Agent-based MapReduce Processing in IoT

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

Keywords: IoT, Mobile Agent, MapReduce Processing.

Abstract: This paper presents an agent-based framework for processing data at nodes on the Internet of Things (IoT). The framework is based on MapReduce processing, where the MapReduce processing and its clones are popular but inherently have been designed for high-performance server clusters. It aims at enabling data to be processed at nodes on IoT. The key idea behind it is to deploy programs for data processing at the nodes that contain the target data as a map step by using the duplication and migration of agents and to execute the programs with the local data. It aggregates the results of the programs to certain nodes as a reduce step. We describe the architecture and implementation of the framework, its basic performance, and its application are also described here.

1 INTRODUCTION

The Internet of Things (IoT) connects devices such as everyday consumer objects and industrial equipment onto the network, enabling for gathering data about the real world. IoTs generates large quantities of raw data generated from sensors. Since such data tend to be raw and contain much noise, they need to be processed before being analyzed. Data processing assumes to be performed in clusters of high performance servers, including data centers. Large quantities of data generated from IoT devices will increase as a proportion of inbound traffics and workloads in networks from IoT to data centers. Transferring the entirety of that data to a single location for processing will not be technically and economically viable.

However, modern IoT 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 IoT 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.

In a big-data setting, MapReduce is one of the most typical and popular computing models among them for processing large data sets in distributed systems. It was originally studied by Google (Dean and Ghemawat, 2004) 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!. MapReduce and its implementations have been essentially designed for be executed on high performance servers.

This paper is to propose a MapReduce framework available at limited computers and network in IoT, e.g., Raspberry Pi computers, independently of Hadoop. The key behind the framework is to implement and operate Mapreduce processing at IoT by using mobile agent technology.@The *map* phase of MapReduce processing is constructed in a mobile agent that duplicates another agent, which can carries programs for data processing and deploys their copies at IoT nodes that has the target data. The clones agents executes their programs at their destinations and then carry their results to a specified node. The node aggregates the results into its final result according its program as the *reduce* phase of MapReduce processing. Our framework is available on a distributed system consisting of Raspberry Pi computers.

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,

Satoh, I.
Agent-based MapReduce Processing in IoT.
DOI: 10.5220/0005802102500257
In *Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016) - Volume 1*, pages 250-257
ISBN: 978-989-758-172-4

such as MapReduce processing. It originally supported *map* and *reduce* processes (Dean and Ghemawat, 2004). 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 (Talbot et al., 2011) and the MATE system (Jiang et al., 2010) supported multiple core processors with shared memory. Also, several researchers have focused on iteratively executing MapReduce efficiently, e.g., Twister (Ekanayake et al., 2010), Haloop (Bu et al., 2010), MRAP (Sehrish et al., 2010). These implementations assume data in progress to be stored at temporal files rather than key-value stores in data nodes. They assume data to be stored in high-performance servers for MapReduce processing, instead of in the edges. These works may be able 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.

A few researchers have proposed their original MapReduce processing frameworks for embedded or mobile computers. The Misco (Dou et al., 2010) system was a framework for executing MapReduce processing on mobile phones via HTTP. Elespuro et al. developed a system for executing MapReduce using heterogeneous devices, e.g., smartphones, from a mobile device client for iPhone (Elespuru et al., 2009). These were aimed at executing data processing at nodes, e.g., smart phones and embedded computers, like ours. Our main differences from theirs are that our framework intends to execute data processing at computers that have the target data at arbitrary stores of the computers rather than at smart phones or embedded computers assigned as certain work nodes. In the literature on sensor networks, the IoT, and Machine-to-Machine (M2M), several academic or commercial projects have attempted to support data at nodes on IoT at sensor nodes and embedded computers. For example, Cisco's *Flog Computing* (Bonomi et al., 2012) and EMC's the IoTs 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 nodes. The author presented a MapReduce-based framework for processing data in a previous paper (Satoh, 2013), but the previous framework assumed to be executed on clusters of high-performance servers, rather than embedded computers.

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 IoT, which generates a large amount of data from sensor nodes.
- Networks in IoT 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.
- Modern computers on IoT have 32 bit processors with small amounts memory, like Raspberry Pi computers.
- In IoT, a lot of data are generated from sensors. Nodes at IoT 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 IoT. Our framework should be available without such a system.

Our framework assumes data can be processed without exchanging data between nodes. In fact, in IoT data that each node has is generated from the node's sensors so that the data in different nodes can be processed independently of one another.

[1]Hadoop has is been not available in Windows, because it needs a permission mechanism peculiar to Unix and its families.

4 MOBILE AGENT-BASED MAPREDUCE FRAMEWORK

To solve the requirements discussed in the previous section, our framework introduces mobile agent technology into data processing. It has 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.

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.

Our framework supports MapReduce processing with mobile agents. Figure 1 outlines the basic mechanism for processing, which involves five steps.

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.

Note that the number of results is by far smaller than the amount of target data. Each *Worker* agent assumes it is to be executed independently of the others. *Mapper* and *Reducer* agents can be running on the same node.

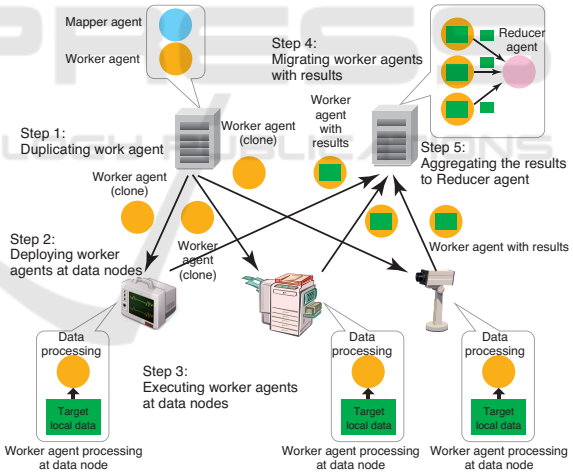

Figure 1: Mobile agent-based MapReduce processing.

5 DESIGN

The original MapReduce consisted of one *master* node and one or more *worker* nodes and Hadoop consisted of a *job tracker*, *task tracker*, *name*, and *data* nodes, where the first and third corresponded to the master node, and the second and fourth to the data nodes in the original MapReduce. Our MapReduce framework has a slightly different architecture from Google's MapReduce, which is quite different from

Hadoop's. The framework itself is a collection of three kinds of mobile agents, called *Mapper*, *Worker*, and *Reducer*, which should be deployed at appropriate nodes, called *mapper nodes*, *data nodes*, and *reducer nodes* respectively. Not only *Worker* agents but also *Mapper* and *Reducer* agents can dynamically be deployed at nodes according to the locations of target data and resources available to process them.

5.1 Enhancement of MapReduce Processing

Mobile agent technology can extend MapReduce processing with two mechanisms:

- Mobile agents can migrate between computers through their own itineraries. Our *Worker* agents can determine their next destinations according to their results from processing. For example, if *Worker* agents cannot find data that they want at their current data nodes, they can migrate to other nodes until they achieve their goals.
- Since *Mapper* and *Reducer* agents in our framework are implemented as mobile agents like *Worker* agents, this means that our framework allows *Mapper* and *Reducer* agents to migrate between computers. Also, we do not need to distinguish between *Mapper*, *Reducer*, and *Worker* agents. Therefore, *Worker* agents can become *Mapper* or *Reducer* agents. Therefore, our framework enables each worker task to work as MapReduce processing.
- *Mapper* and *Reducer* agents can define programs for their favorite data compression/uncompression inside them. Before leaving the source side, e.g., computers at nodes, they compress the data and after arriving at the destination, including clouds, they decompress it.

6 AGENT PROGRAMMING MODEL

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 are aimed at data processing on high-performance servers connected through wideband networks. Therefore, we cannot directly inherit a programming model from existing MapReduce processing.

In comparison with other MapReduce processing, including Hadoop, our framework explicitly divides the *map* operations into two parts in addition to a 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 has executed or the the network domains that contain the nodes. If they still want to define more complicated MapReduce processing, our framework is open to extend the *Mapper* and *Reducer* agents.
- *Application-specific Data Processing.* They define the three functions of reading data locally from nodes, processing 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* agents, the second in only *Worker* agents, and the third in *Worker* and *Reducer* agents, developers focus on these three parts independently of their mobile agent-level implementations. Each *Mapper* agent provides an itinerary to its *Worker* agent, where each itinerary is defined as a combination of the basic itinerary patterns. Figure 2 presents three patterns for the *map* phase.

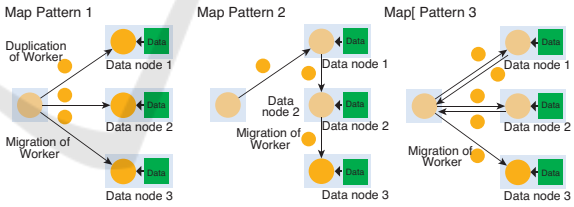

Figure 2: Basic itinerary patterns for map phase.

- The first is to instruct the *Worker* agent to duplicate itself and then instruct the duplicated *Worker* agents to migrate to and execute their data processing on the specified data nodes.
- The second is to instruct the *Worker* agent to migrate to and execute its data processing on one or more specified data nodes.
- The third is to instruct the *Worker* agent to go back and forth between each of the data nodes and the source node (or the reducer node) to execute its data processing on the nodes.

Figure 3 presents three patterns for the *reduce* phase.

- The first is to instruct one or more *Worker* agents to migrate to the reducer node and then pass their results to the *Reducer* agent at the reducer node.
- The second is to instruct the *Reducer* agent to migrate to one or more data nodes to receive the results of the *Worker* agents on the data nodes and then go back to the reducer node.
- The third is to instruct the *Reducer* agent to return and forth between each of the data nodes and the reducer node to receive the results of the *Worker* agents on the data nodes.

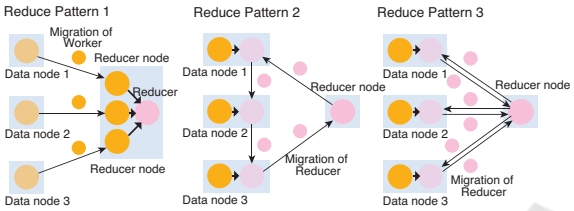

Figure 3: Basic itinerary patterns for reduce phase.

Our framework enables us to easily define application-specific *Mapper*, *Reducer*, and *Worker* agents as subclasses of three template classes that correspond to *Mapper*, *Reducer*, and *Worker*, with several libraries for key value stores (KVSs). When *Mapper* agent gives one or more *Worker* agents no information, we can directly define the agent from the template class for *Mapper*. It can create specified application-specific *Worker* agents according to one or more specified data and deploy them at the nodes. When *Reducer* agents support basic calculations, e.g., adding up, averaging, and discovering maximum or minimum values received from one or more *Worker* agents through KVSs according to the keys, we can directly define them as our built-in classes.

7 AGENT RUNTIME SYSTEM FOR MAPREDUCE PROCESSING

This section describes our mobile agent-based MapReduce framework. It consists of two layers, i.e., mobile agents and runtime systems. The former consists of agents corresponding to job tracker and *map* and *reduce* processing and the latter corresponds to task and data nodes. It was implemented with Java language and operated on the latter with the Java virtual machine (JVM). The current implementation was built on our original mobile agent platform, because existing mobile agent platforms are not optimized for data processing and need the developers for data processing to have knowledge about mobile agent processing.

Each runtime system runs on a computer and is responsible for executing *Mapper*, *Worker*, and *Reducer* agents at the computer and migrating agents to other computers through networks (Fig. 4). The system itself is designed independently of any application-specific data processing. Instead, agents running on it support MapReduce processing. Each runtime system is light so that it can be executed on embedded computers, including JVMs for embedded computers. The runtime system itself is portable because it is available with Java Standard Edition (JavaSE) version 6 or later. It can be executed on cloud computing environments, e.g., Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) with Java SE 6 or later.

7.1 Agent Duplication

Our framework is used to make one or more copies of *Worker* agents before agents are deployed at data nodes. The runtime system can store the states of each agent in heap space in addition to the codes of an agent into a bit-stream formed in Java's JAR file format, which can support digital signatures for authentication. The current framework basically uses the Java object serialization package for marshaling agents. The package does not support the capturing of stack frames of threads. Instead, when an agent is duplicated, the runtime system issues events to it to invoke their specified methods, which should be executed before the agent is duplicated, and it then suspends their active threads.

7.2 Agent Migration

Each runtime system 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 and agents through the connection. When an agent is transferred over a network, the runtime system transfers the agent in a bitstream like that in task duplication and transmits the bit-stream to the destination data nodes through TCP connections from the source node to the nodes. After they arrive at the nodes, they are resumed and activated from the marshalled agents and then their specified methods are invoked to acquire resources and they continue processing. To migrate tasks between nodes and the cloud, our agent migration can be tunnelled through hypertext transfer protocol (HTTP).

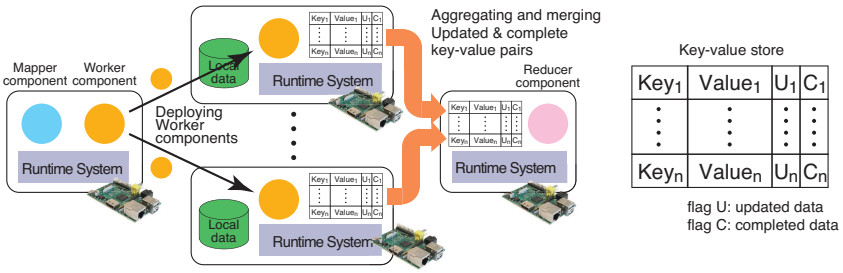

Figure 4: Runtime systems for mobile agent-based MapReduce processing.

7.3 Agent Execution

Each agent can have one or more activities, which are implemented by using the Java thread library. Furthermore, the runtime system maintains the lifecycles of agents. When the life-cycle state of an agent is changed, the runtime system issues certain events to the agent. The system can impose specified time constraints on all method invocations between agents to avoid being blocked forever. Each agent is provided with its own Java class load, so that its namespace is independent of other agents in each runtime system. The identifier of each agent is generated from information consisting of its runtime system's host address and port number, so that each agent has a unique identifier in the whole distributed system. Therefore, even when two agents are defined from different classes whose names are the same, the runtime system disallows agents from loading other agents's classes. To prevent agents from accessing the underlying system and other agents, the runtime system can control all agents under the protection of Java's security manager.

7.4 Key-value-store for Mobile Agents

MapReduce processing should be executed on each of the data nodes independently as much as possible to reduce the amount of data transmitted through networks. However, data may not be divided into independent pieces. Hadoop enables data nodes to exchange data with one another via Hadoop file system (HDFS) to solve this problem, because it is a distributed file system shared by all data nodes in a Hadoop cluster, like the Google file system (GFS).

Unlike other existing MapReduce implementations, including Hadoop, our framework does not have any file system, because nodes in sensor networks and ambient computing systems may lack enriched storage devices. Instead, it provides tree-structured KVSs, where each KVS maps an arbitrary string value and arbitrary byte array data and is maintained inside its agent, and provides directory servers to KVSs in agents. The root KVS merges the KVSs of agents into itself to support *reduce* processing. Each KVS in each data processing agent is implemented in the current implementation as a hashtable whose keys, given as pairs of arbitrary string values are byte array data and it is carried with its agent between nodes. It supports a built-in hash-join to merge more than two KVSs carried by *Worker* agents into a single KVS.

Whenever an agent corresponding to a *job tracker* in Hadoop starts MapReduce processing, it creates one or more processing agents and a root KVS and assigns the references of the KVSs of the newly created processing agents to the root KVS. Each directory server can run in the external system from agent runtime systems and agents, or inside an agent corresponding to a *job tracker*. It tracks the current locations of mobile agents so that it can enable an agent to access KVSs maintained in other agents, which may move to other destinations, with the identifiers of the moving agents. To support *reduce* processing, the root KVS merges the KVSs of agents into itself. Each KVS in each data processing agent is implemented as a hashtable in the current implementation, whose keys, given as pairs of arbitrary string values and values, are byte array data, and is carried with its agent between nodes. However, a KVS in a *job tracker* agent is also implemented as a hashtable whose keys are given as pairs of identifiers of the agents that its agent creates, and the values are references to them.

7.5 Job Scheduling

MapReduce can be treated as batch processing over distributed systems. If there is more than one *Mapper* agent in our framework, they can be running independently. Therefore, we introduced a *schedule* agent between running *Mapper* agents, if we needed to manage the whole system. The agent is responsible for controlling *Mapper* agents and monitoring *Reducer* agents. When it also detects the completeness of *Reducer* agents, it can explicitly send a *start* message to one or more *Mapper* agents to instruct them to start processing.

7.6 Fault-tolerance

One of the most important advantages of MapReduce processing is to conceal the results of difficulties from distributed systems. Our framework provides several mechanisms for dependability. The job manager in Hadoop is responsible for supporting fault tolerances against crash failures in data nodes. The manager detects failures in data nodes because each *task tracker* running on a data node sends heartbeat messages to 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 framework assumes that data are maintained in one data node so that it has a different policy for fault tolerances. If a data node is stopped or disconnected, it needs to exclude such a node. Our framework 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, the *system manager* agent informs *Mapper agents* to omit the crashed node from the list of target data nodes, before the *Mapper* agent dispatches *Worker* agents.
- If a data node has crashed, the *system manager* agent informs the *Reducer* agent to omit agents returned from nodes from the agent's waiting list, after *Worker* agents have been deployed at the target node. Even when the crashed node can be restarted or it continues to work, the *Reducer* agent does not wait for any agents from the node.

The *system manager* agent can explicitly make clones of these agents at other nodes because they are still mobile agents that can mask failures in nodes that run *Mapper* and *Reducer* agents.[2] The current implementation has no fault-tolerant mechanisms for failures while *Worker* agents are deployed and running because our MapReduce processing is not heavy. We should restart processing.

7.7 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.

[2]The current implementation does not support consistency between original agents and their clones.

8 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 (100Mbit) 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 was 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. Figure5 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 IoT.

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

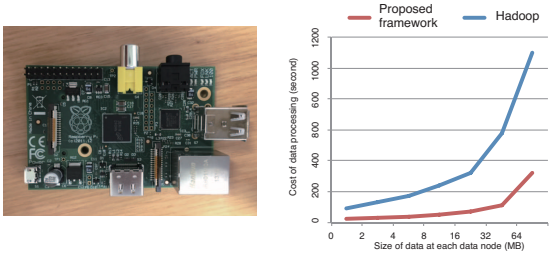

Figure 5: Performance evaluation of propose framework and Hadoop.

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.

9 CONCLUSION

We presented a mobile agent-based MapReduce framework available on IoT. It was designed for analyzing data generated at IoT. It could distribute programs for data processing to nodes at the edges of networks 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 IoT, 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

Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16. ACM.

Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. (2010). Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296.

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10. USENIX Association.

Dou, A., Kalogeraki, V., Gunopulos, D., Mielikainen, T., and Tuulos, V. H. (2010). Misco: A mapreduce framework for mobile systems. In *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '10, pages 32:1–32:8. ACM.

Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. (2010). Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818. ACM.

Elespuru, P. R., Shakya, S., and Mishra, S. (2009). Mapreduce system over heterogeneous mobile devices. In *Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, SEUS '09, pages 168–179, Berlin, Heidelberg. Springer-Verlag.

Jiang, W., Ravi, V. T., and Agrawal, G. (2010). A map-reduce system with an alternate api for multi-core environments. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 84–93. IEEE Computer Society.

Satoh, I. (2013). Adaptive agents for cyber-physical systems. In *Proceedings of the 5th International Conference on Agents and Artificial Intelligence, Volumn II*, pages 257–262.

Sehrish, S., Mackey, G., Wang, J., and Bent, J. (2010). Mrap: A novel mapreduce-based framework to support hpc analytics applications with access patterns. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 107–118. ACM.

Talbot, J., Yoo, R. M., and Kozyrakis, C. (2011). Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 9–16. ACM.