1. Server.go

Certainly! The provided code is an example of a simple HTTP server written in Go that handles the uploading, updating, and deleting of files in chunks. I'll explain the code step by step:

1. Import Statements: The code starts by importing necessary packages for the program's functionality, including handling HTTP requests (http package), reading and writing files (os package), converting string to integers (strconv package), and managing concurrent operations (sync package).

ChunkInfo Struct: The ChunkInfo struct is defined to hold information about the uploaded file's chunks. It contains the total number of chunks and a map to store each chunk's data.

2. Global Variables: Two global variables are declared: chunkMap is a map that associates filenames with their corresponding ChunkInfo structures, and wg is a WaitGroup that will be used to manage concurrent operations.

handleUpload Function: This function handles the uploading of chunks. It extracts relevant information from HTTP headers, reads the chunk's content, and processes the chunk's data. It utilizes two goroutines (concurrent operations) to manage the chunk data and logging separately.

3. handleUpdate Function: This function simulates updating an assembled file. It reads the existing content, appends an empty byte slice to it, and writes the updated content back to the file.

4. handleDelete Function: This function simulates deleting an assembled file. It removes the file and clears the associated chunkMap entry.

5. writeToLogFile Function: This function appends a log line to a log file named after the filename and chunk number. It formats the log line with timestamp, IP address, request details, and response status.

6. assembleAndStore Function: This function assembles the stored chunks for a file, combines them, and writes the assembled content to a file. It also logs a success message when the process is completed.

7. WebServerLog Function: This function creates a log line for each request with relevant information, like IP address, timestamp, content details, and status.

8. main Function: The main function is where the execution of the program starts. It creates necessary directories, sets up the HTTP routes for upload, update, and delete operations, and starts an HTTPS server with TLS security on port 8443.

9. Parsing Chunk Information: The code you asked about parses the chunk number and total chunks information from the HTTP request headers. These headers are typically used when a large file is being uploaded in multiple chunks. The X-Chunk-Number header specifies the current chunk number being uploaded, and the X-Total-Chunks header specifies the total number of chunks in which the file will be uploaded. These values are then used to keep track of the chunks and their order.

Overall, this code sets up a basic HTTP server that handles file uploading in chunks, assembles them into files, updates existing files, and deletes files. It demonstrates how to manage concurrent operations using goroutines and sync mechanisms in Go.

2. Client.go

Certainly! This code demonstrates a client program in Go that splits files into chunks and sends them asynchronously to an HTTPS server for uploading. Let's break down the code step by step:

1. Import Statements: The code starts by importing the necessary packages for working with HTTP requests, file operations, strings, and synchronization.

2. sendChunkToServer Function: This function sends a chunk of data to the server using an HTTP POST request. It constructs the request with necessary headers such as Content-Type, X-Filename, X-Chunk-Number, and X-Total-Chunks. It then checks the response from the server and returns any errors encountered during the process.

2. splitFileAndSendChunksAsync Function: This function is responsible for splitting a file into chunks and sending them to the server asynchronously. It takes the client, file path, chunk size, and a result channel as input. It calculates the total number of chunks and then iterates over the file, reading chunks and sending them to the server using the sendChunkToServer function.

3. main Function: The main function is where the execution of the program starts. It sets the chunkSize to determine the size of each chunk.

filePaths Slice: A slice named filePaths contains the paths to the files that you want to split and upload.

4. HTTP Client Configuration: An HTTP client is configured with a custom http.Transport to enable communication over TLS. The TLSClientConfig is set with an empty list of certificates and InsecureSkipVerify set to true to bypass certificate validation. This is generally not recommended for production use as it disables security checks.

5.Concurrency using WaitGroup: The sync.WaitGroup named wg is created to manage concurrent goroutines. Each file processing operation will increment the wait group, and at the end of the loop, the wg.Wait() call ensures that the program doesn't exit until all processing is complete.

5. Loop over filePaths: The code then iterates over the filePaths slice and spawns a goroutine for each file.

6. Goroutine for File Processing: Inside the goroutine, a new channel named resultChan is created to communicate the result of file processing back to the main thread.

7. Goroutine for Asynchronous File Processing: Another goroutine is launched using the splitFileAndSendChunksAsync function. It handles splitting the file into chunks and sending them to the server. Any errors encountered are sent back through the resultChan.

8. Handling Result: The main goroutine waits for the result from the resultChan. If an error occurs, it prints an error message; otherwise, it prints a success message.

9. WaitGroup Done: The wg.Done() call indicates that the current file's processing is complete.

10. Waiting for All Goroutines to Finish: After all goroutines are spawned, the wg.Wait() call ensures that the program waits until all file processing is complete before moving on.

11. Final Scanln: A fmt.Scanln() call is used to prevent the program from exiting immediately after all file processing is done. It waits for a newline input from the user.

In summary, this client program reads files from the filePaths slice, splits them into chunks, and sends those chunks to an HTTPS server asynchronously. It uses goroutines and channels to handle the concurrency and communication between different parts of the program.