A framework for supporting the bandwidth enforcement of reading from HDFS

Tien Van Do[1], Nam H. Do[1], and Binh T. Vu[2]

[1]Analysis, Design and Development of ICT systems (AddICT) Laboratory, Budapest University of Technology and Economics, Hungary

July 13, 2015

1 Introduction

This document introduces the implementation of the HdfsTrafficControl framework, which was introduced in [1].

2 HdfsTrafficControl

2.1 Terminology

- **Container**: an object represented by a triple $\{containerId, rate, pid\}$, where
  - *containerId*: is an unique name of the given container. A valid containerId has only '-' and word characters ($[a-zA-Z\_0-9]$). It must be started with an alphabetic character and ended with a word character. Its length is not longer than 100,
  - *pid*: is the id of the process spawned by the external framework when starting the given container. This process can be the one spawned by the container executor or its only child process. Only processes with $pid > 1$ are considered,

  - *rate*: is a read bandwidth limit in mbps. It is a positive float number.
  - Examples of containers: YARN containers with containerId is the container's name and the pid is an id of the process spawned by the container executor. Another example is a "virtual" container for the copyToLocal operation using HDFS shell. In this case the containerId is any unique name and the pid is the pid of the copy process.

- **Connection**: a TCP connection identified by the port and address of a source and destination machine.

- **NMContainerConnections**: a container seen by ConnectionHandler. It is extended by a list of considered connections from the given container. These connections are grouped by the destination hosts. Note that tt is just a snapshot of connections from this container at the collected time.

- **DNContainerConnections**: a container seen by TrafficController. It is extended by a list of considered connections from the given host to the given container. Therefore one NMContainerConnections can consists of many DNContainerConnections.

- **TrafficControlEvent**: an event generated based on the retrieved connection data. Possible type of events are: ADD_CLASS, DEL_CLASS, $CHANGE\_CLASS$, ADD_FILTER and DEL_FILTER. One TrafficControlEvent belongs to only one container.

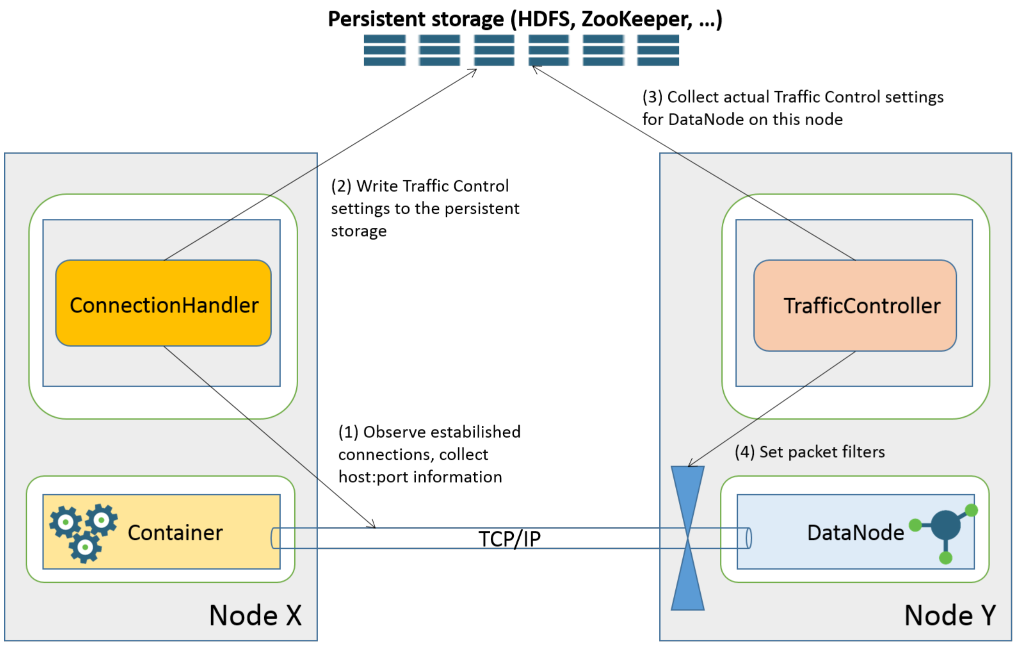
2.2 The framework architecture

The Fig. 1 shows the design of the framework, which is introduced in [1].

2.2.1 ConnectionHandler component

Fig. 2 shows the components of *ProcBasedConnectionHandler* which implements the *ConnectionHandler* block in Fig. 1.

The components of *ProcBasedConnectionHandler* are following:



Persistent storage (HDFS, ZooKeeper, ...)
(3) Collect actual Traffic Control settings for DataNode on this node
(2) Write Traffic Control settings to the persistent storage
ConnectionHandler
TrafficController
(1) Observe estabilished connections, collect host:port information
(4) Set packet filters
Container
TCP/IP
DataNode
Node X
Node Y

Figure 1: The design of the framework.

- **ConnectionCollector**: Collect the TCP connections and inodes of the processes from the */proc* file system and the output of the ss program (shipped with *iproute2*)

- **ContainerRegister**: Manage containers received from the external frameworks through the plugins like P1, P2. One or more such plugins can be registered, which implements the *AbstractContainerService* interface. The external frameworks can add/remove containers or register the PID of containers. Note that the external frameworks are responsible for providing valid rates and PIDs. The containers with invalid rate or PID will be removed by *ContainerRegister*. Furthermore, it also removes a container which doesn't report the PID within 10 minutes after registering. One demo plugin called *FileBasedContainerService* is provided in the current implementation.

- **ProcBasedConnectionMonitor**: Use the container data (*containerId*, *rate*, *pid*) received from *ContainerRegister* and the collected connec-

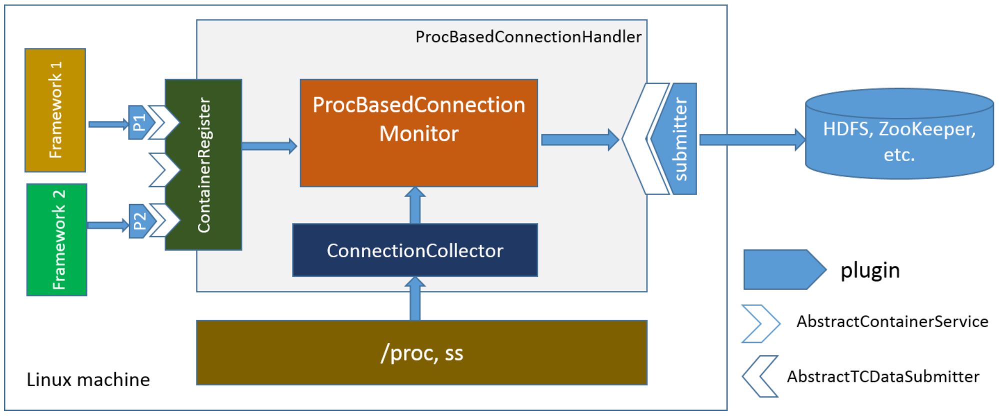


Figure 2: The components of ProcBasedConnectionHandler

tions, inodes from *ConnectionCollector* to build the list of *NMContainerConnections*, collect the data to be persisted (data related to all hosts that have changes since last submit) and pass them to the *submitter*.

- **submitter**: Pluggable component to persist data into the given back-end storage. These plugins must implement the *AbstractTCDataSubmitter* interface. ConnectionHandler keeps track of all collected connections, so it can detect the list of hosts have updates and pass them and their connection data to the submitter. These data will be persisted to the back-end storage by the submitter. Note that the connection data to store for each host contains all connections to that host.

2.2.2 TrafficController component

Fig. 3 shows the components of the *TrafficController*, which implements the *TrafficController* block in Fig. 1.

The components of *TrafficController* are followings:

- **ConnectionCollector**: Collect connections of the monitoring service using */proc* file system and the output of *ss* program. It is used only in some first rounds after starting for the synchronization.
- **collector**: Pluggable component to retrieve data from the given back-end storage. it must implement the *AbstractTCDataCollector* interface.

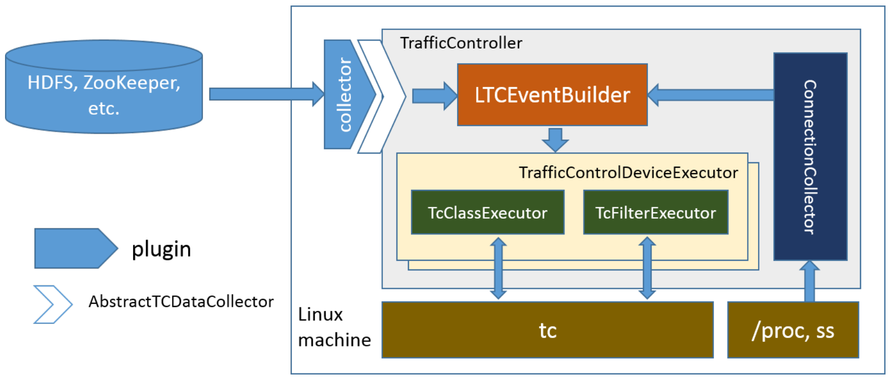


Figure 3: Traffic Controller

In each round it should only retrieve the connection data from those hosts which are updated since the last collection.

- **LTCEventBuilder**: This component use connection data reported by the *collector* and *ConnectionCollector* to construct a list of LTC settings to be executed and pass them to a corresponding *TrafficControlDeviceExecutor*.
- **TrafficControlDeviceExecutor**: One or more *TrafficControlDeviceExecutor* may be available. Each of them is correspond with one NIC device. This component handles all LTC settings for the given device (adding/changing/deleting LTC classes, adding/deleting LTC filters). It also collects and handles the old LTC settings in the setup phase. Note that the user starting TrafficController must be able to exetue the "tc" program as root without password.
- **TcClassExecutor**: A sub-component of *TrafficControlDeviceExecutor*, and is responsible for executing LTC class settings using the "tc" program. It maintains an own index system in order to reuse LTC handles/classifiers.
- **TcFilterExecutor**: A sub-component of *TrafficControlDeviceExecutor*, and is responsible for executing LTC filter settings using the "tc" program. It maintains an own index system in order to reuse LTC handles/classifiers.

2.3 Important interfaces

In this section some important interfaces are introduced.

2.3.1 AbstractService

This interface consists of 3 methods and is implemented by all services/components of *HdfsTrafficControl*.

- **initialize**(String hostId): Post initialization before starting,
- **start**(): Start the service/component,
- **stop**(): Stop the service/component, release all occupied resources.

2.3.2 ConnectionMonitor

It is implemented by *ProcBasedConnectionMonitor* (see Fig. 2).

- **collect**(): Collect connections data to all DataNodes, where changes are detected. The connections are grouped by the DataNodes.

2.3.3 TrafficControlDataSubmitter

It is implemented by the *submitter* plugins (see Fig. 4).

- **submit**(String dnHost, List<DNContainerConnections> connections): Persist data (list of DNContainerConnections) related to one DataNode into the back-end storage.

2.3.4 TrafficControlDataCollector

It is implemented by the *collector* plugins (see Fig. 5).

- **collectData**(Map<String, List<DNContainerConnections>>): Collect the updated remote hosts (from the previous collection) and their corresponding list of DNContainerConnections from the back-end storage.

2.3.5 LTCEventBuilder

It is implemented by *TrafficController* (see Fig. 3).

- **buildTCEvents**(Map<String, List<DNContainerConnections>>): Construct the list of LTC settings to be executed based on the list of the updated remote hosts and their corresponding list of DNContainerConnections.

2.3.6 TrafficControlExecutor

It is implemented by *TrafficController* (see Fig. 3).

- **execute**(List<TrafficControlEvent>): Execute the constructed LTC settings on the corresponding NIC device.

2.3.7 ContainerService

This interface consists of 4 methods to provide functionalities for managing containers.

- **addMonitoringContainer**(String containerId, float rateInMbps): Add new container to monitor. The rate must be specified and cannot be zero,
- **registerPid**(String containerId, String pid): Register the id of the first process spawned by the container,
- **updateContainerRate**(String containerId, float rateInMbps): Set new rate for the container.
- **stopMonitoringContainer**(String containerId): Stop monitoring the container with given id.

3 Collecting TCP connections

The connections between containers and DataNode are TCP connections [1], with source or destination port is a monitoring port (with default value is *50010*). In Linux, the TCP network connections can be collected by reading */proc/net/tcp* and */proc/net/tcp6* files or parsing output of some Linux utilities like “netstat”, “lsof”, “ss”, etc. Note that using these tools we can obtain

2 important data: the TCP connections and their corresponding socket inodes.

It is known problem that sometimes established connections disappear from */proc/net/tcp* and */proc/net/tcp6* files, hence in our implementation 2-round capture is used (with reading from /proc in the first round, and parsing the output of ss in the second round after small delay), and the connections will be collected accumulatively.

It is important to note that Hadoop only works with IPv4. Therefore we only need to take care of IPv6 in the *ConnectionHandler* side and it is expected that we can always convert to IPv4 in this case. Hence first the IP data will be retreived from the /proc/net/tcp, /proc/net/tcp6 and the output of *ss* (may contain both forms of IPv4 and IPv6), and will be converted to a valid Java InetAddress object, then the address will be obtained in the IPv4 form. For example "10.10.0.116" can be represented as ::ff:10.10.0.116 (IPv4 mapped format), or ::10.10.0.116 (IPv4 compatible format) or 2002:0a0a:0074:: (6to4 format).

On *TrafficController* side, only the list of connections from the monitoring port are needed for synchronizing LTC settings after the setup phase. Hence it is enough to using the above procedure. In this case, the used ss command is "*ss -noept state established src *:50010*"

On the other side, *ConnectionHandler* must group connections by the containers. In order to archive this task, the list of processes of each container is needed. First the /proc file system is walked and the tree of the processes will be constructed by parsing the stat file of the processes to collect the process id of their parent. Then the socket inodes of these processes can be determined (This data can be obtained by listing the folder "fd" of the given process in the /proc file system, only socket symlinks are considered). Based on these data we can classify the connections of each container, if the pid of the container is specified (Connections of the container are the connections belong to the processes of the given container, in other word they are the descendants of the container process and itself).

In Linux only the owner of processes can access their fd folder. In this case we cannot rely on /proc system to retrieve data of inodes. However it is a good news that "ss" program with sudo right can provide the process id for each connection. Hence to deal with this problem, the user starting *ConnectionHandler* must be able to execute "ss" program as root without password in case of multiple users are supported (i.e. users starting *ConnectionHandler* are different with users executing containers). The used ss command is "*[sudo] ss -noept state established dst *:50010*"

4 Recovery strategy

As *HdfsTrafficControl* can run as a standalone application, it may be failed or restarted during the its life cycle. Furthermore, the *ConnectionHandler* and *TrafficController* can be started separately. Hence it must be able to synchronize the data from all available sources in the initialization phase before starting the normal operation. The synchronization shouldn't depend on the results of the shutdown phase. In other word in case of the failures, *HdfsTrafficControl* should be able to go back to the normal operation with a minimal loss.

The following process is implemented in order to provide the reliable operation of *HdfsTrafficControl*:

4.1 On ConnectionHandler side

In the initialization phase, *HdfsTrafficControl* will try to obtain the previously active containers and submit them again:

- Plugins implemented *AbstractContainerService* interface should be able to recover as much as possible the active containers reported by them and report them again (they should use own backup approaches for this purpose as the implementation of *FileBasedContainerService*).

- Note that the framework does not rely on the above consumption. It will collect the list of all previously connected hosts from the given backup file (*/tmp/connected_dn_nodes.dat*). The list of actively connected hosts is persisted in the end of each *submit*() operation.

As a result, the connections collected in the first time will be grouped by the destination hosts and containers. Some remote destinations will have empty list of containers, and the submitter will reset their corresponding data on the back-end storage. This step is important as some containers may finish during the downtime of *HdfsTrafficControl* and we may not be able to detect them, and the ralated LTC settings generated by *TrafficController* will remain forever.

4.2 On TrafficController side

As HdfsTrafficControl can stop suddenly due to failures, LTC settings generated previously may still exist. In order to deal with this problem:

- All LTC classes and filters are considered as generated by *HdfsTrafficControl* if the decimal value of their identifiers are at least 1000 (It is needed in order to cooperate with other possible applications/components using LTC). All such LTC classes and filters will be collected in the initialization phase.

- The existing LTC settings will be reseted after the first collecting data collection from the back-end storage. The old LTC classes and filters will be deleted and new LTC settings generated from the collected data are applied (as sometimes it is impossible to retrieve the id of containers from these classes and filters). Hence the effect of shutdown is minimal as the deleting old LTC classes and filters are delayed till the first collection. Furthermore, we can clean the invalid LTC settings thanks to the recovery actions on the *ConnectionHandler* side.

- For the purpose of cleaning data, in some first rounds after the initialization, the connections are also collected for constructing LTC settings (i.e: Only active connections will be taken into account in these rounds). It is needed to deal with a situation when ConnectionHandler of a submitter node is still shutdown and some of its containers are already complete.

5 Inter-process communication (IPC) in HdfsTrafficControl

5.1 IPC between the submitter and the collector

The data submitted by *ConnectionHandler* (see Fig. 2) must be passed to the remote *TrafficController* (see Fig. 3). There are many methods to archive this communication. However the application should be able to recover in case of any errors. Therefore it is recommended to use a back-end storage to store the submitted data and the collector should retrieve data from this storage as illustrated in Fig. 1. The communication of between *ConnectionHandler* and *TrafficController* is realized by a pluggable submitter and collector plugins.

The submitter plugins must implement the *AbstractTCDataSubmitter* class as shown in Fig. 4. Two such plugins are provided in HdfsTrafficControl using HDFS and ZooKeeper as back-end storage. We recommend the use of ZooKeeper back-end storage as HDFS doesn't work well with the rapid changes of many small files. However it is simple to use ino order to validate the application.

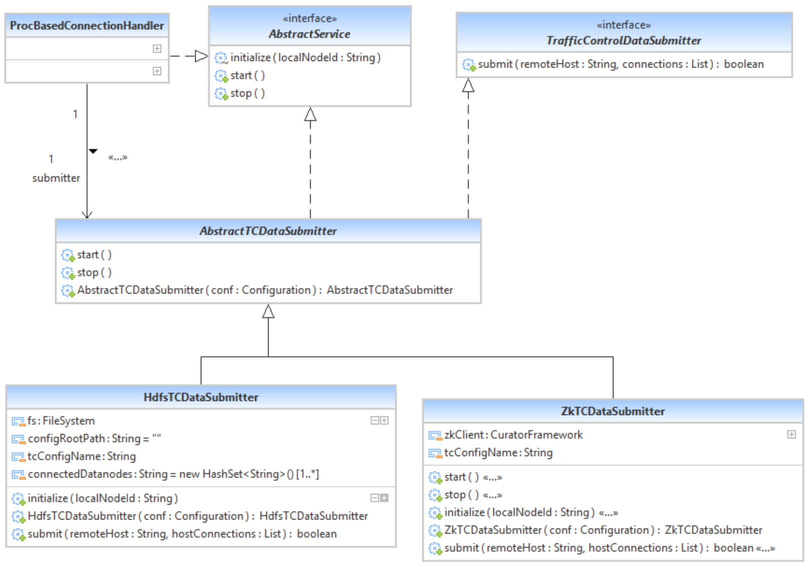
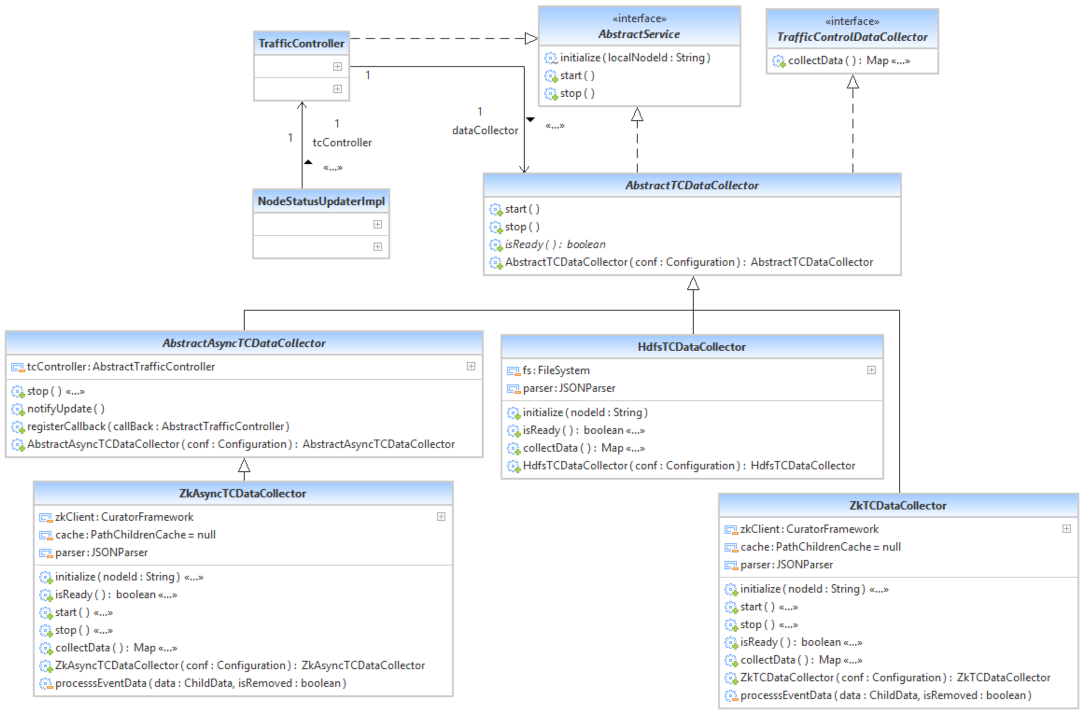

ProcBasedConnectionHandler
«interface»
AbstractService
initialize (localNodeId : String)
start ( )
stop ( )
«interface»
TrafficControlDataSubmitter
submit (remoteHost : String, connections : List) : boolean
1
1
submitter
«...»
AbstractTCDataSubmitter
start ( )
stop ( )
AbstractTCDataSubmitter (conf : Configuration) : AbstractTCDataSubmitter
HdfsTCDataSubmitter
fs : FileSystem
configRootPath : String = ""
tcConfigName : String
connectedDatanodes : String = new HashSet<String>() [1..*]
initialize (localNodeId : String)
HdfsTCDataSubmitter (conf : Configuration) : HdfsTCDataSubmitter
submit (remoteHost : String, hostConnections : List) : boolean
ZkTCDataSubmitter
zkClient : CuratorFramework
tcConfigName : String
start ( ) «...»
stop ( ) «...»
initialize (localNodeId : String) «...»
ZkTCDataSubmitter (conf : Configuration) : ZkTCDataSubmitter
submit (remoteHost : String, hostConnections : List) : boolean «...»

Figure 4: Simplified UML diagram of interfaces/classes for submitting data into the back-end storages

There are two types of collector plugins: asynchronous and synchronous version. The synchronous collectors must implement the *AbstractTCDataCollector* class (see Fig. 5). In this case *TrafficController* will periodically check and retrieve the update data from the storage. On the contrary, the asynchronous collectors will notify *TrafficController* about the changes. They must implement the class *AbstractAsyncTCDataCollector*. Note that in this case *AbstractAsyncTCDataCollector* will make sure that only one data collection is carried out at time and only minimal number of data collections will be called. It means that if there are many notifications occurred during one data collection, then it is enough to call one more data collection at the end of the current one as it is requirement that the data collection must return the data of all updated hosts since the previous collection (This requirement is very important for the initialization of *TrafficController*). HdfsTrafficControl also provides the collector plugins for the HDFS and ZooKeeper back-end storages (see Fig. 5).



TrafficController
«interface» AbstractService
initialize (localNodeId : String)
start ( )
stop ( )
«interface» TrafficControlDataCollector
collectData ( ) : Map «...»
1
1
dataCollector
«...»
1
1
tcController
«...»
NodeStatusUpdaterImpl
AbstractTCDataCollector
start ( )
stop ( )
isReady ( ) : boolean
AbstractTCDataCollector (conf : Configuration) : AbstractTCDataCollector
AbstractAsyncTCDataCollector
tcController : AbstractTrafficController
stop ( ) «...»
notifyUpdate ( )
registerCallback (callBack : AbstractTrafficController)
AbstractAsyncTCDataCollector (conf : Configuration) : AbstractAsyncTCDataCollector
HdfsTCDataCollector
fs : FileSystem
parser : JSONParser
initialize (nodeId : String)
isReady ( ) : boolean «...»
collectData ( ) : Map «...»
HdfsTCDataCollector (conf : Configuration) : HdfsTCDataCollector
ZkAsyncTCDataCollector
zkClient : CuratorFramework
cache : PathChildrenCache = null
parser : JSONParser
initialize (nodeId : String) «...»
isReady ( ) : boolean «...»
start ( ) «...»
stop ( ) «...»
collectData ( ) : Map «...»
ZkAsyncTCDataCollector (conf : Configuration) : ZkAsyncTCDataCollector
processsEventData (data : ChildData, isRemoved : boolean)
ZkTCDataCollector
zkClient : CuratorFramework
cache : PathChildrenCache = null
parser : JSONParser
initialize (nodeId : String) «...»
start ( ) «...»
stop ( ) «...»
isReady ( ) : boolean «...»
collectData ( ) : Map «...»
ZkTCDataCollector (conf : Configuration) : ZkTCDataCollector
processsEventData (data : ChildData, isRemoved : boolean)

Figure 5: Simplified UML diagram of interfaces/classes for retrieving data from the back-end storages

5.2 IPC between HdfsTrafficControl and external scheduler frameworks

The external frameworks can cooperate with HdfsTrafficControl through the pluggable services as illustrated in Fig. 2. One or more such plugins can be registered with *ContainerRegister* component. These plugins must implement the *AbstractContainerService* as shown in Fig. 6. One such plugin is *FileBasedContainerService*, which watches a dedicated folder to handling incoming container requests. These plugins must handle the following operations:

- Register a new monitoring container with a specific id and none-zero rate. Note that the external frameworks are responsible for determining the limit rate. However this rate can be normalized by the plugin. The containers with invalid rate are ignored,
- Report the pid of the first process spawned for the monitoring container,
- Remove a monitoring container. It is optional as the ConnectionHandler will remove a monitoring container if its pid is not existed anymore or a newly registered container doesn't have a pid within 10 minutes.

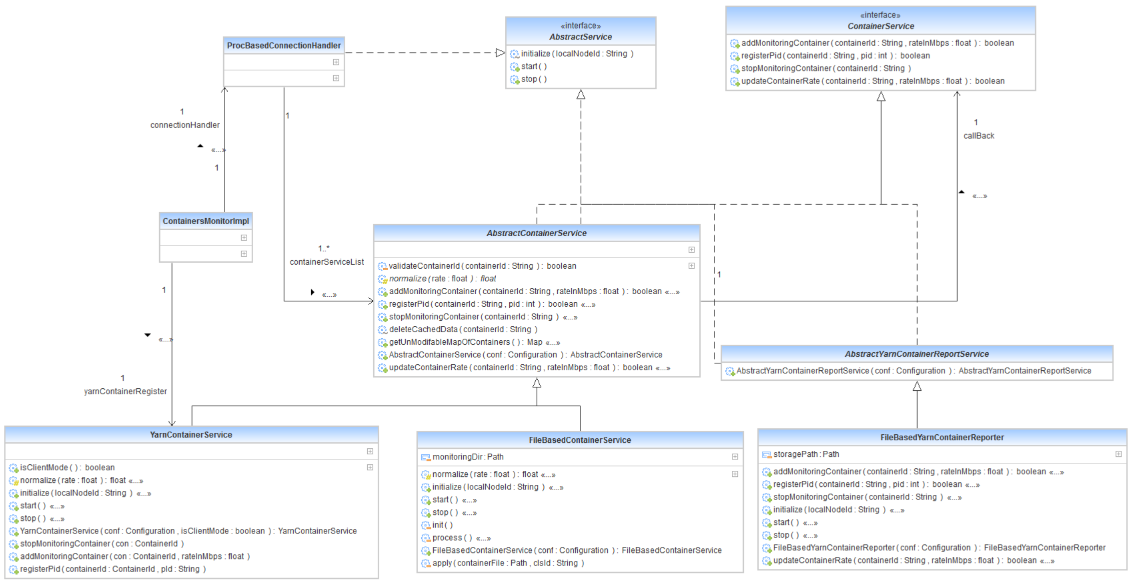

«interface»
AbstractService
initialize (localNodeId : String)
start ( )
stop ( )
«interface»
ContainerService
addMonitoringContainer (containerId : String , rateInMbps : float) : boolean
registerPid (containerId : String , pid : int) : boolean
stopMonitoringContainer (containerId : String)
updateContainerRate (containerId : String , rateInMbps : float) : boolean
ProcBasedConnectionHandler
1
connectionHandler
«...»
1
1
1
callBack
«...»
ContainersMonitorImpl
1..*
containerServiceList
«...»
AbstractContainerService
validateContainerId (containerId : String) : boolean
normalize (rate : float) : float
addMonitoringContainer (containerId : String , rateInMbps : float) : boolean «...»
registerPid (containerId : String , pid : int) : boolean «...»
stopMonitoringContainer (containerId : String) «...»
deleteCachedData (containerId : String)
getUnModifableMapOfContainers ( ) : Map «...»
AbstractContainerService (conf : Configuration) : AbstractContainerService
updateContainerRate (containerId : String , rateInMbps : float) : boolean «...»
1
1
«...»
1
yarnContainerRegister
AbstractYarnContainerReportService
AbstractYarnContainerReportService (conf : Configuration) : AbstractYarnContainerReportService
YarnContainerService
isClientMode ( ) : boolean
normalize (rate : float) : float «...»
initialize (localNodeId : String) «...»
start ( ) «...»
stop ( ) «...»
YarnContainerService (conf : Configuration , isClientMode : boolean) : YarnContainerService
stopMonitoringContainer (con : ContainerId)
addMonitoringContainer (con : ContainerId , rateInMbps : float)
registerPid (containerId : ContainerId , pId : String)
FileBasedContainerService
monitoringDir : Path
normalize (rate : float) : float «...»
initialize (localNodeId : String) «...»
start ( ) «...»
stop ( ) «...»
init ( )
process ( ) «...»
FileBasedContainerService (conf : Configuration) : FileBasedContainerService
apply (containerFile : Path , clsId : String)
FileBasedYarnContainerReporter
storagePath : Path
addMonitoringContainer (containerId : String , rateInMbps : float) : boolean «...»
registerPid (containerId : String , pid : int) : boolean «...»
stopMonitoringContainer (containerId : String) «...»
initialize (localNodeId : String) «...»
start ( ) «...»
stop ( ) «...»
FileBasedYarnContainerReporter (conf : Configuration) : FileBasedYarnContainerReporter
updateContainerRate (containerId : String , rateInMbps : float) : boolean «...»

Figure 6: Simplified UML diagram of interfaces/classes of container plugins

6 HdfsTrafficControl and Apache Hadoop Yarn framework

Beside the standalone mode, HdfsTrafficControl can work as a component of Yarn's NodeManager as illustrated in Fig. 7. Originally it was implemented as a prototype for our concept in [1].

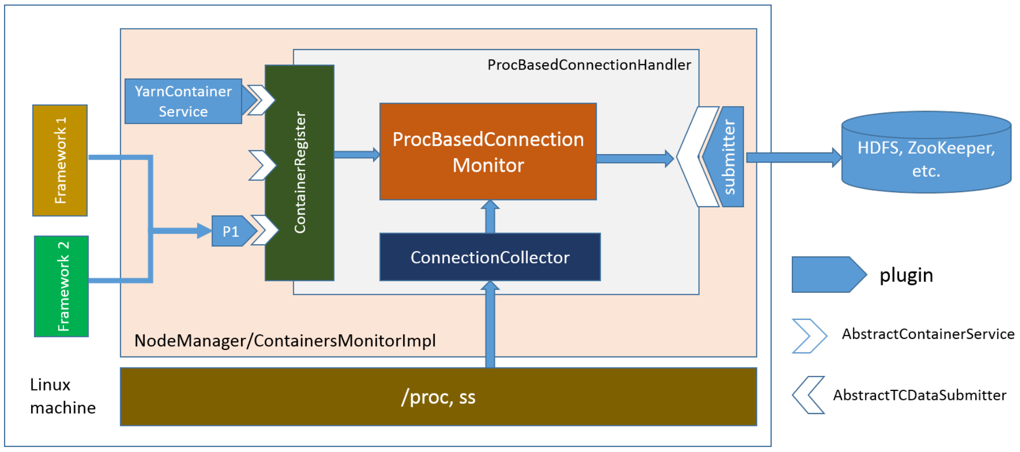


Figure 7: Connection Handler integrated into Yarn

In this mode, *ProcBasedConnectionHandler* is managed by the *ContainersMonitorImpl* component of YARN NodeManager. As it is embedded in NodeManager, the IPC between NodeManager and HdfsTrafficControl becomes inter-thread communication. Hence the new internal *YarnContainerService* plugin was introduced, and acts as a direct handler for Yarn containers. *YarnContainerService* is responsible for adding/removing Yarn containers, normalizing limit rates and reporting the pid of each monitoring container.

When Yarn acts as an external framework, instead of registering with *ContainerRegister*, *YarnContainerService* will try to load the report plugin implemented *AbstractYarnContainerReportService* as illustrated in Figs. 8. This plugin must be compatible one external plugin registered with *ContainerRegister*. For example, the current implementation also provides a *FileBasedYarnContainerReporter* plugin which can cooperate with the external

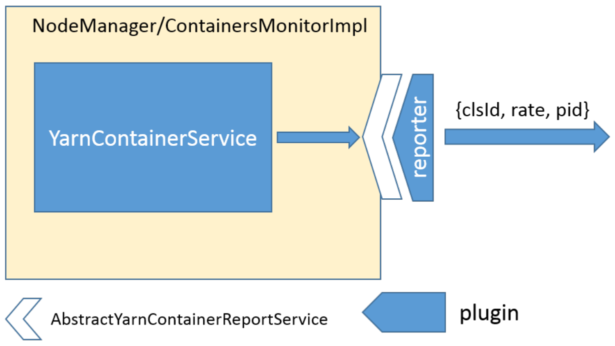

NodeManager/ContainersMonitorImpl
YarnContainerService
reporter
{clsId, rate, pid}
AbstractYarnContainerReportService
plugin

Figure 8: Yarn as a client of HdfsTrafficControl

FileBasedContainerService plugin as shown in Fig. 6.

TrafficController can be started in NodeManager as well. In this case it will be handled by *NodeStatusUpdaterImpl*.

7 Application and Experimental Results

7.1 Application scope

One important aspect can be considered is the application scope of the tool. Some possible use-cases:

- Yarn (with both modes), Mesos, Spark, etc.
- Any frameworks run on Mesos. As the "*container*" term here is just an object with a triple of {id, pid, rate} so very tools can take a role of "external frameworks" if they are able to report these data. It is very useful in Mesos in a private/hybrid cluster as it may be enough that the framework run on Mesos to do this task. It is easier than patching Mesos.
- Control read bandwidth of external applications like HDFS native shell.

7.2 Experimental Results

Fig. 9 shows the captured TCP throughput of TestDFSIO read with/without rate control. Note that the limiting rate setting in LTC is for the IP layer.

The benchmark was run with one split to read one file of 10GB with 1GB block size. The throughput was captured by the modified “nethogs” program.

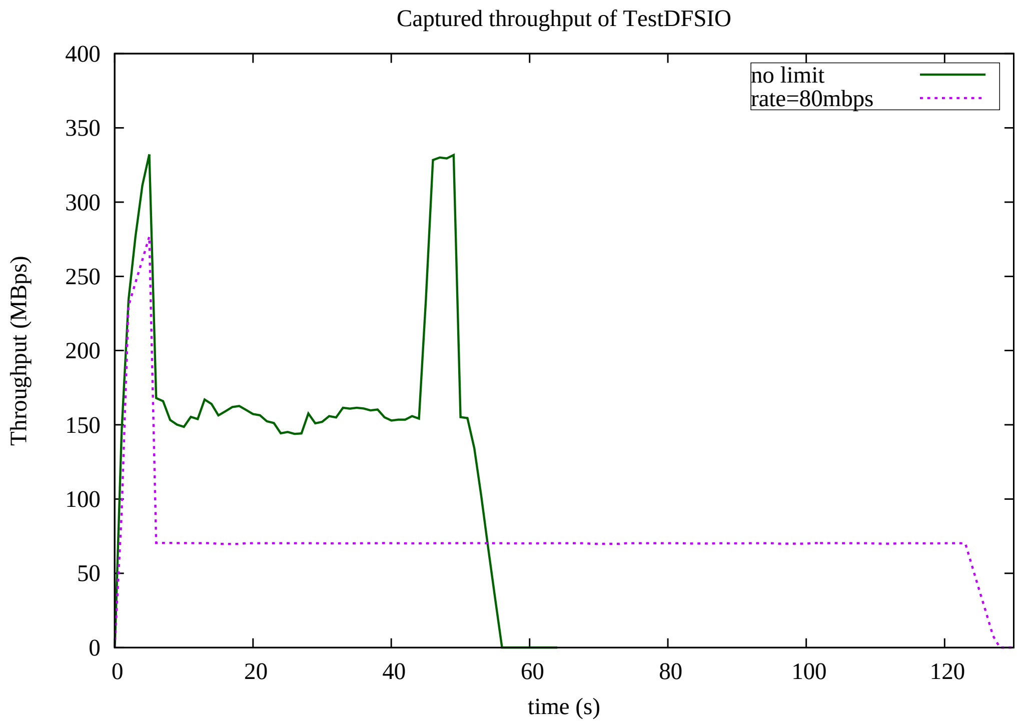


Figure 9: Capture throughput of TestDFSIO

For demonstration of controlling external programs/frameworks, a small Java program was written to scan */proc* file system periodically and find the pattern of HDFS native shell copy command (*'hdfs dfs -copyToLocal'*). If the command is detected, it reports the pid of the copy process, the predefined rate and an unique container id to *HdfsTrafficControl*.

Fig. 10 depicts the captured TCP throughput with/without rate control when one file of 10GB with 1GB block size was copied to the local disk using *'hdfs dfs -copyToLocal'* command.

It can be seen that the HDFS read rates are controlled in both cases of TestDFSIO and HDFS native shell copy command.

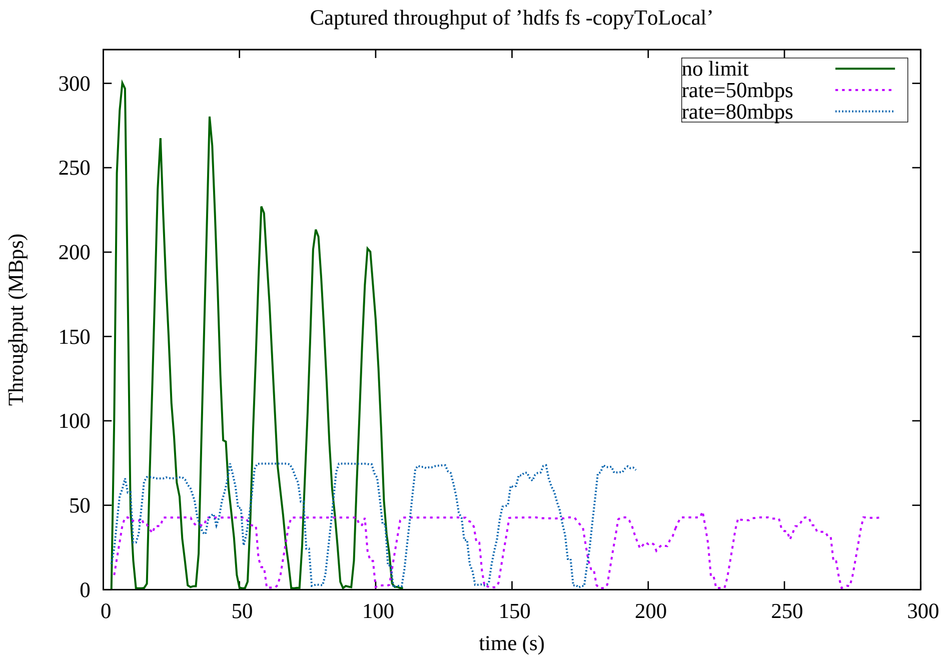


Figure 10: Capture throughput of command 'hdfs dfs -copyToLocal'

Appendix: Configuration settings

A script with following content can be used for starting HdfsTrafficControl as a standalone application:

```
export CLASSPATH=‘yarn classpath‘
java -cp $CLASSPATH PREFIX.HdfsTrafficControl
```

where PREFIX denote “org.apache.hadoop.yarn.server.nodemanager.trafficcontrol”. Only *TrafficControler* can be started by adding the option “-*onlyExecutor*”

The code and test of the implementation of HdfsTrafficControl is found in org.apache.hadoop.yarn.server.nodemanager.trafficcontrol and underlying packages of Yarn nodemanager sub-project.

Regarding the integration with Yarn NodeManager some other classes of Yarn were modified in order to specify the HDFS limit rate in the request of jobs beside *ContainersMonitorImpl* and *NodeStatusUpdaterImpl*.

The following configurations are available:

7.3 Global configurations

- **yarn.nodemanager.hdfs-bandwidth-enforcement.port**
  - The monitoring port
  - Default value: 50010
  - Location: yarn-site.xml
- **yarn.nodemanager.hdfs-bandwidth-enforcement.devices**
  - The comma separated list of NIC devices to apply TC settings
  - Default value: lo
  - Location: yarn-site.xml
- **yarn.nodemanager.hdfs-bandwidth-enforcement.check-tc-config-interval**
  - The time interval in ms for monitoring/collecting connections
  - Default value: 1000

  - Location: yarn-site.xml

- **yarn.nodemanager.hdfs-bandwidth-enforcement.container-plugins**
  - The comma separated list of custom filters for reporting external containers. One possible plugin is PREFIX.impl.FileBasedContainerService
  - Default value: empty
  - Location: yarn-site.xml

- **yarn.nodemanager.hdfs-bandwidth-enforcement.execute-sudo-ss**
  - Whether the user run nodemanager can execute 'sudo ss' without password
  - Default value: false
  - Location: yarn-site.xml

- **yarn.nodemanager.hdfs-bandwidth-enforcement.submitter.class**
  - The name of submitter class for back-end storages. The possible values are PREFIX.impl.HdfsTCDataSubmitter and PREFIX.impl.ZkTCDataSubmitter
  - Default value: PREFIX.impl.HdfsTcDataSubmitter
  - Location: yarn-site.xml

- **yarn.nodemanager.hdfs-bandwidth-enforcement.collector.class**
  - The name of collector class for back-end storages. The possible values are PREFIX.impl.HdfsTCDataCollector and PREFIX .impl.ZkTCDataCollector/ZkAsyncTCDataCollector
  - Default value: PREFIX.impl.HdfsTCDataCollector
  - Location: yarn-site.xml

- **yarn.nodemanager.hdfs-bandwidth-enforcement.zk-server.address**
  - The address of the ZooKeeper server, required for ZooKeeper back-end storage
  - Default value: empty

  - Location: yarn-site.xml

- **yarn.nodemanager.hdfs-bandwidth-enforcement.config-root.path**
  - The root directory to store HDFS connection data in the case of HDFS back-end storage. It is required in case of multiple users.
  - Default value: $/user/user\_start\_yarn/hdfs-bandwidth-enforcement$
  - Location: yarn-site.xml

- **yarn.nodemanager.hdfs-bandwidth-enforcement.container-local-data.path**
  - The monitoring folder in case of PREFIX.impl.FileBasedContainerService
  - Default value: $/tmp/monitoring\_containers$
  - Location: yarn-site.xml

7.4 Configurations for Yarn NodeManager

- **yarn.nodemanager.hdfs-bandwidth-enforcement.enable**
  - Whether we can active this feature in the NodeManager.
  - Default value: false
  - Location: yarn-site.xml

- **yarn.nodemanager.hdfs-bandwidth-enforcement.client_mode**
  - Whether it is started in the client mode as an external framework.
  - Default value:false
  - Location: yarn-site.xml

- **yarn.nodemanager.hdfs-bandwidth-enforcement.enable-submitter-only**
  - When it is not in the client mode then it indicates whether we should start the TrafficController component in the embedded mode.
  - Default value: false

- Location: yarn-site.xml

- **yarn.nodemanager.hdfs-bandwidth-enforcement.container_report_service.class**
  - The plugin to be loaded to report Yarn containers to the standalone HdfsTrafficControl. One possible plugin is PREFIX. impl.FileBasedYarnContainerReporter
  - Default value: empty
  - Location: yarn-site.xml
- **yarn.nodemanager.hdfs-bandwidth-enforcement.minimum-rate-mbps**
  - Minimum HDFS read rate
  - Default value: 0
  - Location: yarn-site.xml
- **yarn.nodemanager.hdfs-bandwidth-enforcement.maximum-rate-mbps**
  - Maximum HDFS read rate
  - Default value: 0
  - Location: yarn-site.xml
- **mapreduce.map.bandwithlimit.mbps**
  - The limit rate for the map tasks
  - Default value: 0
  - Location: mapred-site.xml
- **mapreduce.reduce.bandwithlimit.mbps**
  - The limit rate for the reduce tasks
  - Default value: 0
  - Location: mapred-site.xml

References

[1] T. V. Do, B. T. Vu, H. N. 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*, pages 23–29, Feb 2015.