SUFS Design Document

Component API Calls

We are using RESTful endpoints to communicate between the components. Only JSON is supported.

Name Node API

Create File Request

PUT /file

Adds a new entry to the map of files, the key being the filename and value being an array of blocks. Creates blocks, each with their own id and DnList (list of the data nodes they are stored in) and stores to the map under the given filename. Number of blocks created is the number needed to fit the given file size with a block size of 64 MB. The DnList of each block will be initial empty until a block report is received with that block being stored.

```
{
    "FileName": string, // the filename SUFS will use
    "Size": string // this number of bytes in the file
}
```

Create File Response

BlockInfos stores as many blocks as there needs to be in order to store the file. BlockId is the unique id given to the specific block. The DataNodeList is built using the data nodes known to the name nodes. Returns the info for the CLI needs to store the blocks.

```
{
    "BlockInfos": [ // in-order list of info for each block
        {
            "BlockId": string, // the internal id of the block
            "DataNodeList": [ // list of Data Nodes the block should be
stored on
                string // IP address and port of the Data Node (ex:
"10.0.0.1:8080")
            ]
        }
    ],
    "Error": string // description of the error, empty means no error
}
```

Get File Request

GET /file

Using the passed in file name, attempts to find the file in the map for files.

```
{
    "FileName": string, // the filename in SUFS
}
```

Get File Response

If the map returns a value, it then stores the blocks and their respective DataNodeLists into BlockInfos. Returns the info for the CLI needs to store the blocks.

```
{
    "BlockInfos": [ // in-order list of info for each block
        {
            "BlockId": string, // the internal ID of the block
            "DataNodeList": [ // list of Data Nodes the block should be
stored on
                string // IP address and port of the Data Node (ex:
"10.0.0.1:8080")
            ]
        }
    ],
    "Error": string // description of the error, empty means no error
}
```

Block Report Request

PUT /blockReport

First checks if MyIp is in the DnList for the known DataNodes, if not then it is added. The name node then goes through each received BlockId and adds the requesting Data Node's information to that block's DnList in the map.

```
{
    "MyIp": string, // the public IP address of the sending Data Node
    "BlockIds": []string // the list of IDs of each block stored on the
sending Data Node
}
```

Block Report Response

Only fills out Error if their was an error in the execution of the request. Happy Path, Error is nil.

```
{
    "Error": string // description of the error, empty means no error
}
```

Heartbeat Request

PUT /heartbeat

Delays the time till the Data Node at MyIp is considered dead.

```
{
    "MyIp": string, // the public IP address of the sending Data Node
}
```

Heartbeat Response

Only fills out Error if their was an error in the execution of the request. Happy Path, Error is nil.

```
{
    "Error": string // description of the error, empty means no error
}
```

Data Node API

Store Block Request

PUT /block

Checks its own IP is contained in the DataNodeList. If it is, then it stores the block into the DataNode's directory and removes itself from the DataNodeList. It then forwards the JSON payload to the first DataNode contained in the DataNodeList. This process repeats until the DataNodeList is empty, signifying that forwarding is complete and the blocks have been all stored and forwarded

```
{
    "Block": string, // base64 encoded block data
    "DataNodeList": [ // list of Data Nodes the block should be stored on
                string // IP address and port of the Data Node (ex:
"10.0.0.1:8080")
    ],
    "BlockId": string // the internal ID of the block
}
```

Store Block Response

```
{
    "Error": string // description of the error, empty means no error
}
```

Get Block Request

GET /block

Checks if the BlockId is contained within the DataNode's block directory. If it exists, it base64 encodes the block data into a JSON payload and returns it with an empty 'Error' string. If an error occurs, then the it returns a JSON payload of just an error without any Block data.

```
{
    "BlockId": string // the internal ID of the block
}
```

Get Block Response

```
{
    "Block": string, // base64 encoded block data
    "Error": string // description of the error, empty means no error
}
```

Replicate Block Request

POST /replicate

Checks if it has the given BlockId, and if it doesn't then returns an error that it does not contain the BlockId. If it does, then it essentially calls a store block request on that BlockId onto the given DataNodeList, letting it forward itself to the DataNodes in the DnList.

```
{
    "BlockId": string, // the internal ID of the block
    "DataNodeList": [ // list of Data Nodes the block should be stored on
                string // IP address and port of the Data Node (ex:
"10.0.0.1:8080")
    ]
}
```

Replicate Block Response

```
{
    "Error": string // description of the error, empty means no error
}
```

CLI

Each command can include the `-v` option. This turns verbose mode on. When verbose mode is on the CLI will output log statements as it performs the action.

Create File Command

Using the AWS SDK, the CLI downloads the file from S3 to a temporary file. It calls the Name Node to create a new file in the Name Node. It receives Data Node list from the Name Node's response. It pulls data from the temporary file in block size chunks individually, base64 encodes it, and sends it to the first Data Node in the list. If any upload fails then subsequant Data Nodes in the list for that block are used. If all Data Node uploads fail for a given block then the CLI exits with an error.

```
/path/to/CLI create-file <name_node_address_and_port> <file_name> <s3_url>
```

- name_node_address_and_port
  - address is required
  - :port is optional
  - ex: "10.0.0.8", "10.0.0.8:8080"
- file_name
  - the name of the file in SUFS
- s3_url
  - the URL of the file to put into SUFS

Get File Command

Get the list of blocks and their respective Data Nodes from the Name Node. Per block, the CLI gets from the first Data Node in the list and appends to the file. If any download fails then subsequant Data Nodes in the list for that block are used. If all Data Node downloads fail for a given block then the CLI exits with an error.

```
/path/to/CLI get-file <name_node_address_and_port> <file_name>
<save_location>
```

- name_node_address_and_port
  - address is required
  - :port is optional
  - ex: "10.0.0.8", "10.0.0.8:8080"
- file_name
  - the name of the file in SUFS

- save_location
    - the location on the local host to save the file

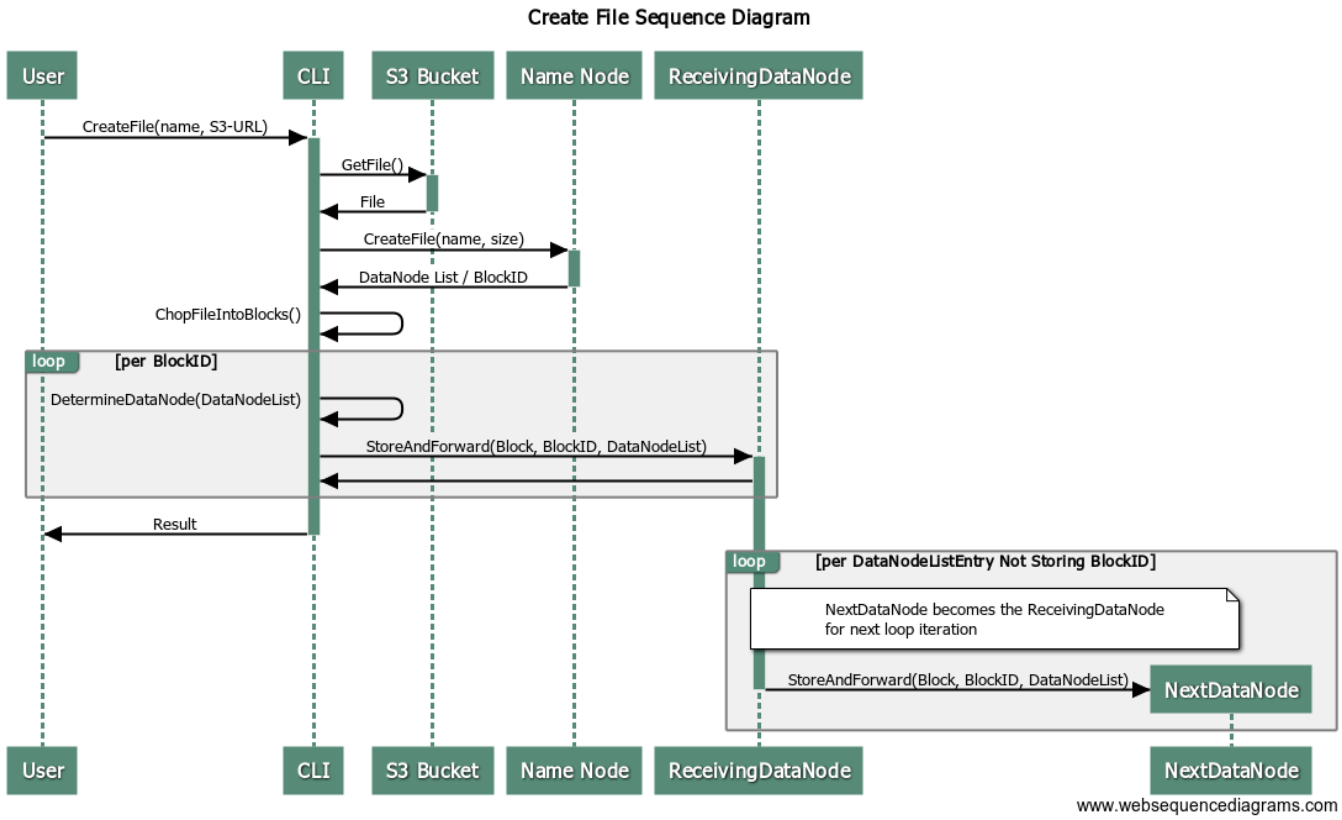
List Data Nodes File Command

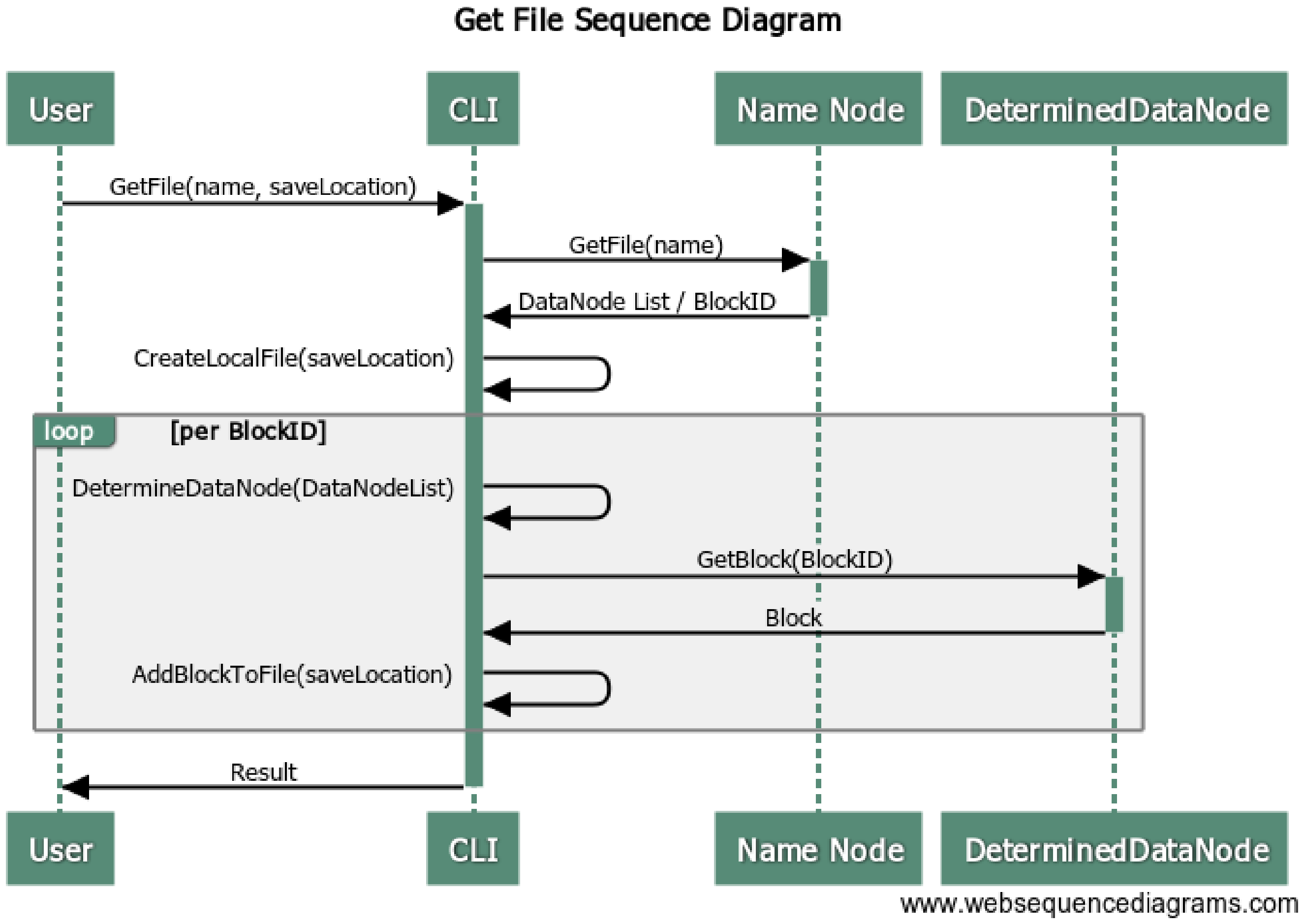
Get the list of blocks and their respective Data Nodes from the Name Node. It beautifies the Data Node list and outputs it.

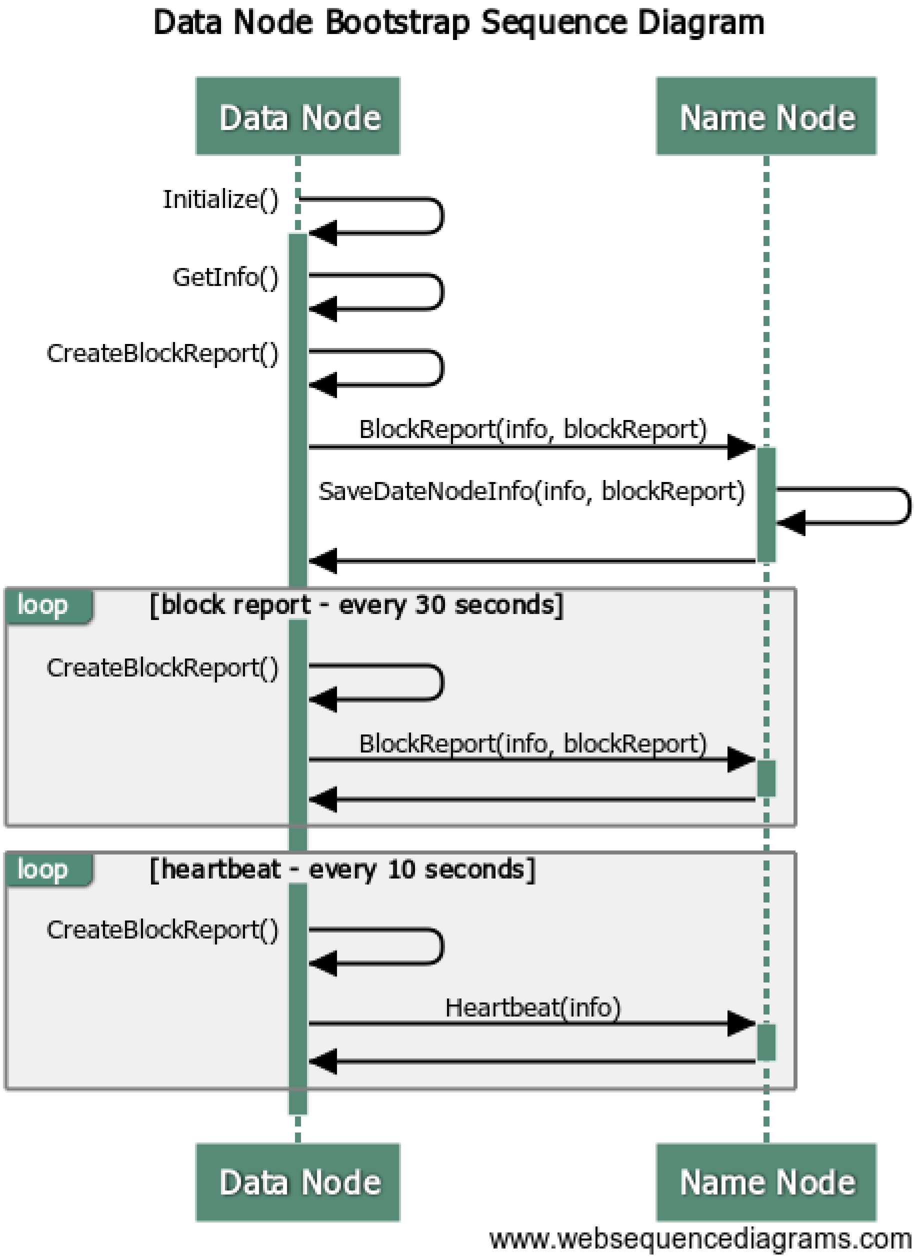
```
/path/to/CLI list-data-nodes <name_node_address_and_port> <file_name>
```

- name_node_address_and_port
    - address is required
    - :port is optional
    - ex: "10.0.0.8", "10.0.0.8:8080"
- file_name
    - the name of the file in SUFS

System Design


Create File Sequence Diagram
User
CLI
S3 Bucket
Name Node
ReceivingDataNode
CreateFile(name, S3-URL)
GetFile()
File
CreateFile(name, size)
DataNode List / BlockID
ChopFileIntoBlocks()
loop
[per BlockID]
DetermineDataNode(DataNodeList)
StoreAndForward(Block, BlockID, DataNodeList)
Result
loop
[per DataNodeListEntry Not Storing BlockID]
NextDataNode becomes the ReceivingDataNode
for next loop iteration
StoreAndForward(Block, BlockID, DataNodeList)
NextDataNode
User
CLI
S3 Bucket
Name Node
ReceivingDataNode
NextDataNode
www.websequencediagrams.com


Get File Sequence Diagram
User
CLI
Name Node
DeterminedDataNode
GetFile(name, saveLocation)
GetFile(name)
DataNode List / BlockID
CreateLocalFile(saveLocation)
loop
[per BlockID]
DetermineDataNode(DataNodeList)
GetBlock(BlockID)
Block
AddBlockToFile(saveLocation)
Result
User
CLI
Name Node
DeterminedDataNode
www.websequencediagrams.com


Data Node Bootstrap Sequence Diagram
Data Node
Name Node
Initialize()
GetInfo()
CreateBlockReport()
BlockReport(info, blockReport)
SaveDateNodeInfo(info, blockReport)
loop
[block report - every 30 seconds]
CreateBlockReport()
BlockReport(info, blockReport)
loop
[heartbeat - every 10 seconds]
CreateBlockReport()
Heartbeat(info)
Data Node
Name Node
www.websequencediagrams.com

Data Node

The StoreBlock API is used by the CLI to store new blocks and by the Name Node for fault recovery.

The Data Node will store the block if it's address is in the Data Node list in the StoreBlock API. Then, it will remove it's address from the list and forward the request to another Data Node also in that list.

Technologies and Tools Used

- Go
    - Using `http` library for all REST calls
- Git
    - Storing all code, documents, and images in a private repo
    - GitHub Repository

System Parameters

- Block Size: 64MB
- Replication Factor: 3
- Name Node
    - Replication Checks are done every minute
    - Data Nodes checks are performed every 1 minute
        - During Data Node checks any Data Node that has not had a successful heartbeat in the previous 2 minutes are considered failed
- Data Node
    - Block Reports are sent every 30 seconds
    - Heartbeats are sent every 10 seconds