Create a TCP and UDP Client and Server using Go

Updated Thursday, March 9, 2023, by Mihalis Tsoukalos

Go is a compiled, statically typed programming language developed by Google. Many modern applications, including Docker, Kubernetes, and Terraform, are written in Go. Go packages allow developers to organize and reuse Go code in a simple and maintainable manner.

In this guide, you will use the `net` package, which is a part of Go's standard library, to create TCP and UDP servers and clients. This guide is meant to provide instructional examples to help you become more familiar with the Go programming language.

Scope of this Guide

Throughout this guide you will create the following:

A TCP server and client. The TCP server accepts incoming messages from a TCP client and responds with the current date and time.

A UDP server and client. The UDP server accepts incoming messages from a UDP client and responds with a random number.

A concurrent TCP server that accepts incoming messages from several TCP clients and responds with the number of clients currently connected to it.

Before You Begin

1. If you are not familiar with using Go packages, review the Getting Started with Go Packages guide.
2. Install Go on your computer if it is not already installed. You can follow our guide How to Install Go on Ubuntu for installation steps.

   This guide requires Go version 1.8 or higher. It is considered good practice to have the latest version of Go installed. You can check your Go version by executing the following command:

   ```
   go version
   ```

> **Note**
>
> This guide is written for a non-root user. Depending on the TCP/IP port number that you use when running the TCP and UDP servers, you may need to prefix commands with `sudo`. If you are not familiar with the `sudo` command, see the Users and Groups guide.

Protocol Definitions

Protocol	Definition
TCP (Transmission Control Protocol)	TCP's principal characteristic is that it is a reliable protocol by design. If there is no proof of a packet's delivery, TCP will resend the packet. Some of the tasks TCP packets can be used for are establishing connections, transferring data, sending acknowledgements, and closing connections.

IP (Internet Protocol)	The IP protocol adheres to the end-to-end principle, which places all network intelligence in the end nodes and not in the intermediary nodes. This design favors a reduction in network complexity over reliability. For this reason, the Internet Protocol does not guarantee a reliable delivery of packets over a network. Instead, IP works together with TCP to reliably deliver packets over a network.
UDP (User Datagram Protocol):	UDP provides a simpler implementation of the transport layer protocol that, while less reliable than TCP, is much faster. UDP does not provide error checking, correction or packet retransmission, which makes it very fast. When speed is more important than reliability, UDP is generally chosen over TCP. UDP is commonly used for online gaming, video chatting, and other real-time applications.

The net Package

Go's `net` package provides a portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets. You will use this package to create TCP and UDP servers and clients in this guide.

net Package Functions

Use the table below as a quick reference for some of the `net` package functions used throughout this guide. To view all types and functions included in the `net` package, see Golang's official documentation.

> **Note**
>
> All versions of `net.Dial()` and `net.Listen()` return data types that implement the `io.Reader` and `io.Writer` interfaces. This means that you can use regular File I/O functions to send and receive data from a TCP/IP connection.

Type	Function
type Listener	`func Listen(network, address string) (Listener, error)` • The `network` parameter defines the type of network to use and accepts values `tcp`, `tcp4` (IPv4-only), `tcp6` (IPv6-only), `unix` (Unix sockets), or `unixpacket`. • The `address` parameter defines the server address and port number that the server will listen on.

type UDPConn	`func ListenUDP(network string, laddr *UDPAddr) (*UDPConn, error)`
	• Used to create UDP servers.
	• The `network` parameter must be a UDP network name.
	• The `laddr` parameter defines the server address and port number that the server will listen on.
	`func DialUDP(network string, laddr, raddr *UDPAddr) (*UDPConn, error)`
	• Used to specify the kind of client you will create.

	• The `network` parameter must be a UDP network name. • The `laddr` is the listening address (server). If `laddr` is nil, a local address is automatically chosen. • `raddr` is the response address (client). If the IP field of `raddr` is nil or an unspecified IP address, the local system is assumed.
type UDPAddr	`func ResolveUDPAddr(network, address string) (*UDPAddr, error)` • This function returns the address of a UDP end point. • The `network` parameter must be a UDP network name. • The `address` parameter has the form `host:port`. The host must be a an IP address, or a host name that can be resolved to IP addresses.
type TCPAddr	`func ResolveTCPAddr(network, address string) (*TCPAddr, error)` • This function returns the address of a TCP end point. • The `network` parameter must be a TCP network name. • The `address` parameter has the form `host:port`. The host must be a an IP address, or a host name that can be resolved to IP addresses.
type Conn	`func Dial(network, address string) (Conn, error)` • This function connects to the address on the named network. • The `network` parameter can be `tcp`, `tcp4` (IPv4-only), `tcp6` (IPv6-only), `udp`, `udp4` (IPv4-only), `udp6` (IPv6-only), `ip`, `ip4` (IPv4-only), `ip6` (IPv6-only), `unix`, `unixgram` and `unixpacket`. • When using TCP or UDP networks, the `address` parameter has the form `host:port`. The host must be a an IP address, or a host name that can be resolved to IP addresses.
type TCPConn	`func DialTCP(network string, laddr, raddr *TCPAddr) (*TCPConn, error)` • This function connects to the address on the TCP networks. • The `network` parameter must be a TCP network name. • The `laddr` is the listening address (server). If `laddr` is nil, a local address is automatically chosen. • `raddr` is the response address (client). If the IP field of `raddr` is nil or an unspecified IP address, the local system is assumed.

Create a TCP Client and Server

In this section, you will create a generic TCP client and server using Go. After creating the client and server, you will run them to test their connection with each other.

> **Note**
>
> The netcat command line utility can be used to test TCP/IP client and server connections.

Create the TCP Client

The TCP client that you will create in this section will allow you to interact with any TCP server.

1. In your current working directory, create a file named `tcpC.go` with the following content:

File: ./tcpC.go

```
package main

import (
        "bufio"
        "fmt"
        "net"
        "os"
        "strings"
)

func main() {
        arguments := os.Args
        if len(arguments) == 1 {
                fmt.Println("Please provide host:port."
                return
        }

        CONNECT := arguments[1]
        c, err := net.Dial("tcp", CONNECT)
        if err != nil {
                fmt.Println(err)
                return
        }

        for {
                reader := bufio.NewReader(os.Stdin)
                fmt.Print(">> ")
                text, _ := reader.ReadString('\n')
                fmt.Fprintf(c, text+"\n")

                message, _ := bufio.NewReader(c).ReadSt
                fmt.Print("->: " + message)
                if strings.TrimSpace(string(text)) == "
                        fmt.Println("TCP client exiting
                        return
                }
        }
}
```

This file creates the `main` package, which declares the `main()` function. The function will use the imported packages to create a TCP client.

The `main()` function gathers command line arguments in the `arguments` variable and makes sure that a value for `host:port` was sent.

The `CONNECT` variable stores the value of `arguments[1]` to be used in the `net.Dial()` call.

A call to `net.Dial()` begins the implementation of the TCP client and will connect you to the desired TCP server. The second parameter of `net.Dial()` has two parts; the first is the hostname or the IP address of the TCP server and the second is the port number the

TCP server listens on.

`bufio.NewReader(os.Stdin)` and `ReadString()` is used to read user input. Any user input is sent to the TCP server over the network using `Fprintf()`.

`bufio` reader and the `bufio.NewReader(c).ReadString('\n')` statement read the TCP server's response. The `error` variable is ignored here for simplicity.

The entire `for` loop that is used to read user input will only terminate when you send the `STOP` command to the TCP server.

Create the TCP Server

You are now ready to create the TCP server. The TCP server will return the current date and time to the TCP client using a single network packet.

1. In your current working directory, create a file named `tcpS.go` with the following content:

File: ./tcpS.go

```
package main

import (
        "bufio"
        "fmt"
        "net"
        "os"
        "strings"
        "time"
)

func main() {
        arguments := os.Args
        if len(arguments) == 1 {
                fmt.Println("Please provide port number
                return
        }

        PORT := ":" + arguments[1]
        l, err := net.Listen("tcp", PORT)
        if err != nil {
                fmt.Println(err)
                return
        }
        defer l.Close()

        c, err := l.Accept()
        if err != nil {
                fmt.Println(err)
                return
        }

        for {
                netData, err := bufio.NewReader(c).Read
                if err != nil {
                        fmt.Println(err)
                        return
                }
                if strings.TrimSpace(string(netData)) =
                        fmt.Println("Exiting TCP server
                        return
                }
```

```
43
44                  fmt.Print("-> ", string(netData))
45                  t := time.Now()
46                  myTime := t.Format(time.RFC3339) + "\n"
47                  c.Write([]byte(myTime))
48          }
49  }
50
```

This file creates the `main` package, which declares the `main()` function. The function will use the imported packages to create a TCP server.

The `main()` function gathers command line arguments in the `arguments` variable and includes error handling.

The `net.Listen()` function makes the program a TCP server. This functions returns a `Listener` variable, which is a generic network listener for stream-oriented protocols.

It is only after a successful call to `Accept()` that the TCP server can begin to interact with TCP clients.

The current implementation of the TCP server can only serve the first TCP client that connects to it, because the `Accept()` call is outside of the `for` loop. In the Create a Concurrent TCP Server section of this guide, you will see a TCP server implementation that can serve multiple TCP clients using Goroutines.

The TCP server uses regular File I/O functions to interact with TCP clients. This interaction takes place inside the `for` loop. Similarly to the TCP client, when the TCP server receives the `STOP` command from the TCP client, it will terminate.

Test the TCP Client and Server

You can now test your TCP client and server. You will need to execute the TCP server first so that the TCP client has somewhere it can connect to.

1. Run your TCP server. From the directory containing the `tcpS.go` file, run the following command:

   ```
   go run tcpS.go 1234
   ```

   The server will listen on port number `1234`. You will not see any output as a result of this command.

2. Open a second shell session to execute the TCP client and to interact with the TCP server. Run the following command:

   ```
   go run tcpC.go 127.0.0.1:1234
   ```

   > **Note**
   >
   > If the TCP server is not running on the expected TCP port, you will get the following error message from `tcpC.go`:
   >
   > ```
   > dial tcp [::1]:1234: connect: connection refused
   > ```

3. You will see a `>>` prompt waiting for you to enter some text. Type in `Hello!` to receive a response from the TCP server:

   ```
   Hello!
   ```

   You should see a similar output:

   ```
   >> Hello!
   ```

```
->: 2019-05-23T19:43:21+03:00
```

4. Send the `STOP` command to exit the TCP client and server:

```
STOP
```

You should see a similar output in the client:

```
>> STOP
->: TCP client exiting...
```

The output on the TCP server side will resemble the following:

```
-> Hello!
Exiting TCP server!
```

> **Note**
>
> The TCP server waits before writing back to the TCP client, whereas the client writes to the TCP server first and then waits to receive an answer. This behavior is part of the protocol definition that governs a TCP or a UDP connection. In this example, you have implemented an unofficial protocol that is based on TCP.

Create a UDP Client and Server

In this section, you will create a UDP client and server. After creating the client and server, you will run them both to test their connection with each other. A UDP client can be generic and can communicate with multiple UDP servers. On the other hand, a UDP server cannot be completely generic, because it typically implements a specific functionality. In the case of our UDP server example, it will return random numbers to UDP clients that connect to it.

Create the UDP Client

The UDP client that you will create in this section will allow you to interact with any UDP server.

1. In your current working directory, create a file named `udpC.go` with the following content:

File: ./udpC.go

```
package main

import (
        "bufio"
        "fmt"
        "net"
        "os"
        "strings"
)

func main() {
        arguments := os.Args
        if len(arguments) == 1 {
                fmt.Println("Please provide a host:port
                return
        }
```

```
        }
        CONNECT := arguments[1]

        s, err := net.ResolveUDPAddr("udp4", CONNECT)
        c, err := net.DialUDP("udp4", nil, s)
        if err != nil {
                fmt.Println(err)
                return
        }

        fmt.Printf("The UDP server is %s\n", c.RemoteAd
        defer c.Close()

        for {
                reader := bufio.NewReader(os.Stdin)
                fmt.Print(">> ")
                text, _ := reader.ReadString('\n')
                data := []byte(text + "\n")
                _, err = c.Write(data)
                if strings.TrimSpace(string(data)) == "
                        fmt.Println("Exiting UDP client
                        return
                }

                if err != nil {
                        fmt.Println(err)
                        return
                }

                buffer := make([]byte, 1024)
                n, _, err := c.ReadFromUDP(buffer)
                if err != nil {
                        fmt.Println(err)
                        return
                }
                fmt.Printf("Reply: %s\n", string(buffer
        }
}
```

This file creates the `main` package, which declares the `main()` function. The function will use the imported packages to create a UDP client.

The `main()` function gathers command line arguments in the `arguments` variable and includes error handling.

Regular File I/O functions are used by the UDP client to interact with the UDP server. The client will terminate when you send the `STOP` command to the UDP server. This is not part of the UDP protocol, but is used in the example to provide the client with a way to exit.

A UDP end point address is returned by the `net.ResolveUDPAddr()` function. The UDP end point is of type `UDPAddr` and contains IP and port information.

The connection to the UDP server is established with the use of the `net.DialUDP()` function.

`bufio.NewReader(os.Stdin)` and `ReadString()` is used to read user input.

The `ReadFromUDP()` function reads a packet from the server connection and will return if it encounters an error.

Create the UDP Server

You are now ready to create the UDP server. You will write the UDP server code to respond to

any connected client with random numbers.

1. In your current working directory, create a file named `udps.go` with the following content:

File: ./udpS.go

```
package main

import (
        "fmt"
        "math/rand"
        "net"
        "os"
        "strconv"
        "strings"
        "time"
)

func random(min, max int) int {
        return rand.Intn(max-min) + min
}

func main() {
        arguments := os.Args
        if len(arguments) == 1 {
                fmt.Println("Please provide a port numb
                return
        }
        PORT := ":" + arguments[1]

        s, err := net.ResolveUDPAddr("udp4", PORT)
        if err != nil {
                fmt.Println(err)
                return
        }

        connection, err := net.ListenUDP("udp4", s)
        if err != nil {
                fmt.Println(err)
                return
        }

        defer connection.Close()
        buffer := make([]byte, 1024)
        rand.Seed(time.Now().Unix())

        for {
                n, addr, err := connection.ReadFromUDP(
                fmt.Print("-> ", string(buffer[0:n-1]))

                if strings.TrimSpace(string(buffer[0:n]
                        fmt.Println("Exiting UDP server
                        return
                }

                data := []byte(strconv.Itoa(random(1, 1
                fmt.Printf("data: %s\n", string(data))
                _, err = connection.WriteToUDP(data, ad
                if err != nil {
```

```
54                              fmt.Println(err)
55                              return
56                      }
57              }
58      }
59
```

This file creates the `main` package, which declares the `main()` function. The function will use the imported packages to create a UDP server.

The `main()` function gathers command line arguments in the `arguments` variable and includes error handling.

The `net.ListenUDP()` function tells the application to listen for incoming UDP connections, which are served inside the `for` loop. This is the function call that makes the program a UDP server.

The `ReadFromUDP()` and `WriteToUDP()` functions are used to read data from a UDP connection and write data to a UDP connection, respectively. A byte slice is stored in the `data` variable and used to write the desired data. The `buffer` variable also stores a byte slice and is used to read data.

Since UDP is a stateless protocol, each UDP client is served and then the connection closes automatically. The UDP server program will only exit when it receives the `STOP` keyword from a UDP client. Otherwise, the server program will continue to wait for more UDP connections from other clients.

Test the UDP Client and Server

You can now test your UDP client and server. You will need to execute the UDP server first so that the UDP client has somewhere it can connect to.

1. Run your UDP server. From the directory containing the `udpS.go` file, run the following command:

   ```
   go run udpS.go 1234
   ```

   The server will listen on port number `1234`. You will not see any output as a result of this command.

2. Open a second shell session to execute the UDP client and to interact with the UDP server. Run the following command:

   ```
   go run udpC.go 127.0.0.1:1234
   ```

3. You will see a `>>` prompt waiting for you to enter some text. Type in `Hello!` to receive a response from the UDP server:

   ```
   Hello!
   ```

   You should see a similar output:

   ```
   The UDP server is 127.0.0.1:1234
   >> Hello!
   Reply: 82
   ```

4. Send the `STOP` command to exit the UDP client and server:

   You should see a similar output on the client side:

   ```
   >> STOP
   Exiting UDP client!
   ```

The output on the UDP server side will be as follows:

```
-> STOP
Exiting UDP server!
```

Create a Concurrent TCP Server

This section demonstrates the implementation of a concurrent TCP server. The benefit of a concurrent TCP server is that it can serve multiple clients. In Go, this is accomplished by creating a separate Goroutine to serve each TCP client.

The example TCP server keeps a running count of the number of TCP clients it has served so far. The counter increases by one each time a new TCP client connects to the TCP server. The current value of that counter is returned to each TCP client.

1. In your current working directory, create a file named `concTCP.go` with the following content:

File: ./concTCP.go

```
package main

import (
        "bufio"
        "fmt"
        "net"
        "os"
        "strconv"
        "strings"
)

var count = 0

func handleConnection(c net.Conn) {
        fmt.Print(".")
        for {
                netData, err := bufio.NewReader(c).Read
                if err != nil {
                        fmt.Println(err)
                        return
                }

                temp := strings.TrimSpace(string(netDat
                if temp == "STOP" {
                        break
                }
                fmt.Println(temp)
                counter := strconv.Itoa(count) + "\n"
                c.Write([]byte(string(counter)))
        }
        c.Close()
}

func main() {
        arguments := os.Args
        if len(arguments) == 1 {
                fmt.Println("Please provide a port numb
                return
```

```
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()

    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
            return
        }
        go handleConnection(c)
        count++
    }
}

```

This file creates the main package, which declares the `handleConnection()` and `main()` functions.

The `main()` function will use the imported packages to create a concurrent TCP server. It gathers command line arguments in the `arguments` variable and includes error handling.

Each TCP client is served by a separate Goroutine that executes the `handleConnection()` function. This means that while a TCP client is served, the TCP server is free to interact with more TCP clients. TCP clients are connected using the `Accept()` function.

Although the `Accept()` function can be executed multiple times, the `net.Listen()` function needs to be executed only once. For this reason the `net.Listen()` function remains outside of the `for` loop.

The `for` loop in the `main()` function is endless because TCP/IP servers usually run nonstop. However, if the `handleConnection()` function receives the `STOP` message, the Goroutine that runs it will exit and the related TCP connection will close.

Test the Concurrent TCP Server

In this section, you will test the concurrent TCP server using the netcat command line utility.

1. Run your concurrent TCP server. From the directory containing the `concTCP.go` file, run the following command:

   ```
   go run concTCP.go 1234
   ```

   The command creates a TCP server that listens on port number `1234`. You can use any port number, however, ensure it is not already in use and that you have the required privileges. Reference the list of well-known TCP and UDP ports, if needed.

2. Use netcat to establish a connection with the TCP server. By default, netcat will establish a TCP connection with a remote host on the specified port number.

   ```
   nc 127.0.0.1 1234
   ```

3. After issuing the previous command, you will not see any change in your output. Type `Hello!` to send a packet to the TCP server:

```
Hello!
```

The TCP server will return the number of current client connections as its response. Since this is your first connection established with the TCP server, you should expect an output of `1`.

```
Hello!
1
```

If you'd like, you can open a new shell session and use netcat to establish a second connection with the TCP server by repeating Step 2. When you send the server a second `Hello!`, you should receive a response of `2` this time.

4. You can also connect to the TCP server using the TCP client you created in the Create the TCP Client section of the guide. Ensure you are in the directory containing the `tcpC.go` file and issue the following command:

```
go run tcpC.go 127.0.0.1:1234
```

5. You will see a `>>` prompt waiting for you to enter some text. Type in `Hello!` to receive a response from the TCP server:

```
Hello!
```

You should see a similar output indicating `3` client connections:

```
>> Hello!
->: 3
```

6. Send the `STOP` command to exit the TCP client:

You should see a similar output on the client:

```
>> STOP
->: TCP client exiting...
```

The output on the TCP server side will be as follows:

```
.Hello!
.Hello!
.Hello!
```

> **Note**
>
> From the shell session running the TCP server, type **CTRL-c** to interrupt program execution and then, **CTRL-D** to close all client connections and to stop the TCP server.

More Information

You may wish to consult the following resources for additional information on this topic. While these are provided in the hope that they will be useful, please note that we cannot vouch for the accuracy or timeliness of externally hosted materials.

Go

This page was originally published on Wednesday, June 26, 2019.

NETWORKING

Your Feedback Is Important

Let us know if this guide was helpful to you.

Provide Feedback

Join the conversation.

Read other comments or post your own below. Comments must be respectful, constructive, and relevant to the topic of the guide. Do not post external links or advertisements. Before posting, consider if your comment would be better addressed by contacting our Support team or asking on our Community Site.