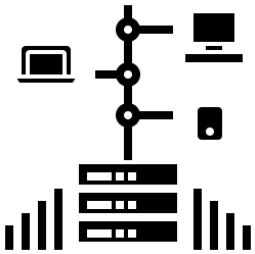
A TCP Server

This lesson introduces TCP servers and provides an implementation and detailed explanation of a TCP server and client in Go.

WE'LL COVER THE FOLLOWING

Go is very usable for writing web applications. Making HTML-screens with strings or templating is a good way to write apps that need a graphical interface.

A client-server application



We will develop a simple client-server application using the **TCP-protocol** and the goroutine paradigm from Chapter 12. A (web) server application has to respond to requests from many clients simultaneously. In Go, for every client-request, a goroutine is spawned to handle the request. We will need the package `net` for networking communication functionality. It contains methods for working with **TCP/IP** and **UDP** protocols, domain name resolution, and so on.

The server side

The server-code resides in its own program as follows:

```
package main
import (

"fmt"
"net"
)

func main() {
  fmt.Println("Starting the server ...")
  // create listener:
  listener, err := net.Listen("tcp", "0.0.0.0:3001")
  if err != nil {
    fmt.Println("Error listening", err.Error())
    return // terminate program
  }
  // listen and accept connections from clients:
  for {
    conn, err := listener.Accept()
    if err != nil {
      return // terminate program
    }
    go doServerStuff(conn)
  }
}

func doServerStuff(conn net.Conn) {
  for {
    buf := make([]byte, 512)
    _, err := conn.Read(buf)
    if err != nil {
      fmt.Println("Error reading", err.Error())
      return // terminate program
    }
    fmt.Printf("Received data: %v", string(buf))
  }
}
```

Server

In `main()`, we make a `net.Listener` variable listener, which is the basic function of a server: to listen for and accept incoming client requests (on IP-address **0.0.0.0** on port **3001** via the TCP-protocol). This `Listen()` function can return a variable `err` of type error. The waiting for client requests is performed in an infinite for-loop with `listener.Accept()`. A client request makes a connection variable `conn` of type `net.Conn`. On this connection, a separate goroutine `doServerStuff()` is started which reads the incoming data in a buffer of size 512 bytes and outputs them on the server terminal; when all the data from the client has been read, the goroutine stops. For each client, a separate goroutine is created. The server-code must be executed before any client can run.

> **Remark**: If you're running locally, then replace **line 10** with `listener, err := net.Listen("tcp", "localhost:50000")`

The code for the client is in a separate file as follows:

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import (
"fmt"
"net"
)

func main() {
  fmt.Println("Starting the server ...")
  // create listener:
  listener, err := net.Listen("tcp", "0.0.0.0:3001")
  if err != nil {
    fmt.Println("Error listening", err.Error())
    return // terminate program
  }
  // listen and accept connections from clients:
  for {
    conn, err := listener.Accept()
    if err != nil {
      return // terminate program
    }
    go doServerStuff(conn)
  }
}

func doServerStuff(conn net.Conn) {
  for {
    buf := make([]byte, 512)
    _, err := conn.Read(buf)
    if err != nil {
      fmt.Println("Error reading", err.Error())
      return // terminate program
    }
    fmt.Printf("Received data: %v", string(buf))
  }
}
```

Press the **RUN** button and wait for the server to get started. Then, open 2 or 3 separate console-windows. In each window, a client process is started by performing the following steps for each separate terminal:

- Type `cd usr/local/go/src` and press ENTER.
- Type `go run client.go` and press ENTER.

> **Remark:** If you're running it locally, then only perform the second step: `go run client.go`

Running locally

Of course, the server must be started first. If it is not listening, it can't be dialed by a client. If a client process would start without a server listening, the client stops with the following error message: `Error dialing dial tcp :3001: getsockopt: connection refused`.

Open a command prompt in the directory where the server and client executables are, type **server.exe** (or just server) on Windows, and **./server** on Linux and ENTER.

The following message appears in the console: Starting the server ... This process can be stopped with CTRL/C. The error: `Error listening listen tcp :3001: bind: Only one usage of each socket address (protocol/network address/port) is normally permitted` means that port 3001 is already occupied on your system. Change the port number to 3002, for example, compile and start the server again.

Here is some output of the server (after removing the empty space from the 512-byte string):

```
Starting the server ...

Received data: IVO says: Hi Server, what's up?
Received data: CHRIS says: Are you a busy server?
Received data: MARC says: Don't forget our appointment tomorrow!
```

The client establishes a connection with the server through `net.Dial`. It receives input from the keyboard `os.Stdin` in an infinite loop until **Q** is entered. Notice the trimming of **\n** and **\r** (both only necessary on Windows). The trimmed input is then transmitted to the server via the `Write` method of the connection.

When a client enters `Q` and stops, the server outputs the following message:

`Error reading read EOF`.

The `net.Dial` function is one of the most important functions of networking. When you Dial into a remote system, the function returns a `Conn` interface type, which can be used to send and receive information. The function `Dial` neatly abstracts away the network family and transport. So, `IPv4` or `IPv6`, `TCP` or `UDP` can all share a common interface. Dialing a remote system on port **80** over TCP, then UDP and lastly TCP over IPv6 looks like this:

```
package main
import (
  "fmt"
  "net"
  "os"
)

func main() {
  conn, err:= net.Dial("tcp", "192.0.32.10:80") // tcp ipv4
  checkConnection(conn, err)
  conn, err = net.Dial("udp", "192.0.32.10:80") // udp
  checkConnection(conn, err)
  conn, err = net.Dial("tcp", "[2620:0:2d0:200::10]:80") // tcp ipv6
  checkConnection(conn, err)
}

func checkConnection(conn net.Conn, err error) {
  if err!= nil {
    fmt.Printf("error %v connecting!")
    os.Exit(1)
  }
  fmt.Println("Connection is made with %v", conn)
}
```

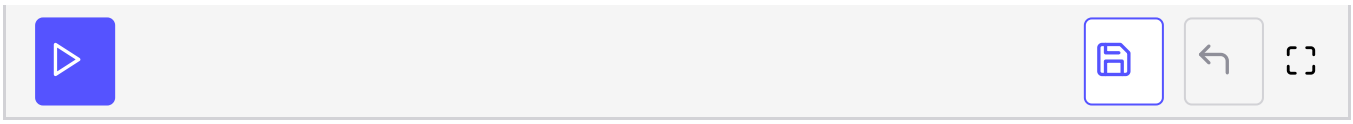
The following program is another illustration of the use of the `net` package for opening, writing to and reading from a `socket`:

```
package main
import (
"fmt"
"net"
"io"
)

func main() {
  var (
    host = "www.apache.org"
    port = "80"
    remote = host + ":" + port
    msg string = "GET / \n"
    data = make([]uint8, 4096)
    read = true
    count = 0
```

```
  )
  // create the socket
  con, err := net.Dial("tcp", remote)

  // send our message. an HTTP GET request in this case
  io.WriteString(con, msg)
  // read the response from the webserver
  for read {
    count, err = con.Read(data)
    read = (err == nil)
    fmt.Printf(string(data[0:count]))
  }
  con.Close()
}
```



Socket

The following version has a much better code structure and improves on our first example of a TCP-server in many ways, using only some 80 lines of code!

```
package main
import (
        "flag"
        "net"
        "fmt"
        "syscall"
)

const maxRead = 25

func main() {
        flag.Parse()
        if flag.NArg() != 2 {
                panic("usage: host port")
        }
        hostAndPort := fmt.Sprintf("%s:%s", flag.Arg(0), flag.Arg(1))
        listener := initServer(hostAndPort)
        for {
                conn, err := listener.Accept()
                checkError(err, "Accept: ")
                go connectionHandler(conn)
        }
}

func initServer(hostAndPort string) *net.TCPListener {
        serverAddr, err := net.ResolveTCPAddr("tcp", hostAndPort)
        checkError(err, "Resolving address:port failed: `" + hostAndPort + "'")
        listener, err := net.ListenTCP("tcp", serverAddr)
        checkError(err, "ListenTCP: ")
        println("Listening to:    ", listener.Addr().String())
        return listener
}

func connectionHandler(conn net.Conn) {
        connFrom := conn.RemoteAddr().String()
```

```
        println("Connection from: ", connFrom)
        sayHello(conn)
        for {

                var ibuf []byte = make([]byte, maxRead + 1)
                length, err := conn.Read(ibuf[0:maxRead])
                ibuf[maxRead] = 0 // to prevent overflow
                switch err {
                case nil:
                   handleMsg(length, err, ibuf)
                case syscall.EAGAIN:  // try again
                   continue
                default:
                            goto DISCONNECT
                }
        }

DISCONNECT:
        err := conn.Close()
        println("Closed connection: ", connFrom)
        checkError(err, "Close: ")
}

func sayHello(to net.Conn) {
        obuf := []byte{'L', 'e', 't', '\'', 's', ' ', 'G', 'O', '!', '\n'}
        wrote, err := to.Write(obuf)
        checkError(err, "Write: wrote " + string(wrote) + " bytes.")
}

func handleMsg(length int, err error, msg []byte) {
        if length > 0 {
                print("<", length, ":")
                for i := 0; ; i++ {
                        if msg[i] == 0 {
                                break
                        }
                        fmt.Printf("%c", msg[i])
                }
                print(">")
        }
}

func checkError(error error, info string) {
        if error != nil {
                panic("ERROR: " + info + " " + error.Error())  // terminate program
        }
}
```

Simple TCP Server

Improvements

What are the improvements?

- The server address and port are not hard-coded in the program, but are given on the command-line and read in via the `flag` package. Note the use of `flag.NArg()` to signal when the expected 2 arguments are not given:

given.

```
if flag.NArg() != 2 {
  panic("usage: host port")
}
```

The arguments are then formatted into a string via the `fmt.Sprintf-function`:

```
hostAndPort := fmt.Sprintf("%s:%s", flag.Arg(0), flag.Arg(1))
```

- The server address and port are controlled in the function `initServer` through `net.ResolveTCPAddr`, and this function returns a `*net.TCPListener`.
- For each connection, the function `connectionHandler` is started as a goroutine. This begins by showing the address of the client with `conn.RemoteAddr()`.
- It writes a promotional Go-message to the client with the function `conn.Write`.
- It reads from the client in chunks of 25 bytes and prints them one by one; in case of an error in the read, the infinite read-loop is left via the default switch clause and that client-connection is closed. In case the OS issues an `EAGAIN` error, the read is retried.
- All error-checking is refactored in a separate function `checkError`, which issues a panic with a contextual error-message in the case of an error occurring.

Start this server program on the command-line with: `simple_tcp_server localhost 50000` and start a few clients with **client.go** in separate command-windows. A typical server output from 2 client-connections follows, where we see that the clients each have their own address:

```
E:\Go\GoBoek\code examples\chapter 14>simple_tcp_server localhost 50000
Listening to: 127.0.0.1:50000
Connection from: 127.0.0.1:49346
<25:Ivo says: Hi server, do y><12:ou hear me ?>
Connection from: 127.0.0.1:49347
<25:Marc says: Do you remembe><25:r our first meeting serve><2:r?>
```

`net.Error`

The `net` package returns errors of type error, following the usual convention, but some of the error implementations have additional methods defined by the `net.Error` interface:

```
package net

type Error interface {
  Timeout() bool // Is the error a timeout?
  Temporary() bool // Is the error temporary?
  ...
}
```

Client code can test for a `net.Error` with a type assertion and then distinguish transient network errors from permanent ones. For instance, a web crawler might sleep and retry when it encounters a temporary error and give up otherwise.

```
// in a loop - some function returns an error err
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
  time.Sleep(1e9)
  continue // try again
}
if err != nil {
  log.Fatal(err)
}
```

To make communication interactive between the client and server, it's a good idea to use a web server, and log the responses as HTML forms that are much more readable and responsive. In the next lesson, you'll learn how to devise a web server.