Mumfie DFS

Analysis and Design, version 0.3

Rasmus Holm, carho647@student.liu.se
(Samuel Trennelius, samtr396@student.liu.se)

Last modified: 2012-04-25

Summary

This document describes MDFS from a programmers perspective. It contains information about classes and how they interact with each other, but also describes use case scenarios. Furthermore it narrates the testing process to describe how test of the system functions will be done. This document should be seen as a work in progress and not as a definitive document over the analysis and design.

Table of content

        - mdfs.utils.parser.FileNameOperations
        - mdfs.utils.parser.Parser
        - mdfs.utils.parser.Session
- Use case diagrams
    - Use Case Description
        - Writing a file to MDFS.
- Interaction diagram
- See attachment 4 for diagram
- State and activity diagams
- Testplanening
    - Must-Have
        - Client API
        - Name Node
        - Data Node
    - Nice-to-have

Document conventions

Mumfie = Name of the project
DFS = Distributed File System.
HDFS = Hadoop Distributed File system.
MDFS = Mumfie Distributed File system.
Node = A computer in a network of other computers.
TTL = Time To Live.
Data Node = A computer/node that stores raw data in MDFS .
Name Node = A computer/node that stores and handles Meta-data for the MDFS.
OS = Operating System.
API = Application Programing Interface.
CLI = Command Line Interface.
must-have = features or functions that a software must-have.
nice-to-have = features or functions that would be nice to have in case of time software.
SSL = Secure Socket Layer, a way for encrypted data transfer over the TCP/IP protocol.
FUSE = A small unix opensorce software that enables third party user-level software to mount on the kernel level.

Class Diagrams

Client API

See attachment 1

Name Node

See attachment 2

Data Node

See attachment 3

Class descriptions

Client API

mdfs.client.api.FileQuery:

This Interface enabels the user to access a MDFS file system in a manner that are similar to generic terminal commands.

mdfs.client.api.FileQueryImpl

This is a basic implementation of FileQuery where the user gains access to a MDFS

mdfs.client.parser.JSONHeaderCreator

A small help class that creates JSON headers to be used when communicates with different parts of MDFS

Name Node

NameNode_Bootstrap

This bootstraps and starts the Name Node an no interaction from a user i available after it is started.

mdfs.namenode.io.ConnectionListener

Creates a Server socket that listen to a port. When a connectionen is accepted a new thread of the type ConnectionSheppard is created that handles it.

mdfs.namenode.io.ConnectionSheppard

ConnectionShepperd handels a connection from start to end in regard to the MDFS comunication protocol. Nothing is done until it is started as a thread

mdfs.namenode.io.DataNodeQuerier

This object queries different data nodes. It provides a way for the name node to tell the data nodes what to do.

mdfs.namenode.parser.ParserFactory

Creates a parsers implementing the interface Parser depending on what is needed to pars a Session

mdfs.namenode.parser.ParserInfoFile

Parses and execute updates regarding Info for Files in the different data repositories that exists on the name node

mdfs.namenode.parser.ParserRequestMetaData

A Parser implementing Parser that handles all Modes when Stage = Request and Type = Meta-Data

mdfs.namenode.parser.SessionImpl

Session is the object that contains request and response for MDFS communication protocol.
It contains all necessary information to handle a request and does so by parsing and executing

mdfs.namenode.repositories.DataNodeInfoRepository

Stores Information about available DataNodes. As this is a shared resource it is a Singelton and synchronized by ReentrantLock. The lock works in the manner that if the thread holding the lock would crach, the lock will still be released

mdfs.namenode.repositories.DataNodeInfoRepositoryNode

A node that hold data regarding a DataNode

mdfs.namenode.repositories.DataTypeEnum

A enum that representents the type of data, a file or a dir

mdfs.namenode.repositories.MetaDataRepository

A implementation of MetaDataRepository that is synchronized via ReentrantLock for allowing multiple threads. When a lock is acquired all other operations on the MetaDataRepository are locked. If the thread that holds the lock fails/crashes, The lock will be unlocked.

mdfs.namenode.repositories.MetaDataRepositoryNode

The node that contains all the meta-data and information neaded for a file or a dir in MDFS

mdfs.namenode.repositories.UserDataRepository

UserDataRepository stores MDFS user data that correlates to the files, authentication and so on. It is a Singelton since it is a shared resource and are synchronized via link ReentrantLock for allowing multiple threads. When a lock is acquired all other operations on the UserDataRepository are locked. If the thread that holds the lock fails/crashes, The lock will be unlocked.

mdfs.namenode.repositories.UserDataRepositoryNode

A node that holds user data

mdfs.namenode.sql.MySQLConnector

This class creates a connection to the MySQL database that is specified in the mdfs/config/config.cfg

mdfs.namenode.sql.MySQLDeleteInsertUpdate

This class simply execute generic update insert and delete queries that does not return anything

mdfs.namenode.sql.MySQLFetch

This class are used to fetch specific element in regard to MDFS from a MySQL Database and the mdfs/config/config.cfg file

mdfs.namenode.sql.MySQLUpdater

This Class is a threaded Singelton that helps other classes to execute query that need no response. It works in a way that a thread is intermittently executing queries to the MySQL database specified mdfs/config/config.cfg The MySQL.update.rate in the config.cfg is the milliseconds that the thread sleeps before checking if any queries are queued to be issued. This enables other classes to just enqueue a query and not having to wait for a response from the SQL server.MySQLUpdater works in a way that other classes can queue items even if MySQLUpdater is currently executing queries from said queue. This helps MDFS performance since no other pars of the software have to wait on a SQL server.

Data Node

DataNode_Bootstrap

Simply starts the DataNode. No interaction with a user is possible

mdfs.datanode.io.ConnectionListener

Creates a Server socket that listen to a port. When a connection is accepted a new thread of the type ConnectionSheppard is created that handles it.

mdfs.datanode.io.ConnectionSheppard

ConnectionShepperd handles a connection from start to end in regard to the MDFS communication protocol. Nothing is done until it is started as a thread

mdfs.datanode.io.NameNodeInformer

This class informs the NameNode of changes to the local file system in regard to the MDFS file system. Nothing is done until it is started as a thread on which the Name Node is updated.

mdfs.datanode.io.Replicator

Handles replication of raw data between Data Nodes.

mdfs.datanode.parser.ParserFactory

Creates a parser in regard to Stage, Type and Mode. I i used i together with a session

mdfs.datanode.parser.ParserRequestFile

A parser that handles all communication with the Stage=Request and Type=File and parses it. This class wraps Session and parses its content.

mdfs.datanode.parser.SessionImpl

An implementation of the interface Session that are related to the operations of the Datanode

Utils

mdfs.utils.ArrayUtils

Useful tools when dealing with Arrays.

mdfs.utils.Config

A class that statically enables access and query the config file mdfs/config/config.cfg for data

mdfs.utils.FSTree<E>

A implementation of a file structure that enable one to store generic nodes with a String key The String key should be the path the node is stored at. ex. c:\example\file.txt or /tmp/example/file.txt

mdfs.utils.Hashing

A simple implementation allowing hashing if String to MD5 or SHA-1

mdfs.utils.SplayTree<Key, E>

Top-Down Splay Tree implementation (http://en.wikipedia.org/wiki/Splay_tree). Some modification is added by Rasmus Holm, Implementing a separated Comparable key to index elements

mdfs.utils.Time

Simple way of getting timestamps in the form of yyyy-MM-dd HH:mm:ss

mdfs.utils.Verbose

Simple class that in a Structured way of printing text

mdfs.utils.io.SocketFactory

A factory that creates Socket for use in MDFS

mdfs.utils.io.SocketFunctions

Handels writing and reading between sockets. SocketFunctions is mapped to a low level protocol for transferring text and binary files

mdfs.utils.parser.FileNameOperations

A small class that helps with File Name operations

mdfs.utils.parser.Parser

A interface that defines what a parser should be able to do. A parser parses the Session and creates the response. A Session is there for more of a container for the request and response.

mdfs.utils.parser.Session

Session is the object that contains request and respose for MDFS comunication protcol. It contains all nessesary information to handel a request and dose so by parsing and executing

Use case diagrams

Since there is minimal user interaction in MDFS a regular Use case diagram is not very informative. Instead a detailed Use case description i provided walking through the different steps in writing and reading files.

Use Case Description

Writing a file to MDFS.

1. A external software starts by providing mdfs.client.api.FileQuery with a username and password for MDFS
2. A request of writing a file is made to FileQuery's method put providing the File to be written and to what logical path in MDFS
3. FileQuery take over and builds a MDFS Transport Protocol header according to the request made via mdfs.client.parser.JSONHeaderCreator
4. FileQuery the opens a socket to the NameNode using mdfs.utils.io.SocketFactory
5. FileQuery then send the request to the NameNode via mdfs.utils.io.SocketFunctions
6. NameNode mdfs.namenode.io.ConnectionListener hers the opening of a socket and start a new Thread of mdfs.namenode.io.ConnectionSheppard and then continues to listen for new sockets.
7. The ConnectionSheppard retrieves the header from the socket and then creates a mdfs.namenode.parser.SessionImpl and asks it to parse the request
8. The Session in turn creates a Parser via mdfs.namenode.parser.ParserFactory and wraps itself in it.
9. The parser takes over and does user authentication via the mdfs.namenode.repositories.UserDataRepository.
10. After this the parser precedes by locating the correct location for the new file in the mdfs.namenode.repositories.MetaDataRepository and adds it, if possible.
    a. When an update is done to a Repository, the mdfs.namenode.sql.MySQLUpdater is informed and the change is queued to be done to the MySQL permanent storage database as well.
11. The parser then creates a response to the Client with the necessary information, ex. the data node on which the data will be stored.
12. The response is then "thrown" back to the ConnectionShepperd which returns it to the client. The ConnectionSheppard then terminates.
13. The FileQuery then receives the response and, if no errors, builds a request via JSONHeaderCreator to write the file to one of the available DataNodes.
14. FileQuery the opens a connection to one DataNode and send the header followd by the raw data.
15. The mdfs.datanode.io.ConnectionListner hears the new socket and kicks it in to a new mdfs.datanode.io.ConnectionSheppard.
16. The ConnectionsSheppard recives the header and strarts a mdfs.datanode.parser.SessionImpl and asks it to parse it.

17. The Session in turn then creates a parser via mdfs.datanode.parser.ParserFactory and wraps itself in it.
18. The parser the reads the request, and via mdfs.utils.io.SocketFunktions saves the raw data to disk
    a. The parser starts mdfs.datanode.io.NameNodeInformer which informs the NameNode that the data now is stored on said DataNode.
    b. The parser starts mdfs.datanode.io.Replicator which replicates the raw data to other DataNodes.
19. The parser creates a response for the client and the ConnectionShepperd then send it back to the client before terminating itself.
20. The FileQuery receives the response from the DataNode and returns true if nothing on the went when wrong

Interaction diagram

The sequence diagram is showing the a simplified model of the sequential states that the MDFS goes through when writing a file. For a more detailed description of the process it is better to combine the Use Case Description with the Class Diagram.

See attachment 4 for diagram

State and activity diagams

MDFS is a file system and very little user interaction takes place in it. After a command is written in the CLI the operation is executed without any graphics or choices for the user. Considering that, a activity diagram wouldn't say anything about this software and therefore this is excluded in this document. Below is a state diagram which probably isn't that informative either.

Testplanening

Must-Have

Client API

1. The Client API shall write and retrieve general meta-data from the Name Node
    a. Testing Client API toward dummy Name Node
        i. Testing interpretation of meta-data
        ii. Testing requests for meta-data
    b. Testing Client API toward developed Name Node
2. The Client API write and read raw data and appends necessary meta-data to and from the Data Nodes
    a. Testing Client API toward dummy Data Node
    b. Testing Client API toward developed Data Node

Name Node

1. The Name Node shall store all meta-data regarding files in MDFS in RAM
    a. Testing storage function with generating dummy meta-data to store in Name Node
    b. Testing storage function by retrieving and storing real meta-data from Client API
2. The Name Node shall replicates all meta-data from RAM in to a SQL database for safe storage continuously.
    a. Testing SQL storage with dummy data
    b. Testing SQL storage from stored dummy meta-data from Name Node RAM
    c. Testing to load meta-data from SQL database to Name Node RAM
3. The Name Node shall provides client authentication.
    a. Testing authentication with dummy data
4. The Name Node shall provides the client with necessary meta-data for reading and writing data to Data Nodes.
    a. Testing to send meta-data to Client API
5. The Name Node shall track and communicate with all the Data Nodes.
    a. Testing communication with dummy interface
    b. Testing communication with developed Data Nodes

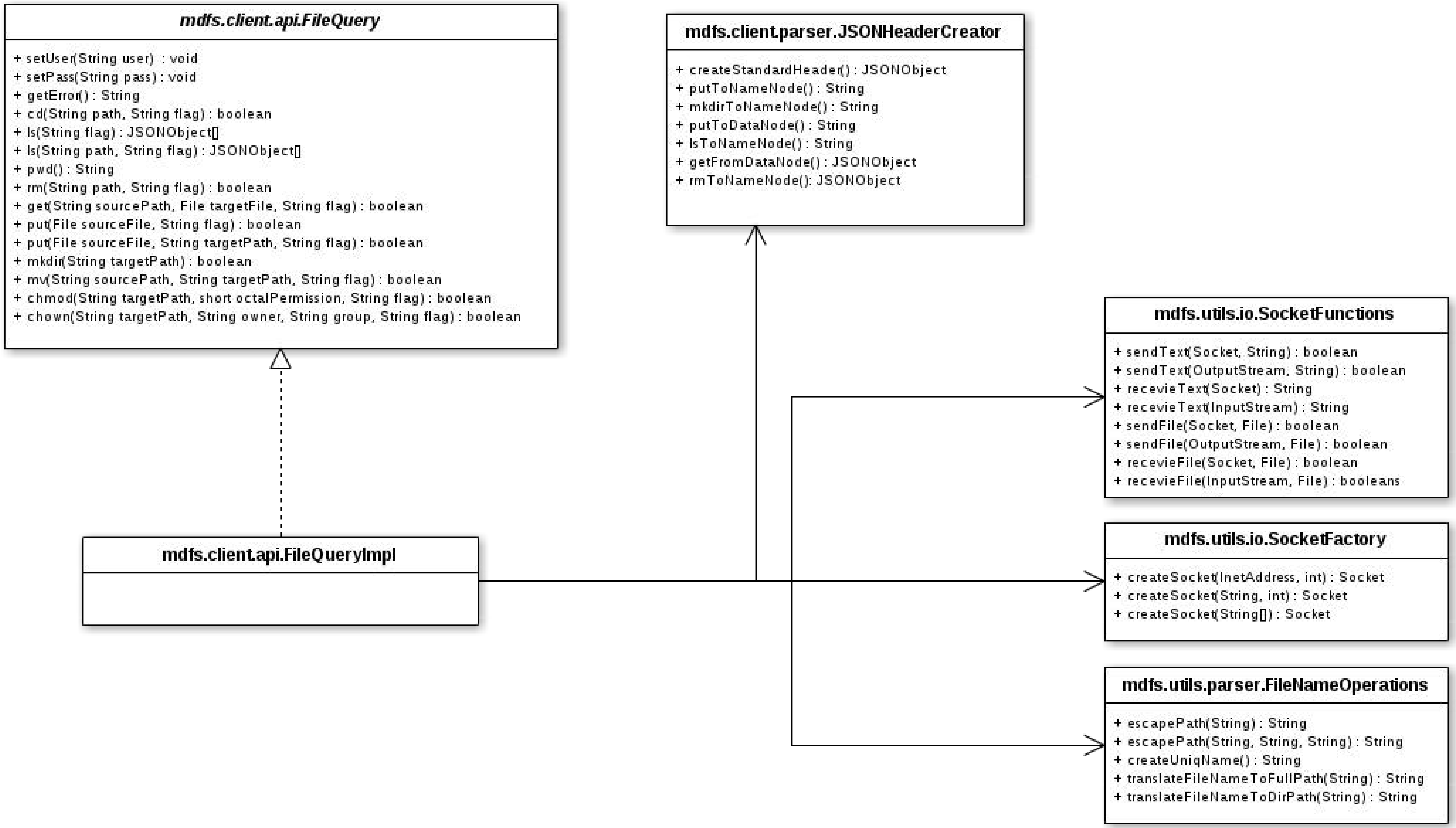
Data Node

1. The Data Node shall retrieve files from the Client and store it on the local file system.
    a. Testing to send dummy data to store on the Data Node
    b. Testing to send data from the Client API to the Data Node
2. The Data Node shall replicates new files to a pre-determined number of Data Nodes
    a. Testing replication function between Data Nodes
3. The Data Node shall updates the Name Node with meta-data updates regarding storage of files.
    a. Testing to update Name Node with meta-data
    b. Testing a automatic updating function with meta-data to the Name Node

Nice-to-have

1. Encryption for files transferred should be supported.
    a. Testing encryption and decryption in Client API and comparing hash codes
2. SSL communication should be supported
    a. Testing SSL NetIO communication between working nodes
3. Automated Building Latency graph for determining distance between different Nodes should be implemented.
    a. Testing to ping from Client API to Data Nodes
    b. Testing Graph/Table to see if it is correct
4. Enable support for multiple Name Nodes ensuring redundancy should be implemented
    a. Testing multiple Name Nodes and a system that can combine them
5. Secure way of authentication of Data Nodes being trusted(not allowing every one to connect as a Data Node) should be implemented.
    a. Testing authentication with dummy Name Node
    b. Testing implemented authentication with developed Name Node
6. Enable support for re-replication of file from a Data Node that is down should be implemented.
    a. Testing to kill data-node and see if re-replication function worksSample clients using the Client API. (CLI, GUI, Dropbox-Like-Client, FUSE-implementation) should be implemented.

Attachment 1



```
mdfs.client.api.FileQuery
+ setUser(String user)  : void
+ setPass(String pass) : void
+ getError() : String
+ cd(String path, String flag) : boolean
+ ls(String flag) : JSONObject[]
+ ls(String path, String flag) : JSONObject[]
+ pwd() : String
+ rm(String path, String flag) : boolean
+ get(String sourcePath, File targetFile, String flag) : boolean
+ put(File sourceFile, String flag) : boolean
+ put(File sourceFile, String targetPath, String flag) : boolean
+ mkdir(String targetPath) : boolean
+ mv(String sourcePath, String targetPath, String flag) : boolean
+ chmod(String targetPath, short octalPermission, String flag) : boolean
+ chown(String targetPath, String owner, String group, String flag) : boolean
```

```
mdfs.client.parser.JSONHeaderCreator
+ createStandardHeader() : JSONObject
+ putToNameNode() : String
+ mkdirToNameNode() : String
+ putToDataNode() : String
+ lsToNameNode() : String
+ getFromDataNode() : JSONObject
+ rmToNameNode(): JSONObject
```

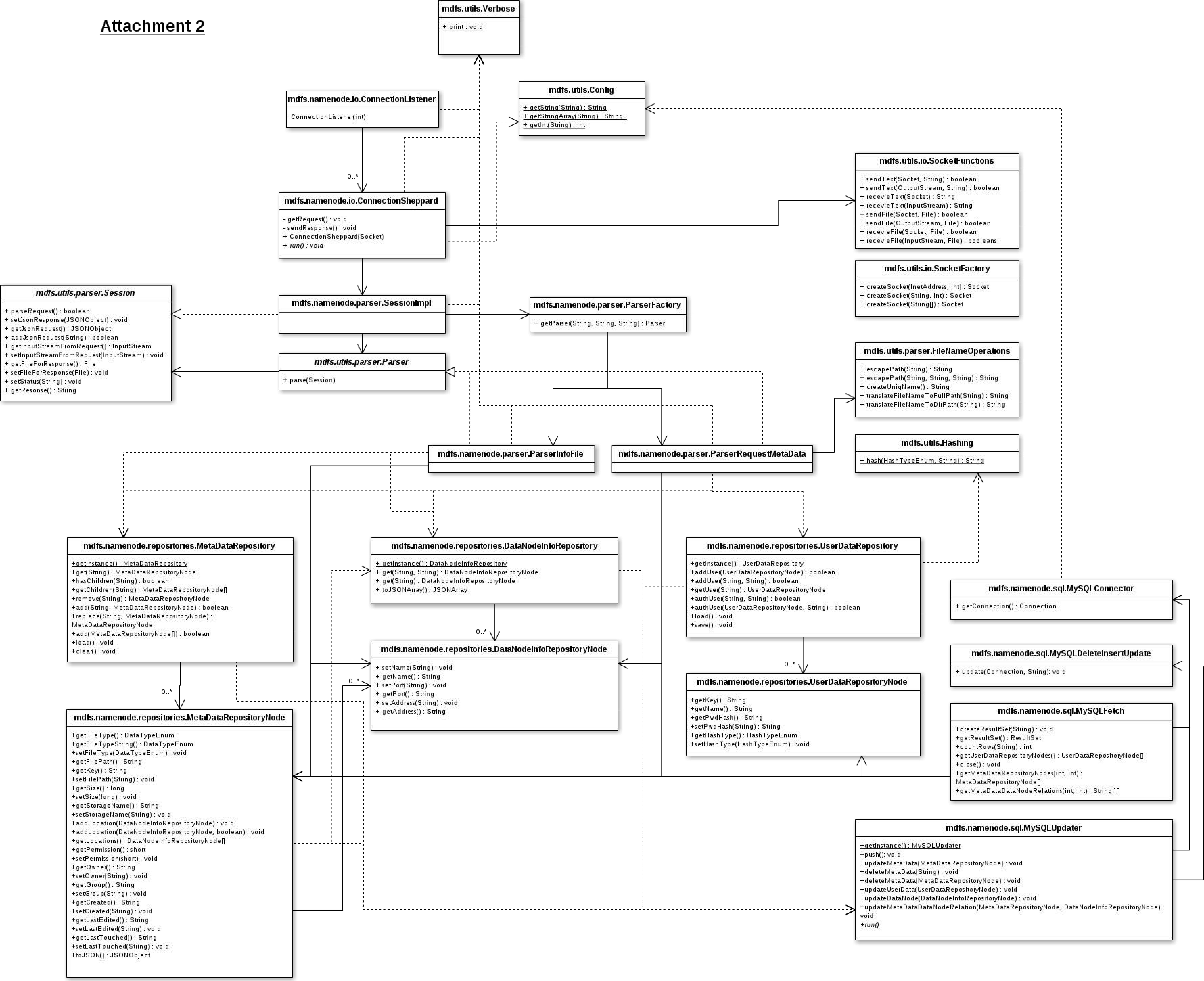
```
mdfs.utils.io.SocketFunctions
+ sendText(Socket, String) : boolean
+ sendText(OutputStream, String) : boolean
+ recevieText(Socket) : String
+ recevieText(InputStream) : String
+ sendFile(Socket, File) : boolean
+ sendFile(OutputStream, File) : boolean
+ recevieFile(Socket, File) : boolean
+ recevieFile(InputStream, File) : booleans
```

```
mdfs.utils.io.SocketFactory
+ createSocket(InetAddress, int) : Socket
+ createSocket(String, int) : Socket
+ createSocket(String[]) : Socket
```

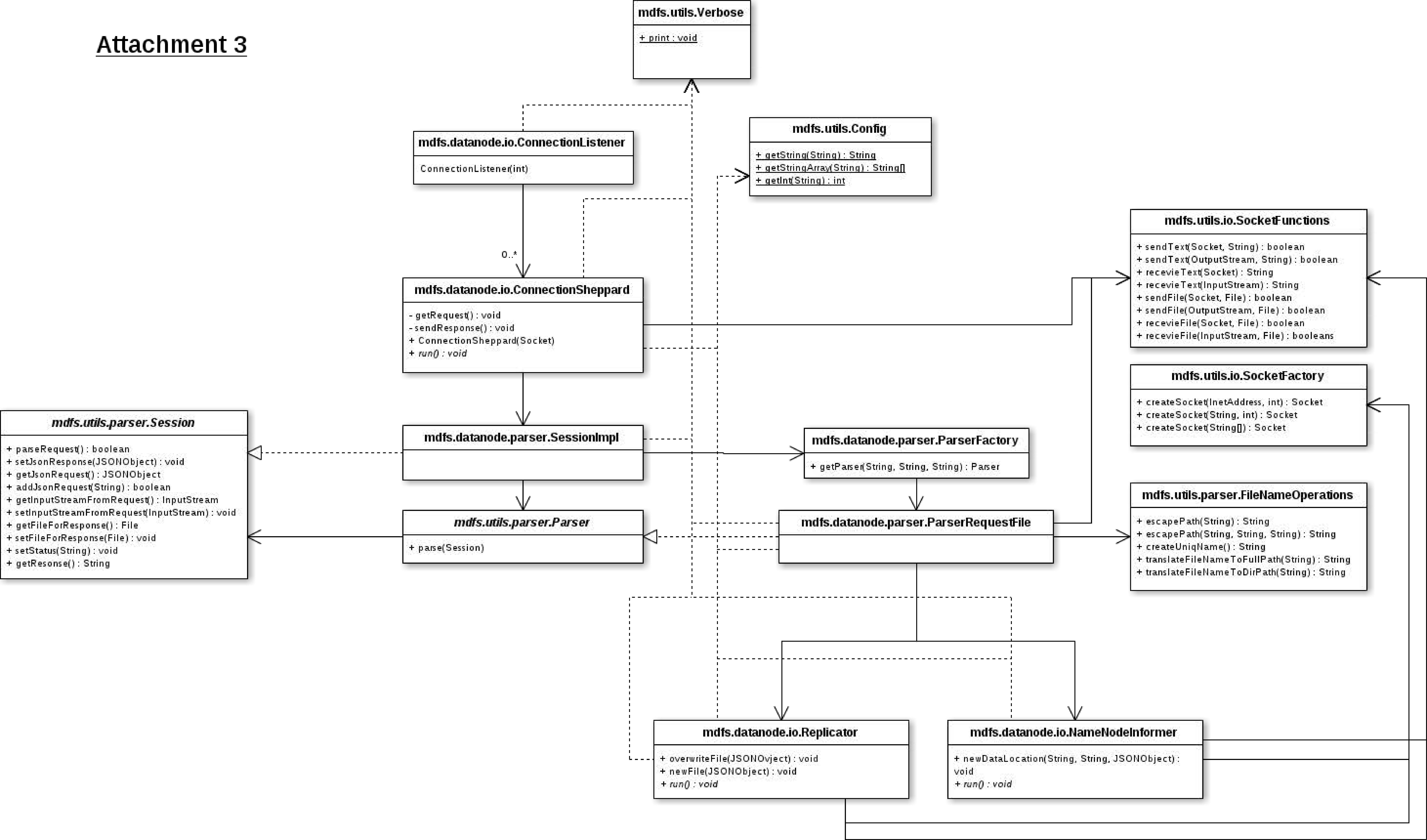
```
mdfs.utils.parser.FileNameOperations
+ escapePath(String) : String
+ escapePath(String, String, String) : String
+ createUniqName() : String
+ translateFileNameToFullPath(String) : String
+ translateFileNameToDirPath(String) : String
```

```
mdfs.client.api.FileQueryImpl
```

Attachment 2

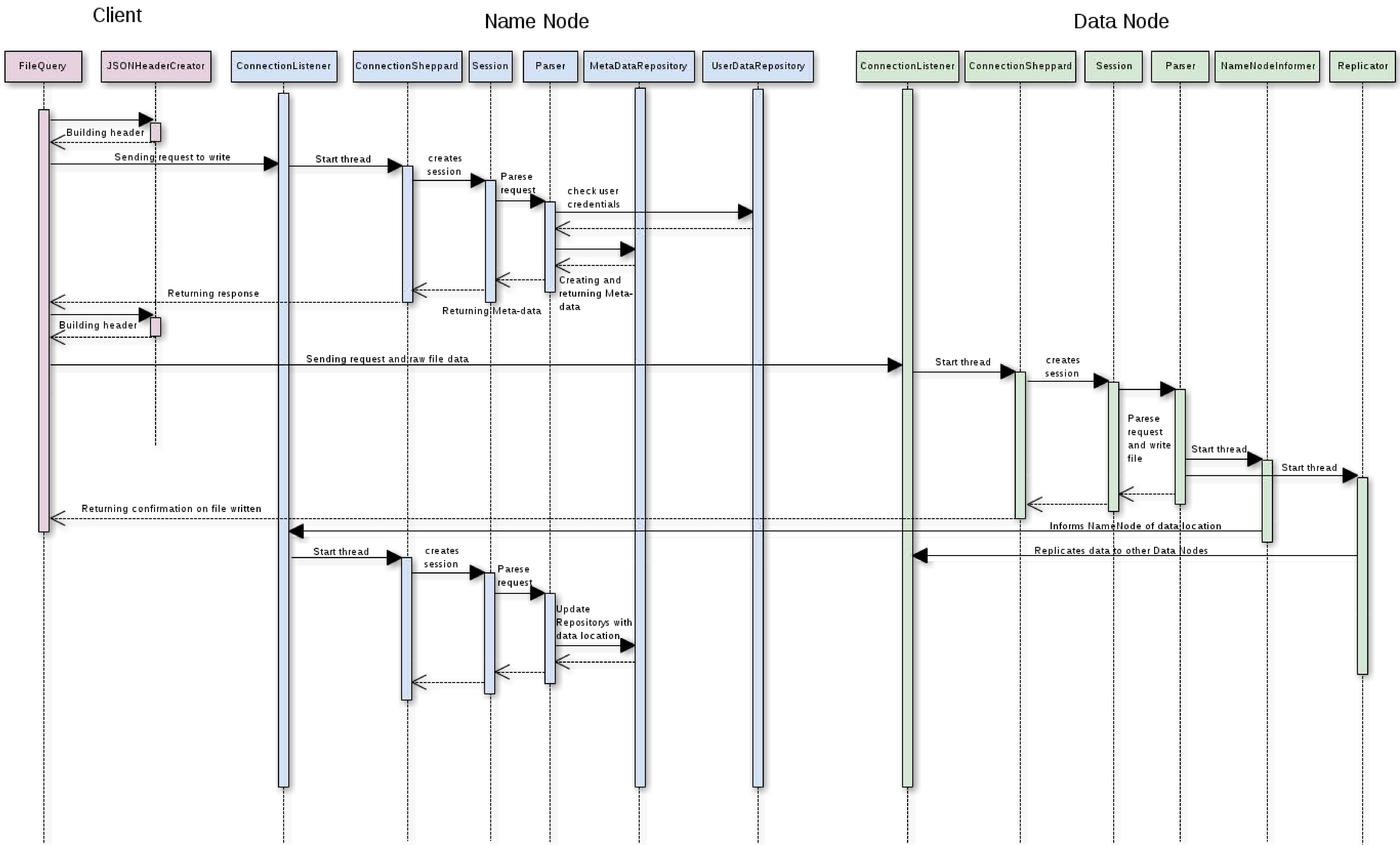

mdfs.utils.Verbose
+ print : void
mdfs.utils.Config
+ getString(String) : String
+ getStringArray(String) : String[]
+ getInt(String) : int
mdfs.namenode.io.ConnectionListener
ConnectionListener(int)
0..*
mdfs.namenode.io.ConnectionSheppard
- getRequest() : void
- sendResponse() : void
+ ConnectionSheppard(Socket)
+ run() : void
mdfs.utils.io.SocketFunctions
+ sendText(Socket, String) : boolean
+ sendText(OutputStream, String) : boolean
+ recevieText(Socket) : String
+ recevieText(InputStream) : String
+ sendFile(Socket, File) : boolean
+ sendFile(OutputStream, File) : boolean
+ recevieFile(Socket, File) : boolean
+ recevieFile(InputStream, File) : booleans
mdfs.utils.io.SocketFactory
+ createSocket(InetAddress, int) : Socket
+ createSocket(String, int) : Socket
+ createSocket(String[]) : Socket
mdfs.utils.parser.Session
+ parseRequest() : boolean
+ setJsonResponse(JSONObject) : void
+ getJsonRequest() : JSONObject
+ addJsonRequest(String) : boolean
+ getInputStreamFromRequest() : InputStream
+ setInputStreamFromRequest(InputStream) : void
+ getFileForResponse() : File
+ setFileForResponse(File) : void
+ setStatus(String) : void
+ getResonse() : String
mdfs.namenode.parser.SessionImpl
mdfs.namenode.parser.ParserFactory
+ getParser(String, String, String) : Parser
mdfs.utils.parser.Parser
+ parse(Session)
mdfs.utils.parser.FileNameOperations
+ escapePath(String) : String
+ escapePath(String, String, String) : String
+ createUniqName() : String
+ translateFileNameToFullPath(String) : String
+ translateFileNameToDirPath(String) : String
mdfs.utils.Hashing
+ hash(HashTypeEnum, String) : String
mdfs.namenode.parser.ParserInfoFile
mdfs.namenode.parser.ParserRequestMetaData
mdfs.namenode.repositories.MetaDataRepository
+getInstance() : MetaDataRepository
+get(String) : MetaDataRepositoryNode
+hasChildren(String) : boolean
+getChildren(String) : MetaDataRepositoryNode[]
+remove(String) : MetaDataRepositoryNode
+add(String, MetaDataRepositoryNode) : boolean
+replace(String, MetaDataRepositoryNode) : MetaDataRepositoryNode
+add(MetaDataRepositoryNode[]) : boolean
+load() : void
+clear() : void
0..*
mdfs.namenode.repositories.DataNodeInfoRepository
+ getInstance() : DataNodeInfoRepository
+ get(String, String) : DataNodeInfoRepositoryNode
+ get(String) : DataNodeInfoRepositoryNode
+ toJSONArray() : JSONArray
0..*
mdfs.namenode.repositories.DataNodeInfoRepositoryNode
+ setName(String) : void
+ getName() : String
+ setPort(String) : void
+ getPort() : String
+ setAddress(String) : void
+ getAddress() : String
0..*
mdfs.namenode.repositories.UserDataRepository
+getInstance() : UserDataRepository
+addUser(UserDataRepositoryNode) : boolean
+addUser(String, String) : boolean
+getUser(String) : UserDataRepositoryNode
+authUser(String, String) : boolean
+authUser(UserDataRepositoryNode, String) : boolean
+load() : void
+save() : void
0..*
mdfs.namenode.repositories.UserDataRepositoryNode
+getKey() : String
+getName() : String
+getPwdHash() : String
+setPwdHash(String) : String
+getHashType() : HashTypeEnum
+setHashType(HashTypeEnum) : void
mdfs.namenode.sql.MySQLConnector
+ getConnection() : Connection
mdfs.namenode.sql.MySQLDeleteInsertUpdate
+ update(Connection, String): void
mdfs.namenode.sql.MySQLFetch
+createResultSet(String) : void
+getResultSet() : ResultSet
+countRows(String) : int
+getUserDataRepositoryNodes() : UserDataRepositoryNode[]
+close() : void
+getMetaDataReopsitoryNodes(int, int) : MetaDataRepositoryNode[]
+getMetaDataDataNodeRelations(int, int) : String][]
mdfs.namenode.repositories.MetaDataRepositoryNode
+getFileType() : DataTypeEnum
+getFileTypeString() : DataTypeEnum
+setFileType(DataTypeEnum) : void
+getFilePath() : String
+getKey() : String
+setFilePath(String) : void
+getSize() : long
+setSize(long) : void
+getStorageName() : String
+setStorageName(String) : void
+addLocation(DataNodeInfoRepositoryNode) : void
+addLocation(DataNodeInfoRepositoryNode, boolean) : void
+getLocations() : DataNodeInfoRepositoryNode[]
+getPermission() : short
+setPermission(short) : void
+getOwner() : String
+setOwner(String) : void
+getGroup() : String
+setGroup(String) : void
+getCreated() : String
+setCreated(String) : void
+getLastEdited() : String
+setLastEdited(String) : void
+getLastTouched() : String
+setLastTouched(String) : void
+toJSON() : JSONObject
mdfs.namenode.sql.MySQLUpdater
+getInstance() : MySQLUpdater
+push(): void
+updateMetaData(MetaDataRepositoryNode) : void
+deleteMetaData(String) : void
+deleteMetaData(MetaDataRepositoryNode) : void
+updateUserData(UserDataRepositoryNode) : void
+updateDataNode(DataNodeInfoRepositoryNode) : void
+updateMetaDataDataNodeRelation(MetaDataRepositoryNode, DataNodeInfoRepositoryNode) : void
+run()

Attachment 3


mdfs.utils.Verbose
+ print : void
mdfs.datanode.io.ConnectionListener
ConnectionListener(int)
mdfs.utils.Config
+ getString(String) : String
+ getStringArray(String) : String[]
+ getInt(String) : int
mdfs.utils.io.SocketFunctions
+ sendText(Socket, String) : boolean
+ sendText(OutputStream, String) : boolean
+ recevieText(Socket) : String
+ recevieText(InputStream) : String
+ sendFile(Socket, File) : boolean
+ sendFile(OutputStream, File) : boolean
+ recevieFile(Socket, File) : boolean
+ recevieFile(InputStream, File) : booleans
0..*
mdfs.datanode.io.ConnectionSheppard
- getRequest() : void
- sendResponse() : void
+ ConnectionSheppard(Socket)
+ run() : void
mdfs.utils.io.SocketFactory
+ createSocket(InetAddress, int) : Socket
+ createSocket(String, int) : Socket
+ createSocket(String[]) : Socket
mdfs.utils.parser.Session
+ parseRequest() : boolean
+ setJsonResponse(JSONObject) : void
+ getJsonRequest() : JSONObject
+ addJsonRequest(String) : boolean
+ getInputStreamFromRequest() : InputStream
+ setInputStreamFromRequest(InputStream) : void
+ getFileForResponse() : File
+ setFileForResponse(File) : void
+ setStatus(String) : void
+ getResonse() : String
mdfs.datanode.parser.SessionImpl
mdfs.datanode.parser.ParserFactory
+ getParser(String, String, String) : Parser
mdfs.utils.parser.FileNameOperations
+ escapePath(String) : String
+ escapePath(String, String, String) : String
+ createUniqName() : String
+ translateFileNameToFullPath(String) : String
+ translateFileNameToDirPath(String) : String
mdfs.utils.parser.Parser
+ parse(Session)
mdfs.datanode.parser.ParserRequestFile
mdfs.datanode.io.Replicator
+ overwriteFile(JSONOvject) : void
+ newFile(JSONObject) : void
+ run() : void
mdfs.datanode.io.NameNodeInformer
+ newDataLocation(String, String, JSONObject) : void
+ run() : void

Attachment 4

Writing a file to MDFS


Client
Name Node
Data Node
FileQuery
JSONHeaderCreator
ConnectionListener
ConnectionSheppard
Session
Parser
MetaDataRepository
UserDataRepository
ConnectionListener
ConnectionSheppard
Session
Parser
NameNodeInformer
Replicator
Building header
Sending request to write
Start thread
creates session
Parese request
check user credentials
Creating and returning Meta-data
Returning Meta-data
Returning response
Building header
Sending request and raw file data
Start thread
creates session
Parese request and write file
Start thread
Start thread
Returning confirmation on file written
Informs NameNode of data location
Replicates data to other Data Nodes
Start thread
creates session
Parese request
Update Repositorys with data location