nodes in the overlay also affects the number of hops. In our proposal, the Anycast method delivers the messages after using the delegate method. Therefore, it only affects the first delegation, and the performance is estimated as normal Suzaku messaging. For this reason, the following evaluation focuses on message delivery from the delegated node (sender) to the nodes in the search range.

4. Comparison of Proposed Component Selection Methods

In this section, we measure the number of hops and the amount of traffic using our proposed methods and then consider the applicability of these methods. In Sections 4.1 and 4.2, we present the assumption of aggregated values and the details of the selection logics. In Sections 4.3 and 4.4, the evaluation environment and procedure are described. The number of hops for the number of nodes contained in the search range according to the selection logics is described in Section 4.5. Then, the effect of the accuracy of the aggregated values on the performance is described in Section 4.6. Furthermore, in Section 4.7, we discuss the amount of increased traffic with an increase in the message size of the update query to update the aggregated values. Finally, in Section 4.8, we compare the two methods and discuss their applicability.

4.1 Assumptions and Investigated Points of Aggregated Values

In this paper, we adopt the number of available components as the target aggregation attribute, as in Ref. [16]. As mentioned, the number of hops of messaging using the Anycast method depends on the accuracy of the aggregated values. In Ref. [16], the authors clarified that the number of hops converges sufficiently when the transmission interval of the component reservation request T_w is longer than $(k + 1)T$. However, cases in which T_w is between 0 and $(k + 1)T$ have not been examined with sufficient granularity, and it is difficult to judge the delay required for an application with a shorter T_w. Thus, in this paper, T_w is set to the range $0 \le T_w \le (k + 1)T$. Although shortening the update query interval T may satisfy the delay requirements of applications with shorter T_w, it also increases the traffic. Therefore, T should be set appropriately considering both the delay requirement and the amount of traffic.

4.2 Details of Selection Logics

In this subsection, we describe the details of the selection logics. The selection logics is used in a function selectOne in Algorithm 1. There may be several strategies to select one element from the FT. We consider four strategies: *Random*, *Many*, *Few*, and *Close*. *Random* selects one element randomly. In Algorithm 1, the logic selects one element from *candidateElems*, which are elements of the FT filtered by match. However, to investigate the number of hops without using aggregated values, the *Random* strategy is prepared. In this case, match does not filter candidates for the strategy. *Many* selects one element using an algorithm similar to the one described in Section 3.4.2. The difference is the weighted random algorithm in selectOne illustrated in Algorithm 2. While the FT element selected by *Many* may have many candidate components in the aggregation range, it tends to select an element with a larger aggregation range, and as a result, the candidates reside in distant nodes.

In contrast to *Many*, *Few* selects an element with a smaller value with priority, and the logic tends to select the element with a smaller aggregation range and send a message with fewer hops. *Close* selects the element pointing to the closer node in the overlay. Since the aggregated values of close nodes are more fre-

Algorithm 2 Weighting algorithm for each selectOne

```
▹ candidateElems: stores filtered FT elements and keeps the element order of FT
function setWeight(strategy, candidateElems)
    if strategy is Many then
        totalNum ← sum(e.value | e ∈ candidateElems)
        for e in candidateElems do
            e.weight ← e.value/totalNum
        end for
    end if
    if strategy is Few then
        maxNum ← max(e.value | e ∈ candidateElems)
        for e in candidateElems do
            e.weight ← (maxNum + 1) − e.value
        end for
        totalNum ← sum(e.weight | e ∈ candidateElems)
        for e in candidateElems do
            e.weight ← e.weight / totalNum
        end for
    end if
    if strategy is Close then
        maxLevel ← max(candidateElems.level(e) | e ∈ candidateElems)
        for e in candidateElems do
            e.weight ← (maxLevel + 1) − candidateElems.level(e)
        end for
        totalNum ← sum(e.weight | e ∈ candidateElems)
        for e in candidateElems do
            e.weight ← e.weight / totalNum
        end for
    end if
    return candidateElems
end function
```

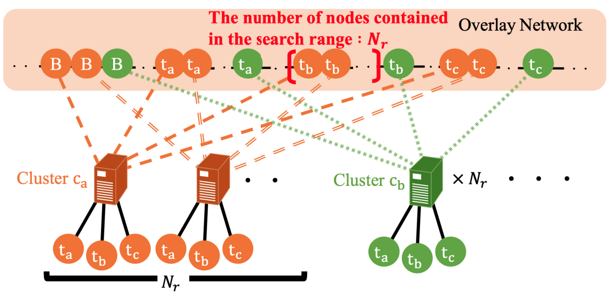


Fig. 8 Overview of the evaluation environment.

quently updated than those of distant nodes, selecting a closer element leads to using more accurate aggregated values, leading to a reduction of the number of hops.

As mentioned, we adopt a weighted random algorithm in selectOne, and the method of weighting the elements is different in *Many*, *Few*, and *Close*. In *Random*, a weighted random algorithm is not introduced. Algorithm 2 illustrates the setWeight function used in selectOne. The logic of *Many* weights the elements in a decreasing order of the aggregated values. In contrast, the logic of *Few* weights the elements in an increasing order. This is realized by subtracting the aggregated values from the maximum aggregated values plus 1. Here, if we call i of FT[i] as the level, the logic of *Close* weights the elements in an increasing order of the level of the FT.

4.3 Evaluation Environment

The network layer of the component location is determined by the component placement method proposed in Ref. [7]. Dataflow components of bus applications are geographically distributed. To connect the components located at various locations and to support local communication in each area, brokers should also be geographically distributed. In this paper, we assume that brokers are evenly deployed in all clusters. Although the construction is simple, it enables an easy analysis of the basic properties of the aggregated values and the selection logics. In this evaluation, we prepare three types of components based on an example of a dataflow application; that is, the number of topics is three. **Figure 8** presents an overview of the evaluation environment.

In Fig. 8, as explained in Fig. 4, each broker can have only one topic in one cluster in the overlay. To increase the number of nodes with the same key (N_r) (i.e., the same {topic}||{cluster name}) in the overlay, the number of brokers with the same key (which also becomes N_r) should be added. N_r represents a search range in which the proposed delivery methods search candidate components. Since the size of N_r greatly affects the performance of the delivery methods, we change N_r in the evaluation. Although each broker can have different topics in a real environment, we assume a simple configuration in which every broker has the same three topics. We focus on analyzing the search behavior for a specified key prefix {topic}||{cluster name} with a specified range size N_r, and omit the variation of the topics here. With this assumption, each cluster has the same number of brokers, N_r. If the total number of brokers is given as N_b, the number of clusters N_c is as follows:

$$N_c = \frac{N_b}{N_r}. \tag{2}$$

Since we simply assume that each broker has the same three topics, each broker registers four nodes to the overlay, including a broker node. Thus, the total number of nodes in the overlay N_n can be represented as

$$N_n = 4N_b. \tag{3}$$

From the above, N_c and N_n can be calculated from N_b and N_r. In the following, we evaluate the performance of proposed delivery methods while changing N_b and N_r.

Here, we construct the overlay using the P2P framework PIAX. PIAX code is designed to work in a large-scale distributed environment. However, verifying our proposal with the code requires many computing resources. PIAX implements a message level event simulator without the use of a network [14], and we evaluate the proposed algorithm using the simulator.

4.4 Evaluation Scenario

In this subsection, we describe the evaluation scenarios. As mentioned, we focus on the reservation request of a specified type of component and measure the number of hops for the reservation. As described in Section 3.4, we assume that a delegated node of a component reservation request is a sender node. The *number of hops* in the evaluation targets the message exchanges after the delegation. As described in Section 3.3, the component reservation status is used as the aggregated values in the evaluation. Status 0 signifies occupied, while 1 signifies available. The component status is registered to a broker, and the summation of the status is provided as a target attribute of the aggregated values.

In the evaluation, one trial involves sending component reservation requests until all components with a specified topic are reserved based on the component selection methods described in Sections 3.2 and 3.4.2. Before starting a trial, all aggregated values are updated by waiting for $(k+1)^2T$ seconds. In the Multicast method, since *InfoReq* is sent to all nodes in the search range and *InfoRep* traces back to the sender node while aggregating the results, the paths of the query and reply construct a tree. The number of edges in the tree traced during the query is measured as the total number of hops in the query phase of the Multicast method. Since these are sent in parallel, the height of the tree is measured as the maximum number of hops. Thereafter, the number of hops in the reservation phase of the Multicast method is also measured. In Section 4.5, to evaluate the delay due to the number of hops, the number of hops in the Multicast method is calculated as the sum of the maximum number of hops in the query phase and the number of hops in the reservation phase. In Section 4.7, to evaluate the amount of traffic, the sum of the total number of hops in the query phase and the number of hops in the reservation phase are used. The number of trials is 50, and the number of hops for the two methods is measured while changing the parameters presented in **Table 2**.

In addition, for comparison, we evaluate a method with the minimum number of hops, which is called the *Optimal* method. In this method, we obtain the status of all nodes before evaluation, and the sender then reserves all nodes in the order of the

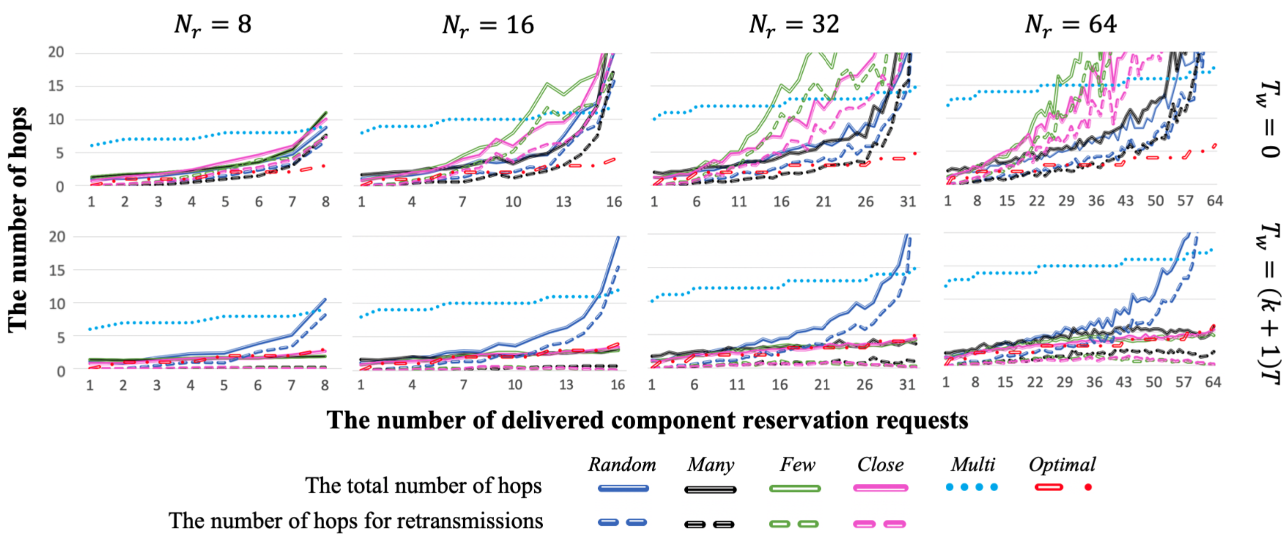


Fig. 9 Transition in hops due to changes in N_r.

Table 2 Parameters.

Parameter types
Sending interval of the component reservation request: T_w
Number of nodes contained in the search range: N_r
Number of brokers: N_b

smallest number of hops using the information. In the following, we compare the number of hops between the *Optimal*, Multicast, and Anycast methods.

4.5 Number of Hops for the Selection Logics and N_r

In this subsection, we first measure the number of hops for the selection logics while changing N_r, and investigate the influence of N_r. Thereafter, to investigate the differences in the number of hops between the selection logics, we analyze the utilization rate of each FT element.

Figure 9 presents the measurement results (where N_b = 128). In each graph, the vertical axis indicates the number of average hops, while the horizontal axis indicates the number of delivered component reservation requests. The legends in Fig. 9 represent the total number of hops and the number of hops for retransmissions in each selection logic. The top and bottom rows illustrate $T_w = 0$ and $T_w = (k + 1)T$ results. N_r is set to 8, 16, 32, and 64 by the power of two as in the FT from the left to right columns. Here, $T_w = 0$ signifies that no aggregated values are updated. In this case, we measure the number of hops without the update.

The results indicate that when using the Anycast method, the number of hops increases with the increase in N_r, and the $T_w = 0$ results show a greater increase than $T_w = (k+1)T$. By comparing the retransmissions between $T_w = 0$ and $T_w = (k+1)T$, the $T_w = 0$ results demonstrate larger hops. This appears to be caused by insufficient updates of the aggregated values. When $T_w = 0$, the number of hops in the latter requests, where few available components remain, is larger than the Multicast and *Optimal* methods. In contrast, even if the accuracy of the aggregated values is low, the request can reach the target node with few hops when there are sufficient available nodes. The *Close* and *Few* strategies have this characteristic. In contrast, when $T_w = (k + 1)T$, the results of *Few* and *Close* demonstrate similar hops to *Optimal*.

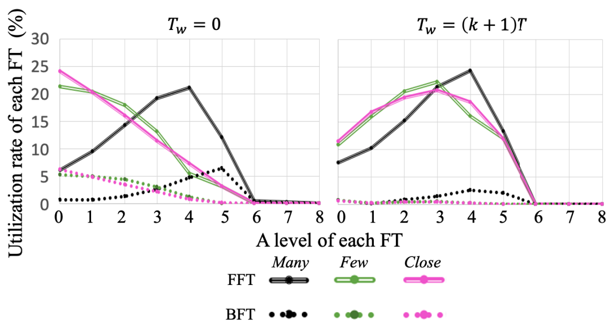


Fig. 10 Utilization rate of each level of finger table (FT) for selection logic ($T_w = 0, (k + 1)T$, N_r = 64).

As mentioned, the aggregated values of the FT with a higher level (i) are updated slowly. We believe that this causes the differences in the number of hops among the selection logics. In **Fig. 10**, we analyze the utilization rate of each FT node for the selection logics with the above measurement results in the case of $T_w = 0, (k + 1)T$ (N_r = 64). In each graph, the vertical axis displays the utilization rate of each FFT and BFT node, while the horizontal axis displays the level of FFT or BFT used in the selection logic. The rates are calculated by dividing the average usage of each FT element in the 50 trials by their sum.

In $T_w = 0$, *Few* and *Close*, which demonstrate a larger number of hops, tend to use the FT of the low level frequently, as described in Section 4.2. Since the FT of the low level has a narrow aggregation range, the number of candidate elements decreases. Therefore, retransmissions frequently occur by using incorrect aggregated values, and the number of hops is also increased. It is possible to reduce the number of hops by caching occupied FT elements based on the reservation results in the implementation; however, this extension is left for future work. In contrast, *Many*, which demonstrates a small number of hops, uses the FT with a higher level because it has a larger aggregate range. This signifies that even if it transmits the reservation request to the FT with a higher level using invalid aggregated values, there are alternative candidates, and it thus enables fewer retransmissions and hops.

In $T_w = (k + 1)T$, all logics tend to use FFTs rather than BFTs.

Few and *Close* use higher levels around FFT[3] compared to the result in $T_w = 0$. It is thought that the logics do not use the FT of the low level, which is already occupied, and the status is reflected in the cache by sufficient updates of the aggregated values. In contrast, *Many* has a similar utilization rate in $T_w = 0$ and $T_w = (k+1)T$; however, it has a larger number of hops. Since it delivers to nodes with more candidates, even if the selected node is an available node, the number of hops is increased by repeatedly delivering to the FT with a higher level. The number of hops can be reduced by extending the algorithm; for example, if the current node satisfies the conditions, it finishes the selection.

Random has the same number of hops even if T_w is changed, as illustrated in Fig. 9. Since the *Random* method does not use the aggregated values, the tendency is not changed. This method does not have a bias to choose an element, and the number of hops can be lower than in *Few* and *Close* when $T_w = 0$. In contrast, if $T_w = (k+1)T$, the selection logics utilizing the aggregated values has a small number of hops.

The BFT utilization rate is significantly lower than that of the FFT in Fig. 10. In this evaluation, since we assume using the delegate method, the sender is always located at the left end of the search range based on the logic in Ref. [13]. Thus, the sender always uses the FFT as long as it does not fail the selection. When the selection fails, the node that failed the selection becomes the sender and retransmits the request. However, in this case, the sender is not located at the left end, and it is possible to use the BFT. Therefore, the utilization rate of the BFT in $T_w = 0$ is higher than $T_w = (k+1)T$ because the number of retransmissions is larger in $T_w = 0$.

The above results indicate that when sufficient updates are performed, *Few* and *Close* can reach a node with an available component with few hops. *Few* simply uses the node with a narrow range but does not always use the neighbor node. Therefore, in the following, we focus on the only *Close* method whose logic is to select the closest node in the overlay, and we investigate the case in which the Anycast method provides better performance than the Multicast method.

Here, for both the FFT and BFT, if the level is higher than 6, the FT is not used, as illustrated in Fig. 10 (where $N_r = 64$). This indicates that when the level is higher than $\log_2 N_r$, the FFT is not used. Therefore, the change in N_b, which affects the size of the FT, has no influence on the number of hops for the Anycast and Multicast methods.

4.6 Number of Hops for the Accuracy of Aggregated Values

In this subsection, to investigate how the accuracy of the aggregated values affects the performance of the Anycast method, we measure the number of hops while changing T_w, especially in the range $0 \leq T_w \leq (k+1)T$.

Figure 11 presents the average number of hops when using the *Close* logic with the following parameters: $N_r = 64$, $N_b = 128$ and $T_w = 0, T, 2T, \ldots, 8T, 9T$. The vertical axis represents the average number of hops, while the horizontal axis represents the elapsed time. The results indicate that if $T_w = (k+1)T$ $(= 9T)$, it takes $64(k+1)T$ $(= 576T)$ seconds to reserve all components. The results also indicate that a shorter T_w increases the number

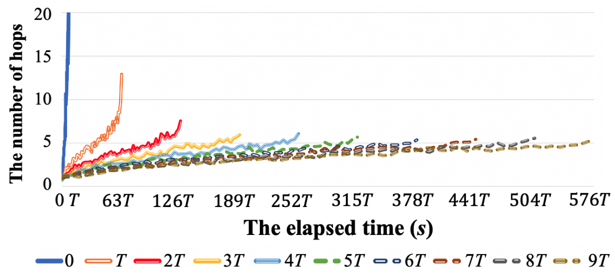
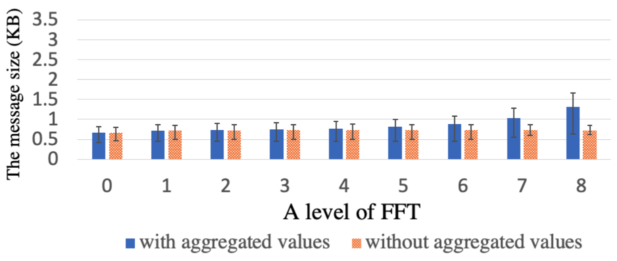
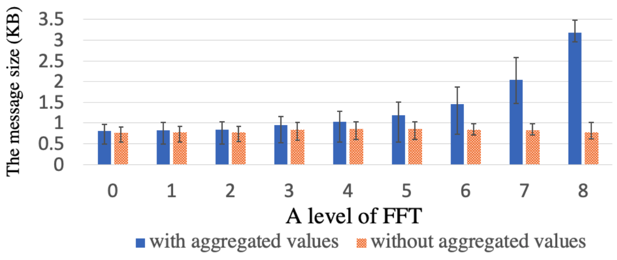


Fig. 11 Number of hops for changes in T_w when using the *Close* logic (N_r = 64, N_b = 128).



(a) Message size of the update query



(b) Message size of the update query reply

Fig. 12 Update message size comparison with/without aggregated values.

of hops. When $T_w = T$, the number of hops to reserve all components is lower than the result using the Multicast method presented in Fig. 9. This indicates that the aggregated values to be used are sufficiently updated and that the Anycast method is more effective than the Multicast method when $T_w \geq T$.

However, when using the Anycast methods, the overlay must support aggregated values, and it increases the amount of overlay maintenance traffic. In the following, to clarify the amount of traffic, we investigate the message size to be added for maintaining the aggregated values.

4.7 Increased Total Message Size of the Update Query and Reply of Each FT Element for Maintaining Aggregated Values

The message for updating the aggregated values of each FT is included in the update query and the update query reply. As described in Section 3.3, since an FT element with a higher level has a wider aggregation range, it requires a larger update message. In this subsection, we measure the message size of the update query and the reply generated during $(k+1)^2T$ seconds in the scenario described in Section 4.3, where $N_b = 128$ and $N_r = 8$.

Figure 12 (a) and (b) present the message size of the update query and the reply for a level of FFT. The vertical axes represent the average message size of each message sent and received in $(k+1)^2T$ seconds, while the horizontal axes represent the level of FFT. Error bars indicate the maximum and minimum size. The blue bar represents the size with aggregated values, and the red bar represents the size without aggregated values. As explained, the number of nodes included in the aggregation range increases with an increase in the level of the FT. Since the proposed method aggregates all values of the nodes in the aggregate range, the more the level of the FT increases, the more the number of aggregated label and value sets increases. As a result, the message size is increased.

As explained in Section 3.2, the update query of the FT element is sent with the interval T in the following order: FFT[0], ... FFT[k]. Thus, the amount of traffic generated by a node in $(k+1)T$ $(=9T)$ seconds can be calculated by summing all of the message sizes from 0 to 8 in Fig. 12 (a) and (b) (where $k = 8$). The total message size with aggregated values becomes approximately 20.00 KB. By averaging the size, the amount of update traffic generated by a node per second becomes 37.04 B/s (where $T = 60.00$). Similarly, the total message size without the aggregated values becomes approximately 13.80 KB, and the amount of update traffic generated by a node per second becomes 25.56 B/s (where $T = 60.00$).

The larger message size increases the communication charges of carriers. As described in Section 4.5, the FT element that has a level higher than $\log_2 N_r$ is not used for the reservation request delivery. For example, by implementing a function to skip aggregation of the FT with a level higher than $\log_2 N_r$, the amount of traffic can be reduced. This extension is left for future work.

4.8 Consideration of Component Selection Methods

Since both the Anycast and Multicast methods have advantages and disadvantages, the component selection method should be chosen according to the condition. In this subsection, we discuss the condition focusing on the sending interval of the component reservation request.

We calculate the amount of traffic based on the measurements in the evaluation to compare the two methods. First, we describe the variables used in the calculation. Let H_a be the total number of hops to deliver component reservation requests to all nodes in the Anycast method, and let H_m be the number of hops in the Multicast method described in Section 4.4. Here, since all message sizes are approximately the same as the size of the update query without aggregated values, for simplicity of calculation, all message sizes are represented as S. The total amount of traffic for the component reservation request in Anycast and Multicast can be calculated by H_aS and H_mS.

We calculate the amount of traffic while changing T_w as 0, T, $2T$, $3T$ in the *Close* logic of the Anycast method and in the Multicast method (where $S = 800.0$ bytes, $N_r = 64$, $N_b = 128$). To calculate H_a and H_m, the average number of hops measured in Fig. 9 and Fig. 11 is used. **Table 3** presents the calculation results of H_aS, H_mS.

From H_aS in Table 3 (a), since the number of hops decreases, the amount of traffic for the reservation also decreases with an increase in T_w. From H_mS in Table 3 (b), the number of hops does not depend on T_w, and the amount of traffic for the reservation is constant. This result demonstrates that even if $T_w = 0$, H_aS is smaller than H_mS because the Multicast method sends the component information request to all candidate nodes. Therefore, the amount of traffic for the reservation in Anycast is smaller than in Multicast.

Table 3 Comparison of the amount of traffic for the reservation.

(a) Amount of traffic for T_w and N_r in H_aS.

N_r \ T_w	0	T	$2T$	$3T$
64	920.5 KB	283.3 KB	194.8 KB	182.1 KB

(b) Amount of traffic for T_w and N_r in H_mS.

N_r \ T_w	0
64	6,450 KB

Table 4 Comparison of the amount of traffic generated per second.

(a) Amount of traffic for T_w and N_r in A_{total}.

N_r \ T_w	T		$2T$		$3T$	
	U_a	H_aS/T_wN_r	U_a	H_aS/T_wN_r	U_a	H_aS/T_wN_r
8	18.92	36.51×10^{-3}	18.92	13.76×10^{-3}	18.92	78.59×10^{-4}
16	18.92	46.89×10^{-3}	18.92	18.11×10^{-3}	18.92	10.84×10^{-3}
32	18.92	56.75×10^{-3}	18.92	21.83×10^{-3}	18.92	13.19×10^{-3}
64	18.92	62.08×10^{-3}	18.92	25.37×10^{-3}	18.92	15.81×10^{-3}

(b) Amount of traffic for T_w and N_r in M_{total}.

N_r \ T_w	T		$2T$		$3T$	
	U_m	H_mS/T_wN_r	U_m	H_mS/T_wN_r	U_m	H_mS/T_wN_r
8	13.08	20.18×10^{-2}	13.08	10.19×10^{-2}	13.08	67.27×10^{-3}
16	13.08	41.67×10^{-2}	13.08	20.83×10^{-2}	13.08	13.89×10^{-2}
32	13.08	84.03×10^{-2}	13.08	42.01×10^{-2}	13.08	28.01×10^{-2}
64	13.08	16.80×10^{-1}	13.08	83.98×10^{-2}	13.08	55.99×10^{-2}

As described in Section 4.7, to maintain the aggregated values, the message size of the update query is increased in the Anycast method. To investigate the amount of increased traffic, we calculate the amount of traffic for update queries for both methods. In this calculation, we introduce new variables in addition to the conditions in Table 3. The amount of traffic generated by update queries per second in the Anycast and Multicast methods is represented as U_a and U_m, respectively. U_a and U_m can be calculated using the amount of update traffic measured in Section 4.7 and N_n. Here, the total amount of traffic per second in Anycast, A_{total}, and Multicast, M_{total} can be represented as follows (where $T = 60.00$ and $k = 8$):

$$U_a = 37.04N_n \tag{4}$$

$$U_m = 25.56N_n \tag{5}$$

$$A_{\text{total}} = U_a + \frac{H_aS}{T_wN_r} \tag{6}$$

$$M_{\text{total}} = U_b + \frac{H_mS}{T_wN_r} \tag{7}$$

With the conditions in Table 3, we calculate the amount of traffic while changing N_r to 8, 16, 32, and 64. **Table 4** presents the calculation results of A_{total}, M_{total}.

Table 4 presents the amount of traffic for the reservation and update queries per second separately, while A_{total} and M_{total} are the sum of the two. As indicated in Table 4, U_a and U_m are dominant in A_{total} and M_{total} in every case. In other words, almost all traffic is generated by update queries, and the total amount of traffic in Anycast is larger than in Multicast when $T_w \geq T$.

Whereas the update query and reply target all nodes in the overlay (N_n), the reservation request targets only one key prefix {topic}||{cluster name}, as described in Section 4.3. If we assume that the reservation requests are sent to all other key prefixes simultaneously, the number of hops for each key prefix becomes similar, as discussed in Section 4.5, and the reservation traffic increases with the number of key prefixes. For example, the reservation traffic becomes 16 (the number of clusters) $\times$ 3 (the number of topics) times larger when $N_r = 8$. In this situation, when T_w is approximately the same as T, Anycast produces better results than Multicast. The results indicate that since Anycast provides a smaller number of hops than Multicast when $T_w \geq T$, the component selection method can be chosen at the sender node based on the importance of the traffic volume using the estimated values. However, when $T_w < T$, both Multicast and Anycast increase the traffic volume and the delay compared to the other situations, and the additional condition should be considered. The details are discussed below.

Although the Anycast method can reduce the number of hops, it increases them in a situation with few available components; thus, the Multicast method should be used. In contrast, even if $T_w < T$, the Anycast method can deliver the component reservation request with few hops when there are sufficient available components. For example, Fig. 9 ($N_r = 64$ and $T_w = 0$) demonstrates that when there are over 30 available components, the number of hops of Anycast (*Close*) is smaller than Multicast. From this consideration, switching the selection method based on the remaining components may be efficient, where the Anycast method is used at the beginning and the Multicast method is used when the number of available components is reduced. However, it is currently difficult to determine the switching condition without assuming more concrete application requirements, deployed environments, and reservation strategies. In future work, it is necessary to investigate the number of hops when $T_w < T$ and to consider a method that combines the Anycast and Multicast methods for practical environments.

5. Related Work

In Reference [17], the authors also targeted dataflow applications, and a load-balancing method among the components subscribing to the same topic in the overlay network was proposed. In the proposed method, each component has an index range, and when one component publishes a message, it also specifies the index in addition to a topic. For example, a publisher can choose a scheduling strategy, such as Round Robin. However, the publisher can only distribute by the ratio of an index, and load balancing according to the resource status (e.g., CPU load and memory consumption) cannot be realized. We propose novel component selection methods using aggregated component information, and they have the potential to achieve load balancing considering the resource status.

P2P-based resource discovery methods include [18] and [19]. In Ref. [18], the authors proposed an efficient resource search method considering the geographical distance. In this method, the resource status, such as the CPU usage and memory usage of the nodes, is stored as a key of the overlay. This method can realize component selection considering the resource status; however, since such information is often changed, it causes the occurrence of frequently joining/leaving the overlay, which is a well-known churn problem. It is thus difficult to store all information as the key of the overlay. In our approach, this information is collected using aggregated values, and this method does not directly affect the overlay structure. In Ref. [19], the authors proposed an efficient service search method for constructing an application consisting of multiple services. In the method, each P2P node has a matrix of services that groups services frequently used together in an application. This matrix is updated by agents that are constantly moving between nodes. In our proposal, we use the structured overlay; however, the problem mentioned in Ref. [19], in which distributed hash tables cannot realize range queries, was solved in Suzaku. Furthermore, a more efficient range search can be achieved in Suzaku.

6. Summary and Future Work

In this paper, we aim to realize inter-component communication considering resource information in a dataflow platform over hierarchical networks. We assume two phases in component communication. The first phase is component reservation while the second phase is actual data transfer. We focus on component reservation considering resource information, and propose two component selection methods (i.e., Anycast and Multicast) utilizing P2P overlay technology.

The Multicast method is able to select an available component because it collects the resource status of the candidate components before requesting. In contrast, the Anycast method realizes an efficient component selection using the aggregated values updated during overlay maintenance. However, inconsistently aggregated values cause request retransmissions and increased hop counts. To determine how to choose component selection methods, we evaluated the number of hops and the amount of traffic, including the maintenance messages, focusing on the interval of the component reservation request.

The results elucidate the characteristics of the two component selection methods. Since Anycast provides a smaller number of hops than Multicast when T_w is approximately the same as T, the component selection method can be chosen at the sender node based on the importance of the traffic volume using the estimated values. If $T_w < T$, Anycast can still deliver the component reservation request with few hops when there are sufficient available components even if the aggregated values are not completely updated. However, in cases in which the available components are insufficient to avoid an increase in hops, Multicast should be used. To choose the selection method under the condition $T_w < T$, the number of remaining components should also be considered.

To optimize the component selection methods, the switching of the two methods should be considered as part of future work. To clarify the switching condition, the component reservation request under a higher transaction rate (e.g., $0 < T_w < T$) should be examined. Furthermore, reducing the update query traffic (e.g., removing the unnecessary payload of aggregated values) is also important in considering a mobile environment.

To realize load balancing using aggregated values, information on how to aggregate CPU usage, memory usage, disk space, and other factors should be considered. Such information is likely to be changed frequently; therefore, further investigation is required, such as an update interval and aggregation granularity.

In the current implementation, if there are no available components in the search range, the retransmissions are repeated. In actual operation, we should consider introducing TTL and other parameters to the Anycast method. For example, if the number of transmissions exceeds the TTL, it is necessary to switch the component selection method to Multicast to verify whether an available component remains. Because the total remaining components can be counted using the aggregated value cache, a time series forecasting method can be applied to detect resource shortage in advance. With such an extension, further improvements should be applied for use in a practical environment.

As described in Section 4.8, when the interval of the component reservation request decreases, the Anycast method creates a larger amount of traffic, and the number of applicable situations of the method is reduced. This is mainly caused by the large size of the FT update query reply message, which includes the entire aggregation labels and values. In our proposal, since the delegate method is adopted, only the aggregation labels and values inside the aggregation range should be collected. This filtering can reduce the size of the messages. The message size reduction and its evaluation is left for future work.

Acknowledgments This work was partially supported by JSPS KAKENHI Grant Numbers 20H04183 and 20H04186. The authors would like to thank Enago (www.enago.jp) for the English language review.

References

[1] Amazon Web Services, Inc.: Amazon Kinesis Data Streams (online), available from ⟨https://aws.amazon.com/kinesis/data-streams/⟩ (accessed 2019-11-09).

[2] Microsoft, Inc.: Azure Stream Analytics (online), available from ⟨https://azure.microsoft.com/services/stream-analytics/⟩ (accessed 2020-06-20).

[3] Google, Inc.: Cloud Dataflow (online), available from ⟨https://cloud.google.com/dataflow⟩ (accessed 2020-06-20).

[4] Satyanarayanan, M.: The Emergence of Edge Computing, *IEEE Computer*, Vol.50, No.1, pp.30–39 (2017).

[5] Zhang, W., Li, S., Liu, L., Jia, Z., Zhang, Y. and Raychaudhuri, D.: Hetero-Edge: Orchestration of Real-time Vision Applications on Heterogeneous Edge Clouds, *Proc. IEEE INFOCOM*, pp.1270–1278 (2019).

[6] Ascigil, O., Phan, T.K., Tasiopoulos, A.G., Sourlas, V., Psaras, I. and Pavlou, G.: On Uncoordinated Service Placement in Edge-Clouds, *IEEE International Conference on Cloud Computing Technology and Science* (*CloudCom*), pp.41–48 (2017).

[7] Ishihara, S., Tanita, S. and Akiyama, T.: A Dataflow Application Deployment Strategy for Hierarchical Networks, *2019 IEEE 43rd Annual Computer Software and Applications Conference* (*COMPSAC*), pp.15–19 (2019).

[8] Nakashima, H., Arai, I. and Fujikawa, K.: Passenger Counter Based on Random Forest Regressor Using Drive Recorder and Sensors in Buses, *IEEE International Conference on Pervasive Computing and Communications Workshops* (*PerCom Workshops*) (2018).

[9] PIQT, distributed pub/sub broker (online), available from ⟨http://www.piqt.org/⟩ (accessed 2020-06-20).

[10] Abe, K.: Proposal of "Conditional Multicasting" using Structured Overlay Networks, *Multimedia, Distributed, Cooperative, and Mobile Symposium* (*in Japanese*), pp.195–204 (2019).

[11] MQTT (online), available from ⟨http://mqtt.org/⟩ (accessed 2020-06-20).

[12] Yoshida, M., Teranishi, T., Harumoto, K. and Shimojo, S.: PIAX: A P2P Platform for Integration of Multi-Overlay and Distributed Agent Mechanisms, *IPSJ SIG Technical Reports 2006-DPS-128* (*in Japanese*), pp.43–48 (2006).

[13] Teranishi, Y., Banno, R. and Akiyama, T.: Scalable and Locality-Aware Distributed Topic-based Pub/Sub Messaging for IoT, *Proc. IEEE Global Communications Conference* (*GLOBECOM*), pp.1–7 (2015).

[14] Abe, K. and Teranishi, Y.: Suzaku: A Churn Resilient and Lookup-Efficient Key-Order Preserving Structured Overlay Network, *IEICE Trans. Communications*, Vol.E102-B, No.9, pp.1885–1894 (2019).

[15] Schutt, T., Schintke, F. and Reinefeld, A.: Range Queries on Structured Overlay Networks, *Computer Communications*, Vol.31, No.2, pp.280–291 (2008).

[16] Ishihara, S., Yasuda, K., Akiyama, T., Abe, K. and Teranishi, Y.: A Proposal of an Inter Dataflow Component Communication Method using Distributed MQTT Broker, *Multimedia, Distributed, Cooperative, and Mobile Symposium* (*in Japanese*), pp.1571–1580 (2019).

[17] Teranishi, Y., Kimata, T., Yamanaka, H., Kawai, E. and Harai, H.: Dynamic Data Flow Processing in Edge Computing Environments, *IEEE 41st Annual Computer Software and Applications Conference* (2017).

[18] Shen, H., Li, Z. and Zhu, Y.: PIRD: P2P-based intelligent resource discovery in Internet-based distributed systems, *Proc. IEEE ICDCS*, pp.858–865 (2008).

[19] Mastroianni, C. and Papuzzo, G.: A Self-Organizing P2P Framework for Collective Service Discovery, *Journal of Network and Computer Applications*, Vol.39, pp.214–222 (2014).

Shintaro Ishihara received his M.E. and Ph.D. degrees in Graduate school of Frontier Informatics from Kyoto Sangyo University, in Japan in 2018 and 2021, respectively. Currently, he is a Research Fellow of the same university. His research interests include Edge Computing and IoT.

Kazuma Yasuda received his M.E. degrees in Graduate school of Frontier Informatics from Kyoto Sangyo University, in Japan in 2019 and 2021, respectively.

Kota Abe received his M.E. and Ph.D. degrees from Osaka University, Japan, in 1994 and 2000, respectively. In 1994, he joined Nippon Telegraph and Telephone Corporation, Japan. In 1996, he started working as a Research Associate with the Media Center, Osaka City University, Japan, where he has been a Professor with the Graduate School of Engineering, since 2018. Also, since 2015, he has been a Cooperative Visiting Researcher with the National Institute of Information and Communications Technology. His research interests include distributed systems and system software. He received the Information Processing Society of Japan Best Paper Award in 2013.

Yuuichi Teranishi received his M.E. and Ph.D. degrees from Osaka University, Japan, in 1995 and 2004, respectively. From 1995 to 2004, he was engaged Nippon Telegraph and Telephone Corporation. From 2005 to 2007, he was a Lecturer with the Cybermedia Center, Osaka University, where he was an Associate Professor with Graduate School of Information Science and Technology, from 2007 to 2011. Since August 2011, he has been a Research Manager with the National Institute of Information and Communications Technology and a Guest Associate Professor with the Cybermedia Center, Osaka University. His research interests include technologies for distributed network systems and applications. He received the IPSJ Best Paper Award in 2011.

Toyokazu Akiyama received his B.E., M.E. and Ph.D. degrees in Information Systems Engineering from Osaka University, Japan in 1997, 1999 and 2003, respectively. Since 2000, he worked as a Research Associate (Assistant Professor) at the Cybermedia Center, Osaka University. Currently, he is a Professor of the Faculty of Information Science and Engineering, Kyoto Sangyo University. His research interests include distributed systems and applications. He is a member of the IEEE Computer Society, IEICE.