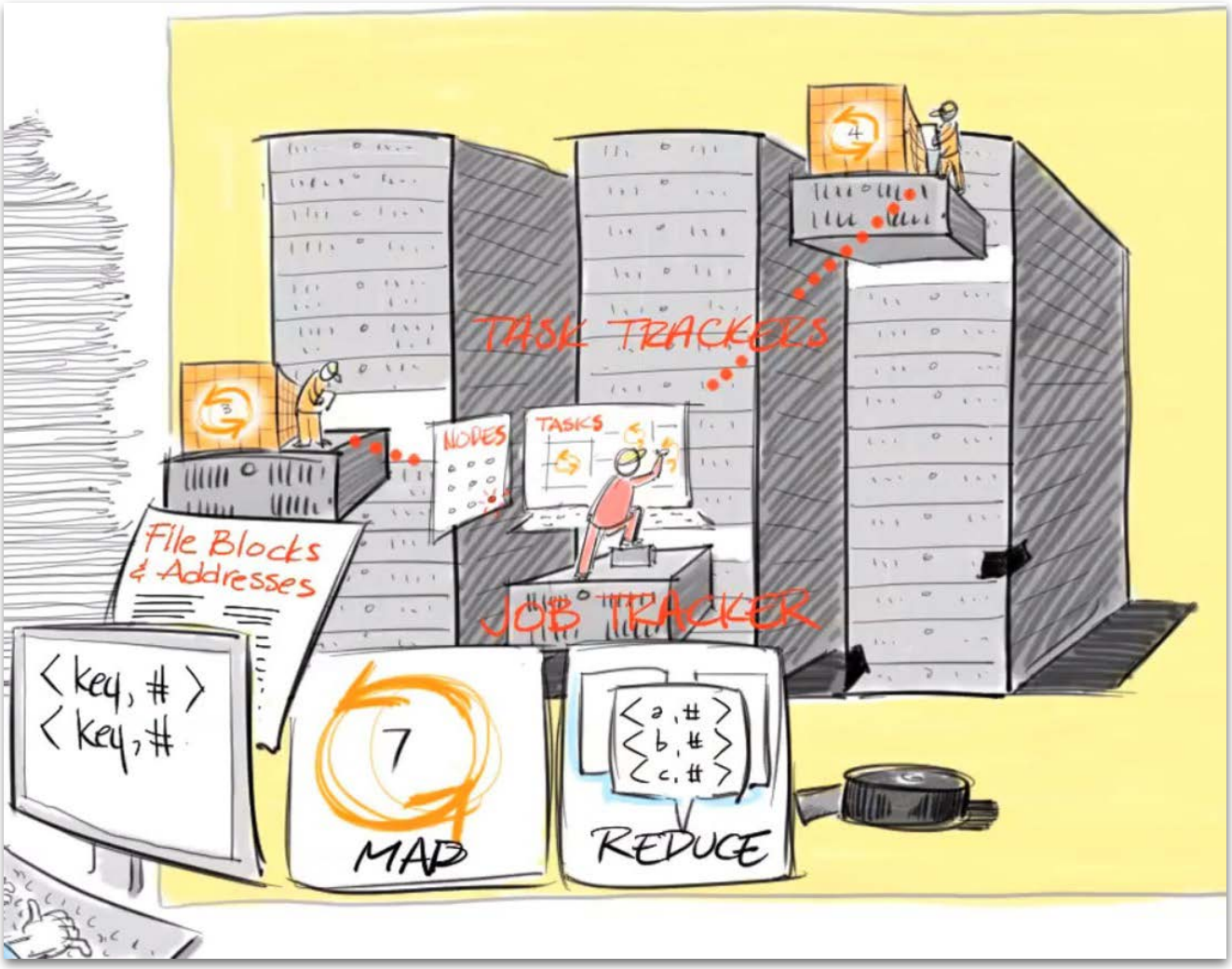


MapReduce Engine

Menglong He (menglongh)

Sidi Lin (sidil)

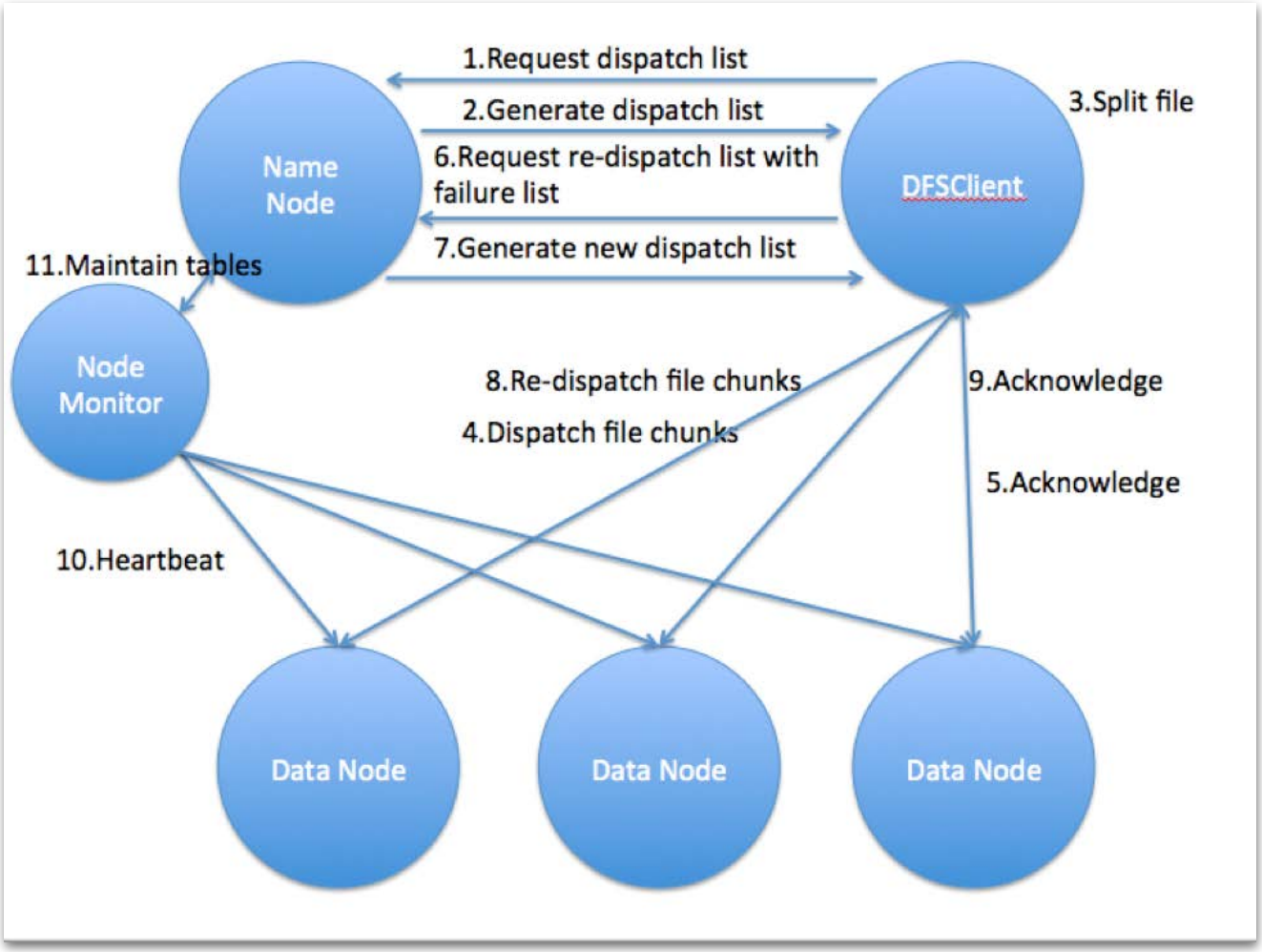
July 17, 2014

CONTENTS

For your attention : This document doesn't contain detailed explanation of methods and parameters. Please check out the APIs in MapReduceEngineAPI folder (index.html) for further reference.

OVERVIEW

DFS



DFS is basic infrastructure for running this system. It provides a distributed data storage environment for all the participants. DFSClient can upload, download or delete files on DFS. DFS consists of three parts: DFSClient, NameNode and DataNode.

DFSClient

DFSClient provides a command line tool for user to put, get and remove files that are stored on DFS. When using DFSClient to put files onto DFS, DFSClient will be responsible for chopping a file into proper size using a method provided in util.IOUtil and guaranteeing the file is successfully upload to DFS. By implementing DFSClientInterface class, DFSClient provides a RMI service for data nodes to acknowledge itself after a chunk is successfully uploaded. Also, DFSClient will be instantiated by reducer process to upload the job result onto DFS after MapReduce job is finished.

NameNode

NameNode should be assigned before running this system (by setting up the configuration file). It manages all the data node information including data node status and available storage slots. as well as how files are distributed among all the data nodes. It provides RMI services for other nodes to query and change the status of DFS. In the case when one or more data node failed, it will try to ensure the number of file replicas in DFS and balance the storage load among data nodes. In order to monitor the status of data nodes, it will instantiate a NodeMonitor to heartbeat each data node see if it is down in a regular manner.

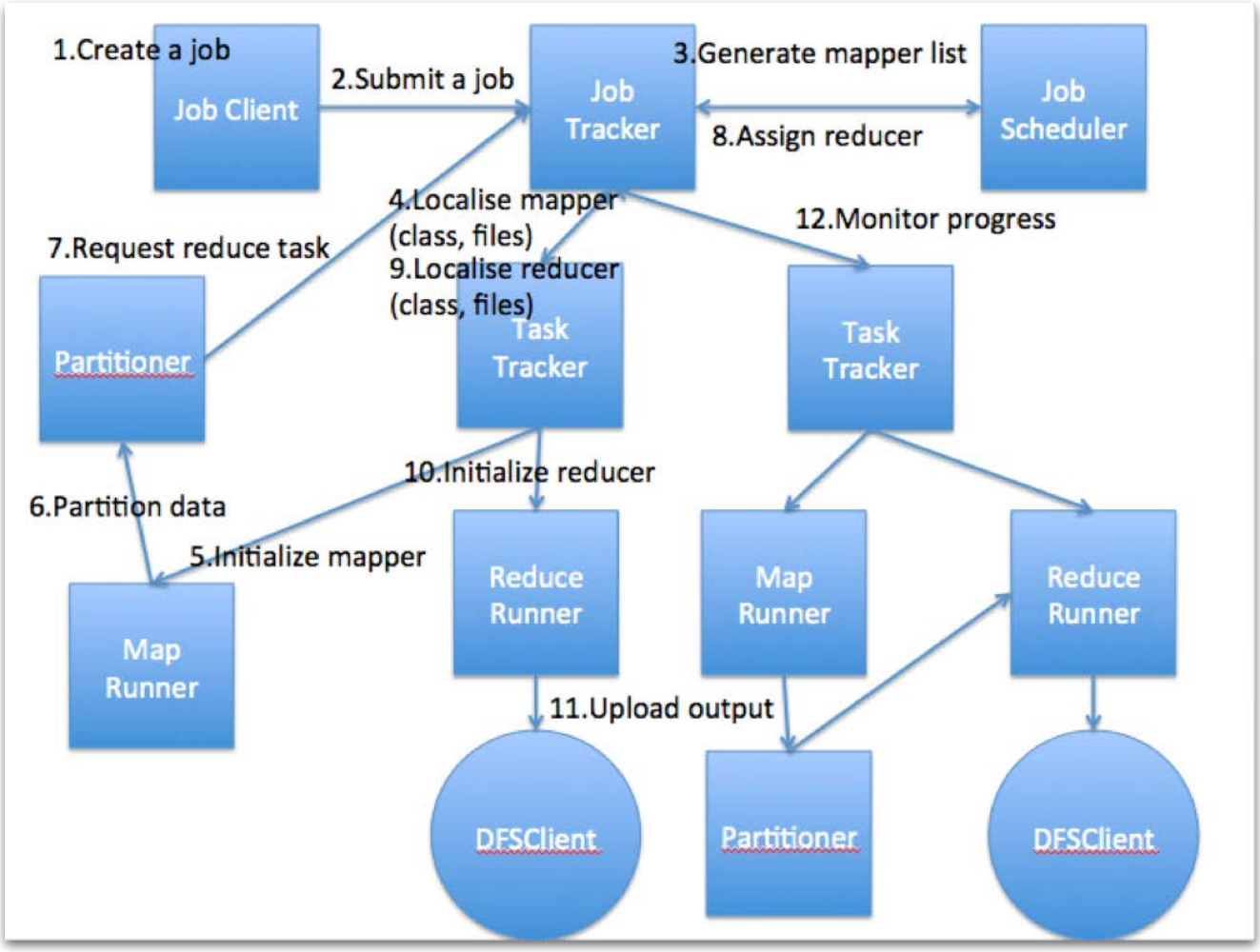
DataNode

DataNode are servers that mainly used for file storage. It maintains a table of slots availability to indicate how many storage space is left. Also it provides RMI services for name node to transfer file chunks among different data nodes.

NodeMonitor

Node Monitor is a persist thread that monitors all the DataNodes status and keep an eye on files status and integrity. It will heart beat each DataNode for a certain period of time. Whenever it detects a failed DataNode, it will retry for a certain amount of times to make sure if that DataNode is down for real. If a DataNode is confirmed failed, then it will clean up all related information and try to duplicate all the data resides on that DataNode. Also it will inform JobTracker to recover from failure.

MapReduce



Once the user create a MapReduce job, the JobClient will read the configuration file and call the JobTracker to submit the job. Meanwhile, the JobClient may monitor the job status for every five seconds. JobTracker calls the JobScheduler to select best nodes for Map, then start a Map task. TaskTracker firstly downloads the Mapper class from JobTracker, then start different MapRunners according to the arrangement. In the MapRunner, Partitioner will be called to partition the OutputCollector. After all the Mapper tasks for a specific job is done, the Reduce task starts. JobScheduler will be called again to choose best nodes for Reduce Tasks. Then the Reduce process begins and Merger is called. Once a Reduce task is done, its output will be uploaded

to the DFS. To monitor all TaskTrackers, JobTracker will transmit a heartbeat to them in every 5 seconds. Since different failures may occur, mechanisms are applied to deal with failures.

MapReduce consists of four parts: Format, JobTracker, TaskTracker and MapReduce configuration.

Format

The format part provides the basic format of the input and output.

In the MapReduce, the Key-Value pair is the most important concept to wrap the data and its characteristics into an object. KVPair class is the carrier of this concept.

In order to satisfy the special input and output of each step, the system provides 5 classes: InputFormat, LineFormat, WordFormat, PairFormat and OutputFormat. InputFormat and OutputFormat are the interfaces to define the methods and fields.

Finally, the system provides the data structure to cache or store the intermediate results. It only contains MapperOutputCollector and ReducerOutputCollector.

JobTracker

JobTracker is running on the master node which is used to coordinate the slave nodes to work and communications among each other. It works among different java machines.All the communications are implemented by using the RMI of Java. Each time, the programmer needs to write three methods: XXXMapper, XXXReducer and XXX (configuration and Main function) classes. When the user run the main function, it will invoke the JobClient to submit the job to the JobTracker. The JobTracker will distribute the different tasks (including mapper task and reducer task) on different nodes. Then it will monitor all the slave nodes using the heartbeat way. It mainly communicates with the TaskTracker.

At the same time, the JobTracker handles the fault tolerance. It can handle the two conditions: mapper failure (including node down) and reducer failure (including node down).

TaskTracker

TaskTracker can run on any node including master node and each node can run just one instance. TaskTracker's only works for one java machine for any IP. It actually start a new thread to run the mapper or reducer work. It waits for the Mapper Runner or Reduce Runner's invoke and update the status of the job and node.

At the same time, the TaskTracker handles two conditional failure: fetching the chunk error and fetching the partition contents which are the results of the map step.

MapReduce Configuration

The framework provides the human-readable and human-editable configuration file. The mapred.conf file contains the 17 key-value pairs. It contains: JobTracker's server port, registry port, service name and IP; TaskTracker's server port, registry port and service name; MaxTaskPerNode which represents the maximum number of tasks on each slave node ; jobMaxFailureThreshold which represents the maximum failure times; mapperChunkThreshold

which defines the maximum chunks each mapper can handle; localWeight and globalWeight define the weight of different nodes, it only uses in selecting the best node to do the mapper worker; partitionNums represents the partition number of the mapper's result.

Utilities and Configuration

IOUtil

IOUtil provides general and specific file manipulation methods for all the modules in this system. It provides byte-based, line-based and object-based read and write method as well as chunk operations towards files.

PathConfiguration

PathConfiguration is used to define the path of configuration file. These configuration files contains most of the default settings of DFS and mapreduce such as the number of copies to maintain and default service ports. These settings can be customized in dfs.conf and mapred.conf.

SYSTEM REQUIREMENTS AND LIMITATION

Requirements

1. The system must be operated on Linux or Unix system, not on Windows;
2. The disk must have enough space to store the upload chunks and intermediate files according to your setting;
3. The machine must have enough memory according to your setting;
4. NameNode must be designated in configuration before running the system;
5. Ant (we use the 1.9.4 version) will be used to compile class files;
6. All classes should be compiled using ant before running this system.

Limitation

1. Assume the master node (NameNode and JobTracker) runs well all the time;
2. The number of replication can not over the total number of all nodes;
3. NameNode must be guaranteed running at all time.
4. DataNode will be treated as new node after recover from failure.
5. For each server there could only be one running instance for each DataNode, NameNode and DFSClient.
6. Number of file chunks to be uploaded should less than maximum capacity of DataNodes.

KEY FEATURES

DFS

DFSClient Console

DFSClient console provide GUI for user to put/get/remove files on to or from DFS.

Reliable File Upload

In order to put a file on to DFS, user can use “dfs put <file>” command.

This command will first try to calculate splitting information of the file to be uploaded and request a dispatch guidance list from NameNode. Then it will extract one chunk at a time from the file and dispatch it to selected DataNode. The process will continue only after acknowledge is received.

If selected DataNode fails during upload, DFSClient will try util retry threshold is met, then DFSClient will move on to next DataNode. These skipped upload will be send back to NameNode for a new distribution table and these chunks will be uploaded to new DataNodes until all chunks are successfully dispatched.

Finally, DFSClient will acknowledge NameNode to update corresponding file information.

Acknowledge

The stability and robustness of this DFS is based on acknowledgements. Whenever there is a file transferring operation, acknowledgement (up to three trips) will be sent between two parties so that the succeed of operation could be confirmed.

Failure Handling

For file uploading failure, DFSClient will be responsible to make sure all the chunks are dispatch correctly. If failure happens, DFSClient will request a redistribution table from NameNode and dispatch again. If this system runs out of available DataNodes, this operation will fail.

Also, if any DataNode is dead, the Node Monitor will try to make copies of those data resides on the dead node to another living node.

Heartbeat

Heartbeat is used by NodeMonitor to update real-time status of DataNodes that are registered in this system. Heartbeat will be performed every certain period, which is designated in configuration file, and maintain available slots and health condition tables of DataNode in real-time.

Load Balancing

To determine which DataNode will be used for storing file chunks, NameNode will first pick out those healthy DataNode that does not possess the file chunk and with highest available slots. In the case when dispatch is failed, there will be an exclude list which will contains those DataNodes that seemed “dead” to client. In this way we can make the most out of balancing the storage while ensuring the quality of service.

Connection Caching

For DFSClient, whose is responsible for dispatching file chunks, it is crucial that we can avoid setting up connection to DataNodes, since there could easily be tens of DataNodes it needs to upload file chunks to. So we design a connection pool to cache all the connection to DataNodes and it could be reach out easily whenever needed.

MapReduce

Load Balancing

In order to avoid a node buried with too much tasks, the system supports load balancing function. JobScheduler is responsible for selecting nodes with lightest loads. In map process, since map node should get chunks from DataNodes, we assigned local weight and global weight while selecting node for map (e.g. local weight = 0.3, global weight = 0.1). If the needed chunk is currently on the node, the cost of network can be avoided. Based on this, we can select best local node as well as best global node with minimal tasks and pick the better one between them. In reduce process, reducers are top k lightest nodes (k is the number of partitions).

Job Localization

Every node has its own TaskTracker. Once the TaskTracker receives a task (map/ reduce) from JobTracker, it will save the map / reduce content into local storage. In this way, all mappers using the same class can just read the class content locally. This will significantly reduce the transmission cost.

Notification to JobTracker

Notification makes sure that reduce tasks can be executed at the correct time. Each TaskTracker maintains a Hashmap to record the status of tasks running on it. When all mappers running on the node have been finished, TaskTracker will send a notification to the JobTracker. Then the TaskTracker will check whether all mappers for the specific job have finished. If so, it will start the reduce phase for the job. In the same way, when all reduce tasks on a node are done, TaskTracker will also notify the JobTracker.

Errors and Failures Handling

There are four kinds of errors during the MapReduce process. (1) The assigned map node is out of service. To deal with this, JobTracker will collect unfinished map tasks running on it and find other suitable nodes to execute the map tasks. (2) The assigned reduce node is out of service. JobTracker will collect unfinished reduce tasks running on it and find other suitable nodes to execute the map tasks. (3) When the map node wants to connect to the data node and fetch chunk from it, then finds the data node is out of service. To deal with this, TaskTracker will redistribute the chunks and restart some new threads to run the work to the end. (4) When the reduce node wants to connect to the map node to fetch partition files from it, then finds the map node is out of service. Since the partition file is stored in the map node locally without having any copies, the TaskTracker will simply terminate the job in this circumstance. All failures will be treated for 3 times. If the failure times exceed a specific number (this can be set in the configuration file), the job will be terminated.

Heartbeat

There is a heartbeat timer running on JobTracker. For every 5 seconds, it will transmit heartbeat to all healthy nodes and ask for their task status. When the JobTracker received task status from mappers and reducers, it will update the information on it’s own table.

Partition

In the partition process, the mapper output result will be distributed to different partitions based on their hashcodes. After partitioning, same keys will be distributed to the same partition, and each reducer can deal with a specific partition.

Merge

In the merge process, TreeSet is used to maintain the order of the key contents in each partition. The merge result is used for calling the reduce function written by the user.

Input and output format

The facility supports 2 kinds of formatting. The first one is to format the input contents into the Key-Value pairs. Each Key presents the line Number and value represents the whole line. The target of this class is to the

original .txt or other raw materials. The second one is to format the Format the input contents into the Key-Value pairs. The target of this class is to the "key value /n" format file, such as the Mapper's output.

ThreadPool

In the whole MapReduce process, we use thread pool to manage threads. A thread pool is a managed collection of threads that are available to perform tasks. Thread pools usually provide: (1) Improved performance when executing large numbers of tasks due to reduced per-task invocation overhead. (2) A means of bounding the resources, including threads, consumed when executing a collection of tasks. (3) In addition, thread pools relieve you from having to manage the life cycle of threads. They allow to take advantage of threading, but focus on the tasks that you want the threads to perform, instead of the thread mechanics.

FUTURE IMPROVEMENT

DFS

CheckPoint of NameNode

NameNode is very possible to failed in a relatively long time period. So without doing a duplicated NameNode, it would be very helpful to create checkpoint of NameNode so that when the NameNode is recovered from failure, it could still preserved the file information and DataNode details. In this way our system would possess to some extent, capability to recover from NameNode failure.

Clear The Cruft Automatically

When the system runs out of available DataNodes because of DataNode failure, even though those slot spaces will be reclaimed in NameNode, those chunks which have already been uploaded will remains there. So it would be good to have some mechanism to clean up those cruft.

Also when a DataNode is recovered from failure, or a certain file chunk is diagnosed lost, this auto clean up will be very useful to clean up all the old files or broken parts since they are already been duplicated to other DataNode or useless to the system.

Parallelize File Upload

Due to design trade-offs, in order to guarantee succeed of upload, it is difficult to upload a chunk to multiple DataNodes at the same time. Hopefully we can achieve this in later version of this system.

MapReduce

Build Much More Robust Fault Tolerance Mechanism

Currently, this framework can support 4 failures in MapReduce phrase.

All these failures happen in building connection phrase. In the future, the system will enable handling the failure happening after the connection has been established. For instance, if the Mapper node turns down when it is running the mapper task.

Kill Jobs Using the Console

Right now, the system will kill the job automatically if the job fails or the node is down. But it does not support the administrator to kill the job by hand. In the future, we will build the console to support the programmer to check the running jobID and support type the command to kill the job. At the same time, we can add the functions to terminate all the nodes connect to the master nodes.

Garbage Collection

Currently, our system cannot support automatically cleaning up the memory. Thus, we need to restart the system in a given period of time. In the future, we will add functions to clean up the memory intelligently.

DEPLOYMENT GUIDANCE

Configuration

In order to run the framework smoothly, the administrator needs to set the configuration file firstly. We recommend the administrator to update the different port number according to the actual condition. The following content will describe the meaning of these fields.

For **dfs.conf** file, there are 20 fields:

1. clientRegPort: DFSClient RMI registry port;
2. clientPort: DFSClient RMI server port;
3. clientPortForTaskTrack: The TaskTracker will use this port to build DFSClient RMI. It is the server port;
4. clientRegPortForTaskTrack: The TaskTracker will use this port to build DFSClient RMI. It is the registry port;
5. clientServiceName: DFSClient RMI service name;
6. maxChunkSlot: the maximum number of the chunk slots of each datanode, the default is 20;
7. maxChunkSize: the maximum chunk size of one chunk, the default is 10M;
8. nameNodeIP: NameNode RMI's IP;
9. nameNodeRegPort: NameNode RMI's registry port;
10. nameNodePort: NameNode RMI's server port;
11. nameNodeService: NameNode RMI's service name;
12. dataNodeRegPort: DataNode RMI's registry port;
13. dataNodePort: DataNode RMI's server port;
14. dataNodeService: DataNode RMI's service name;
15. replicaNum: the number of each chunk's replication;
16. heartbeatCheckThreshold: this number represents the retrying times if the heartbeat cannot ping to the node, the default is 3 times;
17. heartbeatInterval: the interval of heartbeat, the default is 3 seconds;
18. dataNodePath: this path is used to store the uploaded files;

19. chunkTranferRetryThreshold: this number represents the retrying times if the chunk transferring failure;
20. ackTimeout: this number is the acknowledge time from the data node to the client. The default value is 10 seconds.

For **mapred.conf** file, there are 17 fields:

1. jobTrackerIP: JobTracker RMI's IP;
2. jobTrackerPort: JobTracker RMI's server port;
3. jobTrackerRegPort: JobTracker RMI's registry port;
4. jobTrackServiceName: JobTracker RMI's service name;
5. jobUploadPath: the JobTracker will download the mapper and reducer classes from the client using this path;
6. taskTrackerPort: TaskTracker RMI's server port;
7. taskTrackerRegPort: TaskTracker RMI's registry port;
8. taskTrackServiceName: TaskTracker RMI's service name;
9. partitionFilePath: this path is used to store the partition from the mapper;
10. reduceResultPath: this path is used to store the result of reducer;
11. mapResTemporaryPath: this path is used to store the files from different mapper nodes;
12. maxTaskPerNode: the maximum number of tasks of each node;
13. jobMaxFailureThreshold: the maximum number of failure times;
14. mapperChunkThreshold: the maximum number of chunks for one map to handle;
15. localWeight: the weight of nodes which contains the specific chunk;
16. globalWeight: the weight of nodes which not contains the specific chunk;
17. partitionNums: the partition number of the mapper result.

Compile

Setup the Ant Environment

export ANT_HOME=/Users/XXX(account name)/apache-ant-1.9.4

export PATH=${PATH}:${ANT_HOME}/bin

Compile

You need to go to the directory which contains the build.xml folder firstly.

ant -file build.xml

After you executed the command, you can see a new folder called bin. In the following steps, you have to change the directory to "/MapReduce/bin". There is an important rule: **START NAMENODE AND JOBTRACKER FIRSTLY**.

Deploy Master Node

The following class are constant thread, so you need to open a new terminal for each class.

This master node always undertake the coordination and communication with other slave nodes. This node always just deploying the NameNode and JobTracker. If you want this node not only play as master but also one of the slave nodes, you can deploy the DataNode and TaskTracker and others functions as the Slave Nodes' operation showing. In this part, we assume you just deploy the NameNode and JobTracker only.

You need to execute the following command:

to start NameNode: **java dfs.NameNode**

to start JobTracker: **java mapred.JobTracker**

Deploy Slave Nodes

These slave nodes always undertake the mapper and reducer work. Copy the bin folder to the slave nodes and you need to execute the following command:

to start DataNode: **java dfs.DataNode**

to start TaskTracker: **java mapred.TaskTracker**

TWO EXAMPLES

WordCount

Describe

The word count is the number of words in a document or passage of text. Word counting may be needed when a text is required to stay within certain numbers of words. This may particularly be the case in academia, legal proceedings, journalism and advertising. Word count is commonly used by translators to determine the price for the translation job. Word counts may also be used to calculate measures of readability and to measure typing and reading speeds (usually in words per minute). When converting character counts to words, a measure of 5 or 6 characters to a word is generally used.[1]

How to run the WordCount

1) You can use the default files called : **pg_01 and input**. If you want to use your file, you can copy your file in this folder or you can copy them into any directory but you need to remember the path since the "PUT" operation need the file's path. The following two examples we just use the **pg_01** as the experimental file.

2) Use the DFS client to upload your file to the DFS. Using the following command:

java dfs.DFSClient

3) Use put command to upload the pg_01 to the dfs.

dfs put pg_01

After uploading the file to the DFS, you can use the following commands to check the the nodes' available slots and uploaded status.

to check the nodes: **dfs nodes**

to check the file status: **dfs files**

4) Run the WordCount Class using the following command:

java examples.WordCount

[1] http://en.wikipedia.org/wiki/Word_count

5) You can see the progress in the console. When the Mapper and Reducer finished the work, it will upload the generated files onto the DFS. It will promotes that the job has executed successfully. It will tell you the rule of the generated file's name and you can use "dfs files" to check whether the file is on the DFS or not.

The file's name's rule is : **"job - [jobId]-[outputfilename]-[partitionNumber]_[chunkNum]"**

6) If you want to get the result integrated, you can use the "GET" command to download the files which scattered all over the DFS. For example, you see "job-0-output-0_0" file, you can use the following command to download the whole file into your local storage.

dfs get job-0-output-0_0 /tmp/download/

N-Gram

Describe

In the fields of computational linguistics and probability, an n-gram is a contiguous sequence of n items from a given sequence of text or speech. The items can be phonemes, syllables, letters, words or base pairs according to the application. The n-grams typically are collected from a text or speech corpus.

[2]An n-gram of size 1 is referred to as a "unigram"; size 2 is a "bigram" (or, less commonly, a "digram"); size 3 is a "trigram". Larger sizes are sometimes referred to by the value of n, e.g., "four-gram", "five-gram", and so on.

Our example is a 5-Gram.

How to run the N-Gram

1) Prepare your input file

We use the "pg_01" as the default input file. If you did the WordCount example and you want to use the "pg_01" as the input file, you can omit the step 2 and step 3. Or you need to execute the following commands.

2) Use the DFS client to upload your file to the DFS. Using the following command:

java dfs.DFSClient

[2] http://en.wikipedia.org/wiki/N-gram

3) Use put command to upload the pg_01 to the dfs.

dfs put pg_01

4) Run the NGram Class using the following command:

java examples.NGram

5) You can see the progress in the console. When the Mapper and Reducer finished the work, it will upload the generated files onto the DFS. It will promotes that the job has executed successfully. It will tell you the rule of the generated file's name and you can use "dfs files" to check whether the file is on the DFS or not.

The file's name's rule is : **"job - [jobId]-[outputfilename]-[partitionNumber]_[chunkNum]"**

6) If you want to get the result integrated, you can use the "GET" command to download the files which scattered all over the DFS. For example, you see "job-1-output-1_0" file, you can use the following command to download the whole file into your local storage.

dfs get job-1-output-1_0 /tmp/download/

7) If you run the examples and you want to terminated the whole system, you can just exit your terminal or you can use "Ctrl + C" to exit.

Execute Multiple Jobs

You can run the WordCount and N-Gram simultaneous and test the Concurrency of the system.

REFERENCE

1. https://github.com/bruingao/MapReduce
2. http://hadoop.apache.org/
3. http://horicky.blogspot.com/2008/11/hadoop-mapreduce-implementation.html