Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU34031 Advanced Computer Networks
Project 1 - A Web Proxy Server

Liam Junkermann - 19300141

March 6, 2022

Introduction

The goal of this assignment was to build a web proxy with the following details:

1. Respond to HTTP & HTTPS requests and display each request on a management console
2. Handle websocket connections
3. Dynamically block selected URLs via the management console
4. Efficiently cache HTTP request locally
5. Handle multiple requests through threading

Contents

1 Design and Implementation

This webproxy has 3 main parts: the main server (run from `main.go`), the cache (built in `cache.go`) and the web dashboard which manages blocking of urls (run from `webDashboard.go` and `dynamicBlock.go`). The main proxy server handles the proxy connections by either forwarding the necessary https `CONNECT` requests and creating the appropriate tunnels, or repeating, caching, and responding to HTTP requests. The Cache is managed through responses being saved to files with the request content included, this cache can be retained through many stops and starts of the proxy server. The hashes are saved in memory along with timing and bandwidth data (this data is lost when the server is stopped). Finally, a web dashboard was created to manage the URLs and blocking selected URLs, currently the web dashboard needs to request the list of proxied endpoints, but a websocket approach could be used going forward to allow for streaming of URLs to the web management console.

2 Code Listings

main.go

```go
package main

import (
    "flag"
    "io"
    "log"
    "net"
    "net/http"
    "net/http/httputil"
    "strings"
    "sync"
    "time"

    "github.com/golang/glog"
)

var config *Config
var cache *Cache
var urlList *URLlist

func main() {
    configPath := flag.String("config", "./tiny.json", "configuration .json file path")
    flag.Parse()

    loadConfig(*configPath)
    glog.Info("Config Loaded")

    prepare()

    handler := &proxy{}
    webHandler := &WebDashboard{}

    var addr = flag.String("addr", "127.0.0.1:"+config.Port, "The addr of the proxy.")
    var webAddr = flag.String("webAddr", "127.0.0.1:"+config.WebPort, "The web dashboard
        addr.")

    wg := &sync.WaitGroup{}
    wg.Add(1)
    go func() {
        glog.Info("Starting proxy server on ", *addr)
        if err := http.ListenAndServe(*addr, handler); err != nil {
            glog.Fatal("ListenAndServe:", err)
        }
        wg.Done()
    }()
    wg.Add(1)
    go func() {
        glog.Info("Starting web server on ", *webAddr)
        if err := http.ListenAndServe(*webAddr, webHandler); err != nil {
            glog.Fatal("ListenAndServe:", err)
        }
        wg.Done()
    }()
    wg.Wait()
}

func loadConfig(configPath string) {
    var err error

    config, err = LoadConfig(configPath)
    if err != nil {
        glog.Fatal("Could not read config: '%s'", err.Error())
    }
}
```

```
65 func prepare() {
66     var err error
67     cache, err = CreateCache(config.CacheFolder)
68     urlList, _ = CreateList()
69
70     if err != nil {
71         glog.Fatal("Could not init cache: '%s'", err.Error())
72     }
73 }
74
75 var hopHeaders = []string{
76     "Connection",
77     "Keep-Alive",
78     "Proxy-Authenticate",
79     "Proxy-Authorization",
80     "Te", // canonicalized version of "TE"
81     "Trailers",
82     "Transfer-Encoding",
83     "Upgrade",
84 }
85
86 func delHopHeaders(header http.Header) {
87     for _, h := range hopHeaders {
88         header.Del(h)
89     }
90 }
91
92 func appendHostToXForwardHeader(header http.Header, host string) {
93     // Including previous proxy hops in X-Forwarded-For Header
94     if prior, ok := header["X-Forwarded-For"]; ok {
95         host = strings.Join(prior, ", ") + ", " + host
96     }
97     header.Set("X-Forwarded-For", host)
98 }
99
100 type proxy struct{}
101
102 func (p *proxy) ServeHTTP(wr http.ResponseWriter, req *http.Request) {
103     glog.Info(req.RemoteAddr, " ", req.Method, " ", req.URL, " Host: ", req.Host)
104     fullUrl := req.Host + req.URL.Path + "?" + req.URL.RawQuery
105
106     _, isListed := urlList.has(fullUrl)
107     if !isListed {
108         urlList.put(fullUrl, &DynamicBlock{Remoteaddr: req.RemoteAddr, Method: req.Method
            , Url: req.Host + "" + req.URL.Path, Blocked: false})
109     }
110
111     listing, err := urlList.get(fullUrl)
112
113     if err != nil {
114         glog.Fatal("Error getting listing")
115     }
116
117     glog.Info("Blocked Status: ", listing.Blocked)
118
119     if !listing.Blocked {
120         requestDump, _ := httputil.DumpRequest(req, true)
121         glog.Info(string(requestDump))
122
123         if req.Method != "CONNECT" { // if HTTP request
124             glog.Info("Requested: ", fullUrl)
125             client := &http.Client{}
126
127             if busy, ok := cache.has(fullUrl); !ok {
128                 startTime := time.Now()
129                 defer busy.Unlock()
130                 req.RequestURI = ""
131
132                 delHopHeaders(req.Header)
```

```
133                 if clientIP, _, err := net.SplitHostPort(req.RemoteAddr); err == nil {
134                     appendHostToXForwardHeader(req.Header, clientIP)
135                 }
136
137                 resp, err := client.Do(req)
138                 if err != nil {
139                     http.Error(wr, "Server Error", http.StatusInternalServerError)
140                     log.Fatal("ServeHTTP:", err)
141                     return
142                 }
143
144                 var reader io.Reader
145                 reader = resp.Body
146                 endTime := time.Now()
147                 totalTime := endTime.Sub(startTime)
148                 glog.Info("Time Spent: ", totalTime)
149                 err = cache.put(fullUrl, &reader, resp.ContentLength, totalTime)
150                 if err != nil {
151                     http.Error(wr, "Server Error", http.StatusInternalServerError)
152                     glog.Fatal("ServeHTTP:", err)
153                     return
154                 }
155                 defer resp.Body.Close()
156             }
157
158             content, err := cache.get(fullUrl)
159             if err != nil {
160                 http.Error(wr, "Server Error", http.StatusInternalServerError)
161                 glog.Fatal("Serve from Cache", err)
162             } else {
163                 contentWritten, err := io.Copy(wr, *content)
164                 if err != nil {
165                     glog.Fatal("Error writing response: ", err.Error())
166                     return
167                 }
168                 glog.Info("Wrote ", contentWritten, " bytes to client")
169             }
170         } else {
171             // glog.Info(strings.Index(req.Host, ":"))
172             if !strings.Contains(req.Host, ":") {
173                 req.Host += ":80"
174             }
175             // Server connection
176             serverConn, err := net.Dial("tcp", req.Host)
177             if err != nil {
178                 glog.Error(err)
179             }
180
181             // Access Client connection
182             hj, _ := wr.(http.Hijacker)
183             clientConn, _, hjErr := hj.Hijack()
184             if hjErr != nil {
185                 glog.Error(hjErr)
186             }
187
188             clientConn.Write([]byte("HTTP/1.0 200 OK\r\n\r\n"))
189             go io.Copy(clientConn, serverConn)
190             _, srvErr := io.Copy(serverConn, clientConn)
191             if srvErr != nil {
192                 glog.Info(srvErr)
193             }
194         }
195     } else {
196         http.Error(wr, "Proxy Blocked", http.StatusForbidden)
197     }
198 }
```

config.go

```go
package main

import (
    "encoding/json"
    "io/ioutil"
)

type Config struct {
    CacheFolder      string `json:"cache_folder"`
    Port             string `json:"port"`
    WebPort          string `json:"WebPort"`
    MaxCacheItemSize int64  `json:"max_cache_item_size"` // in MB
}

func LoadConfig(path string) (*Config, error) {
    file, err := ioutil.ReadFile(path)

    if err != nil {
        return nil, err
    }

    var config Config
    json.Unmarshal(file, &config)

    return &config, nil
}
```

cache.go

```go
package main

import (
    "bufio"
    "bytes"
    "crypto/sha256"
    "encoding/hex"
    "fmt"
    "hash"
    "io"
    "io/ioutil"
    "os"
    "sync"
    "time"

    "github.com/golang/glog"
)

type Cache struct {
    folder       string
    hash         hash.Hash
    knownValues  map[string][]byte
    timingValues map[string]time.Duration
    busyValues   map[string]*sync.Mutex
    mutex        *sync.Mutex
}

func CreateCache(path string) (*Cache, error) {
    fileInfos, err := ioutil.ReadDir(path)
    if err != nil {
        glog.Error("Cannot open cache folder ", path, ": ", err)
        glog.Info("Create cache folder ", path)
        os.Mkdir(path, os.ModePerm)
    }

    values := make(map[string][]byte)
```

```
37      timeValues := make(map[string]time.Duration)
38      busy := make(map[string]*sync.Mutex)
39
40      // Go through every file an save its name in the map. The content of the file
41      // is loaded when needed. This makes sure that we do not have to read
42      // the directory content each time the user wants data that is not yet loaded.
43      for _, info := range fileInfos {
44          if !info.IsDir() {
45              values[info.Name()] = nil
46          }
47      }
48
49      hash := sha256.New()
50
51      mutex := &sync.Mutex{}
52
53      cache := &Cache{
54          folder:       path,
55          hash:         hash,
56          knownValues:  values,
57          timingValues: timeValues,
58          busyValues:   busy,
59          mutex:        mutex,
60      }
61
62      return cache, nil
63  }
64
65  // Returns true if the resource is found, and false otherwise. If the
66  // resource is busy, this method will hang until the resource is free. If
67  // the resource is not found, a lock indicating that the resource is busy will
68  // be returned. Once the resource has been put into cache the busy lock *must*
69  // be unlocked to allow others to access the newly cached resource
70  func (c *Cache) has(key string) (*sync.Mutex, bool) {
71      hashValue := calcHash(key)
72
73      c.mutex.Lock()
74      defer c.mutex.Unlock()
75
76      // If the resource is busy, wait for it to be free. This is the case if
77      // the resource is currently being cached as a result of another request.
78      // Also, release the lock on the cache to allow other readers while waiting
79      if lock, busy := c.busyValues[hashValue]; busy {
80          c.mutex.Unlock()
81          lock.Lock()
82          lock.Unlock()
83          c.mutex.Lock()
84      }
85
86      // If a resource is in the shared cache, it cannot be reserved. One can simply
87      // access it directly from the cache
88      if _, found := c.knownValues[hashValue]; found {
89          return nil, true
90      }
91
92      // The resource is not in the cache, mark the resource as busy until it has
93      // been cached successfully. Unlocking lock is required!
94      lock := new(sync.Mutex)
95      lock.Lock()
96      c.busyValues[hashValue] = lock
97      return lock, false
98  }
99
100 func (c *Cache) get(key string) (*io.Reader, error) {
101     var response io.Reader
102     hashValue := calcHash(key)
103
104     // Try to get content. Error if not found.
105     c.mutex.Lock()
```

```
106     content, ok := c.knownValues[hashValue]
107     timing := c.timingValues[hashValue]
108     c.mutex.Unlock()
109     if !ok && len(content) > 0 {
110         glog.Info("Cache doesn't know key ", hashValue)
111         return nil, fmt.Errorf("key '%s' is not known to cache", hashValue)
112     }
113
114     glog.Info("Cache has key", hashValue)
115
116     // Key is known, but not loaded into RAM
117     if content == nil {
118         glog.Info("Cache item ", hashValue, " known but is not stored in memory. Using
                file.")
119
120         file, err := os.Open(c.folder + hashValue)
121         if err != nil {
122             glog.Error("Error reading cached file ", hashValue, ": ", err)
123             return nil, err
124         }
125
126         response = file
127
128         glog.Info("Create reader from file ", hashValue)
129     } else { // Key is known and data is already loaded to RAM
130         response = bytes.NewReader(content)
131         glog.Info("Create reader from ", len(content), " byte large cache content")
132     }
133
134     glog.Info("Saved ", timing.Milliseconds(), "ms and ", len(content), "bytes")
135     return &response, nil
136 }
137
138 // release is an internal method which atomically caches an item and unmarks
139 // the item as busy, if it was busy before. The busy lock *must* be unlocked
140 // elsewhere!
141 func (c *Cache) release(hashValue string, content []byte, timing time.Duration) {
142     c.mutex.Lock()
143     delete(c.busyValues, hashValue)
144     c.knownValues[hashValue] = content
145     c.timingValues[hashValue] = timing
146     c.mutex.Unlock()
147 }
148
149 func (c *Cache) put(key string, content *io.Reader, contentLength int64, timing time.
        Duration) error {
150     hashValue := calcHash(key)
151
152     // Small enough to put it into the in-memory cache
153     if contentLength <= config.MaxCacheItemSize*1024*1024 {
154         buffer := &bytes.Buffer{}
155         _, err := io.Copy(buffer, *content)
156         if err != nil {
157             return err
158         }
159
160         defer c.release(hashValue, buffer.Bytes(), timing)
161         glog.Info("Added ", hashValue, "into in-memory cache")
162
163         err = ioutil.WriteFile(c.folder+hashValue, buffer.Bytes(), 0644)
164         if err != nil {
165             return err
166         }
167         glog.Info("Wrote content of entry ", hashValue, " into file")
168     } else { // Too large for in-memory cache, just write to file
169         defer c.release(hashValue, nil, time.Since(time.Now()))
170         glog.Info("Added nil-entry for ", hashValue, " into in-memory cache")
171
172         file, err := os.Create(c.folder + hashValue)
```

```
173         if err != nil {
174             return err
175         }
176
177         writer := bufio.NewWriter(file)
178         _, err = io.Copy(writer, *content)
179         if err != nil {
180             return err
181         }
182         glog.Info("Wrote content of entry ", hashValue, " into file")
183     }
184
185     glog.Info("Cache wrote content into ", hashValue)
186
187     return nil
188 }
189
190 func calcHash(data string) string {
191     sha := sha256.Sum256([]byte(data))
192     return hex.EncodeToString(sha[:])
193 }
```

dynamicBlock.go

```
1  package main
2
3  import (
4      "crypto/sha256"
5      "errors"
6      "hash"
7      "sync"
8
9      "github.com/golang/glog"
10 )
11
12 type DynamicBlock struct {
13     Remoteaddr string `json:"remoteAddr"`
14     Method     string `json:"method"`
15     Url        string `json:"url"`
16     Blocked    bool   `json:"blocked"`
17 }
18
19 type URLlist struct {
20     hash       hash.Hash
21     UrlValues  map[string]DynamicBlock `json:"urlValues"`
22     busyValues map[string]*sync.Mutex
23     mutex      *sync.Mutex
24 }
25
26 func CreateList() (*URLlist, error) {
27     url := make(map[string]DynamicBlock)
28     busy := make(map[string]*sync.Mutex)
29
30     hash := sha256.New()
31     mutex := &sync.Mutex{}
32
33     urlList := &URLlist{
34         hash:       hash,
35         UrlValues:  url,
36         busyValues: busy,
37         mutex:      mutex,
38     }
39
40     return urlList, nil
41 }
42
43 func (u *URLlist) has(key string) (*sync.Mutex, bool) {
```

```
44      hashValue := calcHash(key)
45
46      u.mutex.Lock()
47      defer u.mutex.Unlock()
48
49      if lock, busy := u.busyValues[hashValue]; busy {
50          u.mutex.Unlock()
51          lock.Lock()
52          lock.Unlock()
53          u.mutex.Lock()
54      }
55
56      if _, found := u.UrlValues[hashValue]; found {
57          return nil, true
58      }
59
60      lock := new(sync.Mutex)
61      lock.Lock()
62      u.busyValues[hashValue] = lock
63      return lock, false
64  }
65
66  func (u *URLlist) get(key string) (*DynamicBlock, error) {
67      hashValue := calcHash(key)
68
69      u.mutex.Lock()
70      url, ok := u.UrlValues[hashValue]
71      u.mutex.Unlock()
72
73      if !ok {
74          glog.Info("URL Item", hashValue, " has not been logged before")
75      }
76
77      glog.Info("Found ", hashValue, " hash has been logged")
78      return &url, nil
79  }
80
81  func (u *URLlist) block(hashValue string) (*DynamicBlock, error) {
82      u.mutex.Lock()
83      listing, ok := u.UrlValues[hashValue]
84      u.mutex.Unlock()
85
86      if !ok {
87          glog.Error("URL Item", hashValue, " was not logged properly")
88          return nil, errors.New("hash not found")
89      }
90      listing.Blocked = true
91
92      glog.Info("Blocked hash: ", hashValue)
93
94      defer u.release(hashValue, listing)
95      return &listing, nil
96  }
97
98  func (u *URLlist) unblock(hashValue string) (*DynamicBlock, error) {
99      u.mutex.Lock()
100     listing, ok := u.UrlValues[hashValue]
101     u.mutex.Unlock()
102
103     if !ok {
104         glog.Error("URL Item", hashValue, " was not logged properly")
105         return nil, errors.New("hash not found")
106     }
107     listing.Blocked = false
108
109     defer u.release(hashValue, listing)
110     return &listing, nil
111 }
```

```
113 func (u *URLlist) put(key string, urlListing *DynamicBlock) error {
114     hashValue := calcHash(key)
115     glog.Info("putting ", hashValue, " with listing ", urlListing)
116     defer u.release(hashValue, *urlListing)
117     return nil
118 }

120 func (u *URLlist) release(hashValue string, urlValue DynamicBlock) {
121     u.mutex.Lock()
122     delete(u.busyValues, hashValue)
123     u.UrlValues[hashValue] = urlValue
124     u.mutex.Unlock()
125 }
```

webDashboard.go

```
1  package main

3  import (
4      "encoding/json"
5      "fmt"
6      "io"
7      "net/http"

9      "github.com/golang/glog"
10 )

12 type WebDashboard struct{}

14 func (f *WebDashboard) ServeHTTP(w http.ResponseWriter, r *http.Request) {
15     glog.Info(r.Method, " request from ", r.Host, " for ", r.URL.Path)

17     if r.URL.Path == "/urls" && r.Method == "GET" {
18         glog.Info("handling urls")
19         w.Header().Set("Content-Type", "application/json")
20         json.NewEncoder(w).Encode(urlList.UrlValues)
21     } else {
22         switch r.Method {
23         case "GET":
24             http.ServeFile(w, r, "index.html")
25         case "POST":
26             //TODO: HANDLE POST REQUEST
27             bodyBytes, err := io.ReadAll(r.Body)
28             bodyString := string(bodyBytes)
29             if err != nil {
30                 fmt.Fprintf(w, "Sorry, an error occurred reading the body: %s", err.Error
                       ())
31             }
32             switch r.URL.Path {
33             case "/block":
34                 glog.Info("body:")
35                 glog.Info(bodyString)
36                 urlList.block(bodyString)
37                 json.NewEncoder(w).Encode(urlList.UrlValues[bodyString])
38                 break
39             case "/unblock":
40                 glog.Info("body:")
41                 glog.Info(bodyString)
42                 urlList.unblock(bodyString)
43                 json.NewEncoder(w).Encode(urlList.UrlValues[bodyString])
44             }
45         default:
46             fmt.Fprintf(w, "Sorry, only GET and POST methods are supported.")
47         }
48     }
49 }
```