CSU34031 Advanced Telecommunications Assignment 1 - Web Proxy Server

Samuel Petit, petits@tcd.ie, 17333946

Overall Structure

The program starts by launching a thread which will listen for user input as well as a client that will listen for http requests. Then, any request received on that port will be directed to a method that I wrote called "onRequest".

This method will first check to see if the request comes from a blocked domain. If it does then a response 403 Forbidden is returned. Otherwise it will either start a connexion for https or simply create a http client and send the request. The response is then forwarded back to the original sender.

The only difference for https is that a connection needs to be formed before any content can be exchange which is why the "onRequest" method makes sure to redirect such requests to the method which creates such connections, it is called "httpsHandler".

Management Console

I made a function which constantly listens for user input which runs concurrently. The concurrent part is done quite simply in Go by using the keyword "go" in front of the function to execute, which essentially launches a really lightweight thread.

The actual function itself will listen for user input constantly, here is the list of commands it can execute:

- block < domain name >
- unblock < domain name >

If the user input does not begin with block or unblock then the command will not be supported and the program will then start to listen for the next input.

The actual domain blocking is done through an array of strings. When the program receives a request it will first make sure that the domain to send the request to is not in that array, if it is then the communication ends here and an error message is displayed, if not then the program resumes as usual. Adding a new blocked domain is as simple as appending it to the array, deleting it consists of finding the index of the domain to delete and to remove it from the array. This is done by copying over all of the elements except the one in question.

Caching Requests

I implemented caching by maintaining an map where the key is the dump of a request (i.e. its bytes representation) since go does not let you have http request objects as a key. The value is an instance of a struct I defined as "cacheItem". It contains 2 fields, the first one being the dump of the response (similar to the request, it is the byte representation of an http response instance), the second field is its expiry time.

It's worth noting that I decided here that in the case that the response headers does not contain a max-age attribute to define when the cache entry is supposed to expire it will be set by default to 24 hours by my program. (It must also not contain a no-cache header of course).

The expiry of a cache entry is checked on read. For example if the proxy tries to look for a specific request cached and it finds a response cached then before anything happens it makes sur that the expiry date has not expired. If it has expired then the cache entry is deleted and not used, the proxy will have to send the request again.

Simultaneous requests

Thankfully, Go deals with concurency quite well and the http server can serve multiple requests concurently.

Code

```
package main

import (
	"bufio"
	"bytes"
	"errors"
	"fmt"
	"io"
	"net"
	"net/http"
	"net/http/httputil"
	"os"
	"strconv"
	"strings"
	"time"
)

const BLOCK_CMD = "block "
const UNBLOCK_CMD = "unblock "

var blocked []string
var cache map[string]cacheItem = make(map[string]cacheItem)

type cacheItem struct {
	response   []byte
	expiryTime time.Time
}

func newCacheItem(res []byte, exp time.Time) cacheItem {
	return cacheItem{
		response:   res,
		expiryTime: exp,
	}
}

// helper function for testing which prints a slice
func printSlice(s []string) {
	fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
}

// Copies from src into dst & closes read for src & write for dst.
```

```
func transfer(dst, src *net.TCPConn) {
        _, status := io.Copy(dst, src)
        if status != nil {
                fmt.Printf("[ERROR] Copying to client failed - %s", status)
        }
        src.CloseRead()
        dst.CloseWrite()
}

// Https communication contains
func httpsHandler(w http.ResponseWriter, req *http.Request) {
        dest, err := net.Dial("tcp", req.Host)
        if err != nil {
                fmt.Printf(err.Error())
        }

        hijacker, ok := w.(http.Hijacker)
        if !ok {
                http.Error(w, "[Error] hijacking http failed",
http.StatusInternalServerError)
                return
        }

        client, _, err := hijacker.Hijack()
        if err != nil {
                http.Error(w, err.Error(), http.StatusServiceUnavailable)
        }

        fmt.Printf("[HANDLING CONNECT REQUEST] Host : %s\n", req.Host)
        client.Write([]byte("HTTP/1.0 200 OK\r\n\r\n"))

        // Send data
        destTCP, destOk := dest.(*net.TCPConn)
        clientTCP, clientOK := client.(*net.TCPConn)
        if destOk && clientOK {
                go transfer(destTCP, clientTCP)
                go transfer(clientTCP, destTCP)
        }
}

func checkCache(r *http.Request) (*http.Response, error) {
        // placeholder
        var res *http.Response

        // check is in cache
        if r.Method != http.MethodGet {
                return res, errors.New("Not a get")
        }
        dumpedRequest, err := httputil.DumpRequest(r, false)
        if err != nil {
                return res, errors.New("[ERROR] Could not check if request
is in cache - error while dumping request")
        }
        cachedResponse, ok := cache[string(dumpedRequest)]
        // request not in cache
        if !ok {
                return res, errors.New("Item not in cache")
        }

        // cache expired : remove it
        if cachedResponse.expiryTime.Sub(time.Now().Local()) < 0 {
                delete(cache, string(dumpedRequest))
                fmt.Printf("[INFO] Deleted expired cache\n")
                return res, errors.New("[ERROR] Cache entry expired. It was
deleted.")
        }

        // response is stored as array of byte. Read it to http
        responseReader :=
bufio.NewReader(bytes.NewReader(cachedResponse.response))
        response, err := http.ReadResponse(responseReader, nil)
        if err != nil {
                fmt.Printf("[ERROR] Error while reading dumped response
back to http response instance.")
                fmt.Println(err)
                return res, errors.New("[ERROR] Error while reading dumped
response back to http response instance")
        }

        fmt.Printf("[FOUND RESPONSE FROM CACHE]\n")
        return response, nil
}

func cacheResponse(r *http.Request, res *http.Response, expiryTime
time.Time) {
        // only cache get methods
        if r.Method != http.MethodGet {
                return
        }
        dumpRequest, err := httputil.DumpRequest(r, false)
        if err != nil {
                fmt.Printf("[ERROR] Could not dump request")
                fmt.Println(err)
                return
        }
        dumpResponse, err := httputil.DumpResponse(res, true)
        if err != nil {
                fmt.Printf("[ERROR] Could not cache the response (with
cache body = true)")
                fmt.Println(err)
                return
        }
        cache[string(dumpRequest)] = newCacheItem(dumpResponse, expiryTime)
        fmt.Printf("[SUCCESS] Cached response\n")
}

// Handles http requests
func onRequest(w http.ResponseWriter, r *http.Request) {
        start := time.Now()

        fmt.Printf("[RECEIVED A REQUEST] %s -- %s\n", r.Method, r.URL)
        if isDomainBlocked(r.Host) {
                fmt.Printf("[TRIED TO ACCESS BLACKLISTED DOMAIN] Request 
blocked - %s\n", r.Host)
                w.WriteHeader(http.StatusForbidden)
                w.Write([]byte("403 FORBIDDEN - TRIED TO ACCESS BLACKLISTED 
DOMAIN\r\n\r\n"))
                r.Body.Close()
                return
        }

        // Request URI can't be set in client requests.
        r.RequestURI = ""
        var url string = r.URL.String()

        //switch to https handler on CONNECT request
        if r.Method == http.MethodConnect {
                httpsHandler(w, r)
                return
        }

        // Create new http request
        r2, err := http.NewRequest(r.Method, url, r.Body)
        if err != nil {
                w.WriteHeader(http.StatusBadRequest)
                fmt.Printf("[ERROR BAD REQUEST] Could not write headers to 
new request: %v", err)
                return
        }

        r2.Header = r.Header
        res, err := checkCache(r2)
        isFromCache := err == nil

        // res is not in cache, send request
        if !isFromCache {
                // Round trip is used for single communications instead of 
having a client
                res, err = http.DefaultTransport.RoundTrip(r2)
                if err != nil {
                        fmt.Printf("[ERROR] Could not forward request")
                        w.WriteHeader(http.StatusBadGateway)
                        return
                }
        }

        fmt.Printf("[GOT RESPONSE TO REQUEST] Status - %s\n", res.Status)

        // look for no-cache or max-age headers
        //putInCache := !isFromCache
        var maxAge string = ""
        cacheHeaders := res.Header["Cache-Control"]
```

```
	if cacheHeaders != nil && len(cacheHeaders) > 0 {
		vals := strings.SplitN(cacheHeaders[0], ", ", -1)
		for _, item := range vals {
			if item == "no-cache" {
				//putInCache = false
			} else if strings.Contains(item, "max-age") {
				maxAge = item
			}
		}
	}

	// set expiry time for cache - if none specified keep for 24h
	expiryCacheTime := time.Now().Local()
	if maxAge != "" && !isFromCache {
		splitMaxAge := strings.SplitN(maxAge, "=", -1)
		seconds, err := strconv.Atoi(splitMaxAge[1])
		if err != nil {
			expiryCacheTime = expiryCacheTime.Add(time.Hour *
24)
		} else {
			expiryCacheTime = expiryCacheTime.Add(time.Second *
time.Duration(seconds))
		}
	} else {
		expiryCacheTime = expiryCacheTime.Add(time.Hour * 24)
	}

	fmt.Printf("[INFO] Expiry Time for cache: ")
	fmt.Println(expiryCacheTime)

	// Remove proxy header and copy other headers to return to client.
	header := w.Header()
	res.Header.Del("Proxy-Connection")
	for key, val := range res.Header {
		header[key] = val
	}
	w.WriteHeader(res.StatusCode)

	// should check from putInCache here however this breaks certain
calls for some reason
	if !isFromCache {
		cacheResponse(r2, res, expiryCacheTime)
		res.Body.Close()
	}
	if _, err := io.Copy(w, res.Body); err != nil {
		fmt.Printf("[ERROR] Could not proxy request %v", err)
	}
	fmt.Printf("EXECUTION TIME - %s\n\n", time.Since(start))
}

// remove specified string from slice containing blocked domains.
func unblockDomain(domain string) bool {
	for i, item := range blocked {
		if domain == item {
```

```
                        blocked = append(blocked[:i], blocked[i+1:]...)
                        fmt.Printf("[SUCCESS] Unblocked domain %s\n",
domain)
                        return true
                }
        }
        return false
}

// Returns true if the specified domain is blocked.
func isDomainBlocked(domain string) bool {
        for _, item := range blocked {
                if strings.Contains(domain, item) {
                        return true
                }
        }
        return false
}

func blockDomain(domain string) {
        if !isDomainBlocked(domain) {
                blocked = append(blocked, domain)
                fmt.Printf("[SUCCESS] Blocked domain %s\n", domain)
        }
}

// Continuously obtain user input and execute block / unblock domain
actions.
func userInputHandler() {
        scanner := bufio.NewScanner(os.Stdin)
        for scanner.Scan() {
                input := scanner.Text()
                // get text as lowercase & trim
                input = strings.ToLower(strings.TrimSpace(input))
                len := len(input)
                if len > 6 && input[0:6] == BLOCK_CMD {
                        var domain string = input[6:]
                        blockDomain(domain)
                } else if len > 8 && input[0:8] == UNBLOCK_CMD {
                        var found = unblockDomain(input[8:])
                        if !found {
                                fmt.Printf("[ERROR] Tried to unblock domain
that was not blocked: %s\n", input[8:])
                        }
                } else {
                        fmt.Printf("[ERROR] Command not supported : %s\n",
input)
                }
        }
}

func main() {
        fmt.Printf("[STARTING PROXY]\n")
        httpClient := http.HandlerFunc(onRequest)
        go userInputHandler()
        fmt.Printf("[PROXY LISTENING ON PORT 8080]\n")
        http.ListenAndServe(":8080", httpClient)
}
```