System Behavior Analysis of a MapReduce Application by Probing with Non-intrusive Method

*1Yunhee Kang
1*Division of Information and Communication, Baekseok University, Cheonan, Korea 330-704, yhkang@bu.ac.kr*

Abstract

When a MapReduce application is running, programmers try to analyze system behavior affected by its characteristics to better understand its internal operation and elicit its barrier in a MapReduce framework. But it is difficult to do this work because the internal operation of MapReduce application is handled by its framework. This paper presents a software component for monitoring the behavior of a MapReduce application running on the Twister and the status of its environment. It is used to identify the characteristics of a MapReduce application caused by hiding the behavior in details. The software component for probing behavior of a MapReduce application running on Twister plays a role as a subscriber has an advantage that it has no need of which is modified code.

Keywords: *Software architecture, MapReduce programming model, Behavior monitoring, System Behavior Analysis, Non-intrusive Method*

1. Introduction

The growth of the volume of data requires the amount of computing resources including much larger scales than a traditional computer cluster. Computing resources have been diverse from supercomputers to on-demand resources in cloud computing. This also makes writing parallel programs inevitable. But writing a distributed and parallel application using MPI is not easy caused of low-level primitive like synchronization and inter-process communication. The demanding requirements have led to the development of a new programming model and its implementations like MapReduce [1, 2]. The MapReduce paradigm offers a simple API for computation to a programmer. Performance of a MapReduce application is related with its characteristics.

But it is difficult to understand the behavior of a MapReduce application caused by hiding behavior in details associated with status (when running it) within its runtime. In addition the behavior of a MapReduce application is also depending on its framework when it is running. Especially slow tasks are a major performance bottleneck in MapReduce systems. The characteristics for execution of a MapReduce application also is strongly related with disk IO, CPU cycle and network bandwidth.

This paper is focused on how to analyze a MapReduce application running on Twister. We adopt a non-intrusive gray-box perspective of a MapReduce runtime for analyzing behavior of a MapReduce application dynamically. It is based on functional specification and architectural view whereas not on source code or binaries which makes it invasive. It is good for analyzing the MapReduce application and can be used to identify the bottleneck of a MapReduce application running on the given Twister configuration. The main contributions we make with this work are summarized in the following:

- The method is good for identifying barrier in a MapReduce application
- The result of experiment provides to select a type of resources for running a MapReduce application

Received: Oct. 19, 2014, Revised: Dec. 05, 2014, Accepted: Jan. 06, 2015

The rest of paper is organized as follows. Section 2 describes the related works including an overview of MapReduce programming model and ProtocolBuffer as a format of message representation. Section 3 describes an overall architecture for extended *Twister* that is used for handling an internal status of MapReduce application. Section 4 describes the experimental environment and result. Conclusion follows in section 5.

2. Related Works

2.1 MapRedue

MapReduce is an emerging programming model for a data-intensive application proposed by Google, has been widely used in industry and academia. It borrows ideas from functional programming, where a programmer defines map and reduce tasks to process a large set of distributed data. Traditional parallel applications are based on a runtime library that has some features of communication and synchronization [11]. The features provided by a runtime library are low-level primitive ones. However, a MapReduce programmer is able to focus on the problem that needs to be solved, since only the map and reduce functions need to be implemented. Then, the framework takes care of the burden that the programmer would have to deal with lower-level mechanisms to control the data flow [2, 4]. MapReduce programs are automatically parallelized and executed on a cluster. A Mapreduce consists of three phases. First comes a map phase that takes input records and produces output (key,value) pairs. This is followed by a shuffle phase that groups the (key,value) pairs by common values of the key, and finally a reduce phase that takes all pairs for a given key and produces a new value for the same key and this is the output of the MaprReduce. It consists of two functions: Map and Reduce. These two functions have the following signatures:

$$Map :: (key1, value1) \rightarrow list((key2, value2))$$
$$Reduce :: (key2, list(value2)) \rightarrow list((key3, value3))$$

The map function processes the input pairs (key1, value1) returning some other intermediary pairs (key2, value2). Then, the intermediary pairs are grouped together according to their key values. After that, each group will be processed by the reduce function which will output some new pairs of the form (key3, value3).

2.2 protocolBuffer

ProtocolBuffer allows to define simple data structures in a special definition language, then compile them to produce classes to represent those structures in the specific language [4]. Some classes of protocolBuffer come up with optimized code to parse and serialize a message in a compact format. Thus it is designed to minimize network traffic, and deserialization to maximize performance. Thus it is designed to minimize network traffic, and deserialization to maximize performance. Hence it is used when a non-intrusive monitor is to store data and to serialize message aggregated from messaging infrastructure NaradaBrokering.

3. Extended System Architecture

3.1 Overall Structure

Twister, a kind of MapReduce system, contains three main software components: Twister daemon and worker module, the MapReduce library, and MapReduce application. It provides a lightweight alternative to parallel programming and is becoming popular in variety of domains. The NaradaBrokering is application-independent and incorporates several services to solve network-induced problems as streams traverse domains during disseminations [3]. The system provides a way that is easy to use, while delivering consistent and predictable performance. It can be used as a message-oriented middleware or as a notification framework. Communication within NaradaBrokering is asynchronous, and the substrate places no constraints on the size, rate, or scope of the interactions encapsulated within events or the number of entities present in the system.

This section describes a prototype implementation of a system monitor for handling status of a MapReduce Application. The prototype of the system monitor is implemented by Naradabroker which is a toolkit for delivering a message for event. A monitor channel as a consumer is an entity that is interested in consuming messages. Every monitor channel needs to implement the cgl.narada.service.client. NBEventListener interface. This interface contains the onEvent (NBEvent nbEvent) method that is invoked by the substrate upon receipt of a message which should be routed to that monitor channel.

Figure 1 illustrates the proposed Twister architecture having modules, Probing Agent (PA) and Trace Collector (TC), for aggregating messages representing status in protocolBuf format. PA is hooking of messages via a pub/sub broker network. We can use it when a TC is to store data and to serialize message from PA.

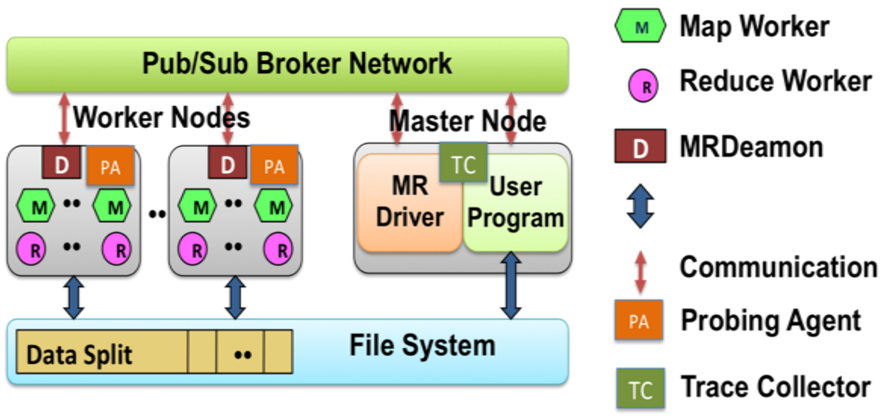

Figure 1. Extended Architecture of Twister

As shown in Figure 1, a distributed messaging infrastructure is used to perform communicating and transferring data via a pub/sub broker network [8]. During the initialization of the runtime, Twister starts a daemon process in each worker node, which then establishes a connection with the broker network to receive commands and data. The daemon is responsible for managing map/reduce tasks assigned to it, maintaining a worker pool to execute map and reduce tasks, notifying status, and finally responding to control events.

As in other MapReduce runtimes, a master worker (MRDriver) controls the other workers according to instructions given by the user program. The worker provides the storage and computing to run the Map and Reduce tasks. Classically, the execution is controlled by the daemon that distributes and schedules the data blocks, launches MapReduce computations, monitors the progress of tasks execution and collects final results.

The client side driver provides an interface of the programming API to the user and converts these Twister API calls to control commands and input data messages sent to the daemons running on worker nodes via the broker network. Twister uses a pub/sub broker network to handle four types of communication needs; (i) send and receive control events, (ii) send data from the client-side MR driver to the Twister MR daemons, (iii) transfer intermediate data between map and reduce tasks, and (iv) send the outputs of the reduce tasks back to the client side MR driver.

3.2 Topic and its message definition

The twister application starts to run when receiving a message named NEW_JOB_REQUEST and ends the application when receiving a message MAP_ITERATIONS_OVER. A construct of a class TwisterDriver send a message ALIVE to notify that a Twister application is started. To clean up the

Twister driver, a method terminate () is called. Then in the method terminate, the Twister application gives a notification named TERMINATE. The system prober distinguishes the type of message and measures its the elapse time when receiving those messages.

Fig. 2 shows an example of a data flow for the proposed method. In this paper we only cover how to collect streams of messages having status data about topics, FT_TOPIC and MR_TOPIC. A message named ALIVE is delivered to a subscriber who registers to FT_TOPIC. The messages such as NEW_JOB_REQUEST, MAPPER_REQUEST, MAP_ITERATIONS_OVER are associated with a topic named MR_TOPIC in Figure 2.

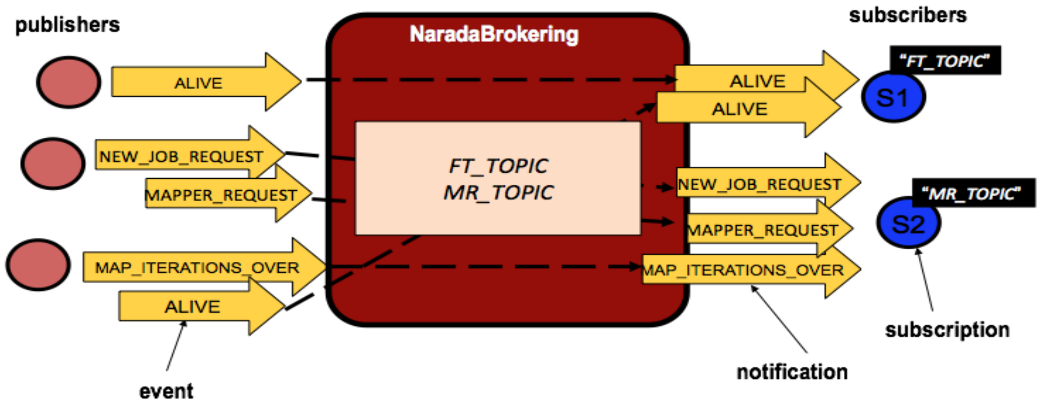

Figure 2. Message flow in pub/sub messaging

3.3 Message format

Figure 3 shows the defined message types written in protocolBuffer for delivering a PA to a TC. The message `SendTwister` consists of a tuple with three values `jobid, key and value`. Jobid represents a name of Twister based MapReduce application. Key represents an internal status operated by Twister runtime. Value represents an elapse time to complete a step related with the status described as the value of key. The message `RcvTwister` is used for replying one associated with `SendTwister`. Those message is operated with services such as `ElapseTwister` by RPC(remote procedure call).

```
message SndTwister {
    required string jobid = 1;
    required int32 key = 2;
    required int64 value = 3;
}
message  RcvTwister {
    required int64 value = 1;
}
message NoInput {
    optional string name = 1;
}
service TwisterService {
    rpc ElapseTwister(SndTwister)
      returns (RcvTwister);
    rpc ResetValue(SndTwister)
      returns (RcvTwister);
    rpc ResetAllValue(NoInput)
      returns (RcvTwister);
}
```

Figure 3. The message and service for collecting the status of a MapReduce application

4. Experiments

4.1 Experiment Environment

To evaluate operations of the tool and show probing mechanism, we conducted experiments about running two MapReduce applications. The prototype for probing internal status in a MapReduce framework, twister, from a topic defined in the NaradaBrokering that transfers all messages and traces the messages uses the following systems:

- Twister 0.8
- NaradaBrokering 4.2.2
- Linux 2.6.x running on the cluster on Emulab

That is implemented in Java and uses protocolBuf for transferring events to TC from PA. In this experiment, two kinds of application are used; k-means and word-count. Both are typical examples in the area of data mining.

1) The k-means application gets the input dataset and generates k groups of data. Each group is a cluster where every element is nearer to each other than to the elements of other groups. This application divides the input dataset into k groups first. Then k centers are computed in k group independently. With k centers, it makes new k groups whose members are nearer to each center than to other centers. This repetition would terminate until no change of k centers.
2) The word-count application gets the input document and counts the frequency of all words. Like the k-means application, this application can divide the input document and treat each partition independently. However, it doesn’t need repetition.

4.2 Analysis

In this section, we show the experimental results and analyze the results. Our probing program measures the processing time of the messages which are transferred between components of the application via NaradaBrokering. This section also shows the variation of the processing times in all tasks. The variation can be used for detecting the obstacle of the performance.

Figure 4 shows the message flow diagram among the components of k-means and word-count. The dashed edge means the transfer of a message and the shadowed arcs represent the iteration. The shadowed arcs are optional since word-count doesn’t require the repetition of Mapper and Reducer.

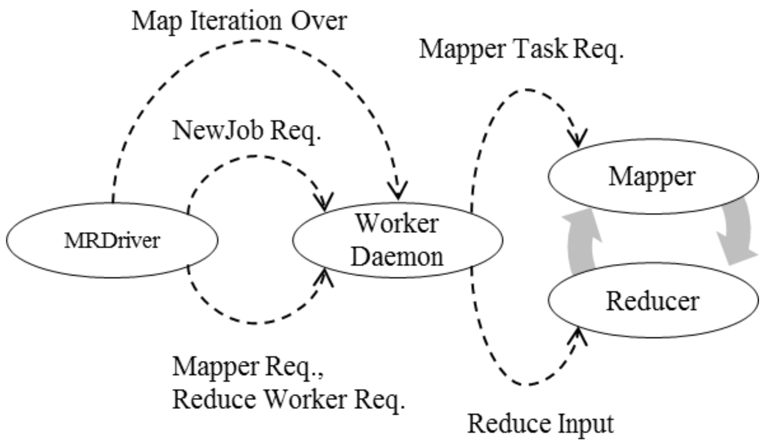

Figure 4. The message flow diagram for *k*-means

In the Figure 5, we can see the variation of MAPPER_REQUEST is larger than others. For the improvement of the efficiency of this application, the processing should be revised. We observed the variation of MAPPER_REQUEST is larger than others. But there is no special stagger when running MapReduce application.

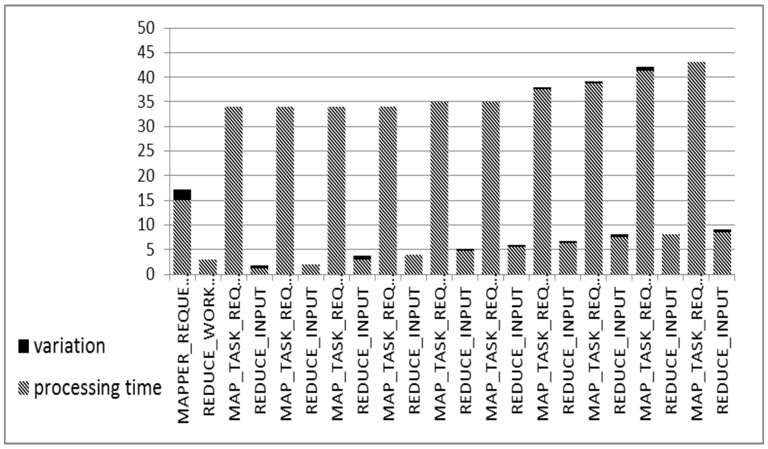

Figure 5. Processing time of messages and Variation for *k*-means

The proposed method is used for detecting the obstacle of the performance. As shown in Figure 6there is high variation caused by stagger. We need to redesign the processing of MAP_TASK_REQUEST and REDUCE_INPUT messages. We found that it is important to determine a configuration of a cluster system in order to run a MapReduce application efficiently.

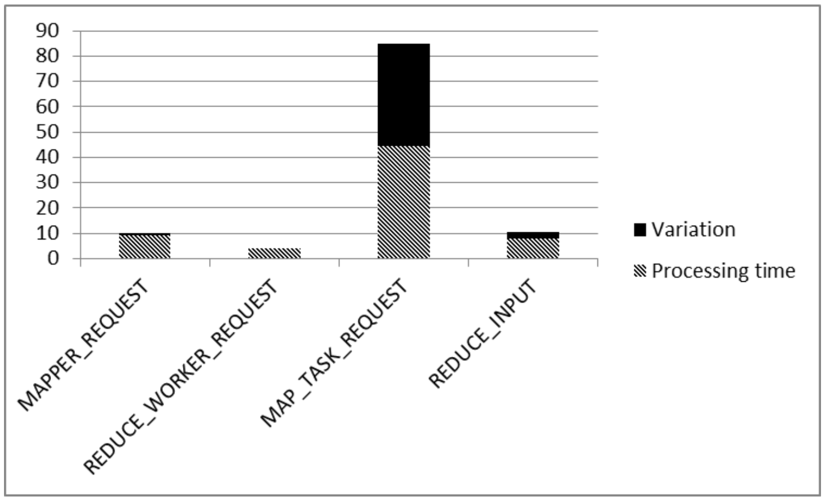

Figure 6. Processing time of messages and Variation for word-count

5. Conclusion

We proposed a probing scheme for handling behavior of MapReduce application easily and implemented the system to aggregate its internal behavior. To implement the probing system, Twister is extended to probe messages by non-intrusive manner. We considered data formats to transfer between subsystems in the system architecture. The feature for probing in Twister is implemented as a message subscriber in pub/sub message. The main advantage of this probing method is to develop the feature without any modification of MapReduce application. It can be used to identify the bottleneck of a MapReduce application running on the resource, given a Twister configuration. System behavior analysis is used for measure the ratio between map and reduce tasks in a MapReduce application. In case of the experiment, the overall performance of data intensive application is strongly affected by the throughput of the messaging middleware. The result of the experiment can be used to identify the bottleneck of a MapReduce application running on the Linux cluster. It will support performance analysis of inner phases in a MapReduce. We will consider a fine-grain data analysis to select computing resources for running a MapReduce application properly.

References

[1] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC) June 21-25, Chicago, Illinois, USA, 2010, pp. 810–818.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, Vol. 51, No. 1, pp. 107–113, Jan. 2008.

[3] J. Dean and S. Ghemawat, "MapReduce: A Flexible Data Processing Tool," *Communications of the ACM*, Vol. 53, No. 1, pp. 72–77, Jan. 2010.

[4] K. Morton, A. Friesen, M. Balazinska, and D. Grossman, "Estimating the Progress of MapReduce Pipelines," in Proc. of the IEEE 26th International Conference on Data Engineering (ICDE), March 1–6, Long Beach, CA, USA, 2010, pp. 681–684.

[5] S. Pallickara and G. Fox, "NaradaBrokering: a distributed middleware framework and architecture for enabling durable peer-to-peer grids," in Proc. of the ACM/IFIP/USENIX International Conference on Middleware (Middleware), Vol. 2672, June 16–20, Rio de Janeiro, Brazil, 2003, pp. 41–61.

[6] Google, ProtocolBuffer. [Online]. Available: http://code.google.com/p/protobuf/

[7] MPI (Message Passing Interface). [Online]. Available: http://www-unix.mcs.anl.gov/mpi/

[8] PVM (Parallel Virtual Machine). [Online]. Available: http://www.csm.ornl.gov/pvm

[9] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernstm "HaLoop: efficient iterative data processing on large clusters," *Proc. VLDB Endow.* Vol. 3, Iss. 1–2, pp. 285–296, Sept. 2010.

[10] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for Large-scale graph algorithms," *Parallel Computing*, Vol. 37, No. 9, pp. 610–632, Sept. 2011.

[11] H. Mohamed and S. Marchand-Maillet, "Enhancing MapReduce Using MPI and an Optimized Data Exchange Policy," in Proc. of the 41st International Conference on Parallel Processing Workshops (ICPPW), Sept. 10-13, Pittsburgh, Pennsylvania, USA, 2012, pp. 11–18.