

Jerzy Jaskuc

Student Number: 17341645

Protocol Design and Implementation

Introduction

The main task was to design and implement a Web Proxy server, which would reside locally.

The program should be able to:

1. Respond to HTTP & HTTPS requests, and should display each request on a management console. It should forward the request to the Web server and relay the response to the browser.
2. Handle Websocket connections.
3. Dynamically block selected URLs via the management console.
4. Efficiently cache requests locally.
5. Handle multiple requests simultaneously by implementing a threaded server.

Implementation and Design

The application is written in **GoLang** and consists of two programs. One is responsible for proxy server and second one handles management console.

Web Proxy

Web Proxy handles HTTPS and HTTP requests separately.

- **HTTP requests** are listened to on port number “:42069” and are handled by a replication method.

First we check if address is not blocked.

```
//check if we blocked this address
for blockedURL, isBlocked := range blockedURLs {
    if strings.Contains(req.URL.String(), blockedURL) && isBlocked {
        w.WriteHeader(http.StatusLocked)
        log.Printf("Request blocked according to administrator policy.")
        return
    }
}
```

In case it is, I decided to use Status Code 423 (Locked) as an indication to the client.

Then we check if the response is cached and if the entry is not yet expired, using “Date” field, combined with “max-age” in “Cache-Control” field of the header.

If everything is up to date, we just supply the cached response and return.

In case it's not in cache or is expired, we proceed to normal procedure.

First we just clone and forward a request:

```
// clone and forward
proxyReq, err := http.NewRequest(req.Method, url.String(), req.Body)
if err != nil {
    log.Print(err)
}
proxyReq.Header = req.Header
resp, err := client.Do(proxyReq)
if err != nil {
    log.Printf("Error sending forwarding request %v",err)
}
log.Printf("forwarded the request and received the response")
```

We need clone the request in order to create a connection between us and the server, not client and the server.

After receiving the response from the server, we check if we can cache it, as some of the flags can indicate that we can't.

```
//conditionally cache the response
canCache := true
for head, values := range resp.Header {
    if head == "Cache-Control" {
        for _, value := range values {
            if value == "no-cache" || value == "no-store" || value == "private" {
                canCache = false
                log.Printf("Response not cache-able")
            }
        }
    }
}
```

To not bother too much with revalidation, the program does not cache if no-cache flag is present.

Also, since our cache is shared and not private (private is on the browser), we do not cache under private flag either.

Now, if we can cache, we save it in a map

```
// cache is shared
var httpCache = map[string]cachedResponse{}
```

Where key is URL and cached response is a struct.

```
type cachedResponse struct {
    response *http.Response //
    responseBody []byte // prc
}
```

We need to store body separately, since it's a stream of Bytes, and It's needed to be read and closed to be reused later.

Once we cached (or not) our response, we just clone it and forward to the client.

- **HTTPS requests** are not cached, since they use TLS and it would be man-in-the-middle attack if we tried to intercept it and replicate. They are listened to on ":42070" port number. Therefore, we use tunneling method instead. As in HTTP, we check if URL is blocked by us, if not we proceed.

Since first request should be CONNECT request, we check if it is

```
if req.Method != http.MethodConnect {
    http.Error(w, http.StatusText(http.StatusMethodNotAllowed), http.StatusMethodNotAllowed)
    return
}
log.Printf("Dialing...")
serverConn, err := net.DialTimeout("tcp", req.Host, time.Second*10)
if err != nil {
    http.Error(w, err.Error(), http.StatusServiceUnavailable)
    return
}
```

And if it actually is, then we Dial the target host to establish TCP connection. We give it 10 sec before timing out.

After establishing is done, we “Hijack” the connection

```
// Hijacking connection leaves it to us to manage the connection manually
hijacker, ok := w.(http.Hijacker)
if !ok {
    http.Error(w, "Could not Hijack the connection", http.StatusInternalServerError)
    return
}
```

That means that we are now responsible for manually closing the connection when we are done with it.

With that, the setup for tunneling is done and we can start data transfer

```
go tunnel(serverConn, clientConn)
go tunnel(clientConn, serverConn)
log.Printf("Finished tunnelling")
}
}

func tunnel(dest io.WriteCloser, src io.ReadCloser) {
    defer func() { _ = dest.Close() }()
    defer func() { _ = src.Close() }()
    _, _ = io.Copy(dest, src)
}
```

Server to client and client to server work in two separate threads, and so transfers can be done both ways at the same time. “*io.Copy*” copies from source to destination, and when it hits EOF, it stops, that’s when we manually close the connections and done with HTTPS request.


Management Console



Console works separately from main proxy. It reads an input and then, depending if it’s block or unblock command, sends a HTTP Post request to our Proxy on either port “:420” for block requests or “:421” for unblock. The body of our POST request is the URL to block/unblock. Once sent, we can take another input.



```
for scanner.Scan() {
    command := scanner.Text()
    if strings.HasPrefix(command, "block") {
        fields := strings.Fields(command)
        b := []byte(fields[1])
        resp, err := http.Post("http://localhost:420", "text/plain", bytes.NewBuffer(b))
        if err != nil {
            log.Printf("Failed blocking the address: %v", err)
        } else {
            defer resp.Body.Close()
            log.Printf("block message delivered")
        }
    } else if strings.HasPrefix(command, "unblock") {
        fields := strings.Fields(command)
        b := []byte(fields[1])
        resp, err := http.Post("http://localhost:421", "text/plain", bytes.NewBuffer(b))
        if err != nil {
            log.Printf("Failed unblocking the address: %v", err)
        } else {
            defer resp.Body.Close()
            log.Printf("Unblock message delivered")
        }
    } else {
        fmt.Println("Invalid input, please try again")
    }
}
```

Timings

Example.com un-cached:


Status	Method	Domain	File
200	GET	 example.com	/
	GET	example.com	favicon.ico


	2 requests	1.23 KB / 1 KB transferred	Finish: 239 ms	DOMContentLoaded: 212 ms	load: 218 ms
200	GET	 example.com	/		
	GET	example.com	favicon.ico		
example.com					

	2 requests	1.23 KB / 0.98 KB transferred	Finish: 226 ms	DOMContentLoaded: 208 ms	load: 215 ms
200	GET	 example.com	/		
	GET	example.com	favicon.ico		

	2 requests	1.23 KB / 1 KB transferred	Finish: 234 ms	DOMContentLoaded: 204 ms	load: 210 ms
---	------------	----------------------------	----------------	--------------------------	--------------

Example.com cached:

Status	Method	Domain	File
200	GET	 example.com	/
	GET	example.com	favicon.ico




2 requests



1.23 KB / 0.98 KB transferred


Finish: 15 ms

DOMContentLoaded: 9 ms



load: 15 ms















Status	Method	Domain	File
200	GET	 example.com	/
	GET	example.com	favicon.ico

	2 requests	1.23 KB / 0.98 KB transferred	Finish: 24 ms	DOMContentLoaded: 14 ms	load: 22 ms
200	GET	 example.com	/		
	GET	example.com	favicon.ico		

	2 requests	1.23 KB / 0.98 KB transferred	Finish: 30 ms	DOMContentLoaded: 7 ms	load: 13 ms
---	------------	-------------------------------	---------------	------------------------	-------------

Tcd commons un-cached:

Status	Method	Domain	File
200	GET	 www.tcdlife.ie	screen.css
200	GET	 www.tcdlife.ie	favicon.ico
200	GET	 www.tcdlife.ie	montage.jpg
200	GET	 www.tcdlife.ie	title-scholars.png
200	GET	 www.tcdlife.ie	crest.png
200	GET	 www.tcdlife.ie	tile.png

	9 requests	81.04 KB / 83.12 KB transferred	Finish: 364 ms	DOMContentLoaded: 131 ms	load: 365 ms
Status	Method	Domain	File		
	GET	 www.tcdlife.ie	screen.css		
	GET	 www.tcdlife.ie	favicon.ico		
	GET	 www.tcdlife.ie	montage.jpg		
	GET	 www.tcdlife.ie	title-scholars.png		
	GET	 www.tcdlife.ie	crest.png		
	GET	 www.tcdlife.ie	tile.png		
	9 requests	81.04 KB / 83.12 KB transferred	Finish: 402 ms	DOMContentLoaded: 133 ms	load: 404 ms

Tcd commons cached:

Status	Method	Domain	File
200	GET	www.tcdlife.ie	screen.css
200	GET	www.tcdlife.ie	favicon.ico
200	GET	www.tcdlife.ie	montage.jpg
200	GET	www.tcdlife.ie	title-scholars.png
200	GET	www.tcdlife.ie	crest.png
200	GET	www.tcdlife.ie	tile.png
9 requests 81.04 KB / 83.12 KB transferred Finish: 312 ms DOMContentLoaded: 96 ms load: 351 ms			
Status	Method	Domain	File
200	GET	www.tcdlife.ie	screen.css
200	GET	www.tcdlife.ie	favicon.ico
200	GET	www.tcdlife.ie	montage.jpg
200	GET	www.tcdlife.ie	title-scholars.png
200	GET	www.tcdlife.ie	crest.png
200	GET	www.tcdlife.ie	tile.png
9 requests 81.04 KB / 83.12 KB transferred Finish: 320 ms DOMContentLoaded: 106 ms load: 337 ms			

Appendix

Proxy Code

```
import (
    "net"
    "net/http"
    "time"
    "io"
    "io/ioutil"
    "log"
    "crypto/tls"
    "strings"
    "strconv"
)

const dateFormat = "Mon, 02 Jan 2006 15:04:05 MST"

type cachedResponse struct {
    response *http.Response // most of the response to use the headers
    responseBody []byte // processed body of the response, since on receival it's
    streamed (can only store it in this form)
}

// cache is shared
var httpCache = map[string]cachedResponse{}

// list of blocked URL's
var blockedURLs = map[string]bool{}

func main() {
    s := &http.Server{
        Addr: ":42070",
        Handler: http.HandlerFunc(httpsRequestHandler),
        TLSNextProto: make(map[string]func(*http.Server, *tls.Conn,
http.Handler)), //Disable HTTP/2
    }
    go func() {
```

```

        log.Fatal(s.ListenAndServe())
    }()
    go func() {

        log.Fatal(http.ListenAndServe(":420",http.HandlerFunc(blockRequestHandler)))
    }()
    go func() {

        log.Fatal(http.ListenAndServe(":421",http.HandlerFunc(unblockRequestHandler)))
    }()
    log.Fatal(http.ListenAndServe(":42069",http.HandlerFunc(httpRequestHandler)))
}

// This function handles incoming requests and responds to them if needed
func httpRequestHandler(w http.ResponseWriter, req *http.Request) {
    log.Printf("received the http request")

    //check if we blocked this address
    for blockedURL, isBlocked := range blockedURLs {
        if strings.Contains(req.URL.String(), blockedURL) && isBlocked {
            w.WriteHeader(http.StatusLocked)
            log.Printf("Request blocked according to administrator policy.")
            return
        }
    }
    client := &http.Client{}
    url := req.URL
    url.Host = req.Host
    url.Scheme = "http"

    // take response from cache the response and send to client, if it's not expired
    and us there
    cachedResp, exists := httpCache[req.URL.String()]
    if exists {
        var unformattedDate string
        var maxAge int
        var err error
        for head, values := range cachedResp.response.Header {
            if head == "Cache-Control" {
                for _, value := range values {
                    if strings.Contains(value, "max-age") {
                        newVal := strings.Split(value, ",") //safety
                        guard for wrongly parsed/constructed headers
                        value = newVal[0]
                        maxAge, err = strconv.Atoi(value[8:])
                        if err != nil {
                            log.Fatalf("Failed converting max-age
value to integer: %v", err)
                        }
                    }
                }
            } else if head == "Date" {
                unformattedDate = values[0]
            }
        }
        formattedDate, err := time.Parse(dateFormat, unformattedDate)
        if err != nil {
            log.Fatalf("Failed formatting Date from header into variable: %v", err)
        }
        expiryTime := formattedDate.Add(time.Duration(maxAge)*time.Second)
        if expiryTime.After(time.Now()){
            cachedBody := cachedResp.responseBody

```

```

        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        log.Printf("Read the cached body")
        for head, values := range cachedResp.response.Header {
            for _, value := range values {
                w.Header().Add(head, value)
            }
        }
        w.Write(cachedBody)
        log.Printf("Supplied cached response")
        return
    } else {
        log.Printf("Response timed out, fetching new one")
    }
}

// clone and forward
proxyReq, err := http.NewRequest(req.Method, url.String(), req.Body)
if err != nil {
    log.Print(err)
}
proxyReq.Header = req.Header
resp, err := client.Do(proxyReq)
if err != nil {
    log.Printf("Error sending forwarding request %v", err)
}
log.Printf("forwarded the request and received the response")

body, err := ioutil.ReadAll(resp.Body)
if err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
}

//conditionally cache the response
canCache := true
for head, values := range resp.Header {
    if head == "Cache-Control" {
        for _, value := range values {
            if value == "no-cache" || value == "no-store" || value ==
"private" {
                canCache = false
                log.Printf("Response not cache-able")
            }
        }
    }
}
if canCache {
    toCache := cachedResponse{
        response : resp,
        responseBody : body,
    }
    httpCache[req.URL.String()] = toCache
    log.Printf("Response cached")
}

// clone the response and send to client
defer resp.Body.Close()
log.Printf("Read the body")
for head, values := range resp.Header {

```



```

        for _, value := range values {
            w.Header().Add(head, value)
        }
    }
    w.Write(body)
    log.Printf("Finished cloning")
}

func httpsRequestHandler(w http.ResponseWriter, req *http.Request) {
    log.Printf("received https request")
    //check if we blocked this address
    for blockedURL, isBlocked := range blockedURLs {
        if strings.Contains(req.URL.String(), blockedURL) && isBlocked {
            w.WriteHeader(http.StatusLocked)
            log.Printf("Request blocked according to administrator policy.")
            return
        }
    }
    if req.Method != http.MethodConnect {
        http.Error(w, http.StatusText(http.StatusMethodNotAllowed),
http.StatusMethodNotAllowed)
        return
    }
    log.Printf("Dialing...")
    serverConn, err := net.DialTimeout("tcp", req.Host, time.Second*10)
    if err != nil {
        http.Error(w, err.Error(), http.StatusServiceUnavailable)
        return
    }
    w.WriteHeader(http.StatusOK)
    // Hijacking connection leaves it to us to manage the connection manually
    hijacker, ok := w.(http.Hijacker)
    if !ok {
        http.Error(w, "Could not Hijack the connection",
http.StatusInternalServerError)
        return
    }
    log.Printf("Hijacking finished")
    clientConn, _, err := hijacker.Hijack()
    if err != nil {
        http.Error(w, err.Error(), http.StatusServiceUnavailable)
        return
    }
    // create a https transfer tunnel between server and the client through our
    proxy,
    // works both ways independently
    go tunnel(serverConn, clientConn)
    go tunnel(clientConn, serverConn)
    log.Printf("Finished tunnelling")
}

func tunnel(dest io.WriteCloser, src io.ReadCloser) {
    defer func() { _ = dest.Close() }()
    defer func() { _ = src.Close() }()
    _, _ = io.Copy(dest, src)
}

// This method handles the block command requests from console
func blockRequestHandler(w http.ResponseWriter, req *http.Request) {
    log.Printf("Received block request")
    body, err := ioutil.ReadAll(req.Body)
    if err != nil {

```

```

        log.Fatalf("Problem reading block request body: %v", err)
    }
    url := string(body)
    blockedURLs[url] = true
}

// This method handles the unblock command requests from console
func unblockRequestHandler(w http.ResponseWriter, req *http.Request) {
    log.Printf("Received unblock request")
    body, err := ioutil.ReadAll(req.Body)
    if err != nil {
        log.Fatalf("Problem reading unblock request body: %v", err)
    }
    url := string(body)
    blockedURLs[url] = false
}

```

Console

```

package main

import (
    "bytes"
    "bufio"
    "os"
    "log"
    "strings"
    "net/http"
    "fmt"
)

// this file implements the console, that is used to control the proxy
func main() {
    scanner := bufio.NewScanner(os.Stdin)
    fmt.Println("Welcome to proxy management console")
    fmt.Println("To block a website, write 'block: url'")
    fmt.Println("To unblock, write 'unblock: url'")
    for scanner.Scan() {
        command := scanner.Text()
        if strings.HasPrefix(command, "block") {
            fields := strings.Fields(command)
            b := []byte(fields[1])
            resp, err := http.Post("http://localhost:420", "text/plain",
bytes.NewBuffer(b))
            if err != nil {
                log.Printf("Failed blocking the address: %v", err)
            } else {
                defer resp.Body.Close()
                log.Printf("block message delivered")
            }
        } else if strings.HasPrefix(command, "unblock") {
            fields := strings.Fields(command)
            b := []byte(fields[1])
            resp, err := http.Post("http://localhost:421", "text/plain"
,bytes.NewBuffer(b))
            if err != nil {
                log.Printf("Failed unblocking the address: %v", err)
            } else {
                defer resp.Body.Close()
                log.Printf("Unblock message delivered")
            }
        }
    }
}

```

```
        } else {
            fmt.Println("Invalid input, please try again")
        }
    }

    if err := scanner.Err(); err != nil {
        log.Fatal(err)
    }
}
```