P2P File-Sharing System in Golang - Design Document

Juan Pedrajas

Design Overview:

Architecture:

The P2P file-sharing system is designed as a decentralized network of nodes, each acting as a client and a server concurrently. Nodes communicate with an indexing server to share information about their files and presence in the network. Key components include:

1. **Peer Node:**
   - Combined client and server components.
   - Background thread for server functionality, monitoring the local filesystem and handling incoming requests.
   - Goroutines are utilized to manage concurrent tasks, ensuring efficient handling of multiple operations simultaneously.
   - All configuration is loaded from a config.json inside the same directory where it's being called from.
2. **Indexing Server:**
   - Centralized server responsible for maintaining a global view of the network.
   - Handles peer registrations, file updates, and queries from peer nodes.
   - Employs different muxes to manage concurrency, addressing simultaneous requests from multiple nodes.

Design of the indexing server

The core of the ingress server system is encapsulated in the primary file, 'indeingServer.go.' This file orchestrates the reception of connections and intelligently directs them to various threads. Concurrent management of connections is facilitated by the 'handleConnection' function, specializing in generic communication aspects. Subsequently, the 'process_command' function is invoked, triggering a tailored response from the server by calling a specific function. All connections adhere to RESTful principles, ensuring a standardized and efficient communication protocol.

Crucially, the indexing of both files and servers is meticulously maintained in a global map. This comprehensive map correlates each node's unique identifier with a corresponding node object, as detailed in the 'node.go' file. The 'node' object is not only a declaration but a structured entity equipped with specialized functions. These functions are purpose-built for the adept handling of nodes and their associated file data, contributing to the seamless operation of the server system.

Design of the peer node

The peer node represents an evolution of the previous project, consolidating both client and server functionalities into a unified executable. To achieve this, the client's code underwent a transformation into Go, necessitating modifications in the download and list files functions. Specifically, I enhanced these functions by incorporating queries to the indexing server, ensuring a seamless integration of client-server interactions.

A noteworthy improvement in the server's architecture involves the maintenance of a comprehensive list of all files in the monitored directory. This is achieved through a dedicated monitoring thread that systematically checks the file system every 0.1 seconds for any changes. Upon detecting a change, the thread promptly notifies the indexing server, contributing to real-time synchronization between the server and the monitored directory.

Furthermore, the server's interaction with the indexing server has been refined. It now actively registers and unregisters itself, even in the event of an unexpected termination. This enhanced robustness ensures that the indexing server is consistently updated with the status of the peer node, promoting a resilient and reliable system operation.

Communication Protocol:

The communication protocol between nodes and the indexing server is designed to be lightweight and efficient. Messages include:

1. **Peer Registration:**
    - Nodes inform the indexing server about their presence, IP address, port, and the initial list of files upon startup.
2. **File Update Notification:**
    - Nodes periodically notify the indexing server about changes in their files, ensuring the server's file registry is up to date.
3. **File Query:**
    - Nodes request information about peers possessing a specific file.
4. **File Transfer:**
    - Direct file transfer between nodes is initiated based on the information provided by the indexing server.
5. **File listing:**
    - Nodes can request a list of files in the network.

Tradeoffs:

1. **Simplicity vs. Flexibility:**
    - The design prioritizes simplicity for ease of understanding and rapid development. This decision may limit some advanced features that could be added in the future.
2. **Real-time vs. Periodic Updates:**
    - The choice of a 0.1s interval for monitoring the filesystem strikes a balance between real-time updates and reducing the overhead on the system. Fine-tuning this interval could be explored based on system requirements.

3. **Centralized Indexing Server:**
    - While centralization simplifies the design, it may introduce a single point of failure. Considerations for a distributed indexing approach or backup servers could be explored for enhanced fault tolerance.

Possible Improvements and Extensions:

1. **Distributed Indexes:**
    - In order to enhance the fault tolerance and eliminate the risk of a single point of failure, the indexing server can be designed to have slave nodes, which are replicas capable of seamlessly taking over in the event of a fault on the main indexing server. These slave nodes mirror the primary indexing server, keeping a synchronized copy of the index data. If the main server experiences a failure or becomes unavailable, one of the slave nodes can seamlessly step in to continue serving requests and maintaining the distributed index. This approach ensures continuous operation and resilience against potential disruptions, reinforcing the system's reliability in handling various failure scenarios. A list of indexing servers could be supplied to the nodes in the config.json
2. **Dynamic File System Monitoring:**
    - Implement an adaptive monitoring mechanism that adjusts the frequency of filesystem checks based on the file activity, reducing unnecessary updates during periods of inactivity. To implement an adaptive monitoring mechanism in Golang that adjusts the frequency of filesystem checks based on file activity, you can use a combination of a goroutine, channels, and a timer.
3. **Load Balancing:**
    - Explore load balancing strategies for distributing file download requests evenly among nodes possessing the requested file, optimizing network utilization. Insead of picking a random server that contains the file, have a set of rules that maximizes throughput in different use cases.

Conclusion:

In conclusion, the P2P file-sharing system presents a thoughtfully designed architecture that seamlessly integrates the roles of client and server nodes in a decentralized network. The decision to consolidate functionalities into a unified executable enhances deployment simplicity while leveraging the efficiency and concurrency features of Go.

The indexing server, at the core of the system, adeptly manages connections, orchestrates communication threads, and maintains a meticulous global map for effective indexing. This centralized approach, while simplifying design, prompts considerations for fault tolerance through distributed indexing or backup servers.

Trade-offs, such as prioritizing simplicity over advanced features and the periodicity of file system updates, were carefully weighed to strike a balance between functionality and ease of development. The communication protocol, designed to be lightweight, facilitates efficient interactions between nodes and the indexing server.

Looking forward, potential improvements and extensions, such as implementing distributed indexes for fault tolerance, adaptive file system monitoring, and load balancing strategies, offer avenues for enhancing the system's reliability and performance. These refinements, along with ongoing adaptation to evolving requirements, will be pivotal for the sustained success of the P2P file-sharing system. In essence, the design reflects a commitment to efficiency, scalability, and continual improvement.