The 6th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2015)

Pervasive Data Processing

Ichiro Satoh

National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430 Japan

Abstract

This paper presents a data processing framework for enabling MapReduce approach to be available in pervasive networks, including sensor networks and Internet of Things (IoT). It is unique among other existing MapReduce-based approaches, because it can locally process data maintained on nodes in pervasive networks. It dynamically deploys programs for data processing at the nodes that have the target data as a map step and executes the programs with the local data. Finally, it aggregates the results of the programs to certain nodes as a reduce step. The paper proposes the architecture of the framework and describes its basic performance and application.

© 2014 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license

Keywords: Data processing, distributed systems, sensor network

1. Introduction

Pervasive networks connects a variety of devices such as everyday consumer objects and industrial equipment onto the network, enabling information gathering and management of these devices via software to increase efficiency, enable new services, or achieve other health, safety, or environmental benefits. The networks generate large quantities of data that need to be processed and analyzed in real time. They assumed to transfer massive amounts of small message sensor data to data centers or cloud computing environments for processing, because the computational resources of their devices have assumed to be limited. Several approaches to processing a large amount of data at data centers. Among them, MapReduce is one of the most typical and modern computing models for processing large data sets in distributed systems. It was originally studied by Google and inspired by the map and reduce functions commonly used in parallel list processing (LISP) or functional programming paradigms. Hadoop, is one of the most popular implementations of MapReduce and was developed and named by Yahoo!

* Corresponding author. Tel: +81-3-4212-2546.
E-mail address: ichiro@ni.ac.jp

1877-0509 © 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the Program Chairs
doi:10.1016/j.procs.2015.08.307
Processing large quantities of data generated from devices in real time will increase as a proportion of inbound traffics and workloads in networks from pervasive networks to data centers. However, bandwidth of networks between pervasive networks and data centers tend to be slow and unreliable. Pervasive networks generate massive amounts of input data from nodes. Transferring the entirety of that data to a single location for processing will not be technically and economically viable.

However, modern pervasive devices tend to have certain amounts of computational resources. For example, a Raspberry Pi computer, which has been one of the most popular embedded computers, has 32 bit processor (700 MHz), 512 MB memory, and Ethernet port. Therefore, such pervasive devices have potential capabilities to execute a small amount of data processing. In fact, we have already installed and evaluated Hadoop on Raspberry Pi computers with Linux, but its performance is not practical even when the size of the target data is small, e.g., less than 10MB.

Hadoop has been essentially designed for be executed on high performance servers and it is complicated so that it is almost impossible to redesign Hadoop for pervasive devices, e.g., embedded computers. Therefore, this paper is to propose a MapReduce framework available at limited computers and network, e.g., Raspberry Pi computers, independently of Hadoop. The framework has three key ideas. to save computational resources at node. The first is to deploy and execute programs for data processing at nodes that has the target data. The second is to introduce management functions into programs for data processing. The third is to provide KVS for MapReduce processing available with limited memory.

The author proposed another MapReduce framework based on mobile agent technology, independently of the previous one except for the notion of the deployment of programs for data processing. The framework proposed in this paper is constructed based on our previous framework but it is designed for executing on embedded computers or IoT devices.

2. Related Work

The tremendous opportunities to gain new and exciting value from big data are compelling for most organizations, but the challenge of managing and transforming it into insights requires new approaches, such as MapReduce processing. It originally supported map and reduce processes. The first is invoked dividing a large scale data into smaller sub-problems and assigning them to worker nodes. Each worker node processed the smaller sub-problems. The second involves collecting the answers to all the sub-problems and aggregates them as the answer to the original problem it was trying to solve. There have been many attempts to improve Hadoop, which is an implementation of MapReduce by Yahoo! in academic or commercial projects. However, there have been few attempts to implement MapReduce itself except for Hadoop. For example, the Phoenix system and the MATE system supported multiple core processors with shared memory. Also, several researchers have focused on iteratively executing MapReduce efficiently, e.g., Twister, Haloop, MRAP, and SSS. These implementations, except for SSS, assume data in progress to be stored at temporal files rather than key-value stores in data nodes and SSS executes data stored in a key-value store shared from task nodes and then its results the key-value store. They assume data to be stored in high-performance servers for MapReduce processing, instead of in the edges.

Google’s MapReduce, Hadoop, and other existing MapReduce implementations have assumed their own distributed file systems, e.g., the Google file system (GFS) and Hadoop file system (HDFS), or shared memory between processors. For example, Hadoop needs to move target data from the external storage systems to HDFS via networks before processing these.

Our MapReduce system does not move data between nodes. Instead, it deploys program codes for defining processing tasks to nodes that have data by using the deployment of components corresponding to the tasks and it executes the codes with their current local data. Hadoop and its extensions are unsuitable in sensor networks and embedded computers, because its file system, HDFS, tends to become a serious bottlenecks in the operation of Hadoop and it often requires wide band networks, which may not be available in sensor nodes or embedded computers. In the literature on sensor networks, The Internet of Things (IoT), and machine-to-machine (M2M) communications, several academic or commercial projects have attempted to support data at the edge, e.g., at sensor nodes and embedded computers. For example, Cisco’s Flog Computing and EMC’s computing intend to integrate cloud computing over the Internet and peripheral computers. However, most of them do not support the aggregation of data generated and processed at the edge.
3. Requirements

Before explaining our system, let us discuss requirements.

- MapReduce processing and its clones, e.g., Hadoop, are one of the most popular data processing framework. It should be available in pervasive networks, which generates a large amount of data from sensor nodes.
- Pervasive networks tend to be wireless or low-band wired, like industry-use networks. They have non-neglectable communication latency and are not robust in congestion. The transmission of such data from nodes at the edge to server nodes seriously affects performance in analyzing data and results in congestion in networks.
- Nodes in pervasive networks have non-powerful 32 bit processors with small amounts memory, like Raspberry Pi computers. We assume that our framework is available on a distributed system consisting of Raspberry Pi computers.
- In pervasive networks, a lot of data are generated from sensors. Pervasive nodes locally have their data inside their storage, e.g., flash memory.
- Every node may be able to support management and/or data processing tasks, but may not initially have any codes for its tasks.
- Unlike other existing MapReduce implementations, including Hadoop, our framework should not assume any special underlying systems.¹ There is no centralized management system in pervasive networks. Our framework should be available without such a system.

Our approach assumes data can be processed without exchanging data between nodes. In fact, in pervasive networks data that each node has is generated from the node’s sensors so that the data in different nodes are independent of one another. Therefore, this assumption is reasonable. One of the most popular extensions of MapReduce, including extensions of Hadoop, to improve performance of iterative processing the same data. However, our framework does not aim at such a iterative processing. This is because most data at sensor nodes or embedded computers are processed only once or a few times. Suppose analyzing of logs at network equipment. Only updated log data are collected and analyzed every hour or day instead of the data that were already analyzed.

4. Approach

To solve the requirements discussed in the previous section, we propose the following design principles.

- *Dynamically deployable component* Our framework enables us to define data processing tasks as dynamically deployable components. To save network traffics, task should be deployable at computers that have the target data. In fact, the sizes of programs for defining tasks tend to be smaller than the sizes of the data so that the deployment of tasks rather than data can reduce network traffics.
- *Data processing-dependent networking* In MapReduce processing communication between nodes tend to depend on application-specific data processing. Each node, including master and data nodes in Hadoop, must have a general-purpose runtime systems to support a variety of data processing. However, such a runtime system tends to consume more memory rather than peculiar purpose one. Our framework enables networking for MapReduce processing to be defined in programs for data processing so that our runtime systems do not need to provide a variety of networking.
- *MapReduce’s KVS for limited memory* In general, MapReduce processing tends to spend a much amount of memory in its reduce phase, because the phase combines than two data entries via KVS. The KVS that our approach introduces should be designed to save memory. Reducing data entries in KVSs, which are located at different computers, tends to have much traffics. Our framework transmit data between nodes in a desynchronization for the reason of avoiding congestion.

¹ Hadoop has is been not available in Windows, because it needs a permission mechanism peculiar to Unix and its families.
The framework introduces the deployment of software components as Map phase in MapReduce processing like Hadoop. However, the components are autonomous in the sense that each component can control its destinations and itineraries under its own control. The framework allows developers to define their MapReduce processing from three parts, map, reduce, and data processing, as Java classes, which can satisfy specified interfaces. The map and reduce classes have similar methods in the Mapper and Reducer classes in Hadoop. The data processing parts are responsible for data processing at the edges. They consist of three methods corresponding to the following three functions: reading data locally from nodes at the edge, data processing of the data, and storing their results in a key-value store format. Figure 1 outlines the basic mechanism for processing.

**Map phase**  A Mapper component makes copies of Worker component and dispatches the copies to the nodes that locally have the target data.

**Data processing phase**  Each of the Worker component executes its processing at its current data node. After executing its processing, it stores its results at the KVS of its current node.

**Reduce phase**  The KVS of each of the node returns only the updated data to the computer that the Reducer component is running according to their networking. The Reducer component collects the results from the Worker components via its KVS.


diagram

Each Worker component assumes to be executed independently of the others. Mapper and Reducer components can be running on the same node.

5. Design and Implementation

Our framework consists of two layers; runtime systems and components (Figure 2). It was implemented with Java language and operated on the Java virtual machine. The former is defined with built-in Java classes by users
as Jar-components corresponding to map and reduce processing and data processing tasks, where they are defined as Mapper, Reducer, and Worker components. We call a set of Mapper, Reducer, and Worker components as a session.

---

**Fig. 2. System structure**

5.1. Runtime system

Each runtime system runs on a computer and is responsible for executing Mapper, Worker, and Reducer component. It also establishes at most one TCP connection with each of its neighboring systems in a peer-to-peer manner without any centralized management server and it exchanges control messages, components, and KVS-formatted data through the connection. Each runtime system is light so that it can be executed on embedded computers, including JVMs for embedded computers.

5.2. Key-value-store

MapReduce processing should be executed on each of the data nodes independently as much as possible to reduce data transmissions through networks. However, data may not be divided into independent pieces. To solve this problem, Hadoop enables data nodes to exchange data with one another via HDFS, because HDFS is a distributed file system shared by all data nodes in a Hadoop cluster, like the GFS. Instead, our framework provides a tree structure key value store (KVS), where each KVS maps an arbitrary string value and arbitrary byte array data and is maintained inside its agent, and directory servers for KVSs. Each session consists of a tree structure KVS, where its root tree is managed by a Reducer component and its subtrees are located at nodes that the session’s Worker runs at. To support reduce processing, the root of the KVS merge its subtrees into itself. To reduce the amount and congestion of data transmission for merging subtrees, each of keys’ entries in subtree has two flags.

- **Update flag** specifies whether the value of the key needs to be merged into the root tree. Only when the flag is positive, the runtime system transmits a pair of its key and value to the node that the Reducer component runs. The flag is useful to reduce the amount of data transmission.
- **Complete flag** specifies whether the value of the key will not be changed. After the flag becomes positive, the runtime system starts to transmit a pair of its key and value to the node that the Reducer component runs, but not after the Worker component completes. The flag is useful to avoid congestion at a network between the Worker’s node and Reducer’s node.

Runtime systems for executing Worker components transmit only the results on KVS whose update and complete flags are positive before the components finish so that data transmission from Worker to Reducer components can be minimized and temporally distributed. Therefore, we can relax limitations at pervasive networks.
5.3. Component

Our framework supports data processing on nodes, e.g., sensor nodes and embedded computers, which may be connected through non-wideband and unstable networks, whereas existing MapReduce implementations aim at data processing on high-performance servers connected through wideband networks. Therefore, we cannot directly inherit a programming model for existing MapReduce processing. In comparison with other MapReduce processing, including Hadoop, our framework explicitly divides the map operations into two parts in addition to the part corresponding to the reduce operation in MapReduce.

- **Duplication and deployment of tasks at data nodes** Developers specify a set of the addresses of the target data nodes that their data processing are executed or the network domains that contains the nodes. If they still want to define more complicated MapReduce processing, our framework is open to extend the Mapper and Reducer components.

- **Application-specific data processing** They define the following three functions: reading data locally from nodes at the edge, data processing of the data, and storing their results in a key-value store format. These functions can be isolated so that developers can define only one or two of the functions according to the requirements of their data processing.

- **Reducing data processing results** They define how to add up the answers of data processing stored in a key-value store.

Although the first is constructed in Mapper and Worker components, the second in only Worker components, the third in Worker and Reducer components, developers focus on the above three parts independently of their runtime systems. Our framework enables us to easily define application-specific Mapper, Reducer, and Worker components as subclasses of three template classes that corresponds to Mapper, Reducer, and Worker respectively, with several libraries for KVS. When an Mapper component gives one or more Worker components no information, we can directly define the component from the template class for Mapper. It can create specified application-specific Worker components according to the number of one or more specified data and deploy them at the nodes. When Reducer components support basic calculations, e.g., adding up, averaging, and discovering maximum or minimum values received from one or more Worker components through KVSs according to the keys, we can directly define them as our built-in classes.

5.4. Fault-tolerance

The job manager in Hadoop is responsible for supporting fault-tolerance against crash failure in data nodes. The manager detects failures in data nodes, because each task tracker running on a data node sends heartbeat messages to the job tracker every few minutes to inform of its status. Since data are shared by worker nodes, the job tracker pushes work out to available task tracker nodes in the cluster, striving to keep the work as close to the data as possible.

However, our system assumes that data are maintained in one data node so that it has a different policy for fault tolerance. If a data node is stopped or disconnected, the job tracker does not need to exclude such a node. Our system introduces a mobile agent-based job tracker manager, called a system manager agent, which has a Java Management Extension (JMX) interface to monitor data nodes and it periodically sends messages to data nodes. When they receive a message, data nodes returns their status to the system manager agent.

- If a data node has crashed, before the Mapper agent dispatches Worker agents, the system manager agent informs Mapper agents to leave out the crashed node from the list of the target data nodes.

- If a data node has crashed, after Worker agents are deployed at the target node, the system manager agent informs the Reducer agent to leave out agents returned from nodes from the agent’s waiting list. Even when the crashed node can be restarted or continue to work, the Reducer agent does not wait for any agents from the node.
To mask failures in the nodes that run *Mapper* and *Reducer* agents, the *system manager* agent can explicitly make clones of these agents at other nodes, because they are still mobile agents.\(^2\) The current implementation has no fault-tolerant mechanism for failures while *Worker* agents are deployed and running, because our MapReduce processing is not heavy. We should restart the processing again.

5.5. Security

The current implementation is a prototype system to dynamically deploy the components presented in this paper. Nevertheless, it has several security mechanisms. For example, it can encrypt components before migrating them over the network and it can then decrypt them after they arrive at their destinations. Moreover, since each component is simply a programmable entity, it can explicitly encrypt its individual fields and migrate itself with these and its own cryptographic procedure. The JVM could explicitly restrict components so that they could only access specified resources to protect computers from malicious components. Although the current implementation cannot protect components from malicious computers, the runtime system supports authentication mechanisms to migrate components so that all runtime systems can only send components to, and only receive them from, trusted runtime systems.

6. Performance Evaluation

Although the current implementation was not constructed for performance, we evaluated that of several basic operations in a distributed system consisting of five networked embedded computers as data nodes connected through Fast (100 Mbps) Ethernet via an Ethernet switch. Each embedded computer was a Raspberry Pi, where its processor was Broadcom BCM2835 (ARMv6-architecture core with floating point) running at 700Mhz and it has 512MB memory, a Fast Ethernet port, and SD card storage (16GB SDHC), with Raspbian, which is a Linux optimized to Raspberry Pi, and OpenJDK 6. Java heap size was limited to 384 MB. We compared the basic performances of our framework and Hadoop. Among the five computers, one executes our *Mapper* and *Reducer* components or the master node in Hadoop. Others are data nodes in our framework and Hadoop. The Reducer component added up the numbers of each of the words received from the four *Worker* components for word counting obtained from their nodes via KVS. We compared between our system and Hadoop-based system. Figure 3 shows the costs of counting words by our framework and Hadoop. The former is faster than the latter, because the former is optimized to be executed in pervasive networks.

\(^2\) The current implementation does not support consistency between original agents and their clones.
The readers may think that the application is not real. We evaluated our approach in an abnormal detection from data measured by sensors. It detected anomalous data, which were beyond the range of specified maximum and minimum values. This evaluation assumed each data node would have 0.01% of abnormal data in its stream data generated from its sensor every 0.1 second and each data entry 16 bytes. We detected abnormal values from the data volume corresponding to data stream for one year at each of eight data nodes. The whole the amount of the whole data was about 5.04 GB and the amount of abnormal data was 504 KB in each node. When we used Hadoop, we need to copy about 40 GB data, i.e., multiply 5.04 GB by 4, from data nodes to HDFS.

7. Discussion

Our framework is to deploy programs for data processing at the nodes, e.g., sensor nodes and embedded computers, that stores the target data, where existing MapReduce implementations deploy the data at the file systems, GFS and HDFS, shared by the servers that execute their processing. Our framework has several advantages. For example, it has no cost to deploy data, which tend to be huge, to the file system. In fact, it has better results in its performance in comparison with Hadoop. It can process the data at the original nodes so that it can process the data that cannot be taken out for the reason of security and access limitation. Its architecture is simple because its whole processing is constructed from a combination of Mapper, Worker, and Reducer agents rather than the underlying centralized management system. As a result, it can support light-weight data processing. However, it has several disadvantages. It assumes that data processing on each data node is isolated from other nodes, because it lacks any mechanisms to exchange the data between data nodes. However, in our potential applications, e.g., data collections from sensor nodes, data processing at nodes can be executed isolatedly. It is not suitable for iteratively executing MapReduce processing for the same data.

8. Conclusion

We presented a distributed processing framework based on MapReduce processing. It was designed for analyzing data at the edges of networks. It could distribute programs for data processing to nodes at the edges as a map operation, execute the programs with their local data, and then gather the results according to user-defining reduce operation at a node. As mentioned previously, our framework is useful for thinning out unnecessary or redundant data from the large amounts of data stored at nodes in pervasive networks, e.g., sensor nodes and embedded computers, connected through low-bandwidth networks. It enables developers to focus on defining application-specific data processing at the edges without any knowledge on the target distributed systems.

References