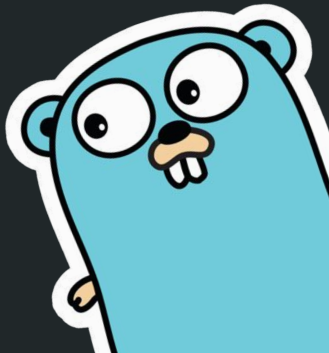
Cliente/Servidor **TCP & UDP**

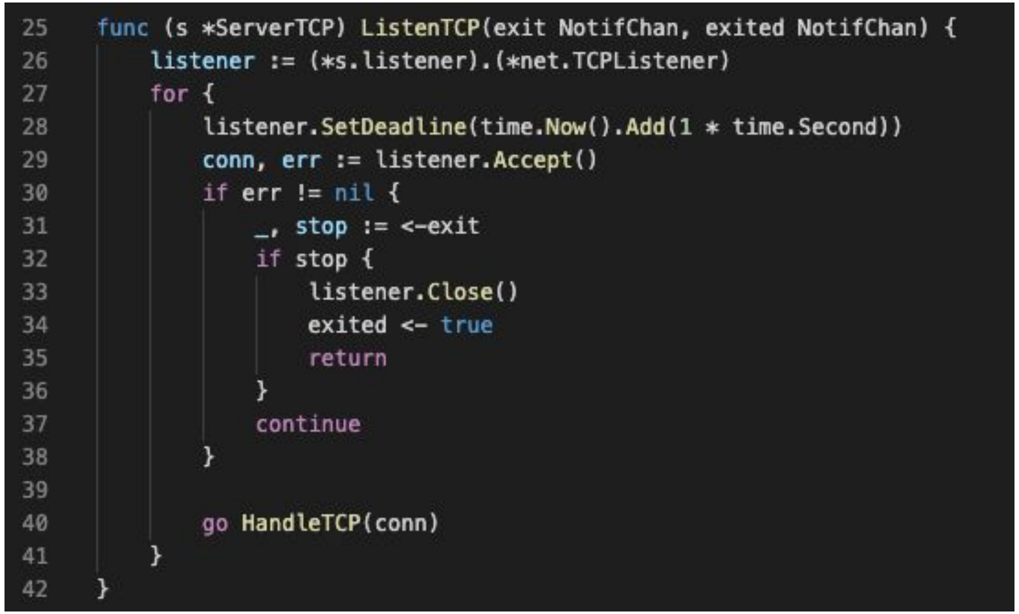
Uma implementação em **Go**



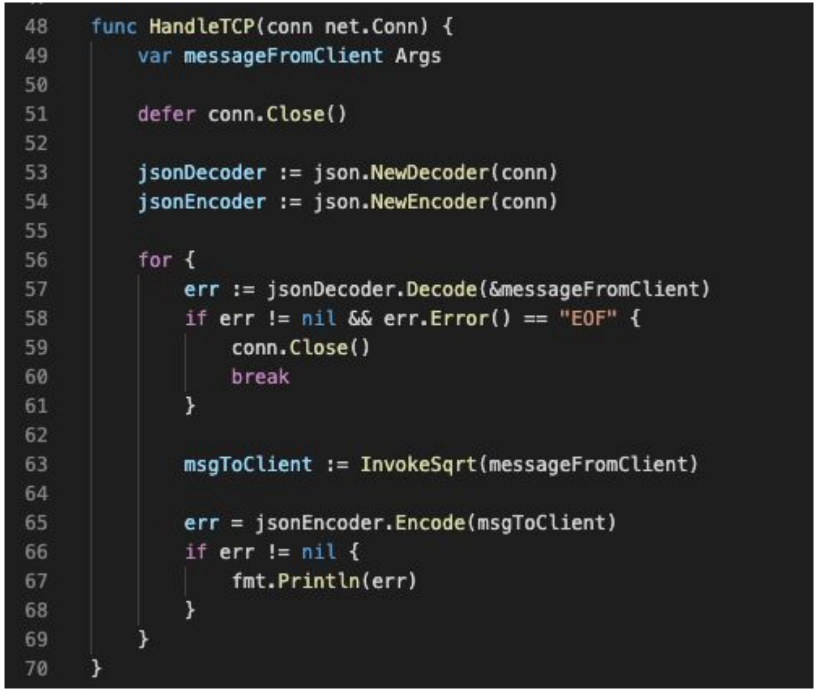
Equipe:

- Gabriel Pessoa
- Luan Brito
- Pedro Queiroga
- Ramon Saboya
- Rodrigo Cunha
- Saulo Guilhermino

Servidor **TCP**

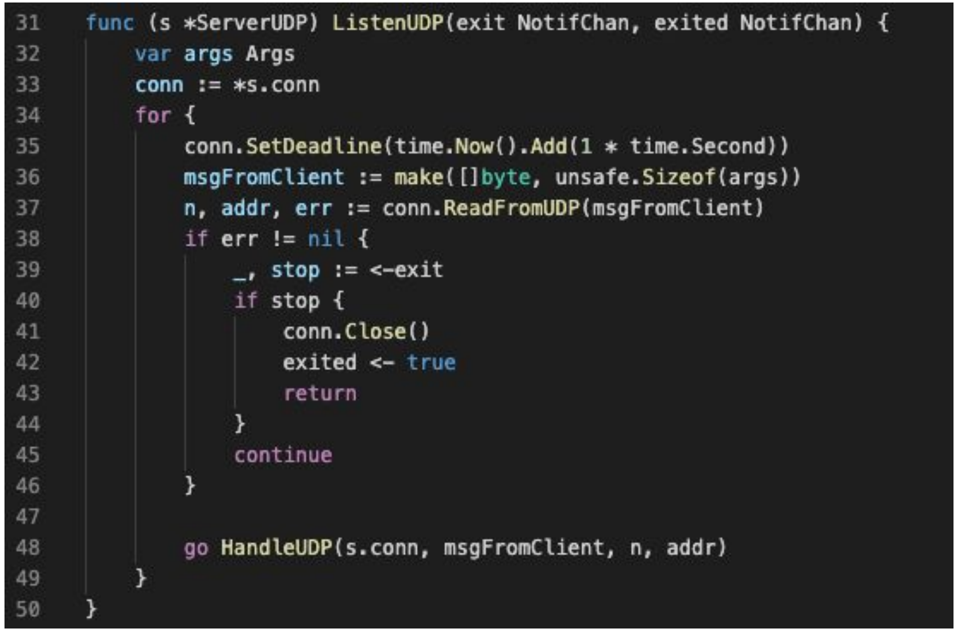


```go
func (s *ServerTCP) ListenTCP(exit NotifChan, exited NotifChan) {
	listener := (*s.listener).(*net.TCPListener)
	for {
		listener.SetDeadline(time.Now().Add(1 * time.Second))
		conn, err := listener.Accept()
		if err != nil {
			_, stop := <-exit
			if stop {
				listener.Close()
				exited <- true
				return
			}
			continue
		}

		go HandleTCP(conn)
	}
}
```

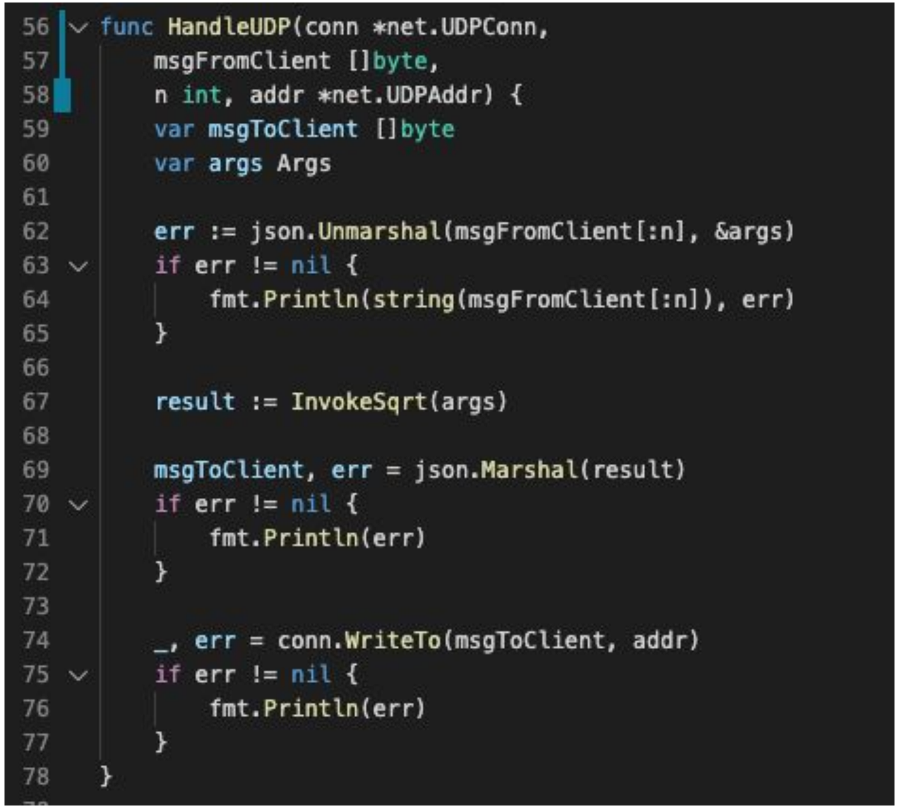


```go
func HandleTCP(conn net.Conn) {
	var messageFromClient Args

	defer conn.Close()

	jsonDecoder := json.NewDecoder(conn)
	jsonEncoder := json.NewEncoder(conn)

	for {
		err := jsonDecoder.Decode(&messageFromClient)
		if err != nil && err.Error() == "EOF" {
			conn.Close()
			break
		}

		msgToClient := InvokeSqrt(messageFromClient)

		err = jsonEncoder.Encode(msgToClient)
		if err != nil {
			fmt.Println(err)
		}
	}
}
```

Servidor **UDP**

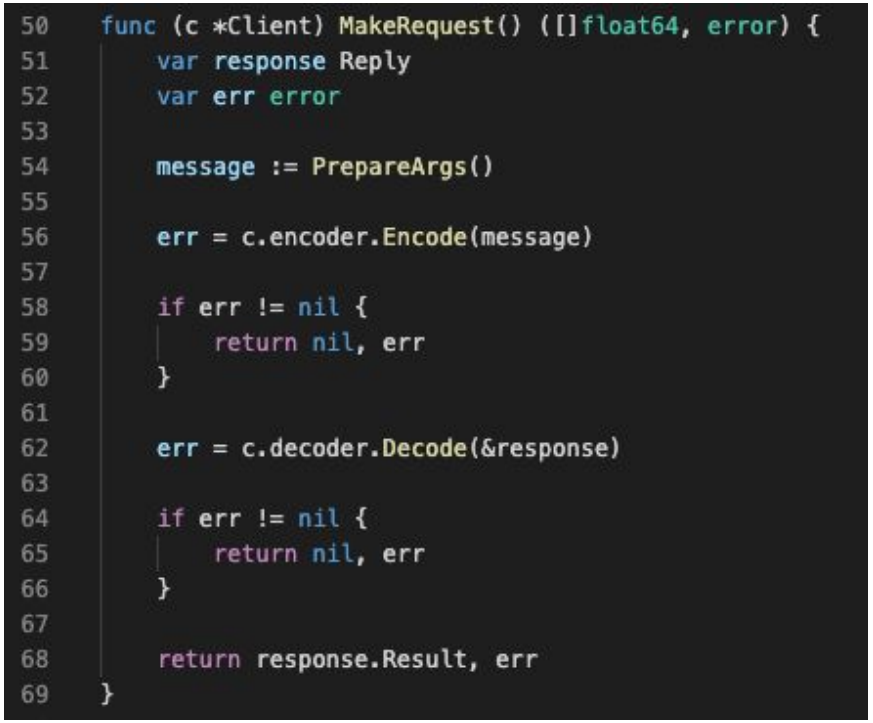


```go
func (s *ServerUDP) ListenUDP(exit NotifChan, exited NotifChan) {
    var args Args
    conn := *s.conn
    for {
        conn.SetDeadline(time.Now().Add(1 * time.Second))
        msgFromClient := make([]byte, unsafe.Sizeof(args))
        n, addr, err := conn.ReadFromUDP(msgFromClient)
        if err != nil {
            _, stop := <-exit
            if stop {
                conn.Close()
                exited <- true
                return
            }
            continue
        }

        go HandleUDP(s.conn, msgFromClient, n, addr)
    }
}
```

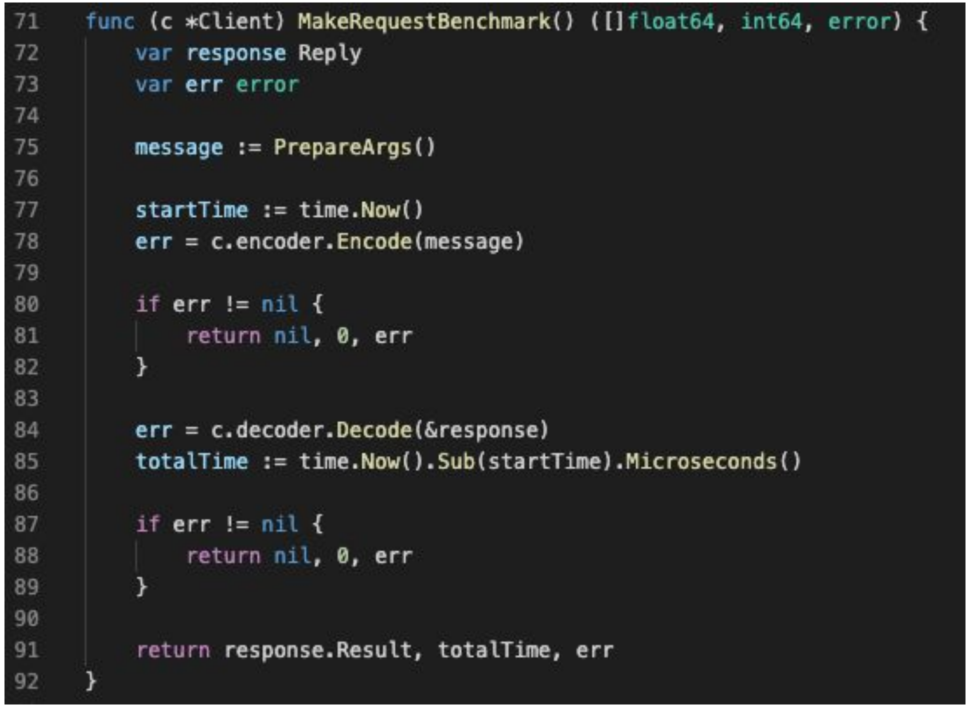


```go
func HandleUDP(conn *net.UDPConn,
    msgFromClient []byte,
    n int, addr *net.UDPAddr) {
    var msgToClient []byte
    var args Args

    err := json.Unmarshal(msgFromClient[:n], &args)
    if err != nil {
        fmt.Println(string(msgFromClient[:n]), err)
    }

    result := InvokeSqrt(args)

    msgToClient, err = json.Marshal(result)
    if err != nil {
        fmt.Println(err)
    }

    _, err = conn.WriteTo(msgToClient, addr)
    if err != nil {
        fmt.Println(err)
    }
}
```

Cliente

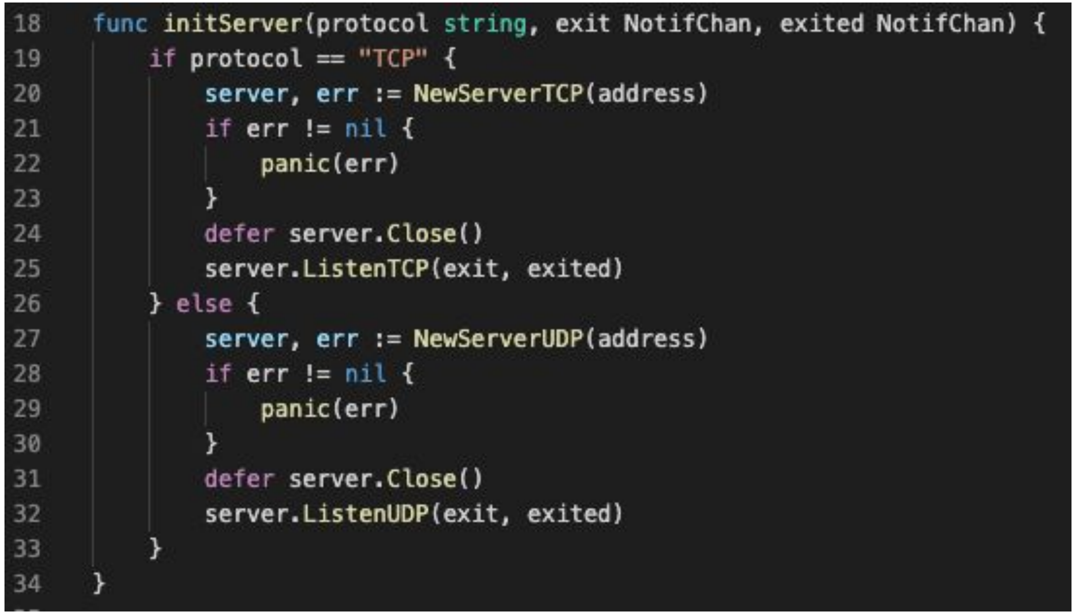


```go
func (c *Client) MakeRequest() ([]float64, error) {
    var response Reply
    var err error

    message := PrepareArgs()

    err = c.encoder.Encode(message)

    if err != nil {
        return nil, err
    }

    err = c.decoder.Decode(&response)

    if err != nil {
        return nil, err
    }

    return response.Result, err
}
```

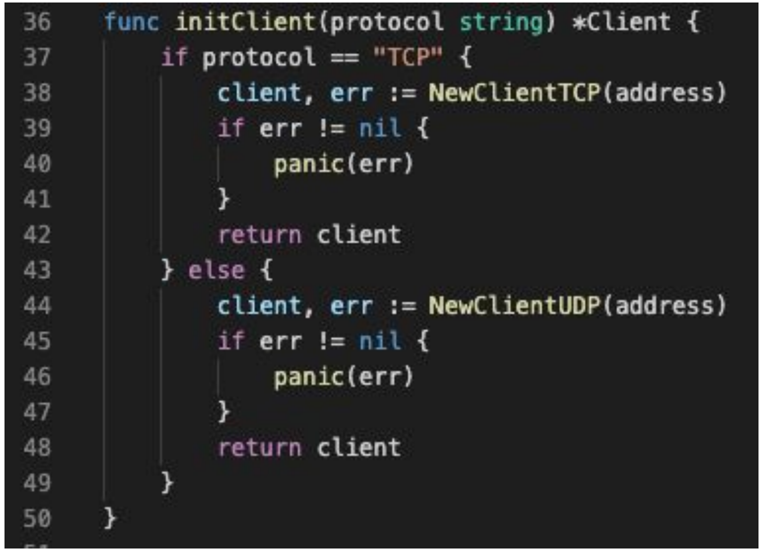


```go
func (c *Client) MakeRequestBenchmark() ([]float64, int64, error) {
    var response Reply
    var err error

    message := PrepareArgs()

    startTime := time.Now()
    err = c.encoder.Encode(message)

    if err != nil {
        return nil, 0, err
    }

    err = c.decoder.Decode(&response)
    totalTime := time.Now().Sub(startTime).Microseconds()

    if err != nil {
        return nil, 0, err
    }

    return response.Result, totalTime, err
}
```

Main

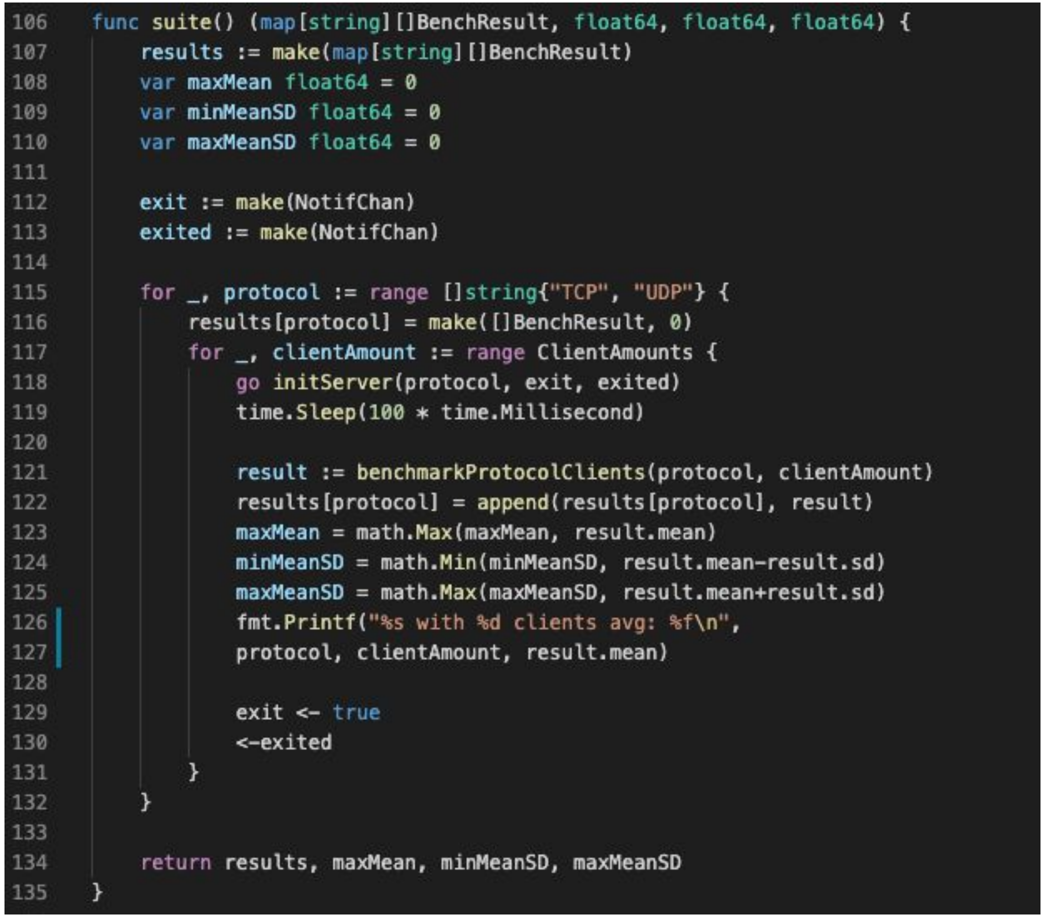


```go
func initServer(protocol string, exit NotifChan, exited NotifChan) {
    if protocol == "TCP" {
        server, err := NewServerTCP(address)
        if err != nil {
            panic(err)
        }
        defer server.Close()
        server.ListenTCP(exit, exited)
    } else {
        server, err := NewServerUDP(address)
        if err != nil {
            panic(err)
        }
        defer server.Close()
        server.ListenUDP(exit, exited)
    }
}
```



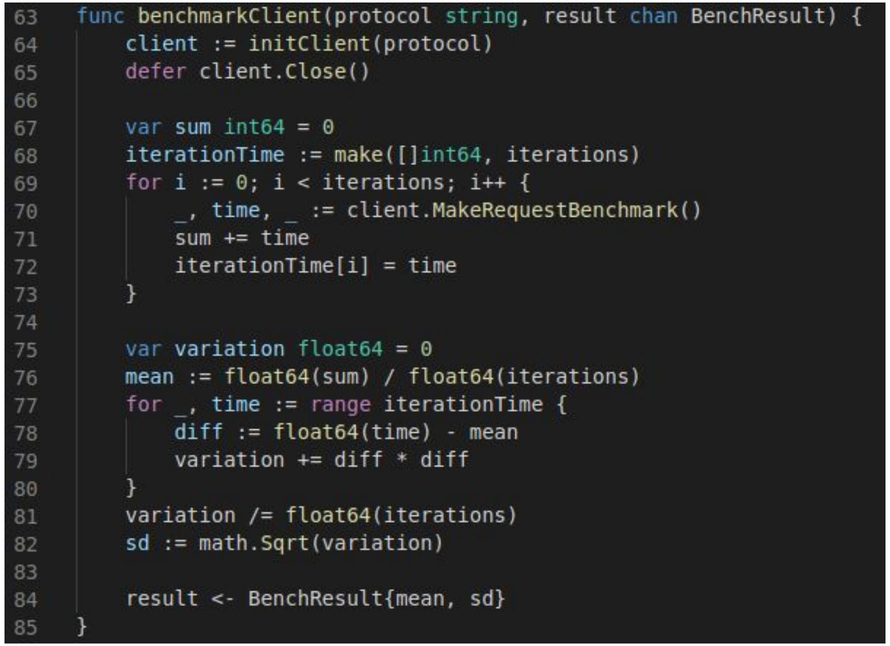
```go
func initClient(protocol string) *Client {
    if protocol == "TCP" {
        client, err := NewClientTCP(address)
        if err != nil {
            panic(err)
        }
        return client
    } else {
        client, err := NewClientUDP(address)
        if err != nil {
            panic(err)
        }
        return client
    }
}
```

Main

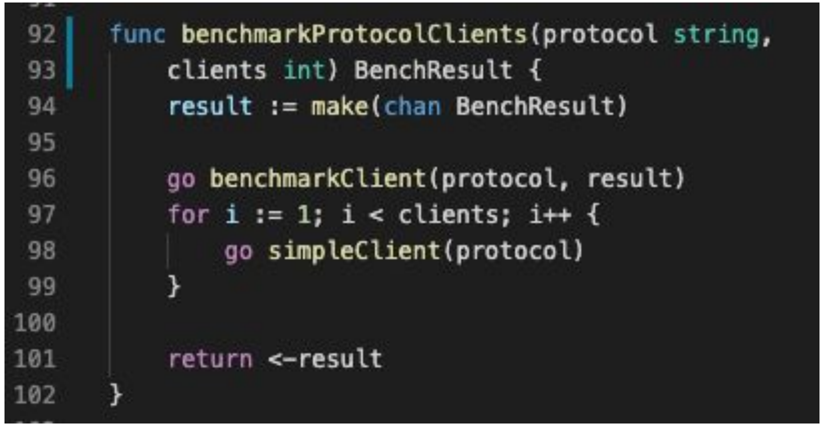


```go
func suite() (map[string][]BenchResult, float64, float64, float64) {
    results := make(map[string][]BenchResult)
    var maxMean float64 = 0
    var minMeanSD float64 = 0
    var maxMeanSD float64 = 0

    exit := make(NotifChan)
    exited := make(NotifChan)

    for _, protocol := range []string{"TCP", "UDP"} {
        results[protocol] = make([]BenchResult, 0)
        for _, clientAmount := range ClientAmounts {
            go initServer(protocol, exit, exited)
            time.Sleep(100 * time.Millisecond)

            result := benchmarkProtocolClients(protocol, clientAmount)
            results[protocol] = append(results[protocol], result)
            maxMean = math.Max(maxMean, result.mean)
            minMeanSD = math.Min(minMeanSD, result.mean-result.sd)
            maxMeanSD = math.Max(maxMeanSD, result.mean+result.sd)
            fmt.Printf("%s with %d clients avg: %f\n",
            protocol, clientAmount, result.mean)

            exit <- true
            <-exited
        }
    }

    return results, maxMean, minMeanSD, maxMeanSD
}
```

Main

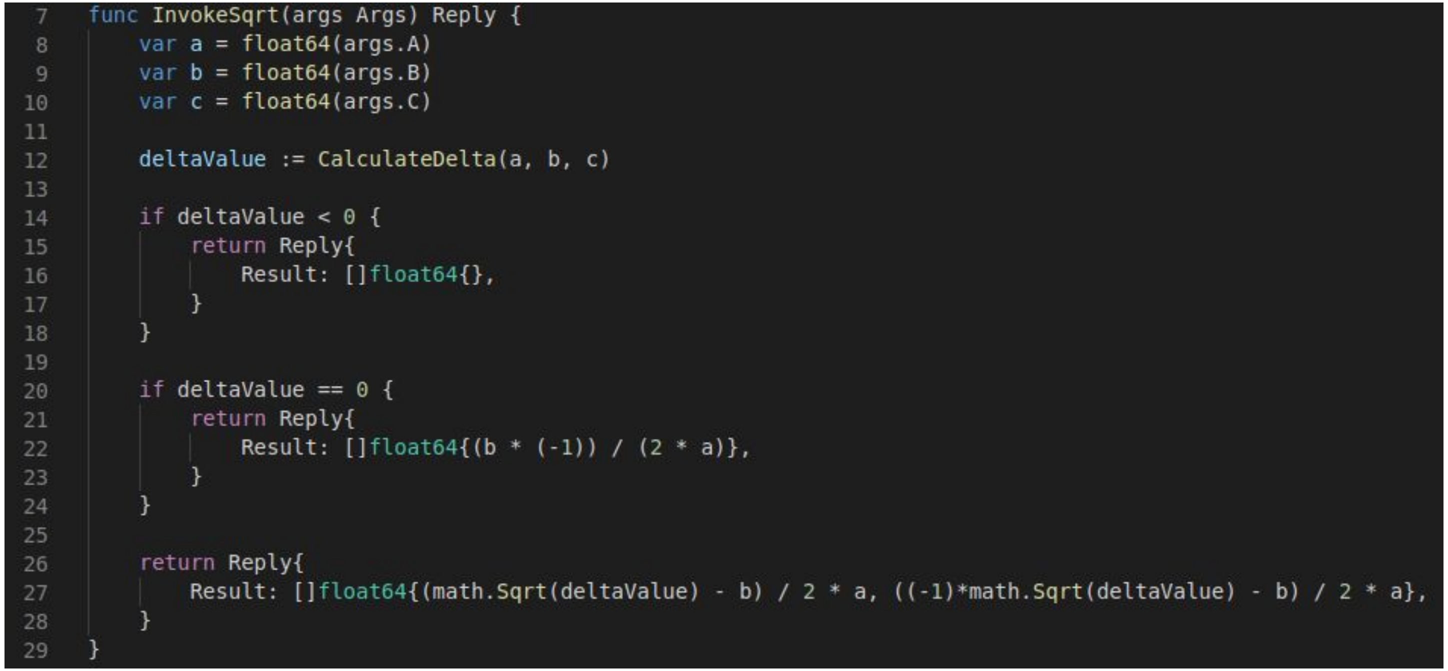


```go
func benchmarkClient(protocol string, result chan BenchResult) {
    client := initClient(protocol)
    defer client.Close()

    var sum int64 = 0
    iterationTime := make([]int64, iterations)
    for i := 0; i < iterations; i++ {
        _, time, _ := client.MakeRequestBenchmark()
        sum += time
        iterationTime[i] = time
    }

    var variation float64 = 0
    mean := float64(sum) / float64(iterations)
    for _, time := range iterationTime {
        diff := float64(time) - mean
        variation += diff * diff
    }
    variation /= float64(iterations)
    sd := math.Sqrt(variation)

    result <- BenchResult{mean, sd}
}
```



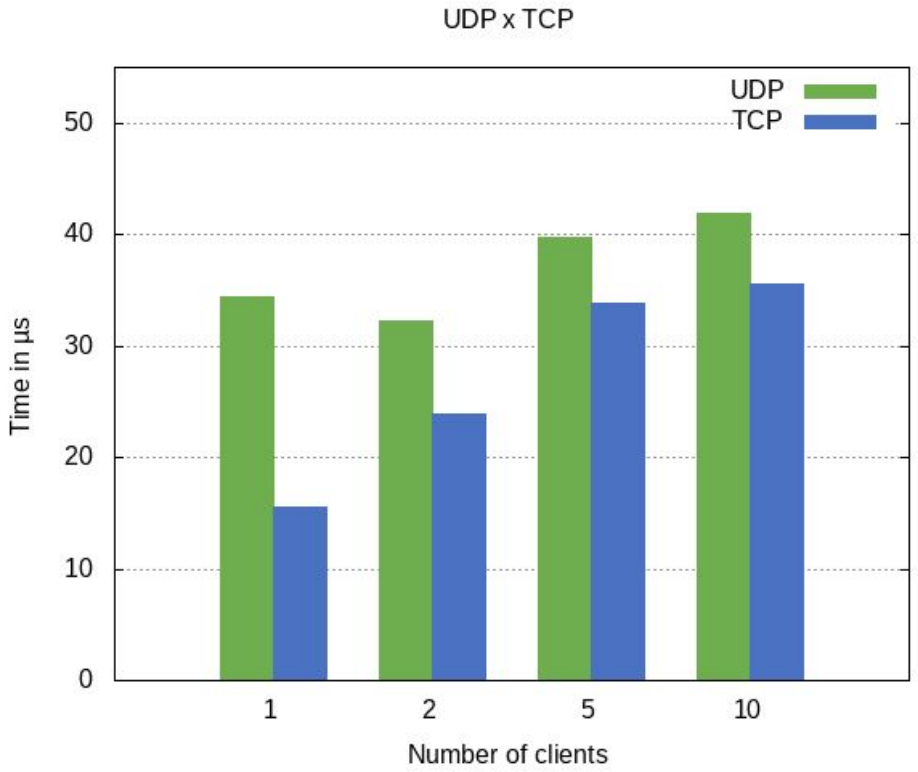
```go
func benchmarkProtocolClients(protocol string,
    clients int) BenchResult {
    result := make(chan BenchResult)

    go benchmarkClient(protocol, result)
    for i := 1; i < clients; i++ {
        go simpleClient(protocol)
    }

    return <-result
}
```

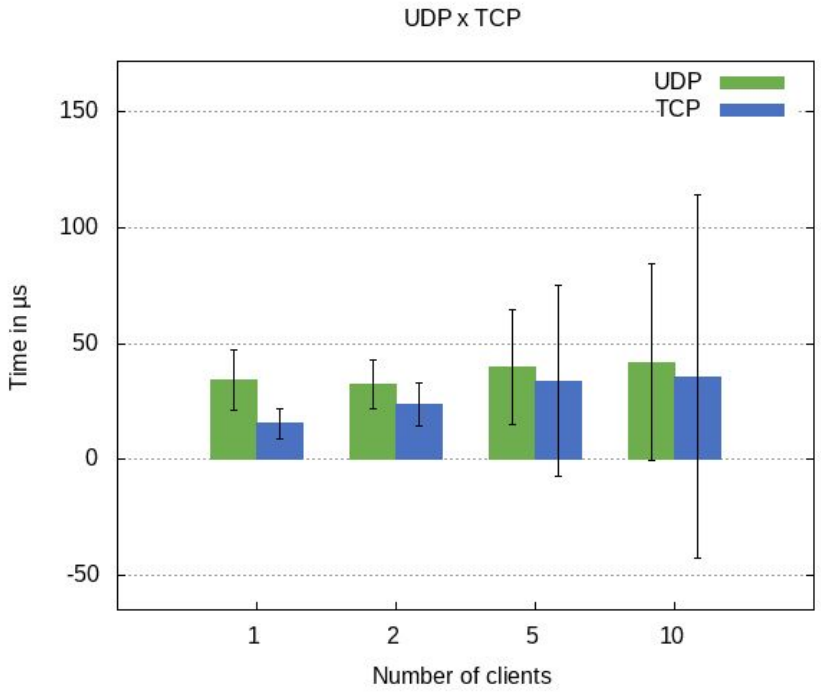
Core



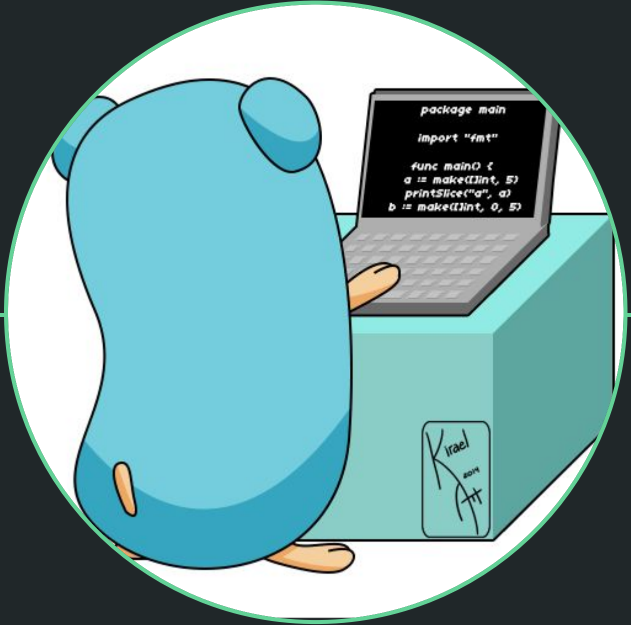
```go
func InvokeSqrt(args Args) Reply {
    var a = float64(args.A)
    var b = float64(args.B)
    var c = float64(args.C)

    deltaValue := CalculateDelta(a, b, c)

    if deltaValue < 0 {
        return Reply{
            Result: []float64{},
        }
    }

    if deltaValue == 0 {
        return Reply{
            Result: []float64{(b * (-1)) / (2 * a)},
        }
    }

    return Reply{
        Result: []float64{(math.Sqrt(deltaValue) - b) / 2 * a, ((-1)*math.Sqrt(deltaValue) - b) / 2 * a},
    }
}
```

Avaliação de **Desempenho**





Demo!



```
package main

import "fmt"

func main() {
  a := make([]int, 5)
  printSlice("a", a)
  b := make([]int, 0, 5)
```