Distributed Hash Table

A Chord-based DHT Implementation using the Apache Thrift Library

Team Members:

- Danh Nguyen (nguy1952)
- Wen Chuan Lee (leex7095)

For a quick overview on how to run the project as as well test, please go to the **How to Run the Project section**, as well reading the **Client** section regarding the UI commands.

Overall Design

The SuperNode holds a master list of nodes a new node must contact in order to be part of the cluster. Nodes immediately close their connection to the SuperNode after they have successfully joined the cluster and updated the finger tables of all other nodes.

A finger table is stored in every node, it is based on the Chord lookup to allow a node to contact other nodes spread across the cluster in a ring like fashion. Each node is ordered in the ring equentially from zero to the total number of nodes on the cluster.

Clients interact with the distributed hash table through a randomly assigned node by the SuperNode. The node connected to by the client will perform the actual recursive look ups across the entire network to find a file or to write a file.

The minimal supervision of the SuperNode ensures that there isn't too much of intelligence on the SuperNode that the entire network must rely on. We designed to SuperNode as an index and entry point for Nodes and Clients, as well as act as the gatekeeper to the cluster.

The entire DHT is consists of single threaded processes using `join` and `postjoin` on the SuperNode as a mutex lock. This allowed us to avoid most race conditions that may occur in a distributed environment. However, despite being able to create multiple connections across

multiples nodes, **the DHT currently appears to only support one client, for now.**

Recursive look ups on the DHT are `O(log 2^n)` and uses `O(log 2^n)` memory. While we can achieve `O(1)` lookup, each node would have to keep a large table containing the entire cluster, which does not scale.

Marker values indicating a null or empty were also used as Apache Thrift does not support null values.

The SuperNode

`main(String[] args)`

The superNode starts with at a speficied port and an expected number of Nodes that will join the cluster is passed into `main`. A Thrift Server is initialized, allowing Nodes and Clients to connect to the SuperNode.

The SuperNode only allows one node to join the DHT at any one time. Using this blocking method we simplify the need to handle race conditions. On a single thread, we simply keep track of the last node that wants to join the DHT, and disallow other nodes to join until the node has updated the Finger Table across other nodes and perform a `postJoin` call.

The SuperNode also allows a client to connect and receive a node to talk to. Clients should not call `Join` or `PostJoin` on the SuperNode. While Nodes should never call `GetNode`.

`join(Machine node)`

Nodes that want to join the cluster identify themselves using a Machine object containing their own IP Address and Port.

The join function is called by a new Node that wants to join the cluster. The SuperNode returns a list of other nodes if there isn't another node in the process of joining the network. Else a list with a null marker value is returned to the Node.

A variable `lastJoiningNode` in the SuperNode is set by this function to act as a 'mutex lock'

to avoid other nodes attempting to join the node. Synchrony across multiple machines/process is achieved through this method. (As a node will call `postJoin`).

`postJoin(Machine node)`

After a node has contacted all other nodes on the cluster (if any), this function `postJoin` is called. PostJoin first performs another safety check to see if the node that has called `postJoin` is equal to the `lastJoiningNode`, if it is, the superNode adds the node to the master list of nodes on the cluster and sets the `lastJoiningNode` to `null`, allowing the next node to join.

The master list of nodes is also printed for an overview of the current system.

`getNode()`

Function called by a Client after it has connected to the SuperNode. This function keeps the promise that a Client is not allowed to join until the minimum number of nodes specified have joined the cluster. This is done by returning a null Machine (a Thrift object with a null marker value) if there are not enough nodes on the cluster yet.

Once there are at least the minimum specified number of nodes on the cluster, a Client will be allowed to join the cluster. The call to `getNode` will also prevent future nodes from attempting to join the cluster, as the scope of this project does not include addtional nodes joining after the client has joined.

Node

`FindMachine(String Filename, List<Integer> Chain)`

A RPC function used to find the machine that has the file specified by the file name. First the filename is hashed then to locate which machine it would be put into the module of the number of machines in the system is taken. Read and write depend on the function FindMachine to find a machine that has the resource the user is requesting to read or write. If the machine is the current machine then no RPC calls have to be made. Otherwise the

current machine will have to make hops across the network to find the resource.

findMachine's recursion to find the resource works as follows:

1. Check if the current machine is in the call chain to prevent loops.
2. Check if the current machine has the file
3. Check if we need to jump to a machine in the DHT
4. Make a recursive RPC call until we reach step 1.
5. Bubble the target machine back to the origin.

FindMachine maintains a chain that it appends to with each node it visits. If it has visited a node thats already in the chain, then it will not call FindMachine again in that node and thus the file does not exist in the system.

If its the case that the file resides in the system there are three scenarios covered in this case:

1. The file resides in the current machine
2. The file resides in a machine in the current machine's DHT
3. The file resides in a successor of a machien in the current machine's DHT

This recursive lookup nature was designed to be similar to the CHORD DHT.

`write(String Filename, String Contents)`

After finding a machine, write behaves as follows depending who that machine is:

1. If it is the current machine, write the file and the contents to the the current machine's in memory filesystem.
2. If it is located across the network, make a write call through the use of RPC on that machine, passing the file and it's contents.

This will accomplish putting the file the user uploaded into the system.

If the file does not exist in the system, FindMachine will return a invalid machine and return false to the client.

`read(String Filename)`

After finding a machine, read behaves as follows depending who that machine is:

1. If it is the current machine, read the file from the in memory file system and return it.
2. If it is located across the network, make a read call through the use of RPC on that machine, passing the filename.

This will accomplish returning the contents of a requested resource.

If the file does not exist in the system, `findMachine` will return a invalid machine and read will detect this and return a blank to the user.

`updateDHT(List<Machine> NodesList, List<Integer> Chain)`

UpdateDHT is split into two parts:

1. A call to update it's own finger table
2. A recursive call to update all machine's in it's finger table.

Upon calling UpdateDHT, the current machine passes the list of Nodes to it's finger table where the DHT class it houses will calculate the successors at each index.

Afterwards the current machine will make recursive RPC calls to all machine's in it's finger table to update their finger tables. While traversing machines, a chain is passed throughout the process to keep track of machines that have already been traversed to prevent loops and redundancy.

`main(String[] args)`

The node first starts up by connecting to the SuperNode and making a `join` call through Thrift. If there are no other nodes joining the SuperNode will return a list of nodes for the node to contact.

If the SuperNode returns a null marker value to `join`, the Node will go to sleep (by forcing the Thread to sleep) for a second before performing the call to SuperNode again.

Upon Node performaing a successful call to `join`, the node then initializes itself witha nodeID that is equal to the size of the list, and then perform recursive calls to each node on the list through a single-threaded process. Each node then also opens connections to other nodes. All nodes on the cluster and notified and the recursive call exists when the chain of contacted nodes contain the nodes that made the call.

After this, the SuperNode's `postJoin` is called to notify the SuperNode it has successfully contacted all nodes on the network, and closes the connection to the SuperNode.

The Node then starts it's own Thrift Server allowing Clients to connect to them, and waits.

DHT

`searchDHT(String filename, Integer target)`

SearchDHT handles two scenarios:

1. If the machine resides in the DHT
2. Return a machine in the fingertable with the ID less than or equal to the target.

If the machine resides in the DHT then it will return that a object that contains information to connect to that machine.

If the machine does not reside in the DHT then the table will need to make a network hop to a successor machine. So the the fingertable will return the first machine with a ID less than or equal to the target machine.

SearchDHT is called whenever a node joins the system so the recursive nature of this call allows the nodes in the system to reindex to account for new members joining the system.

`Update(List<Machine> NodesList>`

Given a NodesList then the machine will perform the calculation:

```
successor = (v + 2^i) % number of machines
```

... for each index up to the number of indexes, where `number of indexes = lg2 number of machines`.

The successor is then put into that index and thus forms the finger table.

Client

The Client is a terminal to the Distributed Hash Table.

The Client will establish a connection to the SuperNode and perform a `getNode()` Thrift call. This will either succeed (with the SuperNode returning a random Node to contact), or fail (as there are not enough nodes on the cluster yet.

If the Client is successful, the connection to the SuperNode is closed and the Client connects to the Node provided by the SuperNode, an simple interactive terminal asking for user input is then launched.

If the Client is unsuccessful, the Client will go to sleep for one second before retrying indefinitely.

The terminal contains a few simple commands to interact with the DHT.

- `get <filename>` - requests a file from the DHT (Thrift call is made to the Node, which the Node then does the lookup recursively.
- `put <filename>` - reads a file stored in `uploads` and uploads it to the Node, which the Node then does the lookup recursively to decide which Node should store the file and contents.
- `ls` - lists all files in the `uploads` directory
- `put-all` - performs a batch upload operation to put all files stored in `uploads` into the DHT.
- `exit` - closes the connection to the Node and quits the interactive terminal

How to Run the Project

The Chord-based DHT was built with the help of Ant. A prerequisite to run this project requires Ant. Additionally, the generated thrift files were also kept in a version control, but the `Thrift` compiler is also needed if Thrift files need to be generated (else only the Java Thrift library is needed).

Ant was used to handle automatic building and generation of class files (into a separate `bin` directory) as well as creating short targets for rapid developement.

To start the SuperNode and Nodes on localhost

(This starts all processes on one terminal using forked processes and will cause all processes to print to the same `System.out`)

```
ant start-all
```

To start the DHT across multiple machines

```
1. ssh username@x32-05.cselabs.umn.edu
2. cd into project directory
3. ant start-supernode
4. ssh username@x32-XX.cselabs.umn.edu (open another terminal)
5. cd into project directory
6. ant start-node
7. Repeat steps 4 - 6 for the other Nodes
8. ssh username@x32-XX.cselabs.umn.edu (or open another terminal)
9. cd into project directory
10. ant start-client
```

The nodes and clients connect to assume the supernode is located at `x32-05` on port `9090`, **to override this, on Step 6 do**:

```
ant -DsuperNode.address='x32-XX' -DsuperNode.port='XXXX' start-node
```

And on **Step 10 do**:

```
ant -DsuperNode.address='x32-XX' -DsuperNode.port='XXXX' start-client
```

Please see Ant Targets and Overriding Properties for more information.

Ant Targets

The fastest way to test out the entire DHT is to create 1 SuperNode and multiple Nodes on the same machine (but with a different port). Multiple targets have been provided for automating the process of starting everything.

Note: Run commands from the project directory, where `build.xml` is localed.

- `ant start-all`
    - Create 1 SuperNode (port 9090) and 5 Nodes (ports 8000, 8001, 8002, 8003, 8004), and a Client on the same machine.
    - All processes run in their own Java VM and as a forked process.
    - Client waits for 4 seconds before deciding to joining the cluster.
    - Note: see Ant Properties on how to start targets with different values.
- `ant start-supernode`
    - Starts the superNode on the current machine on the port specified by `superNode.port` with `superNode.minNode` number of nodes.
- `ant start-node`
    - Starts a node that will connect to the superNode located at `superNode.address` and `superNode.port`.
- `ant start client`
    - Starts the client that connects to the supernode specified under

`superNode.address` and `superNode.port`
    - *The client will not be allowed to join until there is a minimum number of nodes in the cluster.*
    - *After the client has joined, no new nodes are allowed to join the DHT.*

- `ant thrift`
    - Fires up the thrift compiler to generates Java versions of the `.thrift` files located in `src` and saves them to `src/gen-java`.
    - This target should be called everytime the new *thrift* interfaces have been defined pr to update to a newer version of *thrift*.
    - Targets must be modified if additional *thrift* files are created, as I could not find a way for *Ant* to pass files individually to the *Thrift* compiler without introducing Ant plugins or Maven.
- `ant build`
    - Build but don't run the project. Note all targets that run depend on this target, so there is no need to run `ant build` before running the target.
- `ant clean`
    - Deletes all compiled bytecode in `bin`
- `ant diagnostics`
    - Prints out the Java Class Path defined in *master-classpath*

Ant properties (and overriding them)

Many of the above targets depend on some known information about the SuperNode and the ports to connect to. Sometimes these constants should be overridden (for example when starting multiple nodes/clients on the same machine). **To Override these constants**, use the format:

```
-D(propertyName)=(value)
```

Examples:

To start the SuperNode with a minimum of `10` nodes:

```
ant -DsuperNode.minNode=10 start-supernode
```

With a different port number `5555`:

```
ant -DsuperNode.minNode=8 -DsuperNode.port=5555 start-supernode
```

Alternatively open up `build.xml` and edit the properties by hand:

```
<!--Default SuperNode and Node properties-->
<property name="superNode.address" value="localhost" />
<property name="superNode.port" value="9090"/>
<property name="superNode.minNode" value="5"/>
<property name="node.port" value="8080"/>

<property name="src.dir" value="src" />
<property name="build.dir" value = "bin"/>
<property name="thrift.install" value="/usr/local/Thrift/" />
```

Test Cases

Expected Finger Table For 5 Nodes

```
Machine 0:
Index | Machine
0     | 1 = (0 + 2^0 % 5)
1     | 2 = (0 + 2^1 % 5)

Machine 1:
Index | Machine
0     | 2 = (1 + 2^0 % 5)
1     | 3 = (1 + 2^1 % 5)

Machine 2:
Index | Machine
0     | 3 = (2 + 2^0 % 5)
1     | 4 = (2 + 2^1 % 5)

Machine 3:
Index | Machine
0     | 4 = (3 + 2^0 % 5)
1     | 0 = (3 + 2^1 % 5)


Machine 4:
Index | Machine
0     | 0 = (4 + 2^0 % 5)
1     | 1 = (4 + 2^1 % 5)
```

Positive Scenarios

Put a file in the machine

Keep putting in different files until you find a file that was put in the same node you were given. The expected result if the machine the client is talking to is 2:

```
put Bet.txt
[java] Client: Writing Bet.txt to DHT
[java] New file name Bet.txt
[java] Success: true
[java] Bet.txt was written to Node2
```

Put a file in a successor machine in the DHT

Look at your machine’s finger table and add files until you find a file that is added to a successor.

```
put Staunch.txt
[java] Client: Writing Staunch.txt to DHT
[java] New file name Staunch.txt
[java] Success: true
[java] Staunch.txt was written to Node3
```

Put a file in a successor machine of a successor machine

Since finger table for Machine 2 contains pointers to Machines 3 & 4. When a file is put, then it has taken two successor hops.

```
put Myopia.txt
[java] Client: Writing Myopia.txt to DHT
[java] New file name Myopia.txt
[java] Success: true
[java] Myopia.txt was written to Node0
```

Get a file in the machine

The only node that should be traversed is Node 4(the current machine).

```
get Bet.txt
[java] Client: Reading Bet.txt from DHT
[java] Content :
[java]     Bet
```

Get a file in the successor machine in the DHT

```
get Staunch.txt
[java] Client: Reading Staunch.txt from DHT
[java] Content :
[java]     Staunch
[java] Machine(2) Contacting Machine(3)
```

Get a file in a successor machine of a successor machine

```
get Myopia.txt
[java] Client: Reading Myopia.txt from DHT
[java] Content :
[java]     Myopia
[java] Machine(2) Contacting Machine(4)
[java] Machine(4) Contacting Machine(0)
```

Negative Scenarios

Put a file you dont have

```
put helloworld.txt
[java] Client: Writing helloworld.txt to DHT
[java] Client: Not a file or file doesn't exist.
[java] Success: false
```

Read a file that hasn’t been put

```
get Myopia.txt
[java] Client: Reading Myopia.txt from DHT
[java] Machine(2) Contacting Machine(4)
[java] Machine(4) Contacting Machine(0)
[java] Content :
[java]
[java] ========  DHT: ERROR 404 FILE NOT FOUND IN DHT. ========
[java]
```

Node tries to Join after a Client has joined

```
`ant start-node` called after client has joined the DHT
[java] Connected to SuperNode.
[java] Could not join cluster, retrying ...
```