2014 43rd International Conference on Parallel Processing

SMARTH: Enabling Multi-pipeline Data Transfer in HDFS

Hong Zhang[1], Liqiang Wang[1], and Hai Huang[2]

[1]Department of Computer Science, University of Wyoming. {hzhang12,lwang7}@uwyo.edu
[2]IBM T.J. Watson Research Center. haih@us.ibm.com

***Abstract*—Hadoop is a popular open-source implementation of the MapReduce programming model to handle large data sets, and HDFS is one of Hadoop's most commonly used distributed file systems. Surprisingly, we found that HDFS is inefficient when handling upload of data files from client local file system, especially when the storage cluster is configured to use replicas. The root cause is HDFS's synchronous pipeline design. In this paper, we introduce an improved HDFS design called SMARTH. It utilizes asynchronous multi-pipeline data transfers instead of a single pipeline stop-and-wait mechanism. SMARTH records the actual transfer speed of data blocks and sends this information to the namenode along with periodic heartbeat messages. The namenode sorts datanodes according to their past performance and tracks this information continuously. When a client initiates an upload request, the namenode will send it a list of "high performance" datanodes that it thinks will yield the highest throughput for the client. By choosing higher performance datanodes relative to each client and by taking advantage of the multi-pipeline design, our experiments show that SMARTH significantly improves the performance of data write operations compared to HDFS. Specifically, SMARTH is able to improve the throughput of data transfer by 27-245% in a heterogeneous virtual cluster on Amazon EC2.**

I. Introduction

The amount of generated and stored data has been growing rapidly, and the rate of the growth is only increasing. It is estimated that 2.5 quintillion bytes of data are generated every day, and 90% of the data in the world today has been created in the last two years [1]. How to solve these big data issues has become a hot topic in both industry and academia.

Apache Hadoop [2] is a popular open-source implementation of the MapReduce programming model to handle large data sets. Hadoop hides the complex details of parallelization, fault tolerance, data distribution, and load balancing. It is designed to be deployed onto commodity hardware ranging from a few nodes to thousands or more. Hadoop has two main components: MapReduce and Hadoop Distributed File System (HDFS). MapReduce paradigm is composed of a Map function that performs filtering and sorting of input data and a Reduce function that performs a summary operation. HDFS is a distributed, scalable, and portable file system written in Java for the Hadoop framework. There are some major differences from other distributed file systems, *e.g.*, highly fault-tolerant, and can be easily deployed on low-cost hardware.

HDFS contains a single namenode that manages the entire file system, and one or more datanodes to serve read and write requests from client systems. HDFS assumes that all nodes in a cluster are homogeneous and can process requests with similar speed. However, in real world, the performance of (*e.g.*, network, disks, and CPU) nodes could be different from one another due to various reasons, *e.g.*, different generations of hardware, different virtual resource allocation, resource contention in virtualized environments, *etc*. We found that this disparity in performance amongst datanodes within an HDFS cluster can significantly hamper its write performance.

In this study, we propose an asynchronous multi-pipeline file write protocol to replace the traditional stop-and-wait protocol in HDFS. Instead of transferring data blocks one by one and waiting for *ACK* (acknowledgement) packets from all datanodes involved in the transmission, SMARTH (Smart HDFS) builds a new pipeline after it finishes sending the current block to the first datanode in the pipeline so that it can start sending the next data block right away. This new design makes better use of the network capacity of the client as well as the datanodes' accessing bandwidth within the cluster. In order to minimize the time of the file importing process, we introduce a flexible sorting algorithm of datanodes based on real-time and historical datanode accessing status (including network and storage I/O). We employ the heartbeat mechanism to report the data transmission speed on each client to the namenode every three seconds. Based on the collected information, the namenode can give a good estimate of which set of datanodes a client should use for best performance. When the replication factor of an HDFS cluster is greater than one, which is often used in production environments, we optimize the way that a client interacts with each of the datanodes in a pipeline to allow additional parallelism in data transfer. However, this also changes the way that HDFS ensures data fault tolerance, and thus, we revise its fault tolerance method so that the new way is compatible with the asynchronous multi-pipeline protocol.

We simulate various network conditions using bandwidth throttling on Amazon EC2, we demonstrate that the asynchronous multi-pipeline algorithm is able to remove the single pipeline barrier and effectively overlap data transfer in different pipelines for HDFS file write operations. Overall, SMARTH is able to improve the throughput of data transfer

0190-3918/14 $31.00 © 2014 IEEE
DOI 10.1109/ICPP.2014.12

CPS Conference Publishing Services

by 27- 245% in a heterogeneous virtual cluster of Amazon EC2.

The contributions of this paper are: (1) introducing an innovative asynchronous data transmission approach to greatly improve the write operation's performance in HDFS, (2) supporting flexible sort of datanode pipelines based on real-time and historical datanode accessing condition, and (3) providing a comprehensive fault tolerance mechanism under this asynchronous transmission approach.

The rest of this paper is organized as follows. Section II discusses background on file write in Hadoop distributed file system. We then describe the multi-pipeline design of SMARTH in Section III focusing on performance, followed by Section IV that details the accompanying fault tolerance algorithm. Section V shows experimental results. Section VI provides a review of related work. Conclusions and possible future work are summarized in Section VII.

II. BACKGROUND

An HDFS cluster is comprised of a namenode and one or more datanodes. In this section, we give a comprehensive analysis about how a client communicates with the namenode and datanodes when uploading data to HDFS. As shown in Figure 1, there are 6 steps to upload data from a local file system into HDFS.

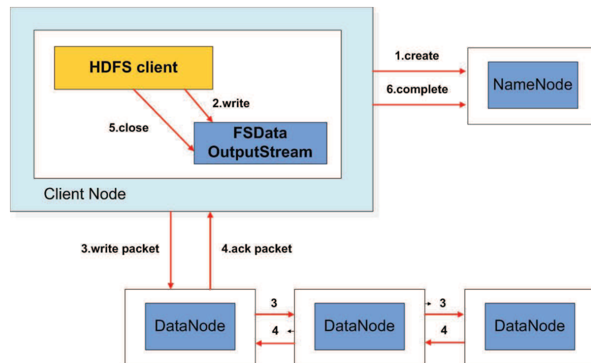


Figure 1. Workflow of an HDFS file write operation.

1) **Creating a file into the file system's namespace.** The client first makes a `create()` HDFS call, which results in a `ClientProtocol` RPC being invoked to create a new file on the namenode. Before the creation of the file in the namespace, the namenode conducts several checks, *e.g.*, whether the file already exists, whether the user has the right to create the file, and whether safe mode is disabled. If all these checks pass, the namenode would create the corresponding file in the file system's namespace; otherwise it would throw an exception.
2) **Splitting data into packets and inserting into a data queue.** To write data to HDFS, client applications consider the data file as a standard output stream. This data stream is fragmented into blocks, each of which has a default size of 64MB. In turn, each block is split into 64KB packets by default when being transmitted onto the network. When the client writes a new block, a `DataStreamer` thread would send an `addBlock()` call to the namenode to ask for a new block ID and the datanode IDs to store the block. After the corresponding packets are generated, the client sends these packets to a FIFO queue and then to the datanodes.
3) **Sending packets to Datanodes.** `DataStreamer` uses the datanode IDs to build a pipeline between the client and these datanodes, streams the packets to the first datanode in the pipeline one by one, and stores these packets into another queue called ACK queue in case some datanodes require retransmitting due to packet loss. When the first datanode receives a packet, it verifies the packet's checksum, stores the packet, and transfers it to the next datanode in the pipeline. This procedure will repeat until the packet reaches the last datanode at the end of the pipeline.
4) **Sending acknowledgement (ACK) back to the client.** When the last datanode obtains the packet, it would send an ACK through the pipeline in a reverse order. The client has a thread called `PacketResponder` that is responsible for receiving response ACKs. If the `PacketResponder` thread receives a packet ACK from all datanodes, it removes this packet from the ACK queue.
5) **Closing the output stream.** When the client has flushed all data into the output stream, it calls `close()` on the stream, and waits for all packets' ACKs.
6) **Completing file write.** When all packets' ACKs are received by the `PacketResponder` thread, it wakes up the client. The client would send a complete signal to the namenode to complete this file write operation.

In Steps 3 and 4, the client has to wait until it received all ACKs through the pipeline, during which the client could not optimally make use of network capacity. In other words, only one pipeline is utilized even though there are many other available nodes in the cluster. This motivates us to design a new protocol to better exploit the network bandwidth of the client node and within the HDFS cluster.

III. DESIGN AND IMPLEMENTATION

In this section, we introduce SMARTH and compare it with HDFS. First we use a common scenario to illustrate how existing HDFS design cannot take advantage of the full network bandwidth of the client node when data is uploaded into an HDFS cluster. This motivates us to design an asynchronous block transmission scheme to overcome the existing limitations.

A. Design of Asynchronous Multi-pipeline Protocol

In the original HDFS design, when a client wants to write a data block to an HDFS cluster, it receives a list of datanodes from the namenode to form a pipeline. The data block travels from the client to each of the datanodes sequentially, and the client will only mark a block as completed when the ACK packets from all the datanodes in the pipeline are received. Therefore, the effective bandwidth of the pipeline is limited by the slowest datanode in the pipeline.

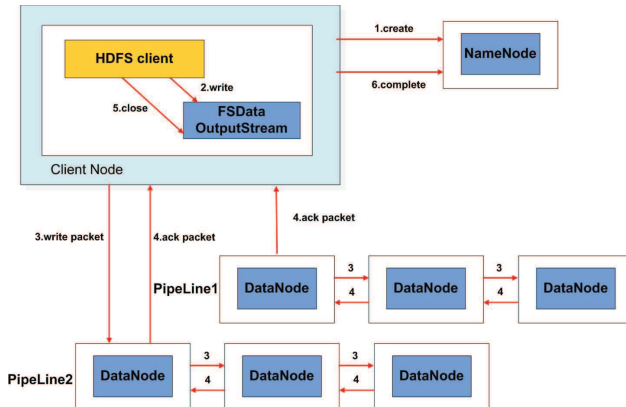


Figure 2. Workflow of a SMARTH file write operation.

The way that SMARTH handles data write operations is shown in Figure 2. Step 1 is similar to Hadoop. When the client writes a block, it first asks for a block ID and a list of datanodes to store the data. The SMARTH namenode then chooses a high-bandwidth node relative to the client as the first datanode in the pipeline (based on historical information, which we will describe later). In step 2, the client splits data blocks into same size packets and puts them into a data queue.

During data transmission, the client sends these packets to the first datanode, and after storing them locally, this datanode forwards them to the second datanode and so on and so forth until the last datanode receives all the packets (step 3). When the first datanode receives all the packets of a certain block, it sends back a special ACK packet called `FIRST_NODE_FINISH_ACK (FNFA)` to the client. This packet indicates to the client that the entire block has been received and stored by the first datanode in the pipeline. Instead of waiting for ACKs from the other datanodes, the client continues to send the next data block by requesting another block ID and datanodes from the namenode. This results in a new pipeline being formed for sending the next data block. Additional pipelines can be formed if the client can send packets to the first datanode quicker than the speed that the packets travel to the other datanodes in the pipeline.

After creating a pipeline, we create an ACK queue and a `PacketResponder` thread for it. Each pipeline transfers ACKs back to the corresponding `PacketResponder` thread (step 4). As the `PacketResponder` thread receives an ACK from its pipeline, it removes the corresponding packet from its ACK queue. At the client, we use a set to enumerate all active pipeline objects. When the `PacketResponder` thread receives all ACKs, it will be removed from this set. When the pipeline set is empty, we close output stream (step 5) and complete this uploading (step 6).

Using this method to upload files to HDFS, the client can fully utilize its bandwidth capacity and reduce idle time on waiting for ACK messages. Thus, the speed of the asynchronous pipeline transmission is now determined not by the minimum bandwidth amongst client and datanodes but the network speed between the client and the first datanode in the pipeline. In the following subsections, we describe how SMARTH namenode finds the "best" first datanode for each client while keeping the cluster balanced.

B. Global Optimization for Data Transmission

Traditional HDFS represents a network topology as a tree structure [2]. When the client requests a list of datanodes for storing a data block, the namenode chooses the target nodes according to this network topology tree, *e.g.*, to optimize for performance and to maximize data fault tolerance. However, it cannot accurately capture the real-time network condition as this information is usually not directly correlated with the network topology.

In SMARTH, client records the transmission speed of data blocks to all the first datanodes in transfer pipeline that it had communicated before and sends these records to the namenode every three seconds by remote procedure calls (RPCs), following the default heartbeat mechanism in Hadoop. When the client subsequently requests datanodes to place additional data blocks, the namenode utilizes this information to choose a set of best performing datanodes in the cluster according to our global optimization algorithm shown below.

Algorithm 1 describes SMARTH namenode's global optimization algorithm for choosing datanodes. When the namenode receives a request to upload files from a client, it calculates the maximum number of pipelines allowed for the client, and assign it to a variable n. Our design selects a datanode randomly from the n best performing nodes for this client as the first datanode so that we can guarantee the bandwidth between the client and the first datanode is relatively higher in the pipeline. The second replica is selected from a different rack and the third replica is placed on the same rack as the second.

C. Local Optimization for Data Transmission

Since network status varies all the time, we utilize a local optimization algorithm to sort the datanodes order in pipeline by the newly records and give a chance to test

Algorithm 1 Algorithm for global optimization

1: num = the number of active datanodes
2: $repli$ = the number of replica factor
3: n = num / $repli$ // *the maximum pipeline size*
4: **if** (namenode has transmission records for the client) **then**
5: $TopN$ = top n datamodes in terms of transfer speed
6: // the number of datanodes we have choosen
7: $results$ = 0
8: **while** ($results$!= repli) **do**
9: **if** ($results$ == 0) **then**
10: $targets$[0] = randomDatanode($TopN$)
11: **else if** (results == 1) **then**
12: $targets$[1] = randomRemoteRackNode()
13: **else if** (results == 2) **then**
14: $targets$[2] = nodeOnSameRack($targets$[1])
15: **else**
16: $targets$[$results$] = randomDatanode()
17: **end if**
18: $results$++
19: **end while**
20: **else**
21: $targets$ = employ the original HDFS method to select datanodes
22: **end if**

the bandwidth performance of nodes with poor performance previously.

Algorithm 2 Algorithm for local optimization

1: $repli$ = the number of replica factor
2: $TransSpeedVector$ = the transmission speed of every nodes in $targets$
3: sort $targets$ in descending order by $TransSpeedVector$
4: r = a random number between 0 to 1
5: **if** (r ¿ $threshold$) **then**
6: *//the target index to switch the first datanode*
7: $index$ = a random integer between 1 to $repli - 1$
8: swap($targets$[0], $targets$[$index$])
9: **end if**

Algorithm 2 shows details of local optimization algorithm executes in the client node. We use block transfer records locally to calculate the transmission speed for each datanodes assigned to $TransSpeed$, and employ sort algorithm to reorder the targets set. We calculate a random number r between 0 to 1 to decide whether to swap the first datanode with another datanode in pipeline so that we can update the transmission records of that node. In this way, we may keep transmission information for all datanodes updated occasionally. In our algorithm, if r is greater than $threshold$ that is assigned to 0.8, we use another random integer $index$ to choose which datanode to switch with the first one.

D. Cost-Benefit Analysis

To pinpoint how SMARTH outperforms HDFS, we analyze a file write operation in details and compare the two designs step by step. Data transfer between a client and datanodes for the original HDFS is shown in Figure 3. When the number of replica is greater than one, datanodes will forward each packet to the next datanode along the pipeline until the last datanode receives it. Client will wait for ACKs from all datanodes in the pipeline before it can start sending the next data block.

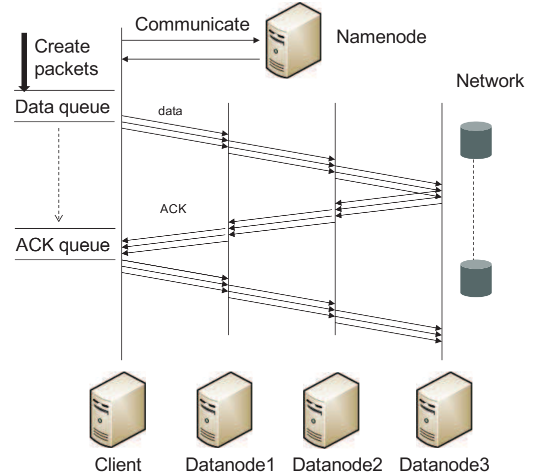


Figure 3. Data Transmission of HDFS

Assume that data file size is D, block size is B, and data file size is greater than one block, the file is split into $\lceil D/B \rceil$ blocks. Assume that the packet size is P, the number of packets transferred is $\lceil D/P \rceil$. Let T_n denote the communication time between client and namenode for each block. Let T_c denote the average production time (read data from local file, compute the checksum and append the data and checksum to a packet) for a packet by the client. When the datanode receives a packet, it verifys the checksum and writes the packet to the local disk that takes T_w on average. Let B_{min} represent the minimum bandwidth between client and the first datanode and amongst adjacent datanodes. Since the size of ACK packets is smaller than the data packets, and the time of transferring ACKs and the time of sending data packets overlaps, we only need to take the packet transmission time into account.

As the production and the transmission of packets are executed by different threads, there is an overlap between the production time and the transmission time of packets. If the average production time of packets is greater than or equal to the average packet transmission time along the pipeline,

there is no packet waiting for sending on data queue. The total production time for all packets is the major factor to the whole importing time. In this scenario, $T_c >= P/B_{min}$, and the total time consuming is shown in Formula (1). However, even in the small instance, to produce a packet is very fast compared with the speed to send a packet in our experiments.

$$T = T_n * \ulcorner D/B \urcorner + (T_c + T_w) * \ulcorner D/P \urcorner \quad (1)$$

If the packet production time is less than the packet transmission time, there must exist blocking on data queue. So the total cost relies on the minimum bandwidth amongst client and datanodes. In this scenario, $T_c < P/B_{min}$, and Formula (2) shows the total time consuming.

$$T = T_n * \ulcorner D/B \urcorner + (P/B_{min} + T_w) * \ulcorner D/P \urcorner \quad (2)$$

From the analysis above, we know that the time of importing file is determined by the production time or the transmission time of packets, depending on which is larger.

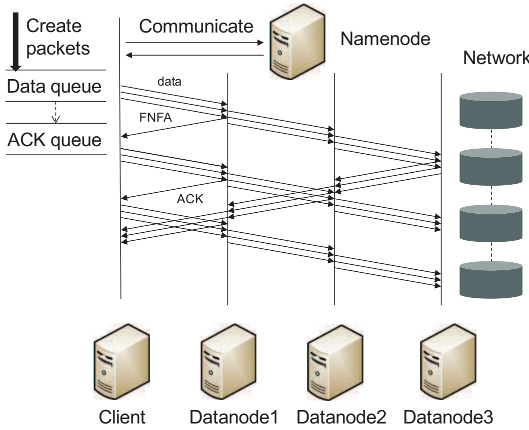


Figure 4. Data transmission of SMARTH

Figure 4 shows the process of data transmission for SMARTH. When the first datanode receives all packets of a block, it sends a FNFA back to the client, and the client can then create a new pipeline to prepare for transmitting the next block. Assume the bandwidth between the client and the first datanode is B_{max}. If the average production time of packets is greater than or equal to the packet average transfer time from the client to the first datanode ($T_c >= P/B_{max}$), the speed of seeding a packet is slower than the speed of producing a packet. Then there is no blocking in data queue, and the total time consuming is as Formula (1) shown. If the average production time of packets is less than the packet average transfer time from the client to the first datanode ($T_c < P/B_{max}$), the total cost relies on the bandwidth between the client to the first datanode as Formula (3) shows.

$$T = T_n * \ulcorner D/B \urcorner + (P/B_{max} + T_w) * \ulcorner D/P \urcorner \quad (3)$$

It is obvious that B_{max} is greater than or equal to B_{min}. So our improved HDFS is more efficient than the existing one. We can also find out that idle time waiting for ACK is reduced when we compare Figure 3 with Figure 4.

IV. Fault Tolerance

A. Fault Tolerance For Original HDFS

Since Hadoop is often deployed on a large cluster of commodity nodes, being able to automatically handle faults is a crucial part of its design. This section provides an overview of the fault tolerance mechanism in original HDFS and then discusses our own fault tolerance approach for the multi-pipeline design in SMARTH.

Algorithm 3 Algorithm for fault tolerance of HDFS

```
1:  checks the validity of parameters
2:  close all streams related to the block
3:  moves all packets in ACK queue back to data queue
4:  success = false
5:  while (!success) do
6:      if (targets is not empty) then
7:          return an exception
8:      else
9:          primaryNode = the first datanode in targets
10:         add new datanodes to replace error nodes in targets
11:         success=recoverBlock(primaryNode, targets)
12:         if (!success) then
13:             remove primaryNode from targets
14:         end if
15:         recreate block streams
16:     end if
17: end while
18: recreate ResponseProcessor thread
```

Algorithm 3 shows how a typical process handles errors during uploading files to HDFS. When the client catches an error in the process of transmitting a block, it first checks the validity of parameters, and closes all streams related to the block. Then it moves all packets in ACK queue back to data queue. It picks the primary datanode from active datanodes in pipeline, and uses it to recover the other datanodes. If fails, picks another primary datanode and recover again until recovering the block successfully or throwing an exception. At the end, the client recreates the `ResponseProcessor` thread for receiving remaining ACKs.

B. Fault Tolerance for Multi-Pipelines

Since we employ an asynchronous multi-pipeline design, we need to replace the original fault tolerance mechanism with a new design.

Algorithm 4 Algorithm for fault tolerance of SMARTH

1: stop the current block transfer
2: moves all packets in ACK queue back to data queue
3: **while** ($errorPipelineSet$ is not empty) **do**
4: recover one error pipeline as Algorithm 3
5: remove the error pipeline from $errorPipelineSet$
6: **end while**
7: start transferring the interrupted block

Algorithm 4 shows our approach to handle the multi-pipeline fault tolerance. When an error occurs in a pipeline, SMARTH adds the error pipeline into an error pipeline set. It firstly stops the current block sending, and starts a recovery process to recover error pipelines in this set. Each pipeline's recovery process is similar to the original single pipeline recovery of HDFS. If the error pipeline is recovered, we delete the error pipeline from the error pipeline set. We continue recovering error pipeline until the error pipeline set is empty, then the client restart sending the interrupted block.

C. Buffer Overflow Problem

In SMARTH, since we employ global optimization and local optimization, the first data node is always a high bandwidth node compared with other datanodes. So the client can send data to the first datanode quickly, but the first datanode cannot send packets quickly to the second datanode. Therefore it is possible that the buffer in the first datanode overflows. When the size of data file is large, and the bandwidth varies considerably from node to node, the buffer of the high bandwidth nodes has higher chance for overflow.

We limit the pipeline size to a maximum number (the cluster size / the number of replica), and if a datanode is already in a pipeline, it cannot be added into other pipelines created by the same client. Then each datanode belongs to only one pipeline, and its buffer is set to be 64 MB, *i.e.*, the default size of block, for each client.

V. Experiments

A. Experiment Setup

The study was conducted using Amazon EC2's compute instances. Amazon EC2 supports servers of different types such as small, medium, and large instances. These instances differ in the number of cores, the memory allocated to them, bandwidth, and price (see Table I). An Elastic Compute Unit (ECU) is an EC2-specific unit to express the computational performance of a CPU core. 1 ECU is the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or Xeon processor.

Instance Type	Memory	ECUs	Network
Small	1.7 GB	1	$\approx 216Mbps$
Medium	3.75 GB	2	$\approx 376Mbps$
Large	7.5 GB	4	$\approx 376Mbps$

Table I
AMAZON EC2 INSTANCE TYPES

We use four different clusters in our evaluations. Three of the clusters are homogeneous consisted of one namenode and nine datanodes, i.e., of small, medium, or large instances. The other cluster is heterogeneous consisted of 3 small, 4 medium, and 3 large instance nodes, where one medium instance is the namenode and the others are datanodes. Each node runs CentOS Linux Server 6.2 with kernel 2.6.32-220, and the original Apache Hadoop version 1.0.3. We use Amazon EC2 ephemeral storage to store our data file.

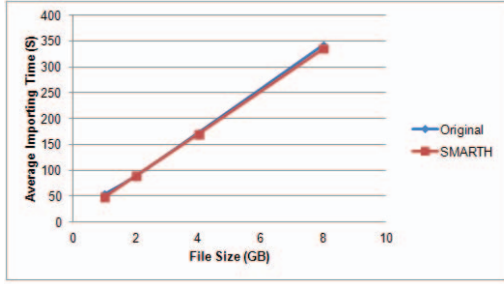
B. Experimental Results

Our goal in this study is to evaluate the impact of various network conditions on both HDFS and SMARTH. We employ a Linux utility called `tc`, which is used to control network traffic, to limit both ingress and egress bandwidth between VMs.

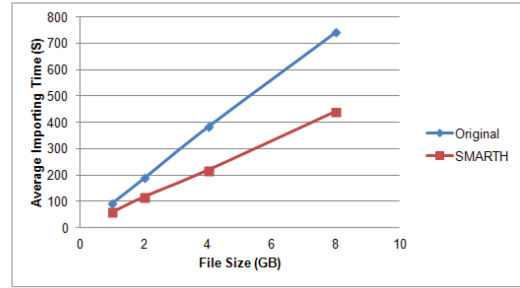
1) Two-Rack Cluster Scenario: For a large cluster, a common practice is to employ its nodes across multiple racks, or even across multiple data centers, for load balancing and fault tolerance reasons. Network bandwidth between nodes in the same rack is often greater than the bandwidth between nodes across racks. To consistently simulate this behavior (as EC2 does not expose VM's physical location), we throttle the network bandwidth of nodes using `tc`.

The default strategy of HDFS is to place the first replica on the client node itself if the client is a datanode; otherwise, the namenode picks nodes that are not too full or busy. The second replica is placed on a different rack from the first and the third is placed on the same rack as the second, but on a different node. Although this strategy offers a good reliability, it is at the cost of performance.

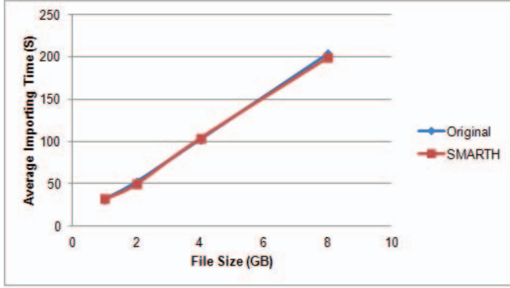
In our experiments, our file sizes vary from 1GB to 8 GB, and we measure the time to upload the files in both original HDFS and SMARTH using an HDFS put command. We have performed these experiments on small, medium, and large instances. Figure 5(a) and Figure 5(b) show that the file size is proportional to the time consumed when importing file to HDFS and SMARTH in small cluster without and with bandwidth throttling of 100 Mbps between two racks. The same conclusion can be also found in medium and large instances from Figures 5(c) and 5(d), Figures 5(e) and 5(f). Due to these results, we only consider the input file size



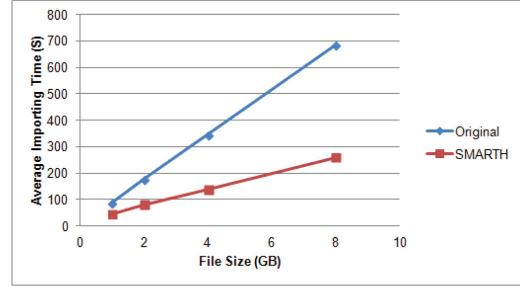
(a) default bandwidth in small cluster



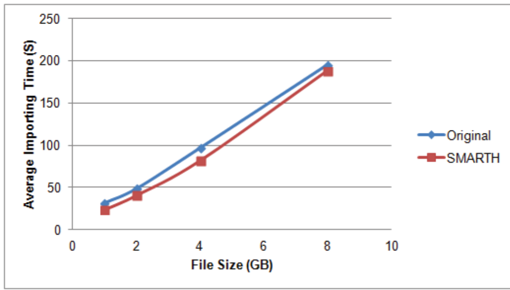
(b) bandwidth throttling in small cluster



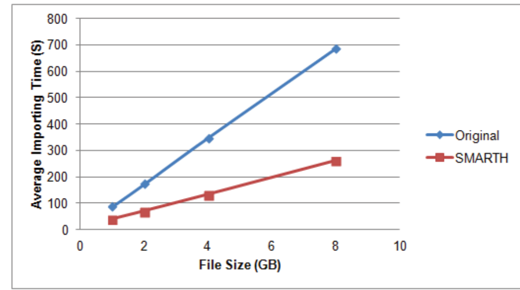
(c) default bandwidth in medium cluster



(d) bandwidth throttling in medium clustere



(e) default bandwidth in large cluster



(f) bandwidth throttling in large cluster

Figure 5. Comparison of uploading time on different clusters with and without network throttling.

is 8 GB in the rest of the paper when we measure the performance of HDFS and SMARTH.

Figures 5(c) and 5(e) as well as Figures 5(d) and 5(f) also show that the file importing performance of large cluster is roughly the same with the performance of medium cluster. That is because the medium cluster and large cluster have the same networking capacity. Figures 5(a), 5(c), and 5(e) show that there is no big gain if the cluster's network status is homogeneous, where network is in the default bandwidth and without throttling.

Figure 6 shows the file write times on Hadoop and SMARTH when we throttle the network to different bandwidth in a small cluster. As Figure 6 shows, the more we throttle the network, the better the performance of SMARTH is compared to HDFS. The new design of SMARTH gains an improvement of 130% when the bandwidth throttling is at 50 Mbps; even when the bandwidth throttling is 150 Mbps, the performance can improve about 27%. We have

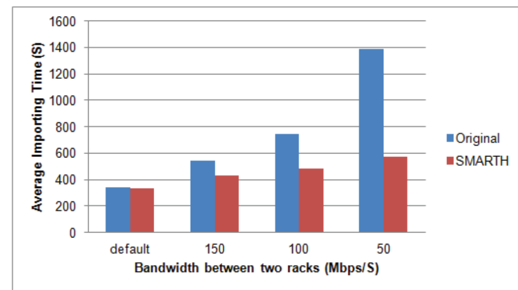


Figure 6. Comparison of small instances' uploading time when throttled bandwidth between two racks varies.

measured the file write speed in medium and large clusters and observe similar big gains. Figure 7 and Figure 8 show that SMARTH achieves an improvement of 225% in medium cluster and outperforms HDFS by 245% in large cluster when the network bandwidth is throttled to 50 Mbps.

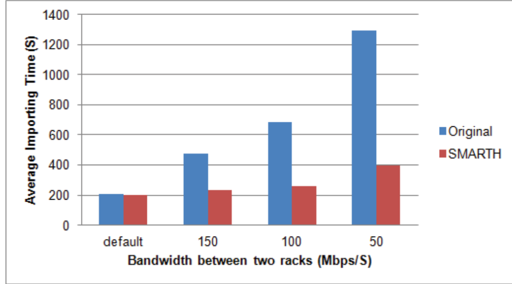


Figure 7. Comparison of medium instances' uploading time when throttled bandwidth between two racks varies.

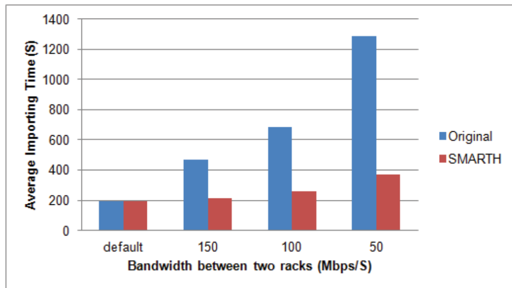


Figure 8. Comparison of large instances' uploading time when throttled bandwidth between two racks varies

Figure 9 shows the relationship between how much we throttle the network bandwidth of nodes in small, medium and large clusters and the improvement of SMARTH. The benefit of our design depends on the extent of bandwidth throttling between two racks. When the network bandwidth between nodes in the same rack is much greater than the network bandwidth between nodes in different ranks, SMARTH can gain more benefit. In a large cluster, where nodes are often allocated in different data centers, network performance between a pair of nodes can vary even more significantly within the cluster.

2) Bandwidth Contention Scenario: In the real world, the bandwidth between nodes in the same rack still varies all the time, and some other procedures also can occupy

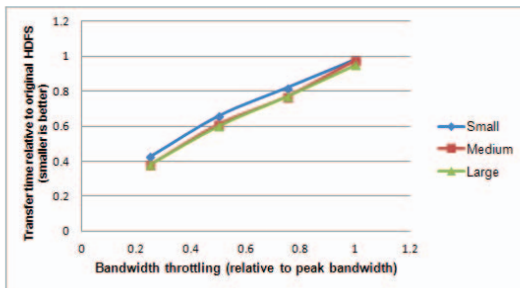


Figure 9. Relationship between bandwidth throttling and performance improvement.

the bandwidth and contend with Hadoop program. In this scenario, if some nodes with lower network capacity are selected as datanodes to transfer blocks, they can degrade the performance of file write. In SMARTH, we would select the faster nodes as the first datanode and when the first datanode receives the full block, the client builds a new pipeline to continue the file write in order to avoid the idle wait time of the client network and make the best use of the bandwidth between the client and datanodes.

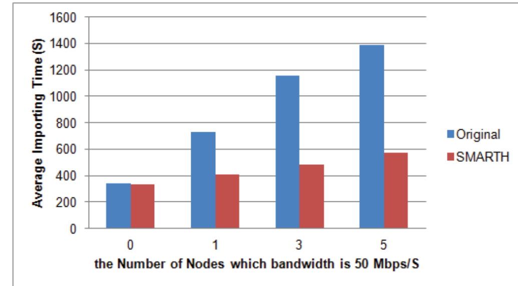


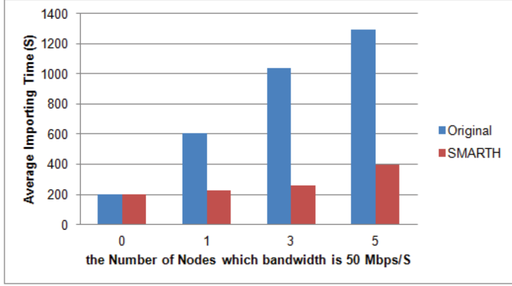
Figure 10. Comparison of small instances' uploading time when the number of nodes with 50Mbps throttling varies.

Figure 10 shows the time spent during file write when we vary the number of nodes with 50 Mbps throttling from 0 to 5. As shown in Figure 10, even there is only one node whose bandwidth is lower than other datanodes, SMARTH can outperform the traditional Hadoop cluster by 78%. We also can find that the more nodes with lower bandwidth, the more improvement can be gained by SMARTH.

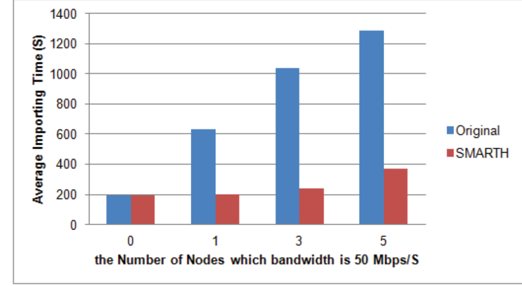
As we would expect, performance gain increases when we evaluate in medium and large clusters due to the big gap between the default bandwidth and the throttling bandwidth. From Figure 11(a), we observe an improvement of 167% when uploading a 8 GB data file in medium cluster, we can find the similar result in large cluster from Figure 11(b) when only one node's bandwidth is limited to 50 Mbps. The results also illustrate that the medium cluster and large cluster have the similar performance when the bandwidth limitation is the same.

We also test the import time when we vary the number of nodes with bandwidth throttling of 150 Mbps in small and large clusters. From graphs in Figure 12(a) and 12(b) , the benefit of SMARTH is reduced to 19% in small cluster and 59% in medium cluster compared with the bandwidth throttling of 50 Mbps.

3) Heterogeneous Cluster Scenario: For power, cost, and pricing reasons, clusters are evolving towards heterogeneous hardware. Heterogeneity also arises due to phased hardware upgrades over years. For example, data center expansion or upgrade will often result in multiple generations of hardware so that network topology may vary, with some routers having lower latency or supporting higher bandwidth than others [3].

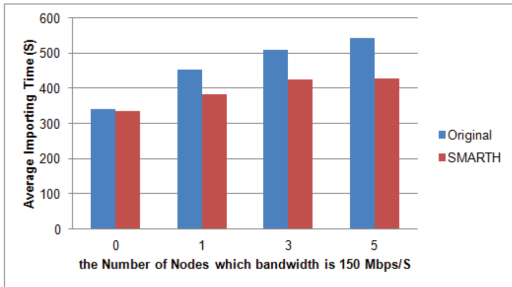


(a) medium cluster

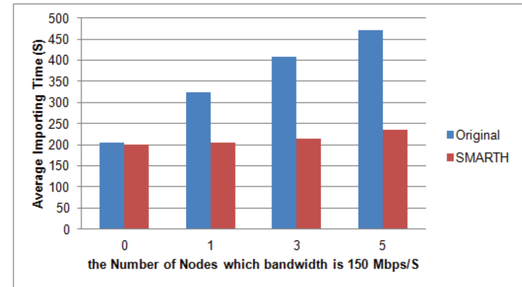


(b) large cluster

Figure 11. Comparison of uploading time for medium and large clusters when the number of nodes with 50Mbps throttling varies.



(a) small cluster



(b) medium cluster

Figure 12. Comparison of uploading time for small and medium clusters when the number of nodes with 150Mbps throttling varies.

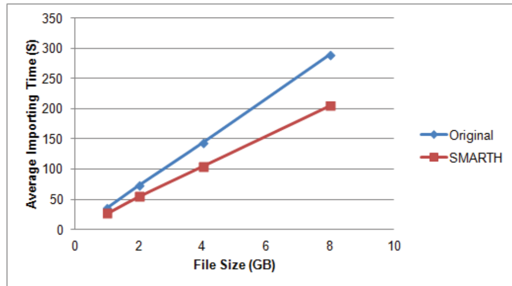


Figure 13. Comparison of uploading time of different data size in a heterogeneous cluster.

We repeat the same set of experiments in a heterogeneous cluster consisted of a mixture of small, medium, and large EC2 instances. Without any network throttling, Figure 13 shows that it takes 289 seconds to upload an 8 GB file in HDFS, but SMARTH only takes 205 seconds, which is 41% faster.

VI. Related Work

Although a rich set of research has been published on improving the performance of Apache Hadoop nowadays, there is little work in literature to analyze and improve the file transmission paradigm in the HDFS architecture. Xu et al.[4] tries to figure out a cost model to describe the data import and verify this cost model with practical evaluations. In their approach, Instead of opening an input stream to the local file and passing it along to the first datanode through a socket, the original data storage can be directly accessed by datanodes.

There are some literatures related to file write that mainly focus on adjustments to Hadoop parameters and codes to adapt HDFS to a specific scenario. For instance, Shafer et al. [5] analyze the performance of HDFS, and find out bottlenecks existing in the Hadoop implementation that result in inefficient HDFS usage. Their paper focuses on adjustments of Hadoop parameters to boost the overall efficiency of MapReduce applications. CoHadoop[6] is a lightweight extension of Hadoop that controls where data are stored. It uses hints given by applications to locate data files to improve efficiency. HDFS+[7] is an extended distributed file system from existing HDFS that can accept concurrent writes with multi data sources. In HDFS+, files are divided into fragments not sent in a sequence order, instead, each fragment can be written individually by a client.

A number of other research work have been proposed to make Hadoop more efficient than the original Hadoop. Islam et al.[8] introduce a novel design of HDFS using Remote Direct Memory Access (RDMA) on InfiniBand. The design is able to provide low-latency and high throughput for HDFS write operations as it leverages the RDMA capability of high performance network like InfiniBand. Yee et al.[9] introduce a generic socket API called Hadoop Filesystem

Agnostic API (HFAA) to allow Hadoop to integrate with any distributed file system over TCP sockets. This socket API can eliminate the demand to customize Hadoop's Java implementation, and move the implementation responsibilities to the file system. Hadoop-A[10] introduces a novel network-levitated merge algorithm to merge data without repetition and disk access to optimize data processing throughput of Hadoop.

VII. Conclusions and Future Work

Motivated by the increasing popularity of Hadoop applications, in this paper, we introduce an asynchronous multi-pipeline file transfer protocol with a revised fault tolerance mechanism instead of the HDFS's default stop-and-wait single-pipeline protocol. We employ global and local optimization techniques to sort datanodes in pipelines based on the historical data transfer speed. We conduct a series of experiments on Amazon's EC2 by varying the instance type, number of instances, and network bandwidth. Our experiments reveal significant improvement by 27-245% compared with HDFS.

In the future, we plan to investigate SMARTH's impact on MapReduce jobs and tasks. We also plan to evaluate SMARTH on different storage platforms and types such as RAID and SSD.

Acknowledgement

This work was supported in part by NSF-CAREER-1054834.

References

[1] Sachchidanand Singh and Nirmala Singh. Big data analytics. In *International Conference on Communication, Information & Computing Technology*, pages 1–4, Mumbai, India, 2012.

[2] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2012.

[3] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *the Third ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[4] Weijia Xu, Wei Luo, and Nicholas Woodward. Analysis and optimization of data import with hadoop. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1–9, 2012.

[5] Jeffrey Shafer, Scott Rixner, and Alan L. Cox. The hadoop distributed filesystem balancing portability and performance. In *IEEE International Symposium on Performance Analysis of Systems & Software*, pages 1–12, 2010.

[6] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Ozcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. Cohadoop: flexible data placement and its exploitation in hadoop. In *the VLDB Endowment*, pages 575–585, 2011.

[7] Kun Lu, Dong Dai, and Mingming Sun. Hdfs+: Concurrent writes improvements for hdfs. In *the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 1–2, 2013.

[8] N.S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2012.

[9] Adam Yee and Jeffrey Shafer. Hfaa: a generic socket api for hadoop file systems. In *the 2nd Workshop on Architectures and Systems for Big Data*, pages 15–20, 2011.

[10] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. Hadoop acceleration through network levitated merge. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2011.

[11] Chuck Lam. *Hadoop in Action*. Manning Publications, 2010.

[12] Eric Sammer. *Hadoop Operations*. O'Reilly Media, 2012.

[13] Apache Hadoop website. http://hadoop.apache.org/.

[14] Amazon EC2 website. http://aws.amazon.com/ec2/.

[15] Apache Hadoop from Wikipedia website. http://en.wikipedia.org/wiki/Apache_Hadoop.

[16] Andromachi Hatzieleftheriou and Stergios V. Anastasiadis. Improving bandwidth efficiency for consistent multistream storage. *ACM Transactions on Storage*, pages 2:1–2:26, 2013.

[17] Mengwei Ding, Long Zheng, Yanchao Lu, Li Li, Song Guo, and Minyi Guo. More convenient more overhead: the performance evaluation of hadoop streaming. In *ACM Symposium on Research in Applied Computation*, pages 307–313, 2011.

[18] Florin Dinu and T.S. Eugene Ng. Understanding the effects and implications of compute node related failures in hadoop. In *the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 187–198, 2012.

[19] Sven Groot, Kazuo Goda, Daisaku Yokoyama, Miyuki Nakano, and Masaru Kitsuregawa. Modeling i/o interference for data intensive distributed applications. In *the 28th Annual ACM Symposium on Applied Computing*, pages 343–350, 2013.

[20] Zhendong Cheng, Zhongzhi Luan, You Meng, Yijing Xu, Depei Qian, Alain Roy, Ning Zhang, and Gang Guan. Erms:an elastic replication management system for hdfs. In *IEEE International Conference on Cluster Computing Workshops*, pages 1–9, 2012.

[21] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing hadoop provisioning in the cloud. In *the 2009 conference on Hot topics in cloud computing*, pages 1–5, 2009.

[22] Shivnath Babu. Towards automatic optimization of mapreduce programs. In *the 1st ACM symposium on Cloud computing*, pages 1–6, 2010.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplied data processing on large clusters. In *the 6th conference on Symposium on Opearting Systems Design Implementation*, pages 1–13, 2004.

[24] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies*, pages 1–10, 2010.

[25] Xuhui Liu, Jizhong Han, Yunqin Zhong, Chengde Han, and Xubin He. Implementing webgis on hadoop: A case study of improving small file io performance on hdfs. In *IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, 2009.