PAPER *Special Section on Data Engineering and Information Management*

Dynamic Load-Distribution Method of uTupleSpace Data-Sharing Mechanism for Ubiquitous Data

Yutaka ARAKAWA[†*a)], **Keiichiro KASHIWAGI**[†], **Takayuki NAKAMURA**[†b)], *and* **Motonori NAKAMURA**[†], *Members*

SUMMARY The number of networked devices of sensors and actuators continues to increase. We are developing a data-sharing mechanism called uTupleSpace as middleware for storing and retrieving ubiquitous data that are input or output by such devices. uTupleSpace enables flexible retrieval of sensor data and flexible control of actuator devices, and it simplifies the development of various applications. Though uTupleSpace requires scalability against increasing amounts of ubiquitous data, traditional load-distribution methods using a distributed hash table (DHT) are unsuitable for our case because of the ununiformity of the data. Data are nonuniformly generated at some particular times, in some particular positions, and by some particular devices, and their hash values focus on some particular values. This feature makes it difficult for the traditional methods to sufficiently distribute the load by using the hash values. Therefore, we propose a new load-distribution method using a DHT called the dynamic-help method. The proposed method enables one or more peers to handle loads related to the same hash value redundantly. This makes it possible to handle the large load related to one hash value by distributing the load among peers. Moreover, the proposed method reduces the load caused by dynamic load-redistribution. Evaluation experiments showed that the proposed method achieved sufficient load-distribution even when the load was concentrated on one hash value with low overhead. We also confirmed that the proposed method enabled uTupleSpace to accommodate the increasing load with simple operational rules stably and with economic efficiency.
key words: *tuple space, load distribution, DHT*

1. Introduction

The number of various networked sensor devices such as temperature sensors, rain gauges, and GPS devices has been rising recently. To make the best use of sensor data acquired by the devices, we should ensure that data are shared among applications. Sharing data increases the type and amount of data that an application can use and enables the application to recognize events in the real world more multilaterally and minutely. This facilitates the development of new applications. Share data of networked actuator devices among applications enables the application to control systems in the real world more freely and also encourages the development of various applications. For this purpose, we are working on the data-sharing mechanism called uTupleSpace for ubiquitous data [1]. Ubiquitous data include sensor data and actuator device data. As the types and amounts of shared data increase, the more value such data stores will gain. As this happens, it will be important to achieve scalability of uTupleSpace.

Meanwhile, distributed hash tables (DHTs) function as scalable data stores. A DHT is a system for managing a large hash table by a number of peers in a decentralized way. They are being studied intensively as a representative scale-out technology. DHTs are thought to have strong potential to realize scalability of uTupleSpace.

However, to the best of our knowledge, traditional DHTs cannot distribute load adequately among peers. Each peer controls a section in the whole hash space and is burdened with the load caused by data whose hash value is within that section. The DHTs initially decide which peer will manage which section. Then, most of the methods of traditional DHTs involve adjusting the size of the section according to the load on the section dynamically. However, such methods cannot handle loads that are concentrated on particular hash values. Even if the methods reduce the size of the section to only one hash value, the load on the hash value could not be accommodated by the peer. Moreover, when the amount of accumulated data on each hash value gets too large, the load of migration processes caused by relegation of hash subspaces among peers will also become large.

DHTs distribute data according to the value of a distribution key, and therefore, the load distribution depends on the distribution key. Each datum contains one or more key-value pairs; the distribution key is one of the keys that is contained by each datum. Because the value of the distribution key is used as an argument of a hash function, this value needs to be specified in all data insertion processes and all search processes. Therefore, when we design the uTupleSpace with a DHT, we have to choose the key that will be used as a distribution key, and that will be contained by the sensor data and used by applications in search conditions naturally. For example, the sensing time and the type of sensor can potentially be keys. However, because there may be a large amount of sensor data that have the same sensing time or are from the same type of sensor, the problem of load concentration can occur. By using a distribution key made of two or more keys, the amount of data related to one hash value is reduced, and the problem will be alleviated to some degree. However, the problem cannot be solved perfectly, and it will reduce the usability because the number of keys that must be specified in the insertion and search

Manuscript received July 4, 2013.
Manuscript revised October 29, 2013.
[†]The authors are with NTT Network Innovation Laboratories, NTT Corporation, Musashino-shi, 180–8585 Japan.
[*]Presently, with NTT Comware Corporation.
a) E-mail: arakawa.yutaka@nttcom.co.jp
b) E-mail: nakamura.takayuki@lab.ntt.co.jp
DOI: 10.1587/transinf.E97.D.644

process increases. With the traditional DHTs, it would be assumed that the amount of data related to one hash value is not very large. However, that cannot be assumed in our situation.

Therefore, we propose in this paper a new load-distribution method for uTupleSpace. The load related to one hash value can be handled by one or more peers in the proposed method. This makes it possible to handle the large load related to one hash value by distributing the load among the peers. Moreover, the proposed method reduces the load of migration processes. This prevents insertion and search processes from stacking up because of the migration process load. We evaluated the performance of the proposed method by using a micro-benchmark and confirmed that it achieves adequate load-distribution with low overhead. We also evaluated the feasibility of the proposed method through simulation and confirmed that the proposed method enables uTupleSpace to accommodate the increasing load with simple operational rules stably and cost-effectively.

In this paper, the following points are newly added to our previous work [2]. (1) We introduce an simple CommandActual server (sCA server) in order to distribute the load of the application that mainly uses actuators. (2) The performance of the proposed method with the sCA server is revaluated. (3) We evaluated the feasibility of the proposed method by conducting an operational simulation.

The rest of the paper is organized as follows. Related studies are described, and the differences between them and the proposed method are discussed in Sect. 2. The proposed method is explained in Sect. 3. The communication model of uTupleSpace is explained first, then the uTupleSpace system using the proposed method is explained. The evaluation of the proposed method is presented in Sect. 4, and we conclude the paper in Sect. 5.

2. Related Work

uTupleSpace is based on the tuple space [3], a communication model used in parallel computing. Many studies have been conducted to investigate the scalability of tuple space.

TinyLIME [4] and TeenyLIME [5] organize ad-hoc tuple spaces over available terminals using the connectivity of the local wireless network. W4TupleSpace [6] converts data into a uniform tuple format called W4 (Who, What, Where and When), and stores it into a tuple space divided regionally or by topic. Agimone [7] integrates two systems, Agilla [8] for wireless sensor networks (WSNs) and Limone [9] for an IP network. Agimone organizes a tuple space for each WSN and enables communication only among prearranged tuple spaces. These methods prevent a tuple space from becoming too large by dividing it into pre-defined units like geometric areas or connectivities and make it possible to accommodate a large volume of data and communications. However, it is difficult to pre-define the unit for dividing a tuple space when the sensor data are shared by various applications because the range of needed sensor data depends on the applications. Division by a pre-defined unit would undermine the flexibility and restrict application development.

De at al. [12] tried to achieve scalability by speeding up the matching processes instead of dividing the tuple space. However, it is hard to say whether the method secures sufficient scalability because the search time is proportional to the number of tuples.

DTuples [10] and BISSA [11] make each tuple contain a subject key or message key and use a DHT to construct a huge tuple space by using the key as a distribution key. The load-distribution methods of these systems depend on the DHT used in the systems.

On the other hand, the load-distribution in DHTs has also been studied intensively. Chord [13] introduces the notion of virtual servers to cope with the heterogeneity of peers. Peers participating in a DHT can host different numbers of virtual servers, thus taking advantage of peer heterogeneity. Most of the studies of load-distribution in DHTs focus on how the virtual servers are managed [14]–[16].

A. Rao et al. [14] proposed a method of reallocating virtual servers hosted by an overloaded peer to a lightly loaded peer. Because the migration process for reallocation of virtual servers is computationally expensive, the technical problem is how to reduce the cost without unbalancing the load. To solve the problem, S. Surana et al. [15] and C. Chen et al. [16] introduced centralized load balancing algorithms, where dedicated servers manage the migration of virtual servers among the peers. Furthermore, L. Yang et al. [17] proposed a method of dividing virtual servers if a peer becomes overloaded even when the peer hosts only one virtual server.

However, these traditional methods cannot handle the case when the load related to only one hash value causes the peer to be overloaded. This is because the data related to one hash value are hosted by only one peer in these methods. In the case of uTupleSpace in particular, as previously described, this is a major problem because the amount of data related to one hash value tends to be large. On the other hand, the proposed method permits peers to host data redundantly and makes it possible to handle cases where the data related to one hash value become too concentrated, as in a hotspot.

Because these methods adopt centralized algorithms and introduce a dedicated server for load-distribution, the server can be a performance bottleneck and the single point of failure. Therefore, Hsiao [18] proposed a decentralized algorithm for load-distribution. Although the proposed method uses a dedicated server, it would be possible to adopt a decentralized algorithm and remove the dedicated server.

3. uTupleSpace with Dynamic-Help Method

In this section, we describe our proposed dynamic-help method, which balances the load dynamically and makes uTupleSpace scalable. This method balances the load among servers existing in the uTupleSpace as much as possible and requests the system operator to add a new server to the uTupleSpace only if the load increases and reaches the

Table 1 Data format of uTuple.

Metadata	Address	Identifier of sensor/actuator
	Time	Time when data were stored by sensor/actuator
	Position	Position (latitude and longitude) of sensor/actuator
	Subject	Type of sensor/actuator
	Type	Type of data
Data	(user-defined)	

capacity of all the existing servers. This approach effectively utilizes existing resources and keeps equipment investment at a minimum.

The load can be classified as a processing load or memory load. The former includes CPU load and I/O load. The latter refers to the amount of data accumulated on a hard disk or to hard drive utilization. In the following section, we describe the details of the uTupleSpace model and the basic idea of the proposed method. We also give an overview of the system and present examples of a dynamic load balancing process for processing load and memory load. In addition, we discuss the process for adding a new server. We also discuss the additional processes for achieving better performance in the Appendix.

3.1 uTupleSpace Model

In the uTupleSpace model, we introduce the metadata fields in a tuple. The new tuple, called *uTuple*, consists of metadata and data. Table 1 lists the uTuple data format. The metadata contains an address (ID of the device), the time the data were created, and the position at which the device resides. Value ranges are accepted for these fields. The metadata also contains the type of device and data. For these fields, value ranges are not accepted, and exact matching must be performed in order to use these fields as a DHT distribution key.

The uTupleSpace model supports two types of communication; event communication achieves selective read, while command communication achieves selective write (Fig. 1). The former is a style original to the tuple space model; the latter is an extension we proposed. In short, the metadata of a uTuple pair used in each type is different. That is, in the event communication, the writer sets an exact value, and the reader sets the range condition; in the command communication, the roles are reversed.

In event communication, the writer (sensor program) registers the *EventActual* (EA) uTuple, which consists of the writer's own entries in the metadata and data. The reader (application program) registers the *EventFormal* (EF) uTuple, which consists of templates in the metadata and data for matching the desired data and writers. In command communication, the reader (actuator program) registers the *CommandFormal* (CF) uTuple, which consists of the reader's own entry in the metadata and a template in the data for matching the desired command data. The writer (application program) registers the *CommandActual* (CA) uTuple, which consists of an entry in the data and a template in the

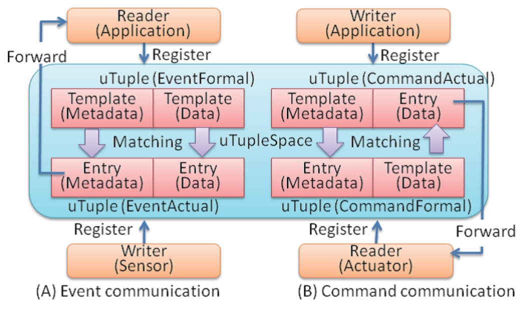


Fig. 1 Model of uTupleSpace.

metadata for matching the desired readers. These communications achieve selective read and write processes with, for example, a space range or time range. The scope of the uTuple for read or write is thus flexible.

When a uTuple is registered to uTupleSpace, the uTuple is compared with its counterparts first (matching process). If the uTuple is EA, the counterpart is EF; if the uTuple is CA, the counterpart is CF, and vice versa. If the EA or CA is respectively matched to EF or CF in the matching process, the EA or CA is forwarded to the reader that registered the EF or CF. Second, the uTuple is written to the hard disk and accumulated (perpetuating process). A lifetime can be set for each uTuple, and expired uTuples are removed via garbage collection, which is executed periodically. When the uTuple whose lifetime is set as 0 is registered, the matching process is executed, but the perpetuating process is skipped.

3.2 Basic Idea of Dynamic-Help Method

When the processing of server A is overloaded, the idle server B partially takes over the load as follows using the proposed method. The sensor programs register the EAs, which should all be registered to server A up to that time, to either A or B, so as to partially move the load of A to B. However, the application programs register the EFs, which should all be registered to A up to that time, to both A and B. This is because if they register the EFs to either A or B, they cannot fully communicate with the sensor programs as the program developer intended.

It is okay for EFs to be registered to one server and EAs to be registered to both servers in order to achieve communication integrity. However, to achieve efficient load-distribution, EFs should be registered to both servers, and EAs to one server. This is because the amount of sensor data is prone to be enormous, while the number of search queries is prone to be much fewer in most ubiquitous applications. That is, the number of EAs registered by sensor programs is much higher than that of EFs registered by application programs. This characteristic enables the proposed method to achieve high load-distribution efficiency and to reduce the overhead caused by tuple duplications.

Though the proposed method applies similar considerations to CF and CA, they cannot be applied in exactly the same way. According to our analysis, the relationship between the quantities of accumulated EAs and EFs is similar to the relationship between the frequencies of registration of EAs and EFs. That is, *the number of accumulated EAs ≫ the number of accumulated EFs*, and *the registration frequency of EAs ≫ the registration frequency of EFs*. In contrast, CF and CA have different tendencies; that is, *the number of accumulated CFs ≫ the number of accumulated CAs*, and *the registration frequency of CFs ≪ the registration frequency of CAs*. The reason for this is as follows.

First, an actuator device is controlled more than once. This means that an actuator device accepts two or more CAs. Also, the number of CFs is proportional to the number of actuator devices. Therefore, the formula *the registration frequency of CFs ≪ the registration frequency of CAs* becomes true. Second, because the control of actuator devices affects the real world, most applications control only the particular actuator devices that they have permission to control, and only at the time that the applications issue the control command. For this reason, most CAs are registered with the address field set to a particular device ID and the lifetime set to 0 (we refer to this simple type of CA as sCA). The perpetuating process is skipped in the registration of sCA. Therefore, the formula *the number of accumulated CFs ≫ the number of accumulated CAs* becomes true. Because of the characteristics of CF and CA, the simple method in which a CF is registered to either A or B and a CA is registered to both A and B, as with the EAs and EFs, causes a problem in that the large CA load is placed on both A and B.

Therefore, a server that is specialized to process sCA is newly introduced in the proposed method. This server is called the sCA server. When a CF is registered, the CF is duplicated; one is registered to either A or B, as with EA. The other is registered to the sCA server for matching with sCA.

When a CA is registered, the CA is first evaluated to determine whether or not it is an sCA. If the CA is not an sCA, it is registered to both A and B as with the EF. If the CA is an sCA, it is registered to the sCA server and compared with the accumulated CFs. At the sCA server, all the matching process needs to do is to compare the three fields (subject/type/address) of uTuples and determine whether the values match each other exactly or not. Moreover, because the lifetime of an sCA is 0, the perpetuating process can be skipped. Thus, since only simple processes are executed on the sCA server, high throughput can be achieved. Even when the sCA server requires load-distribution, the traditional load-distribution methods of DHT which concatenate the three fields and use it as a distribution key would function well.

When server A's memory is overloaded, idle server B partially takes over the load. In this case, no more uTuples can be registered to A. However, if all uTuples are registered only to B, communication integrity will be undermined. Therefore, the application program registers EFs/CAs not only to B but also to A, and it skips the perpetuating process in the registration to A by setting the lifetime of the uTuple as 0. EFs/CAs accumulated to A by that time are copied to B. This enables EFs/CAs accumulated to A to be properly compared to newly registered EAs/CFs. Because the number of accumulated EFs is equal to the number of applications that search EAs continuously, and the number of accumulated CAs is equal to the number of CAs that are not sCAs, the numbers of accumulated EFs/CAs are both relatively small, and the overhead of the migration process for EFs/CAs is also small.

3.3 System Overview

An overview of a system in which the proposed method is applied is shown in Fig. 2. The uTupleServer executes the matching process and accumulates uTuples. The Load Monitoring Server monitors the load of each uTupleServer and balances them as necessary. The Assignment Manager Server manages the assignment table, which indicates which uTuple should be registered to which uTupleServer. The assignment table lists the correspondence between each section of divided hash space and the addresses of the uTupleServers in charge of the section. Additionally, an accumulatable flag is added to each address, which will be described later. Two or more Assignment Manager Servers are installed, and they manage the assignment table in a distributed manner with a DHT. The DHT is formed by the peers running on the Assignment Manager Servers. The same number of peers as that of uTupleServers are invoked using virtual servers.

The DHT is a bit different from usual DHTs though the basic algorithms such as query routing are the same. Each peer holds information of the range of the section which the peer is in charge of in the whole hash space, and the list of the addresses of the uTupleServers which are in charge of the section and the accumulatable flags. When the peer receives a query, it does not return data but returns the list. When a new uTupleServer is added, a new peer is also invoked and added to the DHT. The DHT achieves high throughput because the amount of information held by each peer is very small, it can be kept on memory, and the peer can execute query-processing with memory access only. The sCA server executes the matching process between sCAs and CFs and accumulates CFs. We assume that these servers are connected by Ethernet. The proposed method is achieved with these servers.

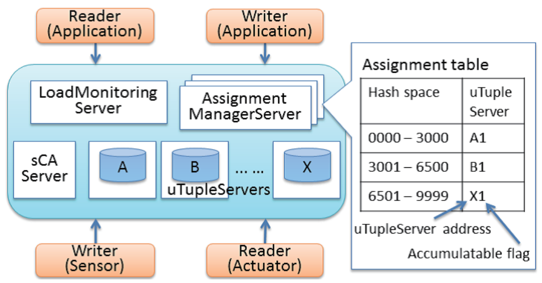


Fig. 2 System overview.

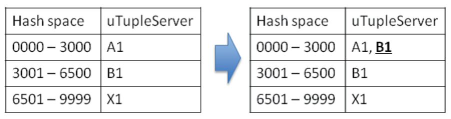
The advantage of the proposed method is that it can distribute the load caused by uTuples that have the same distribution key. The proposed method separates the main load, that is, matching processes and perpetuating processes, from peers of the DHT, and makes uTupleServers burden the load in a distributed manner. When the system scale is small enough, only one Assignment Manager Server is sufficient. In such a case, the assignment table does not need to be managed in a distributed manner with DHT.

The client program concatenates the specific values in a uTuple (the subject key and type key values) and obtains a distribution key with a hash function applied with the concatenated value. It then looks up the distribution key in the DHT and obtains addresses of uTupleServers to be registered using the assignment table. Next, it registers the uTuple to the uTupleServers. However, when a CF is registered, in addition to the registration to the uTupleServer described above, the client program also registers it to the sCA server. When an sCA is registered, the client program does not register it to uTupleServer but only to the sCA server.

The registration to uTupleServer requires specific values of the subject key (the type of device) and type key (the type of data) in the uTuple to determine which uTupleServers to register the item to. An application program for the ubiquitous environment, however, is always aware of the values of these keys. This assumption is quite natural, and there will be no practical restrictions.

3.4 Example of Dynamic Distribution of Processing Load

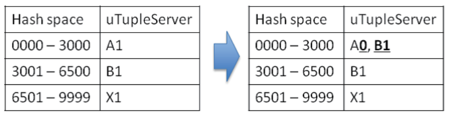
1. The Load Monitoring Server detects that the processing load of uTupleServer A exceeds a threshold level T_p.
2. The Load Monitoring Server selects uTupleServer B, which has the lightest load among the uTupleServers at that time, as a helper server to share the load of A.
3. The Load Monitoring Server makes A copy the EFs and CAs that have accumulated in A up to that time, to B.
4. The Load Monitoring Server makes the Assignment Manager Servers add the address of B with the accumulatable flag ON to the line including the address of A in the assignment table. Figure 3 shows how the assignment table is updated by this operation. The accumulatable flag set to ON is indicated by 1 in the figure.
5. The reader and writer obtain the addresses of both A and B from the Assignment Manager Server when they look up the distribution key related to the section that A is in charge of. When the application program registers EF or CA, it registers it to both A and B. When the device program registers EA or CF, it randomly selects one from A and B, and registers the EA or the CF to the selected one.



Hash space	uTupleServer
0000 – 3000	A1
3001 – 6500	B1
6501 – 9999	X1

Hash space	uTupleServer
0000 – 3000	A1, **B1**
3001 – 6500	B1
6501 – 9999	X1

Fig. 3 Example of updating assignment table by dynamic distribution of processing load.



Hash space	uTupleServer
0000 – 3000	A1
3001 – 6500	B1
6501 – 9999	X1

Hash space	uTupleServer
0000 – 3000	A**0**, **B1**
3001 – 6500	B1
6501 – 9999	X1

Fig. 4 Example of updating assignment table by dynamic distribution of memory load.

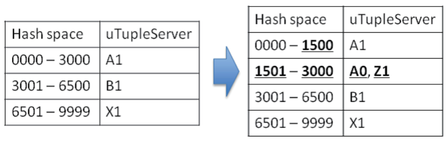
3.5 Example of Dynamic Distribution of Memory Load

1. The Load Monitoring Server detects that the memory load of uTupleServer A exceeds a threshold level T_m.
2. The Load Monitoring Server selects uTupleServer B, which has the lightest load among the uTupleServers at that time, as a helper server to share the load of A.
3. The Load Monitoring Server makes A copy the EFs and CAs that have accumulated in A up to that time, to B.
4. The Load Monitoring Server makes the Assignment Manager Server set the accumulatable flag of A to OFF and add the address of B with the accumulatable flag ON to the line including the address of A in the assignment table. Figure 4 shows how the assignment table is updated by this operation.
5. The reader and writer obtain the address of A with accumulatable flag OFF and the address of B with accumulatable flag ON from the Assignment Manager Server when they look up the distribution key related to the section managed by A. When the application program registers EF or CA, it registers it to A with the lifetime set as 0 and to B with the original lifetime. When the device program registers EA or CF, it registers it to B, whose accumulatable flag is ON.

3.6 Addition of New uTupleServer to uTupleSpace

If the load increases and reaches the capacity of all the existing uTupleServers, a new uTupleServer must be added to the uTupleSpace. To add a new uTupleServer, we adopt a method in which the new uTupleServer partially takes over a section of hash space from an existing uTupleServer. This method is similar to that of adding a new peer to a DHT; however, the existing uTupleServer does not move all the uTuples, in which the distribution key is included in the section taken over by the new uTupleServer, to the new uTupleServer because the cost of doing so is prohibitively high, especially for EAs/CFs. An example of this method is described below.

1. A new uTupleServer, Z, sends an add-server-request to the Load Monitoring Server.
2. The Load Monitoring Server selects uTupleServer A, whose load is the heaviest among uTupleServers at that



Hash space	uTupleServer
0000 – 3000	A1
3001 – 6500	B1
6501 – 9999	X1

Hash space	uTupleServer
0000 – **1500**	A1
1501 – **3000**	**A0**, **Z1**
3001 – 6500	B1
6501 – 9999	X1

Fig. 5 Example of updating assignment table by addition of new uTupleServer.

time, and makes the Assignment Manager Server divide the section managed by A. Specifically, uTupleServer A randomly selects a uTuple from accumulated uTuples, and returns the hash value of the uTuple to the Load Monitoring Server. The Load Monitoring Server requests the Assignment Manager Server to invoke a new peer corresponding to Z and to add it to the DHT with the hash value and addresses of A and Z as parameters. The Assignment Manager Server invokes a new peer and adds it to the DHT. In the addition process, the peer in charge of the section containing the hash value divides the section in half, takes charge in one side by itself, makes the new peer take charge in the other side, and copies the list of uTupleServer to the new peer.

3. The Load Monitoring Server makes A copy the EFs and CAs that have accumulated in A up to that time and that have the distribution key included in one side of the divided sections, to Z.
4. The Load Monitoring Server makes the Assignment Manager Server set the accumulatable flag of A to OFF and add the address of Z with the accumulatable flag ON to the section. Figure 5 shows how the assignment table is updated by this operation.
5. The subsequent operation is the same as step 5 of the dynamic distribution of memory load.

4. Experiments

We evaluated the performance and the feasibility of the proposed method. The results are described below.

4.1 Performance Evaluation

We ran a micro-benchmark on a uTupleServer to evaluate the performance from the aspects of throughput and memory capacity. From the micro-benchmark results, we estimated the overall performance of the whole system under the load of assumed applications.

Although it is ideal with this method to increase the performance of uTupleSpace in proportion to the number of uTupleServers, the performance actually depends on the load distribution. If the load is uniformly distributed over hash space, that is, the amounts of data and access related to each hash value are equal to each other, the load would be distributed equally among the uTupleServers by the DHT without dynamic load-balancing, and uTupleSpace would provide ideal performance (hereinafter called the best condition). If the load is distributed unequally, it would be necessary to copy EFs and CAs, which would cause processing and memory overhead. The worst condition is the condition where the overhead reaches a maximum level because the load is concentrated at one hash value. The traditional methods cannot distribute the load in the worst condition. Therefore, we evaluated how the throughput and the number of accumulatable uTuples vary with the number of uTupleServers in the ideal condition and the worst condition.

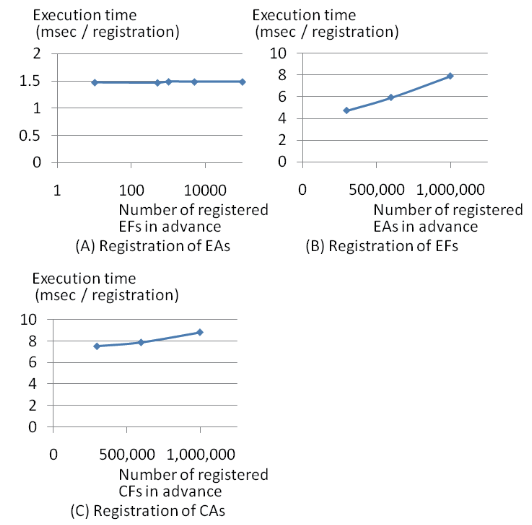


Fig. 6 Execution time for registering uTuples.

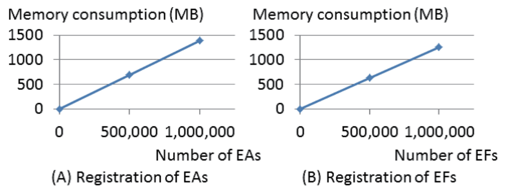


Fig. 7 Memory consumed in accumulating uTuples.

4.1.1 Results of Micro-Benchmark

The host for the uTupleServer was Xserve (Quad-Core Intel Xeon 2.8 GHz, 2 GB memory). We registered uTuples of EA, EF, and CA to the uTupleServer to which the uTuples of EF, EA, and CF were respectively registered in advance.

Figure 6 plots the variation in execution time for registration with respect to the number of uTuples registered in advance. In each registration, one uTuple matched another uTuple in a matching process. Figure 7 plots the result of measuring the memory size for accumulating EAs and EFs. Each EA has a small quantity of data, which are assumed in applications A and B (described later), in the body field.

4.1.2 Overall Performance Estimation

On the basis of the results of the micro-benchmark test, we estimated the performance of an entire system consisting of two or more uTupleServers. We assumed the load of two typical applications as shown in Table 2. Application A uses

Table 2 Load of assumed applications.

		Application A	Application B
Event	# of registered EAs	350D	720D
	# of registered EFs	D/10000	D/10000
	EA registration freq.	50D (per day) [1]	12D (per day) [1]
	EF registration freq.	0.1 (per sec) [1]	0.1 (per sec) [1]
Command	# of registered CFs	0	D
	# of registered CAs	0	0
	CF registration freq.	0	1 (per sec) [0]
	CA registration freq.	0	12D (per day) [1]

D: # of devices, [●]: # of matched uTuples in each registration

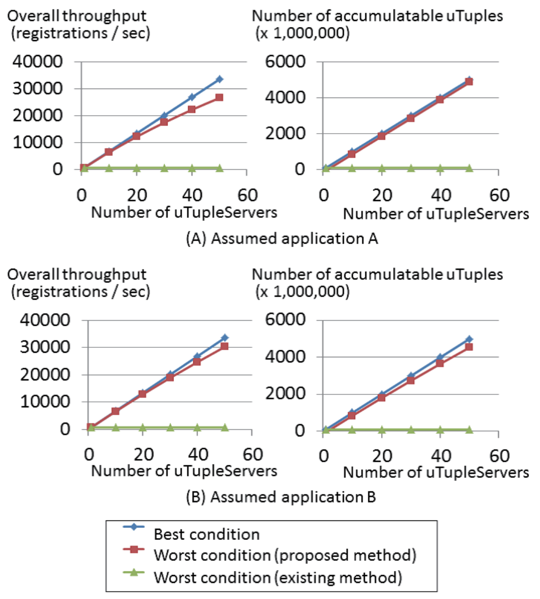


Fig. 8 Overall performance.

sensor data. Application B mainly uses actuators. Application A is an application for improving the efficiency of physical distribution using sensors that sense the quantity of a truckload. Although the event communication load for sensor data accumulation is large, The command communication load is 0. The loads of applications that mainly monitor things by using sensor devices will have such a load. Application B is an application for remote management of information appliances. The event communication load for status confirmation and the command communication load for sending control commands are both large. All CAs are sCAs. The loads of applications that control things by using sensor and actuator devices will have such loads.

Figure 8 plots the overall throughput (the number of registered uTuples per second) and the number of accumulatable uTuples under these assumed loads. We estimated these values for the existing and proposed methods under the best and worst conditions. The existing method is the load-distribution method of traditional DHTs. It cannot distribute the load caused by uTuples that have the same distribution key. The situation in which the distribution keys in uTuples are spread evenly is the best condition. In this situation, the load caused by the uTuples is distributed evenly to uTupleServers in both the existing and proposed methods. On the other hand, the situation in which all distribution keys in uTuples are the same is the worst condition. In this situation, the load of the uTuples is concentrated to one uTupleServer in the existing method. In the proposed method, all EFs and CAs are copied to all uTupleServers. This leads to an increase in memory load caused by EFs and CAs and an increase in the processing load for the matching processes in the registration of EAs and CFs. These additional loads affect the overall throughput and number of accumulatable uTuples on the uTupleServers. The size of the effect can be determined from Figs. 6 and 7.

In estimating the overall throughput and the number of accumulatable uTuples, we assumed that the results shown in Figs. 6 and 7 are linear, the uTuples of each type consume processing resources and memory resources independently, and a mix of uTuple types does not cause a gain or loss of the amount of consumption. For example, when the registration of EAs takes 200 msec and the registration of EFs takes 100 msec, we estimate that the registration of the EAs and EFs takes 300 msec. We also assumed that the memory capacity is 140 GB per uTupleServer, and that the processing load for copying EFs and CAs and the network delay can be ignored. As described previously, the DHT running on the Assignment Manager Servers achieves high throughput because the peers can execute query-processing with memory access only. For example, it has been reported that one node of memcached [19] can process about one hundred thousand queries per second. The peer on memory would achieve the equivalent performance. As the sCA server also keeps a relatively small amount of information and executes simple processes, which search the hash table, the sCA server is expected to achieve equivalent performance. In addition, it is possible to distribute the load with the DHT. Therefore, we assumed that the Assignment Manager Server and the sCA server exhibit sufficient good performance not to affect the performance of the uTupleServers. We also assumed that the traffic from the Load Monitoring Server is small enough not to affect the performance of the uTupleServers.

In the proposed method, communication among uTupleServers is not needed in a steady state. In other words, a uTupleServer is independent of other uTupleServers. The performance of a uTupleServer does not depend on the performance or the load of other uTupleServers. Therefore, we were able to estimate the overall throughput and number of accumulatable uTuples in a steady state from the micro-benchmark result on a uTupleServer.

4.1.3 Discussion of Estimated Performance

In the best condition, dynamic load-balancing of the proposed method does not occur, and the performance is the same as that of the existing method. The effect of the proposed method is seen in the worse condition, where the load is distributed unequally. As can be seen in Fig. 8, under the load of both applications, the performance of the proposed method is dramatically improved in the worst condition and

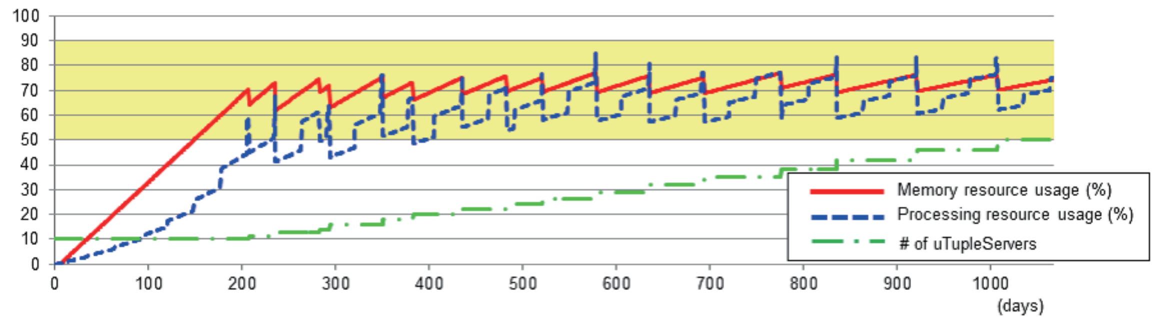


Fig. 9 Simulation result.

comes close to the performance in the best condition.

Therefore, we can say that the proposed method distributes both loads of two typical applications with low overhead even if the load is concentrated on a hash value.

4.2 Feasibility Evaluation

We conducted an operational simulation of uTupleSpace under the load of application A. Although the performance evaluation showed that the proposed method can distribute the load of applications A and B with low overhead, that is not enough to apply the proposed method to a real system operation. In a real system operation, the system needs to start off on a small scale in terms of the number of servers, the load needs to be monitored, and more servers have to be added as necessary. In the operation, the proposed method has to enable the operator to balance a small investment in equipment with stable operation that maintains some margin of system resource usage. Therefore, to confirm whether such operation can be achieved with the proposed method, we conducted the operational simulation. In the simulation, we assumed the following conditions.

- The number of sensor devices increases by 10,000 each day, and reaches ten million after about three years.
- A month is needed from the decision to buy a server to the installation of the server in the system because of necessary procedures such as the actual purchase and the configuration.

The operational rules are simple, as indicated below.

- The system starts with ten uTupleServers.
- The operator predicts the load of 60 days later based on the change in the load in the last week with linear prediction every day.
- When the resource usage is predicted to exceed 90% two months later, the operator decides to buy as many servers as the system needs in order to reduce the resource usage to below 70%.

The simulation result is shown in Fig. 9. The graph shows the time variation of the average resource usage and the number of uTupleServers in the operational simulation following the above operational rules. As can be seen in the figure, the number of uTupleServers increases little by little and reaches 50 on the 1000th day, by which point the number of sensor devices has reached ten million. And all during that time, both the memory resource usage and processing resource usage fall within the range from 50% to 90%, and maintain an average of about 70%. In fact, the averages of the memory resource usage and the processing resource usage between the 500th day and the 1000th day are 72.8% and 66.6% respectively.

The proposed method can thus be said to achieve efficient system operation that requires a low investment and that maintains stability even under the load of ten million devices.

5. Conclusion

We proposed a new load-distribution method called the "dynamic-help method" that enables uTupleSpace to distribute load dynamically. Experimental results indicated that our method achieved good scalability under the load of typical applications using sensor and actuator devices and good feasibility for economical and stable operation.

References

[1] T. Nakamura, M. Nakamura, A. Yamamoto, K. Kashiwagi, Y. Arakawa, M. Matsuo, and H. Minami, "uTupleSpace: A bi-directional shared data space for wide-area sensor network," 2nd International Workshop on Sensor Networks and Ambient Intelligence (SENAMI 2009), pp.396–401, 2009.

[2] Y. Arakawa, K. Kashiwagi, T. Nakamura, M. Nakamura, and M. Matsuo, "Dynamic scaling method of uTupleSpace data-sharing mechanism for wide area ubiquitous network," Proc. Asia-Pacific Symposium on Information and Telecommunication Technologies (APSITT), pp.1–6, June 2010.

[3] D. Gelernter, "Generative communication in Linda," ACM Transactions on Programming Language and Systems, vol.7, no.1, pp.80–112, Jan. 1985.

[4] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A.L. Murphy, and G.P. Picco, "TinyLIME: Bridging mobile and sensor networks through middleware," Proc. International Conference on Pervasive Computing and Communications (PerCom '05), pp.61–72, 2005.

[5] P. Costa, L. Mottola, A.L. Murphy, and G.P. Picco, "TeenyLIME: Transiently shared tuple space middleware for wireless sensor networks," Proc. International Workshop on Middleware for Sensor Networks (MidSens '06), pp.43–48, 2006.

[6] G. Castelli, A. Rosi, M. Mamei, and F. Zambonelli, "A simple model and infrastructure for context-aware browsing of the world," Proc.

International Conference on Pervasive Computing and Communications (PerCom'07), pp.229–238, 2007.

[7] G. Hackmann, C.-L. Fok, G.-C. Roman, and C. Lu, "Agimone: Middleware support for seamless integration of sensor and IP networks," Proc. International Conference on Distributed Computing in Sensor Systems (DCOSS'06), pp.101–118, 2006.

[8] C.-L. Fok, G.-C. Roman, and C. Lu, "Rapid development and flexible deployment of adaptive wireless sensor network applications," Proc. International Conference on Distributed Computing Systems (ICDCS'05), pp.653–662, 2005.

[9] C.-L. Fok, G.-C. Roman, and G. Hackmann, "A lightweight coordination middleware for mobile computing," LNCS (COORDINATION 2004), vol. 2949, pp.135–151, 2004.

[10] Y. Jiang, G. Xue, Z. Jia, and J. You, "DTuples: A distributed hash table based tuple space service for distributed coordination," Grid and Cooperative Computing, Fifth International Conference, pp.101–106, Oct. 2006.

[11] U. Wickramasinghe, C. Wickramarachchi, P. Fernando, D. Sumanasena, S. Perera, and S. Weerawarana, "BISSA: Empowering Web gadget communication with tuple spaces," Proc. Gateway Computing Environments Workshop (GCE), pp.1–8, Nov. 2010.

[12] S. De, S. Nandi, and D. Goswami, "On performance improvement issues in unordered tuple space based mobile middleware," Proc. 2010 Annual IEEE India Conference (INDICON 2010), pp.1–5, Dec. 2010.

[13] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," Proc. SIGCOMM '01, pp.149–160, San Diego, California, United States, 2001.

[14] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in structured P2P systems," Proc. 2nd Int'l Workshop Peer-to-Peer Systems (IPTPS'02), pp.68–79, Feb. 2003.

[15] S. Surana, B. Godfrey, K. Lakshminarayansan, R. Karp, and I. Stoica, "Load balancing in dynamic structured P2P systems," Performance Evaluation, vol.63, no.6, pp.217–240, March 2006.

[16] C. Chen and K.-C. Tsai, "The server reassignment problem for load balancing in structured P2P systems," IEEE Trans. Parallel Distrib. Syst., vol.12, no.2, pp.234–246, Feb. 2008.

[17] L. Yang and Z. Chen, "A VS-split load balancing algorithm in DHT-based P2P systems," Proc. 2012 International Conference on Systems and Informatics (ICSAI 2012), pp.1581–1585, 2012.

[18] H.-C. Hsiao, "Load balance with imperfect information in structured peer-to-peer systems," IEEE Trans. Parallel Distrib. Syst., vol.22, no.4, pp.634–649, April 2011.

[19] Dormando, "memcached," http://memcached.org/, accessed Oct. 25, 2013.

Appendix A: Processes for Better Performance

The number of sections a uTupleServer is in charge of increases with dynamic load balancing. If the number or type of uTuples registered to uTupleSpace changes constantly and dynamic load balancing occurs frequently, every uTupleServer can be in charge of all sections of the hash space (hereinafter called the worst condition). Because all EFs and CAs are copied to all uTupleServers in the worst condition, the overhead reaches a maximum.

To avoid such a situation, we add some of the following processes. Although we report the evaluation results for the worst condition in Sect. 4, we can avoid the worst condition as much as possible with these processes.

A.1 Assignment Manager Server Preferentially Selects uTupleServer, which is Already in Charge of Section that Overloaded uTupleServer is in Charge of, as Helper Server

In the proposed method, a section of hash space is managed by one or more uTupleServers with accumulatable flag ON and zero or more uTupleServers with accumulatable flag OFF. If any of these uTupleServers becomes overloaded, the Assignment Manager Server preferentially selects another one of the uTupleServers as a helper server. This prevents an increase in uTupleServers that take charge of new sections due to dynamic load balancing. For example, if uTupleServer A with accumulatable flag ON is overloaded, and uTupleServer B, which is in charge of the same section with accumulatable flag ON, has a low load, B should be the helper server. In this case, the load of A can be moved simply by setting the flag of A to OFF. If B has already set its flag to OFF, the load of A can be balanced simply by setting the flag of B to ON. In these cases, it is not necessary to copy EFs and CAs, unlike in the above example of dynamic distribution.

A.2 Dynamic Distribution of Processing Load is Executed Gradually

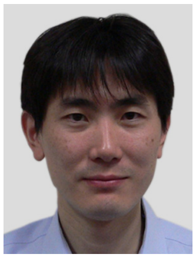
In the proposed method, a uTupleServer is in charge of more than one section of hash space. If the processing of a uTupleServer is overloaded, the Assignment Manager Server does not need to select a helper server for each section of which the overloaded uTupleServer is in charge. Even one helper server can be enough to reduce the load. Because having more helper servers than necessary results in a worse condition, dynamic distribution of the processing load is not executed all at once, but only partially. If the uTupleServer is still overloaded, dynamic distribution is executed again. This prevents an increase in uTupleServers that manage new sections due to dynamic load balancing.

A.3 Load Monitoring Server Confirms Necessity for uTupleServers to Manage Each Section of Hash Space

New uTuples are not accumulated in the uTupleServer with accumulatable flag OFF. In addition, each uTuple has a lifetime, and an expired uTuple is removed by the Garbage Collection process mentioned in Sect. 3.1. Therefore, all the uTuples that are accumulated in the uTupleServer will be removed over time. In such a situation, the counterparts of EF/CA no longer exist on the uTupleServer. This means the uTupleServer does not have to be in charge of that section. Thus, the Load Monitoring Server queries the uTupleServers to find out whether there are any sections that a uTupleServer is managing that have no uTuples. If there are, the Load Monitoring Server orders the uTupleServer to stop managing those sections. This means the address of the uTupleServer is removed from the line corresponding to the section in the assignment table.

Yutaka Arakawa graduated from the University of Tokyo with a B.E. and an M.E. degree in Electoronic Engineering in 2001 and 2003. He joined NTT in 2003. He is currently working on energy management systems at the NTT Comware Corporation. His reearch interests include distributed systems and data structure design.

Keiichiro Kashiwagi received the B.E., M.E. and D.E. degrees in Information Engineering from Waseda University, Tokyo, Japan, in 2006, 2008 and 2013 respectively. He is currently working for NTT Network Innovation Laboratories, and his recent work is focused on software platforms for ubiquitous services. He is a member of JSIAM.



Takayuki Nakamura received his B.E. and M.E. degrees in computer science from the University of Tokyo, Japan, in 1995 and 1997. Since joining NTT in 1997, he has been working in the field of distributed system software, network communities, network storage, and fundamental software for sensor network services. He is a member of IPSJ.

Motonori Nakamura received the B.E. and M.E. degrees in information engineering from Nagoya University, Aichi, in 1990 and 1992, respectively. Since joining NTT Switching System Laboratories in 1992, he has been researching distributed telecommunication service software architectures, ad-hoc routing protocols, and software platforms for ubiquitous services. He is currently working at NTT Network Innovation Laboratories as a Senior Research Engineer, Supervisor. He is a member of IPSJ.