

1) TCP connection that is established between the client and server and how the server handle incoming connections:

For the TCP connection establishment, the server runs `net.Listen("tcp": ":8080")` which opens a socket and waits for a connections. The client calls `net.Dial("tcp": "localhost:8080")`, which sends a tcp syn request the the server accept receives the syn completing the tcp three way handshake. Once connected, both sides can read/write using the socket.

For how the server handles connections, the server sits in the loop (`conn, _ := listener.Accept()`), and each connection is handled in goroutine (`go handleClient(conn)`) which allows the server to process multiple clients at the same time.

2) The challenges the server faces when handling multiple clients and how Go's concurrency model helps solve this problem:

Some of the challenges are multiple clients sending messages at the same time, shared data can cause race conditions and server must remain responsive even when one client is slow.

Go's solution is that Go uses goroutines which are lightweight threads managed by the Go runtime. Each client runs the `handleClient` and `sync.Mutex` protects the shared data which helps the server to handle many clients smoothly without blocking.

3) How the server assigns tasks to clients and what real world distributed systems scenario the model resembles:

For each client, the server generates a number, then the client receives that number squares it (process it) and sends it back the result. A real world distributed system analogy would be Apache Spark workers, Hadoop MapReduce and etc.