# Program Assignment 3

COMP310/ECSE 427, Fall 2013

## Introduction

You're expected to implement a simple Distributed File System (DFS) in this assignment. This Assignment will introduce the basic ideas about the distributed file system such as basic architecture, metadata, etc. The goal of this assignment is to teach you socket programming, Inter-processes Communication, and a bit multi-thread programming.

## Architecture

The following figure describes the architecture of the DFS in this assignment:

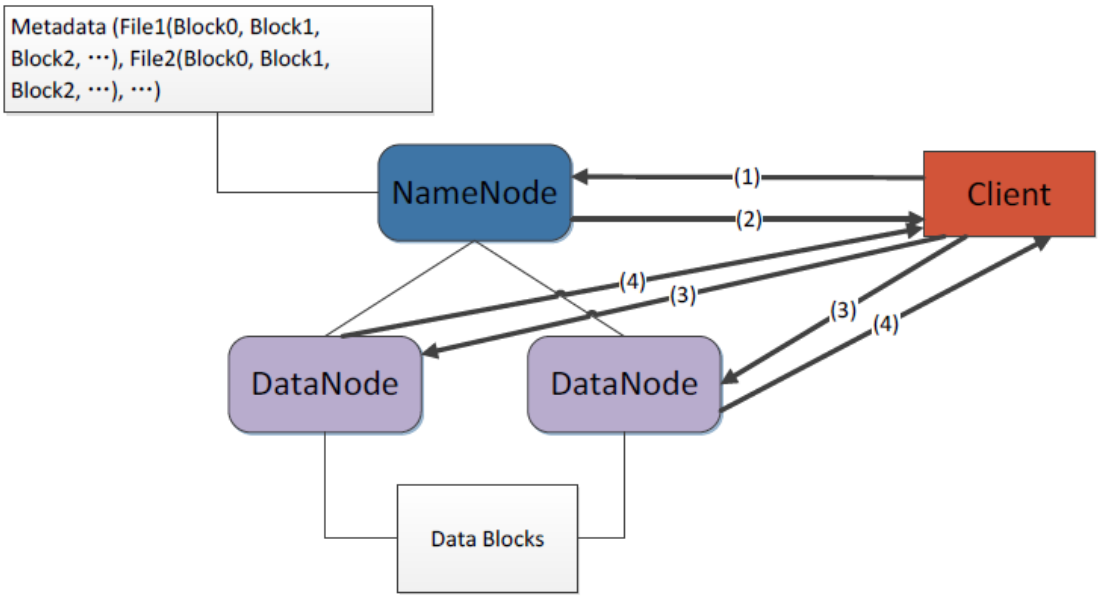


Figure 1

## Components and Functionalities

In this DFS, the file is split into multiple blocks, and these blocks are assigned to DataNodes for load balancing and fault tolerance, e.g. suppose the DFS_BLOCK_SIZE is 1024 bytes, when a user uploads a file with 4096 bytes, the whole file will be divided into four pieces which are assigned to DataNodes with certain load-balancing strategy. These blocks are very similar to "block" concept in PA1 and PA2, while they are usually with larger size.

- **NameNode**: NameNode maintains information about a map from file to blocks, the information of DataNodes and the location information of each block. When a client sends a request to

upload/download a file to/from DFS, the request is firstly routed to the NameNode. The NameNode queries the in-memory data structure (usually, we call it metadata, inode and superblock appeared in PA2 is just one kind of metadata) and response to the client with the IP address and port number of DataNodes. Depending on the request types, the client either splits the file into chunks (each is DFS_BLOCK_SIZE bytes) and uploads the chunks to the DataNodes, or downloads the chunks from the DataNodes and resemble them into a complete file.

- **DataNode**: DataNode stores the data blocks in its local file system (like what you build in PA2) and response the client requests to receive/send them from/to the client. Each data block is actually a file from the perspective of local file system, e.g. a user wants to upload a file to the DFS. We set DFS_BLOCK_SIZE as 1024 bytes, then a user file with 4096 bytes named "file_1" is split into 4 blocks, from "file_1_blk_000" to "file_1_blk_003", each 1024 bytes. User uploads each of them to the DataNodes after it has contacted NameNode to get the location of them. Each block is saved as a file by DataNodes in its local file system. Given the BLOCK_SIZE (the constant in your PA2) is 512 bytes, the file_1_blk_000 will be saved as a file with the size of 1024 bytes and occupies 2 local file system blocks.

- **Client**: client is very simple as it only sends requests to NameNode and DataNodes and receive responses from them.

## Workflow

As specified in Figure 1, there are 4 steps for the clients to upload/download the file to/from DFS

### Upload:

(1) Client sends a request to NameNode
(2) NameNode checks the available number of DataNodes, file size and DFS_BLOCK_SIZE setup, allocate this file to multiple DataNodes. The principle is a round-robin-like strategy: if the file consists of 3 blocks (in the DFS context), and there are two DataNodes in running, the first DataNode will be assigned with block 0 and block 2, the second one is assigned with block 1. When the file is appended with one additional block, the new block will be assigned to the second DataNode since its last block is assigned to the first one. The NameNode saves allocation in its internal data structure. After all of these, the NameNode sends the response which contains information about the block and its corresponding DataNode to the client.
(3) Client uploads data blocks to DataNodes;
(4) DataNodes receives the blocks from the clients

Download:
(1) Client sends a request to NameNode
(2) NameNode checks its internal data structure to finds the DataNodes storing the blocks of this file, and response to the client with the DataNode location information.
(3) Client sends download requests to DataNodes;
(4) DataNodes sends the blocks to the clients

## Socket Programming

The communication among client, NameNode and DataNode are implemented through sockets. Socket is a layer locating above the network stack implementation of your operating system. It provides the API for users to call the network functionality of your OS, e.g. sending/receiving data, listen on certain port to serve for user requests (like a web server usually listens on port 80 to provide website access service to the users), etc.

In this assignment, we transmit messages among processes through network. In network, the processes to communicate through transport layer are identified by their IP address and Port number. The IP address is like the street number of your apartment (it's the identification of the host where the process run), and the port number is your apartment number (each process which wants to use transport layer functionality applies a port number from the operating system, in this assignment, the port number for NameNode is 50070, DataNode starts from 50060). **We recommend you to read some network textbooks to get familiar with the basic concepts (The first chapter of** http://mcgill.worldcat.org/title/computer-networks/oclc/34079009&referer=brief_results (or newer versions) **is good).**

We call the process providing service (e.g. NameNode and DataNode) as server process, they usually opens a port and **listen** on it. A client with the knowledge of server process location **connects** to the server's port. The server receives the connection request from the client and **accepts** it. At this point, the connection between the client and server process is established, both end of the connection can use Send and Receive primitives in Socket to transmit messages.

Socket programming is a large topic which cannot be covered completely in this document, we recommend you to learn it from the examples, http://www.linuxhowtos.org/data/6/server.c, http://www.linuxhowtos.org/data/6/client.c.

## Multi-Thread

Thread has been taught in class. Under Linux, we usually use POSIX Thread library to develop multi-thread programs. In this assignment, we use one of the functions of pthread library, create a thread, to maintain the connections between NameNode and DataNodes.

To learn the POSIX thread library, please read: https://computing.llnl.gov/tutorials/pthreads/

## Code Structure

### Directory

/include: header files

/include/common/: shared structure among implementations of NameNode, DataNode, Client

/include/namenode/: interface of NameNode

/include/datanode/: interface of DataNode

/include/client/: interface of Client

/namenode: implementation of NameNode

/datanode: implementation of DataNode

/client: implementation of Client

/common: implementation of socket operations is here

### Data Structure

- namenode/dfs_namenode.c: **dnlist**, the array maintains the information about DataNodes, **dfs_datanode_t** is defined in include/common/dfs_common.h; **file_images**, the metadata in DFS, the information about the components of a file, **dfs_cm_file_t** is defined in include/common/dfs_common.h; int fileCount; int dncnt; int safeMode = 1**; file Count** counts the file number stored in DFS, **dncnt** counts the DataNode number, **safeMode** indicates whether there is DataNodes connecting with NameNode, 0 – yes, 1 - no
- datanode/dfs_datanode.c: **int datanode_id** = 0, the unique id of this DataNode; **int datanode_listen_port** = 0, the port number to be used by the client to send request; char *working_directory = NULL, each datanode has a working directory, which contains all data stored on it;
- include/common/dfs_common.h: **dfs_cm_datanode_status_t**, the location information about DataNodes, the message transmitted between DataNode and NameNode periodically; **dfs_cm_block_t** block information about each file block, e.g. the DataNode location, the file name which contains this block, etc.; **dfs_system_status**, the information which may be requested by the client, contains the current alive DataNodes and their locations; **dfs_cm_file_res_t** when the client request to upload/download data, this structure would be sent by NameNode to the client indicating the file name, block number, block list, etc.; **dfs_cm_client_req_t** the request sent from the client to the NameNode; **dfs_cli_dn_req_t**, the request sent from the client to the DataNode.

## Integration with PA2

To integrate with PA2, you have to modify a bit on your implementation of PA2 and copy fs.h, fs.c ext.h, ext.c to a2/;

1. replace sfs_init_storage() in ext.c with the following implementation

```
void sfs_init_storage(char *local_image_path)
{
        fh = fopen(local_image_path, "w+b");
}
```

2. add a new function in ext.c

```
void sfs_remount_storage(const char* fs_name)
{
        fh = fopen(fs_name, "r+b");
}
```

3. add a new function in fs.c

```
int sfs_reloadfs(char* fsimg_path)
{
        FILE *fp = fopen(fsimg_path, "r+b");
        fread((void *) &sb, BLOCK_SIZE, 1, fp);
        freemap = (u32 *) malloc(BLOCK_SIZE);
        memset(freemap, 0, BLOCK_SIZE);
        freemap[0] = 0x3;
        fseek(fp, BLOCK_SIZE, SEEK_SET);
        fread(freemap, BLOCK_SIZE, 1, fp);
        fclose(fp);
        sfs_remount_storage(fsimg_path);
        return 0;
}
```

## Test and Grade

There are 11 test cases in this assignment. Case 0 – 8 (test.c) is to test your implementation on inter-process communication, which is a "standalone" version of PA3, 9 – 10 (testa2.c) is to test your integration of PA2 and PA3.

To test 0 – 8, please run "make" to generate executable files, 9 – 10, "make witha2". Please note that the even you can pass all test cases in PA2, it is still possible to fail on test case 9 and 10.

Test 0 – test your connection between NameNode and DataNodes. You are supposed to create a new thread to maintain the connection between these two kinds of process. (5 points)

Test 1, 2 – with a small file upload/download request, we are testing your message handling logic. (5 points * 2)

Test 3, 4 – with a medium size file, we are testing your implementation with multiple DataNodes. (10 points * 2)

Test 5, 6 – with a large size file, we are testing your implementation about handling multiple packets (see hints). (15 points * 2)

Test 7 – with a request to read all files above, we are testing your implementation about metadata. (15 points)

Test 8 – with a request to modify file, we are testing your implementation about DataNode assignment and metadata. (20 points)

Test 9, 10 – we are testing your integration with PA2. (10 * 2, Bonus Points)

**Note: You can only get the score for Test 9, 10 if you have passed all test cases from 0 – 8.**

## Hints

The data we transmit can be too large to be contained in a single network packet, i.e. they may arrive in different packets. Please be careful to handle this problem when you fill TODOs in /common/dfs_common.c, otherwise, your program behavior can be unstable.

Do not develop under windows or Mac, even you did that, please make sure your program runs well on Trottier Machines, since we only test there.

## Submission Check List

```
zhunan@zhunan-OptiPlex-790:~/code/pa3$ ls
a2  client  common  d1  d2  datanode  include  Makefile  Makefile.common  namenode  testa2.c  test.c  test.sh  WHO
```

The WHO file looks like

```
zhunan@zhunan-OptiPlex-790:~/code/pa3$ cat WHO
Zhu Nan
1234567890
```

The first line is your full name, and the second line is your McGill ID

## Reading List


- Stevens, Fenner, and Rudoff, “Unix Network Programming volume 1: The Sockets Networking API”, 3rd Edition, Addison-Wesley, 2003.
- Andrew S. Tanenbaum, “Computer Networks, 5th edition”.
- http://hadoop.apache.org/docs/hdfs/current/hdfs_design.html