Dripcast – server-less Java programming framework for billions of IoT devices

Ikuo Nakagawa
Intec, Inc. & Osaka University

Masahiro Hiji
Tohoku University

Hiroshi Esaki
University of Tokyo

***Abstract*—We propose "Dripcast", a new server-less Java programming framework for billions of IoT (Internet of Things) devices. The framework provides a simple and easy way to develop device applications working with a cloud, that is, scalable computing resources on the Internet. The framework consists of two key technologies; (1) transparent remote procedure call (2) mechanism to read, write and process Java object with scale-out style distributed datastore. A great benefit of the framework is no need of writing server-side program nor database code. A very simple client-side program is enough to work with the framework, to read, write or process Java objects on a cloud. The mechanism is infinitely scalable since it works with scale-out technologies. In this paper, we describe the concept and the brief architecture of the Dripcast. We also introduce very simple example of a use case of the Dripcast.**

I. Introduction

Today, a huge amount of devices and sensors are connecting to the Internet under the concept of IoT (Internet of Things). Some expectations [1] said the number of IoT devices would be tens (or hundreds) of billions by 2020. There are many working sensors, in home electric devices, healthcare devices, etc. A smartphone is a set of various sensors and is also a useful IoT device. It has a GPS receiver, an accelerometer, a thermometer, etc. There will be unlimited number of applications working with such IoT devices.

Most of such device applications work with cloud services, that is highly scalable computer resources on the Net. One of major computing models for such device applications is 'cloud-ful'. In the model, applications run on user-side devices (smartphone, tablet, any small devices or gateways for sensors) while they store and process data on a cloud.

On the other hand, programming such applications is still difficult. In general, we have to develop and deploy server side applications for such device applications. which work with backend databases. Many such applications are still designed in 3 layer model, where we need not only develop client side application but also server side programs and database code.

In this paper, we propose "Dripcast", a new Java programming framework which is suitable device applications. The framework provides a simple and easy framework for small devices to *operate* data on a cloud environment, where, *operate* means any set of reading, writing and processing data.

The Dripcast consists of two key technologies ;

1) transparent Java remote procedure call
2) a mechanism to store, read and process Java object with scale-out style distributed datastore.

As the result, a very simple client-side programming is enough for small devices to work with highly scalable cloud platform. There is no need to write server-side program nor database code for such devices to *operate* Java objects on a cloud.

Note that, we focus into Java programming language. We have several application environment for IoT devices, today. For example, Android [2] is powerful application platform for smartphones, tablets and various mobile devices. OSGi [3] is a gateway for small devices in houses or buildings.

A goal of this paper is providing simple and easy developing model of "server-less". In the server-less programming model, developers need not take care about databases, server side applications nor communications between application components. We need not to study SQL, server programming languages such as php, perl, Ruby, etc. We need not care of REST or XML definitions, as well.

We also propose a highly scalable mechanism to *operate* process Java objects on a cloud environment. Since the Dripcast is based on a scale-out style computing model, the framework provides practically unlimited scalability.

In this paper, we refer existing technologies and researches in Section II. We summarize assumptions and objectives of this paper in Section III, and describe basic architecture in Section IV. We introduce an example Dripcast application in Section V. Finally, we discuss for further understanding in Section VI and conclude in Section VII.

II. Related Works

The Dripcast framework consists of two key technologies, (1) transparent remote procedure call, and (2) mechanism to *operate* Java object on scale-out style distributed datastore. We survey related works in such points of view.

A. RPC & ORB

Several RPC (remote procedure call) and ORB (object request broker) mechanisms have been discussed. Java RMI (Java Remote Method Invocation) [1] is an application programming interface for remote procedure calls. JRMP (Java Remote Method Protocol) is categorized as ORB technology and defined the protocol for remote procedure call from

[1] www.gartner.com, www.idc.com, www.cisco.com, etc.

[2] www.android.com

[3] www.osgi.org

a JavaVM to another JavaVM. CORBA (Common Object Request Broker Architecture) [2] is also the mechanism for remote procedure calls which works in non JavaVM context, and RMI-IIOP (RMI over IIOP) is Java RMI interface in CORBA systems.

Interface definitions and method invocation mechanisms are based on OOM (Object oriented modeling) in both RMI and ORB. On the other hand, such technologies assumes client–server programming model (and 3 layers of client–server–database programming model, in general). Programmers need to write both client and server side programs.

Dripcast assumes a new programming model, "server-less", in which programmers do not need to care of server side programming.

B. Distributed Datastore

The Dripcast framework provides programming framework which is working with scale-out style distributed datastore. From the view of point of scale-out datastore, Several services and technologies are available.

Some service providers have their own scalable datastore. For example, Google App Engine (GAE) [3] is a programming environment for the Google cloud. They published GFS (Google File System) [4], BigTable [5] and MapReduce [6] as Google technologies. Windows Azure [7] provided by Microsoft, has scalable storage service called 'Azure Storage', SQL database called 'Azure SQL database', analyzing engine using Hadoop called 'Azure HDinsight' and so on.

Although service providers have their own proprietary implementation, there are various open source implementations. Hadoop [8] is a famous open source project for scalable cloud computing platform. It provides scalable storage space (HDFS), reliable database (hbase), parallel and distributed processing model (MapReduce) and other many useful features. To handle a Hadoop system effectively, developers have to develop a complex and efficient programs in the Hadoop manner.

Several distributed KVS (Key Value Store) and object store mechanisms also exist. Cassandra [10], CouchBase [11], Tokyo/Kyoto Cabinet [12] [13], Roma [14], Basho [15] and many open sources exists for scale-out object management. All of these datastore softwares provide a mechanism to read and write data from/to scale-out computer cluster. Providing a programming framework is out-of-scope for such technologies.

Our research purpose in this paper is providing a framework in front of such scale-out distribute datastore. Nakagawa proposed Jobcast [16] which is an intuitive extension of KVS mechanism in which we achieve parallel and distributed processing mechanism. Jobcast works in very scalable environment and there is no SPOF (Single Point Of Failure) by nature. The Jobcast might be a part (as backend) of the Dripcast framework. We will denote the relation of the Dripcast and Jobcast in Section IV.

III. Assumptions and objectives

At first, we summarize assumptions and objectives of the research.

A. Assumptions

We discuss IoT (Internet of Things) applications working on small devices. Especially, we focus into developing device applications working with cloud platform on the Net. We assume some conditions for such device applications :

- Applications work on small device, such as smartphones, tablets, navigation systems, home gateways, etc. Such devices have small and limited computer resources.
- Applications operate (read, write and process) data on the cloud, while they provide graphical or non-graphical UI (user interface) on devices.
- Applications might work on tens (or hundreds) of billions devices in a same time.

Note that, we focus into developing device applications on Java based platform such as Android or OSGi. The Dripcast framework is designed for Java programming environment.

B. Objectives

The objectives of this research is providing a new framework for device applications described in the previous subsection. We have two goals to achieve, as follows.

1) Server-less programming mechanism: The framework provides a very simple and easy mechanism to operate (read, write and process) Java objects on a cloud. It enables device applications to upload, refer or share data object with a cloud.

Any object on the cloud has its world unique identifier (UUID, for example). The framework allows developers not only to read, write data object via the identifier but also to invoke Java method transparently via standard Java interface.

Although a transparent Java method call is similar to RMI (Remote Method Invocation) or related technologies, we propose more effective mechanism of 'server-less', so that developers need not to write any server programming nor database code.

2) Unlimited scalability for IoT applications: Scalability is strict requirement for IoT application platforms. There will be tens (or hundreds) of millions devices, and the number of devices may grow infinitely. Thus, cloud platform must be designed in scale-out style.

The Dripcast framework works with scale-out style distributed datastore to achieve unlimited scalability. The backend datastore should be designed by 'shared nothing' technology. It is really suitable for most IoT applications. It can handle a huge amount of simultaneous simple access, while it is not good for relational management or transaction processes in legacy applications.

IV. Architecture

The Dripcast is a framework for storing and processing Java objects. Any object has the world unique *ID* represented as `UUID`. The Dripcast framework always takes *ID* as an argument to identify the object on the cloud.

The Dripcast framework consists of four components that is *Client*, *Relay*, *Engine* and *Store*.

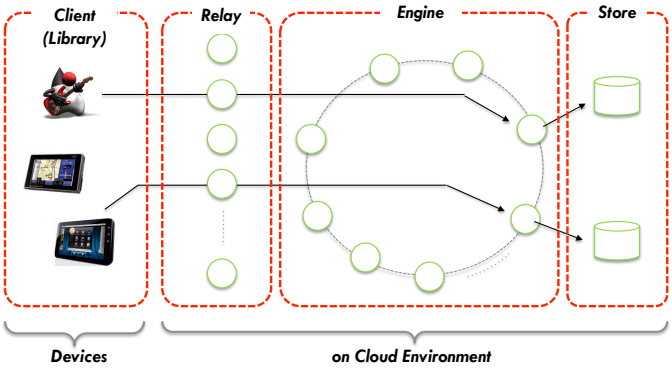

Fig. 1. Architecture

A. Client

Client is a small Java library which works on user devices such as smartphones, tablet, home gateways and so on. There are two major roles : (1) managing transparent Java object in client devices, and (2) sending remote procedure call requests to the *Relay*.

To realize transparent Java object, we use Proxy mechanism. For a given `id` and an interface class `YourInterface`, we create a Proxy object (defined in standard JDK) which supports the `YourInterface` interface. The Dripcast framework has `attach` method, to realize a transparent Java object in a simple way, as :

```
Dripcast d = new Dripcast();
YourInterface v
  = d.attach(id, YourInterface.class);
```

On a method call for the Proxy object, we would have *method-name*, *argument-classes* and *arguments-objects* by Proxy mechanism. The client library creates a new `Job` instance which has *ID*, *method-name*, *arguments-classes* and *arguments-objects* as its instance variables. The library sends the `Job` instance to *Relay* as a request. After getting the result of the request, the method call returns it as the result.

There is no need to care about RPC nor communication flow in each method call since the Proxy object provides transparent method call mechanism, Developers may call interface methods in normal way.

The *Client* also supports simple mechanism to send CREATE or REMOVE request to the *Relay*, so that the client library supports `create` method for creating a new Java object on the cloud, as follows.

```
Dripcast d = new Dripcast();
d.create(id, YourClass.class);
```

B. Relay

Relay is a set of relay servers. A relay server is a distribution gateway, who receives requests (`Job` instances) from clients and delivers such requests to engine servers described in the next subsection.

A relay server knows association of *ID* and engine servers. The association is managed by Distributed Hash Table (DHT) [17][18]. There is only one engine server for an *ID* in a same time so that distribution gateway could select the unique engine server for a request. Relay servers also deliver very simple operations such as create and remove data in the same manner.

Relay servers are stateless so that the Relay mechanism would be highly scalable.

We denote that the Jobcast is a parallel and distributed processing mechanism which is suitable to implement Relay, Engine and Store mechanism in the Dripcast framework. Relay servers correspond to clients in Jobcast architecture.

C. Engine

Engine is a set of engine servers. Each engine server has its own key space assigned by DHT, so that it would read, write and process Java object in consistent environment for authorized key space. Each engine server runs JavaVM. In other words, Engine is the distribute JavaVM for parallel and distributed processing environment, managed by DHT.

The most important role of an engine server is executing Java method for remote procedure call requests encapsulated in `Job` instances. When an engine server receives a `Job` instance which contains *ID*, *method-name*, *argument-classes* and *arguments*. The server loads the Java object `x` with *ID* as the key from the *Store*, and tries to invoke the method of `x` specified by the *method-name* and *argument-classes* with given *arguments*. If there is any change in `x`, the engine server stores it back into the *Store*. At last, the engine server returns the result back to the relay server who sent the request.

Engine servers correspond to a part of backend servers in the Jobcast architecture. The processing framework on Jobcast nodes is suitable for engine servers to execute jobs.

D. Store

The Dripcast assumes there are highly scalable datastore in backend. Any scale-out NoSQL described in Section II might be applicable. Store should provide mechanisms for replication management and automatic failover for resiliency.

The Dripcast may call following method with *ID* as a key.

1) GET – get a serialized Java object.
2) PUT – put (update) a serialized Java object.
3) REMOVE – remove existing data.

Note that, Store might be separated from engine servers in the Dripcast framework, while datastore is implemented as a part of backend servers in the Jobcast architecture.

V. Example

In this section, we introduce a simple example of the Dripcast application. Let's think of a meeting assistance application. There are some (2 to 50 for example) members who will join the meeting. For simplicity, we assume all members have Android smartphones. Each smartphone collects GPS location information and uploads the location into the cloud so that all members can show members' location map using Google Map or similar geographical map service.

A. How to use

We describe an example of Dripcast use case, briefly.

1) Preparation: At first, we assign **uid** which is a unique ID (identifier). We also create an actual Java object on the cloud, associated with **uid**. In this example, we use **TreeMap** object which is defined in standard JDK.

```
Dripcast d = new Dripcast();
d.create(uid, TreeMap.class);
```

Here, **d** is a Dripcast instance, which enables users to use the Dripcast framework. **d.create** creates a new Java object on the cloud. In this example, it creates a new **TreeMap** object associated with **uid**. Note that, it is easy to share the **uid** among all members, by sending service URL (containing **uid**) or via e-mail, for example.

Now, all members can access the Java object by **uid**, by creating a Dripcast enabled object.

```
NavigableMap map = d.attach(uid, NavigableMap.class);
```

d.attach generates a virtual object in a local device. It acts as Proxy object and a user can call remove method invocation transparently (like, RMI). In this example, each user has the Dripcast enabled object **map**, which supports **NavigableMap** interface (defined in standard JDK, as well).

2) Upload GPS information: To upload his/her GPS information into the cloud, just put a pair of phone number **pn** and location information **x,y**, into the map.

```
map.put(pn, x + "," + y);
```

map.put method call causes transparent remote procedure call. It communicates with the cloud to invoke **put** method on the cloud.

3) Show locations on a map: It is easy to show locations on the map, as well.

```
Entry e = map.firstEntry();
while (e != null) {
  String pn = (String)e.getKey();
  String[] pos = ((String)e.getValue()).split(",");
  // show location of with pn at (pos[0], pos[1]).
  e = map.higherEntry(pn);
}
```

firstEntry and **higherEntry** (also defined as methods of **NavigableMap** in standard JDK) method calls causes transparent remote procedure calls. It communicates with the cloud to invoke **firstEntry** and **higherEntry** methods on the cloud.

B. Behaviors & Communications

We note more details about behaviros and communication flow related to the Dripcast framework. Fig. 2 shows brief communication flow in the given example.

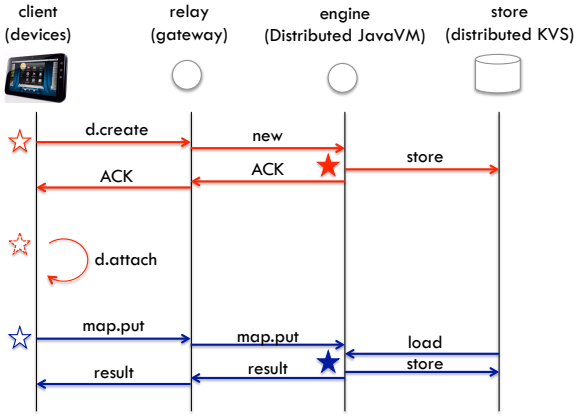

Fig. 2. Meeting assist service

In the example, there are two explicit calls related to the Dripcast, that is, **d.create** and **d.attach**. By calling **d.create** (first red empty star, in Fig. 2), the Dripcast framework communicates with backend servers; the client sends a request to a relay. the relay selects the authorized engine server by **uid**. the engine creates a new Java object (red filled start) in the server, and store the object into backend store.

On the other hand, calling **d.attach** (second red empty and dotted start) is just for a declaration. The Dripcast create a Proxy object in the local-device, which is associated with **uid** and Java interface (**NavigableMap**, in this example).

After these preparation steps, all method calls for the Dripcast enabled object, **map**, cause Proxy method calls. For example, **map.put** (blue empty star) causes Proxy method invocation as; the client send a request to a relay, the relay selects the authorized engine server by **uid**, the engine loads the Java object for **uid** if required, and invokes **put** method (blue filled star) for the object.

All method calls for **map** work as similar.

Note that, after preparation steps, in which we call only **d.create** and **d.attach** methods, the client can access the Java object via simple Java interface. There is no need to write server-side program nor database code.

C. Benefit of the Dripcast

There are several benefits of using the Dripcast. We describe major two key benefit, in this section.

1) Server-less: All we need to implement the application are three steps described in Section V-A1, V-A2 and V-A3. All what developers need to do is writing only the client (Android) side logic. It is very simple development model and there is no need to ;

- define database schema

- write SQL codes
- implement server (cloud) side system.
- define REST over HTTP
- write codes to communicate with server (cloud)

The Dripcast framework takes care all of these steps instead of developers. Developers need not to study SQL/RDB, REST/HTTP nor server side programming such as php, ruby, perl or others. Only what developers have to study is Java programming on Android devices.

2) Intuitive understanding: Note that, there is no special code depends on the Dripcast. All what we do is, just call JDK standard method, as well.

Figure 3 shows the concept of the meeting assistance application for the intuitive understanding.

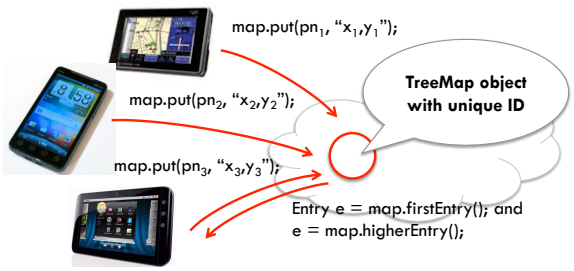

Fig. 3. Meeting assist service

There is the unique **`TreeMap`** object on the cloud and all members (with Android devices) share the object. Only what a member needs to do to upload his/her location is calling **`map.put`** method. A member also calls **`map.firstEntry`** and **`map.higherEntry`** to list the location information for all members.

3) Reuse of existing libraries: If there is an application library written in really OOM style, it is possible to reuse such libraries. The Dripcast requires a little change (or no change) to *operate* Java objects on a cloud.

In the example, described in this section, there is no special code for the Dripcast in V-A2 and V-A3.

D. Tips

In the example, we use **`TreeMap`** instead of **`HashMap`**. This is because we would like to call **`firstEntry`** and **`higherEntry`** methods of the stored object.

Although **`HashMap`** has **`keySet`** or **`entrySet`** methods to iterate all entries stored in the object, the result of **`keySet`** or **`entrySet`** may not be serializable in some JDK implementation. It causes **`NotSerializableException`** during transmitting the data between client and cloud. To handle iterative (**`Iterator`**) objects in the Dripcast framework, we have to transform such objects to be serializable in Engine logic. The mechanism will be a part of future researches.

VI. Challenges

There are some challenges in develop and deploy the Dripcast framework. In this section, we summarize the challenges in view points of ;

1) Object oriented modeling on a cloud
2) Server-less programming model
3) Cloud as a large JavaVM

We describe and discuss these challenges for further researches.

A. Object oriented modeling with a cloud

The Dripcast framework is based on Object Oriented Modeling. Any transparent Java object is tied with interface declaration and handles remote procedure calls inside method calls. The key technology of the Dripcast framework is providing fully compatible interface as local-device programming. If developers write their program based on interface definitions rather than accessing instance or class variables, source code is fully compatible in both cases for local-device programming or for cloud-ful programming.

B. Proposing 'server-less programming model'

A great benefit of the Dripcast framework is changing developing model. In the historical developing model, called *3 Layers Model*, we have to take care of at least three components, client-side application, server-side application and database. As shown in Fig 4, developers need to write programming code not only for client application logic but also for server side application or communication logic between client and server.

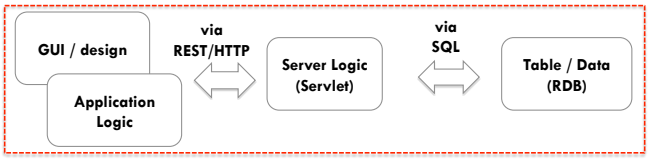

Fig. 4. 3 Layers Model

In the Dripcast model, developers only need to take care of application logic. the Dripcast framework automatically converts method calls to the associated remote procedure calls on a cloud. The framework handles data persistency of associated Java objects in the cloud environment as well.

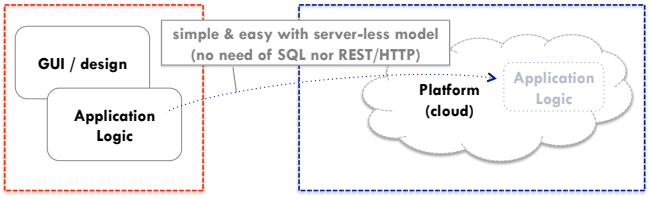

Fig. 5. Dripcast Model

The Dripcast framework has a great benefit of scalability since the framework supports the highly scalable cloud environment as the backend system. The framework may have billions of Java object on a huge scale cloud environment and deliver millions of simultaneous remote procedure call requests to thousands or tends of thousands of computers in parallel.

In historical 3 Layers Models, developers are responsible for scalability and reliability of the system. Developers must design and implement such scalable systems in mind.

In the Dripcast framework, scalability and reliability is automatically managed by the framework itself.

C. Cloud as a large JavaVM

As described in Section IV, the *Engine* works as distributed JavaVM with DHT technology. We think that the architecture enables to achieve a new Java environment in highly scalable (scale-out style) cloud platform.

In the Dripcast framework, any Java object has its own *ID* and stored into one of engine servers distributed by DHT. Key based shared nothing mechanism is not effective for relational operations and transactions for legacy applications, but is very suitable for IoT applications.

From a view point of client-programming, the Dripcast framework seems a huge and scale-out style JavaVM. In the framework, developers do not care of object location (since the location is managed by DHT), remote procedure call (it's transparent by Proxy mechanism), any network communication nor anything about JavaVM distribution at all.

There are two more important mechanism we have to design and implement to achieve more practical distributed Java environment, that is ;

1) Reference model among Distributed JavaVM
2) Global garbage collection

Global reference model can be achieved by handling references in Java objects during serialization (when we store or transmit them). We can hook any serialize operation by defining **`writeReplace`** method call to check reference information. We also define **`readResolve`** to generating transparent Java object from reference information. More details will be available in our future researches.

Global garbage collection is also important to release computer resources for unused objects from distributed JavaVM environment. We have to traverse all references by global reference model describe in above and check if each Java object has reference from any other object or not.

We already started to design and implement these two mechanisms. Such mechanisms will be proposed in a future researches.

VII. Conclusion

In this paper, we propose the Dripcast, a new server-less Java programming framework. The Dripcast provides a simple and easy way to write device applications which read, write and process Java object in a cloud. The framework consists of two key technologies, (1) transparent remote procedure call, and (2) distributed JavaVM mechanism working with scale-out distributed datastore. We also describe a simple example of GPS application in which the Dripcast framework enables server-less device programming. We will report the experimental result of our testbed, in a near future.

Acknowledgments

We thank to Associate Prof. Kondo in Hiroshima University and all members of Transparent Cloud Computing Consortium for useful discussions about device application models and its implementations. We also thank to WIDE Project, University of Tokyo, HOTnet, Smart Technologies, A.T.Works, ASTEM and Ehime CATV for operating the Dripcast testbed.

References

[1] "RMI Tutorial", http://docs.oracle.com/javase/tutorial/rmi/index.html
[2] Gerald Brose, Andreas Vogel, Keith Duddy: "Java Programming with CORBA", John Wiley & Sons, ISBN 0-471-37681-7
[3] Google: "Google App Engine", https://developers.google.com/appengine/
[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: "The Google File System", SOSP ' 03, October 19—22, 2003.
[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber: "BigTable: A Distributed Storage System for Structured Data", OSDI'06, November, 2006.
[6] Jeffrey Dean and Sanjay Ghemawat : MapReduce: Simplied Data Processing on Large Clusters OSDI ' 04, December, 2004.
[7] Microsoft: "Windows Azure", http://www.windowsazure.com/
[8] Hadoop Project: "Hadoop", http://hadoop.apache.org/
[9] Memcached project: "memcached - a distributed memory object caching system", http://memcached.org/
[10] Cassandra project: "The Apache Cassandra Project", http://cassandra.apache.org/
[11] CouchBase: "Document-Oriented NoSQL Database", http://www.couthbase.com/
[12] FA Labs: "Tokyo Cabinet : a modern implementation of DBM", http://fallabs.com/tokyocabinet/index.html
[13] FA Labs: "Kyoto Cabinet : a straightforward implementation of DBM", http://fallabs.com/kyotocabinet/
[14] ROMA project: "A Distributed Key Value Store in Ruby", http://code.google.com/p/roma-prj/
[15] Basho: "makers of the Riak distributed database", http://basho.com
[16] Ikuo Nakagawa and Ken Nagami: "Jobcast – Parallel and distributed processing framework", IPSJ Journal, Vol 21, No 3, Jul 2013
[17] 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", In Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing (El Paso, Texas, United States, May 04 - 06, 1997). STOC '97. ACM Press, New York, NY, 654-663, 1997.
[18] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H.: "Chord: A scalable peer-to-peer lookup service for internet applications", In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (San Diego, California, United States). SIGCOMM '01. ACM Press, New York, NY, 149-160, 2001.