In the name of God

DISTRIBUTED STORAGE PROTOCOL

June 11, 2021

Author: Sepehr Javid
Supervisor: Dr. Fakhrahmad
Shiraz University
Computer Science and Engineering Department
javid.sepehr77@gmail.com

Contents

1 Introduction

Considering the distributed storage protocols, the first that comes in mind is *HDFS*. The main architecture of this protocol consists of two types of nodes; *Namenode*, and *Datanode*. In general, there is only one Namenode that serves the client requests and monitors the Datanodes' condition, and multiple Data Nodes to store the actual file or chunks of an uploaded file. However, HDFS suffers from an issue so-called *Single Point of Failure*, implying that in case of the Namenode malfunction, the entire service will fail to respond to client requests. This project chiefly aims to provide a solution regarding the mentioned issue.

2 General Idea

The general idea to solve the aforementioned issue is to add the capability to serve a client to each Datanode. The reason behind such idea is that since there are more than one Datanode in the file system, each could act as an individual Namenode if the active Namenode breaks down. On the other hand, some data center architects prefer to setup an idle Namenode to provide redundancy, However, doing so increases costs and reduces the risk by half while using the current Datanodes as Namenodes may sound more economical. According to the HDFS protocol, the Namenode holds a database storing all data regarding the file system events such as directories, files, chunks, physical location of each chunk, etc. With this new idea, one of the concerns would be to assure consistency among all nodes' database which requires an organized communication between Datanodes in order to provide fully awareness of modifications taking place in the file system. As a result, each Datanode will have two major responsibilities; One to serve clients (if having been elected), and the other to organize communication with other Datanodes. For this matter, two separate processes called *Peer Controller* and *Client Controller* are Incorporated in each Datanode to provide both services. The architecture of each Datanode is depicted in figure 1 whose each part we will discuss later in this document.

The network of the connected Datanodes called *Chord* is a cyclic graph with each node's degree being less than or equal to two. This implies that each node is connected to two other nodes using established TCP connections. It is obvious that in case of having less that three Datanodes, each may have none or only one alive connection to a peer. An example of the chord is provided in figure 2.

3 Peer Controller

Peer controller is chiefly responsible for organizing the communication between Datanodes and consists of two main components; *Peer Server*, and *Peer Recv Thread* where Peer Server is responsible for acting on received broadcasts from other peers and peer recv thread's obligation is to handle the TCP connection of each adjacent peers and messages received through that channel. In addition, we can divide the Peer controller responsibilities into four main categories.

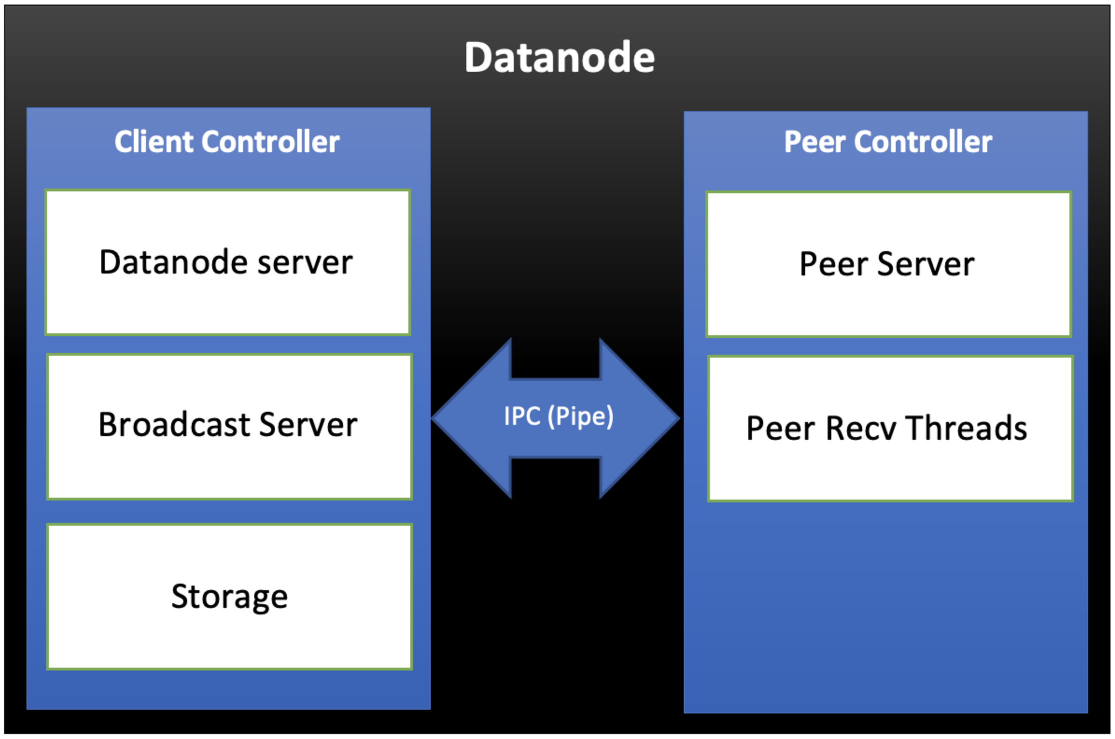


Figure 1: Architecture

3.1 Active Namenode Selection

As discussed before, the file system requires an active Namenode to serve client requests and organize saved chunks. According to the initial aim in this protocol, we need more than one Namenode; Specifically speaking, all Datanodes should be able to act as a Namenode but only one can be active at a time. In case of failure in the active Namenode, another will be elected and takes control of the file system to keep providing the service. The election takes place subsequent to a peer joining to the chord or the active Namenode malfunction which is described thoroughly in *Perform Recovery Actions*.

The election relies on *priority* attribute configured in the *dfs.conf* of each node with zero being the highest priority. In case of equality of two or more priorities, the node with lower IP address is elected.

3.2 Add New Peer

In case of adding a new Datanode to the file system, we need to assure that the chord will maintain its properties. In means of achieving that, the joining node broadcasts a message to the network and awaits until one of the current Datanodes respond. The steps

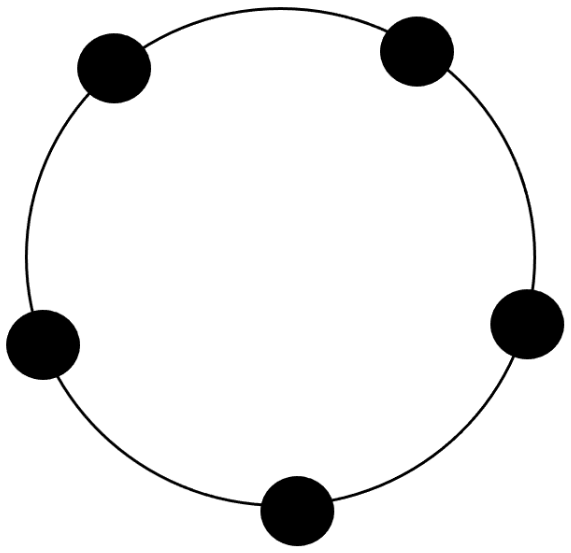


Figure 2: Chord Graph Example

required to join the chord is illustrated in figure 3.

It is notable that the steps may differ in special cases with less than two nodes in the chord before the join. It is Also worth mentioning that in case of being the first node to join the chord, the broadcast attempts will have to fail for three times before the node assumes it is the only node in the file system and starts the service solely.

Once the TCP connection with both sides is established, the offering node sends the file system database to the joining node, containing the entire data regarding the file system except the actual file chunks. During this process a broadcast is transmitted by the offering node, in order to lock the Namenode so that it no longer distributes database changes to other peers concurrent with the database transfer. This is to make sure that the joining node will receive the latest version of the database and will not miss database updates prior to receiving the database itself. As a result, The joining node will transmit a broadcast to release the Namenode's lock upon the full reception of the database. Once the database is thoroughly received, the joining node will inform all others nodes of its addition to the chord in order for them to add it their databases.

3.3 Distribute File System Changes

As discussed before, the Namenode is aware of all events taking place in the file system such as created directories, uploaded files and the stored location of their chunks, etc. On the other hand, other Datanodes need to be prepared to switch Namenode mode in case

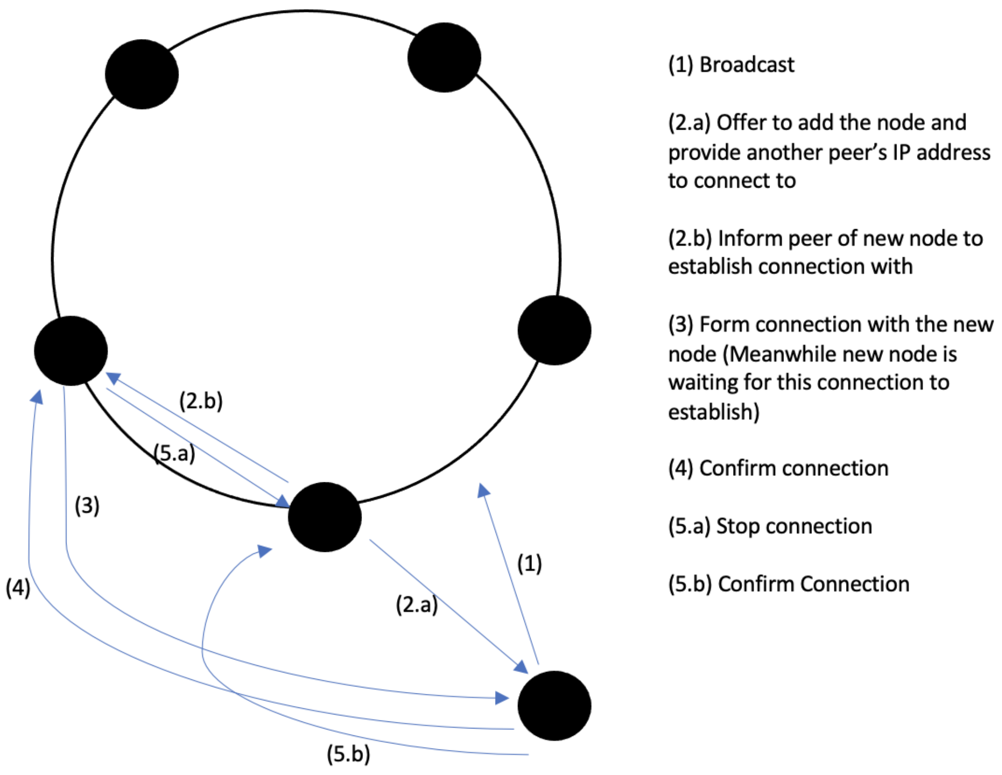

(1) Broadcast
(2.a) Offer to add the node and provide another peer's IP address to connect to
(2.b) Inform peer of new node to establish connection with
(3) Form connection with the new node (Meanwhile new node is waiting for this connection to establish)
(4) Confirm connection
(5.a) Stop connection
(5.b) Confirm Connection
(2.b)
(5.a)
(3)
(4)
(1)
(2.a)
(5.b)

Figure 3: Joining Chord Steps

of the current active Namenode's failure. To provide such preparation, the Namenode distributes any changes to the database using its connected peers. The process starts by sending a message containing the update to both adjacent peers which then they distribute to their neighboring peers until the message is received and acted upon by every peer in the chord. (Figure 4)

To avoid the update message from looping through the chord more than once, each peer who receives the message adds its IP address at the end of the message so that in case of receiving the message again, it can drop the message and never send it through again.

One of the important points to take into account is that each node receives the same update message from both sides. The reason behind this is to make sure the message will reach all nodes even if one node along the way of one side fails.

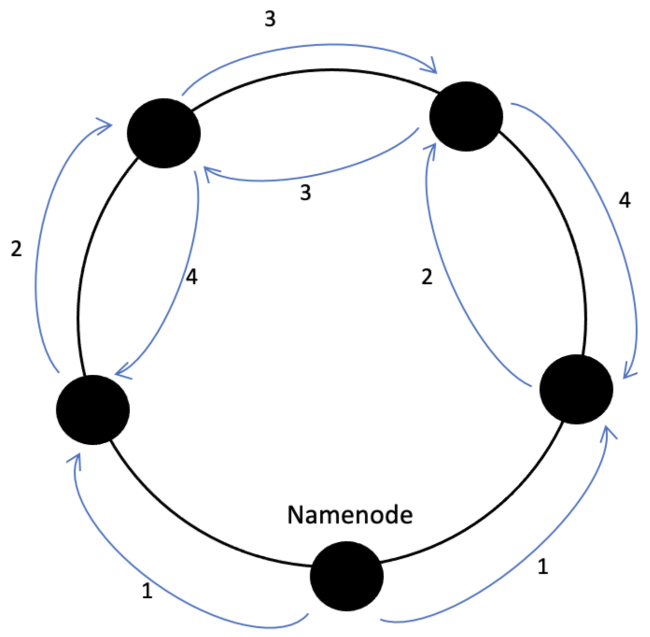


Figure 4: Update Message Distribution

3.4 Perform Recovery Actions

As previously stated, each node has at most two active TCP connections with its peers. Should any of the connections fail, the peer controller thread corresponding to the connection enters the recovery state which is broken down into multiple steps. (Figure 5)

3.4.1 Namenode Election

In case the failed node is a Namenode, an instant decision must be made for the new Namenode in order to minimize the down time. The current node will reelect the Namenode locally and broadcast a message to other peers to trigger their reelection as well.

3.4.2 Find Other Peer

In this step, the node will attempt to find the other node whose neighbor failed too. As a result, a broadcast is transmitted, reporting the failed node and looking for the other peer.

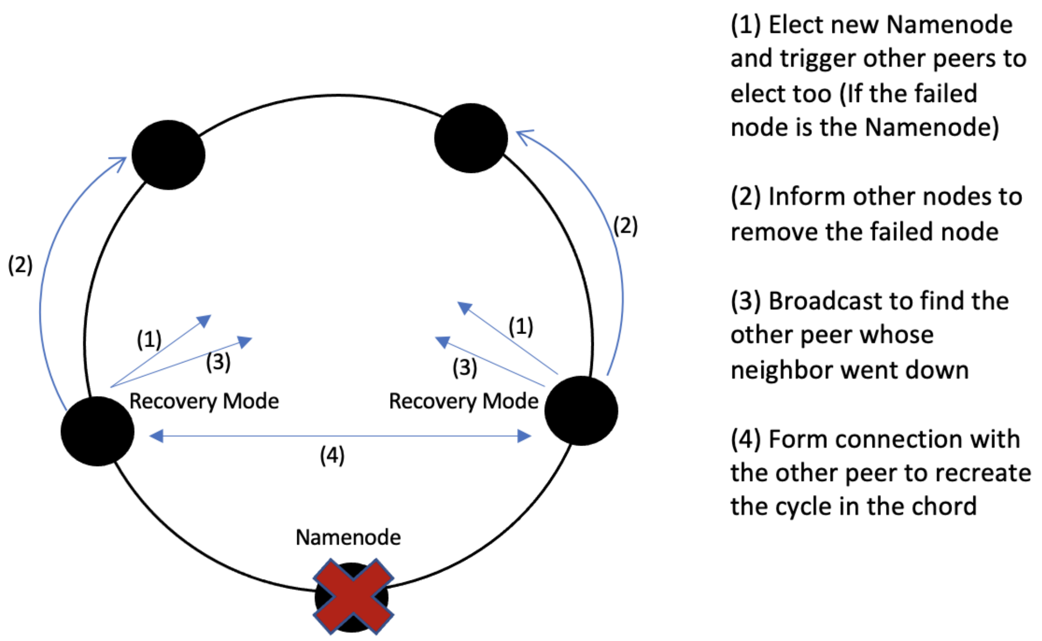

(1) Elect new Namenode and trigger other peers to elect too (If the failed node is the Namenode)
(2) Inform other nodes to remove the failed node
(3) Broadcast to find the other peer whose neighbor went down
(4) Form connection with the other peer to recreate the cycle in the chord
(2)
(2)
(1)
(3)
(1)
(3)
Recovery Mode
Recovery Mode
(4)
Namenode

Figure 5: Recovery Steps

3.4.3 Establish Connection

As the last step, the node with greater IP address will wait for the other peer to initiate a TCP connection and form the cycle in the chord again.

4 Client Controller

This component is dedicated to serving client requests as well as store actual chunks in the server. This section in divided into three major parts *Broadcast Server*, *Datanode Server*, and *Storage*.

4.1 Broadcast Server

As discussed before, active Namenode could be any one of the Datanode servers in the chord; As a result, the client will not have a specific IP address to reach. To that end, the client asking for a service will have to transmit a broadcast message to the broadcast address of the chord's network in order to call out the Namenode. All Datanodes will receive the broadcast, however, only the active Namenode gets to respond to the client.

Once the Namenode is resolved, a TCP connection is established to trade data between the two. The details of the TCP connections are covered in later sections.

Another aspect of this component is that it serves all clients concurrently using the *ClientThread* class which parses and serves the client. However, should a client thread stall, the thread may remain running with no progress while holding system's resources. To address such issue, the broadcast server component runs an inspection every ten seconds to remove terminated thread or kill the stalled ones in order to free resources for later clients. It is notable that each client thread will be killed after a time limit being four minutes.

4.2 Datanode Server

This component is chiefly used for uploading or retrieving chunks to or from the server as well as replicating uploaded chunks (More on replication later). Additionally, this component is always active since it is not related to the current Namenode of the file system.

4.3 Storage

Last but not least is the storage component which is responsible for the actual chunks, their names on the physical drive, informing other nodes about the current available byte size on the hard drive of the current node, and suggesting Datanodes for replication or uploading purposes.

4.3.1 Informing Current Available Size

One of the obligations of the storage component is to keep other nodes updated on the available byte size of the current Datanode. Therefore, any active Namenode can decide whether to assign any chunks to the current Datanode or not. The storage simply tries to update the available size locally first, and then advertises the new size to other nodes in the same way it distributes database changes.

4.3.2 Suggest Upload Datanode

Each client with a request to upload a file should be provided with the IP address of the Datanodes that can host the physical chunks. The storage component of the active Namenode is responsible for this matter. The algorithm used is a greedy algorithm, trying to full each Datanode to its maximum byte size available, or the maximum byte size of a chunk. This process continues until the entire file is fit in multiple Datanodes. Most important property of the algorithm is that the first Datanodes that get picked are the ones with most byte size available. As a result, files are roughly equally distributed between nodes and there will not be a case in which one Datanode runs out of space much sooner than others.

4.3.3 Suggest Replication Datanode

In the process of the upload chunk request, the Datanode receives suggestions from the storage on which Datanodes to replicate the chunk to. The utilized algorithm is a greedy algorithm which firstly eliminates all nodes nodes with in the same rack as the current node and starts picking the ones with the most available byte size first. If the algorithm finishes with less Datanode suggestions than the replication factor, the deficiency is compensated with the Datanodes within the current Datanode's rack. If the number of suggestions is still less than the replication factor, the suggestion is returned regardless of this fact.

5 Controllers' Inter-process Communication

As described earlier, client services is a part of the client controller's obligation whereas distributing the change according to the service is the peer controller's responsibility; Therefore, each modification taken place in the client controller, needs to be transferred to the peer controller to be distributed. In order to achieve that, there is an individual thread in the peer controller designated for receiving modifications from client controller which later gets distributed to other nodes.

Additionally, since the broadcast server and the storage are a part of the client controller, and the election of the Namenode as well as reception of the distributed modification information is a part of the peer controller's job, therefore existence of a thread in the client controller to receive information form peer controller is essential too.

6 Metadata

Based on the previous explanations, each Datanode has an individual yet consistent database containing file system data. The ERD of the database consisting of six tables is depicted in figure 6. Each table object has an *id* which provides uniqueness in the scope of each Datanode and a specific property to guarantee uniqueness in the scope of the whole file system (*global uniqueness*). Global uniqueness is an important factor to consider since in modification distribution, only one object from a specific table must be interpreted from the transferred message to provide consistency among databases. More specifically, it means that the id filed in each table is simply a primary key for local usages and it is not guaranteed that a certain object in a specific Datanode, has the same id in another.

6.1 Datanode

This table represents the current active Datanodes in the chord. Uniqueness of each object is provided by the *IP address* field. *Rack number* represents the identifier of the rack in which the server is. This field is mainly used for *rack awareness* in replication process which we will discuss in details in a bit.

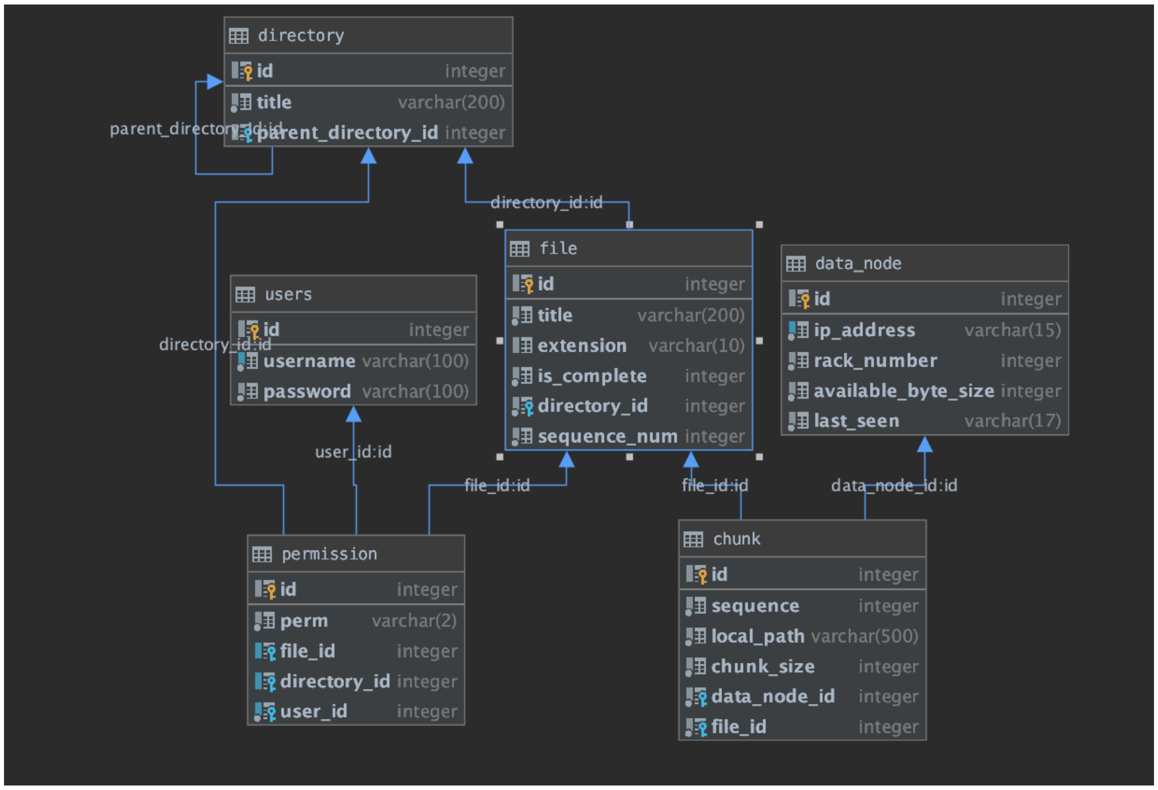


Figure 6: Database ERD

6.2 Users

One of the most practical tables to store file system users is *Users* table. Each user's username is distinctive among all others. The *Password* field is plain text since the main focus of this project was not security while in some parts, a hint of security may have been considered.

6.3 Directory

Directory table is dedicated to storing all directories in the file system. Each user has a base directory called *Main*. Uniqueness of a directory is a result of their path. Each path starts with a username followed by their main directory and any other directory following that.

Example: sepehrjavid/main/project/test.png

Since username is a unique field, and each path in a users profile is unique locally, therefore the entire path could be considered unique globally.

6.4 File

This table, as the name suggests, stores all files in the file system and to assure global uniqueness we use the same path strategy as we did with directories. It is notable that it does not store the actual data within a file. Most important fields to point out in this table are *title* begin the file name, *directory_id* being the parent directory id, and *sequence_num* which is the number chunks the file has been divided to.

6.5 Chunk

Following the File table comes a table representing chunks stored physically on all Datanodes in the entire file system. It is important that each Datanode is aware of all chunks stored on each node and the physical path to retrieve it since each node may become the Namenode and will have to provide the chunks to the asking client. Notable fields in this table are *sequence* being the sequence of the chunk in creating the file, *data_node_id* which represents the Datanode object in which the chunk is stored, and *local_path* that is the physical path on the mentioned Datanode. The distinctive property of this table is the combination of data_node_id, sequence, and file_id which assures that no two chunks representing the same file and with the same sequence will not be stored on the same node. The size of each chunk in this protocol, similar to HDF, is 64 MB.

6.6 Permission

Last but not least comes a table restricting user access to files and directories. This table acts as a middle table for user-file and user-directory many to many relation middle table. The permission is simply state by the *perm* field. The global uniqueness of this table is fulfilled by the distinctiveness of the file, directory, and the user it is referring to.

7 Sessions and Broadcasts

As briefly pointed out previously, all connections use TCP except the broadcasts which are UDP messages. Broadcast messages used a slightly simple method to transmit and receive a message whereas TCP sessions were divided into three different kinds.

7.1 Simple Session

Simple session initiates a connection with a TCP socket. This session divides the message into 4096 bytes chunks in order to send a message and send the byte size of the message prior to sending the actual message. This is so that the receiver knows when the message is completely received.

7.2 Encrypted Session

Encrypted session is the same as simple session to some degree with a slight difference. This type of session, transfers RSA-encrypted messages through a TCP socket. In the construction method of this session, an RSA key exchange between two sides is carried out so that the encryption could be executed properly.

7.3 File Session

This type of session uses the encrypted session to transfer files over TCP connections. It has the ability to transfer a part of or an entire file and receive on the other side. Additionally, the replication is handled in this layer. As previously mentioned, data is first split into chunks of 4096 bytes before being sent over socket; therefore, when receiving a file with replication enabled, each chunk of the file will be sent to the replication nodes instantly subsequent to their reception and prior to the entire file's reception. This method make the replication consume much less time that forwarding the file sequentially and after its complete reception. It is notable that forwarding chunks of a message to replication nodes is carried out using threads where each thread is responsible for one node.

8 Replication

Consider a case where a file consisting of three chunks is stored in the file system where each chunk is stored on a separate Datanode. It is obvious that if any of the mentioned nodes fails, other chunks will be impractical and the file will be corrupted. As the HDFS protocol implies, replicating chunks by the replication factor (which is three in this protocol) would be a remedy to this issue. However, a stored chunk on a certain node, cannot be replicated in the same node since this will contradict with the globally unique property of each chunk in this protocol.

The replication process is carried out simultaneous with the client's chunk transfer to the Datanode server. The steps of this procedure is illustrated in figure 7 using a file containing two chunks with replication factor three.

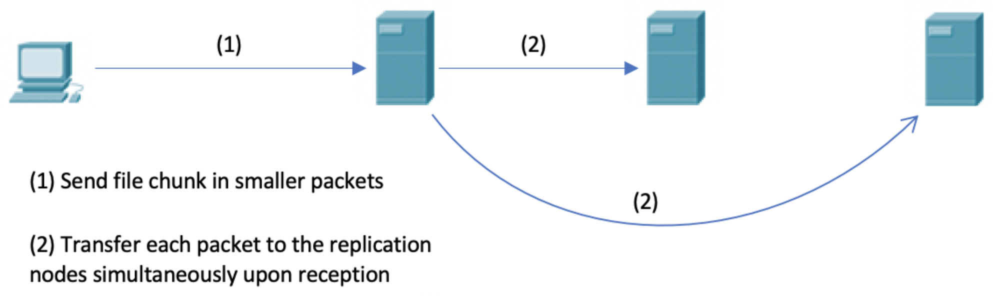

(1)
(2)
(2)
(1) Send file chunk in smaller packets
(2) Transfer each packet to the replication nodes simultaneously upon reception

Figure 7: Replication Procedure

9 Client Side

Aside from the server part of the protocol, client side matters as much. According to previous explanations, client needs to send a broadcast message to the broadcast address of the file system's network, asking for the active Namenode. Each broadcast has an expiration of three seconds meaning that if no Namenode offers to establish a connection, within three seconds, another broadcast will be triggered in order to ask for service again. This process terminates after three unsuccessful broadcast requests for service. An encrypted TCP connection will be established, if a Namenode offers service, through which the client can make its request and receive the requested results. An example of client request to create file is illustrated in figure 8.

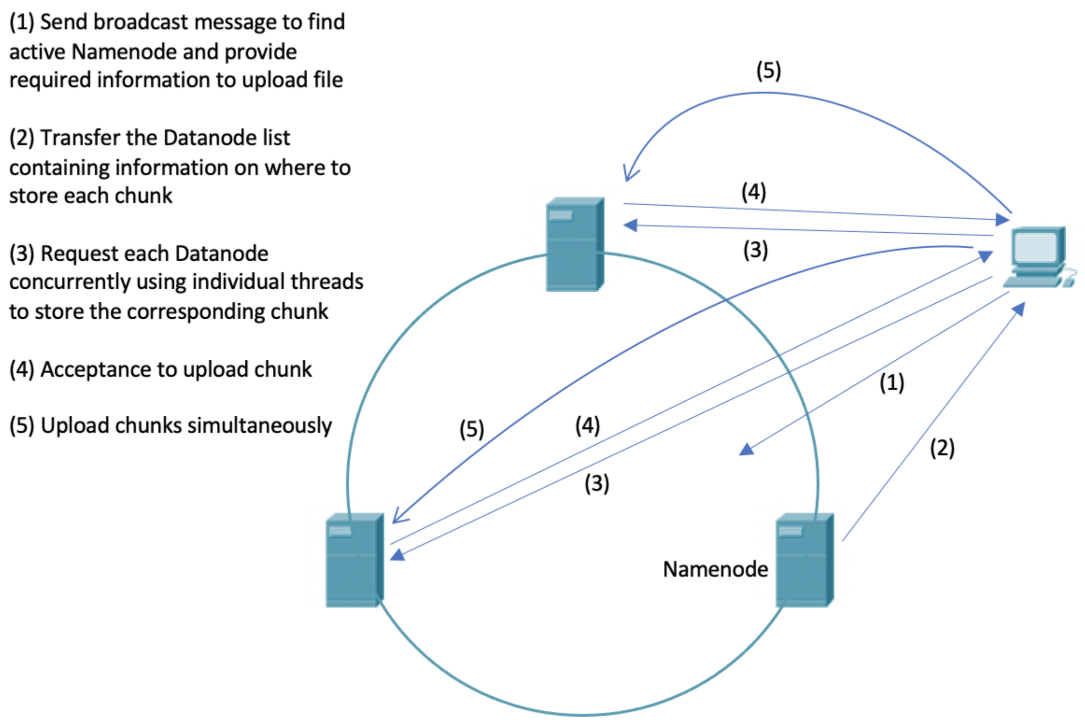
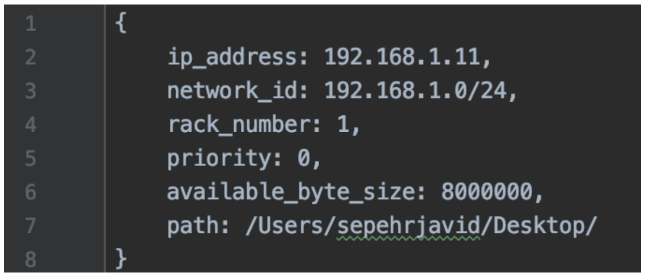

(1) Send broadcast message to find active Namenode and provide required information to upload file
(2) Transfer the Datanode list containing information on where to store each chunk
(3) Request each Datanode concurrently using individual threads to store the corresponding chunk
(4) Acceptance to upload chunk
(5) Upload chunks simultaneously
(5)
(4)
(3)
(1)
(2)
(5)
(4)
(3)
Namenode

Figure 8: Create file procedure

One of the most significant properties of the client side is that in the process of sending a file, each chunk is sent by its own thread concurrently. Since this is an IO bound process, it make the entire procedure much faster. The same applies to file reception, however, once a thread receiving chunk number *m*, has received its chunk, it will have to wait until the previous *m - 1* chunks are written into the destination file before it can be written itself.

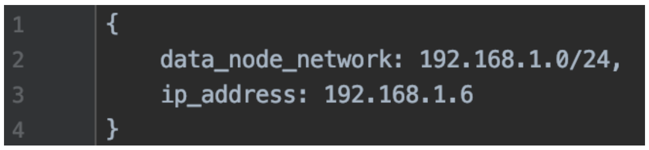
10 Configuration Files

Both the clients and the servers must have configuration files in which some settings are included. Datanode configuration file must consist of server's IP address, network address, rack number, priority (for Namenode election), available byte size which is size free to store data, and path which specifies the local path on the server where we want the chunks to be stored. Client configuration file must include client's IP address as well as the file system's network address. An example of Datanode and client configuration file could be found in figures 9 and 10.



```
{
    ip_address: 192.168.1.11,
    network_id: 192.168.1.0/24,
    rack_number: 1,
    priority: 0,
    available_byte_size: 8000000,
    path: /Users/sepehrjavid/Desktop/
}
```

Figure 9: Server Configuration Sample



```
{
    data_node_network: 192.168.1.0/24,
    ip_address: 192.168.1.6
}
```

Figure 10: Client Configuration Sample

11 Startup Procedure

In order for the server to startup, the first component to start is the peer controller. It is notable that the client controller starts concurrently, however, after parsing the configuration file, it stops for the peer controller to complete the join process and then starts the Datanode and broadcast server to serve the clients. Meanwhile, the peer controller, having parsed the configuration file, starts the *join network* process and once the database creation or reception from other nodes is completed, the peer controller notifies the client controller to start the servers. It is worth mentioning that should either of the controllers encounter any errors while parsing their configuration file, an exception will be raised before starting the servers.

12 Conclusion

As expressed, this protocol was mainly designed to address the *Single Point of Failure* issue of HDFS protocol which now many data center architects solve using a single redundant Namenode. However, given the current available resources of each Datanode, we could utilize them to act as Namenodes too which is the focus of the described protocol. It is also vital to point out that security was not a concern in the initial development of the protocol, however, minor steps towards this have been taken but many are yet to be taken for the next versions.