CS 485 PA3

UIC Fall 2024

Design Document

Mira Sweis – ECE MS

1. Introduction

This project implements a hierarchical Gnutella-style peer-to-peer (P2P) file-sharing system that supports both push-based and pull-based consistency mechanisms for maintaining file validity across nodes. The system simulates two network topologies and enables testing of the trade-offs between these consistency methods:

All-to-All Topology: Each super-peer is connected to every other super-peer.

Linear Topology: Super-peers are connected in a chain-like structure.

The project utilizes Python to create a distributed system where leaf nodes communicate through super-peers to query, share, and validate files. File consistency is ensured by propagating invalidation messages in push mode or periodic polling in pull mode.

2. Program Design

2.1 Overall Architecture

The system consists of the following components:

Leaf Nodes: Clients that register their files with connected super-peers, query for files, and validate file consistency using push or pull mechanisms.

Super-Peers: Intermediaries between leaf nodes and the larger network, managing file registries and forwarding queries to neighboring peers. Super-peers facilitate consistency by querying origin nodes for file versions or propagating invalidation messages.

Network Setup: Configures the network statically using JSON files (linear.json or all_to_all.json) to define connections between super-peers and leaf nodes.

File class: to me this was an easier way than having two directories. I would use a file class to store information regarding the files saved a t leaf nodes.

2.2 Modules

leaf_node.py: Manages file registration, queries, and file transfers. Implements push or pull mechanisms for consistency, depending on the mode.

super_peer.py: Handles file queries, consistency checks, and communication with neighboring super-peers and leaf nodes.

network_setup.py: Initializes the network by reading configuration files and setting up super-peers and leaf nodes**.**

file.py: initialized files at leaf node startup to carry information regarding the file, such as if it a copy, if it is valid, name, and original leaf node.

2.3 Main Functions in leaf_node.py and super_peer.py

Functions in leaf_node.py

register_files()

- Registers the files available in the leaf node with its connected super-peer.
- Sends a list of file names to the super-peer, which indexes them for query handling.

query_file(file_name, TTL=17)

- Allows the leaf node to search for a file in the P2P network.
- Sends a query to its connected super-peer, which checks locally and forwards the query to neighboring super-peers if necessary.
- If a query hit is found, the leaf node can choose a node to retrieve the file from.

retrieve_file(leaf_node_id, file_name)

- Downloads a file from another leaf node once the file query has returned a hit.
- Establishes a socket connection to the target leaf node and downloads the file in chunks, saving it locally.

push(filename, version)

- Broadcasts an invalidation message when a file is updated in the leaf node (used in push mode).
- Sends invalidation messages to the connected super-peer, which propagates them to other peers and leaf nodes.

poll_for_updates(ttr=30)

- Periodically queries the super-peer to check if cached files in the leaf node are still valid (used in pull mode).
- If a file is stale, the leaf node invalidates it and prompts the user to decide whether to re-download the updated version.

handle_connection(client_socket, address)

- Processes incoming messages from super-peers or other nodes.

- Handles requests for file transfers, invalidations, and version queries.
- Supports VERSION_REQUEST messages to provide the current version of a file to the super-peer.

edit_file(name, input)

- Allows the user to edit a file in the leaf node.
- Updates the file's version and broadcasts an invalidation message in push mode.
- Ensures that only valid, original copies of the file can be edited.

handle_version_request(client_socket, request)

- Responds to VERSION_REQUEST messages from the super-peer by returning the current version of a file stored in the leaf node.

handle_invalidation(message)

- Processes invalidation messages from the super-peer to mark a file as invalid locally.
- Optionally removes the invalidated file from the directory.

Functions in super_peer.py

handle_query(query, response, back_prop)

- Processes file queries from connected leaf nodes or neighboring super-peers.
- Checks if the requested file exists in the super-peer's registry and generates a query hit if found.
- If the file is not found locally, propagates the query to neighboring super-peers, using a TTL to limit propagation.

propagate_query(query, response, back_prop)

- Forwards file queries to all neighboring super-peers unless the TTL has expired.
- Ensures that queries are not forwarded back to the origin peer.

handle_pull_request(message, client_socket)

- Processes pull requests from leaf nodes to verify the validity of cached files.
- Queries the origin leaf node for the current version of the file and compares it with the cached version.
- Responds with VALID if the cached version is up-to-date or STALE if the file is outdated.

handle_invalidation(message)

- Handles invalidation messages from origin leaf nodes in push mode.
- Notifies all connected leaf nodes to discard the invalid file and removes it from the super-peer's registry.
- Propagates the invalidation message to neighboring super-peers.

send_invalidation_to_leaf_node(leaf_node_id, message)

- Sends invalidation messages directly to leaf nodes connected to the super-peer.

propagate_invalidation(message)

- Forwards invalidation messages to neighboring super-peers in push mode.

handle_connection(client_socket, address)

- Processes incoming messages from leaf nodes or neighboring super-peers.
- Handles queries, pull requests, invalidations, and file registrations.

2.4 Design Considerations and Trade-offs

The system implements both push and pull consistency models, each with distinct trade-offs. The push model offers immediate consistency by broadcasting invalidation messages, ensuring low latency but increasing network traffic, especially in large-scale networks. In contrast, the pull model reduces unnecessary messages by having nodes periodically check file validity, though it risks stale data between polling intervals and requires careful tuning of polling frequency. Concurrency is handled using Python's threading module, enabling simultaneous processing of queries, invalidations, and requests. The time-to-live (TTL) mechanism prevents infinite query propagation, while the message_log ensures queries are not duplicated. Additionally, the choice of topology—all-to-all or linear—balances between fast query resolution with higher overhead and reduced network load at the cost of slower resolution, respectively.

3. Possible Improvements and Extensions

Key improvements to the system include optimizing query propagation through selective forwarding or caching popular queries to reduce network traffic. Dynamic peer discovery could allow nodes to join or leave the network seamlessly, improving flexibility and fault tolerance. Fault tolerance can also be enhanced by implementing backup super-peers or replicating file indices and files across multiple nodes to ensure availability during failures. Refining the pull model with lazy polling—where nodes only check file validity when accessed after its time-to-refresh (TTR)—can further reduce overhead. Finally, integrating performance monitoring tools to track metrics such as query latency and consistency costs could guide future optimizations and improve scalability.