CSU34031 Advanced Telecommunications

Project-1 - Senán d'Art - 17329580

Introduction

This project is to:

- Develop a web proxy that can handle HTTP, HTTPS and Websocket connections.
- Support caching of HTTP packets
- Support the dynamic blocking of URLs through a management console
- Support multithreaded operation

I chose to use NodeJS and JavaScript to accomplish these goals.

High-Level Overview

For every connection to the proxy server (multiple occurences possible for each user):

- If the requested **URL is not blacklisted**:
  - If the connection **uses TLS or is a Websocket**:
    - Pipe all packets from server to client
    - Pipe all packets from client to server
    - Notes:
      - There is no reason to parse this data further
  - Else the connection **does not use TLS and is not a websocket**:
    - If the **packet is cached already**:
      - If the **cached item is not expired**:
        - Update the age of the packet and send it as response to the client
      - Else the **cached item is expired**:
        - Remove from cache and continue as if it was never cached
    - Else **packet is not cached**:
      - Send request to server
      - On server response:
        - If the **response packet is cache-able** (based on header params):
          - If **response is chunked**:
            - Send each chunk to the user as it arrives
            - Temporarily store each chunk
            - When transmission is complete -> add all chunked data to cache
          - Else **response is not chunked**
            - Send response packet to client
            - Add packet to cache
        - Else the **response packet is not cache-able**:
          - Pipe the response to the user
- Else the requested **URL is blacklisted**:
  - Send a `403 FORBIDDEN` response to the client
  - Close the connection

Installation

Requires:

- NodeJS 12.13.1+
- npm (if for some reason it wasn't bundled with Node)

Set up:

1. Open a terminal in the directory containing `proxy.js` and `package.json`
2. Run `npm install` (may require sudo)
3. Run `npm start`

Usage

Commands:

- `block <domain>` - adds specified domain to blocklist
- `unblock <domain>` - removes domain from blocklist
- `cache` - enables caching
- `nocache` - disables caching
- `verbose` - prints all connections to console
- `noverbose` - disables printing of all connections to console
- `timing` - print timing data of cache hits/misses
- `notiming` - disables printing of cache hit/miss timing data
- `showsaving` - shows how many bytes have been served from cache
- `showcachesize` - shows the current cache size

Testing

In order to test the performance of the proxy and the effect caching has on responsiveness, I loaded some websites with and without caching.
All tests were conducted with Firefox 73.0.1 and repeated 3 times with a restart of the proxy used between each to clear the cache. The `noverbose` flag was enabled for all tests to reduce clutter.
I ran the same test on the following sites:

- www.example.com
  - No-Cache
    - Webpage
      - `99ms`, `100ms`, `99ms`
      - Average: ~`100ms`
    - Favicon
      - `532ms`, `550ms`, `565ms`
      - Average: ~`545ms`
  - Cache
    - Webpage
      - `0ms`, `1ms`, `1ms`
      - Average: ~`0.5ms`
    - Favicon
      - `550ms`, `541ms`, `511ms`
      - Average: ~`540ms`
      - This one had a header requesting not to be cached

        - Bandwith saved: `3,162 Bytes`
- www.writephponline.com
    - Notes:
        - There are lots of requests for various assets on this page so the following are averages
        - Most of the requests under the hood are actually HTTPS or not cache-able
    - No-Cache
        - `1880ms`, `2200ms`, `2280ms`
        - Average: ~`2100ms`
    - Cache
        - `1520ms`, `1280ms`, `1400ms`
        - Average: ~`1400ms`
    - Bandwitdh saved: `3,818 Bytes`

Code

```
process.env.UV_THREADPOOL_SIZE = 1000
const net = require('net')
const fs = require('fs')
const path = require('path')
const { exec } = require('child_process')
const server = net.createServer()
const port = 4000
const stdin = process.openStdin()
const blockListName = 'blockList.json'
const blockListPath = path.join(__dirname, blockListName)
let verbose = true
let caching = true
let timing = true
let currentCacheSize = 0
let bytesSavedFromNetwork = 0 //how much bandwidth has been saved
let suppressErrs = true

/**
 * Read and return blocklist
 * @return {{blockedURLs:Array}}
 */
let readBlockList = () => {
    return JSON.parse(fs.readFileSync(blockListPath, 'utf8'))
}
let blockList = readBlockList()

let cache = []

//No clients connected
server.on('close', () => {
    console.log(`All clients disconnected`)
})

//Something broke
server.on('error', (err) => {
    console.error({ ERROR: err })
    exec('npm start', (err, stdout, stderr) => {
        if (err) {
            console.error(err)
        } else {
            console.log(`stdOUT: ${stdout}`)
            console.log(`stdERR: ${stderr}`)
        }
    })
})

//new connection to server
server.on('connection', (clientProxyConn) => {
    //create the connection

    clientProxyConn.once('data', (data) => {
        let theData = data.toString()
        let reqData = getAddrAndPort(theData)
        //if it's blacklisted, send 403 and kill connection
        if (blockList.blockedURLs.includes(reqData.host)) {
            clientProxyConn.write('HTTP/1.1 403 FORBIDDEN\r\n\r\n')
            clientProxyConn.end()
            clientProxyConn.destroy()
            if (verbose)
                console.log({
                    Message: 'Connection Blocked',
                    Hostname: reqData.host,
                    Port: reqData.port,
                    HTTPS: reqData.isHTTPS
                })
            return
        }

        let toServerConn = net.createConnection({
            host: reqData.host,
            port: reqData.port
        }, () => {
            //if is HTTPS, confirm connection
            //else send the request to the server
            if (reqData.isHTTPS || !caching) {
                if (reqData.isHTTPS)
                    clientProxyConn.write('HTTP/1.1 200 OK\r\n\r\n')
                //Don't manually handle subsequent data streams, this is easier, faster and uses less memory
                //readableSrc.pipe(writableDest)
                clientProxyConn.pipe(toServerConn).pipe(clientProxyConn)
            } else {
                if (isWebsocketRequest(data)) {//don't cache websockets, or headers that request 'no-cache'
                    console.log(`Not caching websocket request for: ${reqData.rawURL}`)
                    clientProxyConn.pipe(toServerConn).pipe(clientProxyConn)
                } else { //need to manually handle chunked data
                    let cachedRes = getFromCache(reqData.rawURL)
                    let startTime = Date.now()
                    if (!cachedRes) { //response not already cached
                        toServerConn.write(data)
                        let dataWhole = []
                        let isChunked = false
                        let cacheableResponse = false
                        let checkedCachability = false
                        toServerConn.on('data', (resData) => {
```

```
        if (!checkedCachability) { //need to check if we can cache it
            cacheableResponse = isCacheableResponse(resData)
            checkedCachability = true
        }

        if (!cacheableResponse) { //not cacheable
            console.log(`Packet from '${reqData.rawURL}', does not allow caching`)
            clientProxyConn.write(resData)
            clientProxyConn.pipe(toServerConn).pipe(clientProxyConn)
        } else {
            if (isChunked || resData.toString().includes('Transfer-Encoding: chunked\r\n')) { //for chunked data need to cache differently
                console.log('Chunked data incoming!')
                clientProxyConn.write(resData)

                dataWhole.push(resData)
                if (!isChunked)
                    isChunked = true

                if (resData.toString().slice(-5) == '0\r\n\r\n') { //end of chunked encoding
                    addToCache(dataWhole[0], reqData.rawURL, dataWhole.splice(0, 1))
                }
            } else {
                // console.log('multiple requests in same connection, oh no!')
                let rawURL_arr = [reqData.rawURL]
                let startTimeArr = [startTime]
                clientProxyConn.on('data', (data) => {
                    let tmpDetails = getAddrAndPort(data.toString('binary'))
                    rawURL_arr.push(tmpDetails.rawURL)
                    startTimeArr.push(Date.now() / 1000)
                    toServerConn.write(data)
                })
                // console.log(resData.toString())
                clientProxyConn.write(resData)
                let tmpRawURL = rawURL_arr.shift()
                if (timing) {
                    // console.log({ url: reqData.rawURL, cached: false, time: `${(Date.now() - startTime).toString()}ms` })
                    console.log({ url: tmpRawURL, cached: false, time: `${(Date.now() - startTimeArr[0]).toString()}ms` })
                    startTimeArr.shift()
                }
                addToCache(resData, tmpRawURL)

                // clientProxyConn.pipe(toServerConn).pipe(clientProxyConn)

            }
        }
    })
} else {
    if (cachedRes.chunkArr) { //data was chunked, send each TCP packet
        clientProxyConn.write(cachedRes.cachedStr)
        cachedRes.chunkArr.forEach(e => {
            clientProxyConn.write(e)
        })
    } else {
        // console.log(cachedRes.toString())
        clientProxyConn.write(cachedRes)
    }
```

```
                        if (timing)
                            console.log({ url: reqData.rawURL, cached: true, time: `${(Date.now() - startTime).toString()}ms` })
                    }
                }
            }

            if (verbose)
                console.log({
                    Message: 'Connection Established',
                    Hostname: (reqData.isHTTPS ? reqData.host : reqData.rawURL),
                    Port: reqData.port,
                    HTTPS: reqData.isHTTPS
                })
            toServerConn.on('error', (err) => {
                if (!suppressErrs)
                    console.error({ 'Server Error': err })
            })
            toServerConn.on('close', () => {
                console.warn({ 'Server Closed Conn': `${reqData.host}:${reqData.port}` })
            })
        })
        clientProxyConn.on('error', (err) => {
            if (!suppressErrs)
                console.error({ 'Client Error': err })
        })
        clientProxyConn.on('close', () => {
            console.warn({ 'Client Closed Conn': `${reqData.host}:${reqData.port}` })
        })
    })
})

/**
 * Parses out: hostname, port and if a connection is HTTPS
 * @param {string} data data object stringified
 * @returns {{host:string, port:string, isHTTPS:boolean, rawURL:string}} hostname, port, whether
 *  the connection is using HTTPS and the full path trying to be accessed (if HTTP)
 */
let getAddrAndPort = (data) => {
    let hostData = []
    /*
     * Cannot actually read the data if using TLS but
     * HTTPS connections contain the keyword 'CONNECT'
     */
    hostData['isHTTPS'] = data.indexOf('CONNECT') !== -1
    if (hostData.isHTTPS) {
        let splitStr = data.split(` `)[1].split(`:`)
        hostData['host'] = splitStr[0]
        hostData['port'] = splitStr[1]
    } else {
        // console.log(data)
        hostData['rawURL'] = data.split('http://')[1].split(' ')[0]
        let splitStr = data.split(`Host: `)[1].split(`\r\n`)[0].split(`:`)
        hostData['host'] = splitStr[0]
        //HTTP defaults to port 80 but just in case...
        hostData['port'] = splitStr[1] ? splitStr[1] : '80'
    }
    return hostData
}

server.listen(port, () => {
    console.log(`Server running on: ${server.address().address !== '::' ? server.address().address : 'localhost'}:${server.address().port}`)
})

stdin.addListener('data', (data) => {
    handleInput(data.toString().trim())
})


/**
 * Handles the strings input by the user
 *
 * @param {String} consoleInput The console input stringified and trimmed
 *
 * Commands:
 * block <domain> - adds domain to blocklist
 * unblock <domain> - removes domain from blocklist
 * cache - enables caching
 * nocache - disables caching
 * verbose - prints all connections to console
 * noverbose - disables printing of all connections to console
 * timing - print timing data of cache hits/misses
 * notiming - disables printing of cache hit/miss timing data
 * showsaving - shows how many bytes have been served from cache
 * showcachesize - shows how big the current cache is
 */
let handleInput = (consoleInput) => {
    let splitData = consoleInput.split(' ')
    let keyword = splitData[0]
    let param = splitData[1]
    switch (keyword) {
        case 'block':
            if (!blockList.blockedURLs.includes(param)) {
                blockList.blockedURLs.push(param)
                writeBlockList(blockList)
            } else {
                console.warn(`${param}, has already been blocked!`)
            }
            break
        case 'unblock':
            if (blockList.blockedURLs.includes(param)) {
                blockList.blockedURLs.splice(blockList.blockedURLs.indexOf(param), 1)
                writeBlockList(blockList)
            } else {
                console.warn(`${param}, was not blacklisted!`)
            }
            break
        case 'verbose':
            verbose = true
            break
        case 'noverbose':
            verbose = false
            break
        case 'cache':
            caching = true
            break
        case 'nocache':
            caching = false
            break
        case 'timing':
            timing = true
            break
        case 'notiming':
            timing = false
            break
        case 'showsaving':
            console.log({ 'Saved Bytes': bytesSavedFromNetwork.toLocaleString() })
            break
        case 'showcachesize':
            console.log({ 'Cache Size': currentCacheSize.toLocaleString() })
            break
        default:
            console.error(`Input not recognised: ${keyword}, is not a keyword`)
            break
    }
}

/**
 * Write updated blocklist
 * @param {blockList}
 */
let writeBlockList = (blockList) => {
    fs.writeFile(blockListPath, JSON.stringify(blockList), (err) => {
        if (err)
            console.error(`Could not write updated blocklist to disk!`)
        else
            console.log(`Updated blocklist written to disk`)
    })
}

/**
 * @param {string} url The requested URL
 * @return {Buffer|{cachedStr:Buffer, chunkArr:Array<Buffer>}}
 */
let getFromCache = (url) => {
    if (!caching)
        return false
    if (cache[url]) {
        let tmpCache = cache[url]
        if (tmpCache.expiryTime > (Date.now() / 1000)) {
            console.log(`Cached data for ${url}, found`)
            let cachedStr = tmpCache.firstHalfData + (Math.floor(Date.now() / 1000) - tmpCache.startTime) + tmpCache.secondHalfData

            cachedStr = Buffer.from(cachedStr, 'binary')
            // console.log({ cacheURL: url, cachedStr: cachedStr, str: cachedStr.toString('binary') })
            bytesSavedFromNetwork += tmpCache.size
            //handle chunked and non-chunked data differently
            if (!tmpCache.chunkArr)
                return cachedStr
            return { cachedStr: cachedStr, chunkArr: tmpCache.chunkArr }
        } else { //data is old
            console.log(`Cached data for ${url}, expired... purging`)
            cache = cache.splice(cache.indexOf('url'), 1)
            return false
        }
    }
    return false
}

/**
 * @param {Buffer} responseBuffer The raw data response from the server
 * @param {string} url The url the request if for
 * @param {Array<Buffer>} chunkArr The chunks for a chunked response
 */
let addToCache = (responseBuffer, url, chunkArr = false) => {
    let parsedBuffer = responseBuffer.toString('binary')
    let parsedTotal = parsedBuffer.split('\r\n\r\n')
    let parsedBody = parsedTotal[1]
    let parsedBufferHead = parsedTotal[0] //extract the header
    if (parsedBufferHead.includes('404 Not Found\r\n')) {//dont cache 404
        // console.log(`Not caching 404 responses`)
        // return
    }
    if (parsedBufferHead.includes('Cache-Control: max-age=')) {
        let expiryTime = parsedBufferHead.split('Cache-Control: max-age=')[1]
        if (expiryTime && expiryTime.split('\r\n', 1)[0]) {
            let size = responseBuffer.length
            if (chunkArr)
                chunkArr.forEach(e => {
                    size += e.length
                })
            expiryTime = expiryTime.split('\r\n', 1)[0].split(',')[0]
            expiryTime = parseInt(expiryTime) + Math.floor((Date.now() / 1000))
            if (parsedBufferHead.includes('Age: ')) { //header includes age, this is ideal
                var ageSplit = parsedBufferHead.split('Age: ')

                var secondHalfData = ageSplit[1].split('\r\n')
                expiryTime -= parseInt(secondHalfData[0])
                var startTime = Math.floor(Date.now() / 1000) - parseInt(secondHalfData[0])
                secondHalfData.splice(0, 1)
                secondHalfData = secondHalfData.join('\r\n') + '\r\n\r\n' + parsedBody.toString('binary')
            } else { //header does not include age, this is not ideal
                let theHeaderArr = parsedBufferHead.split('Cache-Control: max-age=')
                var startTime = 0
                theHeaderArr[0] += 'Cache-Control: max-age='
                theHeaderArr[1] = theHeaderArr[1].split('\r\n')
                theHeaderArr[0] += theHeaderArr[1][0] + '\r\n'
                var ageSplit = []
                ageSplit[0] = theHeaderArr[0]
                var secondHalfData = parsedBuffer.slice(ageSplit[0].length)
                // console.log({ secondHalf: secondHalfData })
            }

            cache[url] = {
                expiryTime: expiryTime,
                firstHalfData: ageSplit[0] + 'Age: ',
                secondHalfData: '\r\n' + secondHalfData,
                startTime: startTime,
                chunkArr: chunkArr,
                size: size
            }
            currentCacheSize += size
            console.log({ CachedURL: url, Size: `${size.toLocaleString()} bytes` })
        } else {
            // console.log(parsedBufferHead)
            console.log(`Could not cache response from: ${url}, due to header parameters`)
        }
    } else {
        // console.log(parsedBufferHead)
        console.log(`Could not cache response from: ${url}, due to header parameters`)
    }
}

/**
 * Determines if a HTTP request is for a websocket
 * @param {Buffer} rawData The raw request data
 * @param {Boolean} allowNoCache Respond true to no cache requests
 */
let isWebsocketRequest = (rawData) => {
    let stringifiedData = rawData.toString()
    if (stringifiedData.includes('Upgrade: websocket\r\n') || stringifiedData.includes('Connection: upgrade\r\n'))
        return true
    return false
}

/**
 * Determines if a response wants to be cached
 * @param {Buffer} rawData
 */
let isCacheableResponse = (rawData) => {
    let stringifiedData = rawData.toString()
    if (stringifiedData.includes('Cache-Control: no-cache\r\n') || stringifiedData.includes('Pragma: no-cache\r\n') || !stringifiedData.includes('Cache-Control: max-age='))
        return false
    return true
}
```