Addis Ababa Institute of Technology

Reflection Report

Fundamentals of Distributed Systems

Nathan Mesfin Shiferaw
UGR/0534/14
Software Stream

Reflection Report

What are the benefits of combining RPC with message passing in this project?

- Using RPC, we have developed a way for a client to call a function on an entirely different machine. This enables us to execute synchronous and direct operations in a distributed environment. Message passing provides a method of sending asynchronous messages across the board. The usage of messages passing with RabbitMQ provides a persistent means of communication such that if a client is temporarily unavailable, the message can be stored until it comes back on. This makes the system fault-tolerant in cases of client failure. Combining RPC and message passing, we gain the advantage of allowing our application to be decomposed into services that can be separately deployed such that they communicate synchronously over RPC (for direct operations) and asynchronously using RabbitMQ (for notifications).

How does message passing enhance the scalability of the server?

- By delegating the task of notifying the clients to a publisher-subscriber message passing, the server is freed from the overhead of keeping track of all the clients and sending them notifications. Using message passing, the clients need only subscribe to the exchange channel used by the server to receive their notifications. The server needs only publishing a notification, and the clients can process the message asynchronously, making it easier for the server to handle multiple notification recipients and, hence, making it more scalable.

How does Go handle concurrent connections in the server?

- Go handles concurrent connections using goroutines, a lightweight alternative to traditional threads. Because of their lightweight nature, a lot of these goroutines can be running at any time. To ensure data consistency, a built-in synchronization primitive (*sync.Mutex*) is used. This makes sure that data is safely shared by goroutines that are trying to access the same piece of data.

Why is it important to handle synchronization when using message passing?

- Synchronized messages can be used when communication between components needs to be tightly coordinated. If message passing is used as a primary communication tool in a distributed system, the messages must be synchronized well primarily because it makes sure that the data is kept consistent and all the operations are executed in the correct order, enabling the system to perform and act as expected.

What would happen if the RabbitMQ service went down?

- Assuming it had not been configured to persist messages that are currently enqueued, it would lead to the loss of the messages/notifications that were meant to be sent. In our case, it would simply mean that clients won't be able to receive notifications when other clients add papers, and any pending notifications might be lost. But in a more critical system, like with work queues, it would mean that tasks would be lost, or they would arrive late for processing.

How could you make the notification service more resilient?

- One option would be to have mirrored queues and notification clusters so that the notification system can continue working even if a few of the RabbitMQ instances have failed. A simpler solution would be to enable persistent and durable messages so that even if RabbitMQ fails, the messages would endure the failure and would be ready for processing as soon as the service restarts. This can be supplemented with a method of detecting failure and temporarily storing notifications in the server so that they could be published to RabbitMQ as soon as the service returns. This will ensure that no notifications will be lost while the RabbitMQ service is down, hence making it more resilient.

What are the limitations of storing paper content in memory, and how would you modify this design for a larger system?

- The biggest limitations of storing the paper content in memory are its lack of scalability and lack of persistence. Large files could easily use up all the memory available, heavily restricting the amount of data that can be stored in the server and reducing its performance. Since memory is a volatile storage mechanism, a server restart would essentially cause the system to lose all the files. For larger files and more users, it would be better to use a distributed file storage system, like Amazon S3, to handle files while making sure they are available and fault tolerant. Additionally, files can be cached in memory to handle cases where certain files are accessed a lot of times.

How could the system be extended for more features, such as keyword search or paper downloads by multiple formats?

- To better support keyword search, the paper metadata should first be stored in a database system. The system can then support keyword searches by indexing the titles and authors. This can be used to quickly look up papers using metadata. Database operations can also be performed to support additional filtering operations.
- To support various paper formats, third-party APIs can be used to convert between the various paper formats. To minimize the calls to these third-party APIs, the papers can be

stored with all their variations. This can be implemented using a layer of caching where recent conversions are stored for one to two days.

- Additional information, such as the last view count and date of creation, can also be added to provide more information about the papers.
- Papers can also support update operations by implementing a method in the RPC server. It should also be able to update the existing papers, their metadata, and the underlying file itself. Storing the last update time can also help with keeping the metadata comprehensive and up to date.