Final Report: Implementing Multi-cast Data Replication for Hadoop

Akhila Athresh

May 18, 2013

1 Motivation

The motivation for this project is the need to implement an efficient data replication mechanism between nodes of the Hadoop Distributed File System (HDFS). At present, the Hadoop framework employs a pipeline-based replication method that performs efficiently on most network configurations designed for clusters. However, this method adds performance overhead, particularly in bandwidth constrained networks that do not inherently support point-to-point communication. The proposed multi-cast solution for data replication is likely to serve as an improvement of data replication process in Hadoop-based systems.

2 Project Objective

The objective of the project is to design multicast-based data replication protocol as an alternative to the existing pipeline-based approach, explore variants of multi-cast protocols, evaluate their performance against each other and against the pipeline based approach and arrive at a solution that is likely to be the best fit for data replication. Next set of steps will involve fully integrating the protocol with the Hadoop framework, optimizing the code so that it can be ported to embedded environments with ease and testing replication performance on bandwidth constrained network that lacks point-to-point communication.

3 Protocol Design

We started with building a stand-alone multi-cast protocol for basic file transfer. The setup consists of a single sender and two receivers. This emulates a multi-cast data transfer scenario in a Hadoop cluster consisting of one client node and two receiver datanodes, where the client initiates a multi-cast data transfer to the other two. The protocol was refined over multiple iterations to provide support for reliable transfers, i.e. in-order delivery of packets and re-transmissions of packets that were dropped by the network. Reliability is not a feature of UDP based multi-cast transmissions, but it is mandatory for high volume and reliable systems like HDFS.

The following parameters were considered while designing the protocol:

1. *Payload and Burst Size* The theoretical maximum size of an IP datagram is 64KB, but in practice the size of the payload is governed by the Maximum Transmission Unit(MTU) of the underlying physical network, which is 1500 bytes(Ethernet) in our case. The payload size is 1472 bytes, the remaining 28 bytes are reserved for standard IP and UDP headers. This is the maximum size of the payload after which it will be fragmented at the physical layer. Adhering to this size achieves lower end-to-end processing times with respect to fragmentation and re-assembly by the underlying physical layer. The receiver buffer was fixed at 64KB in order to support packet buffering while it performed a slower disk write operation. This leads to a burst size of around 40 packets,which is the number of unacknowledged packets that can be in flight at a point in transmission without overwhelming the receivers. Both the receivers reassemble the out-of-order packets upon receipt and send an ACK on successful receipt of all the packets in the burst. The client then sends the next set of 40 packets and this process continues till all of the data is multi-casted to both the receivers. Deterministic packet loss was simulated to test the re-transmission logic, which in-

volves the receivers sending the sequence numbers of lost packets in the transmission.

2. *Use of both UDP and TCP Sockets*The protocol uses *MulticastSocket* for sending data and *Server Socket* for sending and receiving acknowledgements. This is because HDFS already has TCP transmissions in place for acknowledgment and we wanted to reuse them for sending and receiving acknowledgments in the multi-cast approach as well

4 JGroups Evaluation

JGroups is an open source Java library for reliable multi-cast, that implements protocols for fragmentation, re-transmission of lost messages, failure detection and packet ordering among others [1]. JGroups conceptualizes a multi-cast channel as a *JChannel API* that can be used to connect to a cluster, send and receive messages and to register *listeners* that are invoked when events, such as a new member addition, occur. Data is encapsulated as *Messages* that has the byte buffer, sender and receiver's addresses. A *View* is an encapsulation of details of all the members of a cluster. While these are the fundamental elements of JGroups, it provides few advanced protocols like:

1. *NAKACK* that handles First-In-First-Out (FIFO) ordering of packets and ensures reliability.

2. *FRAG2* that fragments large messages into small chunks and reassembles them back at the receiver end.

3. *FD (Failure Detection)* that can be used to create heartbeat or 'are-you-alive' messages to test connectivity.

4. *MERGE2* that provides features for cluster partitioning.

5. *STABLE* which provides methods for a distributed garbage collection among nodes.

6. *STATE* which maintains state of cluster nodes that can be used to restore information in case of node failure.

Data Size (MB)	JGroups Transfer Time (s)	Custom Protocol Transfer Time (s)
40	4.6034	4.612
80	9.687	6.331
227	17.7891	16.787
454	24.6	22.5

The table compares the transfer times in a 3 node set-up for both JGroups and the protocol developed by us.

Figure 1: JGroups and Custom Protocol Comparison

While JGroups has some promising features like distributed data cleaning and cluster partitioning that might enhance our multi-cast performance, the protocols and their implementation are not well documented. The little documentation that is available shows basic use of *JChannel*, but to fully exploit benefits of JGroups, we will need to further study the source code of advanced JGroups protocols. JGroups also comes with a large set of configuration files for each of the available protocols, that need to be configured for each of the protocol implemented in the client application. Various parameters like retransmission time-out, frequency and buffer size can be set in these configuration files. JGroups offers a robust protocol stack but with configuration parameters that is difficult to integrate with the current HDFS code. Further research is required to determine the feasibility of using JGroups with HDFS. *Figure 1* highlights the comparison of JGroups and the custom protocol we have designed. The results are comparable for most part.

5 HDFS Integration

The next step was to integrate the protocol with HDFS and test it on a Hadoop cluster for anticipated performance improvement. We will briefly explain the HDFS write mechanism and proceed to explain modification of HDFS source code. *Figure 2* illustrates the write mechanism between a client and the datanodes. The replication factor, which is the number of datanodes to which a block is written to, is 3 in the figure. The client first consults the namenode to obtain permissions to write file to the datanodes. Upon success, the namenode returns a list of datanodes to which the client can write a single block of the file. A block in HDFS is generally 64MB in size and a file is split up into smaller

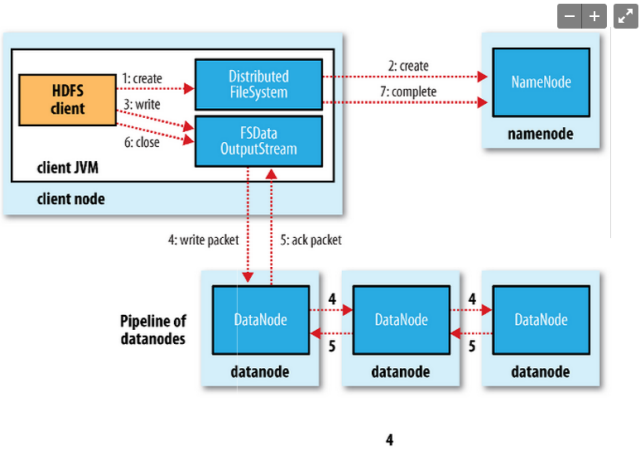

Figure 2: HDFS Write

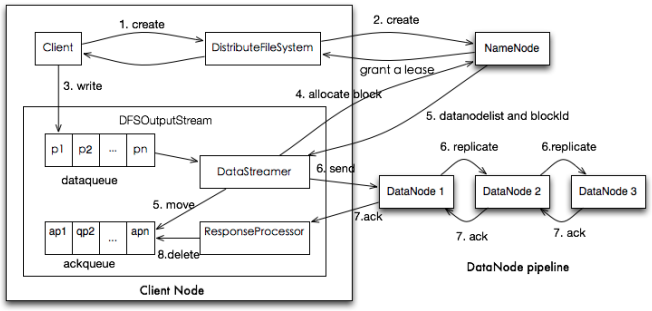

Figure 3: Pipeline Transfer:Client View

chunks that is written into these 64MB blocks on the HDFS. Suppose we have a 100 MB file that has to be written to the HDFS, the file is then split into 2 chunks, one is of size 64 MB and it occupies a full block on HDFS and the other is of size 36MB, which partially occupies a 64 MB block on HDFS. Every chunk is written to a block in one of the datanodes and replicated to the other downstream datanodes involved in the pipeline process. In *Figure 3*, datanodes 2 receives a chunk from datanode 1, which is then forwarded to datanode 3. The datanodes are placed such that one of them is on one physical rack and the other two are on another physical rack.

Upon successful receipt of a 64MB chunk (or less) and a write to the file system, every datanode in the pipeline sends acknowledgement to its immediate upstream datanode. The acknowledgements get appended as they are passed upwards in the pipeline and finally reaches the first datanode in the pipeline which forwards the collective acknowledgement back to the client. The client does not proceed to write the second block of 64MB till it has received acknowledgements from all the datanodes in the pipeline. The client then consults the namenode for the next set of datanodes to write the next block and this process continues till entire data is written to the datanodes. This means that a file of 1TB will cause a network traffic of 3TB and the time taken for a write grows proportionally with the size of the data and the number of nodes in the pipeline. It is to be noted that the communication between the datanodes in the pipeline and between the first datanode and the client is through TCP streams. This was the reason we maintained TCP connection between sender and receiver between the standalone protocol discussed in the earlier sections.

To integrate the multi-cast protocol, we had to first understand the interaction of various classes in the HDFS source code and how the pipelined transfer logic is built into these classes. *Figure 3* illustrates the interaction on the client end at a more granular level. The following classes have being modified and tested in HDFS source:

1. *DFSClient* connects to the HDFS and performs basic file tasks. It uses the *ClientProtocol* to communicate with a *NameNode* daemon, and connects directly to DataNodes to read/write block data. The *DFSClient* class internally maintains a *DataStreamer* daemon, that is responsible for sending data to the first datanode in the pipeline and a *ResponseProcessor* daemon that handles ACK responses from downstream datanodes. The *DFSClient* takes the data that corresponds to the first block write and divides it into packets of 64KB each. Therefore, in our earlier example, the first block of 64MB is sent as packets of 64KB leading to a total of 1000 packets. When the client fills a packet, it is pushed onto the *dataQueue*, which is an thread-safe internal queue structure that maintains a set of unacknowledged packets. The *DataStreamer* picks a packet from the *dataQueue* and sends it to the first datanode in the pipeline. It then puts the packet in the *ackQueue* and it is held there until it receives a collective acknowledgment from the first datanode.

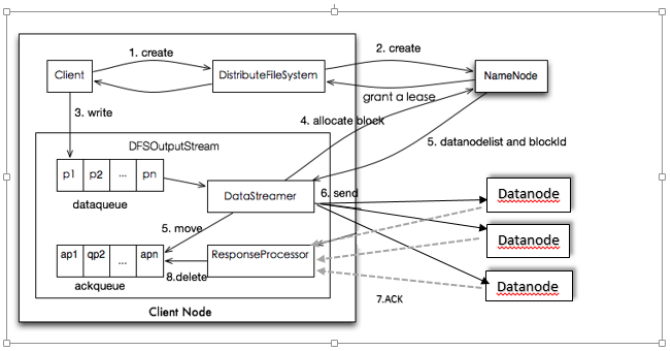

Figure 4: Multicast Transfer:Client View

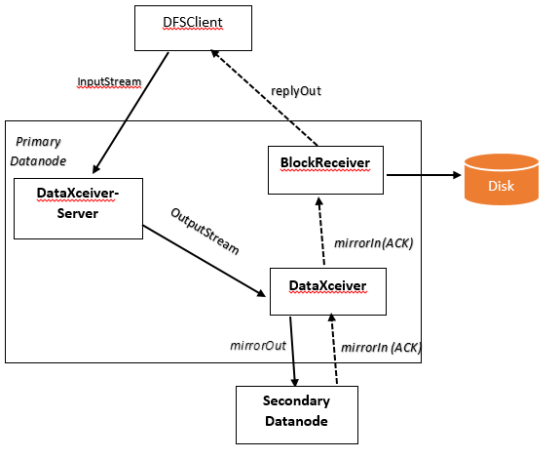

Figure 5: Pipelined Transfer:DataXceiver-BlockReceiver

2. *DataXceiverServer* is the driver class for the *DataXceiver* class. It runs a thread which in turn spawns *DataXceiver* for every packet received.

3. *DataXceiver* is responsible for processing incoming/outgoing data streams.*Figure 5* illustrates the interaction between the *DataXceiver* and *BlockReceiver* on each datanode. For data writes, *DataXceiver* invokes methods in the *BlockReceiver* class. The *BlockReceiver* receives data and writes to a block on the disk and at the same time initiates a downstream data transfer to the next downstream datanode.

   As of now, the multi-cast send and receive components have been integrated with HDFS but the full functionality of writing data to the filesystem and sending ACK back to sender is

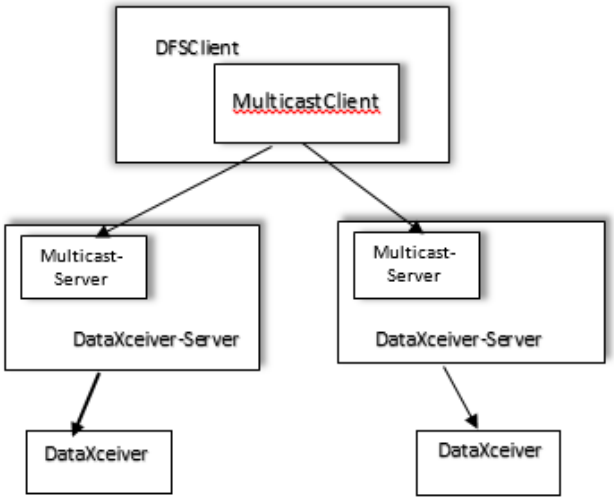

Figure 6: Multicast Internal View

pending. A switch has been provided between pipeline and multi-cast approaches. *Figure 6* gives an outline of the classes that have been modified and introduced. *DataXceiverServer* joins the multi-cast group to which the *DFSClient* is multi-casting the data. Internal to these two classes, are the *MulticastClient* and *MulticastServer* classes that were the standalone sender and receiver pair in the custom protocol mentioned in the earlier sections. The difference is that the classes and the functionality they provide are now wrapped as Output and InputStreams respectively, so that they can blend with the existing stream based communication between the components with ease. *DataXceiverServer* receives the packets from the *DFSClient* and buffers it, until it has received a single packet's worth data. It then wraps the buffer in an *InputStream* and sends it to the *DataXceiver*. The Integration is complete till this point and tests have been conducted to estimate the packet delivery time to each of the nodes which is described in the next section

Packet Arrival Times	Multicast Based	Pipeline Based
DataNode 1	5.61 s	14.662 s
DataNode 2	5.67 s	19.780 s

The table shows the arrival time (with reference to start of the data node) of the first packet in a multicast cluster and a pipelined cluster for a 64KB packet. As seen, the packet arrival time at DataNode 2 in Pipeline-based transfers arrive a little later because of the pipeline induced wait.

Figure 7: Packet Arrival Times

6 Performance

The difference between the two kinds of transfers, can be seen in the delay introduced in packet delivery between subsequent datanodes in the pipeline. In a pipeline, unless an upstream datanode has received the data, the downstream datanode cannot proceed and unless the client has received acknowledgements from chained nodes, it will halt and not transfer the next block. In a multi-cast transfer, each rack switch can independently multi-cast the packets to datanodes in that rack, unlike a pipeline transfer where the secondary datanode receives packets from the primary datanode from across the rack.

Thus, there are two distinguishing parameters that needs to be measured. One, the time it takes for every datanode to receive a packet i.e., the packet arrival times at each datanode for both kinds of transfer. Second, the time it takes for the acknowledgements from all the downstream datanodes to reach the DFSClient. Since the integration is only partially complete, we focused on finding the packet arrival time at each of the datanodes. If our assumption is correct, the time elapsed between the node start-up and receipt of the first 64KB packet should only slightly differ for the datanodes in a multi-cast transfer and the packet arrival times at each of the datanodes should differ significantly in a pipelined set-up.

Our experiment set-up consists of one client and two datanodes. Either the datanodes are on the same rack or on two different racks. In both the cases, the packet arrival times at the datanodes should be very similar in a multi-cast transfer and otherwise in a pipelined transfer.

The results of preliminary tests are summarized in *Figure 7*. The data for which the arrival times have been tested is around 64KB or less because the integration is not fully complete . We can only test the arrival time for the first 64KB (or less) packet to the datanodes as the multicast ACK feature is not fully built in. Once we fully integrate the protocol with HDFS code, we will able to perform end-to-end tests for large realistic file transfers and be able to account for delays induced by ACK responses as well.

7 Timeline

February 25th - Completed design and development of initial version of the multi-cast protocol and tested the same for transfers of 40, 80, 200 and 400 MB data on a 3 node cluster.

March 10th - Refined the code, fixed few issues with threads in the sender class.

March 23rd - Researched JGroups and built a simple file transfer application using JChannel, Messages and Views and compared performance with custom multi-cast protocol.

April 2nd - Testing the protocol for retransmission performance on a 3 node cluster.

April 18th - Refined the protocol to allow selective retransmissions.

April 30th - Started reading HDFS source code to understand the existing implementation.

May 14th - Started modifying the receiver classes and DFSClient class.

May 28th - Completed Multicast Sender and Receiver Integration with HDFS. Acknowledgement feature and file write, however, needs to be built into the multi-cast set-up and is in the pipeline.

8 Future Work

(a) Complete the integration with HDFS source code and tests for large file transfers and report time taken for transfers.

(b) Test transfers and re-transfers for node failure scenarios.

(c) Optimize the code for further performance improvement.

References

[1] B. Ban. JGroups - A Toolkit for Reliable Multicast Communication. [Online]. `http://www.jgroups.org/`.

[2] The Apache Software Foundation. Hadoop. [Online]. `http://hadoop.apache.org`.

[3] Bjorn Gronvall, Ian Marsh, and Stephen Pink. A Multicast-based Distributed File System for the Internet. pages 1–3.

[4] YoungHoon Jung, Richard Neill, and Luca P. Carloni. A broadband embedded computing system for mapreduce utilizing hadoop. pages 1 –9, December 2012.

[5] L. Parziale, D.T. Britt, C. Davis, J. Forrester, W. Liu, C. Matthews, and N. Rosselot. TCP/IP Tutorial and Technical Overview. [Online]. `http://www.redbooks.ibm.com/redbooks/pdfs/gg243376.pdf`.

[6] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. pages 1–9.

[7] T.White. Hadoop -The Definitive Guide. [Book]., 2012.