YARN File Transfer Service Design

1. Motivation

In Hadoop and related softwares, there are demands of broadcasting large files. For example, YARN application may localize large jar files on each node, and Hive may distribute large tables in fragment-replicate joins; docker integration may broadcast large container image. Current solution is to put the files on HDFS and let each node download from HDFS, which is inefficient and not scalable. So we want to build a better file transfer service in YARN so that all applications can use it broadcast large file efficiently.

2. Existing Solutions

2.1 Centralized HDFS Broadcast

In this way, files are put into HDFS, and all nodes download the files from HDFS. The pro is fault tolerance can be achieved easily, the con is it doesn't scale because of limited network bandwidth of source nodes (and cross-rack bandwidth). Experiments in [1] show this way works with less than 50 nodes and then gets linearly increasing download time.

2.2 Multicast

Multicast is an efficient way of one-to-many communication. Basically, instead of sending data to each receiver, sender just needs to send data once to the receiver group, and network devices will route the data to all receivers. This requires hardware support and non-trivial operation work, especially in enterprise environment.

2.3 D-ary distribution tree

In this way, nodes form a d-ary tree, get data from parents and send data to children. If d is 1, that's the pipelined chain-based solution used by HDFS in replication. The pro is it's more efficient than centralized solution, the con is a slow inner node will affect all nodes of its subtree. Another con is it needs extra effort in fault tolerance. Experiments in [1] show it's more scalable than centralized solution but there is no test showing the max number of nodes it can scale up to.

2.2 BitTorrent based Broadcast in Spark

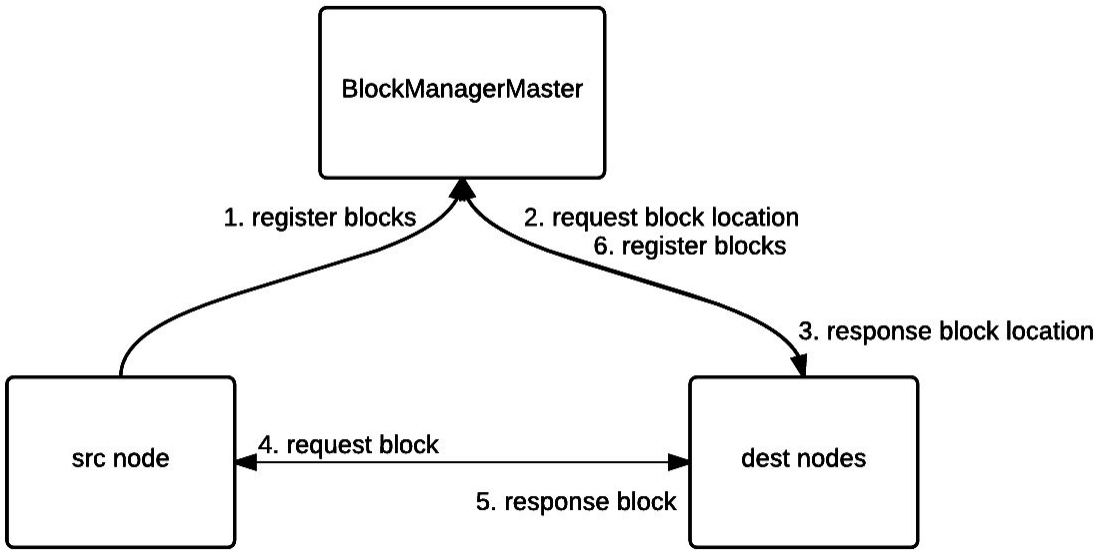
In paper [1], researcher of Berkeley tried original BitTorrent algorithm and found it scaled poorly because there are many overhead to make BitTorrent suitable for Internet. In paper [1][2] They modified the algorithm to fit it into data center environment and new version scaled very well.

The main modifications are the following:
(1) instead of breaking data into 256KB piece and then into 16KB blocks, only breaking data into large block(4MB) and no subdivide. Breaking data into pieces enable concurrent downloading from multiple peers, and breaking piece into blocks enable requesting pipelining to one peer.

These are designed for Internet which has slow and lagging. Hadoop environment typically has better network resource, so using large block without subdivision will be better.
(2) instead of using choking algorithm for fairness, letting all nodes satisfy all requests they get. Originally, BitTorrents clients use choking algorithm to choose the nodes to which it uploads data for tit-for-tat purpose. This doesn't make sense in our case.
(3) instead of using SHA1 checksum on all blocks, using simple integrity check on each block. BitTorrent uses SHA1 to verify data, but in our case, there won't be malformed data as long as the identification is authenticated.

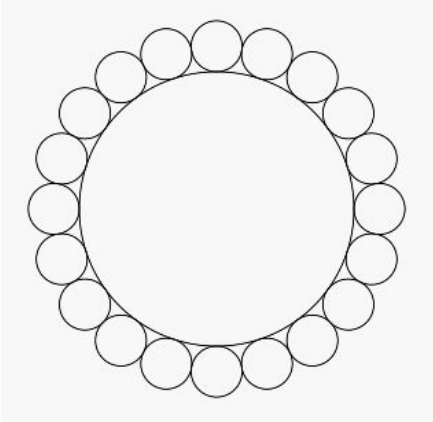
They also tried topology-aware solution by giving priority to nodes on the same rack when choosing the node to download data from, and it worked very well. Experiments in paper[1][2] shows it outperforms other solutions and scales well (tested with 100 nodes). The pro is it has good performance and scalability, and it's easy to achieve fault tolerance due to the robustness of BitTorrent.

Current Spark implementation adopts this solution with modification. It uses single BlockManagerMaster to maintain the location info of blocks. To broadcast data, source node first tells BlockManagerMaster it has blocks. Then all nodes randomly choose a unfinished block, ask BlockManagerMaster for which nodes have this block, get block from a randomly-chosen node and then tells BlockManagerMaster it now has this block. Different from the paper, there is no checksum and topology-aware support. Also, fault tolerance is not supported yet.



3. Design

We regard the cluster as a two-level ring: the top level is the ring of rack, and in each rack there is a second level ring of nodes. In each ring, we use torrent data and consistent hashing to choose the master, so there will be a master rack and a local master in each rack. The master of master rack is the global master. Notice for different torrents, we get different master racks and masters node.



We will have user to specify the max bandwidth consumption of each connection and the max number of simultaneous broadcast connections. These configuration can be easily implemented by throttling network usage of broadcast on each node. (NOT IMPLEMENTED)

When a node initiates broadcast, it announces the global master it's seeding. When a node wants to download file, it use local master's tracker to find peers. If it finds itself the local master, it use global master's tracker to find peers. Local master only know the peers on the same rack unless it's also global master, in which case it will return peers on the same rack for local nodes and peers on the different racks for non-local nodes.

Non-master nodes can only get peers on the same rack from local master, master nodes can only get peers on masters of other racks. So the file will be first spreaded to all master nodes, and then all other nodes get file from their masters. This is how topology awareness is achieved. If any node fails, other nodes will use consistent hashing to find next node on the ring. This is how fault tolerance is achieved. For different torrents, we have different masters, so this is how load balancing is achieved.

Notice this design doesn't consider the case when there is no topology input and when there is new nodes added. If there is no topology input, we can get a list of all nodes and form random rings. If new nodes is added, we can ask nodes to refresh the topology input.

4. Implementation

Instead of writing our own BitTorrent-like broadcast protocol, we decide to use existing BitTorrent library because that's a much easier way to go. We find a BitTorrent library in java called ttorrent. It's pretty mature and still have active maintenance, so we decide to use this library.

For optimization purpose, we tailored BitTorrent protocol in several aspects. Also for this reason, we basically put ttorrent code in YARN and modified it, which is pretty hacky. Beyond changes suggested in papers[1][2], UPD related parts and multiple tracker support has been removed.

Localization service has been modified to make use of broadcast service. A new resource type called BTFILE has been added. Whenever a BTFILE resource need to be localized, it's localized using broadcast service.

5. Usage

5.1 Configuration

First, you need to modify yarn-site.xml and add broadcast service in the auxiliary service. Please add broadcast in 'yarn.nodemanager.aux-services' and set 'yarn.node.aux-services.broadcast.class' to 'org.apache.hadoop.yarn.server.broadcast.service.BroadcastService'.

Second, you need write a topology file and set 'yarn.nodemanager.broadcast_service.topology_file' to the absolute path of it.
The format of topology file is contains pair of node ip and rack name which is separated by a space, like '192.168.3.2 rack-0'. There should be only one pair per line.

5.2 Demo app

There is a demo app in hadoop-yarn-applications-broadcast-demo module, which basically localize resource and check md5.

Please run this command:
yarn jar <local path to hadoop-yarn-applications-broadcast-demo-2.9.0-SNAPSHOT.jar> org.apache.hadoop.yarn.applications.broadcast.Client <local path to hadoop-yarn-applications-broadcast-demo-2.9.0-SNAPSHOT.jar> <local path to any data file> <hdfs/torrent, hdfs for original localization, torrent for broadcast service based location> <BitTorrent block size in MB> <number of containers to launch excluding AM>

To see whether localization succeed, please check whether md5 in each container's output file are consistent with md5 of your data file.

Reference

[1] Chowdhury, Mosharaf, Matei Zaharia, and Ion Stoica. "Performance and Scalability of Broadcast in Spark."

[2] Chowdhury, Mosharaf, et al. "Managing data transfers in computer clusters with orchestra." *ACM SIGCOMM Computer Communication Review* 41.4 (2011): 98-109.