IEICE TRANSACTIONS

on Communications

VOL. E103-B NO. 1
JANUARY 2020

A PUBLICATION OF THE COMMUNICATIONS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER *Special Section on Internet Architecture, Applications and Operation Technologies for a Cyber-Physical System*

Distributed Key-Value Storage for Edge Computing and Its Explicit Data Distribution Method*

Takehiro NAGATO†, **Takumi TSUTANO**††, **Tomio KAMADA**†a), ***Nonmembers***, **Yumi TAKAKI**†, ***and* Chikara OHTA**†††, ***Members***

SUMMARY In this article, we propose a data framework for edge computing that allows developers to easily attain efficient data transfer between mobile devices or users. We propose a distributed key-value storage platform for edge computing and its explicit data distribution management method that follows the publish/subscribe relationships specific to applications. In this platform, edge servers organize the distributed key-value storage in a uniform namespace. To enable fast data access to a record in edge computing, the allocation strategy of the record and its cache on the edge servers is important. Our platform offers distributed objects that can dynamically change their home server and allocate cache objects proactively following user-defined rules. A rule is defined in a declarative manner and specifies where to place cache objects depending on the status of the target record and its associated records. The system can reflect record modification to the cached records immediately. We also integrate a push notification system using WebSocket to notify events on a specified table. We introduce a messaging service application between mobile appliances and several other applications to show how cache rules apply to them. We evaluate the performance of our system using some sample applications.

key words: *edge computing, pub/sub system, distributed key-value storage, distributed cache*

1. Introduction

Recently, smart mobile devices (e.g., smartphones and tablets) have allowed us to use geolocation information in many kinds of applications, including transition services and augmented reality games. Intelligent Transportation System (ITS) and the Internet of Things (IoT) are opportunities to create real time applications that connect various sensor data to users or devices. Fog/edge computing [2] will become a powerful tool to process sensor data "on the fly" and deliver them quickly to subscribers. Edge servers are often used to offload tasks from IoT devices. Much research proposed efficient handling of task offloading in handover. In this research, we focus on how to deliver data to subscribers in motion. Developing such applications will be tedious since data delivery to mobile subscribers requires careful treatment to ensure data consistency. Moreover, the publish/subscribe relationship between devices and servers depends on the applications. A solution that suits the needs of a particular application may not be reusable for an other application whose requirements differ.

Our goal is to provide a data framework for edge computing where developers can easily attain efficient and immediate data transfer between mobile devices. The system should follow the publish/subscribe relationships between mobile clients to perform data transfers. To achieve this goal, we propose a distributed key-value storage platform for edge computing and its explicit data distribution management method for proactive cache placement. The system consists of several edge servers which organize a distributed key-value storage. Each server holds some records which are kept in a uniform namespace. A record is only assigned to a single server at a time. However, cached values can be kept on several other servers. The platform provides the mechanisms to monitor modifications and immediately reflect them on cached records. The system also supports mobility of records. Developers can define cache rules in a declarative manner to define the cache behavior of records depending on the record type or status. We use a schema to define the data structure of records and introduce some extensions to specify its cache rule.

In this article, we introduce some sample applications to demonstrate how cache rules apply to various requirements, namely, a messaging service, a group chat service, and a location-based game. We developed a test-bed system of our platform and evaluated its performance using these applications. Our prototype system offers a Web API for data creation, modification, and retrieval. It also offers WebSocket connections to send push notifications to subscribers. This paper is an extended version of work published in [1]. We present in more details the cache rule and add several applications and performance evaluation.

In Sect. 2, we summarize related works. We present our system's design in Sect. 3, introducing our cache mechanisms. Implementation details are given in Sect. 4. We introduce sample applications in Sect. 5 and evaluate the performance of our prototype in Sect. 6. We then conclude in Sect. 7.

2. Related Work

Many kinds of distributed database management system (e.g. AWS DynamoDB, Apache Cassandra, MongoDB) are

Manuscript received February 15, 2019.
Manuscript revised June 3, 2019.
Manuscript publicized August 5, 2019.

†The authors are with Graduate School of System Informatics, Kobe University, Kobe-shi, 657-8501 Japan.
††The author is with Dept. of Computer Science and System Engineering, Kobe University, Kobe-shi, 657-8501 Japan.
†††The author is with Graduate School of Science, Technology and Innovation, Kobe University, Kobe-shi, 657-8501 Japan.

*This paper is an extended version of work published in [1].

a) E-mail: kamada@fine.cs.kobe-u.ac.jp

DOI: 10.1587/transcom.2019CPP0007

used to achieve the scalability needed in cloud computing. These system's appeal lies in their performance and scalability as key-value storage rather than support of relational operations. They run on computer clusters in the data center but offer little support for locality-aware assignment of data contents between worker nodes.

Fog computing and Multi-access Edge Computing (MEC) assume geographically distributed servers, called fog nodes or MEC hosts, located close to clients. MEC applications are running as virtual machines (VMs) on MEC hosts [3]. When clients move around, there are two choices: maintaining connectivity between mobile clients and MEC applications or relocating application data and/or the application instance itself to VM hosts in appropriate locations. In order to reduce task offloading delay for mobile clients, Plachy el al. [4] exploit prediction of client movement. Bao et al. [5] introduces pre-migration mechanism of offloaded tasks. When the handover is expected from received signal srength (RSS) monitoring of neighoring hosts, the system starts pre-migration of application data or instance to the candidate hosts. These techniques exploit the relashionship between mobile clients and MEC applications, rather than the publish/subscribe relationships between mobile devices or clients. In MEC environments, various wired and wireless technologies are already used and more technologies will be introduced in the future. Liu el al. [6] propsed a software defined network (SDN) based architecture to separates control, traffic forwarding, and processing entities.

When multiple applications share the same data resources, applications benefit from state externalization. FogStore [7], [8] proposes a distributed data store for Fog/edge computing. It uses a distributed data store for cloud data centers as a base system, and introduces location-consious replica placement strategy using context (location) based index. It places some replicas on neighboring hosts to enable efficient quorum based queries, and also allocates remote replicas to provide tolerance from geographically-corelated failures. These works only considers the location context of data and do not support publish/subscribe relationships between remote clients.

Some work has been done to manage dynamic data provided by mobile IoT devices, offering fast queries and supports for mobile records. Auspice [9] is a name service to locate mobile devices. It is designed as a scalable distributed key-value storage and can allocate replica data on hosts in a demand-driven manner. Kafle et al. proposed an architecture for an IoT directory service [10]. It allows general data structures and supports on-demand caching of records. The system can reflect record modification to the cached records immediately. It also allows data queries with various search keys. They have plans to support proactive data propagation, the details of which remain unclear.

To treat data in motion, some database systems support stream processing [11], [12]. These systems provide continuous query that receives input data stream of sensor events or user action logs and outputs on-the-fly reaction. StreamSpinner [13] offers a query system that continuously gathers events from distributed sensor nodes, conducts the queries over distributed servers using stream processing and delivers the result to the subscribers. AWS lambda [14] is the other type of computing service relying on stream processing. It offers lambda functions that are invoked when certain events occur in specified services (e.g. storage or database services). Developers can register their functions that can check the event's context and act on the cloud data services, potentially triggering other lambda functions. The system scales automatically without explicit server management or provisioning. These systems for stream processing offer little support for subscription of mobile users or devices.

Publish/subscribe systems are also used to realize stream processing between multiple services. MQTT [15] is a protocol enabling machine-to-machine IoT communication. Messages are categorized into topics. A message broker receives messages with their topics from publishers and delivers them to the subscribers of each topic. MQTT assumes there is a unique broker per topic and does not support distributed broker situation. In cloud computing, scalable pub/sub systems are used to connect multiple servers with low latency [16]–[18]. Apache Kafka [16], which is developed by LinkedIn, can allocate multiple brokers for each topic and supports consumer groups. Each record published to a topic is delivered to one consumer in a group. This system focuses on scalability but offers limited support for locality-aware data transfer.

3. Platform Design

3.1 System Overview

An overview of our platform is presented in Fig. 1. Our system consists of several edge servers, distributed across places. Each server holds a part of the data records in a uniform namespace. There can be multiple *tables* (e.g., `user`, `station`) in the namespace. Each *record* in a table is represented in JavaScript Object Notation (JSON) format. A record has its information stored on a single *home* server. Copies of said record can be given to other *cache* servers.

We assume mobile clients access the nearest edge server for their target application. This information is as-

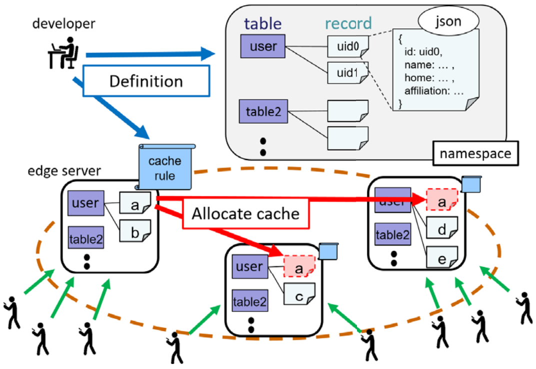

Fig. 1 System overview.

sumed to be obtained from the access points or cloud services used at connection. Clients access their closest edge server to put, get and update records. We treat MEC applications as a kind of clients. To quickly deliver record modification to whom it may concern, developers can explicitly define cache rule depending on the record type and status. Servers monitor modification events and reflect the modifications on cached copies. Depending on status changes and cache rules, the server may make cached copies on new servers proactively.

Each edge server of our test-bed system is implemented as a Web server offering a Web API for clients to get, create, and update records. Push messages and notifications are implemented using WebSocket. Developers can define entry points for push notifications in the schema.

Our test-bed system does not support fault-tolerancy or access control. We will tackle these challenges in future work.

3.2 Requirements

Distributed dynamic data structure requires careful programming to ensure data consistency. Object relocation is even more challenging. We first summarize two important features to realize application for edge computing environment.

Support for mobile objects: To treat records that represent mobile users or vehicles, we think it is reasonable to change the home servers of these records depending on their locations. Otherwise, events occurring on the targets would need to be first delivered to their respective fixed servers before delivering it to whoever is concerned. However, to explicitly treat such relocation of mobile records, programmers have to announce the relocation of these objects. A programmer must take into account concurrent object relocation and message delivery. Moreover, when a message is delivered to the old home of a mobile record, the system has to forward the message to the new home server. We call these records *mobile records*.

Data propagation based on subscription requests: There are many kinds of applications where a user or device is only concerned with events that occur with users or devices connected to the same edge server. However, some applications require communications between users or devices connected to different servers. For example, a user communicating with others located in remote places through a messaging application, or a traveler subscribing to transportation information of a remote destination. This kind of pub/sub relationships are often created dynamically and applications must deliver messages to subscribers who may change their connection edge-server often.

3.3 Approach

We propose a platform that offers a distributed object that can dynamically change home server and proactively allocate cache objects based on user defined rules. In this subsection, we first illustrate how the cache is maintained in a sample application of message service between mobile users. Then we summarize the features of our framework that participate in the cache behavior. The message service is a simple application but it can be used as a pub/sub system to develop many kinds of applications connecting mobile devices.

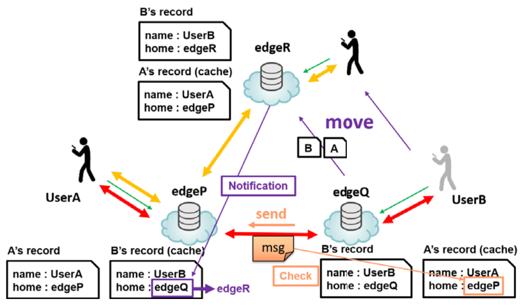

Fig. 2 Sample application image based on cache mechanism.

Figure 2 shows a situation where two users, A and B, are connected to different edge servers, P and Q. In this situation, messages to users A and B are delivered to the edge servers P and Q to which the users are connected. The users are notified of new messages using push notifications from the server they are connected to. When B moves to a position close to the server R and switches its connecting server to R, the application relocates the record that represents the user B from the server Q to R while the server P is informed of the relocation to change the destination of messages for B from Q to R.

In order to send messages to or receive messages from communication partners, the record of the communication partners are cached on the connecting server of each user. For example, the record B is cached to the server P and the cached record is used to know where the record B resides and forward messages to the record body. The cache record maintains a reference to its home server even after the record body is relocated. When A sends a message for B to the server P, the message is immediately forwarded to the server Q using the reference to the home server. Finally, when the server Q (or R) receives messages for the record B, it can immidiately send a push notification to the user using WebSocket.

In our framework, the system automatically maintains the consistency of records. The cache record forwards update requests to the body record and the modifications on the body record can be immediately reflected onto cache records. On the other hands, the developper has to specify where to place cache records beucase it depends on data structures of respective applications. In the previous application, each user record holds the list of its communication partners and the record must allocate a cache on the edge servers that its communication partners are connected to. In addition, the develloper specify the range of the associated records that is transferred along the record when it is relocated. To avoid procedural programming style, which explicitly manipulates data stream, we use cache rules in a declarative manner.

3.4 Comparing with Other Methods of Record Placement

In this subsection, we compare our approach with other record placement methods using our message application. In this application (Fig. 3(a)), messages from the user A to the user B are sent via the edge servers P and Q. The cache records of B on the server P forwards the message to the home record on the server Q and the user B can subscribe to modifications made on the record B located on the server Q. If the application needs data backup, it can allocate cache records on a cloud data service that runs our edge instances.

When using a central server (Fig. 3(b)), the communication path between two users will be long even if two users are connected to neighboring edge servers. If we allocate the central server on a cloud service that covers a large area, the network latency to the server will be large. For example, round-trip time from Kobe to Tokyo region will need 10 msec or so. Even when the application establishes demand-driven caches on nearby edge servers, it is difficult to bypass the central server to transfer dynamically created data. To bypass the central server, the system has to know who are the current subscribers and where they reside. To enable this, we rely on the explicit description of cache rules defined by developers.

Suppose a situation where we can prepare a more regional home server for each user (Fig. 3(c)). We do not change the home server dynamically. We may expect more short network latency for access than accessing a cloud server. In the case of the message service application, messages to the user B is gathered to B's home server and B can subscribe messages from a single server. However, when the user B wants to subscribe to several kinds of information at the same time, a problem remains in which server to get the information from. If B wants to subscribe to A's location continuously, one solution is to have a continuous connection to A's home server for the subscription. This means that when B has subscriptions from multiple data sources, multiple connections to respective sources must be maintained. The other solution is that A sends information directly to B's home server or A's home server forwards information to B's server. This means A has to manage the home servers of its subscribers. In our framework, the developer prepares a rule to specify who is a subscriber and the system automatically maintains proactive cache placement and update. Moreover, it allows relocation of home servers and offers shorter communication paths between clients.

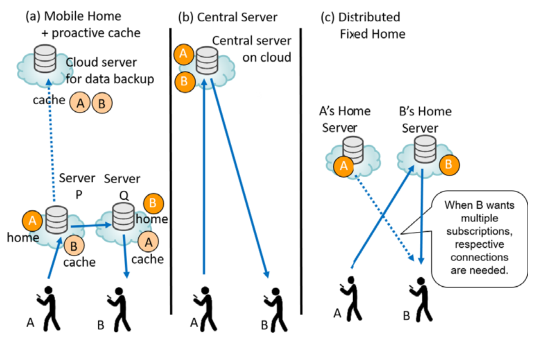
(a) Mobile Home + proactive cache
Cloud server for data backup
cache
A
B
Server P
Server Q
home
A
B
cache
home
A
cache
B
A
B
(b) Central Server
Central server on cloud
A
B
A
B
(c) Distributed Fixed Home
A's Home Server
B's Home Server
A
B
When B wants multiple subscriptions, respective connections are needed.
A
B

Fig. 3 Comparing with other methods of record placement.

3.5 Data Model

We adopt a simple data consistency model to manage the cache. The update request to a cache record is forwarded to the record body on the home server and then propagated to all the cache records on other servers.

Our system does not use separate tables to store record bodies and cached records. Each edge server accepts get/update operations only on records that reside on the edge server. An update operation to a cache record is forwarded to the home server and processed on the record body. The modified information is then propagated to the cache objects according to the cache rule. Each record is specified by its record id. No search queries are supported.

Developers define the data type and cache rule of their records for each table. In a cache rule, developers can define where to place cache objects depending on the status of the target record and its associated records as described in the next subsection. Using this feature, developers can prepare cached copies of associated records on the same edge server with a subscriber.

When a record represents a mobile object, it is typed as *mobile*. Each mobile record has a `home` property that represents its current home server. Relocation of a mobile record is triggered by modification of its `home` property. The system moves the record to the new home server while a cache record is kept on the old server to forward requests that may arrive afterward. A copy of the associated records can also be transferred along the mobile record, the range of which can be defined in the cache rule.

Sample records used in the previous application are shown in Fig. 4. For simplicity, our messaging application uses a single table that holds `user` records. The `_id` property represents the identifier for the record (e.g. `uidA, uidB`).

```
//user table
{ // record A representing user A
  "_id": "uidA",
  "name": "UserA",
  "home": "edgeP",
  "inContact": { "uidB": "UserB" },
  "message": {
    "uidB": {
      "2018-02-03T22:15:40:000+09:00": "Hello!",
      "2018-02-03T22:15:42:000+09:00": "How are you?"
    }}}
{ // record B representing user B
  "_id": "uidB",
  "name": "UserB",
  "home": "edgeQ",
  "inContact": { "uidA": "UserA" },
  "message": {
    "uidA": {
      "2018-02-03T22:15:41:000+09:00": "Hi there.",
      "2018-02-03T22:15:43:000+09:00": "Fine!"
    }}}
```

Fig. 4 Sample records of message service.

The `inContact` property lists the communication partners and the `message` property contains messages from the respective partners. User A can send a message to user B by issuing a request to insert the message into the `message` property of the record B. If the record B is a cached copy, the request is forwarded to the home server of the record B.

When the application wants to change the location of the record B, the application requests for a change in the `home` property of the record B and the platform relocates the body record along with the copies of associated records. The records for the communication partner are treated as associative records in this case.

3.6 API Specification

The frontend offers the Web API to get, create, and update records. A client wishing to retrieve a record needs to perform an HTTP GET request to the server, specifying the table and the targeted record's id. The server will then look for the record in its local database. If found, the record is returned to the client. Otherwise, an error code is returned.

Record creation is done using the HTTP POST method. The client makes the POST request to the server, providing the record to be stored in the JSON format. The server then assigns a globally unique id to the record and registers it into its database. The unique id is then returned to the client.

Finally, record updates are performed using HTTP PATCH methods. Three type of operations are provided (e.g. `add/change/delete`). The client performs to PATCH request on the server, providing the table, the targeted record's id, the operation type, and the patch context in JSON format. The context contains the target attributes and their values to be added, changed, or deleted. In the case of a delete operation, the values are ignored. If the target record has duplicated attributes for an `add` operation, or fails to have specified attributes for change or delete operations, the system rejects the request without retuning any response to the client.

Communication between edge servers is performed using a different Web API where HTTP PUT requests are used for record transportation.

3.7 Cache Rule Description

This section summarizes the current specification of the cache rule. The specification is under consideration and may evolve in the future. We will introduce the system behavior to maintain the consistency of cache in Sect. 4.

As described in Sect. 3.3, the developer has to specify where to place cache records in our framework. In addition, the developer has to specify the range of the associated records that are transferred along with the record when it is relocated. These rules depend on the data structure of each application. Therefore, we use a schema of JSON data, JSON Schema [19], to define both the application's data structure and its associated cache rulers.

JSON Schema is used to define the structure of some JSON data. It is intended to enable validation, documentation, hyperlink navigation, and interaction control of JSON data. The schema of the user records and its embedded cache rule are presented in Fig. 5. JSON Schema offers types of values (e.g. `"null"`, `"boolean"`, `"object"`, `"array"`, `"number"`, or `"string"`). An `object` represents a JSON object that has unordered set of properties (lines 2, 11, 20, and 23). The `properties` attribute lists the names of the child properties and the validation for their values (lines 3–34). The `patternProperties` attribute allows the arbitrary property names that satisfy the given regular expression (lines 12–18, 21–33, and 31–33). It is often used to represent associative lists. JSON Schema allows reference to an other schema (or the self schema for recursive structures) using a `"$ref"` keyword but our system does not support schema references yet.

```
 1 { "$schema": "http://json-schema.org/draft-04/schema#",
 2   "type": "object",
 3   "properties": {
 4     "_id": {
 5       "type": "string",
 6       "pattern": "^[0-9a-f]{24}$"
 7     },
 8     "name": { "type": "string" },
 9     "home": { "type": "string" },
10     "inContact": {
11       "type": "object",
12       "patternProperties": {
13         "^[0-9a-f]{24}$": {
14           "type": "string",
15           "keyTableInfo": "user",
16           "followingData": {
17             "target": "key", "chain": false }}
18       }},
19     "message": {
20       "type": "object",
21       "patternProperties": {
22         "^[0-9a-f]{24}$": {
23           "type": "object",
24           "keyTableInfo": "user",
25           "push": {
26             "tag": "receiveMessage",
27             "method": "PATCH",
28             "patchParam": "add",
29             "to": ["/_id"]
30           },
31           "patternProperties": {
32             "..time..": { "type": "string" }
33           }}}
34     }},
35   "cacheType": {
36     "centralized": {
37       "targetEdge": ["/inContact/@key->home"] }
38   },
39   "switchEdge": true
40 }
```

Fig. 5 JSON Schema describing the data and the cache rule for our message service application.

We introduce extra attributes into the JSON Schema to describe the cache rule using dedicated keywords. JSON Schema allows extension of JSON Schema using additional keywords. A schema definition is prepared for each table to represent the data structure of the records in the table and specify their cache rule. We also introduce an attribute to enable push notifications from edge servers to devices. The attributes used to define cache rules are the following:

- The `switchEdge` attribute is used at the document root of the target record and specifies whether the records are "mobile records" or not. The system automatically relocates the records when the value of their `home` property changes.

- The `cacheType` attribute is used at the document root of the target record and specifies that the system automatically places cache of the records to the edge servers specified by the paths written in its `targetEdge` attribute.
- The `keyTableInfo` and `valueTableInfo` attributes are used in `properties` and `patternProperties` when their property names and values contain identifiers of records, respectively. They denote the table that contains the records denoted by the identifiers.
- The `followingData` attribute is used in `properties` and `patternProperties` and specifies the range of the associated records of the target record that should be transferred along when it is relocated.
- The `push` attribute is used in `properties` and `patternProperties` and specifies which properties to monitor. When the system finds modification on the attributes, it sends a push notification following the specification.

The `cacheType` attribute specifies the cache placement (e.g. lines 35–38). It contains one cache strategy in its body. Currently, only the `centralized` strategy is supported. Within the strategy, the records which should be cached locally are listed under the `targetEdge` attribute. In the case of `cacheType` in lines 35–38, "`/inContact/@key->home`" is a path to a property that contains the references to cache servers. The path starts from the root of the record and '`/`' is used to access a child element. The following token (e.g. `inContact`) is a property name and `@key` is a special token that matches with all the pattern property names. The arrow "`->`" represents a reference to an other record using record id. To reference records in different tables, we use `keyTableInfo` and `valueTableInfo` attributes. In this example, `keyTableInfo` attributes are used in line 15 and 24. In this case, The path gathers the home servers of the user records with whom the user is in contact (`inContact`) in this application. The system traverses the paths and gathers the list of servers and allocate cache records on the servers. In the situation corresponding to Fig. 4, the record A is cached to server `Q`, which is the home server of the record B. When the server finds a modification on the records, it delivers the modification to the cache records. The data transfer to allocate new cache records or reflect modifications on the cache record is triggered immediately by the system and reflected on the receiving servers eventually.

For the relocation of objects, `switchEdge` specifies whether the target record is mobile or not. When the mobile record contains references to other records, the developer can specify whether the referenced records should be transferred along when it is relocated using `followingData` attributes. A `followingData` is used in `properties` and `patternProperties` and the `target` attribute represents whether the record id is contained in the `key` or the `value`. The other attribute, `chain`, specifies whether or not associated records are recursively gathered from the record id. In this application, a `followingData` attribute in lines 16–17 specifies that the system takes copies of the records representing the current communication partners when a user record moves to another server.

Finally, we introduce the specification of push notification. The `push` attribute is attached to the properties to be monitored. The `method` and `patchParam` parameters (e.g. `POST/PATCH/DELETE` and `add/change/delete`) specify the type of access made to the monitor. The `tag` parameter declares the topic name attached for the notification and the `to` parameter indicates the subscribers using the same path notation as in the `cacheType` attribute. Modifications in the `message` property are notified only to the subscribers connecting to the server. In the case of lines 25–30, when a new message is added into the `messages` property using the `add` operation of a `PATCH` request, the server sends a notification message with the tag, `receiveMessage`, if the user represented by the record is connecting to the server using WebSocket. The notification contains the tag, the id of the target record, and the context of the `PATCH` request.

4. Implementation

An overview of our system's internals is presented in Fig. 6. An edge server consists of three components: the frontend service, the cache manager and the database. The frontend service manages requests from clients as well as push notifications through WebSocket. When the frontend service receives create or update requests, it triggers the cache manager which manages the behavior of records based on the cache rules given by developers. The database keeps the records in several tables. Edge servers use MongoDB as their database management system. MongoDB can be used as a key-value storage and can hold JSON documents as values. We adopted the Go language to implement the frontend server and the cache manager and used the Gin framework to process HTTP requests along with the Gorilla web toolkit to manage WebSocket connections. We prepared a docker container for easy deployment of edge servers. The container is based on Ubuntu 16.04 and contains all the components mentioned above.

Creation requests and update requests are processed by the cache manager according to the schema definitions of the corresponding table. The server holds the schema definitions in a table of the database and converts them into

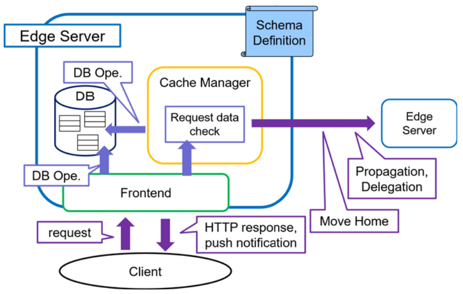

Fig. 6 Overview of system inside.

memory objects when the schema of respective tables are needed. When receiving a creation request, the server stores the record to the database with a generated id. If a cache rule is defined for the record's table, the cache manager checks the specification for the cache servers of the record. If the path includes references to associated records, the cache manager loads the referred records to determine the servers on which cached copies of the record need to be made. Each cache record holds a `_cache` property to represent whether the record is a cache or not and a `_homeServer` property to indicate its home server.

When a server receives an update request to a record, it first looks up the record using its id and conducts the following operations:

- If the record is a cache record, the server forwards the request to the record on the home server (delegation).
- If the record is a body record, the server updates its record and forwards the update request to the record's cache servers (propagation).
- When the `home` property of a mobile record is changed on its home server, the server relocates the record on the server denoted by the new `home` value (moveHome). When the cache rule of the record has a rule of associated records, the server also copies the associated records to prepare their cache records on the new home server.
- If a server notices the modification of a reference in the path to the cache servers (checkPath), it notifies the root record of the path to check the current range of cache servers and conducts cache update if necessary.

Our current implementation offers the first three but lacks the checkPath feature. Therefore, applications cannot use fully managed cache mechanisms if their cache rule contains two record references in a single path. This is not an issue in our sample application as it contains at most one reference in its paths. To reduce communication cost, we introduced an optimization which consists in omitting to propagate update requests to the source of the delegation. Using this optimization, when a server receives an update request to a cache record, it applies the update to its cache and forwards the request to its home server which then omits to propagate the updated record. This optimization may change the order of update operations and violate record consistency. We are planning to introduce an operation ordering mechanism in future works.

To ensure mobile records consistency, our implementation handles the concurrent occurrence of moveHome operations with delegation or propagation operations. To avoid loss of concurrently sent delegation messages, our system does not remove the record from the old home server for a certain amount of time, long enough to allow the system to announce the relocation of the record to all cache servers. The old home record is added a `_forward` property and subsequent messages are forwarded to the new home server. As per the propagation requests, when a server receives a record copy from a cache record, it checks the home server of the record and sends a request to the home server to get the latest state of the record. This action notifies the home record of the allocation of a new cache record and circumvents concurrent propagation requests.

5. Sample Applications

We introduce two other applications, a group chat service and a location-based game.

In the group chat service, a group is used for communication between its members. This sample is similar to the situation of Fig. 3(c). When a client participates to several chat groups, the client must have several connections when it uses direct connections to the servers. In this application, the record for each group is located on a fixed home server while cache records are prepared on the edge servers to which the members of the group are currently connected. The edge server is used to gather messages from multiple fixed home servers. Figure 7 shows two sample records of the application. The user record in lines 2–9 holds a `chatGroup` property in which the chat groups in which the user is involved are kept.

The group record in lines 11–29 uses a `member` property to hold its members and a `messageBox` property to store the messages sent by each member. User records are cached to the home server of each group they belong to while the group records are cached to the edge servers to which members of the group are connected. When users change their connection server, the application relocates their user record along with the cached records of the groups they are involved in. Developers can easily enforce this data distribution management using our platform.

A location-based game is an application that utilizes the users' location in its game mechanisms. Since players mainly depend on events that occur in nearby locations, edge computing fits such applications. Figure 8 shows the data structure of the game. The game field is separated into

```
 1 // user table
 2 { //user record
 3   "_id": "uidA",
 4   "name": "UserA",
 5   "home": "edgeP",
 6   "chatGroup": {
 7     "gidX": "GroupABC"
 8   }
 9 }
10 // group table
11 { // group record
12   "_id": "gidX",
13   "name": "GroupABC",
14   "home": "edgeR",
15   "member": {
16     "uidA": "UserA",
17     "uidB": "UserB",
18     "uidC": "UserC"
19   },
20   "messageBox": {
21     "uidA": {
22       "2018-02-03T22:15:41:000+09:00" : "Hello!",
23       "2018-02-03T22:15:43:000+09:00" : "How are you?"
24     },
25     "uidB": {
26       "2018-02-03T22:15:42:000+09:00" : "Hi there."
27     }
28   }
29 }
```

Fig. 7 Sample records of group chat.

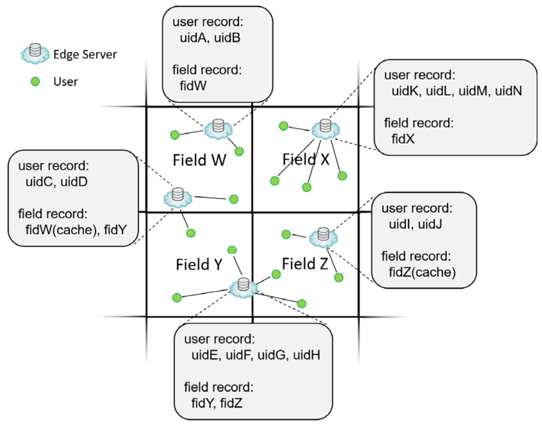

Fig. 8 Overview of location-based game.

```
 1 // user table
 2 { // user record
 3   "_id": "uidA",
 4   "name": "UserA",
 5   "home": "edgeP",
 6   "friend": { "uidB": "UserB" },
 7   "gameData": {...}
 8 }
 9
10 // field table
11 { // field record
12   "_id": "fidW",
13   "home": "edgeP",
14   "name": "FieldW",
15   "cloud": "CloudDomain",
16   "userList": {
17     "uidA": "UserA",
18     "uidB": "UserB" },
19   "serverList": {
20     "ServerName1": "edgeP",
21     "ServerName2": "edgeQ" },
22   "event": {...}
23 }
```

Fig. 9 Sample records of location-based game.

cells. Each cell manages its status and events that occur within its designated area. In this application, cells are supported by multiple neighboring servers since multiple edge servers often cover a common area. Each edge server holds record bodies or cache records of the neighboring cells. Therefore, the players can quickly access events that occurred in the neighboring cells through the edge server they are connected to. Moreover, even when the users are executing the message service concurrently, they can subscribe all the notifications for the game and the messaging service from a unique edge server using a single WebSocket connection.

Figure 9 presents some sample records of the application. The user record in lines 2–8 holds a `friend` property to represent the friend list of the player and a `gameData` property that contains the private status of the player. The field record in lines 11–23 represents a field cell. The `home` property indicates its home server and the `serverList` property represents the list of the neighboring edge servers on which cache records are placed. The `cloud` property represents a backup server on the cloud service. Figure 10 presents a cache rule of the application. User records are cached to the servers on which friends are connected to (lines 13–16) and field records are cached to their neighboring edge servers and backup servers (lines 40–44). The players can quickly receive the events that occurred in their current field cell via push notifications (lines 32–37). When a player moves to a different cell, the application removes the player from the former cell and adds the player to the new cell. When a player switches connection server, the application updates the `home` property of the user record and the system relocates the record to the new server.

```
 1 // user schema
 2 { "properties": {
 3     "home": { "type": "string" },
 4     "friend": {
 5       "type": "object",
 6       "patternProperties": {
 7         "^[0-9a-f]{24}$": {
 8           "type": "string",
 9           "keyTableInfo": "user",
10           "followingData": {
11             "target": "key", "chain": false}}
12       }}},
13   "cacheType": {
14     "centralized": {
15       "targetEdge": ["/friend/@key->home"]}
16   },
17   "switchEdge": true
18 }
19
20 //field schema
21 { "properties": {
22     "userList": {
23       "type": "object",
24       "patternProperties": {
25         "^[0-9a-f]{24}$": {
26           "type": "string",
27           "keyTableInfo": "user"}
28       }},
29     "serverList": { "type": "object" },
30     "event": {
31       "type": "object",
32       "push": {
33         "tag": "receiveEvent",
34         "method": "PATCH",
35         "patchParam": "add",
36         "to": ["/userList/@key"]
37       }},
38     "cloud": { "type": "string" }
39   },
40   "cacheType": {
41     "centralized": {
42       "targetEdge": [
43         "/serverList/@value",
44         "/cloud"]}}
45 }
```

Fig. 10 Cache rule of location-based game.

6. Evaluation

We conducted two experiments to evaluate the performance of our system. The first one is a ping-pong latency measurement using the messaging service described in Sect. 3. The other is a workload test using 1000 clients and 16 edge servers. We used the message service and the location-based game separately for this second test.

To simulate client behavior, we prepared a program that runs multiple threads accessing the edge servers. Each thread represents a client and accesses the edge servers using HTTP requests and WebSocket connections. In our second experiment, clients change their connection servers several times. The program is executed on a machine equipped with a single CPU (Intel Xeon X3430, 2.40 GHz, 4 cores) and 16GB of memory. Each edge server is wrapped in a docker container. The Docker version we used is 18.03.1-ce. We run one or more containers on each host, each fitted with two CPUs (Intel Xeon E5410, 2.33 GHz, 4 cores) and 16GB

Table 1 Latency measurement.

clients connect to / elapsed time			same server	different servers	
				body	cache
total of one-way trip (A)			1898.5 us	3692.3 us	
	Server total (B)		1468.3 us	1589.9 us	1237.0 us
		Record update	716.4 us	764.6 us	800.1 us
		Field check	143.0 us	145.7 us	161.9 us
		Cache update @body	393.1 us	455.4 us	-
		Delegate @cache	-	-	59.3 us
		Push notification	82.6 us	87.7 us	85.0 us
		Others	133.2 us	136.5 us	130.7 us
	Network (A - B)		430.2 us	865.3 us	

memory. We assign different IP addresses to the containers. The local database of each edge server is contained in its container and stored in the local HDD of the host. Both the edge servers and the host machines are located in a local network without any traffic control.

6.1 Latency Measurement

We prepared two clients and conducted 20 times round trips of messages between the clients. We measured the average time for the one-way trip and the time taken by the respective components of the edge servers. The results are presented in Table 1 for two situations: when the clients are connected to the same server, and when the clients are connected to different servers.

The elapsed time for a one-way trip is less than 2 msec when clients are connected to the same server and less than 4 msec when connected to different servers.

The cost of accessing the database dominates the performance. The record update operation includes one read and one write access to the database, which account for 85% of the elapsed time of the record update. The field check operation includes parsing and checking of JSON properties. The cache update includes the read operation of associated records which consists of one database read access in this application. The time for push operation is the elapsed time needed to execute a push operation on edge servers. The network elapsed time is computed as the difference between the total elapsed time (A) and the combined elapsed time on the server side (B).

When the clients are connected to different servers, the update record and the field check operations are executed on both servers. The delegation operation does not contain any additional DB access. The network elapsed time includes the communication between the clients and their connection server, and the communication between the servers.

When using cloud services, we can reduce the number of DB accesses but we will suffer more from network latency (e.g. 10msec for round-trip time from Kobe to Tokyo region). In order to gain a significant advantage over cloud services, our system needs to reduce DB access latency.

6.2 Workload Test

In this second experiment, we perform a workload test on the message service described in Sect. 3 and the location-based game described in Sect. 5. We used one client simulator with 1000 client threads and eight hosts, each running two edge servers. Figure 11 is a flat field with a torus-like topology on which clients are placed. This field is used by both applications. Each rectangular cell has four neighbors (the cells against the side of the figure connect with the cell located on the opposite side). Each cell holds a single edge server and the clients in the cell are connected to this server.

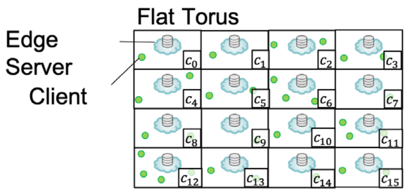

Fig. 11 Experiment field for workload test.

In both applications, clients are initially placed in randomly selected cells and move to neighboring cells in a randomly selected direction following the probability law described later. When a client moves to a cell, it connects to the edge server in the new cell.

In the message service, the clients form a one-way ring network that is independent of the location of clients. Each client connects to exactly two other clients and sends messages to the next client in a clockwise direction. Every 400 msec, each client has a 20% probability to send a message and a 5% probability to move to a neighboring cell.

In the location-based game, each cell represents a field cell of the game. A field cell uses the edge server in the cell as its home server and allocates cache records to the edge servers in the neighboring cells. Every 400 msec, each field cell has a 20% probability to generate an event in the cell and each client has a 5% probability to move to a neighboring cell.

We prepared the initial records and WebSocket connections beforehand and executed the simulation for 20 sec and 300 sec in the message service and the location-based game, respectively.

Test under uniform distribution: Table 2 shows the numbers of messages and operations in the applications when the clients are distributed on each cell with the same probability. The clients are initially placed on each cell with the same probability and select the next cell to be visited from the neighboring cell with the same probability.

In the message service, the total number of messages and home relocation were 9978 and 2512 respectively and our system succeeded to deliver a message in 4.7/1.6/23.2 msec in average/minimum/maximum. There were, on average, 168 requests/sec to each edge server. Each user record had at most two cache records for the connected clients. When a client sends a message, the sender server issues at most one propagation request and the home server issues at most one delegation request. The numbers of requests issued for each operation are shown in Table 2. Without the propagation optimization technique described in Sect. 4, the number of propagation request was at most two. The technique reduced the requests needed for inter-edge message delivery by 1/3. Concurrent moveHome and delegation operations only occurred four times during the experiment.

In the case of the location-based game, the total number of event occurrence was 2388 and the events were delivered to the clients in 6.1/1.5/68.9 msec in average/minimum/maximum. We think such latency is acceptable for real time games. Clients moved to a neighboring cell and changed connection server 37557 times in total. The latency of push notifications comes from two reasons. When a client is not connected to the home server of the cell where an event occurs, the event first needs to be delivered to the cache records kept by the neighboring servers. Only then will the notification be delivered to the client. The second reason is the sequential execution of push notifications. There is an average of 62 clients in one field cell and push notifications of a record are executed sequentially in our current implementation. To shorten the latency, we are planning to support parallel execution of push notifications in future works.

Test under non-uniform distribution: Next, we introduced non-uniform distribution of clients. Fig. 12 shows the numbers of clients and latency until the occurred events are notified to the clients. We picked up three types of cells: the cell 0, 1, and 5. We assign the preference degree ρ_i to each cell c_i. The probability that clients are initially placed on the cell c_i is set to $\frac{\rho_i}{\sum_{k\in S}\rho_k}$ where $S = \{0, \cdots, 15\}$. The probability that a client on the cell c_i selects c_j as the next cell from the set of the neighboring cells N_i ($c_j \in N_i$) is set to $\frac{\rho_j}{\sum_{k\in N_i}\rho_k}$. In this trial, $\rho_i(t)$ ($0 \leq t \leq 300$ is the elapsed time in seconds after the setup) is set to $3 - 2cos(\pi \times t/150)$ and $3+2cos(\pi\times t/150)$ for $i \in \{5, 6, 9, 10\}$ and $i \notin \{5, 6, 9, 10\}$, respectively. This setting is intended to emulate day and night distribution of populations. Figure 12 shows that the number of clients on each edge has a variation from 10 to 200 at most. The latency becomes longer when the number of clients exceed the performance of edge servers. This suggents that load balancing between neighboring edge servers will be important under non-uniform workload. We are interested in how our cache mechanism can be extended or applied to offload tasks from crowded servers.

Comparison with Fixed Home Servers: Finally, we compare our cache allocation method with the approach using fixed home servers illustrated in Fig. 3(c)). Suppose there is a specialized client program for one-way ring network for the message service. It subscribes messages from its home server. and sends messages directly to the home server of the receiver. The total number of the requests (client-server only) will be the twice of the number of the sent messages in Table 2. The average network path length of each request corresponds to the average network distance between arbitrary selected two cells from the experiment field. Even when the sender and the receiver reside on neighboring cells, the network path will be long when each of the home servers resides in a remote location. In contrast, our framework can reduce the network path length for such a situation.

In the case of the location-based game, the member list on each game field must be cached to the neighboring edge servers if the client program needs to subscribe to the member list of the neighboring game fields from the edge server they are connected to. The developer has to implement a client program or a server program that updates the member list cached to the neighboring edge servers. Using our framework, the developer only has to prepare cache rules to prepare such edge servers. There is little difference in the number of operations or their network traffic.

Table 2 Workload test (uniform distribution).

application		message service		location-based game		
operation		message send	client relocation	event occurrence	moving to an other field	client relocation
total numbers of operations		9978	2512	2388	37557	37557
number of requests issued for one operation	request from client	1	1	1	2	1
	propagation	≦1	≦1	5	9	
	delegation	≦1			1	
	"_homeServer" update		≦2			
	record relocation		≦3			1
	sync. of cache record		≦2			
total numbers of requests		53796 (client-server : 12490) (server-server : 41306)		540126 (client-server : 115155) (server-server : 425475)		
number of requests per second per edge server		168 request/sec		113 request/sec		
latency (avg./min./max.)		4.7/1.6/23.2 ms *1		6.1/1.5/68.9 ms *2		

*1 message deliveery between clients
*2 event notification to clients

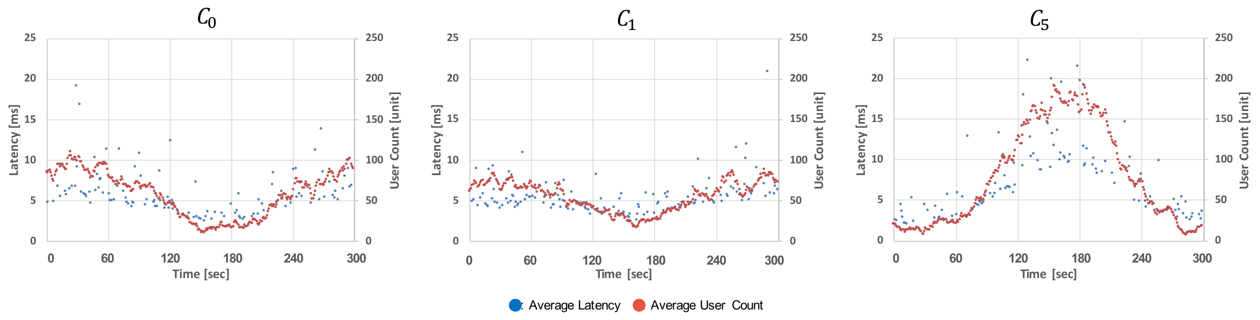

Fig. 12 Workload test (non-uniform distribution).

7. Conclusion and Future Work

In this article, we proposed a distributed key-value storage platform for edge computing and its explicit data distribution management method. The platform offers a uniform namespace and allows allocation of cached records following user defined rules. Each rule is defined in a declarative manner and specifies where to place cache objects depending on the status of the target record and its associated records. We prepared three sample applications, including a messaging service between mobile appliances, to demonstrate how cache rules apply to applications. In our experiments, the application needed 5 msec on average to deliver a message between two clients connected to different servers. In a workload test, we succeeded in running the application with 1000 clients using 16 edge servers.

The current system has some bugs and limitations in terms of fault tolerance and consistency management. Using our system, developers can prepare cache records on a cloud service for data backup. However, when the connection between the edge server and the cloud service is lost, the edge server becomes a single point of failure. We are interested in introducing the elegant record replication method of FogStore [8] to enable global lookup of records and data backup. We intend to address these issues in future work.

Acknowledgments

We would like to thank Patrick Finnerty for his precious comments. This work was supported by JSPS KAKENHI Grant Numbers JP16K12488, JP15K00124, JP15K00125, JP18H03232.

References

[1] T. Nagato, T. Tsutano, T. Kamada, Y. Takaki, and C. Ohta, “Distributed key-value storage for edge computing and its explicit data distribution method,” Proc. 33rd International Conference on Information Networking (ICOIN 2019), pp.147–152, Jan. 2019.
[2] J. Dizdarevic, F. Carpio, A. Jukan, and X. Masip-Bruin, “A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration,” ACM Comput. Surv., vol.51, no.6, pp.116:1–116:29, Jan. 2019.
[3] ETSI, GS MEC 003 - V2.1.1, “Multi-access Edge Computing (MEC); Framework and Reference Architecture,” Jan. 2019.
[4] J. Plachy, Z. Becvar, and E.C. Strinati, “Dynamic resource allocation exploiting mobility prediction in mobile edge computing,” 2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), pp.1–6, Sept. 2016.
[5] W. Bao, D. Yuan, Z. Yang, S. Wang, W. Li, B.B. Zhou, and A.Y. Zomaya, “Follow me fog: Toward seamless handover timing schemes in a fog computing environment,” IEEE Commun. Mag., vol.55, no.11, pp.72–78, Nov. 2017.
[6] J. Liu, J. Wan, B. Zeng, Q. Wang, H. Song, and M. Qiu, “A scalable and quick-response software defined vehicular network assisted by mobile edge computing,” IEEE Commun. Mag., vol.55, no.7, pp.94–100, July 2017.
[7] R. Mayer, H. Gupta, E. Saurez, and U. Ramachandran, “FogStore: Toward a distributed data store for Fog computing,” 2017 IEEE Fog World Congress (FWC), pp.1–6, Oct. 2017.
[8] H. Gupta and U. Ramachandran, “FogStore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access,” Proc. 12th ACM International Conference on Distributed and Event-based Systems, DEBS’18, pp.148–159, ACM, New York, NY, USA, 2018.
[9] A. Sharma, X. Tie, H. Uppal, A. Venkataramani, D. Westbrook, and A. Yadav, “A global name service for a highly mobile internetwork,” SIGCOMM Comput. Commun. Rev., vol.44, no.4, pp.247–258, Aug. 2014.
[10] V.P. Kafle, Y. Fukushima, P. Martinez-Julia, and H. Harai, “Scalable directory service for iot applications,” IEEE Commun. Stand. Mag., vol.1, no.3, pp.58–65, Sept. 2017.
[11] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, “Aurora: A new model and architecture for data stream management,” The VLDB Journal, vol.12, no.2, pp.120–139, Aug. 2003.
[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S.R. Madden, F. Reiss, and M.A. Shah, “TelegraphCQ: Continuous dataflow processing,” Proc. 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD’03, pp.668–668, ACM, New York, NY, USA, 2003.
[13] S. Yamada, Y. Watanabe, H. Kitagawa, and T. Amagasa, “Location-based information delivery using stream processing engine,” Mobile Data Management, 2006. MDM 2006. 7th International Conference on, p.57, IEEE, 2006.
[14] Amazon Web Services, Inc., AWS Lambda Developer Guide, 2017.
[15] W.J. Chen, R. Gupta, V. Lampkin, D.M. Robertson, N. Subrahmanyam, et al., Responsive Mobile User Experience Using MQTT and IBM MessageSight, IBM Redbooks, 2014.
[16] J. Kreps, N. Narkhede, J. Rao, et al., “Kafka: A distributed messaging system for log processing,” Proc. NetDB, pp.1–7, 2011.
[17] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V.Y. Ye, “Building linkedin’s real-time activity data pipeline,” IEEE Data Eng. Bull., vol.35, no.2, pp.33–45, 2012.
[18] Google Cloud Platform, Google CLOUD PUB/SUB.
[19] json-schema.org, “Json schema specification (draft-04),” https://json-schema.org/specification.html

Takehiro Nagato received B.E. degrees in engineering from Kobe University, Hyogo Japan in 2017. Currently, he is a master course student in the Graduate School of System Informatics, Kobe University. His research interest includes efficient data management platform for edge computing.

Takumi Tsutano is an undergraduate student of a department of Computer Science and Systems Engineering, Kobe University. His research interest includes secure data platform for cloud and edge computing.

Tomio Kamada received his B.Sc. and M.Sc. degrees from the University of Tokyo in 1993 and 1995, and received his Ph.D. (Eng.) degree from Kobe University in 2004. He was an Assistant Professor at Kobe University during 1998–2010. Since April 2010, he has been a lecturer of Graduate School of System Informatics, Kobe University. His research interest includes parallel and distributed computations, and the runtime systems of programming languages. He is a member of IPSJ, JSSST, and ACM.

Yumi Takaki received B.E. of science and technology from Saga University, Saga Japan, in 1989. From April 1989, she joined FUJITSU LIMITED. She then joined Hitachi Zosen Corporation. In 1999, she joined Kobe University, Hyogo Japan. She became an Assistant at the Graduate School of Engineering, Kobe University in 2007. Since April 2010, she has worked as an Assistant at the Graduate School of System Informatics, Kobe University. Her current interests include the research of wireless ad-hoc networks and network simulation techniques.

Chikara Ohta received the B.E., M.E., and Ph.D. (Eng.) degrees in communication engineering from Osaka University, Osaka, Japan, in 1990, 1992, and 1995, respectively. From April 1995, he was with the Department of Computer Science, Faculty of Engineering, Gunma University as an assistant professor. From October 1996, he was a lecturer of Department of Information Science and Intelligent Systems, Faculty of Engineering, University of Tokushima, and an associate professor there from March 2001. From November 2002, he was an associate professor of Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University. From April 2010, he was an associate professor of Graduate School of System Informatics, Kobe University, and a professor there from January 2015. Since April 2016, he has been a professor of Graduate School of Science, Technology and Innovation, Kobe University. From March 2003 to February 2004, he was a visiting scholar in the University of Massachusetts at Amherst, USA. His current research interests include performance evaluation of communication networks. He is a member of IPSJ, IEEE, and ACM SIGCOMM.