CSU34031 Advanced Telecommunications

Web Proxy Server - John Sinclair - 16325734

The repository for this project can be found at https://github.com/johnmarksinclair/web-proxy

Task

Implement a web proxy server which fetches items from the web on behalf of a web client instead of the client fetching them directly. This allows for caching of pages and access control.

The program should be able to:

1. Respond to HTTP & HTTPS requests and display each request on a management console. It should forward the request to the web server and relay the response to the browser.
2. Handle websocket connections.
3. Dynamically block selected URLs via the management console.
4. Efficiently cache HTTP requests locally and thus save bandwidth. You must gather timing and bandwidth data to prove the efficiency of your proxy.
5. Handle multiple requests simultaneously by implementing a threaded server.

Overview

I implemented this server in NodeJS using the net and axios modules. The net module is used to create both servers and clients, and axios is a JavaScript library for making HTTP requests from NodeJS, based on the Promise API.

The net module allows for creation of servers, this proxy server listens to port 8080 and when a new connection is made (multiple are possible per client) it awaits a data request. This request is then processed using a server method, the processed or formatted request is printed to the management console and if the requested site is not blocked it creates a connection between the client and server, if it is blocked the server sends a 403 forbidden response to the client, prints the blocked request message to the management console and terminates the client connection using the destroy() method, ensuring no more activity occurs on this socket.

Requests

When a connection is made between a client and the server, the server will process the request differently depending on the type of request.

HTTPS requests - First the connection is confirmed by sending a status 200 "OK" message and then the following data streams are piped directly from the server to the client connection and vice versa. This is because there is no need to manually handle these packets as to do so would take longer and use more server memory.

HTTP requests - WebSockets begin as a standard HTTP request and response. The client asks to open a connection and the server responds. If successful the connection is used as a WebSocket connection. WS connections are established by upgrading an HTTP request response pair, this is initiated by the client sending a HTTP request using certain headers like "Connection: Upgrade" and "Upgrade: websocket". So our servers request processor method parses incoming HTTP requests to set the "ws" field of the processed request to true if the incoming request contains the keyword "websocket".

- WebSocket requests - if a WebSocket request is detected, all subsequent data streams are piped directly from the server to the client connection and vice versa.
- Standard HTTP requests - if the request is just a standard HTTP request the server first checks whether the requested URL is cached:
    - Cached - if cached the previously cached response body is written to the client connection and the "serving cached site" message along with information on time and bandwidth savings is printed to the management console. The connection is then closed.
    - Uncached - if not cached the clock is started (for cache timing data) and the request is performed. When the request receives a response the clock is stopped and the body of the response and request time is stored. The body of the response is then written to the client and the connection is ended.

Dynamic Blocking

The proxy server supports dynamic blocking via the management console. Using the commands `/b` and `/u` in the management console the proxy admin block and unblock specified URLs or domains. Additionally the admin can use the command `/sb` to print a list of the currently blocked domains to the console.

Below is a snapshot of the output seen on the management console:

```
server listening on 8080
[
  type: 'http',
  ws: false,
  url: 'http://example.com/',
  host: 'example.com',
  port: '80'
]
    requested site is not blocked
    bandwidth used: 2512 bytes
    time taken: 206 ms
    closed: example.com
/b example.com
    blocked example.com
[
  type: 'http',
  ws: false,
  url: 'http://example.com/',
  host: 'example.com',
  port: '80'
]
    requested site is blocked
    closed: example.com
```

As you can see, after the administrator inputs the command `/b example.com` the clients are unable to access example.com and receive an `ERROR 403` access denied message instead.

Cache Efficiency

I implemented a cache using a JavaScript map, pairing a URL with the body of the response received. Since the cached response body is in the form of characters the bandwidth saved is the length of the body multiplied by 2 (the number of bytes a char takes up in memory).

Below is an example of the output seen on the management console when www.example.com is loaded twice, the first where a request needs to be made and the second where a cached response can be served.

```
[
  type: 'http',
  ws: false,
  url: 'http://example.com/',
  host: 'example.com',
  port: '80'
]
    requested site is not blocked
    bandwidth used: 2512 bytes
    time taken: 216 ms
    closed: example.com
[
  type: 'http',
  ws: false,
  url: 'http://example.com/',
  host: 'example.com',
  port: '80'
]
    requested site is not blocked
    serving cached site
    bandwidth saved: 2512 bytes
    time taken: 0 ms
    216 ms saved by caching
    closed: example.com
```

As you can see serving the cached response is far quicker and saves bandwidth. I tested loading sites three times and recorded the results:

- www.example.com
    - Caching disabled: 327 ms, 225 ms, 354 ms
        - Total: 906 ms, 7536 bytes
    - Caching enabled: 331 ms, 1 ms, 1 ms
        - Total: 333 ms, 2512 bytes
    - Efficiency: 573 ms and 5,024 bytes saved
- www.example.org
    - Caching disabled: 194 ms, 386 ms, 338 ms
        - Total: 918 ms, 7536 bytes
    - Caching enabled: 236 ms, 1 ms, 1 ms
        - Total: 238 ms, 2512 bytes
    - Efficiency: 680 ms and 5,024 bytes saved

Threading

This proxy server facilitates simultaneous requests via multithreading, the NodeJS net module selects an available thread to handle each event as it occurs. These threads are asynchronous, allowing for multiple connections and simultaneous requests.

Installation

Requires NodeJS and yarn or npm. First run `yarn install` in the root of the cloned repository in order to add the required dependencies. `yarn start` will then start the server, listening to port 8080 (configurable). Configure your computer or browser proxy settings and the traffic will be displayed in the terminal.

Management Commands

`/b example.com` - blocks the specified url

`/u example.com` - unblocks the specified url

`/sb` - shows a list of blocked urls in the management console

`/sc` - shows a list of the urls that are cached

`/cc` - clears the cache

`/ss` - shows the current proxy time and bandwidth savings

Code

```
const net = require("net");
const axios = require("axios");
const readline = require("readline");
const buffer = "    ";
// specify port to listen to
const port = 8080;
// creating the server
const server = net.createServer();
```

I first imported the required packages and initialised constants such as the port and buffer for console print outs. I then initialize the server using the createServer() method.

```
// set the server to listen to specific port
server.listen(port, () => {
  console.log(`server listening on ${port}`);
});
// setting the server to perform actions upon new connections
server.on("connection", (clientConnection) => {
  // when the server receives data (requests) perform operations
  clientConnection.once("data", (data) => {
    // process the raw request
    let processed = reqFormatter(data.toString());
    // display the formatted request to the management console
    console.log(processed);
    // check if the requested site is blocked
    if (!isBlocked(processed.url)) {
      console.log(buffer + "requested site is not blocked");
      // if not blocked create a connection using the server host and port info
      let serverConnection = net.createConnection(
        {
          host: processed.host,
          port: processed.port,
        },
        () => {
          // check request type
          if (processed.type === "https") {
            // perform HTTPS handshake and pipe further packets
            clientConnection.write("HTTP/1.1 200 OK\r\n\n");
            clientConnection.pipe(serverConnection).pipe(clientConnection);
          } else {
            if (processed.ws) {
              // pipe any websocket connection requests directly from client to
server
              clientConnection.pipe(serverConnection).pipe(clientConnection);
            } else {
              if (processed.url === "/" || processed.url === "/favicon.ico") {
                clientConnection.end("http proxy server");
              } else {
                // check whether cached response available
                if (isCached(processed.url)) {
                  cachedHandler(processed, clientConnection);
                } else {
                  uncachedHandler(processed, clientConnection);
                }
              }
            }
          }
        }
      );
      serverConnection.on("error", (err) => {
```

```
        console.log(err);
      });
      serverConnection.on("close", () => {
        console.log(buffer + `closed: ${processed.host}`);
      });
    } else {
      // if requested site is blocked
      console.log(buffer + "requested site is blocked");
      // serve error 403 to client
      clientConnection.write("HTTP/1.1 403 FORBIDDEN\r\n\r\n");
      // end and destroy the connection preventing further activity
      clientConnection.end();
      clientConnection.destroy();
    }
    clientConnection.on("error", (err) => {
      console.log(err);
    });
    clientConnection.on("close", () => {
      console.log(buffer + `closed: ${processed.host}`);
    });
  });
});
```

The above function sets the server to listen for new connections and run a function when a new connection is detected. When the connection receives a request it formats it with a function I wrote to parse the useful information from the incoming requests. The server then checks whether the requested site is blocked. If the site is blocked an error 403 status code is written to the client and the connection is destroyed, preventing any further activity on the socket. If the requested site is not blocked it processes the request differently depending on the type. The formatting function sets the “type” field in the formatted request object to “https”, “http” or “ws” accordingly. Secured and websocket connections are directly piped from the server to the client and http requests are sent to either the uncached or cached handler functions accordingly.

```
// handles cached http requests and wrties responses to the passed client
// params: processed: the processed request, clientConnection: connection info
const cachedHandler = (processed, clientConnection) => {
  let start = new Date().getTime();
  console.log(buffer + "serving cached site");
  // serve the cached response to the client
  clientConnection.write(getCachedSite(processed.url));
  let end = new Date().getTime();
  console.log(buffer + `time taken: ${end - start} ms`);
  let time = cachedTimes.get(processed.url);
  console.log(buffer + `${time - (end - start)} ms saved by caching`);
  timeSaved += time - (end - start);
  // close the client connection
  clientConnection.end();
};
```

Above you can see the function used to handle cached requests. It first logs the current timestamp in a variable for timing purposes. It then calls the getCachedSite function and writes the return value to the client. Timing data is then stored and the connection is closed.

```
// handles uncached http requests and writes responses to the passed client
// params: processed: the processed request, clientConnection: the connection
info
const uncachedHandler = (processed, clientConnection) => {
  let start = new Date().getTime();
  // perform request for uncached request
  axios
    .get(processed.url)
    .then((response) => {
      // cache the response data and associate with url
      cacheSite(processed.url, response.data);
      // serve the response to the client
      clientConnection.write(response.data);
      let end = new Date().getTime();
      console.log(buffer + `bandwidth used: ${response.data.length * 2}
bytes`);
      console.log(buffer + `time taken: ${end - start} ms`);
      // note request time for cache analysis
      cacheTime(processed.url, end - start);
      // close the clients connection
      clientConnection.end();
    })
    .catch((error) => {
      console.log("error occurred");
    });
};
```

Above is the uncached HTTP request handler. It uses the axios module to perform a request to the host. The response is then cached and the data is written to the client before the connection is closed.

```
// request processing/ formatting method
// params: data: raw request data
// returns: formatted request array object
const reqFormatter = (data) => {
  //console.log(data);
  let processed = [];
  if (data.includes("CONNECT")) processed["type"] = "https";
  else processed["type"] = "http";
  if (data.includes("websocket")) processed["ws"] = true;
  else processed["ws"] = false;
  if (processed.type === "https") {
    let host = data.split("CONNECT ")[1].split(":")[0];
```

```
    processed["url"] = host;
    processed["host"] = host;
    processed["port"] = data.split(":")[1].split(" ")[0];
  } else {
    processed["url"] = data.split(" ", 2)[1];
    processed["host"] = data.split("Host: ")[1].split("\r\n")[0];
    processed["port"] = "80";
  }
  return processed;
};
```

This is the function I wrote to parse the important information from the raw incoming requests. It checks for identifying keywords to determine the request type and stores the requested url, host and port information.

```
// init the management console input
const rl = readline.createInterface(process.stdin, process.stdout);

// check for management console commands and route accordingly
rl.on("line", (inp) => {
  if (inp.includes("/b")) block(inp.substring(3));
  else if (inp.includes("/u")) unblock(inp.substring(3));
  else if (inp.includes("/sb")) showBlocked();
  else if (inp.includes("/sc")) showCached();
  else if (inp.includes("/cc")) clearCache();
  else if (inp.includes("/ss")) showStats();
  else if (inp.length == 0) console.log(buffer + "input a command");
  else console.log(buffer + "invalid command: " + inp);
});
```

The first line of the above code is where I initialise the input interface for the administrator in the management console. The next function waits for a line to be inputted in the console and processes the requests depending on the command prefixes available.

```
// array of blocked sites/ domains
var blocked = [];
// blocks specified url or domain
// params: inp: admin inputted domain or url to be blocked
const block = (inp) => {
  if (!isBlocked(inp)) {
    blocked.push(inp);
    console.log(buffer + `blocked ${inp}`);
  } else {
    console.log(buffer + `${inp} is already blocked`);
  }
};
```

```
// unblocks url or domain
// params: inp: admin inputted domain or url to be unblocked
const unblock = (inp) => {
  if (isBlocked(inp)) {
    blocked = blocked.filter((ele) => {
      return ele != inp;
    });
    console.log(buffer + `unblocked ${inp}\n`);
  } else {
    console.log(buffer + `${inp} is not blocked`);
  }
};

// checks if specified url is blocked
// params: url: admin inputted url or domain
// returns: boolean
const isBlocked = (url) => {
  var flag = false;
  if (blocked.length > 0) {
    blocked.forEach((site) => {
      if (site.includes(url) || url.includes(site)) {
        flag = true;
      }
    });
    return flag;
  } else {
    return false;
  }
};

// prints list of currently blocked sites to the management console
const showBlocked = () => {
  console.log("blocked sites:");
  if (blocked.length > 0) {
    blocked.forEach((site) => {
      console.log(buffer + site);
    });
  } else {
    console.log(buffer + "none");
  }
};
```

The four functions above are helper functions used for dynamic blocking and unblocking of websites. The first checks whether the passed url has been blocked and if not, blocks it. The second unblocks a passed url. The next is a helper function to check whether a passed url is in the block list and the last function prints a list of the blocked domains to the management console when required.

```
var cached = new Map();
var cachedTimes = new Map();
var timeSaved = 0;
var bandwidthSaved = 0;

// caches a site
// params: url: url to be cached, body: response body for url
const cacheSite = (url, body) => {
  cached.set(url, body);
};

// records time taken to perform request
// params: url: url to cache time, time: time taken to perform request
const cacheTime = (url, time) => {
  cachedTimes.set(url, time);
};

// gets a site from the cache
// params: url: url required to serve to client
// returns: body: the body of the cached response
const getCachedSite = (url) => {
  let body = cached.get(url);
  bandwidthSaved += body.length * 2;
  console.log(buffer + `bandwidth saved: ${body.length * 2} bytes`);
  return body;
};

// checks whether a site is cached or not
// params: url: site to check if cached
// returns: boolean
const isCached = (url) => {
  if (cached.has(url)) return true;
  else return false;
};

// clears the cache
const clearCache = () => {
  cached.clear();
  cachedTimes.clear();
  console.log(buffer + "cache has been cleared");
};

// prints the currently cached site urls to the management console
const showCached = () => {
  console.log(cached.keys());
};
```

These functions are used to maintain the cache. I implemented my cache using a javascript map, so the response bodys were stored with their respective urls. The first function caches the response body, the next records the time taken by the request. The third function returns the body associated with the passed url. The forth checks whether the cache has a url's response stored in the cache. The clearCache function clears the cache and timing maps when required and the last function showCached() prints out the urls of the currently cached sites to the management console.

```
// prints the current time and bandwidth savings from serving cached sites
const showStats = () => {
  console.log(
    `${buffer + bandwidthSaved} bytes and ${timeSaved} ms saved by proxy cache`
  );
};
```

This function just prints out the current time and bandwidth savings (due to caching) to the management console