GO

-Report-

The code constructs a matrix multiplicator with a Go application. A client sends it matrices, which it receives, multiplies, and then sends back the answer.

Function for Matrix Multiplication:

The function "matrix multiplication" handles matrix multiplication. The two matrices "a" and "b," the matrix "c" to hold the result, the row and column indices for which the multiplication is to be done, and a mutex lock "mutex as parameter" are all required arguments for this function. The function multiplies the matrices and stores the result in the "c" matrix using a for loop.

Handling Connections:

Incoming connections from clients are handled by the "handleConnection" function. It requires a WaitGroup "wg" and a "connection" object denoting the client. In addition to wishing the user a warm welcome, the function also asks for the two matrices' dimensions. The function reads the client input using a "scan" object. An error message is provided to the client and the connection is cut off if the input is not in the proper format. The "matrix multiplication" function is used to multiply the matrices after they have been read from the client. The connection is then ended after the result has been returned to the client.

Technical Aspects:

Our code uses the bufio package for buffered I/O operations like reading from or writing to files. Instead of reading or writing data one byte at a time, buffered I/O transfers data in larger chunks. Since more data may be sent in a single call, this can increase the performance of I/O operations by lowering the number of read or write system calls that are necessary. Buffered readers and writers, which wrap an existing I/O stream and add the buffering functionality, are offered by the bufio package.

In the Go programming language, routines are used to run functions concurrently. By breaking the task up into smaller components and running each component in a separate go routine, they were employed in this example to parallelize the matrix multiplication operation. This enables various operations to be conducted concurrently, potentially lowering the total amount of time needed to do the activity.

A mutex, which stands for "mutual exclusion," is used in the code to guarantee that only one goroutine can access the shared resource (such as the matrix) at once. This is done to avoid race circumstances, which occur when several goroutines attempt to edit the same resource at the same time and produce unpredictable consequences. The code can synchronize access to the shared resource and avoid any data damage by utilizing a mutex.

Before the program terminates, WaitGroup is used to wait for all go routines to complete their execution. The number of goroutines that have not yet completed is counted using the WaitGroup type. The counter is incremented using the Add function and decremented using the Done method. Until all goroutines have completed their task, the Wait method blocks until the counter hits zero. The WaitGroup is utilized in this scenario to ensure that the final result matrix is appropriately computed before printing.

Conclusion

In summary, the code effectively manages client connections and offers a straightforward implementation of a matrix multiplicator.