

# Federated Training System PA2 - Design Document

## Supernode Design Overview

Supernode is the central coordinator of membership in our distributed machine learning platform. It serves as the entry point for compute nodes to become part of the network, providing basic services to facilitate the joining process. Through maintaining node IDs and keeping a list of active nodes in the registry, the Supernode maintains the Chord-like ring well-formed and keeps tabs on the compute nodes that currently reside within the ring, even though it itself does not directly participate in joining nodes.

## Responsibilities

As a compute node desires to join the network, it joins with the Supernode through the `request_join` function. The Supernode assigns the node a unique ID as a hash of its port address based on the SHA1 algorithm of a 6-bit address space. It returns this unique ID to the node so that it can join the network. Once the node is joined, it informs the Supernode of the successful integration by invoking the Supernode back through the `confirm_join` method. The model assumes that one node can join the network at a time; if a node is not yet complete in joining, other nodes will wait.

Nodes and clients, during bootstrapping, request an initial entry into the network from the Supernode but only if there is another node or more. The Supernode maintains two lists: pending nodes containing nodes who have initiated the process of joining, and a list of joined nodes who are in the network completely. A starting point is selected from the list of joined nodes.

Furthermore, the Supernode stores every node's IP address, port, and node ID in a shared repository. It can be queried with this information to obtain the IP address of a specified node if its node ID is provided.

## Core Data Structures

`active_nodes`: A mapping of node ID to a tuple of (IP, port) of all nodes that have registered or are joining.

`nodes_in_network`: A dictionary containing just the nodes which have fully confirmed membership; it is primarily used for bootstrapping and choosing nodes at random.

`pending_node` and `pending_node_info`: Temporary holders of a node which has requested to join but has not yet completed the process, so that it can move seamlessly from pending to active membership.

## **Thrift Interface**

The system uses a `JoinResponse` struct with an integer `id` and an integer `no_of_nodes`. The struct is used to assign a node an identity and indicate the size of the network at present when requesting membership through the `request_join` method. The service further includes:

`confirm_join`: Completes adding the node to the network.

`get_node`: Gives a randomly selected active node to bootstrap from.

`get_compute_node_address`: Gives the network address of a node given its ID.

`remove_join`: Removes a node if it needs to be removed.

## **Conclusion**

The Supernode is resilient and lightweight, with optimal handling of node joins and a dynamic registration of network members. Its support of random bootstrapping and address resolution aids the fault-tolerant and scalable structure of the distributed machine learning system as a whole.

# Compute Node Design Overview

## Overview

Every Compute Node is an independent unit of the Chord-like network with the responsibility of accepting, forwarding, and running training jobs. A Compute Node joins the network by communicating with the Supernode, setting its routing with a finger table along with predecessor and successor pointers, and managing training for the designated files. The node performs training at the same time, saves the gradient model for each file, and returns the outcome to the client upon request.

## Responsibilities

Each Compute Node is responsible for obtaining its own unique identifier and the original gateway to the distributed hash table (DHT) network. Upon establishing a connection, the node initializes its routing by invoking the fix finger function, which recursively computes its own successor for each start entry in the finger table. The successor is identified by identifying the smallest node ID greater than the start entry. After identifying the correct range and its successor, the predecessor and successor pointers of the node are set appropriately.

If a node is the first one to join the network, it requires only to initialize its own finger table. If, however, there are already other nodes, the new node updates not just its finger table but also assists in updating the finger tables of the other nodes to provide ongoing network reachability.

Since the communication is done via Thrift, each node must be aware of the IP address and port number of neighboring nodes. For simplicity in parameter passing, it's assumed that if a node knows the target node ID, it might retrieve the target node's corresponding IP address with an additional RPC request.

The finger table is made up of entries that have a start value—in the instance of an interval—and the successor node of the interval. This is crucial in order for nodes to update their own finger tables recursively. Once the finger table is initialized and updated, the node proceeds to notify the Supernode of its successful addition to the network, and this is when its own ID and IP address are added to the list of active nodes.

After the network setup has been completed, the data is transmitted by the clients to the network. When a file has been received, the node hashes the file using the SHA1 algorithm to convert it into a 6-bit space and assigns an ID to the file. The node then checks whether the file ID is in its own address space; if not, it forwards the file to the closest node that services that hashed ID. When the data reaches the correct node, training starts immediately. The node develops a model using a machine learning library and starts processing the file. After training, the node caches the gradient for every file in a local dictionary keyed on the filename, with the node ID used to route future traffic.

When a client uses the `get_model` method to obtain a trained model, the filename is hashed under the same process to discover the corresponding node. By querying the finger table of the successor of the hashed key, the request is routed to the proper node. Once training is complete, the node sends the trained model; otherwise, it provides a status report that training is ongoing.

In addition, the Compute Node also calculates gradients of the trained model.

## Core Data Structures

**Finger Table:** Sustains routing data with each node having a start value (beginning of an interval) and the corresponding node ID responsible for that interval. This is employed to quickly retrieve the successor of a given key.

**active\_nodes Dictionary:** Track members within the network by correlating a node ID with its IP and port, for node-to-node communication.

**Models Dictionary:** Store training results or gradient updates for each file by using the Thrift-specified struct `Model` with matrices  $V$  and  $W$ , train error, and status.

**Training Status Dictionary:** Track the training status of each file, from pending to in progress to complete.

## Thrift Interface

Inside the namespace `"py compute_node"` of the Thrift definition, a `Model` struct contains the results of training, which are:

Two matrices, V and W (as lists of lists of doubles)

An error\_rate (as a double)

A status string reporting the status of training

The matching ComputeNode service provides a complete interface for the distributed environment, including:

put\_data (asynchronous): Inserts a filename into the network.

get\_model: Retrieves the trained model for an input file.

Chord-specific operations:

fix\_fingers: Updates the node's finger table.

find\_successor and find\_predecessor: Locate nearby nodes in the ring.

update\_finger\_table: Modifies specific finger table entries.

Helper Methods:

get\_predecessor, get\_successor, set\_predecessor, closest\_preceding\_node, and set\_successor assist in handling and querying the network topology.

print\_finger\_table: Provides diagnostic output by displaying the state of the finger table.

## **Client Node Design Overview**

## Overview

The Client is the gateway to distributed training. Its main responsibilities include kicking off the training, sending out the training files to compute nodes, and gathering the results to build a final, validated machine learning model. This is done using Apache Thrift RPC to interact with the Supernode and also the Compute Nodes.

## Key Responsibilities

**Establish Connection:** The client first contacts the Supernode by invoking the `get_node` function to obtain the IP address of an available compute node.

**Distribute Files:** Upon receipt of a connection point, the client connects to the specified compute node and calls the `put_data` function. It sends over each filename in a specified directory (e.g., the "letters" directory) and determines how many files must be dispatched for training.

**Track and Retrieve Training Results:** The client maintains a record of all the files it has sent for training. For each file, it awaits the completion of the training process by invoking the `get_model` function at frequent intervals to retrieve the model gradients.

**Aggregation and Validation:** Once training on a file is complete, the client aggregates the gradients along with the initial weights. If training is not yet complete, the client continues to poll for updates. The global model is updated with the new weights after aggregating the gradients. The client then calculates the final validation error on a special validation dataset (e.g., `validate.txt`).

**Note:** The compute nodes themselves are responsible for routing the data to the appropriate compute nodes. The client also implements a retry mechanism for the `get_model` function so that all models trained are combined appropriately, even if some others are still pending.

## Core Data Structures

**File List:** Client collects filenames of training from a particular directory (typically "letters") to build a list that dictates tasks to be distributed across the network.

**Models Dictionary:** As training results are received, each file's Model—which has gradient matrices, an error rate, and status—is stored in a dictionary with the filename as a key. The organized storage accelerates the aggregation process.

**Aggregation Function:** A custom function (i.e., `average_models`) computes the average of gradients received from various compute nodes. The aggregated result is utilized to update the global model so that all nodes' contributions are effectively incorporated into the final training outcome.

## Assumptions:

- Each node is assigned a unique port.
- The maximum network size will not exceed 10 nodes.
- The maximum number of training files will not exceed five times the initial network size (for example, a network of 5 nodes will have at most 25 files).
- Nodes will not fail or leave the network at any given time.
- The system does not require persistence.

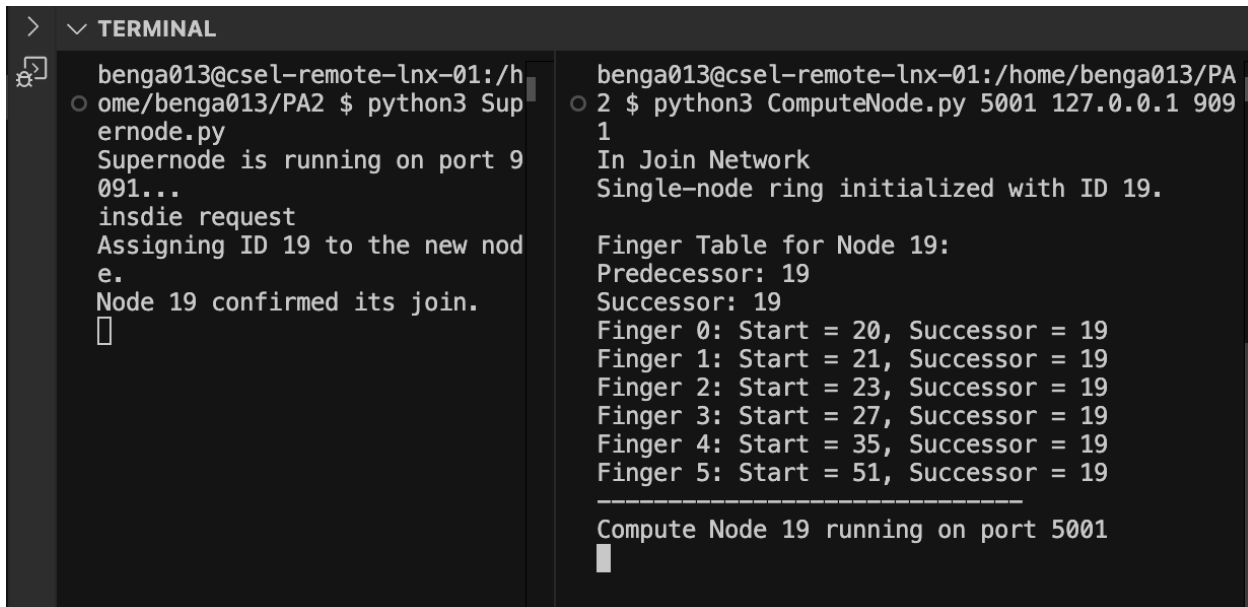
## Test Cases:

### 1. Node Join and Finger Table Updates

- **Test Case 1:**

Single Node Initialization

- **Validation:** We can see that when the first node joins the network its finger tables are updated successfully. It also sets the successor and predecessor nodes correctly.
- Commands:
  - `python3 Supernode.py`
  - `python3 ComputeNode.py 5001 127.0.0.1 9091`



```
> ▾ TERMINAL
benga013@cse1-remote-lnx-01:/home/benga013/PA2 $ python3 Supernode.py
Supernode is running on port 9091...
insdie request
Assigning ID 19 to the new node.
Node 19 confirmed its join.
█

benga013@cse1-remote-lnx-01:/home/benga013/PA2 $ python3 ComputeNode.py 5001 127.0.0.1 9091
In Join Network
Single-node ring initialized with ID 19.

Finger Table for Node 19:
Predecessor: 19
Successor: 19
Finger 0: Start = 20, Successor = 19
Finger 1: Start = 21, Successor = 19
Finger 2: Start = 23, Successor = 19
Finger 3: Start = 27, Successor = 19
Finger 4: Start = 35, Successor = 19
Finger 5: Start = 51, Successor = 19
-----
Compute Node 19 running on port 5001
█
```

### Test Case 2:

Second Node Initialization

- **Validation:** We can see below that when the second node joins the network its finger tables are updated successfully and sets the successor and predecessor nodes correctly. It also updates the finger tables of the other nodes that are present in the network.  
We can see that the finger tables for the node 19 is updated and also its successor and predecessor is updated to 51.
- Commands:
  - `python3 Supernode.py`



- python3 ComputeNode.py 5001 127.0.0.1 9091
- python3 ComputeNode.py 5002 127.0.0.1 9091

```

> ▾ TERMINAL
benga013@cse1-remote-lnx-01:/home/benga013/PA2 $ python3 Supernode.py
Supernode is running on port 9091...
insdie request
Assigning ID 19 to the new node.
Node 19 confirmed its join.
insdie request
Assigning ID 51 to the new node.
Returning random active node:
ID 19, Address 127.0.0.1:5001
Node 51 confirmed its join.
[]

Finger 2: Start = 23, Successor = 51
Finger 3: Start = 27, Successor = 51
Finger 4: Start = 35, Successor = 19
Finger 5: Start = 51, Successor = 19
=====
Processing update request for finger[4] to point to 51
Current finger[4]: 19
Updating finger[4] from 19 to 51
Notifying predecessor 51
Could not connect to any of [('127.0.0.1', 5002)]
Error connecting to compute node 127.0.0.1:5002 - Could not connect to any of [('127.0.0.1', 5002)]
Failed to connect to predecessor

Finger Table for Node 19:
Predecessor: 51
Successor: 51
Finger 0: Start = 20, Successor = 51
Finger 1: Start = 21, Successor = 51
Finger 2: Start = 23, Successor = 51
Finger 3: Start = 27, Successor = 51
Finger 4: Start = 35, Successor = 51
Finger 5: Start = 51, Successor = 19
=====

Updating node 19's finger[3] to point to 51
Before update:

After update:
Updating node 19's finger[4] to point to 51
Before update:

After update:
Skipping finger[5] - predecessor is self

=== Finished Updating Others ===
My final finger table:

Finger Table for Node 51:
Predecessor: 19
Successor: 19
Finger 0: Start = 52, Successor = 19
Finger 1: Start = 53, Successor = 19
Finger 2: Start = 55, Successor = 19
Finger 3: Start = 59, Successor = 19
Finger 4: Start = 3, Successor = 19
Finger 5: Start = 19, Successor = 51
=====
Compute Node 51 running on port 5002

```

### Test Case 3:

#### Third Node Initialization

- **Validation:** When the third node (node 6) joins the network, its finger table is updated successfully, and its predecessor and successor pointers are correctly set. Specifically, node 6's finger table includes entries for both node 19 and node 51, accurately reflecting its position within the ring. Additionally, the finger tables for node 19 and node 51 are updated such that node 6 becomes the predecessor of node 19 and the successor of node 51. This test case confirms that as new nodes join the network, the system maintains the correct routing relationships essential for efficient data lookup and message passing.
- For getting the id of a file we use the following  $\rightarrow \text{sha1}(\text{encoded\_filename}) \% (64)$  This will essentially get the hashed id and perform modulus of the id obtained to wrap it to an address space of 64 ids, which is of 6 bits.
- We can also see the finger tables to determine the address space of the nodes  
Address space of node 6 = 52 to 6  
Address space of node 19 = 7 to 19  
Address space of node 51 = 20 to 51

- Commands:
  - python3 Supernode.py
  - python3 ComputeNode.py 5001 127.0.0.1 9091
  - python3 ComputeNode.py 5002 127.0.0.1 9091
  - python3 ComputeNode.py 5003 127.0.0.1 9091

```

> TERMINAL
1 conf
irmed
its jo
in.
insdie
reque
st
Assign
ing ID
6 to
the ne
w node
.
Return
ing ra
ndom a
ctive
node:
ID 19,
Addre
ss 127
.0.0.1
:5001
Node 6
confi
rmed i
ts joi
n.
[]

Finger 2: Start = 23, Successor = 51
Finger 3: Start = 27, Successor = 51
Finger 4: Start = 35, Successor = 51
Finger 5: Start = 51, Successor = 19

Processing update request for finger
[5] to point to 6
Current finger[5]: 19
Updating finger[5] from 19 to 6
Notifying predecessor 6
Could not connect to any of [('127.0
.0.1', 5003)]
Error connecting to compute node 127
.0.0.1:5003 - Could not connect to a
ny of [('127.0.0.1', 5003)]
Failed to connect to predecessor

Finger Table for Node 19:
Predecessor: 6
Successor: 51
Finger 0: Start = 20, Successor = 51
Finger 1: Start = 21, Successor = 51
Finger 2: Start = 23, Successor = 51
Finger 3: Start = 27, Successor = 51
Finger 4: Start = 35, Successor = 51
Finger 5: Start = 51, Successor = 6

Finger Table for Node 51:
Predecessor: 19
Successor: 6
Finger 0: Start = 52, Successor = 6
Finger 1: Start = 53, Successor = 6
Finger 2: Start = 55, Successor = 6
Finger 3: Start = 59, Successor = 6
Finger 4: Start = 3, Successor = 19
Finger 5: Start = 19, Successor = 51

Processing update request for finger
[4] to point to 6
Current finger[4]: 19
Updating finger[4] from 19 to 6
Notifying predecessor 19

Finger Table for Node 51:
Predecessor: 19
Successor: 6
Finger 0: Start = 52, Successor = 6
Finger 1: Start = 53, Successor = 6
Finger 2: Start = 55, Successor = 6
Finger 3: Start = 59, Successor = 6
Finger 4: Start = 3, Successor = 6
Finger 5: Start = 19, Successor = 51

Before update:
After update:
Updating node 51's finger[4] to point to 6
Before update:
After update:
Updating node 19's finger[5] to point to 6
Before update:
After update:
=== Finished Updating Others ===
My final finger table:
Finger Table for Node 6:
Predecessor: 51
Successor: 19
Finger 0: Start = 7, Successor = 19
Finger 1: Start = 8, Successor = 19
Finger 2: Start = 10, Successor = 19
Finger 3: Start = 14, Successor = 19
Finger 4: Start = 22, Successor = 51
Finger 5: Start = 38, Successor = 51

Compute Node 6 running on port 5003

```

## 2. Model Training

### Test Case 4:

Training a subset of the files and model validation with 2 nodes

- **Validation:** In this test case, the system demonstrates distributed training using two Compute Nodes. After starting the Supernode, two Compute Nodes join the network, each obtaining a unique ID and correctly setting up their finger tables, predecessors, and successors. The client then sends a subset of training files via a put\_data call, and the receiving nodes either process or forward the files based on their hashed IDs. As training progresses concurrently on both nodes, the client retrieves the model gradients with get\_model and aggregates them to update the global model. Finally, validation on a designated dataset confirms that the distributed training and model aggregation were successful.
- Commands:
  - python3 Supernode.py
  - python3 ComputeNode.py 5001 127.0.0.1 9091
  - python3 ComputeNode.py 5002 127.0.0.1 9091

- python3 client.py 127.0.0.1 9091

```

> TERMINAL
st Assign ing ID 51 to the new node. Return ing random active node: ID 19, Address 127.0.0.1:5001 Node 51 configirmed its join. Return ing random active node: ID 51, Address 127.0.0.1:5002

Processing file: FF1I48s675
Path taken: 19 -> 51 -> 19
Predecessor: 51
Successor: 51
Finger Table:
Entry 0: Start 20 -> Node 51
Entry 1: Start 21 -> Node 51
Entry 2: Start 23 -> Node 51
Entry 3: Start 27 -> Node 51
Entry 4: Start 35 -> Node 51
Entry 5: Start 51 -> Node 19
=====
Get Model - filename: FF1I48s675 ID: 61
Computed gradients for FF1I48s675 (error rate: 0.2380)

==== Node 19 ====
Processing file: FF1I48s675
Path taken: 19 -> 51 -> 19
Predecessor: 51
Successor: 51
Finger Table:
Entry 0: Start 20 -> Node 51
Entry 1: Start 21 -> Node 51
Entry 2: Start 23 -> Node 51
Entry 3: Start 27 -> Node 51
Entry 4: Start 35 -> Node 51
Entry 5: Start 51 -> Node 19
=====
Get Model - filename: FF1I48s675 ID: 61

Computed gradients for byKwt8V4pT (error rate: 0.2700)

==== Node 51 ====
Processing file: FF1I48s675
Path taken: 51 -> 19
Predecessor: 19
Successor: 19
Finger Table:
Entry 0: Start 52 -> Node 19
Entry 1: Start 53 -> Node 19
Entry 2: Start 55 -> Node 19
Entry 3: Start 59 -> Node 19
Entry 4: Start 3 -> Node 19
Entry 5: Start 19 -> Node 51
=====
Get Model - filename: byKwt8V4pT ID: 39

benga013@cscel-remote-lnx-01:/home/benga013/PA
2 $ python3 client.py 127.0.0.1 9091
Connected to node 51 at 127.0.0.1:5002
Submitted FF1I48s675 61
Submitted byKwt8V4pT 39
Waiting for FF1I48s675...
Waiting for byKwt8V4pT...
Retry 1/10
Waiting for FF1I48s675...
Waiting for byKwt8V4pT...
Retry 2/10
Waiting for FF1I48s675...
Waiting for byKwt8V4pT...
Retry 3/10
Waiting for FF1I48s675...
Waiting for byKwt8V4pT...
Retry 4/10
Waiting for FF1I48s675...
Waiting for byKwt8V4pT...
Retry 5/10
Waiting for FF1I48s675...
Waiting for byKwt8V4pT...
Retry 6/10
Acquired model for FF1I48s675
Acquired model for byKwt8V4pT
Aggregating models...
Final validation error: 0.38%
benga013@cscel-remote-lnx-01:/home/benga013/PA
2 $

```

Training with 2 compute nodes with eta as 250 showing convergence with 2 files

## Test Case 5:

Training a subset of the files and model validation with 3 nodes

- **Validation:** In this test case, three Compute Nodes join the network, each obtaining a unique node ID and setting up their ring-based routing (including finger tables, predecessors, and successors). The client sends two files for training, which are distributed among the three nodes based on the file hashes. As each node processes its assigned files, the logs confirm that the nodes' finger tables are updated correctly to maintain proper routing. Once training completes, the client retrieves the models from the responsible nodes, aggregates the results, and validates the final model. This demonstrates the system's ability to handle distributed training and model validation effectively with three Compute Nodes.
- **Commands:**
  - python3 Supernode.py
  - python3 ComputeNode.py 5001 127.0.0.1 9091
  - python3 ComputeNode.py 5002 127.0.0.1 9091
  - python3 ComputeNode.py 5003 127.0.0.1 9091
  - python3 client.py 127.0.0.1 9091

```

benga013@cse1-remote-lin
x-01:/home/benga013/PA2
$ python3 Supernode.py
Supernode is running on
port 9091...
insdie request
Assigning ID 19 to the
new node.
Node 19 confirmed its j
oin.
insdie request
Assigning ID 51 to the
new node.
Returning random active
node: ID 19, Address 1
27.0.0.1:5001
Node 51 confirmed its j
oin.
insdie request
Assigning ID 6 to the n
ew node.
Returning random active
node: ID 51, Address 1
27.0.0.1:5002
Node 6 confirmed its jo
in.
Returning random active
node: ID 51, Address 1
27.0.0.1:5002
[]

m 19 to 6
Notifying predecessor
6
Could not connect to a
ny of [('127.0.0.1', 5
003)]
Error connecting to co
mpute node 127.0.0.1:5
003 - Could not connec
t to any of [('127.0.0
.1', 5003)]
Failed to connect to p
redecessor

Finger Table for Node
19:
Predecessor: 6
Successor: 51
Finger 0: Start = 20,
Successor = 51
Finger 1: Start = 21,
Successor = 51
Finger 2: Start = 23,
Successor = 51
Finger 3: Start = 27,
Successor = 51
Finger 4: Start = 35,
Successor = 51
Finger 5: Start = 51,
Successor = 6
=====
[]

Node 6
Entry 3: Start 59 ->
Node 6
Entry 4: Start 3 -> N
ode 6
Entry 5: Start 19 ->
Node 51
=====
=== Node 51 ===
Processing file: byKwT8
V4pT
Path taken: 51 -> 6 ->
51
Predecessor: 19
Successor: 6
Finger Table:
Entry 0: Start 52 ->
Node 6
Entry 1: Start 53 ->
Node 6
Entry 2: Start 55 ->
Node 6
Entry 3: Start 59 ->
Node 6
Entry 4: Start 3 -> N
ode 6
Entry 5: Start 19 ->
Node 51
=====
Get Model - filename:
byKwT8V4pT ID: 39
[]

Entry 5: Start 38 ->
Node 51
=====
Get Model - filename:
FF1I48s675 ID: 61
Computed gradients for
FF1I48s675 (error rate:
0.2380)
=== Node 6 ===
Processing file: FF1I48
s675
Path taken: 6 -> 51 ->
6
Predecessor: 51
Successor: 19
Finger Table:
Entry 0: Start 7 -> N
ode 19
Entry 1: Start 8 -> N
ode 19
Entry 2: Start 10 ->
Node 19
Entry 3: Start 14 ->
Node 19
Entry 4: Start 22 ->
Node 51
Entry 5: Start 38 ->
Node 51
=====
Get Model - filename:
FF1I48s675 ID: 61
[]

Waiting for byKwT8V4pT..
.
Retry 2/10
Waiting for FF1I48s675..
.
Waiting for byKwT8V4pT..
.
Retry 3/10
Waiting for FF1I48s675..
.
Waiting for byKwT8V4pT..
.
Waiting for FF1I48s675..
.
Waiting for byKwT8V4pT..
.
Retry 4/10
Waiting for FF1I48s675..
.
Waiting for byKwT8V4pT..
.
Retry 5/10
Waiting for FF1I48s675..
.
Waiting for byKwT8V4pT..
.
Retry 6/10
Acquired model for FF1I4
8s675
Acquired model for byKwT
8V4pT
Aggregating models...
Final validation error:
0.38%
benga013@cse1-remote-lin
x-01:/home/benga013/PA2 $

```

### 3. File Distribution and Routing

#### Test Case 6:

Get model routing

#### Validation:

Submitting 20 files to observe the routing of the files. We can see the hashed id that is shown in the client for reference. We can see the respective terminals where we find the model, this shows where the model is used to get after routing. We can verify the following.

File with id 15 is found in node 19 as 15 belongs to the address space owned by node 19. Similarly we can find a file with id 35 in the node 51 and the file with id 2 in the node 6.

For getting the id of a file we use the following ->  $\text{sha1}(\text{encoded\_filename}) \% (64)$  This will essentially get the hashed id and perform modulus of the id obtained to wrap it to an address space of 64 ids, which is of 6 bits.

#### Commands:

- python3 Supernode.py
- python3 ComputeNode.py 5001 127.0.0.1 9091
- python3 ComputeNode.py 5002 127.0.0.1 9091
- python3 ComputeNode.py 5003 127.0.0.1 9091
- python3 client.py 127.0.0.1 9091

```
python3 client.py 127.0.0.1 9091
Connected to node 19 at 127.0.0.1:5001
Submitted FF1I48s675 61
Submitted byKWt8V4pT 39
Submitted lmLheWMOVV 15
Submitted tLrRvgmGK3 48
Submitted uLghCnZwzj 2
Submitted y9HtLcmrkD 16
Submitted zx0uRKUQj5 58
Submitted NGjFPtxrBh 6
Submitted fcR2pg5Jdx 7
Submitted FI7SzKMGBs 59
Submitted V0msie6NEw 63
Submitted qmdnz8v9H5 57
Submitted c5nGyKeD26 58
Submitted Yor05gpN2K 26
Submitted rM9QAoFYcB 39
Submitted 3m4oh4vcuR 42
Submitted PxeA7dV8Qs 5
Submitted YHCHiXhKWy 35
Submitted 26UFHSJHcQ 15
Submitted 2aM2VBjKp1 42
Waiting for FF1I48s675...
Waiting for byKWt8V4pT...
Waiting for lmLheWMOVV...
```

```
=====  
Get Model - filename: lmLheWMOVV ID: 15
```

```
=== Node 19 ===  
Processing file: tLrRvgmGK3  
Path taken: 19 -> 51  
Predecessor: 6  
Successor: 51  
Finger Table:  
Entry 0: Start 20 -> Node 51  
Entry 1: Start 21 -> Node 51  
Entry 2: Start 23 -> Node 51  
Entry 3: Start 27 -> Node 51  
Entry 4: Start 35 -> Node 51  
Entry 5: Start 51 -> Node 6  
=====
```

```
=====  
Get Model - filename: YHCHiXhKWy ID: 35
```

```
=== Node 51 ===  
Processing file: 2aM2VBjKp1  
Path taken: 51 -> 6 -> 51  
Predecessor: 19  
Successor: 6  
Finger Table:  
Entry 0: Start 52 -> Node 6  
Entry 1: Start 53 -> Node 6  
Entry 2: Start 55 -> Node 6  
Entry 3: Start 59 -> Node 6  
Entry 4: Start 3 -> Node 6  
Entry 5: Start 19 -> Node 51  
=====
```

```
=====  
Get Model - filename: uLghCnZwzj ID: 2
```

```
=== Node 6 ===  
Processing file: zx0uRKUQj5  
Path taken: 6 -> 51 -> 6  
Predecessor: 51  
Successor: 19  
Finger Table:  
Entry 0: Start 7 -> Node 19  
Entry 1: Start 8 -> Node 19  
Entry 2: Start 10 -> Node 19  
Entry 3: Start 14 -> Node 19  
Entry 4: Start 22 -> Node 51  
Entry 5: Start 38 -> Node 51  
=====
```

### Test Case 7:

Routing and printing its information

- **Validation:**

We can see that for each data routing operation, the path that is taken is printed. Each of the node id that it goes to is added to the list and it is printed. For example `zx0uRKUQj5` has the id 58. So it is routed from 19 to 51 first. And then from the finger tables of 51 it goes to the destination 6 where it is trained.

This is because the finger tables of node 19 contains information only till the node 51 which is it's last finger table entry. Therefore 19 forwards the file to 51. 51 then finds the appropriate entry for 58. It finds that 58 belongs to node 6 in its finger table and then sends the file to node 6 for training.

```
Submitted zx0uRKUQj5 58
```

```
Processing file: zx0uRKUQj5  
Path taken: 19 -> 51 -> 6
```

### Test Case 8:

Put Data Routing

- **Validation:**

As is apparent from the output, every file is hashed into a specific ID, which denotes the node where it is trained. For instance, first file gets an ID of 15 and is sent to node 19, and second file gets an ID of 35 and is processed by node 51. Similarly, the third file gets an ID of 2 and gets trained on node 6. This confirms that the system correctly hashes and sends files based on their IDs and ensures that each file is processed by the node responsible for its corresponding address space.

- **Commands:**

- `python3 Supernode.py`
- `python3 ComputeNode.py 5001 127.0.0.1 9091`
- `python3 ComputeNode.py 5002 127.0.0.1 9091`
- `python3 ComputeNode.py 5003 127.0.0.1 9091`
- `python3 client.py 127.0.0.1 9091`

<pre> Entry 5: Start 51 -&gt; Node 6 ===== === Node 19 === Processing file: 26UFHSJHcQ Path taken: 19 -&gt; 6 -&gt; 19 Predecessor: 6 Successor: 51 Finger Table: Entry 0: Start 20 -&gt; Node 51 Entry 1: Start 21 -&gt; Node 51 Entry 2: Start 23 -&gt; Node 51 Entry 3: Start 27 -&gt; Node 51 Entry 4: Start 35 -&gt; Node 51 Entry 5: Start 51 -&gt; Node 6 ===== Starting training for 26UFHSJHcQ on node 19 Put Data - Model - filename: 26UFHSJHc Q ID: 15 </pre>	<pre> ID: 26 hellooo: Yor05gpN2K ===== === Node 51 === Processing file: YHCHiXhKWy Path taken: 51 -&gt; 6 -&gt; 51 Predecessor: 19 Successor: 6 Finger Table: Entry 0: Start 52 -&gt; Node 6 Entry 1: Start 53 -&gt; Node 6 Entry 2: Start 55 -&gt; Node 6 Entry 3: Start 59 -&gt; Node 6 Entry 4: Start 3 -&gt; Node 6 Entry 5: Start 19 -&gt; Node 51 ===== Starting training for YHCHiXhKWy on node 51 Put Data - Model - filename: YHCHiXhKWy ID: 35 </pre>	<pre> Starting training for FF1I48s675 on node 6 Put Data - Model - filename: FF1I48s675 ID: 6 1 hellooo: FF1I48s675 ===== === Node 6 === Processing file: uLghCnZwzj Path taken: 6 -&gt; 51 -&gt; 6 Predecessor: 51 Successor: 19 Finger Table: Entry 0: Start 7 -&gt; Node 19 Entry 1: Start 8 -&gt; Node 19 Entry 2: Start 10 -&gt; Node 19 Entry 3: Start 14 -&gt; Node 19 Entry 4: Start 22 -&gt; Node 51 Entry 5: Start 38 -&gt; Node 51 ===== Starting training for uLghCnZwzj on node 6 Put Data - Model - filename: uLghCnZwzj ID: 2 </pre>
---	---	--

## Test Case 9 :

Two files that have the same id routed to the same node

### Validation:

In this test case, two different file names are hashed using SHA1, and both result in the same ID. This occurs because we do a modulus operation to restrict the address space to 64 entries or 6 bits space. Consequently, both files are directed to the same Compute Node, as that node is responsible for handling the training associated with that specific ID. This scenario confirms that the file hashing mechanism is deterministic and that the node assigned to a particular ID correctly processes all files mapping to that ID.

### ○ Commands:

- python3 Supernode.py
- python3 ComputeNode.py 5001 127.0.0.1 9091
- python3 ComputeNode.py 5002 127.0.0.1 9091
- python3 ComputeNode.py 5003 127.0.0.1 9091
- python3 client.py 127.0.0.1 9091

=== Node 51 ===

Processing file: byKWt8V4pT

Path taken: 51 -> 6 -> 51

Predecessor: 19

Successor: 6

Finger Table:

Entry 0: Start 52 -> Node 6

Entry 1: Start 53 -> Node 6

Entry 2: Start 55 -> Node 6

Entry 3: Start 59 -> Node 6

Entry 4: Start 3 -> Node 6

Entry 5: Start 19 -> Node 51

=====

**Get Model** - filename: byKWt8V4pT ID: 39



=== Node 51 ===

Processing file: rM9QAoFYcB

Path taken: 51 -> 6 -> 51

Predecessor: 19

Successor: 6

Finger Table:

Entry 0: Start 52 -> Node 6

Entry 1: Start 53 -> Node 6

Entry 2: Start 55 -> Node 6

Entry 3: Start 59 -> Node 6

Entry 4: Start 3 -> Node 6

Entry 5: Start 19 -> Node 51

=====

Get Model - filename: rM9QAoFYcB ID: 39

Node 51

## References:

[1] Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications Ion Stoica , Robert Morris, David Karger, M. Frans Kaas

We partially relied on ChatGPT for code suggestions in two areas of our distributed system:

1. Connecting to compute nodes using sockets
2. Reading port and hostname information from a file
3. Returning the value by looking up a dictionary of a specific key
4. A small portion of the code—such as parts of the `train_model` function responsible for file handling, gradient computation, and error management—benefited from AI-based refinements. These minor adjustments helped improve readability and consistency without affecting the core DHT functionality.

## Steps to execute

### 1. Unzip the Folder

- a. Extract the zipped folder (e.g., distributed\_training.zip) into a local directory of your choice.
- b. Navigate to PA2 within the unzipped folder.

### 2. Create Virtual Environment and install all the requirements

- a. `python3 -m venv myenv`
- b. `source myenv/bin/activate`
- c. `pip install -r requirements.txt`
- d. Execute the ComputeNode, Supernode and client in the virtual environment

### 3. Configure Compute Nodes text file

- a. Open the file `compute_node.txt` (if used) and ensure it lists the host and port for each compute node you plan to run.

For example:

127.0.0.1,5001

127.0.0.1,5002

127.0.0.1,5003

### 4. Generate thrift files

Run: `thrift --gen py supernode.thrift`

Run: `thrift --gen py compute_node.thrift`

### 5. Start the Super Node:

`python3 Supernode.py`

Note: Supernode.py runs on port 9091

### 6. Start the Compute Nodes

Open additional terminals—one for each compute node.

`python3 ComputeNode.py <port> <Supernode_ip> <Supernode_port>`

Example:

`python3 ComputeNode.py 5001 127.0.0.1 9091`

`python3 ComputeNode.py 5002 127.0.0.1 9091`

`python3 ComputeNode.py 5002 127.0.0.1 9091`

Each node will start listening on its respective port and be ready to accept tasks from the other nodes.

## **7. Run the Client**

Once the Supernode and ComputeNode are running, open another terminal for the client.

Run:

```
python3 client.py <Supernode_ip> <Supernode_port>
```

Example:

```
python3 client.py 127.0.0.0 9091
```

8. You can find the final Validation after training in the client console.
9. You can find the routing and finger tables in the compute nodes console once each compute node comes up.