Building Block Components to Control a Data Rate in the Apache Hadoop Compute Platform

Tien Van Do*, Binh T. Vu*, Nam H. Do*, Lóránt Farkas§, Csaba Rotter§, Tamás Tarjányi§
*Analysis, Design and Development of ICT systems (AddICT) Laboratory
Budapest University of Technology and Economics
Magyar tudósok körút 2, Budapest, Hungary
Email: do@hit.bme.hu
§Nokia Networks
Köztelek utca 6, Budapest, Hungary

***Abstract*—Resource management is one of the most indispensable components of cluster-level infrastructure layers. Users of such systems should be able to specify their job requirements as a configuration parameter (CPU, memory, disk I/O, network I/O) that are translated into an appropriate resource reservation and resource allocation decision by the resource management function. YARN is an emerging resource management framework in the Hadoop ecosystem, which supports only memory and CPU reservation at present.**

In this paper, we propose a solution that takes into account the operation of the Hadoop Distributed File System to control the data rate of applications in the framework of a Hadoop compute platform. We utilize the property that a data pipe between a container and a DataNode consists of a disk I/O subpipe and a TCP/IP subpipe. We have implemented building block software components to control the data rate of data pipes between containers and DataNodes and provide a proof-of-concept with measurement results.

I. Introduction

Heterogeneous compute clusters can be easily established using physical machines incorporating memory, disks and powerful CPUs to offer Information and Communications Technology (ICT) services. Hadoop [1], [2], [3], [4] is a software framework that has been developed to satisfy the need of processing big data [5], [6] in the scale of petabytes/day with the use of resources offered by compute clusters.

The design of Hadoop considers several factors such as reliability, scalability, programming model diversity, a flexible resource model, etc [4]. The popularity of Hadoop is mainly due to a design decision that allows parallel and distributed computing meanwhile it hides a complexity from users [2], [3], [7]. The first versions of Hadoop consist of a distributed file system and the MapReduce processing framework. Later on, the need for supporting different processing paradigms was recognized and YARN was introduced [4], [8].

Application layer software (customer experience management, operation support system, customer care are typical examples in telecommunication environments) use the services of the infrastructure layer (e.g. Hadoop) and they together reserve resources through the platform (OS, kernel, firmware, etc.), like CPU, memory, disk I/O and network I/O. Scheduler translates those reservations into an appropriate allocation of resources [9].

Job scheduling is an additional task of resource management [4], [10], [11], [12], [13], which is one of the most indispensable components of cluster-level infrastructure layers. YARN [8] is a distributed resource management system for resource allocation in compute clusters [2], [4] and job scheduling, in the particular case of Hadoop MapReduce. It is recognized that job scheduling is a challenging issue because various factors (quality of service, performance, etc) should be taken into consideration to improve the degree of satisfaction of users.

In mobile network environments network equipment vendors are increasingly facing the challenge that their solutions and products need to be deployed in a so called white box scenario, where they run on the same physical infrastructure as applications of the mobile operator, and even more, they share some of the cluster level infrastructure (e.g. shared Hadoop cluster). In such a scenario, a typical Big Data application may consist of multiple jobs that are executed a distributed manner (up to several thousands machines). Some customers may require a data rate guarantee because their jobs should be finished by a certain deadline. Therefore, the provision of the quality of service regarding a data rate guarantee may play a key factor to attract customers. However, YARN (up to version 2.5.1 [8]) only supports the reservation of memory and CPU in compute clusters at present.

In this paper, we exploit the special feature of the Hadoop Distributed File System –HDFS (which is the part of Hadoop [1]) and the capability of Linux Traffic Control –LTC (which was developed under Linux kernels 2.2 and 2.4 and now is incorporated in the newest Linux kernels as a module) subsystem to control the data pipes of containers to HDFS DataNodes in YARN. We propose building block software components that can be integrated into YARN to control the data throughput of applications. We use ZooKeeper [14] to maintain persistent information to control the throughput of data pipes.

The rest of this paper is organized as follows. In Section II, some technical backgrounds on Hadoop, HDFS and resource management are presented. In Section III, a proposal is described. In Section IV, a proof-of-concept is illustrated with measurement results. Finally, Section V concludes our paper.

II. Technical Background

In this Section, we provide a short summary of features and properties we use to construct our proposed solution.

A Hadoop (version 2.0 or higher) compute cluster normally consists of four main groups (one hardware group and three software groups) that are illustrated in Figure 1:

- The hardware infrastructure includes a platform of machines/servers with CPUs, memory and disks, and a network that connects the machines. The hardware infrastructure and the operating systems (running directly on physical or virtual machines) provides resources for the Hadoop system and applications.
- The Hadoop Distributed File System (HDFS) with functional entities (NameNode and DataNodes) is a distributed file system that runs on physical or virtual machines. It stores large data sets (files of gigabytes to terabytes) and provides streaming data access to clients and applications.
- The resource management group with functional entities (Resource Manager –RM, NodeManagers –NM, ApplicationMasters –AM) is responsible to process the resource requirement of applications, and decides where (which physical machines) a specific application should run based on the knowledge of the hardware infrastructure and the locations of data blocks in a HDFS storage. Containers (C) are allocated based on the requirement of applications to execute the tasks of jobs.
- Applications analyze Big Data and do some computations on Big Data. One type of applications use MapReduce, that is a programming model for data processing [3], [2] and imposes a typical workload on top of HDFS associated with the 3 phases (map, shuffle and reduce) of the processing paradigm. Another type of applications of the HDFS is HBase, which is a distributed, column-oriented database to support real-time read/write random access to very large datasets [2], [3]. A systematic survey of application types and their characteristic workload on HDFS is beyond the scope of this paper.

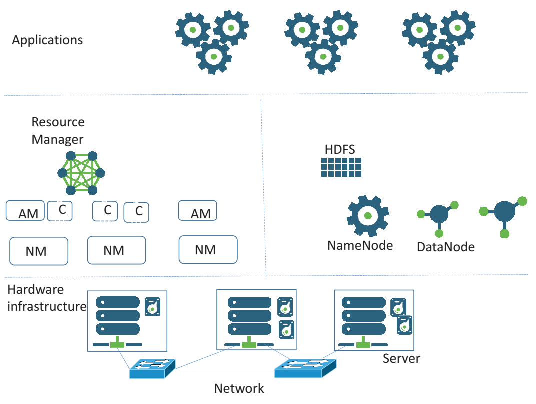

Fig. 1. Main components in a Hadoop Compute Platform

A. Operation of HDFS

In the Hadoop, NameNode and Datanodes are functional components that realize the Hadoop Distributed File System to store files and retrieve blocks of data [2], [3]. Files are splitted into blocks and stored in Datanodes. To ensure the reliable service against failures, the replication mechanism may be applied to allow the placement of the same blocks in different DataNodes. The NameNode is responsible for storing the filesystem tree, the metadata of all the files and directories in the HDFS file system. Information about the locations of the blocks of a specific file is also maintained by the NameNode.

To access/read a specific file, a HDFS client initiates a request to the NameNode to enquire about the list of Datanodes that stores replicas of the blocks of the file. Then, the HDFS client chooses a Datanode that stream data blocks to the client (see Figure 2). In HDFS all communications and data transfers are performed using the TCP/IP protocol stack.

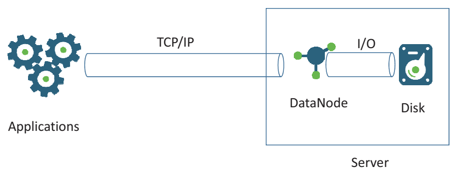

Fig. 2. Data pipes between an application and a DataNode

It is worth emphasizing that the streaming data of a specific file block is conveyed through two pipes (as illustrated in Figure 2): a TCP/IP pipe (through either a network or the loopback interface of a Datanode's machine) between an application and a DataNode, and a disk I/O pipe between a DataNode and a certain disk.

B. Resource Management

YARN decouples the programming model from the resource management infrastructure [4]. In the YARN architecture there are several important entities: Resource Manager (RM), Node Managers (NM), Application Masters (AM). There is a special term "container" that is the collection of resources (CPU and memory) centrally assigned by the RM. In YARN, negotiations regarding resources are performed between a client, its ApplicationMaster and RM, and decisions are taken by RM.

However, the resource usage related to a HDFS storage is not covered by YARN, which may cause performance problems because certain types of applications such as MapReduce have HDFS-intensive resource consumption. Furthermore, the identification of HDFS data pipes is hidden from other resource management functions, which causes a challenge for resource management. We describe our approach in Section III to handle these problems.

III. A Proposed Solution

In this Section, we propose a solution that allows service providers to control the data rate (customer's QoS requirement) of applications from a HDFS storage in Hadoop compute clusters.

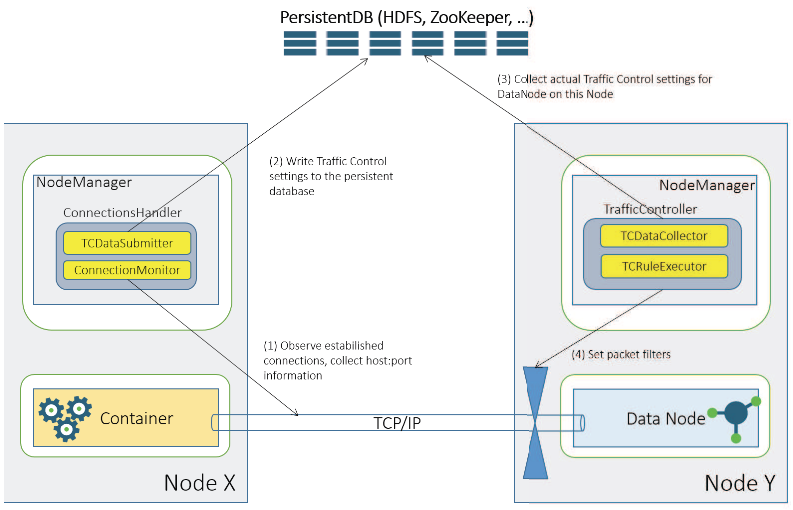
PersistentDB (HDFS, ZooKeeper, ...)
(3) Collect actual Traffic Control settings for DataNode on this Node
(2) Write Traffic Control settings to the persistent database
NodeManager
ConnectionsHandler
TCDataSubmitter
ConnectionMonitor
NodeManager
TrafficController
TCDataCollector
TCRuleExecutor
(1) Observe estabilished connections, collect host:port information
(4) Set packet filters
Container
TCP/IP
Data Node
Node X
Node Y

Fig. 3. Data rate control of pipes

Quality of Service (QoS) is defined by Recommendation ITU-T G.1000 [15] as the collective effect of service performances that characterize the degree of satisfaction of a user. There are several QoS criteria (speed, accuracy, availability, reliability, security, simplicity and flexibility) [16] that serve as the base for setting QoS parameters and performance objectives. Furthermore, there are four viewpoints [15] 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.

Today YARN is not supporting yet metrics like delay/latency of the query execution, which would be very useful to specify/enforce from the viewpoint of an end user or application. The reservation of CPU and memory is supported in YARN. However, the inclusion of other metrics like the disk I/O rate and execution time, etc, is far from trivial and deep application insight/fingerprinting is necessary, which is subject for future work. Following the general principles regarding the provision of QoS from the viewpoint of service provider, mechanisms to control the data rate of applications should include

- the specification of requirements of users (applications),
- Information about a specific compute cluster, i.e.,
  - the maximum capacity of the resource of a cluster (i.e., the maximum capacity of disk I/O),
  - the network topology and the network capacity (i.e., the maximum capacity of network I/O between machines) of a cluster,
  - the amount of resource occupied by containers in a cluster, and the identification of pipes between applications and DataNodes, and amongst DataNodes,
- resource management policy (i.e., a strategy to allocate resource) and decision procedures (admission control and policing) performed by RM.

Within the YARN framework, a client submits a job to the RM. The submission of a job contains the resource requirements for a container that will host the ApplicationMaster of the client [2], [4]. ApplicationMaster is responsible to request a set of containers to run its tasks on. An instance of `Resource` class conveys the resource type requirements of containers. Therefore, to support a new type of resource, `Resource` class should be extended to contain the requirements of a new resource type (e.g., the IOPS, the reads per second, the writes per seconds, throughput).

Upon the arrival of requests for containers, RM should perform an admission control procedure to check whether current available resources are sufficient for the requested containers. The decision by the admission control procedure is based on the information about the capacity of the cluster, the amount of resources occupied by the allocated containers in the cluster and the resource requirement of containers that are being requested by the ApplicationMaster of a specific client.

If the admission control allows the allocation of a container, the ApplicationMaster sets up the ContainerLaunchContext and communicate with the ContainerManager of a specific NodeManager to start its allocated container. The ApplicationMaster also monitors the status of the allocated containers. If a task running on a container finished, the ApplicationMaster will get updates of completed containers.

A policing function is responsible to keep and guarantee the required resource for the allocated containers. To control the data rate, the operation of the Hadoop Distributed File System (i.e., how files and blocks are streamed by DataNodes to applications) is taken into consideration in what follows.

Since the streaming data of a specific file block is conveyed through two pipes as illustrated in Figure 2,

- the enforcement of the I/O usage of containers and HDFS

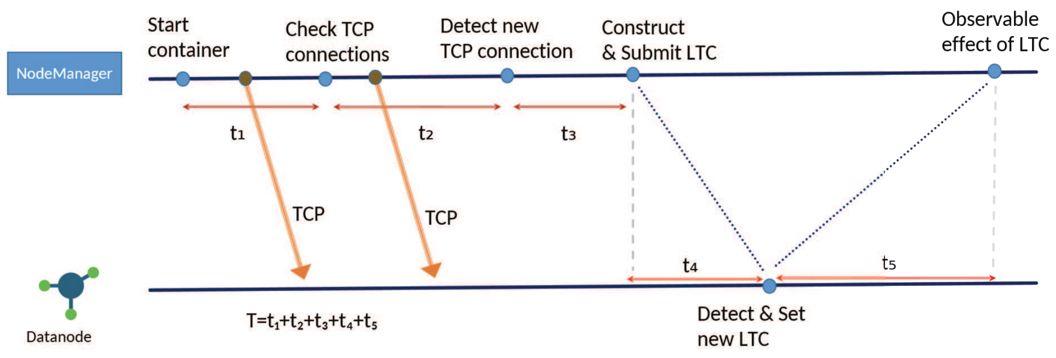

Fig. 4. The time line of the flow execution

could be done at DataNodes in the machine level,

- all the I/O activities of tasks depends on the TCP connections handled by HDFS datanodes,
- the throughput of TCP connections can be controlled by Linux Traffic Control (LTC) –see Figure 3 where the order of steps is illustrated.

Due to the decoupling of functionality, the opening of data pipes (Figure 2) for the usage of a HDFS storage is not explicitly covered by the resource negotiation process. However, applications like MapReduce intensively access the data blocks of big files stored in HDFS. Furthermore, the identification of data pipes is hidden from other resource management functions and can not be revealed at the beginning (i.e., which DataNode is to be contacted for a certain data block by a specific Map task), which causes a challenge for resource management. To configure LTC and control a TCP pipe, the information about the existence of pipes must be obtained. For this purpose, we either take the creation of pipes to the negotiation process or implement a monitor function that senses the setup of TCP pipes between containers and DataNodes. Figure 4 illustrates the time line of the flow execution for the latter alternative with the following durations from the aspect of controlling pipes:

- t_1 is the duration between starting the container and begin of checking TCP connections. Note that TCP connections may be existing in this time interval.
- t_2 is the duration needed to detect TCP connections.
- t_3 is the duration to construct and submit new LTC settings.
- t_4 is the duration needed to configure LTC for a DataNode.
- t_5 is the latency between the start of control and observable effect.

It will be shown in Section IV that the delay ($T = t_1 + t_2 + t_3 + t_4 + t_5$) is acceptable for certain cases (especially when big HDFS blocks are streamed).

A. *Interfaces*

It is worth mentioning that there are a number of alternatives to implement mechanisms and procedures. Therefore, our approach is to define clear interfaces between functions and mechanisms (see Figure 3). Next, we implement building-block functions in Figures 5 and 6 where interfaces and functions for the exchange of information and LTC setting are shown:

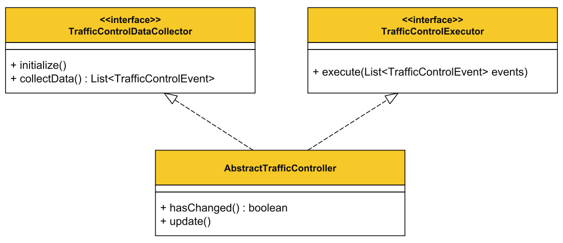
<<interface>> TrafficControlDataCollector
+ initialize()
+ collectData() : List<TrafficControlEvent>
<<interface>> TrafficControlExecutor
+ execute(List<TrafficControlEvent> events)
AbstractTrafficController
+ hasChanged() : boolean
+ update()

Fig. 5. Interface for DataNodes

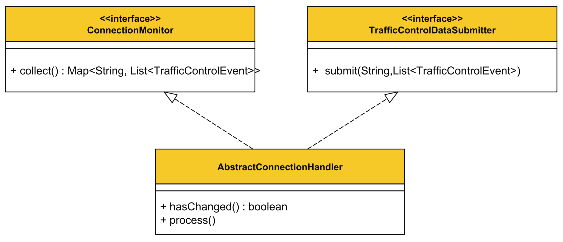
<<interface>> ConnectionMonitor
+ collect() : Map<String, List<TrafficControlEvent>>
<<interface>> TrafficControlDataSubmitter
+ submit(String,List<TrafficControlEvent>)
AbstractConnectionHandler
+ hasChanged() : boolean
+ process()

Fig. 6. Interface in machines with a NodeManager

- `ConnectionMonitor` maintains the information of connections between container nodes and DataNodes.
- `TrafficControlDataSubmitter` submits data collected by ConnectionMonitor to a persistent component.
- `TrafficControlDataCollector` collects data submitted by `TrafficControlData` and creates the list of appropriate events.
- `TrafficControlExecutor` performs the configuration of Traffic Control actions on devices according to the list of events collected by `TrafficControlDataColletor`.

Because the amount of persistent information and control information to support the provision of QoS is huge, a Best of Practice approach (to ensure a lean operation) is to define an operation policy. For example, a limited number of container classes (defined based on data rates) should be supported for containers, or containers are allocated in each machine based on the number of cores and data rates (i.e., preplanned container classes based on cores and data rates).

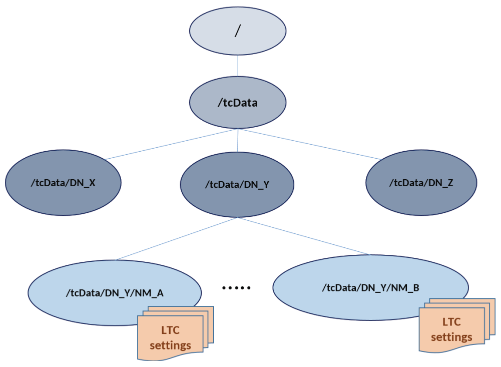

Fig. 7. Data structure for storing LTC settings in ZooKeeper

B. Keeping the information of containers's pipes with ZooKeeper

The maintenance of traffic control parameters and changes of the traffic control parameters requires a persistent data structure that can be established with the use of Apache ZooKeeper [14]. The ability to provide high availability and high performant service in distributed systems (i.e., to handle partial failures, to support loosely couple interactions) gives the rationale behind the choice of ZooKeeper.

The following features of ZooKeeper are taken into account in our design:

- Reading/writing data of zNode is atomic, appending is not possible in ZooKeeper.
- A zNode can be either persistent or ephemeral. A persistent zNode can be only deleted manually. An ephemeral zNode, in contrast, will be deleted if the client that created it crashes or simply closes its connection to ZooKeeper.
- ZooKeeper deals with changes using watches. With watches, a client registers its request to receive a one-time notification of a change to a given zNode.

In this implementation ZooKeeper acts as a persistent layer for storing and delivering LTC settings between nodes. Note that only one ZooKeeper Server is needed for the operation (of course, additional ZooKeeper servers can be operated to increase the reliability). The data structure is illustrated in Figure 7. All related traffic control data will be stored under the */tcData* root zNode. Each Datanode will register itself with ZooKeeper server by creating $/tcData/DN_{ID\_OF\_DN}$ where ID_OF_DN is its identification, it can be the hostname or the IP address of the Datanode.

In each NodeManager, `ConnectionMonitor` investigates the traffic connections and constructs the content of LTC settings and pass them to `TCDataSubmitter`. Then `TCDataSubmitter` notify new demands to Datanodes by creating new $/tcData/DN_{ID\_OF\_DN}/NM_{ID\_OF\_NM}$ zNode with LTC data settings (if this zNode did not exist) under the zNode node of the corresponding Datanode or replacing the data of this zNode with the new one.

In order to get new LTC settings, `TCDataCollector` uses watches to collect data related to its Datanode. The first watch is associated to $/tcData/DN_{ID\_OF\_DN}$ zNode for tracking the creation/deletion of child zNodes (e.g. whether we have new demands from new NodeManager node). So we can get notifications about new demands from the corresponding NodeManager. When changes are detected, the LTC settings will be processed by `TCDataCollector`. It retrieves modifications by comparing with previous one and pass them to `TCExecutor` to set the LTC table.

C. LTC

In Linux, queues[1] can be setup to manage the bandwidth of TCP/IP pipes. That is, filters can be specified to classify traffic based on the source address, the destination address, and the port numbers of TCP sessions, and/or u32[2] condition. Then, different algorithms (e.g., Token Bucket Filter, Stochastical Fairness Queueing, Random Early Detection) can be used to control the rate of TCP/IP sessions [17].

Function `TrafficController` and its relation with the other components of the YARN framework are illustrated in Figure 8.

IV. A PROOF-OF-CONCEPT

Our proposal has been implemented within the YARN framework. The source codes of software components can be obtained from [18]. Using machines with Intel Core i5-4670 CPU, 16GB DDR3 1600 MHz RAM memory and 1TB hard disks, we have created a small Hadoop cluster to demonstrate the capability of our building block software components.

In our testbed, we provide an illustration to control a data rate for the following class of applications:

- a job consists of multiple tasks,
- the execution of a job can be divided into several phases,
- the majority of tasks should be executed within a specific phase, and few tasks span several phases,
- tasks belonging to one specific phase can be executed in parallel, each of them require one container,
- there are HDFS intensive tasks. It is reasonable to require that those tasks process data as streams of bytes. HDFS intensive tasks that are simultaneously executed require different HDFS data blocks.

Figure 9 depicts the TCP throughputs of two containers that read data blocks from the same DataNode. Note that the limiting rate setting in LTC is for the IP layer. It is observed that container 1 that reads data blocks before container 2 gets a higher throughput if no LTC is applied. Note that disk I/O

[1] http://lartc.org/howto/lartc.qdisc.html

[2] a match on any part of a packet

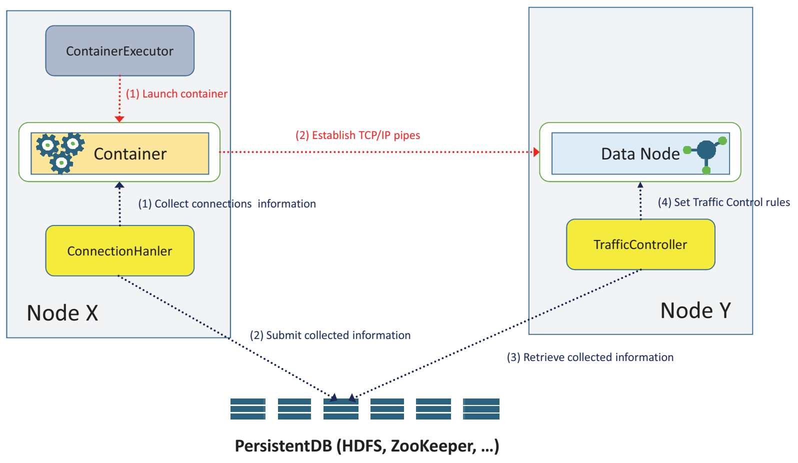
ContainerExecutor
(1) Launch container
Container
(2) Establish TCP/IP pipes
Data Node
(1) Collect connections information
(4) Set Traffic Control rules
ConnectionHanler
TrafficController
Node X
Node Y
(2) Submit collected information
(3) Retrieve collected information
PersistentDB (HDFS, ZooKeeper, ...)

Fig. 8. The flow of executions when TCP connections are detected

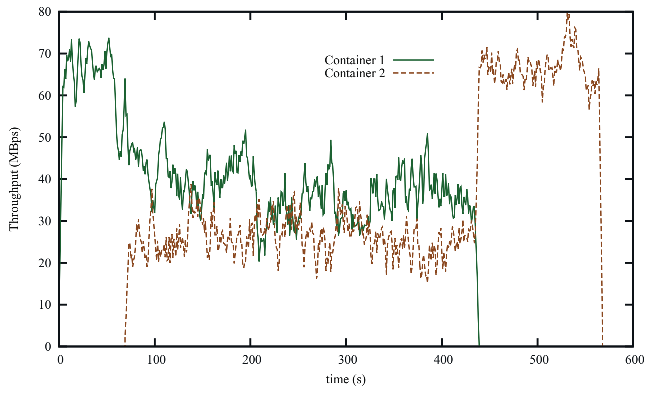

Fig. 9. TCP throughput of pipes without LTC

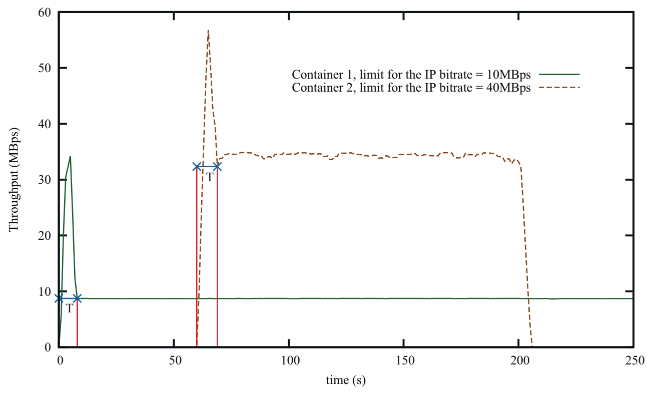

Fig. 10. TCP throughput of pipes controlled by LTC

bottlenecks may happen due to certain conditions. E.g., one cause of disk I/O bottlenecks is the consequence of the large amount of disk blocks sequentially read by applications from a HDFS storage. In such a case, applications that read a huge volume of data blocks from disks (I/Os per second and the amount of bytes per I/O are high) may greedily seize the whole disk I/O capacity.

To provide a proof-of-concept, we apply LTC to limit the IP bitrate of container 1 to be 10 MBps and the container 2 to be 40 MBps (note that the rates can be set to other values as well). From Figure 10 we can conclude that the throughput control mechanism of LTC practically eliminates the disadvantage of the later born data pipes in the comparison with the original YARN (where there is not any limit for pipes –see Figure 9). It worth emphasizing that the greedy nature related to the I/O activity under the uncontrolled period can be observed in Figure 10 too: applications with the sequential reads of data blocks tend to capture all available I/O capacity during the uncontrolled period if there is a room to increase its I/O activity. From the perspective of resource management and containers that read a huge volume of data, the uncontrolled period of pipes (i.e., the delay denoted by T in Figure 10 from the start of containers until the LTC has the impact on the data rate of containers) is negligible. As we mentioned earlier, the information about the establishment of data pipes should be a part of the QoS negotiation process to eliminate the uncontrolled period, which will be done in our future work.

V. Conclusion

We have presented an approach to take into account the HDFS feature to control the throughput of data pipes between applications and HDFS DataNodes in the YARN framework. Some basic building block functions have been implemented to exploit the property (data pipes between DataNodes and containers) of the HDFS architecture. It has been shown through measurement results that the throughput of applications (jobs, containers) can be controlled within YARN using our building block components.

At present we are working on a prototype to extend YARN. The prototype will include the specification of I/O rates by applications, the admission control procedure, and scheduling and management policy along with building block software components described in this paper.

REFERENCES

[1] Apache Hadoop, *http://hadoop.apache.org*.

[2] T. White, *Hadoop: The Definitive Guide*, 3rd ed. O'Reilly Media, Inc., 2012.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, May 2010, pp. 1–10.

[4] 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

[5] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using Hadoop," in *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, and V. J. Tsotras, Eds. IEEE, 2010, pp. 996–1005. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2010.5447738

[6] P. Zikopoulos and C. Eaton, *Understanding Big Data: Analytics for enterprise class Hadoop and streaming data*. McGraw-Hill, 2011.

[7] I. Polato, R. R, A. Goldman, and F. Kon, "A comprehensive view of Hadoop research: A systematic literature review," *Journal of Network and Computer Applications*, vol. 46, pp. 1 – 25, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804514001635

[8] Apache Hadoop NextGen MapReduce (YARN), *http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html*.

[9] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013. [Online]. Available: http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024

[10] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, pp. 1–53, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10922-014-9307-7

[11] S. Suresh and N. Gopalan, "An optimal task selection scheme for Hadoop scheduling," *IERI Procedia*, vol. 10, pp. 70 – 75, 2014, international Conference on Future Information Engineering (FIE 2014). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2212667814001415

[12] 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

[13] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The Condor experience: Research articles," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, Feb. 2005. [Online]. Available: http://dx.doi.org/10.1002/cpe.v17:2/4

[14] Apache ZooKeeper, *http://zookeeper.apache.org/*.

[15] Recommendation ITU-T G.1000, *Communications quality of service: A framework and definitions*. International Telecommunication Union, 2001.

[16] 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.

[17] B. Hubert et al., *Linux Advanced Routing and Traffic Control: HOWTO*. http://lartc.org/, 2009.

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