

# 3 Cloud Computing Technology

## 3.1 Introduction

It is said that cloud computing represents a paradigm shift from owning computing resources to using them. At IJ, we have been developing and operating technology for supporting cloud computing for the past few years, in order to implement the immense data processing power and efficient infrastructure that it provides.

Here, we will explain the distributed system called “ddd,” which is one of the proprietary cloud technology infrastructures that IJ has developed and implemented, and is using as a service infrastructure.

## 3.2 About Distributed Systems

Before we explain ddd, we will define the distributed systems that we will discuss here. By definition, a distributed system is (1) - made up of multiple computer nodes and (2) visible as a single system to its users.

Some examples of distributed systems are distributed storage used for retaining large quantities of data, and distributed data processing. (Figure 1, Figure 2)

The purpose of distributed systems is to use multiple computer nodes (hereinafter referred to as “nodes”) to improve the processing power of a system to levels higher than a single computer is capable of, or to increase availability. Distributed systems involve more than simply setting up multiple computers, also requiring technology for coordinating their operation.

Large quantities of data must be collected in the cloud, because it is expected to handle the storage and processing of vast amounts of data efficiently. Availability requirements are also increasingly rigorous. Large-scale distributed systems are becoming an extremely important element of cloud computing.

## 3.3 Examples of Distributed Systems

Distributed systems are a field in which technical development is currently being carried out actively. Here are several well-known examples.

- Google File System (GFS)  
This is a distributed file system created by Google for handling large quantities of high-volume data. Its implementation has not been revealed outside Google. (<http://labs.google.com/papers/gfs.html>)
- MapReduce  
A framework for large-scale distributed processing devised by Google (details described later). (<http://labs.google.com/papers/mapreduce.html>)

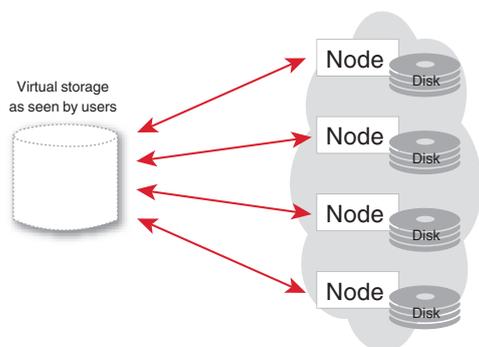


Figure 1: Distributed Storage

Stored data is distributed between multiple nodes. This appears as a single storage device to users.

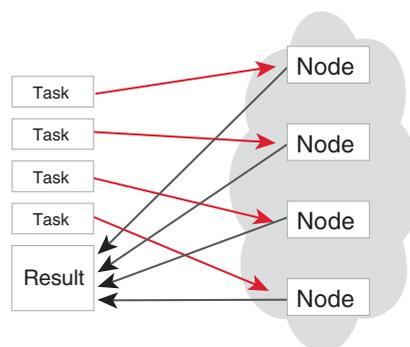


Figure 2: Distributed Data Processing

Data to be processed is divided into many tasks, and processed in parallel using multiple nodes.

- Amazon Dynamo

A distributed key-value store developed by Amazon. Its implementation has also not been made public. ([http://www.allthingsdistributed.com/2007/10/amazons\\_dynamo.html](http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html))

- Hadoop

This is a distributed system that was created using the above GFS and MapReduce systems as a reference. It is written in Java, and published as open source software. (<http://hadoop.apache.org/>)

Of these systems, Google File System and Amazon Dynamo are categorized as distributed storage, and MapReduce is categorized as distributed data processing technology. The ddd service that IJ has developed incorporates both of these functions.

### 3.4 History of DDD Development

“ddd” is an abbreviation of “distributed database daemon,” and was created to analyze the enormous volume of traffic that flows through the IJ backbone.

Internet service providers (ISPs) obtain and analyze backbone router traffic information to operate their backbone in a stable manner. In many cases, the counter value for each router interface is obtained via SNMP and converted into a graph. However, as it is not possible to gauge traffic conditions from simple interface IN/OUT information alone, there may be inadequacies with regard to monitoring and analysis.

For this reason, more detailed traffic information known as flow information is used. Some high-end routers and switches used in ISP backbones are capable of outputting flow statistics which include the source of packets, the IP addresses of destination, and port numbers. By obtaining and analyzing this information, it is possible to monitor the status of communications in a level of detail that SNMP cannot capture.

For example, Figure 3 is a graph that indicates traffic color-coded by destination AS number based on flow information. There is a sudden drop in traffic at the right edge of the graph, and when this is analyzed using flow information, it is possible to determine that communications to only a portion of AS have decreased. As only the total traffic volume can be ascertained using SNMP, when it is difficult to identify the cause of anomalies or status changes, flow information can make a detailed analysis possible.

One issue with analyzing flow information is the enormous quantities of data to be handled. While it makes the detailed analysis of communication content possible, the amount of data to be handled also inevitably increases. Generally, flow information is aggregated to some extent immediately after it is received, and then stored after the volume of data is reduced. For example, one method involves aggregating information in source address units, and only storing the total value. However, aggregate values make it impossible to perform a detailed analysis at a later stage, so at IJ we preserve almost all flow information, and only extract or aggregate information when necessary. For ISPs such as IJ that operate a large number of network equipment, this method makes the handling of extremely large data necessary.

Previously, IJ stored flow information in a standard relational database. However, although each record in flow information is small, in the space of five minutes anywhere from hundreds of thousands to millions of records can be generated, so only information for a very short period of time can be stored using a traditional database.

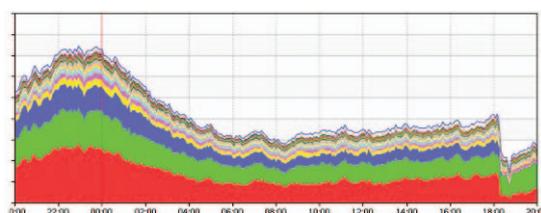


Figure 3: Graph Showing Traffic Color-Coded by AS Number

To handle enormous amounts of flow information over a long period of time, IJ decided to develop a proprietary distributed system. The result is “ddd”. It can store more than tens of billions of flow information records, and sample and aggregate data at high-speed using a variety of criteria.

The development of ddd began with the handling of flow information as the primary goal, but currently it is applied to a wider range of data processing, such as log analysis for security and application services, and analysis of internal and external security information.

### 3.5 Overview of DDD

“ddd” is a distributed system that IJ uses internally, which makes high scalability possible while using low-cost PCs.

The characteristics of ddd are as follows.

- A pure P2P configuration with no single point of failure
- Dynamically extensible distributed storage featuring an automatic data redundancy function
- High-speed data processing via MapReduce

Each ddd node uses the three-layered structure detailed below. All nodes have the same structure. (Figure 4)

Each item is explained in order below.

#### 3.5.1 Pure P2P

Each ddd node forms part of a pure P2P network with no central administrative host. As there is no central administrative host, there is no single point of failure. Each node is equal, and operates in coordination with others. When combined with the data redundancy features described later, the entire system will operate without issue regardless of the time a failure occurs or the specific node it occurs in.

When adding a new node, it is connected to any existing node and booted up. The new node obtains information such as the IP addresses of other nodes from the existing node it is connected to. At the same time, information about the new node is broadcast to other existing nodes.

While ddd is running, it exchanges information with other nodes every few seconds. This means that each node has information about all other nodes. A group of nodes that share information with each other like this is called a cluster.

#### 3.5.2 Distributed Storage

##### ■Key-Value Store

The storage portion of ddd is classified as a distributed key-value store. A distributed key-value store is a type of simple database that stores data as a combination of key and value, and manages these across multiple nodes.

Key-value stores have more limited features compared to existing relational databases (RDB), and have no function for combining data (JOIN) or mechanism for preserving the consistency of transactions. Basically, they only allow a key to be specified, and the corresponding value read or written.

While having limited features, they are ideally suited for distribution across nodes designated according to key value, and as this increases scalability, they are sometimes referred to as the data store for the age of cloud computing.

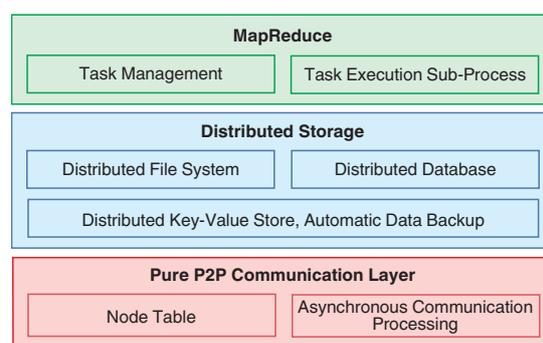


Figure 4: Outline of DDD Internal Structure

To designate nodes according to key, ddd uses an algorithm known as consistent hashing.

Consistent hashing treats each node and key as if they are arranged in a hash space on a logical ring (this arrangement is unrelated to the physical network topology of nodes). A node ID is assigned to each node based on its hashed IP address value. The key to be stored is also mapped to the hash space in the same way, and the first node encountered when traveling in a counterclockwise direction is assigned as the storage node of that key for processing. (As SHA-1 is used as the hash function, the size of the hash space is 2 to the power of 160.) Using this method, it is possible to compute the storage node by simply calculating a hash value from the key value, independent of the number of keys stored or the amount of data. (Figure 5)

The largest merit of consistent hashing is that it limits the scope of keys affected when nodes are added or removed. For example, if Node B in the figure below was removed due to failure, the only keys affected would be those indicated by the light green area, with other keys not affected at all. (Figure 6)

In a distributed system environment, it is necessary to consider the routine addition and removal of nodes as a prerequisite. Consistent hashing has the advantage of being flexible when such addition or removal occurs, and it is an algorithm that is often used in distributed key-value stores.

■Automatic Data Redundancy

For redundancy purposes, ddd duplicates all data over three different nodes. Earlier we explained that when using consistent hashing, keys are stored in the first node encountered from their hash value in a counterclockwise direction, but the same data is copied to the second and third nodes encountered as well. In other words, unless all three nodes fail simultaneously, data is preserved on one of the nodes, thereby achieving highly reliable data preservation.

When a node fails, the number of copies of a piece of data temporarily falls to one or two, instead of three (we do not consider a fall to zero likely). To ensure that this situation is not prolonged, ddd copies remaining data to always preserve three copies. To achieve this, ddd uses a system called ihave/sendme, as shown in Figure 7.

First, each node lists the keys that it has stored. As the nodes that should be assigned to a key can be calculated from the key value using consistent hashing, key lists are sent to the corresponding nodes. This is called an ihave message. Nodes receiving an ihave message compare the keys included in the list with the keys they have stored, and if any of the keys are missing, they return a request for that data to be sent. This is called a sendme message. Nodes receiving a sendme message send data to the source that requested it. As there are a large number of keys, key information is exchanged in small batches instead of all at

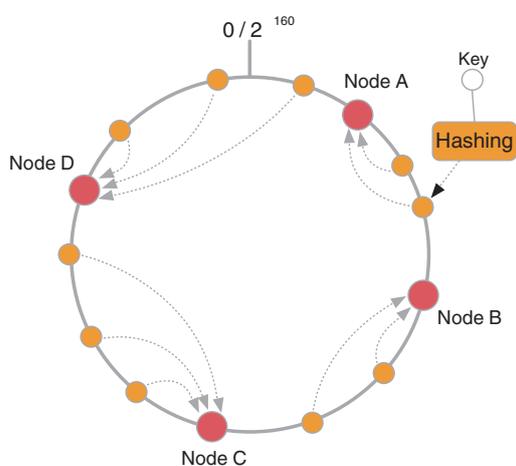


Figure 5: Consistent Hashing - Logical Placement of Keys and Nodes

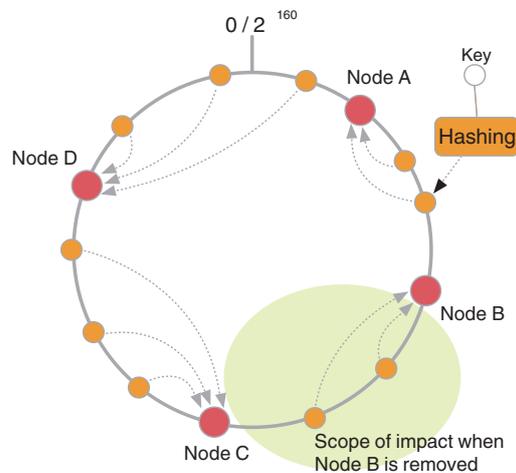


Figure 6: Consistent Hashing - Scope of Impact During Node Failure

once. Additionally, when nodes are added or removed, the *ihave* destination computed using consistent hashing changes for affected keys only, with data transferred to the newly assigned node. In Figure 6, if Node B was removed, the nodes assigned to the keys indicated by the light green area would change from Nodes B, A, and D to Nodes A, D, and C, and data would be transferred from either A or D to the newly assigned C. This cycle is repeated endlessly in *ddd*, and as a result, three copies of data are preserved with only a small amount of lead time.

In order to keep the hardware costs for nodes low, *ddd* does not use disk redundancy technology such as RAID. Instead of this, data redundancy is carried out at a higher level, implementing a system that ensures smooth operation regardless of when a node fails or is removed.

Meanwhile, as the same stored data is distributed over multiple nodes using distributed storage, it is not always possible to preserve the consistency of data. In distributed systems with multiple nodes connected over a network, some nodes may fail or be cut off from the network. Not being able to guarantee consistency is a necessary trade-off for maintaining high availability under these circumstances. In many cases inconsistencies occur for only very short periods, and the system has mechanisms for maintaining consistency over the course of time, but there is also a need for inconsistencies to be dealt with on the application side.

*IJ* stores data such as the aforementioned flow information and logs for each server on *ddd* distributed storage, using the equipment ID and time as keys. Flow information and log information details remain constant, so there is no need to update the information that has been written. Because of this, the previously mentioned characteristic of lacking guaranteed consistency is very rarely a problem. However, there is the possibility of data not being present in a node that it should be stored on during the moment when data is copied after a node is added or removed. In this case, applications using *ddd* will try to obtain the data from the second and third candidate nodes.

### 3.5.3 MapReduce

MapReduce is a programming model for large-scale data processing devised by Google. As evident from its name, under this model data is processed in two phases: the map phase and the reduce phase. When used correctly, it makes efficient distributed parallel processing over multiple nodes possible. (Figure 8)

The source code for Google's MapReduce has not been made public, but a paper on its structure has been published by Google. Today a number of implementations with similar functions based on it have appeared, and are being used for tasks such as indexing search engines and analyzing log files. A MapReduce function is also used in *ddd* for processing distributed storage data.

Another way of describing "map" and "reduce" is "filter" and "aggregate." During the map (filter) phase, necessary information is filtered from data, and converted into a format that makes subsequent processing easier when required. During the reduce (aggregate) phase, mapped information is aggregated. When there are no dependencies between the information to process, processing can be shared over multiple nodes executed in parallel.

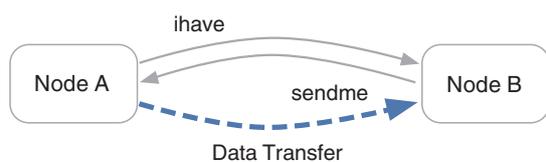


Figure 7 *ihave*/*sendme* System

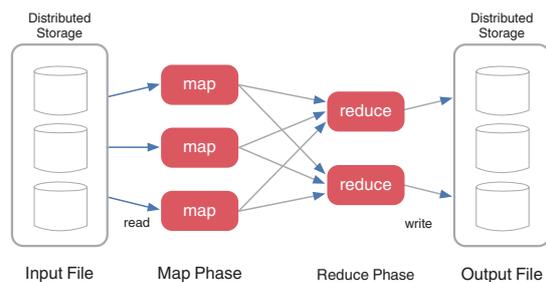


Figure 8: MapReduce Concept

The simplest and easiest to understand application of MapReduce is distributed grep. Grep is a command associated with Unix-based OSes that searches for and outputs lines in a file that match a specified pattern. Because grep performs a round-robin search each time instead of creating a search index in advance, it can be time consuming when used with large files or with many files at once. As grep processing of target files can be shared over multiple nodes when MapReduce is used, users can expect the total processing time to be reduced.

As mentioned previously, flow information and logs output by servers are stored on distributed storage at IJ, and MapReduce is used to filter and process data. A diverse combination of parameters can be used to analyze this data, and we have constructed a system that makes it possible to analyze data using a variety of parameters according to the type of analysis.

Below we explain the actual behavior of MapReduce using the analysis of flow information as an example.

- A MapReduce job request is prepared, including parameters such as the routers to analyze, the target period, the filtering criteria, and the axis to group.
- The client sends the MapReduce job request to a ddd node.
- The node receiving the request breaks down the MapReduce job into multiple map tasks and reduce tasks, dividing the target period into fixed intervals of time.
- The map tasks are assigned to each node, and each node processes filtering and grouping in accordance with the parameters.
- Once execution of the map tasks on each node is complete, the reduce task is initiated and results are aggregated.
- The aggregated results are written to distributed storage. It is also possible to return them to the client.

Using systems such as this, it is often only possible to retain data for an extremely short period, and analysis can take a long time. However, by using a distributed system made up of many nodes at IJ, we have made it possible to analyze data from long periods of time with a practical response time.

### 3.6 Conclusion

Here, we explained the ddd distributed system developed by IJ. Using ddd it is possible to retain and process large amounts of data. In the future, it is conceivable that provider infrastructures will handle even larger volumes of data. IJ will continue to develop ddd and provide reliable services as a social infrastructure.

Author:

**Takahiro Maebashi**

System Development Section, System Infrastructure Division, IJ Service Business Department

Mr. Maebashi works on the development of programs related to IJ backbone network operation, starting with the implementation of ddd.