Web Proxy in GoLang

Luke Hackett - 17324340

Explanation

This web proxy is written in Go using built in libraries. The program starts taking input from the console and will take input from the user to block/unblock sites and view cache information. A thread is started to clean the cache periodically so that there are no sites in cache that shouldn't be. As well as this there is a http server set up to listen and handle requests using the handle function.

Blocking

To block a given domain you can type /b followed by the url. The proxy will block any subdomain on that url. For example, if I typed `/b google.com` then ads.google.com would be blocked as well as www.google.com etc. This is done by extracting the domain from the incoming request and comparing it against a set of domains that are in the blocked list. If you need to unblock a domain you can do it simply using `/u google.com`.

Caching

The proxy also caches http sites that it visits. This cache keeps site responses and headers to serve back to the client if the site is requested again within the expiry time. If you type `/s` the currently cached sites are listed as well as the average time and bandwith saved by caching sites.

Request Handling

When a request comes in the http listener will check if the site is blocked or not before proceding. If the site is not blocked, and the request method is `CONNECT`, the httpsHandler function will be called. Otherwise, if the site is cached it will be served from cache and if not the httpHandler function is called.

HTTPS Handler

The https handler hijacks control of the client connection then makes a connection to the server then copies the data between both the client and server. Once the data transfer is complete the connections are closed.

HTTP Handler

This is very straight forward, if a http request comes in it is made by the proxy to the server and the headers and body are written to the client response before closing the connection.

Code

```
package main

import (
    "bufio"
    "fmt"
    "io"
```

```
    "io/ioutil"
    "log"
    "net"
    "net/http"
    "os"
    "regexp"
    "strings"
    "time"
)

// CacheExpiry time in seconds
const CacheExpiry = 10

var twoDots = regexp.MustCompile("\\.")

var blockedSites = map[string]bool{}
var cachedSites = map[string]*cachedSite{}

var webTimes = make(map[string]time.Duration, 0)
var cacheTimes = make(map[string]time.Duration, 0)

type cachedSite struct {
    siteHeaders map[string]string
    siteBody    []byte
    timeFetched time.Time
}

func newCachedSite(res *http.Response, siteResponse []byte) *cachedSite {
    c := cachedSite{siteHeaders: make(map[string]string, 0), siteBody:
siteResponse}
    c.timeFetched = time.Now()

    for k, v := range res.Header {
        for _, vv := range v {
            c.siteHeaders[k] = vv
        }
    }

    return &c
}

func block(site string) {
    _, exists := blockedSites[site]

    if !exists {
        blockedSites[site] = true
        fmt.Printf("Added %s to Blocked Sites\n", site)
    } else {
        fmt.Println("Site is already blocked!")
    }
}
```

```
func unblock(site string) {
    _, exists := blockedSites[site]

    if !exists {
        fmt.Println("Site is already unblocked!")
    } else {
        delete(blockedSites, site)
        fmt.Printf("Removed %s from Blocked Sites\n", site)
    }
}

func copyDataStream(dst, src *net.TCPConn, host string) {
    io.Copy(dst, src)
    dst.CloseWrite()
    src.CloseRead()
}

func handler(w http.ResponseWriter, req *http.Request) {
    req.RequestURI = ""

    host := req.Host

    if !isBlocked(host) {
        log.Printf("Allowed %s\n", host)

        siteCached := siteCached(req.URL.String())

        start := time.Now()
        if req.Method == http.MethodConnect {
            httpsHandler(w, req)
        } else {
            if siteCached {
                serveCachedSite(w, req.URL.String())
            } else {
                httpHandler(w, req)
            }
        }

        end := time.Now()
        elapsed := end.Sub(start)

        //Split into cache res and normal res times
        if siteCached {
            cacheTimes[req.URL.String()] = elapsed
            log.Printf("Time taken for cache req to complete was %d
Milliseconds\n", elapsed/time.Millisecond)
        } else {
            webTimes[req.URL.String()] = elapsed
            log.Printf("Time taken for web req to complete was %d
Milliseconds\n", elapsed/time.Millisecond)
```

```
        }
    } else {
        log.Printf("Blocked %s\n", host)
        w.WriteHeader(http.StatusForbidden)
    }
}

func isBlocked(host string) bool {
    dots := twoDots.FindAllStringIndex(host, -1)
    if len(dots) > 1 {
        subIndex := dots[len(dots)-2]
        host = host[subIndex[0]+1:]
    }

    portIndex := strings.Index(host, ":")
    if portIndex > -1 {
        host = host[:portIndex]
    }

    _, exists := blockedSites[host]

    return exists
}

func siteCached(url string) bool {
    site, ok := cachedSites[url]
    if ok && site != nil &&
int64(time.Now().Sub(site.timeFetched)/time.Second) < CacheExpiry {
        return true
    }

    delete(cachedSites, url)
    return false
}

func httpsHandler(w http.ResponseWriter, req *http.Request) {
    req.URL.Scheme = "https"

    hij, ok := w.(http.Hijacker)
    if !ok {
        log.Panic("Cannot hijack")
    }

    proxyClient, _, err := hij.Hijack()
    if err != nil {
        log.Panic(err)
    }

    host := req.URL.Host

    targetSiteCon, err := net.Dial("tcp", host)
```

```
    if err != nil {
        log.Panic(err)
    }

    proxyClient.WriteHeader(http.StatusOK)

    targetTCP, targetOK := targetSiteCon.(*net.TCPConn)
    proxyClientTCP, clientOK := proxyClient.(*net.TCPConn)
    if targetOK && clientOK {
        go copyDataStream(targetTCP, proxyClientTCP, "")
        go copyDataStream(proxyClientTCP, targetTCP, host)
    }
}

func httpHandler(w http.ResponseWriter, req *http.Request) {
    c := &http.Client{}
    res, err := c.Do(req)
    if err != nil {
        log.Panic(err)
    }

    for k, v := range res.Header {
        for _, vv := range v {
            w.Header().Set(k, vv)
        }
    }

    bodyBytes, err := ioutil.ReadAll(res.Body)
    if err != nil {
        log.Fatal(err)
    }

    io.WriteString(w, string(bodyBytes))

    //Update cache
    cachedSites[res.Request.URL.String()] = newCachedSite(res, bodyBytes)

    req.Body.Close()
    res.Body.Close()
}

func serveCachedSite(w http.ResponseWriter, url string) {
    cachedSite := cachedSites[url]
    for k, v := range cachedSite.siteHeaders {
        w.Header().Set(k, v)
    }

    io.WriteString(w, string(cachedSite.siteBody))
}

func takeInput() {
```

```
    reader := bufio.NewReader(os.Stdin)
    fmt.Println("Web Proxy Console")
    fmt.Println("--------------------")

    for {
        fmt.Print("-> ")
        text, _ := reader.ReadString('\n')
        // convert CRLF to LF
        text = strings.Replace(text, "\n", "", -1)

        if strings.Contains(text, "/b") {
            site := text[3:]
            block(site)
        } else if strings.Contains(text, "/u") {
            site := text[3:]
            unblock(site)
        } else if strings.Contains(text, "/l") {
            fmt.Println("List of Blocked Sites")
            for k := range blockedSites {
                fmt.Printf("- %s\n", k)
            }
        } else if strings.Contains(text, "/s") {
            timeSavedPerURL := make([]int64, 0)
            for u, s := range cacheTimes {
                timeSaved := int64(webTimes[u]/time.Millisecond) -
int64(s/time.Millisecond)
                timeSavedPerURL = append(timeSavedPerURL,
int64(timeSaved))
            }

            if len(cacheTimes) > 0 {
                averageTimeSaved := int64(0)
                for _, t := range timeSavedPerURL {
                    averageTimeSaved += int64(t)
                }

                averageTimeSaved = averageTimeSaved /
int64(len(timeSavedPerURL))
                fmt.Printf("Average Time saved from Caching: %d
Milliseconds\n", averageTimeSaved)

                fmt.Println("Cached sites:")
                for k := range cachedSites {
                    fmt.Printf("%s: %d bytes\n", k,
len(cachedSites[k].siteBody))
                }
            } else if len(cachedSites) == 0 {
                fmt.Println("Cache empty!")
            }
        }
    }
```

```
}

func keepCacheClean() {
    for {
        for k, v := range cachedSites {
            if int64(time.Now().Sub(v.timeFetched)/time.Second) >
CacheExpiry {
                delete(cachedSites, k)
            }
        }

        time.Sleep(time.Second)
    }
}

func main() {
    go takeInput()
    go keepCacheClean()

    handler := http.HandlerFunc(handler)
    http.ListenAndServe(":8000", handler)
}
```