Journal of Information Processing Vol.21 No.3 450–457 (July 2013)

[DOI: 10.2197/ipsjjip.21.450]

Regular Paper

Jobcast — Parallel and distributed processing framework Data processing on a cloud style KVS database

Ikuo Nakagawa[1,2,a)] Kenichi Nagami[1]

Received: October 22, 2012, Accepted: March 1, 2013

Abstract: In this paper, we propose a new architecture for parallel and distributed processing framework, "Jobcast," which enables data processing on a cloud style KVS database. Nowadays, many KVS (as known as Key Value Store) systems exist which achieve high scalability for data spaces among a huge number of computers. The Jobcast architecture is an extension which has the capability to execute "job" on KVS data nodes so that it can also achieve scalability of processing space. In this paper, we introduce the Jobcast architecture and describe how Jobcast improves performance of some KVS applications especially by reducing data transmission cost. We evaluate and discuss performance improvement for some example applications as well.

Keywords: parallel and distributed processing, cloud database, NoSQL, KVS

1. Introduction

Recently, there are many cloud style databases, as known as "NoSQL." One of major goals of such cloud style databases is achieving scalability. All of such databases consist of a large number of computers as data nodes, and it is easy to add a data node to the existing computer cluster, to increase data space or improve access performance. This type of computing style is called as "scale-out style computing" or "scalable computing."

KVS, as known as Key Value Store, is a mechanism to store key and value pairs. KVS systems provide interfaces to access such pairs. For example, SET, GET and DELETE (or similar to such commands) are major interfaces to handle key and value pairs in KVS systems. Cloud style KVS systems have a large number of computers as data nodes to store many key and value pairs. Such KVS systems are designed to be scalable and it is easy to increase data spaces or to improve performance to access data objects. Consistent hash [1], [2] is a major algorithm to achieve scalability for data spaces. Some KVS implementations use consistent hash algorithm as a distribution strategy and use the hashed value of the key, to identify the data node among a large number of computers to store the key and value pair. These KVS systems are also categorized as "NoSQL" databases.

In this paper, we propose "Jobcast," which is a new architecture for parallel and distributed processing on a cloud style KVS database. Jobcast uses consistent hash as its distribution strategy and has a capability to store key and value pairs in a distributed manner, as a KVS system. Jobcast also has a capability to deliver "jobs," where jobs are processing logic for key value pairs stored in data nodes. In Jobcast architecture, application programmers may define jobs as simple Java programs, that is, simple class definitions. Jobcast distributes such jobs into data nodes for remote processing, using the same distribution strategy. When a data node receives such jobs, the node executes jobs in its processing engine and returns the result.

Jobcast provides a new processing model for some KVS applications, and it improves data transmission behavior by migrating the processing mechanism into backend data nodes. Note that Jobcast is processing framework inside a cloud system, and programming interface will be discussed in future researches.

In this paper, we survey some related works about KVS implementations at first, in Section 2. We describe assumptions and objectives of Jobcast in Section 3 and introduce basic concept of Jobcast in Section 4. We also describe features of Jobcast, including distributing jobs into data nodes and its processing models. We describe benefits of Jobcast, as parallel and distributed processing, in Section 5. We summarize the evaluation result of a typical example for parallel and distributing processing, in Section 6 and discuss further considerations in Section 7. Finally, we conclude our proposal in Section 8.

2. Related Works

Nowadays, many KVS style databases exist. There are some KVS models. At first, we survey such KVS technologies by categorizing them into 3 models.

2.1 Historical Model

Historically, memcached [3] has been used to improve performance of relational database access. In this model, an application system has many memcached servers which are located in front of SQL database, and those servers will have many key and value pairs in its memory space (as cache), while applications will write such pairs into SQL database, directly. Memcashed is designed to enhance read access performance for such key-value pairs with existing relational databases. Note that, there is no significant

[1] Intec, Inc., Koto, Tokyo 136–8637, Japan
[2] Osaka University, Suita, Osaka 565–0871, Japan
[a)] ikuo@inetcore.com

change of processing model in this case.

2.2 Standard KVS

Cassandra [4] is a standard KVS system, which provides simple and basic feature to store, get or delete key value pairs into distributed data store. It supports not only read operation but also write operation, that is it provides data persistency of its data objects, so that any data will survive after write operation. Cassandra supports both synchronous and asynchronous replication on data updates.

Tokyo Cabinet [5] and Kyoto Cabinet [6] are also standard KVS systems, which are simple and high performance implementations provided by FA labs. Standard KVS systems do not have a processing mechanism in backend data nodes, and programmers have to develop processing mechanism in application servers.

2.3 Extendable KVS

ROMA [7] is a KVS implementation written in Ruby, which support not only standard KVS features but also a module extension to execute some functions in backend KVS data nodes. Modules can be written in specific language and provides some simple routines. It is still under development and we need future research to understand its characteristics.

Okuyama [8] is also an extendable KVS system, which is written in Java. It uses HashMap (standard class in Java SDK) to store key value pairs in data nodes, and it also writes operational log and serialized HashMap objects into persistent disk, periodically. Okuyama has the idea to implement a mechanism for server side processing, i.e., processing in data nodes. It will recognize JavaScript code in data nodes, and processes in a distributed way, but we need further research about its use cases and its evaluation.

3. Assumptions and Objectives

Nowadays, high scalability is a critical requirement for some cloud style applications, such as mail, chat, blog, address book, network storage, etc. provided by big service providers. For example, Apple, Google, and Facebook have billions of consumer users in the world. Big mobile carriers also have tens of millions customers, as well (China Mobile for example has a half billion users). Such applications require high scalability for both data space and the number of simultaneous accesses since there are a huge number of small and well distributed data objects and process executions.

KVS systems are one of solutions to achieve scalability as backend data store for such applications. KVS provides scalability of data space and many developers have been tried to implement their applications with KVS systems.

On the other hand, defining and selecting data schema is a difficult problem in developing KVS applications. For example, considering SNS (Social Network Service) like application. A user of the application has his profiles and blog entries. From an application view point, each user has user data consisted with some items. We assign a unique user-id for each user, and each item is identified by user-id and item-id.

To simplify the discussion, we denote two major design patterns to define data schema for such KVS applications (**Table 1**).

Table 1 Data schemas for KVS applications.

	Pros.	Cons.
Item base	Easy to access an item, individually	# of data transmission might be large to access entire contents
User base	Easy to access entire contents in a same time	Data transmission volume might be large to access individual item

(1) Item base;
Assigning “user-id+item-id” as keys of the KVS ID space, and store values for each item as KVS values.

(2) User base;
Assigning “user-id” as keys of the KVS ID space and store the entire associated contents as KVS values.

Note that choosing data schema has some difficulties, because there is trade-off issue of data transmission cost, which is serious in some situations. In this section, we summarize about the trade-off, briefly (We describe more detail in Section 5 and Section 6).

Item base data schema is suitable for accessing each item individually. Instead, it may require a large number of data transmissions between the application server and backend KVS data store, to get the entire contents (or, set of items).

User base data schema is suitable for accessing entire contents (or, set of items). Instead, if the size of the entire contents is big enough, data transmission volume (bytes) might be large only for accessing individual small item.

In following discussions, we focus into KVS applications, which have the following characteristics.

Assumption 1.
There are a large number of users. For example, millions or more users exists.

Assumption 2.
Each user may have several numbers of items. For example, there are 10 or more items in user data.

Assumption 3.
Some of item values may be big enough to transmit. For example 1 MB data takes 8 ms via 1 Gbps NIC.

Assumption 4.
The application accesses user data in several ways. At least, it accesses individual items and entire contents.

There might be many such applications, for example, mail, address book, network storage, etc. SNS provided on the Internet is also suitable for assumptions.

The main objective of this paper is, introducing a new distributed processing architecture for KVS applications. The architecture enables to migrate processing mechanism into KVS data nodes, and provides effective processing models. Especially, we describe that the architecture reduces data transmission cost, to access key and value pairs stored in KVS systems. In other words, we can avoid trade-off issues about data transmission cost of KVS applications described in the assumptions.

4. Jobcast - Data Processing Framework

Jobcast is a new parallel and distributed processing framework,

based on KVS (Key Value Store) data store. It supports not only storing key and value pairs but also processing jobs for such pairs in data nodes. Since KVS data stores achieve high scalability for data space, the Jobcast architecture attains high scalability for processing space, as well.

In this section, we introduce the Jobcast architecture, briefly.

4.1 Distribution Strategy

Jobcast provides high scalability for data space based on consistent hash algorithm. A Jobcast node, which is a data node on which a Jobcast system is running, maintains information about neighboring data nodes by p2p technology. As the result, a set of Jobcast nodes consist a ring topology, called Jobcast ring. This is similar to the Chord [2] algorithm (**Fig. 1**).

Jobcast nodes in a Jobcast ring exchange associated key and successors information between each other, so that Jobcast nodes could maintain their key space information, which denotes the authorized key range for the node. In fact, a Jobcast node does any operation for a certain key and value pair, if and only if the target key is in the authorized key range for that node.

When a Jobcast client wants to store data objects or to deliver jobs, it determines distribution keys for the target data objects. Distribution key is hash value (SHA-1, for example) of original keys, in usual cases. Once the client gets the distribution key, it delivers requests for such key to the associated Jobcast node.

Successors information is also useful to maintain master and slave information for replication and redundancy. These mechanism are not essential in this paper, and discussed in Section 7 for future researches.

4.2 Distributing Data Objects, i.e., Key and Value Pairs

Jobcast stores key and value pairs in its data storage, as a KVS system. Any of the standard KVS commands, such as GET, SET, DELETE is also supported by Jobcast. Depending on the distribution key of data object, Jobcast clients transmit such requests to the Jobcast node, based on distribution strategy describe above.

A mechanism of storing a pair of key and value is that of standard KVS systems, and it provides high scalability for distributing key space, of course.

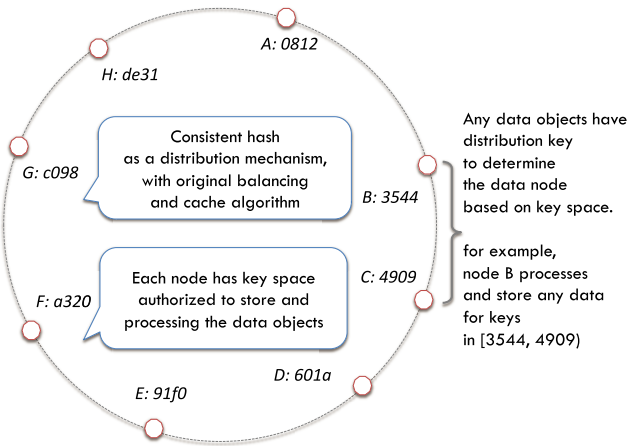

Fig. 1 Distribution strategy.

4.3 Distributing "Jobs" for Processing

As the most characterized feature of Jobcast architecture, it delivers "jobs" into Jobcast nodes for processing. Here, jobs are simple programs. In current Jobcast implementation, application developers may write their own jobs as simple Java classes, which implement specific "Job" interface.

Similar to distributing key and value pairs, a Jobcast client delivers "jobs" based on the same distribution strategy, as well. It transmits only the information about the job definition and then receiving Jobcast node processes the job and returns the result. There is no need to transmit the value itself.

4.4 Processing Model

By definition, each Jobcast node has its own key space for Jobcast operation, such as, storing key and value pairs or processing jobs. It is obviously that there is no interference between Jobcast operations on different Jobcast nodes since each Jobcast node has its own key space, which are separated from the others. That is, we can achieve linear scalability about Jobcast operations, with respect to the number of Jobcast nodes.

In addition, in order to guarantee the consistency of data updates while increasing the parallel processing efficiency, we provide two important features;

(1) Multiple jobs for different keys.
If a Jobcast node has two or more requested jobs for different keys, it executes those jobs simultaneously, in different threads.

(2) Multiple jobs for a same key.
If a Jobcast node has two or more jobs for a same key, the node executes such jobs in concurrent model, that is, it holds shared lock for read operation, and exclusive lock for write (edit) operation, for that key.

5. Benefit of Distributed Processing

As introduced in the previous section, the motivation of distributing "jobs" is achieving a new processing model for KVS applications. The processing model provides parallel and distributed processing environment for key and value pairs, and achieves several benefits for KVS applications. In this section, we describe benefits of the Jobcast architecture.

5.1 Scale-out Both Data Space and Processing Resources

As a KVS system to store key and value pairs, Jobcast provides scale-out data space. In addition, it also provides scale-out processing resources. In the Jobcast processing architecture, a Jobcast client delivers "jobs" which are application functions, and all data nodes can execute the jobs, with their own CPU and memory spaces. So, we can use many CPU power and memory resources on Jobcast nodes, instead of front-end client environment.

We show that the conceptual difference of distribution models of normal KVS and Jobcast in **Figs. 2** and **3**. As shown in Fig. 2, in normal KVS model, we distribute data objects, i.e., key and value pairs into a huge amount of data nodes, but there is no scalable mechanism of application logic. In other words, a KVS client must process application functions in it.

On the other hand, in the Jobcast architecture, a Jobcast client

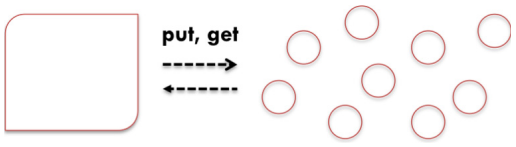

Fig. 2 Distributing key and value pairs in KVS.

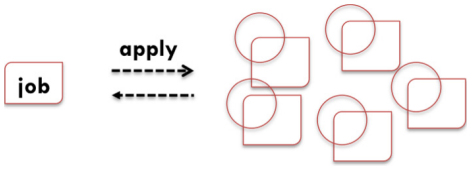

Fig. 3 Distributing both data objects and processing.

may define a job and deliver it into a huge amount of Jobcast nodes to execute application logic for key and value pairs on backend data nodes. That means, we can execute application logic with server side computing resources, such as CPU, memory and so on.

5.2 Minimize Data Transmission Cost

As described in Section 3, there are trade-off issues about data transmission cost to develop applications with standard KVS data store. Jobcast provides a mechanism to reduce data transmission overhead for such applications, by migrating processing mechanism into backend KVS data nodes.

The basic idea is to use user base data schema to store user data and implementing "jobs" to access individual item, instead. By using user base data schema, it's easy to access entire content of user data, of course, as described before. In addition, implementing "jobs" to get a small portion of the content reduces total bytes of data transmission when we need to access small item value. We do not need to transmit entire contents, any more.

Jobcast provides more flexible mechanism to reduce data transmission cost for other situations. For example, think about searching a keyword from entire contents of a given user. Using standard KVS with item base data schema causes many data transmission between application server and KVS data nodes. By using standard KVS with user based data schema, an application server needs to get entire contents from backend KVS data store at first to process search function. On the other hand, Jobcast provides a mechanism to execute search function in backend KVS data nodes where we need to transmit only "jobs" and "result" via the network.

Jobcast dramatically improves network usage for the cases where we need update content stored in backend KVS data store, in some situations, as well. By standard KVS with user base data schema, we need 3 steps, that is, retrieving entire content, update the content in application server and storing entire content, again. It causes transmitting entire content twice on the network. Instead, using Jobcast model, we only need to send "job" to update the content and receive ACK as an answer. There is no need of transmit redundant content.

One of major purposes of this paper is to show the benefits of reducing data transmission cost by the Jobcast architecture, and we evaluate such applications, in Section 6.

5.3 Data Integrity during Data Updates

In standard KVS systems, if an application server wants to update data objects stored in KVS system, it must retrieve such data objects from backend nodes first, modify or edit them, and store them into the backend nodes, again. In this model, there is an issue about data integrity. For example, assume that two client nodes, say client-A and client-B exist and they want to update the same data objects. If client-A retrieves the object for updates, when client-B retrieves, edits and stores the same object, there might be a data corruption.

In the Jobcast processing model, we can define a job to update the data object, and the associated Jobcast node will execute the job with concurrent processing model (described in Section 4). There is no data corruption and we can assure the data integrity during data update. Shared and exclusive lock mechanism in processing jobs also enables us to ensure that there is neither data change nor modification when someone is reading such data object.

5.4 Data Privacy

Jobcast data processing model improves privacy of stored data objects. Let's think about using KVS system to store account information for some network services. By user base data schema, keys are assigned as user-id and values contain account information such as, password, age, sex, address, phone number, and so on. If a user wants to log in to such service and give the password, a KVS client retrieves the entire account information from a backend data node and verify the password against such information. This may not be safe because some personal information is transmitted via network and accessible in a processing node.

In the Jobcast architecture, we can define a job to verify password. The job contains a password given from the user, and a jobcast client transmits the job toward the associated Jobcast node. The Jobcast node executes the job to verify the given password against the stored one. This is safer since any job execution would be done inside the Jobcast node.

6. Evaluation

In this section, we evaluate the proposed architecture, especially, about improvement of data transmission cost. We introduce a mailbox service as an example, to figure out behavior of not only simple data access, like PUT or GET, but also extendable operation such as SEARCH. We also introduce an address book service, to evaluate REPLACE function, which needs to update stored data.

We describe implementation overview, evaluation environment and two evaluations, in following subsections.

6.1 Implementation Overview

We implemented 3 major subsystems for the evaluation (**Fig. 4**). We describe these subsystems briefly.

(1) Jobcast nodes

Jobcast nodes are backend KVS data nodes in which we run JavaVM to execute "jobs." Note that, we implemented data storage in memory space to simplify following evaluation. This is because we'd like to avoid any of HDD overheads.

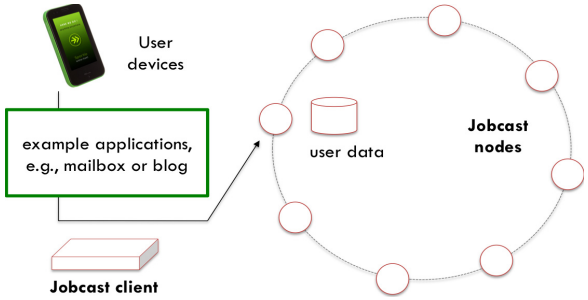

Fig. 4 Overview of Jobcast evaluation system.

Table 2 Spec. of environment for evaluation.

Type	CPU	Memory	Network	#
client	Core 2 Quad 2 GHz	4 GB	1 Gbps	1
node	Core 2 Quad 2 GHz	4 GB	1 Gbps	8

(In general, HDD overhead is larger than network).

(2) Jobcast client

Jobcast client is a frontend of the Jobcase system. It works as an application server, in 3 layers application service model. That is, it gets requests from user devices via HTTP and provides services. If it needs to access stored key valur pairs, or wants to process "jobs," the Jobcast client also communicates with Jobcast nodes via HTTP.

(3) User devices

We also implemented pseudo user devices as clients. In following evaluation, we emulate 100 simultaneous accesses from user devices at the same time.

6.2 Evaluation Environment

Table 2 shows the specifications of the Jobcast client and Jobcast nodes for the evaluation. We have 8 Jobcast nodes and 1 Jobcast client,

6.3 Evaluation & Result #1

At first, we introduce mailbox application. We assume that a large number of users have mailboxes in the service. Each user may have several numbers of messages in his mailbox, of course. As defined in Assumption 4 in Section 3, the application provides several ways to access his mailbox, for example, a user can put/get a message into/from his mailbox, or he can access entire messages. We also introduce an additional feature of searching keyword in users' mailbox, which also requires accessing entire contents of user data.

Each figure (of **Figs. 5–16**) contains the result of standard KVS with item base data schema, standard KVS with user base data schema and Jobcast with user based data schema. We emulate 100 simultaneous accesses and measure an average of 1,000 times of requests for each exam.

6.3.1 Put or Get a Message

At first, we evaluate performance to put/get a message into/from a mailbox. Note that, we assume each mailbox has 10 messages, in each exam. Results are shown in Figs. 5–8, where Figs. 5 and 6 are the result of PUT operation, and Figs. 7 and 8 are that of GET operation. Y-axis of Figs. 5 and 7 shows latency (average response time of requests, in milliseconds). Y-axis of Figs. 6 and 8 shows number of processed request in a second [rps]. X-axis in all figures means average message size as parameter.

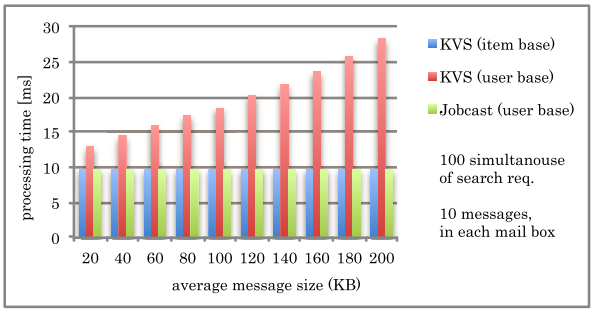

Fig. 5 PUT a message: performance (latency).

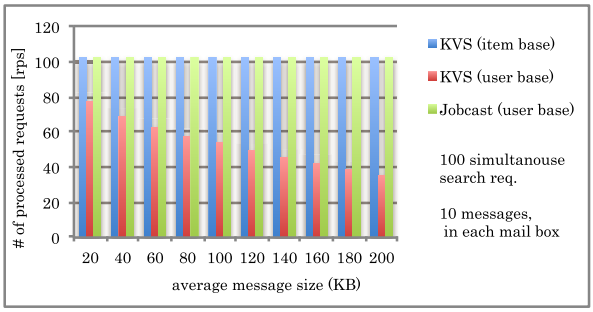

Fig. 6 PUT: performance (# of req.).

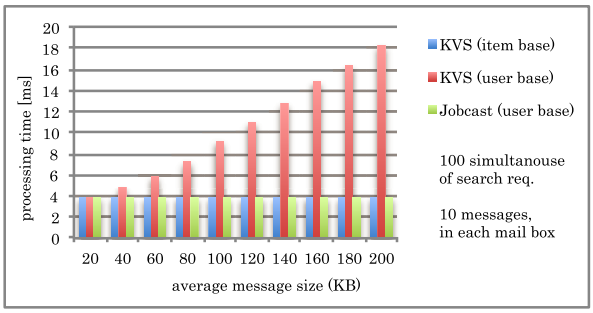

Fig. 7 GET a message: performance (latency).

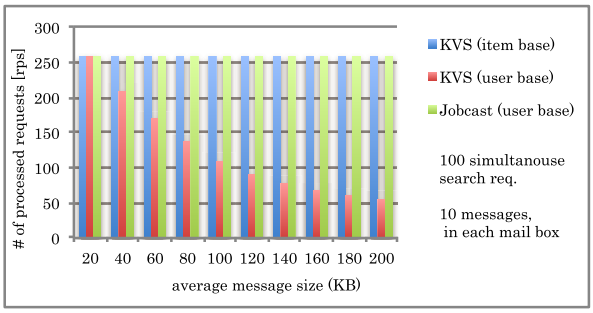

Fig. 8 GET a message: performance (# of req.).

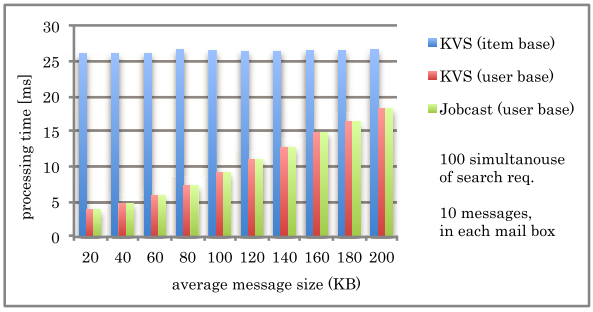

Fig. 9 GET entire contents: performance by size (latency).

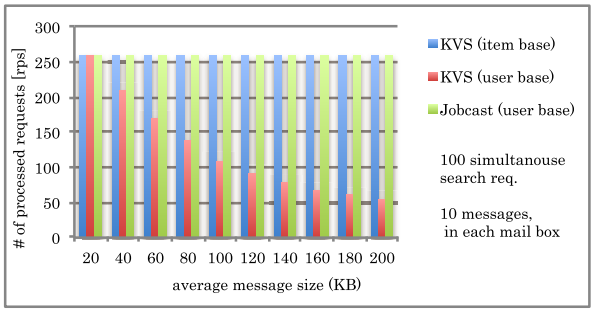

Fig. 10 GET entire contents: performance by size (# of req.).

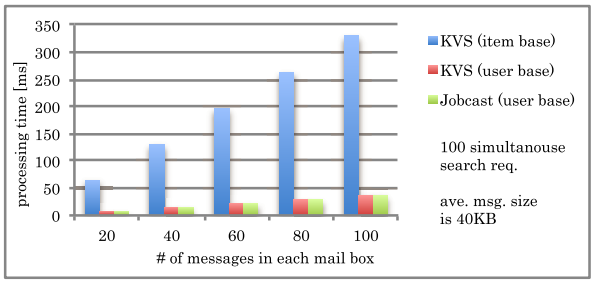

Fig. 11 GET entire contents: performance by # of msg. (latency).

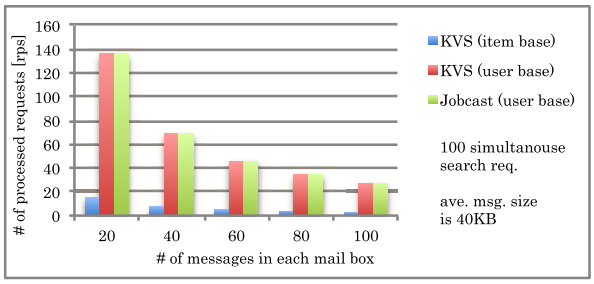

Fig. 12 GET entire contents: performance by # of msg. (# of req.).

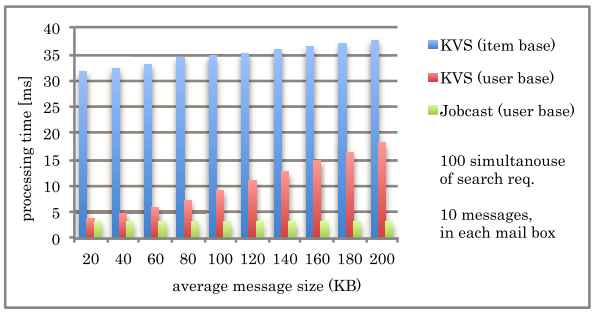

Fig. 13 SEARCH: performance by msg. size (latency).

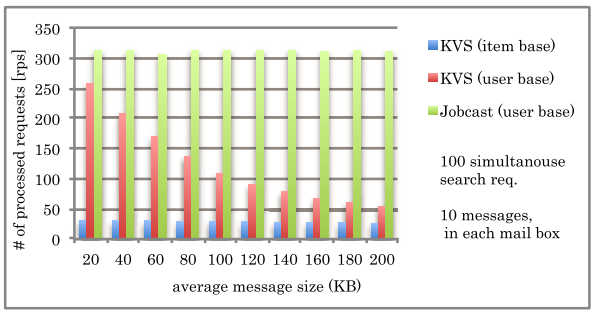

Fig. 14 SEARCH: performance by msg. size (# of req.).

As described in Section 3, item base data schema has benefit to access each item, since it only need to put/get the related key value pair, while the user base data schema causes transmitting entire contents.

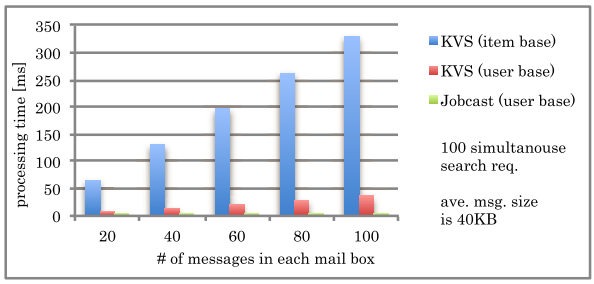

Fig. 15 SEARCH: performance by # of msg. (latency).

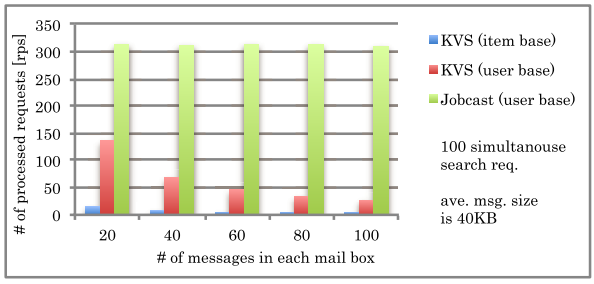

Fig. 16 SEARCH: performance by # of msg. (# of req.).

Note that, by the Jobcast processing model, we achieved almost same performance with item base data schema. That means, by processing job for put/get a message in Jobcast node instead of getting entire contents, we achieved better performance even if we use user base data schema.

6.3.2 Get Entire Content

Figures 9 and 10 are similar graphs for getting entire content of user mailbox, instead. In this case, item base data schema causes 10 times of data transmissions since there are 10 messages in each mailbox, in this exam. It is relatively larger communication cost. User base data schema is beneficial for accessing entire contents, since it can retrieve the entire content by single transmission. Jobcast also takes same advantage since it has same data schema.

In Figs. 11 and 12, we fix average message size as 40 KB, and we change the number of messages (shown X axis) in each mailbox. Note that, increasing number of messages causes serious problem of transmission cost in item base data schema. Both standard KVS with user base data schema and Jobcast model achieve maximum performance for given network bandwidth. (e.g., Receiving 4 MB, e.g., 100 messages of 40 KB, takes 32 ms via 1 Gbps).

6.3.3 Search Keyword from Entire Contents

Search is an extensive feature of mailbox application. There are many cases to search messages that contain some keyword. We provide search mechanism in server side, that is, user devices send keywords to the mailbox application and the application tries to search such keywords in stored messages. Figures 13 and 14 shows the result of searching keyword from 10 messages in a mailbox.

Standard KVS with item base and user base data schema behave similar as GET ENTIRE CONTENT exam. Instead, by Jobcast model, we achieve better performance. This is because Jobcast only needs to send “jobs” for SEARCH and to receive a list

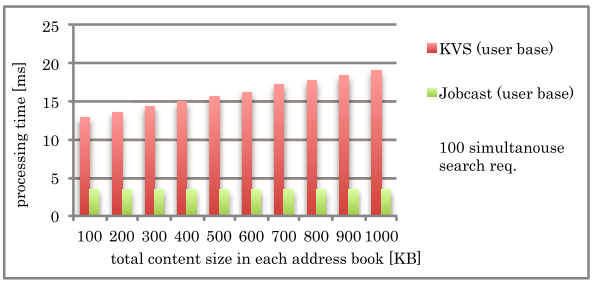

Fig. 17 REPLACE: performance by content size (latency).

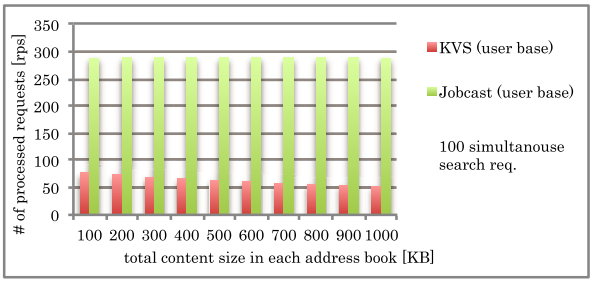

Fig. 18 REPLACE: performance by content size (# of req.).

of message identifiers as "answers." As the result, it requires less data transmission volume.

Figures 15 and 16 also shows the result for when we change the number of messages with fixed average message size of 40 KB. It also shows the Jobcast achieves better performance than both of standard KVS models.

6.4 Evaluation & Result #2

To evaluate performance of updating content, we also introduce an address book application. We assume there are many users for the application and each user has tens or hundreds of items in his address book.

6.4.1 Replace Keywords (Update Content)

As the exam, we evaluate performance of "replacing" keywords. There are several cases where we need to update items in address book. In practice, postal zip code or name of city, town, village (by migration) or a number of digits in phone number were changed several times in Japan.

Results of mailbox applications tell us that it is obviously inefficient to use item base data schema to access entire contents if there are many items in an address book.

So, we evaluated standard KVS with user base data schema and Jobcast model. In **Figs. 17** and **18**, X-axis is average total content size of each address book. Y-axis of Figs. 17 and 18 are latency and number of processed requests, in respectively. In this exam, standard KVS with user base data schema causes 2 times of data transmission for entire contents, since it needs 3 steps of retrieving entire contents, replacing keywords and storing updated contents, again. As shown in these figures, while performance of standard KVS case is limited by network bandwidth (sending 2 times of 1 MB content takes 16 ms by 1 Gbps), Jobcast model dramatically improves performance.

7. Considerations

In this section, we provide an expanded discussion to promote further understanding.

7.1 Additional Evaluation

In the previous section, we evaluated and discussed about data transmission behavior in simplified situations. We need more detailed evaluation for operational applications.

7.1.1 More Parameters

To evaluate data transmission behavior, we simplified conditions, e.g., simple cases of data sizes, number of items, and so on. We also implement data store mechanism in memory space to avoid HDD overhead. As the result, bottleneck of each exam was network and performance was limited by 1 Gbps bandwidth (NIC of Jobcast client). If we use HDD instead, HDD might be another bottleneck. In addition, CPU or memory management, network conditions may be also bottleneck, depend on evaluation environment. These kind of production level evaluation and tuning should be studied in future researches.

7.1.2 Processing Complexity and CPU Load

PUT, GET, SEARCH and REPLACE functions in the evaluation are very simple and we had no significant impact in CPU or memory in both Jobcast client and Jobcast nodes. When we define more complex "job," yet another bottleneck may appear in CPU or memory. We should also evaluate and discuss about complexity of jobs.

7.2 Implementation for Operational Use

For making the Jobcast mechanism to be operational, we need more implementation about reliability or resiliency.

7.2.1 Persistency of Data Objects

We need to store data objects into persistent storage, like HDD. We should implement and evaluate mechanisms to store key and value pairs, for example, B+tree or similar mechanism is better to read, while journal style logging mechanism provides better performance to write data objects.

7.2.2 Failover and Recovery

We also need to maintain copies of key value pairs to achieve reliability. Failover and recovery mechanisms should be designed and evaluated.

8. Conclusion

In this paper, we propose "Jobcast" architecture which is a new processing model for KVS data store. Jobcast distributes not only key and value pairs, but also "jobs" for such pairs, in parallel and distributed processing model. By p2p based consistent hash algorithm, Jobcast architecture achieves high scalability for both data space and processing space.

We also evaluated the Jobcast mechanism in particular regarding reduction of data transmission cost. We introduced two applications, such as mailbox application and address book application to compare performance of processing requests. As the result we showed that the Jobcast achieved equal or better performance rather than standard KVS with item base and user base data schemas, in any case.

Acknowledgments We wish to thank the members of RICC-WG (Regional Inter-Cloud sub-Committee) of ITRC (Internet Technology Research Committee) for their discussion of require-

ments and technologies for scalable computing. We also thank the EXAGE members in Intec, Inc., for cutting-edge implementation of the Jobcast.

References

[1] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. and Lewin, D.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web, *Proc. 29th Annual ACM Symposium on Theory of Computing*, *STOC '97*, El Paso, Texas, United States, pp.654–663, ACM Press, New York, NY (May 1997).

[2] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F. and Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications, *Proc. 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, *SIGCOMM '01*, San Diego, California, United States, pp.149–160, ACM Press, New York, NY (2001).

[3] Memcached project: memcached — A distributed memory object caching system, available from ⟨http://memcached.org/⟩.

[4] Cassandra project: The Apache Cassandra Project, available from ⟨http://cassandra.apache.org/⟩.

[5] FA Labs: Tokyo Cabinet: A modern implementation of DBM, available from ⟨http://fallabs.com/tokyocabinet/index.html⟩.

[6] FA Labs: Kyoto Cabinet: A straightforward implementation of DBM, available from ⟨http://fallabs.com/kyotocabinet/⟩.

[7] ROMA project: A Distributed Key Value Store in Ruby, available from ⟨http://code.google.com/p/roma-prj/⟩.

[8] Okuyama: Distributed Key Value Store, available from ⟨http://sourceforge.jp/projects/okuyama/⟩.

Ikuo Nakagawa received a M.S. degree from Tokyo Institute of Technology in 1993, and a Ph.D. from the University of Tokyo in 2005. He joined to Intec, Inc., in 1993 where he has been researching on network technology. He established Intec NetCore, Inc., in 2002. He works for Intec, Inc., as the Executive Chief Engineer, since 2011. He serves concurrently as an associated professor of Osaka University since 2012.

Kenichi Nagami received a M.S. degree from Tokyo Institute of Technology, Japan, in 1992. He received a Ph.D. from Tokyo Institute of Technology, Japan, in 2001. In 1992, he joined Research and Development Center, Toshiba Corporation where he has been working on communication system. From 2002, he works for Intec NetCore as CTO. He currently works for the Research and Development Department at Intec.