Data I/O provision for Spark applications in a Mesos cluster

Nam H. Do[§*], Tien Van Do[§*‖], Xuan Thi Tran[*], Lóránt Farkas[¶], Csaba Rotter[¶]
[§] Division of Knowledge and System Engineering for ICT (KSE-ICT)
Faculty of Information Technology,
Ton Duc Thang University, HCM City, Vietnam
[*]Analysis, Design and Development of ICT systems (AddICT) Laboratory
Budapest University of Technology and Economics
Magyar tudósok körút 2, Budapest, Hungary
[¶]Nokia Networks
Köztelek utca 6, Budapest, Hungary
[‖] (Corresponding author)

***Abstract*—At present there is a crucial need to take into account the I/O capabilities of commodity servers of clusters in production environments. In order to support such a demand in production environments, we propose a solution that can be integrated into Mesos to control I/O data throughput for Spark applications. The proposed solution takes into account the I/O capability to provide resource mapping, control and enforcement functionality for applications that process HDFS data. The aim is to minimize I/O contention situations that lead to the degradation of quality of service offered to Spark applications and clients.**

I. Introduction

The development of computing frameworks such as YARN [1] and Mesos [2] has been motivated by the need of sharing a cluster of commodity servers among different applications. Computing frameworks include appropriate components that run in commodity servers to provide the management and the execution of jobs (submitted by applications) on a specific cluster. These frameworks provide interfaces that hide the complexity of reserving resources and the allocation of resources in a specific cluster from applications. Therefore, for a certain degree computing frameworks simplify the programming of applications for resource reservations from clusters.

Normally, the number of CPUs, memory (in MB), and disk space (in MB) are quantities in resource reservation requests. In addition, Mesos can handle resource reservations in the term of the number of CPUs, memory (in MB), disk quota (in MB), port range as well as the limitation of outgoing traffic [2]. Applications (clients) can submit jobs (consisting of a number of tasks) with a specific resource demand. The resource management functionality decides about the allocation of resources to clients by a simple mapping from resource quantities to the real capabilities of commodity servers. Tasks (and jobs) are scheduled and executed within computing components that are often termed containers [1], [2].

However, the I/O capability of commodity servers is not fully considered in present computing frameworks. When containers compete for a resource in the hardware level, which often happens in production environments, applications may suffer from a disk I/O performance degradation. It is worth emphasizing that the competition in the hardware level is hidden from programmers (and therefore from applications) and is not solved within computing frameworks.

At present the provision of a data rate for real-time big data applications may play a key factor to achieve benefits for operators in telecommunication networks. In such environments there are many sources for real-time unstructured big data which can be analyzed for customer retention, network optimization, network planning, customer acquisition, fraud management [3]–[6]. In order to support such a demand in production environments, we propose a solution that takes into account the I/O capability of computing clusters to provide resource mapping, control and enforcement functionality for applications. The aim is to minimize contention situations that lead to the degradation of quality of service offered to applications and clients. In other words, the main purpose is to create environments where the competition for resources is fully controlled by administrators. In this paper, we present our proposal through the integration with the Mesos computing framework with Spark [7] and Hadoop Distributed FileSystem (HDFS) [8].

The rest of this paper is organized as follows. In Section II, we present a technical background of deployed frameworks and discuss about the data I/O problem in a shared computing environment. The proposal of a general framework is presented in Section III. A proof of concept for our proposal is demonstrated in Section IV. Finally, Section V concludes our paper.

II. Motivation

In this section, we present a scenario where I/O contentions cause a problem for Spark applications in a shared cluster managed by Mesos. To ease the comprehension, we first provide an overview of related software frameworks that are involved.

A. Mesos

Mesos is a resource management framework in a shared cluster. It is capable to cooperate and coordinate Hadoop, Spark, Storm, etc [9]. The Mesos framework consists of a Mesos master and a set of Mesos slaves that are native (C++) processes. The Mesos master is responsible to manage the cluster slaves and the resource reservations for other frameworks (see the illustration in Fig. 1(a)). The Mesos master sends resource offers to application frameworks and coordinates the resource allocations for the shared cluster. The scheduler of a specific framework interacts with the Mesos master to negotiate resource reservations. If a resource offer is accepted, an executor program is launched on Mesos slave nodes to execute tasks. Information a scheduler sends to the Mesos master includes

- life cycle management (Register, Reregister, Unregister)
- resource allocation (Request, Decline, Accept,...),
- task management (Launch, Kill, Acknowledgment,...).

The master can send a framework a "callback" such as Registered (for life cycle management), resourceOffers (resource allocation), statusUpdate (for task management), etc. Mesos uses the Dominant Resource Fairness (DRF) scheduling algorithm to allocate resources for requests [10]. At present, the DRF algorithm only takes into account Memory and CPUs.

Mesos slaves are responsible to launch and kill executors that host tasks of a job from a specific application. In Mesos, resources of machines can be described by slave resources and slave attributes. Slave resources include CPU, memory, etc. Attributes are simple key-value pairs that identify a slave node and can be included in the slave configuration as an optional flag. It is worth emphasizing that any consumable resource can be defined in Mesos slaves using the "–resource" flag. Currently Mesos handles a resource model that can contain quantities such as the number of CPUs, an amount of memory (in MB), disk quota (in MB), outbound traffic rate (in MB/s), and a port range as user demands.

Mesos also provides containerizer APIs to support plug-ins for different containerization and resource isolation implementations such as using Mesos Containerizer, Docker Containerizer, or customized containerizer. The built-in Mesos Containerizer normally launches a container that isolates an executor and its inside tasks using CGroups. Docker can be applied in Mesos to package applications in light-weight container. Docker container supports resource isolation using CGroups, LXC, OpenVZ and kernel namespaces. To control resources available for a Docker container, Docker enables resource limitations in terms of memory, swap space, "cpu shares" (CPU times for a container), "cpuset" (which CPU container can use), "cpu pinning" (the number of CPU for container), etc. From Mesos version 0.20.0, users now can run a Docker image (essentially a snapshot of a Docker container) as a task, or as an executor. Customized containerizations can be done by an executable external program implemented as an external containerizer plugin.
Furthermore, the availability of Mesos modules provides a simple way to extend Mesos functionality and to integrate with external tools and systems.

B. HDFS

In a shared cluster, Hadoop Distributed File System (HDFS) [8] offers big data services for applications. The master NameNode (JVM process) manages the file metadata and DataNodes (JVM processes) keep data in form of data blocks/chunks, as illustrated in Fig. 1(b). To read/write a file, an HDFS client initiates a request to the NameNode for metadata, then directly opens TCP sessions to certain DataNodes to stream data. Note that HDFS can either run alongside Mesos as a standalone distributed filesystem, or a resource allocation to the components of Hadoop Distributed File System (HDFS) can be managed by Mesos as well.

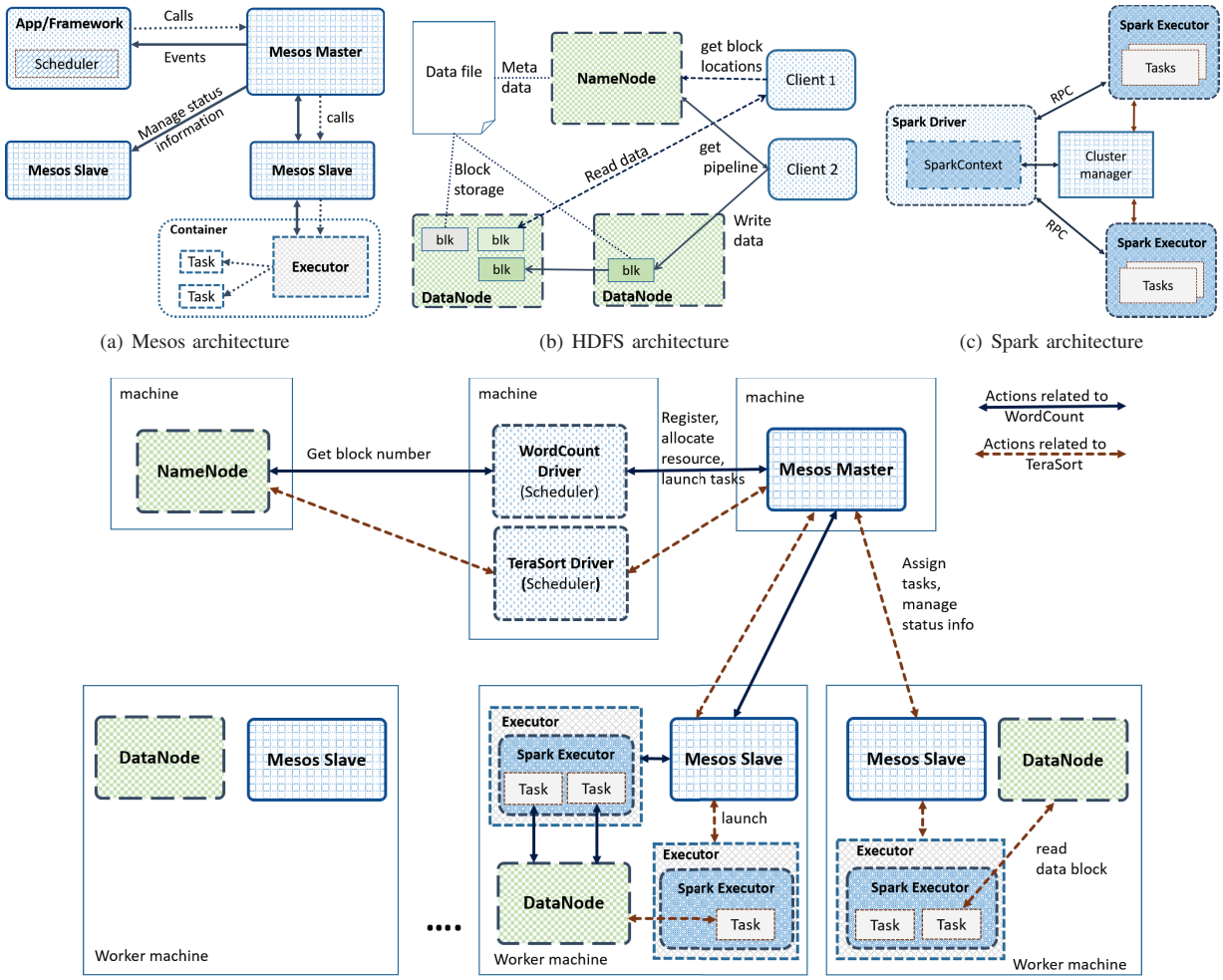
C. Spark

In Apache Spark [7], a high-performance computational engine is implemented with the abstraction of Resilient Distributed Datasets (RDD) and Directed Acyclic Graph (DAG) execution algorithm to support the processing of big data. RDD is simply a partitioned collection of elements that can be operated on in parallel [11]. Spark can create RDDs from existing collections, local filesystem, Hadoop Distributed FileSystem (HDFS), or any data storages supported by Hadoop API. Currently, Spark can run as a standalone system using its built-in cluster manager, or can be managed by YARN or Mesos. For each cluster manager, Spark provides particular interfaces for scheduler implementation and for launching executors.

Each Spark application consists of a main program called driver and an executor program (see Fig. 1(c)). A driver program, which runs in a JVM process, serves as the main entry point of a SparkContext object. The main functional block components of SparkContext are as follows.

- An RDD graph is a direct acyclic graph of task stages to be performed on the cluster.
- A DAG Scheduler builds stages of each job by breaking the RDD graph at shuffle boundaries, computes a DAG of stages with stage-oriented scheduling. Then stages are submitted as TaskSets to an underlying Task Scheduler implementation. In addition, DAG scheduler also determines the preferred locations to run each task on, based on the current cache status, and passes these to the underlying Task Scheduler.
- A Task Scheduler receives TaskSets as the input, negotiates resources with the cluster manager, and determines where tasks should be executed and which resources a task can consume.

Each Spark executor is a process (a standalone JVM or embedded in another JVM process), which is responsible for running tasks and storing RDD data. An application can contain a set of executor processes spawned on worker nodes. Each executor can run multiple tasks in a task thread pool. It also serves and stores data (RDD, shuffle data, etc) for use in further jobs through the Block manager and the cache service.



(a) Mesos architecture (b) HDFS architecture (c) Spark architecture

(d) An example scenario of Spark deployment on Mesos

Fig. 1. Spark on Mesos cluster

Spark enables a configuration for the number of CPUs per executor and per task, as well as a memory amount as resource demands from an application. Those quantities are used in the negotiation process with the Mesos master. Spark application frameworks can be launched with two modes in Mesos: fine-grained and coarse-grained. In the default fine-grained mode, the Mesos master and a Mesos slave are aware of the existence of the Spark tasks run on the slave, thus the Mesos slave can allocate resource (e.g. the number of CPUs required for the tasks) appropriately for the executor, which can provide an efficient resource utilization. On the contrary, in the coarse-grained mode only one long-running Mesos task is launched on each Mesos machine and reserves the allocated resources for the entire duration of the application. This Mesos task then dynamically schedules Spark tasks. Thus, the startup overhead is much lower than in the fine-grained mode, hence it is more suitable for applications with low-latency requirements like interactive queries or serving web requests. The number of executors can be scaled up and down according to the actual workload using the dynamic resource allocation mechanism.

D. Disk I/O contention of Spark applications

To give an example regarding a disk I/O resource contention, we establish a shared cluster that consists of servers with Intel® Core™ i5-4670 CPU @ 3.40GHz, 16GB Kingston HyperX Black DDR3 1600MHz RAM, Asus H87-PLUS Motherborad with Intel® H87 chipset and SATA 6Gb/s Ports, and 1GB WD Black WD1003FZEX 7200 RPM hard drives with SATA 6Gb/s interface and data transfer rate up to 150 MB/s. Ubuntu Server 14.04.3 LTS 64 bit, Hadoop 2.7.1, Mesos 0.24.0, and Spark 1.5.0 are used in our cluster. The configuration with a Mesos master node, a Hadoop NameNode, and a set of machine nodes hosting Mesos slaves and a Hadoop DataNode is illustrated in Fig. 1(d). It is worth mentioning that in each worker machine, HDFS data blocks are stored in a separate hard drive.

We runs Spark TeraSort and WordCount in the default fine-grained mode to process HDFS data with the following settings:

- WordCount counts the occurrence number of each word in 3GB HDFS data with block size of 256MB,
- TeraSort sorts 3 million records stored in HDFS with block size of 512MB,
- By using Mesos roles, WordCount and TeraSort were configured to launch their tasks in a dedicated machine, which stores the needed HDFS data blocks,
- All executors are configured to run up to three tasks in parallel.

The interactions in the cluster are described as follows:

- When each application (TeraSort or WordCount) is submitted, its driver is launched in a separate JVM process.
- The driver creates RDDs from its inputs, builds stages of tasks, determines the preferred nodes where tasks can run (based on data locality constraint), registers with Mesos master, and is ready to negotiate resources for the application's needs.
- The Mesos master sends a resource offer in the callback to the driver,
- The driver accepts an offer and requests the launch of an executor with tasks on the appropriate slave.
- The master asks the slave to launch the executor with its tasks on behalf of the driver.
- The slave spawns a Mesos executor as a stadalone JVM process using MesosExecutorBackend pluggin. The Mesos executor in turn launches a Spark Executor, which registers itself with the driver through a private RPC (Remote Process Call).
- After being launched, the Spark Executor uses a task thread pool to launch the framework's tasks.
- Each task reads a data block from certain DataNode, does computation, and saves results.

In a shared cluster, a disk I/O contention can happen on any worker node where different executors (or even several tasks of an executor) concurrently access the same disk. When Wordcount and Terasort simultaneously run in the cluster (see Fig. 1(d)), the executor of Wordcount launches two tasks to be run in parallel and the Terasort executor runs one task on the same worker machine.

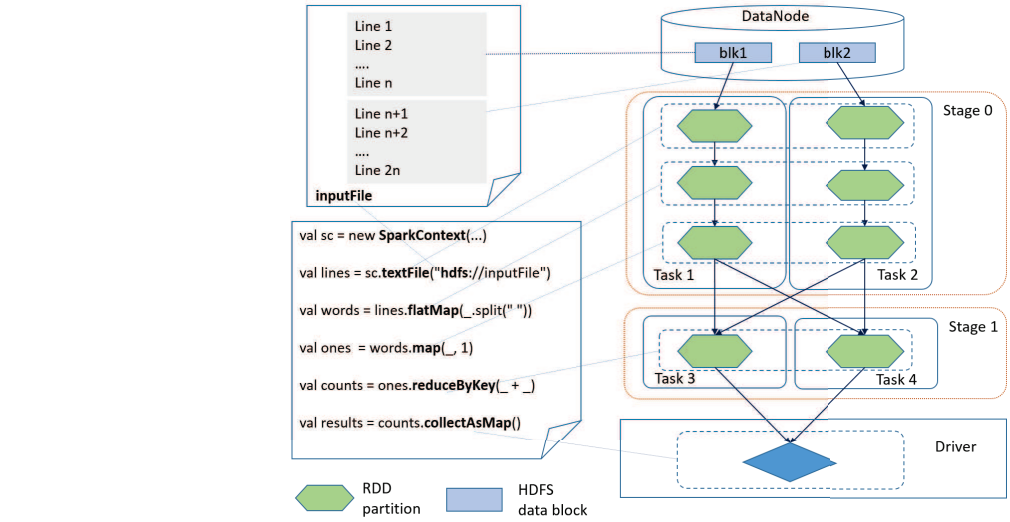
Concurrent activities of reading/writing data blocks by different applications or frameworks on the same DataNode and other intermediate results in shuffle phase and RDD data of each application can be spilled to the local disk can have a bad impact in uncontrolled environments. The execution flow of a particular Spark Wordcount example in a Mesos cluster to process an HDFS file with two data blocks is illustrated in Fig. 2(a). Spark implicitly constructs the DAG graph of RDD operations with all necessary RDDs from the input HDFS text file and operations in the program. When the driver is launched, it converts the logical DAG of operations into the stages of tasks to be executed. With 2-block input file, two 2-task stages are created (see Fig. 2(a)). Thus, the application is executed in two sequential stages. In each stage, resources for two simultaneous tasks are needed.

In our experiment each executor is configured to require 1 CPU and 4GB memory, and each task requires 1 CPU. That means, a resource offer that advertises at least 2 CPUs and 4GB memory may be accepted by an application. Spark also considers the data locality as a constraint in scheduling. The master starts offering resources to the driver. In this case, the master sends an offer that advertised slave X's resource model that looks like {cpus(r1):4; cpus(*):4; mem(*):14895; disk(*):213239; ports(*):[31000-32000]}. With this resource offer, an application with the role r1 can use up to 8 CPUs, while applications with another role can only use 4 CPUs. Since it satisfies resource demands as well as data locality constraint, the driver accepts the offer and launches Task 1 and 2 of stage 0 with introducing the resource demands regarding to memory and CPUs, i.e 4GB memory and 1 CPU for a Spark executor and 2 CPUs for its two tasks (action (1)). When the slave is asked to launch a Mesos executor with two assigned tasks, it also launches a container (action (2)) that provides a resource isolation for the executor (action (3)). In turn, a Spark executor is spawned and actually starts Task 1 and Task 2. In this example only Task 1 and Task 2 need the access of HDFS blocks (action (4)). Moreover, the executor created by "MesosExecutorBackend" can be reused to launch Task 3 and Task 4 in case of enough resources.

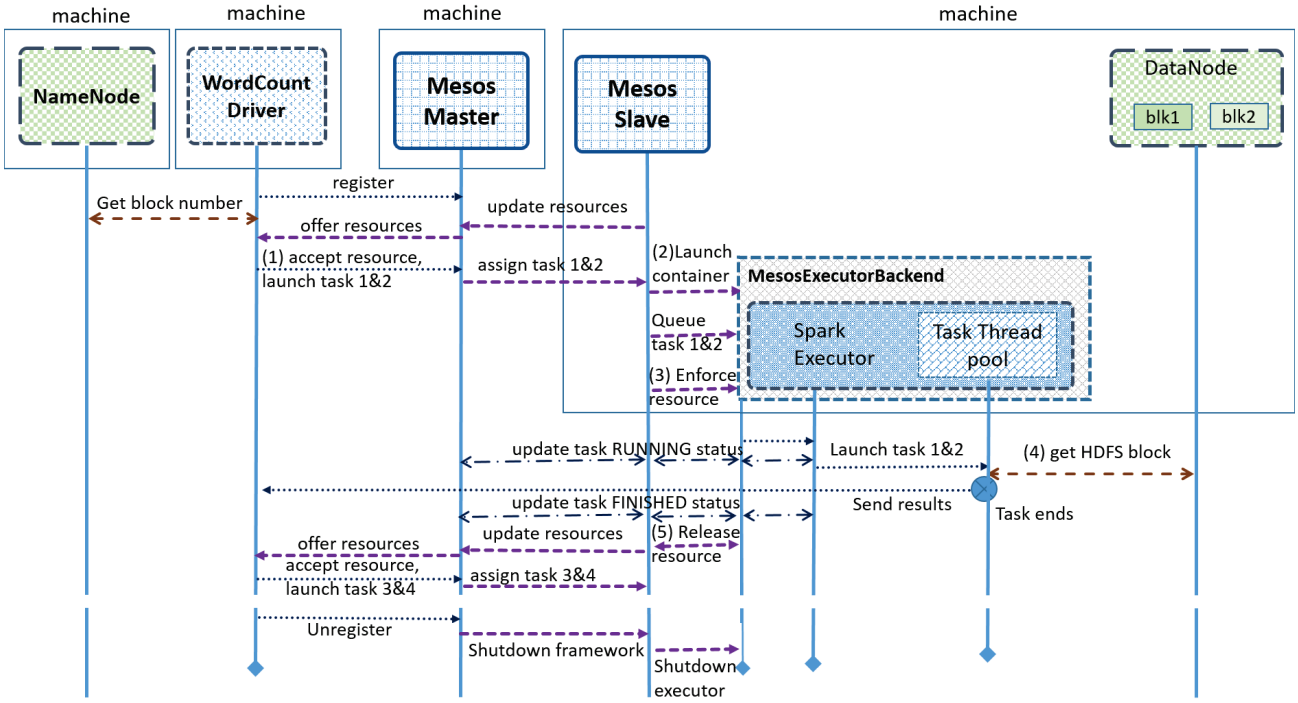
As shown in the work flow of the WordCount application (Fig. 2(b)), the disk I/O contention may happen when tasks read data blocks (action 4). The performance measures related to the I/O activities of WordCount and TeraSort are collected and reported in Table I (note that the average values of ten repeated measurements are presented, the buffer cache was dropped before each measurement and was flushed in the end). It can be observed that I/O operations of another Spark applications or external frameworks cause a decrease in the I/O performance of a given executor. For instance, the read data rate of the WordCount application in cases of running alone and with TeraSort are 44.71 MB/s and 22.18 MB/s, respectively. The average delay time for a disk block I/O is another evidence that the disk contention can cause a performance problem for applications. As a consequence, the application runtime increases due to the I/O contentions. The similar observation can be obtained with the Terasort application as well.

Disk bottlenecks can be caused by sequentially reading/writing activity on a large amount of disk blocks of an application. The I/O activity may greedily seize the whole disk I/O capacity and cause I/O starvation for other applications that access the same disk. Such cases can frequently happen in production environments when data is uploaded to HDFS or data reorganized in HDFS. To emulate such cases, we use some simple applications to process 15GB data:

- HDFS-reader and HDFS-writer are simple applications for reading and writing HDFS data,
- Fio [12] is initiated for synchronous reading and writing data. Fio-reader and Fio-writer are applications calling Fio to process data in the disk drive shared with HDFS data blocks.



(a) The DAG graph



(b) The execution flow

Fig. 2. Spark WordCount example in Mesos

TABLE I

Perfomance of the WordCount and the TeraSort applications (Avg_read and Avg_write are the average data rate of a container for the read activity and the write activity, blkio_delay (ms) is the average delay (in milliseconds) that an application spends waiting for a disk block I/O.)

Scenario		*Avg_read* (MB/s)	*Avg_write* (MB/s)	*blkio_delay* (ms)	*Runtime* (s)
WordCount	Alone	44.76	-	5.19	40.5
	+TeraSort	22.18	-	13.08	71.3
TeraSort	Alone	36.67	69.38	8.51	99.8
	+WordCount	17.37	70.65	15.71	133.2

TABLE II
Perfomance of the WordCount and the TeraSort applications with/without I/O stress influence

Scenario		*Avg_read* (MB/s)	*Avg_write* (MB/s)	*blkio_delay* (ms)	*Runtime* (s)
WordCount	Alone	44.76	-	5.19	40.5
	+Fio-reader	16.76	-	20.64	90.8
	+Fio-writer	28.20	-	58.08	143.6
	+HDFS-reader	34.46	-	8.25	54.9
	+HDFS-writer	22.41	-	56.52	162.1
TeraSort	Alone	38.89	72.67	7.71	106.2
	+Fio-reader	30.35	64.87	10.71	120.6
	+Fio-writer	27.95	66.18	35.19	209.3
	+HDFS-reader	33.34	66.01	9.21	115.9
	+HDFS-writer	27.40	61.57	37.91	216.9

Tables II shows the measured results. As observed, Spark applications suffer in uncontrolled environments.

III. Proposed Solution

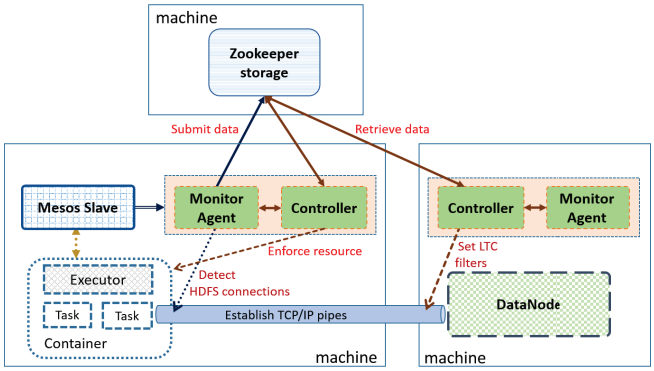


Fig. 3. Enforcing HDFS read data rate in Mesos

Quality of Service (QoS) is defined by Recommendation ITU-T G.1000 [13] to characterize the degree of satisfaction of a user. There are several QoS criteria (speed, accuracy, availability, reliability, security, simplicity and flexibility) [14] that serves as the base for setting QoS parameters and performance objectives. Furthermore, there are four viewpoints [13] of QoS from the perspective of customers and service providers: customer's QoS requirements, QoS offered by a provider, QoS achieved by a provider, QoS perceived by a customer. It is worth mentioning that mechanisms (rules, procedures, policies) should be deployed in the infrastructure of service providers to provision QoS for customers in cloud computing environments as well. Following the general principles regarding the provision of QoS from the viewpoint of service provider in such shared environments, mechanisms for guaranteeing the IO rate of applications should include

- the specification of requirements of users and their applications,
- information about the shared compute cluster, i.e.,
  - the maximum capacity of the resource of the cluster (i.e., the maximum capacity of disk I/O),
  - the amount of resource occupied by containers and application frameworks,
- unified resource management policy (i.e., a strategy to allocate and isolate resource) and decision procedures (admission control and policing).

In [15], we proposed a design of building block components to control a data rate from HDFS for YARN applications. The detailed implementation called HdfsTrafficControl can be found in [16]. This design is refined to support a data I/O provision for Spark applications in Mesos (see Fig. 3).

We propose a process with two main components to be executed in each worker machine: an agent to monitor the resource usage of executors and the worker machine, and a controller to perform resource enforcements. In order to guarantee the data I/O, the Mesos master should known about the I/O usage of the containers, DataNodes, etc. by querying the proposed framework, then offers the usable free disk I/O to the application frameworks, which in turn can decide whether accept or reject the resource offer. The information can be shared amongst the clients through a persistent storage based ZooKeeper, which acts as a passive coordinator in this design.

The integration of our approach into the execution flow illustrated in Fig. 2(b) is described in the following:

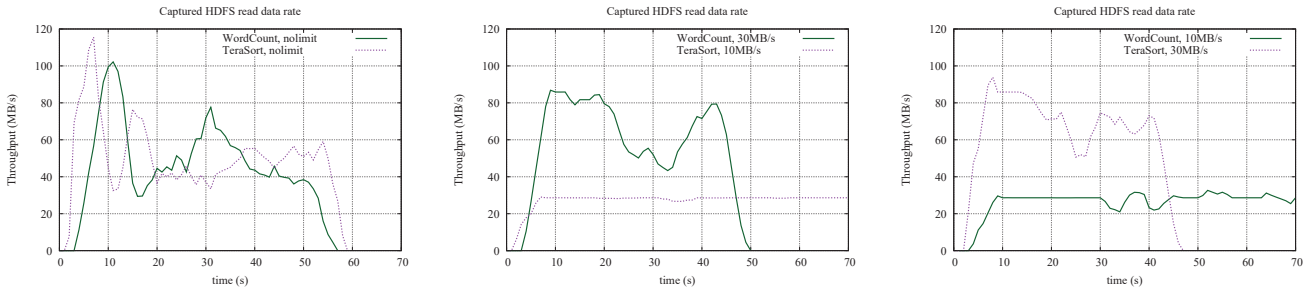
- a Mesos slave may retrieve information from the monitor agent in order to report the amount of usable resource to the Mesos master, which in turn advertise it to Spark drivers. It is worth emphasizing that declaring new resource is easy in Mesos thanks to its flexible resource description model,
- Spark drivers must specify I/O requirements when accepting resources offered by Mesos master (action (1)),
- based on the list of tasks and the I/O requirements (per task), the slave can calculate the total amount of resource for the given container, then it can notify the resource enforcement request to the agent (action (2), (3) and (5)). Mesos already supports the isolator module feature, that can be used to implement this procedure,
- the controller then translates the request into appropriate enforcement settings, and enforcements are performed. The enforcements can be carried out during the lifetime of the executor and resource monitoring may be needed to perform the enforcements (e.g. enforcing HDFS traffic during action (4)). In Linux environment, CGroups Block I/O Controller [17] or LTC based TCP/IP controller [18]

TABLE III
Wordcount application - 3GB data with block size of 256MB

Scenario		Avg_read (MB/s)	$blkio\_delay$ (ms)	$Runtime$ (s)
Baseline		44.76	5.19	40.5
With IO-stress influence	+Fio-reader	16.76	20.64	90.8
	+Fio-writer	28.20	58.08	143.6
	+HDFS-reader	34.46	8.25	54.9
	+HDFS-writer	22.41	56.52	162.1
Under IO controls	+Fio-reader/CGroups	40.51	6.41	44.2
	+Fio-writer/CGroups	28.01	59.71	143.7
	+HDFS-reader/LTC	40.39	6.37	49.6
	+HDFS-writer/LTC	35.85	7.24	52.6

TABLE IV
Terasort application - sorting 3 million records with block size of 256MB

Scenario		Avg_read (MB/s)	Avg_write (MB/s)	$blkio\_delay$ (ms)	$Runtime$ (s)
Baseline		38.89	72.67	7.71	106.2
With IO-stress influence	+Fio-reader	30.35	64.87	10.71	120.6
	+Fio-writer	27.95	66.18	35.19	209.3
	+HDFS-reader	33.34	66.01	9.21	115.9
	+HDFS-writer	27.40	61.57	37.91	216.9
Under IO controls	+Fio-reader/CGroups	34.55	58.77	9.81	108.6
	+Fio-writer/CGroups	27.96	63.25	37.78	216.0
	+HDFS-reader/LTC	33.88	66.64	9.24	113.0
	+HDFS-writer/LTC	29.33	62.90	10.67	118.8



(a) Without limiting rates (b) WordCount is preferred (c) TeraSort is preferred

Fig. 4. Captured HDFS read data rate of the executors of WordCount and TeraSort in the same worker machine

can be applied to control disk I/O. The interested readers can find more information about them in [15], [19].

This design also takes care of the case when multiple nodes are involved in order to perform the enforcement. For example, enforcing data I/O of HDFS read operations with LTC is more complicated [15] as

- the enforcement of the I/O usage of containers and HDFS only can be done at DataNodes in the machine level as LTC has an impact only on outgoing traffic,
- the TCP/IP pipe and disk I/O pipe are hidden from other resource management functions due to the implementation of HDFS and can not be revealed at the beginning.

Therefore, the connection data related to the given container must be available in order to perform the enforcement in the Datanode machine: the agent periodically checks and detects a TCP/IP connection to the DataNode, uploads the monitored connection data to the data persistent storage (e.g. using ZooKeeper). The controller component alongside the DataNode, in turn, will be notified, and then appropriately applies LTC rules for the concerned HDFS connections in the Datanode machine.

IV. Proof-Of-Concept

In this section, we illustrate that administrators of a shared cluster can apply our proposed solution to give a preference for certain applications, a feature is much needed in production environments.

Tables III and IV summarize the measured I/O performance of the Spark WordCount and TeraSort. When a specific application (e.g., as Wordcount and Terasort in our application) needs to be executed in a certain time limit, the application of our solution can give a higher data rate (from 16.47 MB/s to 40.51 MB/s). Of course, this can be achieved if administrators limit the I/O activities of less important applications.

The result of the data I/O contention can be observed in Fig. 4(a), which plots the captured HDFS read data rate of the executors of WordCount and TeraSort. Fig. 4(b) shows the measurement result when the limited rate per task of WordCount and TeraSort were 30MB/s and 10MB/s, respectively. Similarly, Fig. 4(c) plots the read data rate when TeraSort was preferred. It can be observed that the HDFS read data rate of Spark applications can be properly controlled by our solution.

V. Conclusion

In this paper, we have presented the interaction of different software frameworks in a shared cluster controlled by Mesos where I/O contention may cause a performance degradation for applications. We have proposed a solution that can be applied to monitor and enforce the I/O throughput for applications. It is worth emphasizing that the I/O enforcements were performed in the container level in the experiments. I.e., the total HDFS data rate of tasks inside an executor is controlled because Spark tasks are Java threads spawned inside an JVM executor. Information regarding each task is needed to control the I/O activity in the task level, which requires the cooperation of the frameworks. This issue will be considered in our future work.

References

[1] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[2] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[3] "Cloudera Industry Brief: Big Data Use Cases for Telcos," https://www.cloudera.com/content/dam/cloudera/Resources/PDF/solution-briefs/Industry-Brief-Big-Data-Use-Cases-for-Telcos.pdf, accessed: 2015-12-17.

[4] O. Acker, A. Blockus, and F. Potscher, "Benefiting from big data: A new approach for the telecom industry", Strategy& report, http://www.strategyand.pwc.com/media/file/Strategyand_Benefiting-from-Big-Data_A-New-Approach-for-the-Telecom-Industry.pdf, accessed: 2015-12-17.

[5] "CEM on Demand - see every aspect of the customer experience," http://networks.nokia.com/portfolio/products/customer-experience-management, accessed: 2015-12-17.

[6] "IBM Service Provider Delivery Environment Framework," http://www-01.ibm.com/software/industry/communications/framework/, accessed: 2015-12-17.

[7] "Apache Spark," http://spark.apache.org/.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496972

[9] "Software projects built-on Mesos," http://mesos.apache.org/documentation/latest/mesos-frameworks, accessed: 2015-07-01.

[10] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 323–336. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972490

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[12] "Fio - an I/O tool for benchmark and stress/hardware verification," http://freecode.com/projects/fio/.

[13] Recommendation ITU-T G.1000, *Communications Quality of Service: A Framework and Definitions*. International Telecommunication Union, 2001.

[14] J. Richters and C. Dvorak, "A Framework for Defining the Quality of Communications Services," *Communications Magazine, IEEE*, vol. 26, no. 10, pp. 17–23, Oct 1988.

[15] T. V. Do, B. T. Vu, N. H. Do, L. Farkas, C. Rotter, and T. Tarjanyi, "Building Block Components to Control a Data Rate in the Apache Hadoop Compute Platform," in *Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on*, Feb 2015, pp. 23–29.

[16] "Building Block Components to Control a Data Rate from HDFS," https://issues.apache.org/jira/browse/YARN-2681.

[17] "Blkio Controller," https://www.kernel.org/doc/Documentation/cgroups/blkio-controller.txt.

[18] "Linux Advanced Routing and Traffic Control: HOWTO," http://lartc.org, Accessed: 2015-04-10.

[19] X. T. Tran, T. V. Do, N. H. Do, L. Farkas, and C. Rotter, "Provision of Disk I/O Guarantee for MapReduce Applications," in *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015,* Volume 2, 2015, pp. 161–166.