



UNIVERSITY OF CALIFORNIA IRVINE- WINTER'23

COMPSCI 230: DISTRIBUTED SYSTEMS

P2P FILE DISCOVERY SERVICE

SEPTEMBER 7, 2023

<i>Author</i>	<i>Student ID</i>
Antriksh Ganjoo	77525906
Maganth Seetharaman	64409266
Hemanth Kota	53649643

Contents

Background	3
1 Abstract	3
2 Architecture	3
2.1 Primary Node	3
2.2 Data Node	3
2.3 Node Discovery	3
2.4 Node Health Check	4
2.5 Node Communication	4
3 File Discovery	4
3.1 File Search	5
4 Secure File Transfer	6
5 File Replication	6
6 Election Algorithm	7
7 Implementation	7
7.1 List of API Endpoints	7
8 Testing	7
8.1 Health Check	7
8.2 File Search	8
8.3 Node Replication	8
8.4 Leader Election	8
8.5 Web Admin (Front End)	8
9 Future Work	9
10 Conclusion	9

1 Abstract

Peer-to-peer (P2P) file discovery services have become an important tool for sharing files among users on the network. These services allow users to search for and download files directly from other users' computers, rather than relying on a central server. The primary use case considered for the development of this project is the secure transfer of files from one medical facility to another when a cloud service is unavailable. This project presents a P2P file discovery service, including its architecture, key features, and challenges. We also discuss the mechanisms used for indexing and searching files, as well as the methods used for maintaining data integrity and preventing malicious activities. Finally, we highlight some of the challenges facing P2P file discovery services, including scalability, and security.

2 Architecture

The services consist of a collection of nodes that are placed in one of two categories - primary and data.

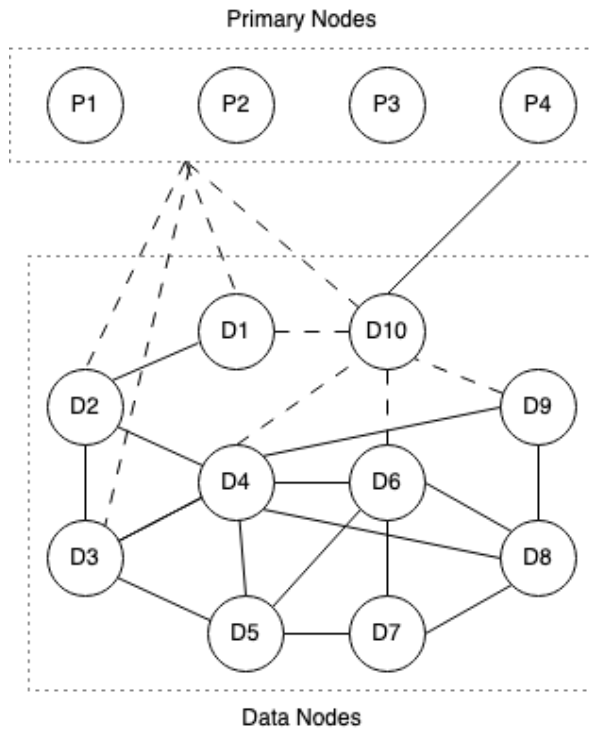


Figure 1: Architecture Overview

2.1 Primary Node

The primary nodes layer serves as a collection of high-availability nodes with specific research-oriented objectives. Its primary function is to act as the point of entry for the P2P network, with all participating data nodes mandated to register with a primary node.

In addition to its entry-point role, the primary nodes layer is tasked with maintaining a comprehensive and up-to-date list of all data nodes present in the network. This list is periodically updated and utilized by data nodes to ensure their network knowledge remains current.

To ensure the overall health and stability of the P2P network, the primary nodes layer also conducts routine health checks on all known data nodes.

2.2 Data Node

The data nodes are the primary storage units of the network, and they contain the data that is shared by the users. The data is heterogeneous and is not structured in any form. Each data node maintains a copy of the list of all other data nodes present in the network, which is retrieved from the primary node periodically. Data nodes can be thought of as analogous to user nodes, where the data represents the files that the user is willing to share with the network. To find data present in other nodes, the data nodes initiate a file search to all other data nodes in the network and wait for the response from all data nodes. More details on the search algorithm are provided in section 3. Data nodes also perform trigger-based replication.

The data nodes are responsible for storing the data that is shared by the users, and they also provide a mechanism for users to find and access the data that they are looking for. Each node's data is periodically refreshed and hashes for each file are computed and stored.

The data nodes are a critical component of the P2P network, and they provide a number of essential services that ensure the network's reliability and security.

2.3 Node Discovery

Every data node that wants to participate in the network must register itself with the primary node. The registration process is done by sending

a request to `POST /discover` endpoint, with the following parameters:

- IP address
- Name
- Public key
- Coordinate (longitude, latitude)

The primary node will then add the data node to its list of registered nodes and return a success message along with a node number that is assigned to the node. The node number is used for leader election algorithm described in section 6. Below API template for registering with the primary node.

```
1 POST /discover HTTP/1.1
2 Content-Type: application/json
3 Host: <Primary Node Addr>
4 Content-Length: 86
5
6 {
7     "ip": "<Registering Node Addr>",
8     "public_key": "<RSA Public Key>",
9     "name": "Einstien's Mug"
10    "coordinate": [longitude, latitude]
11 }
```

The primary node also exposes a GET endpoint that can be used to retrieve a list of all registered nodes, along with their health statuses. This endpoint can be useful for monitoring the health of the network and for troubleshooting problems.

2.4 Node Health Check

The primary nodes perform health checks on all known data nodes at regular intervals and update the node's health status. The health checks are performed by sending a `GET /health` endpoint for each node. If the node responds with a 200 response, it is marked as healthy. If the node responds with an error, it is marked as unhealthy.

Health checks are important for ensuring that the network is reliable and secure. By identifying unhealthy nodes, the data nodes can avoid communicating with the unhealthy nodes.

The health checks are also useful for monitoring the health of the network. By tracking the number of healthy and unhealthy nodes, the primary nodes can identify trends and patterns that may indicate problems.

2.5 Node Communication

The messaging interface provides a way for data nodes to communicate with each other. The messaging interface is used for a variety of purposes, including:

- General communication: Nodes can use the messaging interface to communicate with each other for general purposes.
- File search: Nodes can use the messaging interface to request other nodes to search for a file and download from them.
- Replication: Nodes can use the messaging interface to communicate replicate a node's data to another node.
- Election: Nodes can use the messaging interface to participate in the election of a new primary.

The interface follows a simple structure as given below:

```
1 {
2     "content": "<message content>"
3     "sender": "<originator IP addr>",
4     "type": "TEXT",
5     "is_encrypted": <boolean>,
6     "received_at": <UNIX timestamp>
7 }
```

The `content` attribute is data agnostic - can contain data of type `string` or `number` or `JSON` object.

The `type` attribute indicates the type for the message content. They can be one of the following types:

- `TEXT`
- `FILE_SEARCH_REQUEST`
- `FILE_SEARCH_RESPONSE`

The messaging interface is a critical component of the P2P network. It allows nodes to communicate with each other and to cooperate in a variety of tasks.

3 File Discovery

The primary contribution of this project is the file discovery algorithm. The following section demonstrates the file search algorithm:

3.1 File Search

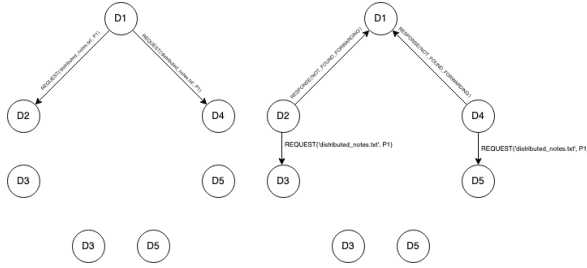


Figure 2: File Search Part 1

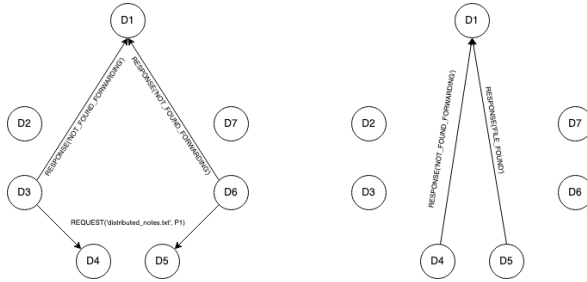


Figure 3: File Search Part 2

The file search algorithm is an asynchronous algorithm that checks each data node's files to determine if the file is present in the network. The algorithm begins by the requester sending a request with the file name to be searched for. The search algorithm executes the following steps:

1. The search algorithm generates a search ID, a UUID in this case, and computes the SHA-256 hash of the file name.
2. The search algorithm searches its local files to identify if the requested file is present.
3. If the file is present, the search algorithm marks the response for its IP address as true.
4. If the file is not present, the search algorithm marks the response for its IP address as false and then forwards the search to n neighbor nodes.

Each neighbor that receives the search request executes the following steps:

1. The neighbor searches its local files for the existence of the requested file.
2. If the file is found, the neighbor sends a message to the requester marking the file is available at its node.

3. If the file is not found, the neighbor forwards the request to another n neighbor node that haven't been forwarded before and sends the requester a message stating it doesn't have the file.

The algorithm eventually completes the search and every node would've been checked. Since the search requires searching through all the nodes, based on the size of the network the amount of time taken to complete the search can increase with the increase in the number of nodes.

The file search algorithm is an efficient way to find files in a P2P network. The algorithm is asynchronous, which means that it can continue to search for files even if some nodes are unavailable. The algorithm is also scalable, which means that it can handle a large number of nodes and files.

The requester node maintains the following structure for the file search request:

```

1 {
2   "file_hash": "f47a71f257033c0d18275f",
3   "file_name": "2b.png",
4   "requested_at": 1679611350272690000,
5   "requestor": "127.0.0.1:3000",
6   "responses": {
7     "127.0.0.1:3000": false,
8     "127.0.0.1:3002": true,
9     "127.0.0.1:3003": false
10  },
11   "search_id": "b06b8017-3b41-446a-8178"
12 }

```

- **search_id**: Unique UUID associated with the search. Each search is given unique search ID and it has no correlation with file being searched.
- **file_hash**: SHA-256 hash of the requested file name
- **file_name**: Name of the file to be searched
- **requested_at**: UNIX timestamp of the requested time in nanoseconds
- **requestor**: IP address of the node requesting the data
- **responses**: A map of node IP address and search result at the node. The search has three possible values:
 - **null** (if search is requested)
 - **False** (if node does not own a copy the file)
 - **True** (if node owns a copy of the file)

4 Secure File Transfer

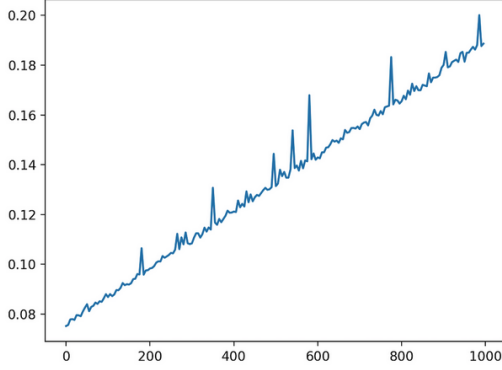


Figure 4: TGDH Key Generation vs Node Count

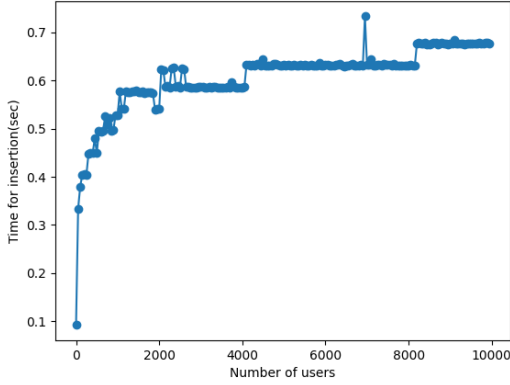


Figure 5: Node addition vs Time Graph

We are using Tree Based Diffie Hellman as group encryption to secure file transfer between data nodes. We implemented a custom Tree data structure to group secret key agreement. We have other options like OGH etc but based on results reported in this project we opted for TGDH.

Key Features of the TGDH protocol include: 1. Group Key Security, 2. Forward Security, 3. Backward Security, 4. Private Key Independence. TGDH architecture constitutes of a server, multiple clients. AES is symmetric encryption algorithm used by all parties once key is calculated by all parties. Also Tree data structure is used to represent the clients as leaf nodes and each have a corresponding pair of (private key, public key).

The strength of the protocol is based on the

difficulty of calculating discrete logarithms in finite fields, therefore a potential attacker who intercepts a message sent via a public channel is not able to recover from it the private key that is local to the user. The shared secret for a pair of leaves is computed in each leaf, therefore no secure channel or trusted third party is required. The TGDH protocol extends the capabilities of the Diffie-Hellman protocol and allows more users to share the secret.

Re-keying based on message count: The communication between them is refreshed every fixed number of sent messages and counted again, this operation is invisible from the point of view of an ordinary user and does not hinder potential communication during the calculation of a new key. This operation keeps the group key fresh and prevents man in the middle attack.

5 File Replication

Distributed file replication is a technique that allows multiple copies of a file to be stored on different computers. This can be useful for improving reliability and availability, as well as for making files available to users on different networks.

The service uses a trigger-based replication strategy, where an event triggers the replication of all the data of the node to other nodes. To make replication more efficient, a subset of nodes are selected for storing the replicated data from the collection of all data nodes. The replication policy followed is based on the largest distance from source node, where n nodes are selected that are located globally the farthest from the replicating node.

When replication is triggered, the node to be replicated requests the primary node to give a list of nodes where the replicated data has to be placed. When the primary node receives the replication request from a node, it computes the distance of all other nodes from the replicating node using the Haversine formula given below:

$$2 * r \sin^{-1} \sqrt{\sin^2 \frac{\phi_2 - \phi_1}{2} + \cos \phi_1 \cos \phi_2 \sin^2 \frac{\lambda_2 - \lambda_1}{2}}$$

The primary then returns the list of n nodes that have the largest distance from the replicating node. The replicating node communicates with the selected nodes to start replicating data and

additionally shares a list of files to be replicated by the destination node. Currently all files are replicated to all the selected nodes. However, a redundant replication mechanism could be employed to allow replicate a subset of the files to particular nodes such that there are m copies of the files available on the network.

6 Election Algorithm

The major assumption of the project has been that primary nodes are highly available. However, this architecture makes the primary nodes analogous to a centralized collection of nodes, and the network will fail to work if the primary nodes become unavailable. To be more resilient to such a scenario, we attempted to employ a leader election algorithm where the primary node is elected.

We chose to implement a version of the bully algorithm for leader election and below are the steps taken by the data nodes:

1. When a data node registers with the primary node, the primary node returns an unique node value v that is chosen in random within the range $[0, 65535]$
2. When data node detects the primary node is unavailable
3. Data node compares it's value v with all other nodes's value v
4. If there exists another node with higher value then it skips election
5. If it has the highest value v , then it assigns itself as the primary node and updates the primary node address.

During the transition phase it is likely that services dependent on the primary node will be blocked until the primary node is established.

7 Implementation

The service was developed with Python. The following libraries have been used to implement the server and other features:

- Flask - Back-end server framework
- Cryptography - Pyca Hazmat library is the recommended cryptography python library
- Redis - External storage for each node. It can replaced any other storage mechanism

- React - Front-end web UI framework

7.1 List of API Endpoints

- Node Discovery
 - GET /discovery
 - POST /discovery
 - POST /discovery/<ip_addr>
- Message Communication
 - GET /communicate
 - POST /communicate
- File Discovery
 - GET /files/
 - POST /files/search/
 - GET /files/search/<search_id>
- File Transfer
 - POST /files/download
 - POST /files/secure/download
 - POST /files/secure/download.to_node
- Node Replication
 - GET /replication/initiate
 - POST /replication/select_nodes
 - POST /replication/request_replication
- Node Health
 - GET /health

8 Testing

A P2P network was simulated with a primary node and four data nodes. Each node was run on a different port and given a different Redis namespace.

8.1 Health Check

The primary node was configured to check the health status of the data nodes at intervals of 15 seconds. The health check log received by the primary node is shown in figure 8

```
2023-03-24 00:50:49.279:279 Thread-8 (execute_health_check) INFO [node_health_handler.py:33] Node 127.0.0.1:3000 health check status: True
2023-03-24 00:50:49.279:279 Thread-7 (execute_health_check) INFO [node_health_handler.py:33] Node 127.0.0.1:3001 health check status: True
2023-03-24 00:50:49.280:280 Thread-10 (execute_health_check) INFO [node_health_handler.py:33] Node 127.0.0.1:3002 health check status: True
2023-03-24 00:50:49.280:280 Thread-9 (execute_health_check) INFO [node_health_handler.py:33] Node 127.0.0.1:3003 health check status: True
```

Figure 6: Primary node health check logs

8.2 File Search

Each data node randomly selects two node as it's neighbor such that each neighbor node is only neighbor for two other data nodes. This property ensures that all nodes are neighbors to maximum of two nodes.

The search was performed from **Node 1** for a file that exists in **Node 3**, following is the flow of traffic was observed:

1. **Node 1** search files with it's data collection
2. File not found in **Node 1**, forwarding request to **Node 4** and **Node 3**
3. **Node 4** receives search request and searches files with it's data collection
4. File not found in **Node 4**, forwarding request to **Node 3** and **Node 2**. **Node 4** sends file not found status to **Node 1**
5. **Node 3** receives search request and searches files with it's data collection
6. File is found in **Node 3**, it proceeds to sends file found status to **Node 1**
7. **Node 2** receives search request and searches files with it's data collection
8. File not found in **Node 2**, forwarding request to **Node 3** and **Node 4**. **Node 2** sends file not found status to **Node 1**
9. **Node 3** and **Node 3** drop the requests as they have already processed the request
10. **Node 1** processes all responses and stores search statuses

8.3 Node Replication

For local experiment, each data node is given fixed latitude and longitude coordinate representing where the server is located. **Node 1** is requested to initiate replication, below are the steps observed:

1. **Node 1** requests primary node to return list of nodes to send the replicated data
2. Primary node return the farthest two nodes from **Node 1** which are **Node 3** and **Node 4**
3. **Node 1** informs **Node 3** and **Node 4** to replicate all data from **Node 1**
4. **Node 3** and **Node 4** replicate all data from **Node 1** and store it in their storage.

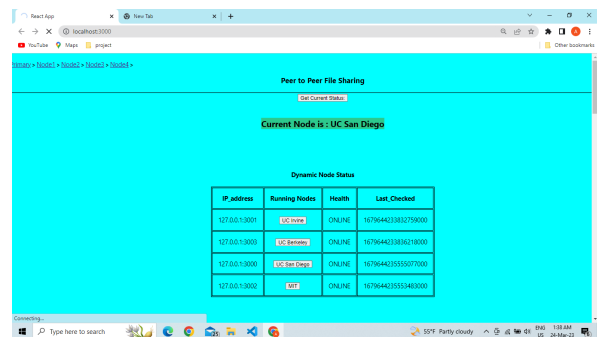
8.4 Leader Election

The primary node service is killed, following are the steps observed

1. **Node 1** detects primary node is unavailable, **Node 1** was assigned the value 2165, but there exists a node with a higher node value. So **Node 1** returns back normal execution
2. **Node 3** detects primary node is unavailable, **Node 3** was assigned the value 25757, but there exists a node with a higher node value. So **Node 2** returns back normal execution
3. **Node 3** detects primary node is unavailable, **Node 3** was assigned the value 59353. There exists no nodes with higher value, hence assigns itself as the primary node and elects itself as the primary node. Updates the primary node address, stops the child process for periodic node discovery and starts the primary node health checker.
4. **Node 4** detects primary node is unavailable, **Node 4** was assigned the value 21042, but there exists a node with a higher node value. So **Node 4** returns back normal execution
5. **Node 1**, **Node 2**, and **Node 4** obtain the latest primary node value from Redis and contain functions as usual.

8.5 Web Admin (Front End)

Below are screenshots of the web frontend developed with React framework



The screenshot shows a web application titled "Peer to Peer File Sharing" with a "Get Current Status" button. Below the button, it states "Current Node is: UC San Diego". A table titled "Dynamic Node Status" displays the following data:

IP address	Running Nodes	Health	Last Checked
127.0.0.1:3001	UC Server	ONLINE	16764443380279905
127.0.0.1:3003	UC Server	ONLINE	16764443380278800
127.0.0.1:3000	UC San Diego	ONLINE	16764443355507700
127.0.0.1:3005	UC Server	ONLINE	16764443355548000

Figure 7: Node List

