Reflection Report

RPC and Message Passing

1. **Benefits of Combining RPC with Message Passing**:

   - **Separation of Concerns**: RPC is ideal for handling client requests and responses (e.g., adding, listing, or fetching paper details), while message passing handles asynchronous notifications (e.g., new paper announcements). This separation ensures clean modularity in the system.
   - **Asynchronous Communication**: Message passing allows real-time notifications without requiring the client to poll the server. This improves efficiency and user experience.
   - **Scalability**: Combining RPC and message passing enables the system to handle diverse use cases. RPC ensures synchronous interactions for direct client-server communication, while message queues handle distributed notifications.
2. **How Message Passing Enhances Scalability**:

   - **Decoupled Architecture**: Publishers (server) and subscribers (clients) do not depend on each other's state, allowing new subscribers to join without affecting existing components.
   - **Load Distribution**: Message queues act as intermediaries, buffering messages and enabling the system to handle spikes in load without overwhelming the server.
   - **Horizontal Scaling**: Multiple instances of the server or subscribers can be added to process messages in parallel, enhancing scalability.

Concurrency and Synchronization

1. **How Go Handles Concurrent Connections**:

    - **Goroutines**: Go's lightweight threads (goroutines) enable handling thousands of concurrent connections efficiently without significant overhead.
    - **Channels**: These provide a safe mechanism for communication between goroutines, simplifying concurrent programming.
    - **Net/RPC Package**: The built-in RPC library uses goroutines to manage multiple client requests concurrently, ensuring non-blocking operations.

2. **Importance of Synchronization in Message Passing**:

    - **Shared State**: If multiple goroutines modify shared data (e.g., the Papers map), race conditions can occur, leading to data corruption or inconsistent results. Synchronization primitives like mutexes ensure thread-safe access to shared resources.
    - **Message Order**: In systems with message passing, synchronization ensures that messages are processed in the correct order, maintaining system integrity.
    - **Data Consistency**: Proper synchronization prevents clients from receiving incomplete or stale data during concurrent updates.

Reliability and Fault Tolerance

1. **Impact of RabbitMQ Downtime**:

    - If RabbitMQ goes down, the server will fail to broadcast notifications, and clients won't receive updates about new papers.
    - While RPC-based functionality (e.g., adding or listing papers) would remain unaffected, the real-time notification feature would be disrupted.

2. **Making the Notification Service More Resilient**:

    - **Retry Mechanisms**: Implement retries for publishing messages in case of temporary RabbitMQ failures.

- **Durable Queues and Messages**: Configure RabbitMQ queues and exchanges as durable, ensuring messages persist even if the broker restarts.
- **Fallback Storage**: Use a local buffer to temporarily store notifications and publish them once RabbitMQ is back online.
- **High Availability Setup**: Deploy RabbitMQ in a clustered configuration to eliminate single points of failure.

File Storage in Memory

1. **Limitations of Storing Paper Content in Memory**:
    - **Scalability Issues**: As the number of papers increases, memory usage will grow, leading to potential exhaustion of system resources.
    - **Volatility**: Data stored in memory is lost if the server crashes or restarts.
    - **Concurrency Bottlenecks**: Frequent access to a shared in-memory store might lead to synchronization overhead.
2. **Modifying the Design for a Larger System**:
    - **Persistent Storage**: Use a database (e.g., PostgreSQL, MongoDB) for storing metadata and a file system (e.g., AWS S3) for storing paper content.
    - **Caching**: Implement a caching layer (e.g., Redis) to store frequently accessed data for faster retrieval.
    - **Sharding**: For large datasets, distribute storage across multiple servers or databases to balance the load.