Mr.Waggel's blog (/)
Some tutorials, code-snippets and helpful things

Open navigation

Golang transfer a file over a TCP socket (/post/golang-transfer-a-file-over-a-tcp-socket/)

21 May, 2016

In this tutorial I'll demonstrate how to send a file in Go over a TCP connection using a server that sends the file and a client that receives it, I'll try to go into detail as much as possible.

You can download the source code here (/blog_files/golang_tcp_file_source.zip).

Comments and remarks are more than welcome.

The server

Introduction

The server will work in the following fashion

1. Wait until a client connects
2. Send the size of the file that will be send
   This is needed so the client knows how long he should read from the server
3. Send the name of the file
4. Send the file in (file size/buffer size) amount of chunks
5. Close connection upon completion

Global space

First of import the following packages

```
import (
        "fmt"
        "io"
        "net"
        "os"
        "strconv"
)
```

Define the following constant

```
const BUFFERSIZE = 1024
```

This constant can be anything from 1 to 65495, because the TCP package can only contain up to 65495 bytes of payload. It will define how big the chunks are of the file that we will send in bytes.

Also note that this constant should be the same on your client.

The main() function

Here we will be creating a TCP listener, it will listen on the given port to incoming connections. And when a connection is made start a go routine to handle that connection.

```
func main() {
	server, err := net.Listen("tcp", "localhost:27001")
	if err != nil {
		fmt.Println("Error listetning: ", err)
		os.Exit(1)
	}
	defer server.Close()
	fmt.Println("Server started! Waiting for connections...")
	for {
		connection, err := server.Accept()
		if err != nil {
			fmt.Println("Error: ", err)
			os.Exit(1)
		}
		fmt.Println("Client connected")
		go sendFileToClient(connection)
	}
}
```

It's pretty straight forward, 'net.Listen' starts listening at the the given host or ip and port. The 'defer' will make sure whenever the main function goes out of scope it will call the 'server.Close()' function.

The for loop works as following; it waits until somebody connects, that's what the 'server.Accept()' does, this function blocks the program until a client connects. When this happens it spawns a go routine that will handle the connection, in this case the function 'sendFileToClient()' will handle it and return back to the beginning of the loop.

The sendFileToClient() function

This is the function that 'go' will call, it takes a 'net.Conn' as argument.

```
func sendFileToClient(connection net.Conn) {
	fmt.Println("A client has connected!")
	defer connection.Close()
	file, err := os.Open("dummyfile.dat")
	if err != nil {
		fmt.Println(err)
		return
	}
	fileInfo, err := file.Stat()
	if err != nil {
		fmt.Println(err)
		return
	}
	fileSize := fillString(strconv.FormatInt(fileInfo.Size(), 10), 10)
	fileName := fillString(fileInfo.Name(), 64)
	fmt.Println("Sending filename and filesize!")
	connection.Write([]byte(fileSize))
	connection.Write([]byte(fileName))
	sendBuffer := make([]byte, BUFFERSIZE)
	fmt.Println("Start sending file!")
	for {
		_, err = file.Read(sendBuffer)
		if err == io.EOF {
			break
		}
		connection.Write(sendBuffer)
	}
	fmt.Println("File has been sent, closing connection!")
	return
}
```

It opens the file, in this case "dummyfile.dat". You can enter either a relational path to the file or an absolute path to the file. If you only enter a "file.extension" it will be treated as a relational path and look in the directory where the executable of the current program is.

It then reads out the information of the file with '.Stat()'. Create strings with a user defined function (scroll down for it). This function will fill in the rest of the bytes so it will match the fixed length of what the client want to read. This is just how TCP programming works, you define upfront how long the reader (client) should read.

For example if you do not fill up the bytes the client will keep on waiting;

Server sends the file name, "afile.dat", 9 bytes long. However since file names can differ and aren't always 9 bytes long we program the client to have a margin on this so it it reads, lets say 64 bytes.

Server sends the 9 bytes, the client receives them but keeps waiting for the other 55 bytes that it is expecting and will never move further until those are filled in.

So if you fill the file name with a filler (I used ':' here since it's an illegal character for file names under windows and linux) to become this:

```
afile.dat::::::::::::::::::::::::::::::::::::::::::::::::::::::: (64 bytes long)
```

So that's what the 'fillString()' function does, it fills it up. This should also be done on the file size. The client will strip it again after it received it. For the file size I first convert it with base ten to a string before passing it to the function.

Afterwards it will write the file size first and then the file name with the 'net.Conn.Write()' function.

Then we create a buffer where the server will read the file in chunk by chunk, the buffer will be the size of the defined constant.

The for loop will read out a chunk of the file with the 'file.Read()' function and write it in the passed buffer, the chunk is the size of the passed buffer. Send the the buffer chunk with 'net.Conn.Write()' and. This will be repeated until the end of the file is achieved and will break out of the loop afterwards. The connection and file will be closed when the function returns because of the stated defers.

Custom defined fillString() function

```
func fillString(retunString string, toLength int) string {
	for {
		lengtString := len(retunString)
		if lengtString < toLength {
			retunString = retunString + ":"
			continue
		}
		break
	}
	return retunString
}
```

This is the function that fills up, it just loops until enough ":" are added.

The client

The client works like the server, only it receives the information and data.

1. Connect to the server
2. Read the file size from the server
3. Read the file name from the server
4. Read the file from the server, chunk by chunk
5. Close connection

Global space

Import the following packages

```
import (
        "fmt"
        "io"
        "net"
        "os"
        "strconv"
        "strings"
)
```

Define the constant as you did in the server part, this must have the same value as the server.

```
const BUFFERSIZE = 1024
```

The main() and only function

```
func main() {
	connection, err := net.Dial("tcp", "localhost:27001")
	if err != nil {
		panic(err)
	}
	defer connection.Close()
	fmt.Println("Connected to server, start receiving the file name and file size")
	bufferFileName := make([]byte, 64)
	bufferFileSize := make([]byte, 10)

	connection.Read(bufferFileSize)
	fileSize, _ := strconv.ParseInt(strings.Trim(string(bufferFileSize), ":"), 10, 64)

	connection.Read(bufferFileName)
	fileName := strings.Trim(string(bufferFileName), ":")

	newFile, err := os.Create(fileName)

	if err != nil {
		panic(err)
	}
	defer newFile.Close()
	var receivedBytes int64

	for {
		if (fileSize - receivedBytes) < BUFFERSIZE {
			io.CopyN(newFile, connection, (fileSize - receivedBytes))
			connection.Read(make([]byte, (receivedBytes+BUFFERSIZE)-fileSize))
			break
		}
		io.CopyN(newFile, connection, BUFFERSIZE)
		receivedBytes += BUFFERSIZE
	}
	fmt.Println("Received file completely!")
}
```

First we open a connection to the server with 'net.Dial', notice that the first argument should be a lowercase string. Tell the program with defer statement it should close the connection once done.

We create two buffers, one for the name and the other one for the file size, the buffers must have the same size as the server will write them to you.

We read out the file size into it's buffer, define a variable containing the size, to achieve this we first convert the buffer to a string, then we trim the excessive ':' 'using strings.Trim()', convert it to a 64 bit integer with 'strconv.ParseInt()'.

We do the same for the file name, only we don't convert it to a integer now.

We create the file with the given file name, if you don't specify a path (absolute or relative) it will be created in the folder where the current program is running from. State the defer to close the file once done.

Initialize a 64 bit integer that will keep count of how many bytes we have received so far, so we can tell when to stop reading the chunks from the server.

The for loop starts the file reading with 'io.CopyN()' and incrementing the counter with the defined constant.

The if statement states "when the received file size minus the received bytes so far, is smaller than the constant buffer size" we do the following; only read out the remaining bytes that are smaller than the buffer size. This is because if the server only has to send 4 bytes for example, it will come in a buffer of 1024, rendering 1020 bytes being empty and added to the file. Which may lead to corruption or unwanted behavior, and a hash check (/post/generate-md5-hash-of-a-file/) to be false. So when true, only write the remaining bytes.

And as you may have noticed in the if scope we read twice, the first read with 'io.Copy()' is for the file, and the other, 'connection.Read()', is to empty the network buffer. And this has a good reason, if you were not to do this and keep on doing other reads on that connection it will first read out what's left in there.

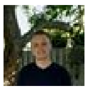
For example, if you were to receive two files on the same connection. And you didn't clear up the network buffer from the first file that you received, and it contained over 374 empty bytes (nulls). The file size of the next read will be 0 and the name will be 64 zeros and the first 300 bytes of your new file to be zero also.

go (/tag/go/) golang (/tag/golang/) tcp (/tag/tcp/) file (/tag/file/) byte (/tag/byte/) bytes (/tag/bytes/) buffer (/tag/buffer/) connection (/tag/connection/) send (/tag/send/) receive (/tag/receive/) example (/tag/example/)

Like Share Sign Up to see what your friends like.

1 Comment Sort by Top

Add a comment...


Richard Lindhout · Software ontwikkelaar at Profects
Thanks a lot for your detailed blog post! However I found a little bug which can be easily triggered if the last slice of bytes is in the end of the file.

The end of line is reached at the last chunk so the last chunk will not be send.

```
for {
_, readError := file.Read(buffer)

if readError != io.EOF {
log.Println(err)
break... See More
```

Like · Reply · 11w · Edited

Facebook Comments Plugin