Rafbook - documentation

Introduction

This project aims to develop a functional distributed system that manages ASCII-encoded text files and images. The system offers users the following capabilities:

- **Adding New Files:** Users can add new files with unique names and paths to the system.
- **Retrieving Files:** Users can retrieve any file from the distributed system.
- **Adding Friendly Nodes:** Users can add friendly nodes to the system.
- **Deleting Files:** Users can delete files on a local servent.
- **Efficient File Retrieval:** The system's topological organization ensures faster file retrieval.
- **Fault Tolerance:** The system is designed to be resilient to failures.

By integrating these features, the system ensures efficient and reliable management of text files across a distributed network, enhancing both accessibility and robustness.

System configuration

At its initialization stage, a node reads a local configuration file, which has the following properties:
When a node starts, it automatically reads a configuration file specifying the following attributes:

- **Working Root:** The path on the local system where files are stored for operation. (string)
- **Listening Port:** The port on which the node will listen. (short)
- **Bootstrap Node Port:** The port of the bootstrap node. (short)
- **Weak Failure Threshold:** The weak failure threshold. (int)
- **Strong Failure Threshold:** The strong failure threshold. (int)

It is assumed that all nodes will have consistent configuration files, eliminating the need for additional verification. The system may not function correctly if the configuration file is improperly set up.

User commands

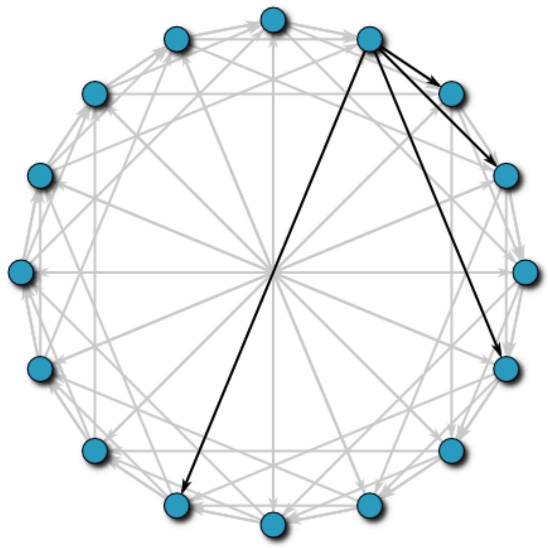
The system allows users to issue the following commands:

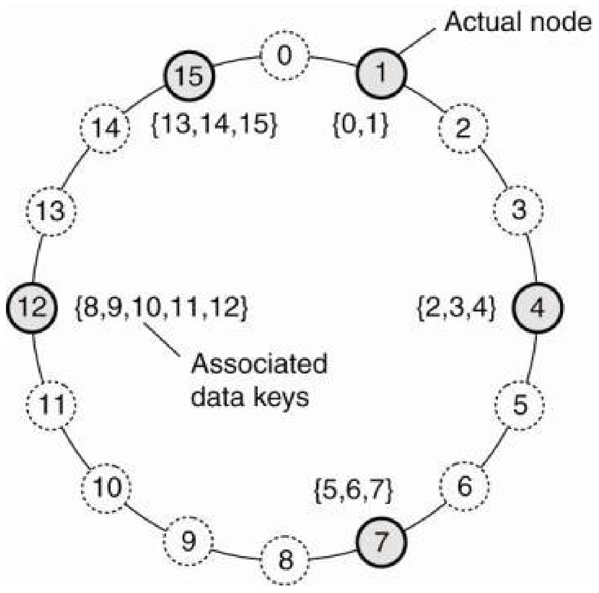
- **add_friend [address]:** Adds a node to the friend list.
- **add_file [path] [private/public]:** Adds a new file to the network with an option for private or public sharing.
- **view_files [address]:** Views public and private files from a friend or another node.
- **remove_file [filename]:** Removes a file from the network.
- **stop:** Gracefully shutdown the node.

For all commands, file names should be provided with paths relative to the working root specified in the configuration file. File names will never contain spaces, so there is no need to support them.

System Implementation

The system is implemented using a Distributed Hash Table (DHT), specifically the **Chord algorithm**. In this DHT implementation, each node has a key derived from hashing and is responsible for a certain range of values. Here, the values represent hashed file names in the virtual file system. Two simple hash functions are used, where the node's port value and the file name are multiplied by prime numbers and then modulo 64.





Each node has a pointer to its first successor and its first predecessor in the chain. Both writing and retrieving keys involve finding the node that is the nearest successor for that key. This search needs to be performed quickly. While nodes could trivially traverse the chain successor by successor, this would have an O(N) complexity, which is not acceptable.

The solution is for each node to maintain a finger table, recording convenient jumps in the chain. Typically, in a system with 2^n nodes, each node keeps a table with n entries, where at node m for the i-th entry in the table (i ranges from 0 to n-1), we have:

$$\boldsymbol{succ_i = sled((m + 2^i) \bmod 2^n)}$$

where succ is a function that gives the nearest successor (i.e., owner) for the requested hash value. With a correctly constructed finger table, the system uses it to find any DHT value as follows:

- Check if the current node or its immediate successor is responsible for the desired value.
- If not, query the node in the finger table with the largest hash that does not exceed the hash of the desired key.
- Repeat the process until the request reaches the node whose direct successor is responsible for the desired value.

This efficient lookup mechanism ensures quick retrieval and storage of files in the distributed system.

Messages and Handlers

New Node

Responsible for the introduction of a new node in the system. If the new node is my predecessor, I need to share all of the elements with the chordId less than the chordId of the new node and remove those elements from my storage, alsoI need to **Welcome it** to the system and notify all the other Nodes so that they can also add them to their network. On the other hand If the new node is not my predecessor, I need to forward the New Node message to the next node determined by the key.

Welcome

Responsible for initializing the new node and all its values in the **Chord State** and sending the Update message to the next node.

Friend Request

Sending a friend request with the parameters of the node that we want to add in our friends list. The payload is the IP address and the port of the node. The port of the node gets hashed so that we can get the chordId of the node. Now we can find the logarithmic path to our potential friend with the key and send them the message.
If we are receiving the Friend Request we need to check the payload if the request is for us. If that is true, we add the Sender Port to our friends list, and send them back a **Friend Accept** message.

Friend Accept

When we are the rightful receiver of a Friend Request message, we add the sender node to our friends list, and return to them a Friend Accept message.
If we are the receiver of a Friend Accept message, we check if we are the rightful receiver of the message by the payload, if we are we add the sender node to our friends list. If we are not the right receiver, we hash the message and send to the next node by the hashed key.

Add File

Add a file to our distributed system. The payload is the files path relative to our workspace directory and an Access type (private/public). The files path gets hashed into an integer key so that we can determine on which node it needs to be stored. If we are responsible for the file, we read all of its contents, create a CloudFile from it and save it in our list. When the file is successfully created we send a Save Backup message to our successor. If we are not responsible for the file, we forward the message to the next node by key.

Save Backup

A save backup message gets sent when a file has been added to the network. It gets sent to the successor of the node that added the file. The payload is the **CloudFile**. When a node failure happens we are sure we don't lose any files, because a successor node holds those files safe.

Remove file

Removes a file from the network. Accepts the payload as the file path. It hashes the file path so that we get the chordId of where the file is stored. If we are responsible for the file, we remove it from our list. If not, we just forward the message along to the next node for the key.

View Files

A message that sends an IP address and port of the node that she is interested in, to see all the files the node has stored. When the message is first being sent, the payload gets hashed so that the message gets sent in logarithmic time. If we are the rightful recipient of the message as a response we send a **Tell Files** message to the sender node with all the file information that is accessible for the sender node. If we are not the rightful recipient, we just forward the message.

Tell Files

A message that is a response to a **View Files** message. Sends a payload of all the files that a sender is accessible to see. If a sender node is in our friends list, it has access to all the private and public files. If a sender node is not our friend it has access only to the public files. The list of files has the information like the file path, file size in Kbs and the access type.

Failure Detection - Heartbeat

The Heartbeat Algorithm in our distributed Chord system is designed to monitor the health and availability of nodes, ensuring the robustness and reliability of the network. This algorithm operates based on periodic heartbeat messages exchanged between nodes. Below is a concise description of its working:

1. **Sending Heartbeats**:
   - Each node periodically sends a heartbeat message to its immediate successor in the Chord ring. This message signifies that the sender node is alive and functioning.
2. **Receiving Heartbeats**:
   - Upon receiving a heartbeat message from its predecessor, a node updates its timestamp, marking the last known active time of the predecessor.
3. **Monitoring Timestamps**:

    - Nodes maintain a record of the last heartbeat received from their predecessor. This timestamp is continuously monitored and compared with the current time.
4. **Detecting Failures**:
    - If the time elapsed since the last received heartbeat exceeds a predefined threshold (known as the strong tolerance), the node assumes that its predecessor has failed.
    - If the elapsed time exceeds a weaker threshold (known as the weak tolerance) but is within the strong tolerance, the node marks the predecessor as possibly dead and may take further steps to confirm its status.
5. **Failure Handling**:
    - When a node is deemed to have failed (after exceeding the strong tolerance), a node removal message is propagated through the Chord ring. This message instructs all nodes to remove the failed node from their routing tables.
    - Additionally, nodes can request their successors to verify the status of the suspected failed node to confirm its unavailability.

Messages used in the Heartbeat algorithm

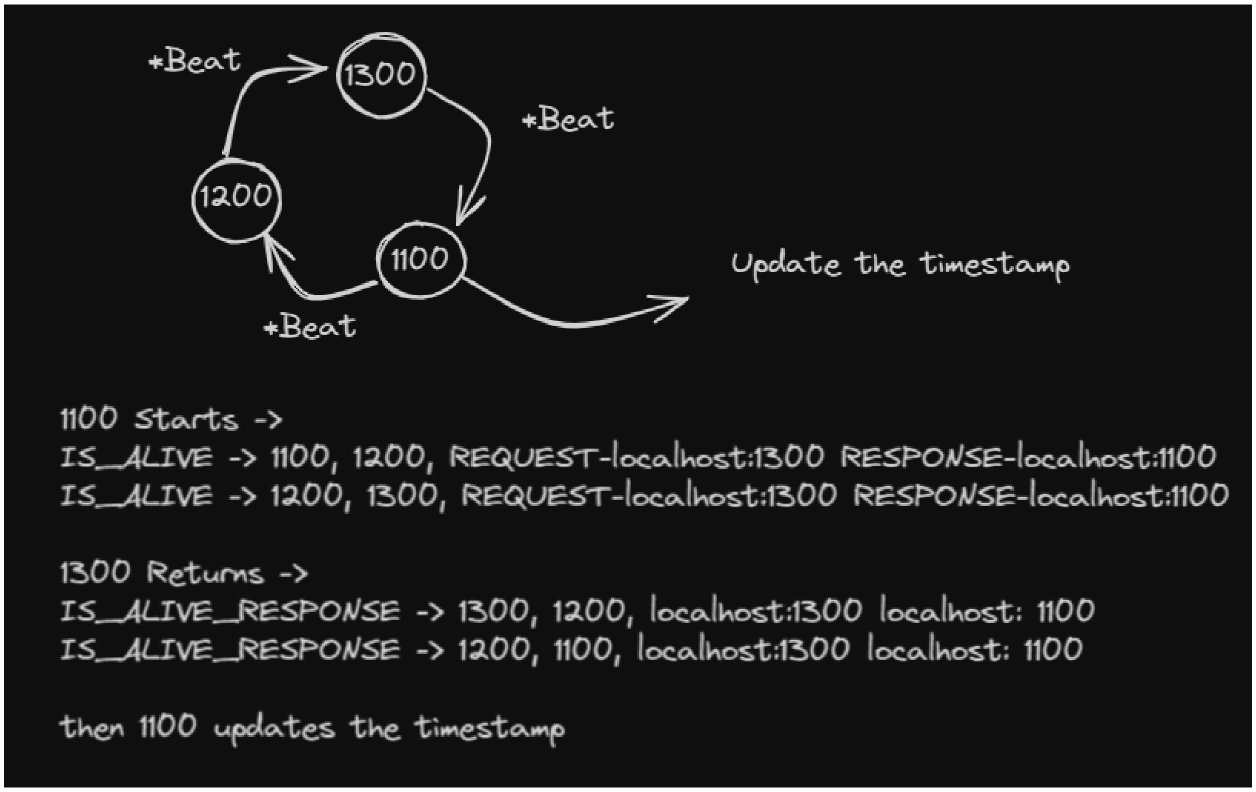
- **Heartbeat Message**:
    - **Purpose**: To periodically notify the successor node that the sender node is still active and functioning.
    - **Details**: Sent at regular intervals by each node to its successor. The message contains the sender's information and helps in updating the successor's last heartbeat timestamp for the sender.
- **IsAlive Message**:
    - **Purpose**: To verify the status of a node suspected of being inactive or failed.
    - **Details**: Sent when a node's predecessor fails to send a heartbeat within the weak tolerance period. The message is directed to the suspected failed node's successor to confirm its status and ensure the reliability of the detection process.

Node Failure and RIPNodeMessage

When a node in the ring detects that its predecessor has failed (i.e., it has not received a heartbeat within the strong tolerance period), the following sequence of events occurs:

- **RIPNodeMessage:**
  - **Purpose:** To notify all nodes in the ring that a specific node has failed and should be removed from the network**.**
  - **Details:** The node detecting the failure sends a RIPNodeMessage to all other nodes in the ring. This message contains the information about the failed node.
- **Updating Successor Tables:**
  - **Upon receiving a RIPNodeMessage:** Each node updates its successor table to remove the failed node and find a new successor if necessary. This ensures the consistency and robustness of the ring structure.
  - **Propagation:** The RIPNodeMessage propagates through the network, allowing all nodes to update their routing information and maintain the ring's integrity.
- **Save backup files if your predecessor has died**

Is Alive Example



```
1100 Starts ->
IS_ALIVE -> 1100, 1200, REQUEST-localhost:1300 RESPONSE-localhost:1100
IS_ALIVE -> 1200, 1300, REQUEST-localhost:1300 RESPONSE-localhost:1100

1300 Returns ->
IS_ALIVE_RESPONSE -> 1300, 1200, localhost:1300 localhost: 1100
IS_ALIVE_RESPONSE -> 1200, 1100, localhost:1300 localhost: 1100

then 1100 updates the timestamp
```

Usage

There are two primary ways to start the application:

1. Running the MultipleServentStarter Class

1. **Navigate to the Main Class**: Locate and open the `MultipleServentStarter` class in your development environment.
2. **Run the Class**: Execute the `MultipleServentStarter` class. This will automatically start multiple nodes as defined in the class configuration.
   - This method is convenient for testing and development purposes, as it starts all necessary nodes with a single command.

2. Configuring Run Configurations Manually

Step-by-Step Guide:

1. **Configure the Bootstrap Node**:
   - **Open Run Configurations**: In your development environment, open the run configurations settings.
   - **Create a New Configuration**: Create a new configuration for the bootstrap class.
     - **Class**: Set the main class to `Bootstrap`.
     - **Program Arguments**: Add the bootstrap port
   - **Save and Run**: Save the configuration and run it to start the bootstrap node.
2. **Configure Each Servent Node**:
   - **Create New Configuration for Each Node**: For each servent node, create a new run configuration.
     - **Class**: Set the main class to `ServentMain`.

Program Arguments: Add the path to the properties file and servent ID. The arguments should look like this:

```
chord/servent_list.properties X
```

- Replace `X` with the appropriate servent ID and name.

3. **Example Configuration for Servent Node 1**:
   - **Class**: `ServentMain`
   - **Program Arguments**: `chord/servent_list.properties 1`
4. **Example Configuration for Servent Node 2**:
   - **Class**: `ServentMain`

    - **Program Arguments**: `chord/servent_list.properties 2`
5. **Save and Run Each Servent Configuration**: Save each configuration and run them sequentially or in parallel as needed.

This manual method is more flexible and allows you to individually configure and start each node, which can be useful for testing specific scenarios or debugging issues in the network.