Journal of Information Processing Vol.23 No.4 458–464 (July 2015)

[DOI: 10.2197/ipsjjip.23.458]

Regular Paper

Dripcast – Architecture and Implementation of Server-less Java Programming Framework for Billions of IoT Devices

Ikuo Nakagawa[1,2,a)] Masahiro Hiji[3] Hiroshi Esaki[4]

Received: September 16, 2014, Accepted: December 3, 2014

Abstract: We propose "Dripcast," a new server-less Java programming framework for billions of IoT (Internet of Things) devices. The framework makes it easy 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 objects with scale-out style distributed datastore. A great benefit of the framework is that there is no need to write a server-side program nor a 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 architecture of the Dripcast framework. We also implement the framework and evaluate from two points of views, 1) from the view point of scalability about cloud resources, 2) from the view point of method call encapsulation overhead in client IoT devices.

Keywords: Internet of Things, Java programming framework, distributed computing

1. 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 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 an unlimited number of applications working with such IoT devices.

Most of these device applications work with cloud services, that is highly scalable computer resources on the Internet. One of the 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 as a 3 layer model, where we need not only to develop the client side application but also the server side programs and database code.

In this paper, we propose "Dripcast," a new Java programming framework which is suitable for 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 objects with scale-out style distributed datastore.

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

A goal of this paper is to provide a 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 about REST or XML definitions, ether. The Dripcast framework provides a highly scalable mechanism to *operate* process Java objects on a cloud environment, with scale-out style computing technology, as well.

In this paper, we also implement a prototype of the Dripcast framework and evaluate the framework from two points of view, that is, from a view point of scalability about computer resources on a cloud, and a view point of automatic method call encapsulation overhead in client IoT devices.

We refer to existing technologies and researches in Section 2. We summarize assumptions and objectives of this paper in Section 3, and describe basic architecture in Section 4. We introduce an example of Dripcast application in Section 5. In Section 6, we implemente a prototype of the Dripcast framework and evaluate the scalability of computing resources on a cloud, and encapsulation overhead in client IoT devices. Finally, we discuss for further understanding in Section 7 and conclude in Section 8.

[1] Intec, Inc., Takaoka, Toyama 933–8777, Japan
[2] Osaka University, Suita, Osaka 565–0871, Japan
[3] Tohoku University, Sendai, Miyagi 980–8579, Japan
[4] The University of Tokyo, Bunkyo, Tokyo 113–8654, Japan
[a)] ikuo@inetcore.com

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

2. Related Works

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

2.1 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 defines the protocol for remote procedure call from 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 assume a 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 be concerned with server side programming.

2.2 Distributed Datastore

The Dripcast framework provides a 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 a 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 a 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 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 exist for scale-out object management. All of these datastore softwares provide a mechanism to read and write data from/to a scale-out computer cluster. Providing a programming framework is out-of-scope for such technologies.

Our research purpose in this paper is to provide a framework in front of such a scale-out distribute datastore. Nakagawa proposed Jobcast [16] which is an intuitive extension of KVS mechanism in which we achieve a parallel and distributed processing mechanism. Jobcast works in a 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 4.

3. Assumptions and Objectives

At first, we summarize the assumptions and objectives of proposing the Dripcast framework.

3.1 Assumptions

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

- Applications work on small devices, 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 hundreds of billions devices at the same time.

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

3.2 Objectives

The requirements for developing the device applications described in the previous subsection are the scalability and transparency of cloud-side objects from IoT devices. The objective of this research is to provide the new framework which makes device application programming easy.

We have three goals to achieve, as follows.

3.2.1 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 objects on 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 the 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 a more effective mechanism of 'server-less,' so that developers need not write any server programming nor database code.

3.2.2 Unlimited Scalability for IoT Applications

Scalability is a strict requirement for IoT application platforms. There will be hundreds of millions devices, and the number of devices may grow infinitely. Thus, the 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.

3.2.3 Reasonable Overhead for IoT Applications

The framework provides a seamless and transparent mechanism for accessing cloud objects, by automatic method call encapsulation in client-side IoT devices. On the other hand, the overhead of such encapsulation should be reasonable for the IoT applications described above. We implement a prototype of the framework and evaluate the overhead in a practical cases for IoT applications.

4. 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 which are *Client*, *Relay*, *Engine* and *Store* (**Fig. 1**).

4.1 Client

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

To realize a transparent Java object, we use the 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 *argument-objects* by Proxy mechanism. The client library creates a new `Job` instance which has *ID*, *method-name*, *argument-classes* and *argument-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.

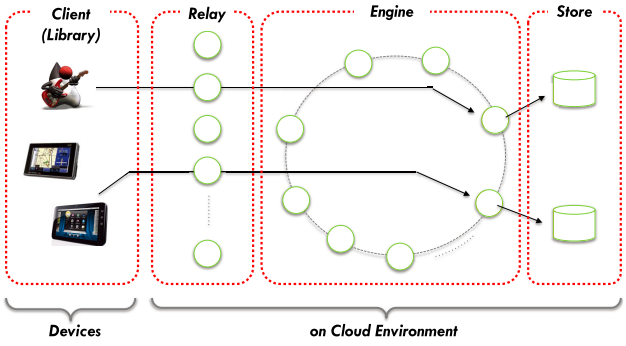

Fig. 1 Architecture.

There is no need to be concerned with RPC nor communication flow in each method call since the Proxy object provides a transparent method call mechanism. Developers may call interface methods in the normal way.

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

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

4.2 Relay

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

A relay server knows the 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* at the same time so that the distribution gateway can 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 for implementing Relay, Engine and Store mechanism in the Dripcast framework. Relay servers correspond to clients in Jobcast architecture.

4.3 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 objects in a consistent environment for authorized key space. Each engine server runs JavaVM. In other words, the engine is the distribute JavaVM for parallel and distributed processing environment, managed by DHT.

The most important role of an engine server is executing the 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 *argument-objects*. 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 *argument-objects*. If there is any change in `x`, the engine server stores it back into the *Store*. Finally, the engine server returns the result back to the relay server that 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.

4.4 Store

The Dripcast assumes there are highly scalable datastore in backend. Any scale-out NoSQL described in Section 2 might be applicable. Store should provide mechanisms for replication management and automatic failover for resiliency.

The Dripcast may call the 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.

5. 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.

5.1 How to Use

We describe an example of Dripcast use case, briefly.

5.1.1 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 a Proxy object and a user can call remote 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).

5.1.2 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.

5.1.3 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.

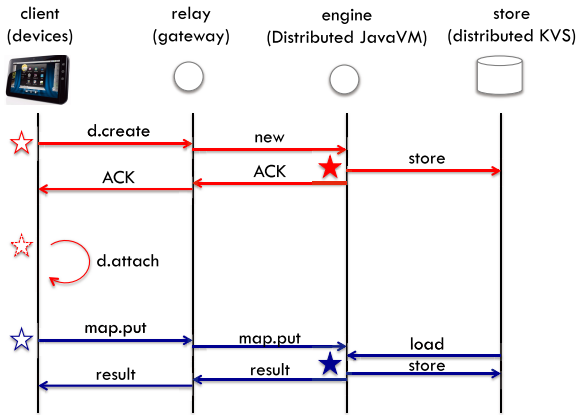

Fig. 2 Meeting assist service.

5.2 Behaviors & Communications

We note more details about behaviors and communication flow related to the Dripcast framework. **Figure 2** shows a brief communication flow in the given example.

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 star) in the server, and stores 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 preparatory 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 sends 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 in similar ways.

Note that, after preparatory 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 a server-side program nor database code.

5.3 Benefit of the Dripcast in the Example

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

5.3.1 Server-less

All we need to implement the application are three steps described in Sections 5.1.1, 5.1.2 and 5.1.3. All the developers need to do is simply to write the client (Android) side logic. It is a very simple development model and there is no need to;

- define database schema
- write SQL codes

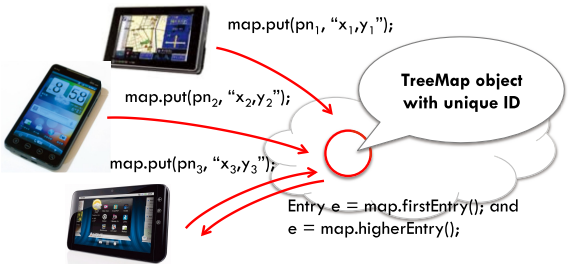

Fig. 3 Meeting assist service.

- 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 study SQL/RDB, REST/HTTP nor server side programming such as php, ruby, perl or others. All the developers have to study is Java programming on Android devices.

5.3.2 Intuitive Understanding

Note that, there is no special code that depends on the Dripcast. All we do is, just call JDK standard method, as well.

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

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

5.3.3 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.

6. Evaluation

In this research, we implemented a prototype of the Dripcast framework and evaluated the framework from two point of views. From the view point of cloud resources, we evaluate the scalability of the framework. Since we assume millions or billions of IoT devices will connect to the Internet, the scalability of the framework is very important. From the view point of the client, that is IoT devices, we also evaluate the encapsulation overhead of using the framework. The framework automatically encapsulates method calls on IoT devices and invokes such method on the cloud, we need additional encapsulation cost (serialization and deserialization, in Java). We measured and discussed such overhead for practical cases of IoT applications.

6.1 Scalability of Cloud Resources

As we described in Section 4, the Dripcast framework consists of four components, which are, *Client*, *Relay*, *Engine* and *Store*. The architecture is fully distributed and designed in the scale-out computing model. The design has no single point of failure nor bottleneck and it's easy to expand and enlarge the computer resources.

We examined the scale-out feature of the Dripcast framework. We evaluated the performance (number of requests, processed on the cloud in a second), while we increase the number servers as 1, 2, 3,... and so on.

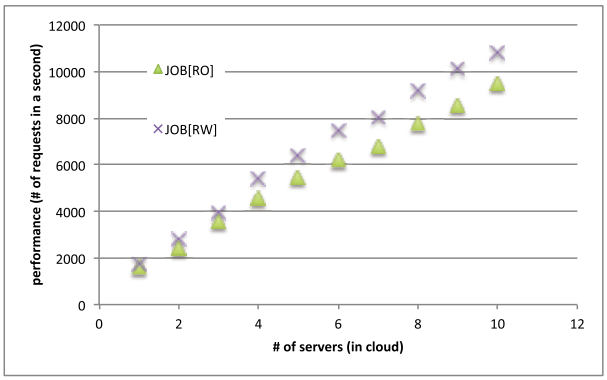

Fig. 4 Scale-out feature of cloud resources.

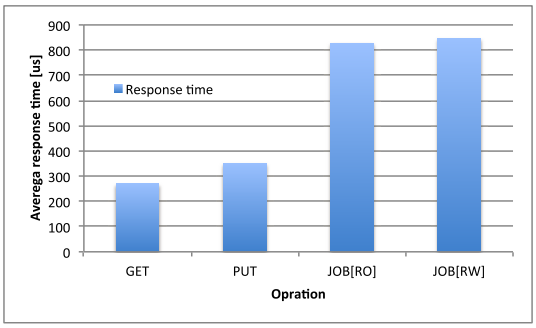

Fig. 5 Encapsulation overhead in client access.

In **Fig. 4**, the Y-axis indicates the performance, the total number of requests the cloud processes in a second. We examined a large number of operations simultaneously more than 60 seconds, and calculate the average performance.

The X-axis is the number of servers (both relay and engine servers). For example, "5" in the x-axis means that we have 5 Relay servers and 5 Engine servers in a cloud for the test. We use small instances of virtual machines for each server with 2 core and 2 GB memory, on a public cloud service.

There are two graphs in the figure. JOB[RO] and JOB[RW] means, "Read only job (only refers existing object and calculate without modification)" and "Read write job (need to store into backend, after execution)," respectively. Both JOB[RO] and JOB[RW], method calls are automatically encapsulated by the Dripcast framework.

The figure shows that we can expand the total performance by simply adding servers on the cloud. This is a simple and important feature of scale-out style computing.

6.2 Encapsulation Overhead in IoT Devices

In this section, we discuss and evaluate the encapsulation overhead of using the Dripcast framework.

The Dripcast framework requires additional overhead for encapsulating method calls for serialization and deserialization. **Figure 5** shows the average response time of standard GET and PUT access for standard KVS, and JOB[RO] and JOB[RW] access via the Dripcast framework. The average response times are 250-810 microseconds, depending on the type of operation, and it is obvious that there is encapsulation overhead to use the Dripcast

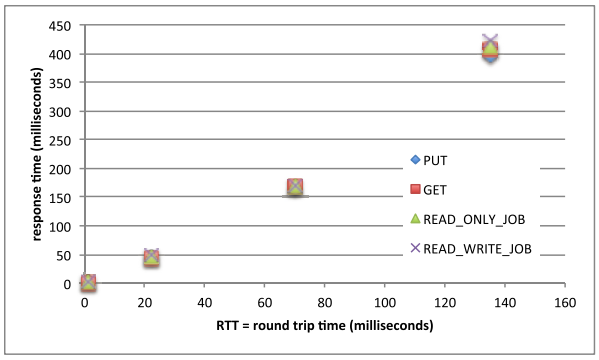

Fig. 6 Response time from various locations.

framework. Note that we measured the response time for clients inside the same DC (data center), that is clients were located in the same location with cloud resources.

On the other hand, in IoT applications, there are a large number of IoT devices and various network conditions exist. Especially, communication delay may vary depending on the location where IoT devices exist or what network types such IoT devices use. For example, it takes less than 5 milliseconds for communication in a city, while we need 100 milliseconds or more for communication over transocean circuits.

To understand the impact of encapsulation overhead of the Dripcast framework, we measured the response time of the standard KVS interfaces (GET, PUT – without encapsulation) and the Dripcast framework (JOB[RO], JOB[RW] – with encapsulation) for practical cases in some different locations.

In **Fig. 6**, X-axis is the RTT (round trip time) between client IoT device and the Dripcast cloud, and y-axis is the response time for GET, PUT, JOB[RO] and JOB[RW] operations. We measured the response time of client access as, dots around RTT=1 ms for the communication in a city, dots around RTT=20 ms for nation wide communication (between Sapporo and Tokyo), dots around RTT=70 ms for LTE connected devices (via mobile environment), dots around RTT=135 ms for transocean communication (between Japan and United States).

As shown in the figure, geographical distance is the major factor for the response time. In the current implementation, we use HTTP (Hyper Text Transfer Protocol) to transmit data, for any operation (GET, PUT, JOB[RO] and JOB[RW]). Since HTTP requires at least 2 round trip communications, that is, 1) TCP/IP hand shake to establish the communication channel, and 2) protocol header processing and request/response communication, this is why the response time is 2 times or more of RTT values.

The encapsulating overhead of the Dripcast is less than 1 millisecond for each operation (JOB[RO] or JOB[RW]) and it is relatively small enough in practical cases of IoT applications.

7. Considerations

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

(1) Object oriented modeling on a cloud
(2) Infinite scalability
(3) Less development cost

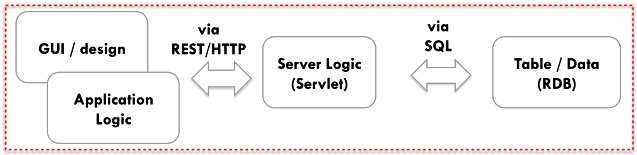

Fig. 7 3 Layers Model.

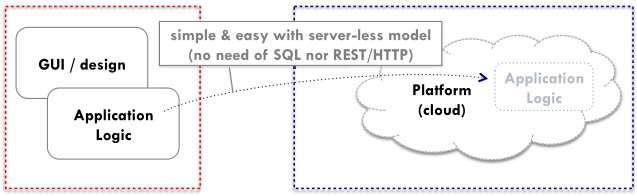

Fig. 8 Dripcast Model.

7.1 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 with local-device programming. If developers write their program based on interface definitions rather than accessing instance or class variables, the source code is fully compatible in both cases for local-device programming or for cloud-ful programming.

7.2 Infinite Scalability

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

7.3 Less Development Cost

The Dripcast framework provides a very simple programming interface for IoT device applications. In the framework, developers have only to write their client-side programs. They do not need to take care of server-side programs nor backend database. This feature dramatically reduces the development cost especially for IoT applications.

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. 7**, developers need to write not only the client application logic but also the server side programs, database programs with SQL and the communication logic between client and server.

In the Dripcast model, developers only need to take care of application logic. As shown in **Fig. 8**, cloud resources are provided as platform service, and the Dripcast framework automatically converts method calls to the associated remote procedure calls on a cloud. There is no need to think of the server side. The framework handles data persistency of associated Java objects in the cloud environment as well.

Of course, such a development model dramatically reduces the development cost for suitable IoT applications.

8. 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 objects 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 describe a simple example of the GPS application in which the Dripcast framework enables server-less device programming.

We also implemented a prototype of the Dripcast framework and evaluated this from two points of view. From the view point of scalability about cloud resources, we described how the architecture is designed in scale-out computing model, and that it's easy to expand or enlarge the cloud computer resources, by simply adding servers. From the view point of client IoT devices, we evaluated the encapsulation overhead in IoT devices. We also described that the overhead is reasonable for practical IoT applications.

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

References

[1] RMI Tutorial, available from ⟨http://docs.oracle.com/javase/tutorial/rmi/index.html⟩.
[2] Brose, G., Vogel, A. and Duddy, K.: *Java Programming with CORBA*, John Wiley & Sons, ISBN 0-471-37681-7 (2001).
[3] Google: Google App Engine, available from ⟨https://developers.google.com/appengine/⟩.
[4] Ghemawat, S., Gobioff, H. and Leung, S.-T.: The Google File System, *SOSP'03* (Oct. 2003).
[5] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E.: BigTable: A Distributed Storage System for Structured Data, *OSDI'06* (Nov. 2006).
[6] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *OSDI'04* (Dec. 2004).
[7] Microsoft: Windows Azure, http://www.windowsazure.com/
[8] Hadoop Project: Hadoop, available from ⟨http://hadoop.apache.org/⟩.
[9] Memcached project: memcached – A distributed memory object caching system, available from ⟨http://memcached.org/⟩.
[10] Cassandra project: The Apache Cassandra Project, available from ⟨http://cassandra.apache.org/⟩.
[11] CouchBase: Document-Oriented NoSQL Database, available from ⟨http://www.couthbase.com/⟩.
[12] FA Labs: Tokyo Cabinet: A modern implementation of DBM, available from ⟨http://fallabs.com/tokyocabinet/index.html⟩.
[13] FA Labs: Kyoto Cabinet: A straightforward implementation of DBM, available from ⟨http://fallabs.com/kyotocabinet/⟩.
[14] ROMA project: A Distributed Key Value Store in Ruby, available from ⟨http://code.google.com/p/roma-prj/⟩.
[15] Basho: makers of the Riak distributed database, available from ⟨http://basho.com⟩.
[16] Nakagawa, I. and Nagami, K.: Jobcast – Parallel and distributed processing framework, *IPSJ Journal*, Vol.21, No.3 (July 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, *Proc. 29th Annual ACM Symposium on Theory of Computing, STOC'97*, pp.654–663, ACM Press (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, *Proc. 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications, SIGCOMM'01*, pp.149–160, ACM Press (2001).

Ikuo Nakagawa is the Executive Chief Engineer at Intec, Inc. and an Invited Associate Professor of Osaka University. He received Ph.D. from The University of Tokyo, Japan, in 2002. His research interest is network, cloud computing and parallel and distributed computing technologies. He is a member of IPSJ.

Masahiro Hiji is a Professor in the Accounting School at Tohoku University, Japan. He works for Hitachi Solutions East Japan, Ltd. as senior chief engineer. He received his Ph.D. in information science from Tohoku University in 1997. His research expertise includes decision support system, distributed processing, simulation and gaming, and sensing network platform. He is member of IEEE, IPSJ, IEICE, and JSSST.

Hiroshi Esaki received his B.E. and M.E. degrees from Kyushu University, Fukuoka, Japan, in 1985 and 1987, respectively. And, he received Ph.D. from The University of Tokyo, Japan, in 1998. In 1987, he joined Research and Development Center, Toshiba Corporeation. From 1994 to 1996, he has been at CTR (Center for Telecommunication Research) of Columbia University in New York (USA). During his staying at Columbia University, he has proposed the CSR architecture, that is the origin of MPLS (Multi-Protocol Label Switching), to the IETF and to the ATM Forum. Since 1998, he has served as a professor at The University of Tokyo, and as a director of WIDE Project (www.wide.ad.jp). He is a fellow of IPv6 Forum, a vice president of JPNIC and a Board of Trustee for ISOC (Internet Society).