File Management System

Group 23: Mohamed Youssef, Selim Elbindary, and Ibrahim Maher

University of Stuttgart

This report outlines the implementation of a Distributed File Management System using Python. The system delivers scalable and fault-tolerant file management, including leader election, dynamic host discovery, replication management, reliable ordered multicast and fault tolerance.

1 Architecture Model

This system is structured using a client-server framework, where the client components initiate various file-related operations and interact directly with the leader, who in turn interacts with server components dedicated to the storage and management of these files. This architecture ensures a well-defined separation of roles and responsibilities: on one side, client components are primarily focused on facilitating user interactions, enabling users to navigate and perform actions within the system seamlessly. On the other side, server components are meticulously designed to handle the complexities of storing, managing, and coordinating file operations, ensuring that data remains consistent and accessible.

- **Client Nodes:** These nodes act as the primary interface for the system's users. They are designed to be user-friendly, allowing individuals to interact with the system's functionalities efficiently. Through these client nodes, users can execute a range of file operations, request data, and receive the information they need, all while the underlying complexities of data management are abstracted away.
- **Server Nodes:** These nodes bear the responsibility for the storage and meticulous management of files within the system. They are the backbone of the system, responding to requests initiated by client nodes. Beyond merely storing data, server nodes are equipped with sophisticated mechanisms to coordinate the execution of file operations, ensuring that every action taken by a user is reflected accurately across the system. They play a crucial role in maintaining the integrity and consistency of data, employing algorithms and protocols to manage concurrent operations and resolve potential conflicts, thereby ensuring that the system remains reliable and efficient for all users. The server nodes are coordinated and managed by the leader, with whom the client interacts directly.

2 Dynamic Discovery of Hosts

2.1 Overview

Discovery Mechanism: A discovery method is implemented to facilitate dynamic discovery of the leader. Clients broadcast a message to the servers requesting the identity of the leader and the leader responds to the client. This allows the client to discover the leader before any operation, enabling efficient up-to-date host discovery.
Server Registration: Server nodes register with the leader upon initialization and de-register upon shutdown. This ensures that the leader maintains an accurate and real-time list of available servers.

2.2 Dynamic Discovery Mechanism

- **Broadcasting Presence:** New servers announce their presence to the entire network by broadcasting a message. The broadcast_presence function creates a UDP socket, sets it to broadcast mode, and sends a message containing the new node's rank, IP address, and port to a designated broadcast port. This allows the new server to introduce itself to the entire network without needing to know the addresses of all other servers.
- **Leader Acknowledgment:** After broadcasting its presence, the new server starts waiting for an acknowledgment from the current leader node (if any) within the network. This is implemented in the wait_for_leader_ack method, where the server waits for a predefined timeout period for a response. If the leader acknowledges the new node, it can start participating in the network's activities as a recognized node.
- **Election Process:** If the new node does not receive an acknowledgment from a leader within the timeout period, it checks for the presence of higher-ranked nodes. If no such nodes are present or if it does not receive a response, the new node declares itself as the leader. This is a simplified version of leader election, where the assumption might be that nodes with higher ranks have priority for leadership, or in the absence of a leader, a new node can take on this role.
- **Client:** The dynamic discovery process on the client side involves broadcasting the client's presence across the network using the `broadcast_presence` method. This method utilizes a UDP socket to send a message containing the client's IP address, and port. The client then enters a waiting state, managed by the `wait_for_leader_ack` method, anticipating an acknowledgment from the current leader. If an acknowledgment is received within the specified timeout, the client recognizes the leader and seamlessly integrates itself into the distributed system. In the absence of an acknowledgment, the client may infer a potential leader failure or absence, triggering a new leader election process. This dynamic discovery mechanism ensures efficient and up-to-date host discovery before initiating any file operations.

2.3 Leader Handling

- **Processing New Node Broadcasts:** The leader node listens for broadcasts from new nodes. Upon receiving a NEW_NODE message, the leader checks if the new node's rank is already known. If not, it initializes the new server in its system, updates its list of servers (servers), and acknowledges the new node. This ensures that the new server is integrated into the distributed system and can start receiving tasks or data for replication.

3 Replication

In the distributed file management system the leader server has a replication manager. This replication manager is used in two situations, either when a new server is admitted to the system or when a file is written by a child server and needs to be replicated to all other storage servers.

- **Initial Server Admission and Data Replication:**
  - **Server Admission:** Upon the introduction of a new server to the network, it must be integrated seamlessly to participate in the system's operations. This integration involves synchronizing the data of its file manager with that of the existing network.
  - **Leader's Role in Replication:** The leader server, acting as the central authority or **replication manager**, takes charge of initiating the data replication process. This ensures the new server is equipped with all necessary files and data to function effectively within the system.
  - **Replication Process:** The replication involves transferring data from the leaders central file manager to the new server. This is done to ensure the new server holds an up-to-date copy of all files and data it needs to participate in the network's operations fully.
- **Replication Management by the Leader:**
  - **Tracking Critical Operations:** The leader server meticulously monitors and records every critical operation that affects data, such as creation, modification, deletion, or editing of files. This tracking is crucial for ensuring that all changes are propagated throughout the network.
  - **Operation Propagation:** Following a critical operation, the leader processes the data change and distributes the necessary updates to all child servers. This distribution is targeted to all child servers, ensuring they receive the latest version of this file.
  - **Mechanism for Update:** The replication of data is conducted using the replicate method in the replication manager. This method loops on all child servers, delivering to them the replicated file.

The process ensures that every server, upon and after joining the network, quickly reaches a state of data parity with existing servers, thanks to the leader's role in replication management. This comprehensive approach to tracking and replicating critical operations, under the oversight of the leader, guarantees that all servers maintain consistent and up-to-date data. This synchronization is vital for the distributed system's overall performance, reliability, and integrity, enabling it to function seamlessly as a cohesive unit.

4 Leader Election (Voting)

The voting algorithm used to elect a leader is the bully algorithm. The algorithm selects the node with the highest ID as the leader. It's triggered when a node detects the absence of a leader or at system start. Nodes with higher IDs can initiate their own elections, ensuring robust and autonomous leader selection. This algorithm promises enhanced efficiency and system reliability. The leader is chosen to handle the authoritative and coordination tasks.

4.1 Bully Algorithm

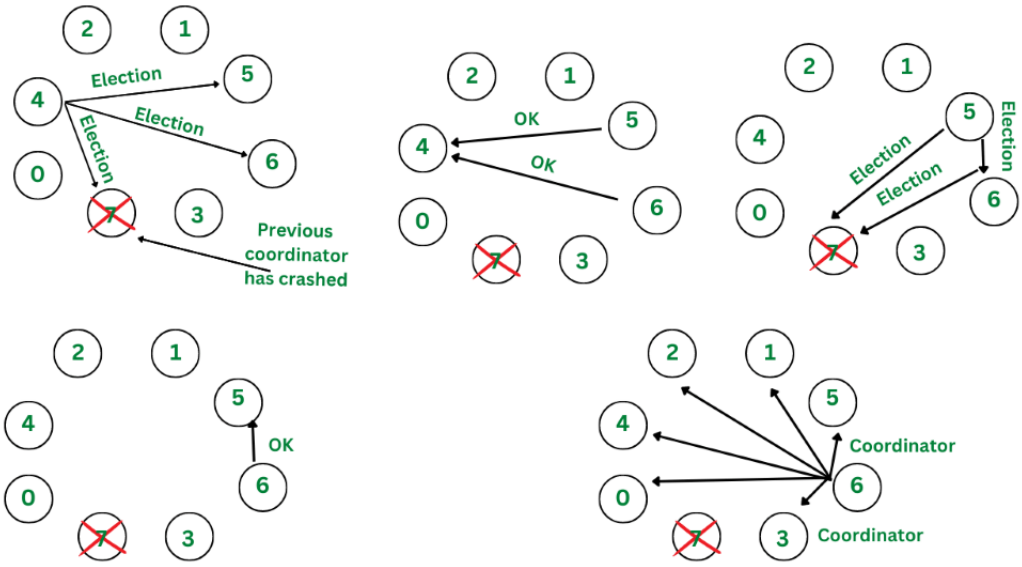
- **Election Trigger:** An election can be triggered under various circumstances, such as when a node detects that the leader has failed, or a node believes it has a higher rank than the current leader, or when a new node joins the system and doesn't recognize a leader.
- **Election Process:** Upon initiating an election, a node sends an election message to all nodes with higher ranks. If it receives no response (indicating no active higher-ranked nodes), it declares itself the leader.
- **Victory Declaration:** The node that determines itself as the leader (either because it's the highest-ranked or received no objections) broadcasts a victory message to all other nodes, establishing itself as the leader.

4.2 Implementation Details

- **Handling Victory Broadcasts:** The handle_broadcast_message method listens for victory announcements. If a node with a higher rank announces itself as the leader, the current node acknowledges this new leader. Conversely, if the announcement comes from a lower-ranked node, the current node initiates a new election process, assuming it has not recognized the leader or there's a discrepancy in the leader's status.
- **Election Initiation:** The handle_incoming function triggers the election process upon receiving an election message, ensuring that the system can dynamically respond to changes in leadership status or node availability.
- **Starting an Election:** The start_election method is where the core of the election logic resides. The node sends election messages to all higher-ranked nodes (determined by comparing ranks). If there are no higher-ranked nodes or no responses are received within a certain timeout, the node declares itself the winner.
- **Victory Declaration:** The declare_victory function broadcasts the victory of the node across the network. This broadcast informs all nodes (including lower and higher-ranked ones) of the new leader, ensuring the system's coherence and operational integrity. This is done both through direct messages to known nodes and a network-wide broadcast, ensuring that all nodes, regardless of their current state, recognize the new leader.

4.3 Bully Algorithm Characteristics in the Code

- **Decentralization:** The algorithm operates in a decentralized manner, with each node capable of initiating an election and declaring victory based on the algorithm's rules, without needing a central authority.
- **Scalability Considerations:** In large-scale systems, the number of messages required for a single election can become a bottleneck. The implementation may need optimizations or adaptations, such as limiting the frequency of elections or using more sophisticated message passing strategies to mitigate this. Type speed visualisation



5 Synchronization

Semaphore in a distributed system is a synchronization mechanism used to control access to a shared resource by multiple processes or threads in a concurrent environment. Unlike in a single-processor system, where semaphores can directly control access to shared resources, distributed systems require semaphores to work across different machines or processes that do not share memory space. This adds complexity due to network communication and the need to maintain consistency across distributed nodes.

- **Distributed Mutual Exclusion:** Semaphore ensures that only one process at a time can access a critical section or a shared resource, such as a database or file system, to prevent data inconsistency or corruption.
- **Implementation:** In the distributed file management system binary semaphores are used. They ensure that only one critical operation can occur per file at a single moment. This allows multiple users to write, edit, create and delete the same file without causing any consistency issues. The system blocks one client while executing the other, before allowing it to proceed.

6 Fault Tolerance

Fault tolerance is integrated in the system's design, ensuring resilience against disruptions. To monitor for possible failures a heartbeat mechanism is used to detect if a leader fails. In case of leader failure, the bully algorithm handles the election of a new leader. To detect a server failure, when a leader sends a request they await for acknowledgment and message response. In case of server failure, data is retrieved from a replica, ensuring continuous availability.

6.1 Heartbeat Monitoring

- **Implementation:** Each server in the system, whether a leader or a child, implements a method named heartbeat that runs in a separate thread. This method is responsible for sending and responding to heartbeat signals, ensuring that it does not interfere with the server's main operations.
- **Mutual Supervision:** In this system there is a mutual monitoring responsibility. This dual-role mechanism enhances the fault tolerance of the system, allowing for prompt detection and response to server failures.
- **Child Servers Check by Leader:**
  - **Frequency:** The leader server initiates a heartbeat check on all child servers every 10 seconds, utilizing a TCP connection for reliability. This interval is chosen to balance between timely failure detection and minimizing network overhead.
  - **Procedure:** When a child server receives a heartbeat signal from the leader, it is expected to respond promptly, indicating its active status.

  - **Failure Handling:** If a child server fails to respond to the heartbeat signal, the leader server attempts to re-establish connection for a predefined number of maximum retries, each after a specific delay. This retry mechanism accounts for temporary network issues or short-lived server problems.
  - **Server Removal:** If the child server remains unresponsive after all retry attempts, the leader concludes that the server is unavailable. It then proceeds to remove this server from its list of active servers, reallocating tasks as necessary to maintain system functionality. The leader also informs all other servers of this child servers removal in order to ensure all servers have a correct view of their environment.
- **Leader Check by Children Servers:**
  - **Frequency:** Similarly, each child server initiates a heartbeat check of the leader server every 10 seconds using a TCP connection. This ensures that the leader's availability is continuously verified.
  - **Procedure:** The child server expects a prompt response to its heartbeat signal, confirming the leader's operational status.
  - **Failure Handling:** If the leader does not respond, the child server retries the connection a set number of times after specific delays, mirroring the leader's approach to handling unresponsive child servers.
  - **Leadership Election:** Should the leader server fail to respond after all retry attempts, the child server assumes the leader is no longer functioning. It then initiates a leadership election process among the available servers to select a new leader. This process is crucial for restoring the system's operational hierarchy and ensuring continuous system operation without central guidance.

6.2 Rollback

The rollback mechanism is vital in the file management system where data consistency is critical. It helps in managing data replication across servers, ensuring that all nodes in the network have the correct and latest data. This is particularly important to our system as the consistency of the files stored is essential.

- **Purpose:** The primary goal here is to ensure that the child server's data is complete and up-to-date. By sending the missing files, the leader server helps the child server restore its data integrity and maintain consistency across the network.
- **State:** The state of any of the server is represented by their critical_operations dictionary, where the critical operations are stored in order of execution.
- **Start:** The rollback method is found in the replication manager. It is initiated after a leader election. At this point the leader does a rollback check on each of its child servers to ensure they are consistent with its state. If they are not consistent then they are made consistent. After the rollback to a server is completed they are added to the operations queue so that they can be used by the leader.

- **Leader:** The leader server, acting as the central or authoritative source of data, identifies these missing files on the child server. It then initiates a process to send the missing files to the child server.
- **Implementation:** In the rollback method, the leader checks the critical state of each child server. If the leader detects any inconsistencies between its critical operations and those of the child server then these must be rectified.
  - **Servers with missing data:** If a child server is missing any files, then the leader sends these files to the child so it can update or remove them if they had been deleted.
  - **Servers with more relevant data:** If a child server has any extra files/data, then the current files at the leader are sent to the child so it can update or remove them if they had been deleted.
- **Result:** After the completes the rollback on a child, this server is considered consistent with the leader and is added to the list of available children to be used.

7 Limitations

1. **Scalability Concerns:** The system may face challenges in scalability, particularly with a growing number of nodes. As the network expands, the frequency of heartbeat checks, leader elections, and replication management may lead to increased network overhead.
2. **Network Overhead:** The implementation of the bully algorithm for leader election and the frequent heartbeat checks might introduce additional network overhead. In large-scale systems, optimizing communication strategies and reducing unnecessary messages could be a consideration.
3. **Rollback Efficiency:** The rollback mechanism ensures consistency but might become resource-intensive, especially when dealing with a large number of files or servers. Optimizations may be required to enhance the efficiency of the rollback process.
4. **Fault Detection Sensitivity:** The system heavily relies on heartbeat mechanisms for fault detection. In scenarios with intermittent network issues, false positives or negatives in fault detection could occur, impacting the accuracy of failure management.
5. **Leader Bottleneck:** The single-leader model might pose a bottleneck in highly dynamic or fault-prone environments. Exploring distributed leadership models or dynamic leadership assignment could address this limitation.

8 Conclusion

The implemented File Management Distributed System demonstrates a robust architecture with features like dynamic host discovery, leader election, replication management, reliable ordered operations, and fault tolerance. The system's design emphasizes data consistency, fault resilience, and efficient operation. While

the system showcases significant capabilities, it is essential to acknowledge the identified limitations.

In conclusion, the File Management System provides a foundation for a scalable and fault-tolerant distributed file system. Future improvements could focus on addressing scalability challenges, optimizing network communication, and refining mechanisms such as rollback for enhanced efficiency. The system's ability to dynamically adapt to changes, elect leaders, and maintain data consistency positions it as a reliable solution for distributed file management.