# Lab 4: HDFS and Fault Tolerance

In this lab, you will realize the power of interface. As long as your code confirms to HDFS protocol, your program could talk to Hadoop seamlessly. And you will learn how to achieve fault tolerance by replicating data.

TA rododo (409146908@qq.com) is responsible for this lab. If you have any problem, contact him by QQ, WeChat or email.

The deadline of this lab is **2018-12-16 23:59**.

## Updates

Some bugs are reported by students and I fixed them in a new commit. If you start your lab4 before **2018-12-01**, `git pull` should make your code updated. You should see a "ino_out -> ino" at "Sat Dec 1 20:17" in `git log` after you correctly update your code.

## Background

### Hadoop and HDFS

Hadoop is a well-known and widely used distributed data processing framework. It includes a distributed file system called HDFS, the data being processed and the output is stored in it. Based on HDFS, it provides a MapReduce framework, so a huge data could be partitioned and processed by multiple servers simultaneously.

In this lab, we will extend our YFS server to support HDFS protocol, so the MapReduce framework could use our YFS as storage system.

Here is some introduction of HDFS architecture: One HDFS cluster has one or multiple **namenodes**, they are responsible for managing FS metadata, including directory structure and block locations. One HDFS cluster also has one or multiple **datanodes**, they are responsible for store the actual data blocks.

Though they are both FS, there are some differences between HDFS and YFS:

1. In YFS, `inode_manager` will store superblock, inodes and indirect blocks in `block_manager`, so they could be persisted. But in HDFS, the network connection between namenode and datanode could be very slow, so it's inefficient for namenode to store data on datanode. Instead, namenode will store all the metadata on its own disk. The fault tolerance is achieved by replicate the namenode server directly.
2. Based on the characteristic of the workload, HDFS only support append write, instead of random write. This makes the design much simpler despite the data blocks is distributed.
3. In HDFS, client will ask namenode only for block locations (which datanodes store it), and connect to the datanodes directly by itself, instead of asking namenode to fetch data for it.

### SSH & SCP

In previous labs you may use VMware/VirtualBox/Parallels or some other desktop virtualization solutions to run Linux on your host OS (usually Windows). These tools provide a graphic interface so you could use a graphic editor (VS code/gedit/Sublime Text/...) in VM or copy your code to host OS easily. But in cloud

environment, it's hard to have a graphic interface due to resource limitation, so you should get used to using text interface. There are several tools to allow you connect to the server remotely for a text interface. SSH is a remote terminal so you could run arbitrary commands on server. SCP is a tool to copy files from and to servers. On Windows, you could use **PuTTY** as a SSH client and **WinSCP** as a SCP client.

Note that the test scripts in this lab will automatically compile and send your program to VMs, so you won't be bothered to transfer your code to VMs again and again. All you need to do manually on VMs is some basic configuration in Part 0 and Part 1.

## Get lab4

First, save your lab3 solution:

```
% git commit -a -m "solution for lab3"
```

Then, pull from the repository:

```
% git pull
remote: Counting objects: 43, done.
…
...
 * [new branch]      lab4       -> origin/lab4
Already up-to-date
```

Then, change to lab4 branch:

```
% git checkout lab4
```

Merge with lab3, and solve the conflict by yourself (Git may not be able to figure out how to merge your changes with the new lab assignment (e.g., if you modified some of the code that the second lab assignment changes). In that case, git merge will tell you which files have conflicts, and you should first resolve the conflicts (by editing the relevant files) and then run `git commit -a`):

```
% git merge lab3
…
```

After merge all of the conflicts, you should be able to compile successfully:

```
% make
```

If there's no error in make, 5 executable files `yfs_client`, `lock_server`, `extent_server`, `namenode`, `datanode` will be generated.

## Part 0: Setup Environment

In the Tencent Cloud preparation part, you have created 4 VMs in your account. Ubuntu 16.04 is alreayd installed, but it need some additional configuration.

### Configuration Overview

As we introduced before, a simple Hadoop cluster consists of clients, namenodes and datanodes. In this lab, we will use one VM called "app" as client machine, one VM called "name" as namenode and two VMs called "data1"/"data2" as datanodes. You should make sure the hostname and parameters are matched on every VM, otherwise the test scripts may get confused.

### Hostname

Each VM has its own hostname, managed by OS. By default, Tencent Cloud generates one for each VM, you should modify it so Hadoop and our test scripts could detect which machine it is on correctly.

Use your favorite editor to open `/etc/hostname` and set it according to VM name. You need root privilege to modify the file. Reboot the VM so the modification could take effect.

### VM Instance Name (Optional)

Besides the hostname, each VM on Tencent Cloud has a name shown in the console page. It's for you to recognize VM in console. Although there is no relation between the two names, we highly recommend you to assign the same console name and hostname to each VM.

### User

By default, an account called 'ubuntu' is created, you need to work on another account called 'cse' so the test script could work.

Execute `sudo adduser cse` and follow the prompt to create the user. You could use any password you like, as we will configure passwordless ssh later.

We assume that you will use the 'cse' account in all following parts of this lab, never use the old 'ubuntu' or 'root'.

### Allow "cse" to sudo

Which users could use `sudo` is controlled by `/etc/sudoers`. You can use `sudo visudo` to edit it. Just add `cse ALL=(ALL:ALL) NOPASSWD: ALL` at the end of the file. This will allow user "cse" to use `sudo` without password.

You can also edit this file using your favorite editor. But be careful with the content. If you make any mistake in the file, you will not be able to use `sudo` again. This means that you have no chance to fix the mistake forever. `visudo` will check the syntax before save so it can help to prevent this happends.

### /etc/hosts

By default, Tencent Cloud will generate a virtual switch to connect the 4 VMs together, and assign private IP & public IP for each. You could login on any of them and `ping` others' **private** IP, it should work normally. But in order to use others' hostname for convenience, `/etc/hosts` should be configured.

Use your favorite editor to open `/etc/hosts` on every VM and setup private IP-hostname mapping for all VM including itself. You need to make sure that the VM's hostname is only mapped to its private IP, delete the VM hostname after '127.0.0.1' if it exists. Note that you need root privilege to modify the file.

### Passwordless SSH

After setup all network stuff, you should be able to do `ssh name`/`ssh data1`/..., but it will ask you for the password, even if the password is same on all VMs. The SSH tool won't try to connect to other system with your current password.

There exists a technique called "public key authentication" could help to solve this problem. The public key authentication requires you to generate a key pair, hold private key on the client host and put public key on the target host. SSH will automatically test the two keys, if they are matched, the system allows you to login without password.

The key pair could be generated by `ssh-keygen` command on Linux or **PuTTYgen** on Windows. If you want to use PuTTY to connect to the server without password, you should use PuTTYgen to generate the key pair because PuTTY doesn't support key format of `ssh-keygen`.

#### Generate Using ssh-keygen

Execute `ssh-keygen` on any Linux/Mac host, it will generate a key pair in `~/.ssh`, public key is `~/.ssh/id_rsa.pub`, private key is `~/.ssh/id_rsa`.

#### Generate Using PuTTYgen

Start PuTTYgen, click "Generate" and follow the instruction. After generation, click "Save public key" to save public key. Click "Save private key" to save private key (.ppk format) for PuTTY usage. Click "Conversions"-"Export OpenSSH key" to save private key (`ssh-keygen` format) for Linux use.

#### Passwordless Login Configuration

After getting public key (a line of text like "ssh-rsa AAAA..."), add it to `~/.ssh/authorized_keys`. On Linux client, put private key at `~/.ssh/id_rsa`. On Windows client, specify the .ppk file in "Connection"-"SSH"-"Auth"-"Private key file for authentication".

The test script will try to manage other VMs from app VM, so you need to make sure the app VM could ssh to other VM without password.

TA will grade your lab by connecting to your app VM, so please add TA's public key to its `authorized_keys`, too.

TA's public key: `ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDPYzrRrs4AKIaEK8AxZxqCHRb1LwxBpRz61zfsw3U9zPcyyPxruiz8sK7ph9+r7Z0dA6rfJY6suO/uQosC5X7NtW0zvplePm4GzxEPKBWtGYC9H3lwU8hZBXhgcNRphRPCITImC4d7/TB+qib7cBzlRO2F9y4k2OcQRnNAiHdq8vmdvxptiT/345F732Ijqyi5hjKE66bIufcmiJi/GugqeBgs8oPnWRMCedzVs17E59nbOUIYqJLQgsz1HD/KGR9Pu3niuTd6djze9c7sFfLQRNZL5TqeWGCPXbYAQYgQMQ/angtsLdKwnh+6Cn5xgwSPBlblbSKiRfD9qkOtx2AH rododo`

### Install fuse

The test script will run `yfs_client` on your app VM, please install fuse on it. `sudo apt-get install fuse` should be enough.

### Tests

We provide a `test-lab4-part0.sh` for you to test the basic configuration. This part won't be graded. But if you feel hard to prepare your environment, you could contact TA and 10 pts will be taken from your final score of this lab.

The test scripts of this lab (not only this part) will firstly read a file called `app_public_ip` in lab directory. If the file doesn't exist, the test script will consider that it's running on app VM. If the file does exist, the test script will try to ssh to the IP in the file and run on it. Please make sure your docker environment could ssh into app VM without password and you write correct IP into the file before using any test script.

## Part 1: Deploy Distributed Hadoop

As Hadoop is designed to run on cluster, the configuration is very easy.

1. Install Java

   Hadoop is written in Java, so you have to install JRE first. Execute `sudo apt-get install openjdk-8-jre-headless` on all VMs to install JRE from Ubuntu repository.

2. Get Hadoop

   Execute `wget http://mirrors.hust.edu.cn/apache/hadoop/common/hadoop-2.8.5/hadoop-2.8.5.tar.gz` in **home directory (/home/cse/)** on every VM to get Hadoop 2.8.5, then execute `tar -xf hadoop-2.8.5.tar.gz` to extract it.

   After that, you should see a directory named `hadoop-2.8.5`. Please make sure the directory is at **/home/cse** as the test scripts will assume that and execute it.

3. Setup `JAVA_HOME`

   You need to tell Hadoop where is Java. Add `JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64` to the head of `hadoop-2.8.5/libexec/hadoop-config.sh` on every VM.

4. Setup parameters

   In order to allow app/namenode/datanode discover each other, you have to specify the hostname in some configuration files. The parameter of HDFS is stored in two XML files.

   - `hadoop-2.8.5/etc/hadoop/core-site.xml`
   - `hadoop-2.8.5/etc/hadoop/hdfs-site.xml`

   Here is an example of the configuration file:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>parameter key</name>
        <value>parameter value</value>
    </property>
    <property>
        <name>parameter key</name>
        <value>parameter value</value>
    </property>
```

```
    ...
</configuration>
```

Please set parameter according to following instructions:

- Set `fs.defaultFS` to `hdfs://name:9000` in `core-site.xml` on all VMs
- Set `dfs.block.size` to `16384` (16K) in `hdfs-site.xml` on all VMs
- Set `dfs.namenode.fs-limits.min-block-size` to `16384` in `hdfs-site.xml` on namenode/datanode
- Set `dfs.replication` to `2` in `hdfs-site.xml` on app and namenode
- Set `dfs.client.max.block.acquire.failures` to `0` in `hdfs-site.xml` on app
- Set `dfs.name.dir` to `/home/cse/hadoop-data` in `hdfs-site.xml` on namenode
- Set `dfs.data.dir` to `/home/cse/hadoop-data` in `hdfs-site.xml` on two datanodes
- Set `dfs.heartbeat.interval` to `1` in `hdfs-site.xml` on namenode/datanode
- Set `dfs.namenode.heartbeat.recheck-interval` to `0` in `hdfs-site.xml` on namenode
- Set `dfs.namenode.replication.pending.timeout-sec` to `3` in `hdfs-site.xml` on namenode

5. (Optional) Try Hadoop

    1. Format namenode

        Like normal FS, the HDFS cluster must be initialized before using. Execute `hadoop-2.8.5/bin/hdfs namenode -format` to initialize an HDFS cluster. Every time you want to reinitialize it, you should first remove the `hadoop-data` directory, then execute the above command again.

    2. Start HDFS daemons

        Now, the HDFS cluster could be started. SSH into every HDFS servers and execute `hadoop-2.8.5/sbin/hadoop-daemon.sh start namenode/datanode` to start namenode/datanode daemon. You can stop them later by execute `hadoop-2.8.5/sbin/hadoop-daemon.sh stop namenode/datanode`.

    3. Try some command

        Hadoop binary distribution provides some command line tools to manage HDFS cluster directly, you can find them in `hadoop-2.8.5/bin/`. The mostly used one is `hdfs`. Execute `hdfs dfs` to see the usage. Try the `ls`, `put`, `get`, `mkdir`, `cat`. You may use these commands to help you test and debug your YFS-HDFS implementation.

    4. Try WordCount

        WordCount is a classic problem. You must have implemented one without parallelism. And it's quite suitable for parallel processing. After split the whole text into several parts, every worker could count the words in one part, then sum them up.

        Hadoop has an example implementation, you can use it directly. But before executing the program you have to prepare some input. Type some random words into a text file or download an english novel, then use `hdfs dfs -put` to send it into HDFS. After that, execute `hadoop-2.8.5/bin/hadoop jar hadoop-2.8.5/share/hadoop/mapreduce/hadoop-mapreduce-`

`examples-2.8.5.jar wordcount <input> <output>`. You could see the output using `hdfs dfs -ls` and download it using `hdfs dfs -get`

5. Try fault tolerance

   In a production HDFS cluster, there may be hundreds or thousands of datanodes. According to the course, it's frequent that one of the datanodes is down. In order to avoid data loss, HDFS will replicate every block three times on different datanodes (Although in our configurtion there is only two datanodes).

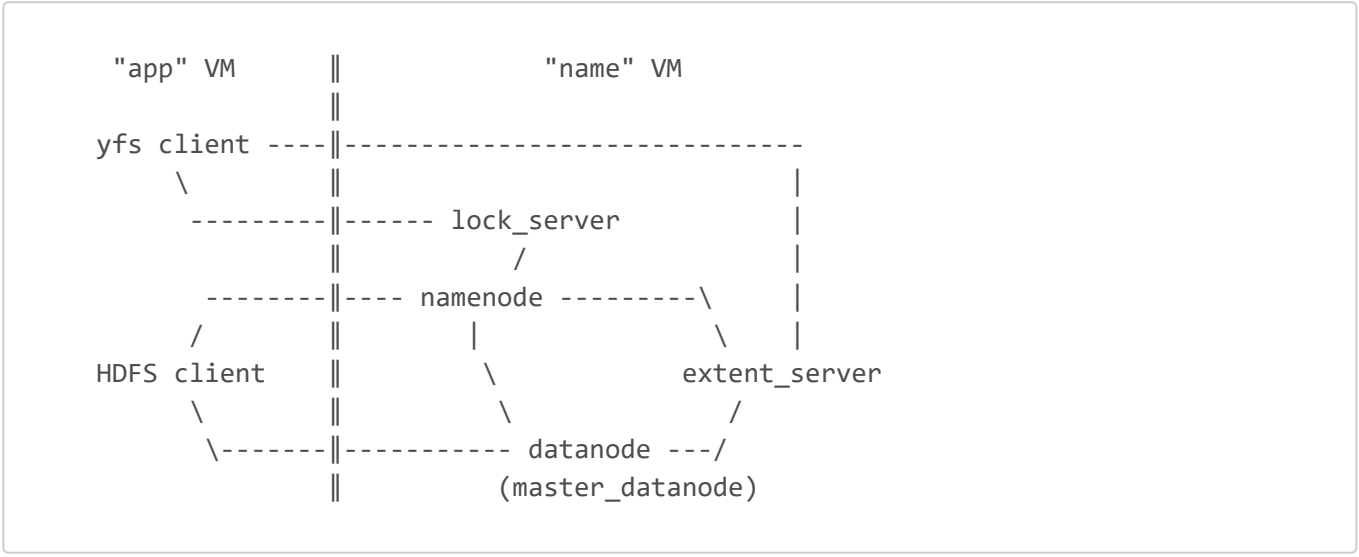
   You could manually crash one of the two datanodes by shuting down it and delete `hadoop-data` directory. After you restart the datanode, namenode will observe the crash and replicate the missing blocks again. As the file is very small, the recovery is very quick. By using `hdfs fsck / -blocks`, you could see which blocks are under-replicated. If the other datanode crashes before the recovery completes, the block will be permanently lost.

6. Test your deployment with `test-lab4-part1.sh`!

## Part 2: Implement a Single Node YFS-HDFS System

In this part, you need to implement a single node YFS-HDFS system, without any fault tolerance mechanism. The system could be used as old YFS and any HDFS-compatible client. You could even access files created by YFS in HDFS and vice versa!

The architecture is:




### HDFS architecture details

In HDFS, file data is organized as blocks. Each block has a global-unique block id. Namenodes are responsible for allocating block ids, keeping file offset-block id mappings. Datanodes only store the blocks, without knowledge of which kind of data is in it.

### How to combine YFS and HDFS

In YFS, both `inode_manager` (in `extent_server`) and `yfs_client` is responsible for manager FS metadata. This makes it very hard to extract the logic into a single, standalone namenode program without tampering original design. We have no choice but expose more inode level interface in

`extent_protocol`(`append_block`, `get_block_ids`), and create a dedicated `namenode` server, which use both old and new interface to provide full HDFS namenode service.

In order to make it compatible with `yfs_client`, the directory structure part in your `namenode` must confirm to your design in `yfs_client`. You should always reuse (call or copy) code in your `yfs_client` if possible.

Some block level interface is required to implement a `datanode` server. HDFS datanode protocol access data block directly with its id, so `read_block` and `write_block` should be added to `extent_protocol`.

Here is a summary of new extent protocol interface:

- `append_block`: Given an inode number, allocate and append a block to the inode and return its id. Note that in HDFS, namenode only manage the metadata, so the actual data (even the size) won't be given in this request.
- `get_block_ids`: Given an inode number, return id of all data blocks of the file.
- `read_block`: Given a block id, return the data of the disk block.
- `write_block`: Given a block id and data, write the data to the disk block.
- `complete`: Given an inode number and size, this request indicates that the writes to the file is finished, so the metedata (file size, modification time) in inode could be updated safely.

And you have to implement the namenode/datanode logic in `namenode.cc`/`datanode.cc`. Here is a summary of all functions you need to write:

- `NameNode::GetBlockLocations`: Call `get_block_ids` and convert block ids to `LocatedBlock`s.
- `NameNode::Complete`: Call `complete` and unlock the file.
- `NameNode::AppendBlock`: Call `append_block` and convert block id to `LocatedBlock`.
- `NameNode::Rename`: Move a directory entry. Note that `src_name`/`dst_name` is entry name, not full path.
- `NameNode::Mkdir`: Just call `mkdir`.
- `NameNode::Create`: Create a file, remember to lock it before return.
- `NameNode::Isfile/Isdir/Getfile/Getdir/Readdir/Unlink`: The same as the functions in `yfs_client`, but the framework will call these functions with the locks held, so you shouldn't try to lock them again. Otherwise there will be a deadlock.
- `DataNode::ReadBlock/WriteBlock`: Call `read_block/write_lock` to read/write block on extent server. Be careful that client may want to read/write only parts of the block.

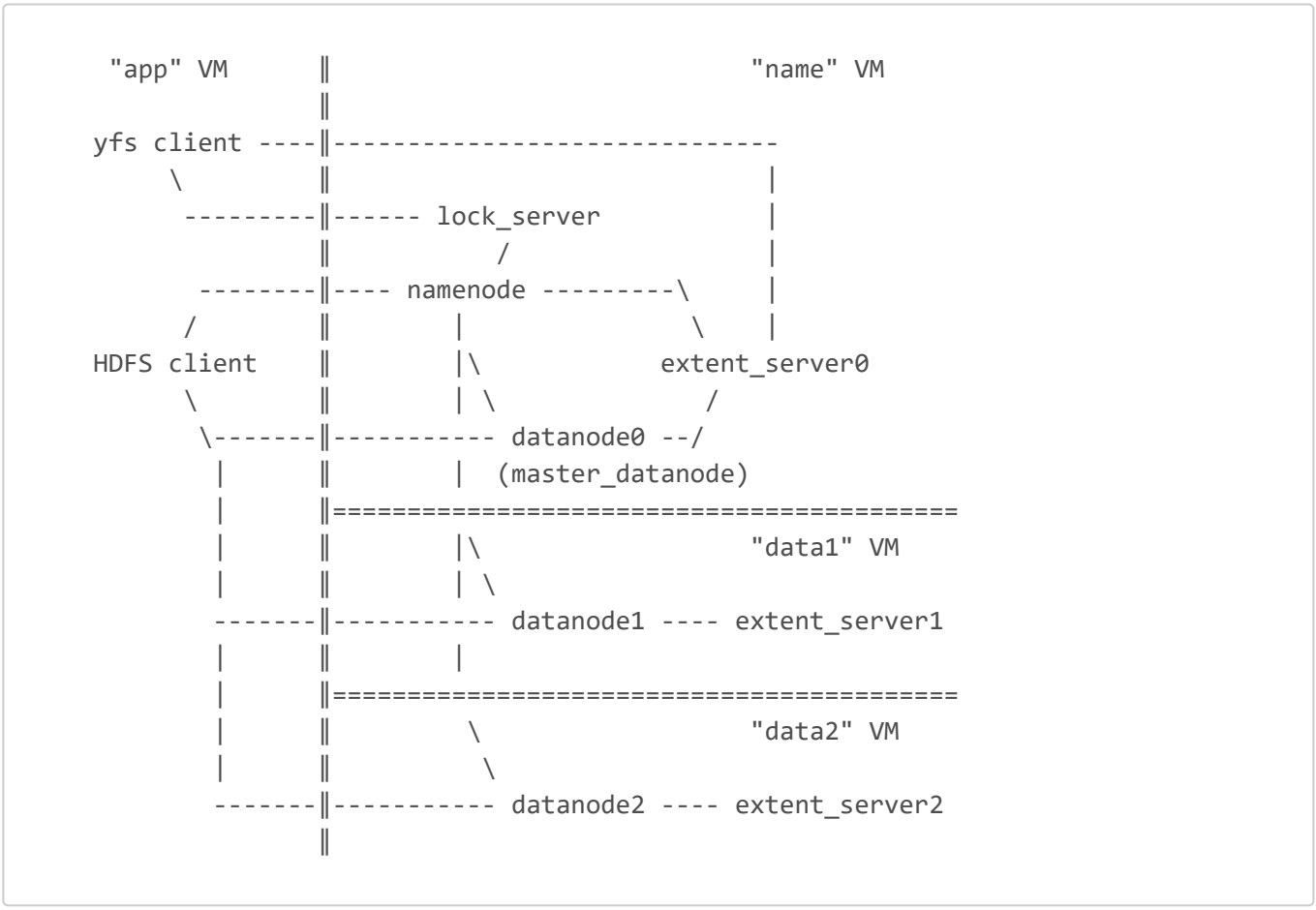
### Hints

- `LocatedBlock` consists of block id, block offset in the whole file, block size and block location. You should be careful about the size of the last block. If the last block is not full (file size % BLOCK_SIZE != 0), you should report the actual data size instead of BLOCK_SIZE. The block location is used in Part 3, in Part 2 you can simply pass `master_datanode` to the constructor.
- Be careful with the lock! Never try to lock a file twice. A typical operation sequence is Create -> AppendBlock -> write to datanode -> AppendBlock -> write to datanode -> ... -> Complete.
- The output of your programs will be collected to `extent_server.log`, `namenode.log`, `datanode.log`, `yfs_client.log`. If you add more `printf`, remember to call `fflush(stdout)`.

## Part 3: Fault tolerance with replication

In this part you need to add replication mechanism to your YFS-HDFS for fault tolerance. And we will add two more datanodes to the system.

The architecture is:




## HDFS architecture details

In HDFS, namenode is responsible for managing all datanodes, distributing and replicating blocks among them. All datanodes will register themselves on namenodes so namenodes know where the blocks could be placed. Namenodes need to track datanodes state. If one datanode fails to send heartbeat normally, all blocks on that datanode should be relocated to other datanodes.

## How to implement replication in YFS-HDFS

In Part 2, we have only one datanode running on `name`, represented by `master_datanode` in namenode. It is directly connected to the extent server used by namenode and `yfs_client`, so even the file is written by `yfs_client`, the master datanode could read it. We treat it as a never-fault server, so the replication management is much simpler.

In order to accept more datanodes and allow them to die, firstly we should learn how to track the state of datanodes. Once a datanode is started, it will register itself to namenode, and `NameNode::RegisterDatanode` is called with the ID of the datanode. You could save the ID into any member variables/data structures so you could connect to it later. Another thing you need to do is monitoring the datanode by heartbeat. Create a thread in datanode to send heartbeat periodically and check the heartbeat in namenode. The only requirement is that you should detect the death of a datanode within 5 seconds, otherwise the test will report a timeout. `NameNode::GetDatanodes` is used to get all **live** datanodes. Don't return any dead datanodes in it!

After finishing the heartbeat mechanism, you should replace the `master_datanode` in your construction of `LocatedBlock` with list of live datanodes, so the client will know it should write to all of the datanodes and could read from any of them.

When a new datanode is started after some data is written into YFS-HDFS, the written blocks should be replicated to the new datanode before it is reported as live datanode. We provide a function `ReplicateBlock` to help you complete the replication. But it's your responsibility to find out which blocks should be replicated. Simply replicate all blocks on extent server is inefficient and may cause the test timeout. You should recover a new datanode in 10 seconds.

You should notice that the data written by `yfs_client` won't be correctly sent to all of the datanodes. It's quite hard to monitor the write operation from `yfs_client`, so we simply ignore this case in this lab. The test script won't test the replication management on files written by `yfs_client`. But if you are interested, you could try to implement it.

### Hints

- You could finish this part step by step (register -> heartbeat -> replicate -> new datanode recovery), as the test script will test them one by one.
- We provide a utility function called `NewThread` to help you run a member function in a new thread. Assume `obj` is a pointer to object of class `ClassA`, to call `Fn(arg1)` on it in a new thread, use `NewThread(obj, &ClassA::Fn, arg1)`. Remember to include `threader.h`.
- As only the files created/written by HDFS should be managed, you could record all blocks in a list in namenode, so you could easily find out which blocks should be sent when a new datanode started.
- Never return a new datanode in `LocatedBlock` or `GetDatanodes` before the recovery is finished. Otherwise the client may try to read from it and got unrecovered block.
- `RegisterDatanode` is called in the new thread. Do any long operation (like replicate blocks to the new datanode) in it and don't worry about making the system stuck.
- `master_datanode` is a never-fault server, so you could ask it to replicate any block to any other datanode at any time.
- You can send heartbeat by calling `SendHeartbeat` after connect to namenode in `init`. `DatanodeHeartbeat` will be called on the namenode with corresponding datanode's ID.

## Handin

Like what you did in previous lab, just

```
make handin
```

This should produce a file called `lab4.tgz` in the directory. Change the file name to your student id, Then upload `lab4_[your student id].tgz` to ftp://SJTU.Ticholas.Huang:public@public.sjtu.edu.cn/upload/cse/lab4/ before the deadline. You are only given the permission to list and create new file, but no overwrite and read. So make sure your implementation has passed all the tests before final submit.