Explain how the TCP connection is established between the client and server. How does the server handle incoming connections?

The server listens on port 8080, accepting incoming connections from clients and handling each one in a separate goroutine. This concurrent handling allows the server to manage multiple clients simultaneously, which is essential for scalability.

Once a client connects, the `handleClient` function reads a message from the client, prints it to the server console, and sends a confirmation response back. The connection is automatically closed when `handleClient` exits.

On the client side, it connects to the server on `localhost:8080` using `net.Dial`. The client then enters a loop, prompting the user for a message. After sending the message to the server, the client waits for and displays the server's response. If the user enters "exit\n", the client closes the connection and exits the loop.

This setup showcases basic client-server communication in Go, with the server concurrently managing multiple clients and each client sending and receiving messages independently until disconnection.

What challenge does the server face when handling multiple clients, and how does Go's concurrency model help solve this problem?

The primary challenge the server faces when handling multiple clients is managing concurrent access to shared resources, specifically the `clients` map. Each connected client is stored in this map to enable message broadcasting across all active connections. Without proper management, concurrent access could lead to data races, where multiple goroutines (threads) attempt to read from or write to the `clients` map simultaneously, potentially causing errors or unpredictable behavior.

Go's concurrency model, powered by lightweight goroutines and synchronization primitives like `sync.Mutex`, effectively addresses this challenge. Each client connection is managed in a separate goroutine, allowing the server to listen and respond to multiple clients simultaneously without blocking operations. To handle the shared `clients` map safely, a mutex (`mu`) is introduced. The mutex locks access to the map whenever a client is added, removed, or a message is broadcasted, ensuring that only one goroutine can modify the `clients` map at a time. This prevents race conditions and keeps the map's data integrity intact.

The use of goroutines allows the server to concurrently manage client connections, read and broadcast messages independently, and effectively scale as more clients join. The mutex, on the other hand, ensures safe access to shared resources, allowing the server to handle multiple clients reliably and efficiently. Through this combination, Go's concurrency model provides a simple yet powerful solution to the challenge of concurrent network communication.

How does the server assign tasks to the clients? What real-world distributed systems scenario does this model resemble?

The server assigns tasks to clients by sending each connected client a timestamp-based number every five seconds. This number is generated by calculating the current Unix time modulo 100. Each client receives this number, computes its square as the "task result," and sends this result back to the server. The server reads and logs each client’s response before assigning a new task in the next cycle.

This model resembles a real-world distributed systems scenario like a distributed task processing system, which commonly used in systems where a central server delegates computation-heavy or repetitive tasks across multiple clients or workers. A common example is in distributed computing environments like scientific research or data processing clusters, where a main server dispatches portions of a workload to available worker nodes (clients) to parallelize the processing. Another example is a load balancer in cloud computing, which distributes incoming tasks to different servers or services to improve processing speed and system efficiency.

Each client acts as a worker node that receives and processes tasks independently. The server, in turn, can focus on delegating tasks rather than performing the computations itself, which is beneficial for scaling and managing heavy workloads in a distributed system. This setup leverages concurrency and task distribution to enhance efficiency and throughput, a pattern essential to distributed systems in large-scale computing environments.