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

February 27, 2023

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 proxy server (handled in `proxyserver.go`), the cache (a file-based implementation built in `filecache.go`), the web dashboard which manages blocking of urls (run from `dashboard.go`, and `dynamicblock.go`). The 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. 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. These pieces are combined into a final executable (`final.go`) which can be deployed to a docker container or directly to an active server. The architecture of this proxy server does not add any headers

2 Code Listings

proxyserver.go

```
1 package proxyserver

3 import (
4     "encoding/json"
5     "fmt"
6     "io"
7     "net"
8     "net/http"
9     "proxyserver/pkg/cache"
10    "proxyserver/pkg/dynamicblock"
11    "strings"
12    "time"

14    "github.com/sirupsen/logrus"
15 )

17 type ProxyServer struct {
18    lg       *logrus.Logger
19    UrlList *dynamicblock.UrlList
20    c        cache.Cache
21 }

23 var hopHeaders = []string{
24    "Connection",
25    "Keep-Alive",
26    "Proxy-Authenticate",
27    "Proxy-Authorization",
28    "Te", // canonicalized version of "TE"
29    "Trailers",
30    "Transfer-Encoding",
31    "Upgrade",
32 }

34 func delHopHeaders(header http.Header) {
35    for _, h := range hopHeaders {
36        header.Del(h)
37    }
38 }

40 func appendHostToXForwardHeader(header http.Header, host string) {
41    // Including previous proxy hops in X-Forwarded-For Header
42    if prior, ok := header["X-Forwarded-For"]; ok {
43        host = strings.Join(prior, ", ") + ", " + host
44    }
45    header.Set("X-Forwarded-For", host)
46 }

48 func New(lg *logrus.Logger, urlist *dynamicblock.UrlList, c cache.Cache) *ProxyServer {
49    return &ProxyServer{lg, urlist, c}
50 }

52 func (p *ProxyServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
53    p.lg.Infof("%s, %s, %s, Host: %s", r.RemoteAddr, r.Method, r.URL, r.Host)
54    fullUrl := r.Host + r.URL.EscapedPath() + "?" + r.URL.RawQuery

56    _, hasLsting := p.UrlList.Has(fullUrl)
57    if !hasLsting {
58        err := p.UrlList.Set(fullUrl, &dynamicblock.DynamicBlock{RemoteAddr: r.RemoteAddr
              , Method: r.Method, Url: r.Host + "" + r.URL.EscapedPath(), Blocked: false})
59        if err != nil {
60            http.Error(w, fmt.Sprintf("failed to set lsting, %s", err), http.
                  StatusInternalServerError)
61            return
62        }
63    }
```

```
65      lsting, err := p.UrlList.Get(fullUrl)
66      if err != nil {
67          if err != nil {
68              http.Error(w, fmt.Sprintf("failed to get lsting, %s", err), http.
                    StatusInternalServerError)
69              return
70          }
71      }

73      str, _ := json.Marshal(lsting)
74      p.lg.Debugf("got listing, %s", string(str))

76      if !lsting.Blocked {
77          // reqDump, _ := httputil.DumpRequest(r, true)
78          if r.Method != "CONNECT" {
79              client := &http.Client{}

81              if busy, ok := p.c.Has(fullUrl); !ok {
82                  p.lg.Debugf("cache does not have %s", fullUrl)
83                  startTime := time.Now()
84                  defer busy.Unlock()
85                  r.RequestURI = ""

87                  delHopHeaders(r.Header)
88                  if clientIp, _, err := net.SplitHostPort(r.RemoteAddr); err == nil {
89                      appendHostToXForwardHeader(r.Header, clientIp)
90                  }

92                  resp, err := client.Do(r)
93                  if err != nil {
94                      http.Error(w, fmt.Sprintf("forwarding error, %s", err), http.
                            StatusInternalServerError)
95                      return
96                  }

98                  reader := io.Reader(resp.Body)
99                  totalTime := time.Since(startTime)
100                 p.lg.Debugf("req took %dms", totalTime.Milliseconds())
101                 err = p.c.Put(fullUrl, &reader, uint64(resp.ContentLength), totalTime)
102                 if err != nil {
103                     http.Error(w, fmt.Sprintf("failed to put to cache, %s", err), http.
                            StatusInternalServerError)
104                     return
105                 }
106                 defer resp.Body.Close()
107             }

109             content, err := p.c.Get(fullUrl)
110             if err != nil {
111                 http.Error(w, fmt.Sprintf("cache error, %s", err), http.
                        StatusInternalServerError)
112                 return
113             } else {
114                 contentWritten, err := io.Copy(w, *content)
115                 if err != nil {
116                     p.lg.Errorf("error writing response: %s", err)
117                     return
118                 }
119                 p.lg.Infof("wrote %d bytes to client", contentWritten)
120             }
121         } else {
122             if !strings.Contains(r.Host, ":") {
123                 r.Host += ":80"
124             }

126             srvConn, err := net.DialTimeout("tcp", r.Host, 10*time.Second)
127             if err != nil {
128                 http.Error(w, err.Error(), http.StatusServiceUnavailable)
```

```
129                 return
130             }
131             p.lg.Debug("created srvConn")

133             w.WriteHeader(http.StatusOK)
134             hj, ok := w.(http.Hijacker)
135             if !ok {
136                 http.Error(w, "Hijacking not supported", http.StatusInternalServerError)
137                 return
138             }

140             clientConn, _, err := hj.Hijack()
141             if err != nil {
142                 http.Error(w, err.Error(), http.StatusServiceUnavailable)
143             }
144             go transfer(srvConn, clientConn)
145             go transfer(clientConn, srvConn)

147             p.lg.Debug("here now")
148         }
149     } else {
150         http.Error(w, "Proxy Blocked", http.StatusForbidden)
151     }
152 }

154 func transfer(destination io.WriteCloser, source io.ReadCloser) {
155     defer destination.Close()
156     defer source.Close()
157     io.Copy(destination, source)
158 }
```

pkg/cache/filecache/filecache.go

```
1  package filecache

3  import (
4      "bufio"
5      "bytes"
6      "crypto/sha256"
7      "fmt"
8      "hash"
9      "io"
10     "os"
11     "path"
12     "proxyserver/pkg/cache"
13     "sync"
14     "time"

16     "github.com/sirupsen/logrus"
17 )

19 type Cache struct {
20     folder        string
21     hash          hash.Hash
22     KnownValues   map[string][]byte          `json:"known_vals"`
23     TimingValues  map[string]time.Duration   `json:"timing_vals"`
24     busyValues    map[string]*sync.Mutex
25     mutex         *sync.Mutex

27     lg *logrus.Logger
28 }

30 func New(path string, lg *logrus.Logger) (*Cache, error) {
31     os.RemoveAll(path)
32     os.MkdirAll(path, os.ModePerm)

34     cache := &Cache{
```

```
35          folder:       path,
36          hash:         sha256.New(),
37          KnownValues:  make(map[string][]byte),
38          TimingValues: make(map[string]time.Duration),
39          busyValues:   make(map[string]*sync.Mutex),
40          mutex:        &sync.Mutex{},
41          lg:           lg,
42      }
43
44      return cache, nil
45  }
46
47  func (c *Cache) Has(key string) (*sync.Mutex, bool) {
48      hash := cache.CalcHash(key)
49
50      c.mutex.Lock()
51      defer c.mutex.Unlock()
52
53      if lock, busy := c.busyValues[hash]; busy {
54          c.mutex.Unlock()
55          lock.Lock()
56          lock.Unlock()
57          c.mutex.Lock()
58      }
59
60      if _, found := c.KnownValues[hash]; found {
61          return nil, true
62      }
63
64      lock := new(sync.Mutex)
65      lock.Lock()
66      c.busyValues[hash] = lock
67      return lock, false
68  }
69
70  func (c *Cache) Get(key string) (*io.Reader, error) {
71      var response io.Reader
72      hashVal := cache.CalcHash(key)
73
74      c.mutex.Lock()
75      content, ok := c.KnownValues[hashVal]
76      timing := c.TimingValues[hashVal]
77      cLen := int64(0)
78      c.mutex.Unlock()
79      if !ok && len(content) > 0 {
80          return nil, fmt.Errorf("key '%s' (%s) not in cache", key, hashVal)
81      }
82
83      c.lg.Debugf("cache has key %s (%s)", key, hashVal)
84
85      if content == nil {
86          c.lg.Debugf("loading %s from file", key)
87
88          file, err := os.Open(path.Join(c.folder, hashVal))
89          if err != nil {
90              return nil, fmt.Errorf("error opening cache file %s, %s", hashVal, err)
91          }
92
93          response = file
94          stat, err := file.Stat()
95          if err != nil {
96              return nil, fmt.Errorf("error getting size of file %s, %s", hashVal, err)
97          }
98          cLen = stat.Size()
99      } else {
100         response = bytes.NewReader(content)
101     }
102
103     c.lg.Infof("saved %dms and fetching %d bytes", timing.Milliseconds(), cLen)
```

```
104     return &response, nil
105 }

107 // internal function to handling blocking of content being added or updated
108 func (c *Cache) release(hashValue string, content []byte, timing time.Duration) {
109     c.mutex.Lock()
110     delete(c.busyValues, hashValue)
111     c.KnownValues[hashValue] = content
112     c.TimingValues[hashValue] = timing
113     c.mutex.Unlock()
114 }

116 func (c *Cache) Put(key string, content *io.Reader, contentLength uint64, timing time.
        Duration) error {
117     hashVal := cache.CalcHash(key)

119     defer c.release(hashVal, nil, timing)
120     file, err := os.Create(path.Join(c.folder, hashVal))
121     if err != nil {
122         return fmt.Errorf("could not create file %s, %s", hashVal, err)
123     }

125     writer := bufio.NewWriter(file)
126     b, err := io.Copy(writer, *content)
127     if err != nil {
128         return fmt.Errorf("could not copy to file %s, %s", hashVal, err)
129     }
130     c.lg.Debugf("wrote %d bytes to %s", b, path.Join(c.folder, hashVal))
131     return nil
132 }
```

pkg/dashboard/dashboard.go

```
1 package dashboard

3 import (
4     "encoding/json"
5     "fmt"
6     "io"
7     "net/http"
8     "os"
9     "path"
10    "proxyserver"

12    "github.com/sirupsen/logrus"
13 )

15 type Dashboard struct {
16     lg *logrus.Logger
17     p  *proxyserver.ProxyServer
18 }

20 func New(lg *logrus.Logger, p *proxyserver.ProxyServer) *Dashboard {
21     return &Dashboard{
22         lg,
23         p,
24     }
25 }

27 func (d *Dashboard) ServeHTTP(w http.ResponseWriter, r *http.Request) {
28     switch r.Method {
29     case "GET":
30         switch r.URL.EscapedPath() {
31         case "/urls":
32             w.Header().Set("Content-Type", "application/json")
33             json.NewEncoder(w).Encode(d.p.UrlList.UrlVals)
34         default:
```

```
35              d.lg.Info("GET default")
36              wd, err := os.Getwd()
37              if err != nil {
38                  http.Error(w, fmt.Sprintf("couldn't get working dir, %s", err), http.
                        StatusInternalServerError)
39                  break
40              }
41              http.ServeFile(w, r, path.Join(wd, "pkg/dashboard", "./index.html"))
42          }
43      case "POST":
44          body, err := io.ReadAll(r.Body)
45          if err != nil {
46              fmt.Fprintf(w, "Error occurred reading body: %s", err)
47          }
48          switch r.URL.EscapedPath() {
49          case "/block":
50              d.p.UrlList.Block(string(body))
51              json.NewEncoder(w).Encode(d.p.UrlList.UrlVals[string(body)])
52          case "/unblock":
53              d.p.UrlList.Unblock(string(body))
54              json.NewEncoder(w).Encode(d.p.UrlList.UrlVals[string(body)])
55          }
56      default:
57          fmt.Fprintf(w, "Sorry, only GET and POST methods are supported")
58      }
59  }
```

pkg/dynamicblock/dynamicblock.go

```
1  package dynamicblock
2
3  import (
4      "crypto/sha256"
5      "fmt"
6      "hash"
7      "proxyserver/pkg/cache"
8      "sync"
9  )
10
11 type DynamicBlock struct {
12     RemoteAddr string `json:"remote_addr"`
13     Method     string `json:"method"`
14     Url        string `json:"url"`
15     Blocked    bool   `json:"blocked"`
16 }
17
18 type UrlList struct {
19     hash     hash.Hash
20     UrlVals  map[string]DynamicBlock `json:"url_values"`
21     busyVals map[string]*sync.Mutex
22     mutex    *sync.Mutex
23 }
24
25 func New() *UrlList {
26     return &UrlList{
27         sha256.New(),
28         make(map[string]DynamicBlock),
29         make(map[string]*sync.Mutex),
30         &sync.Mutex{},
31     }
32 }
33
34 func (l *UrlList) release(hash string, lsting DynamicBlock) {
35     l.mutex.Lock()
36     delete(l.busyVals, hash)
37     l.UrlVals[hash] = lsting
38     l.mutex.Unlock()
```

```
39 }

41 func (l *UrlList) Has(key string) (*sync.Mutex, bool) {
42     hash := cache.CalcHash(key)

44     l.mutex.Lock()
45     defer l.mutex.Unlock()

47     if lock, busy := l.busyVals[hash]; busy {
48         l.mutex.Unlock()
49         lock.Lock()
50         lock.Unlock()
51         l.mutex.Lock()
52     }

54     if _, found := l.UrlVals[hash]; found {
55         return nil, true
56     }

58     lock := new(sync.Mutex)
59     lock.Lock()
60     l.busyVals[hash] = lock
61     return lock, false
62 }

64 func (l *UrlList) Get(key string) (*DynamicBlock, error) {
65     hash := cache.CalcHash(key)

67     l.mutex.Lock()
68     url, ok := l.UrlVals[hash]
69     l.mutex.Unlock()

71     if !ok {
72         return nil, fmt.Errorf("request url item not logged")
73     }

75     return &url, nil
76 }

78 func (l *UrlList) Set(key string, lsting *DynamicBlock) error {
79     hash := cache.CalcHash(key)

81     defer l.release(hash, *lsting)
82     return nil
83 }

85 func (l *UrlList) Block(hash string) (*DynamicBlock, error) {

87     l.mutex.Lock()
88     lsting, ok := l.UrlVals[hash]
89     l.mutex.Unlock()

91     if !ok {
92         return nil, fmt.Errorf("hash %s not found", hash)
93     }
94     lsting.Blocked = true
95     defer l.release(hash, lsting)
96     return &lsting, nil
97 }

99 func (l *UrlList) Unblock(hash string) (*DynamicBlock, error) {
100     l.mutex.Lock()
101     lsting, ok := l.UrlVals[hash]
102     l.mutex.Unlock()

104     if !ok {
105         return nil, fmt.Errorf("hash %s not found", hash)
106     }
107     lsting.Blocked = false
```

```
108     defer l.release(hash, lsting)
109     return &lsting, nil
110 }
```

cmd/final/final.go

```
1  package main
2
3  import (
4      "flag"
5      "fmt"
6      "net/http"
7      "proxyserver"
8      "proxyserver/pkg/cache/filecache"
9      "proxyserver/pkg/dashboard"
10     "proxyserver/pkg/dynamicblock"
11
12     "github.com/sirupsen/logrus"
13 )
14
15 func main() {
16     proxyPort := flag.String("proxy", "8081", "The proxy port")
17     webPort := flag.String("web", "8080", "The web dashboard port")
18     flag.Parse()
19
20     lg := logrus.New()
21     lg.SetLevel(logrus.DebugLevel)
22
23     cache, err := filecache.New("cache", lg)
24     if err != nil {
25         lg.Fatalf("could not init cache, %s", err)
26     }
27     urlList := dynamicblock.New()
28
29     proxy := proxyserver.New(lg, urlList, cache)
30     dash := dashboard.New(lg, proxy)
31
32     errCh := make(chan error)
33
34     go func() {
35         lg.Infof("starting proxy server on :%s", *proxyPort)
36         if err := http.ListenAndServe(":"+*proxyPort, proxy); err != nil {
37             errCh <- fmt.Errorf("proxy error, %s", err)
38         }
39     }()
40
41     go func() {
42         lg.Infof("starting dash server on :%s", *webPort)
43         if err := http.ListenAndServe(":"+*webPort, dash); err != nil {
44             errCh <- fmt.Errorf("proxy error, %s", err)
45         }
46     }()
47
48     fatalErr := <-errCh
49     lg.Fatal(fatalErr)
50 }
```