Implementation of the HTTP Protocol in Go
ELEN4017 Project Report

James Allingham (672732) and Devin Taylor (603956) - Group 11

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Abstract: Hyper Text Transfer Protocol is a fundamental component of the World Wide Web and understanding it is important not only to understanding the Internet but networking as a whole. The design, implementation and testing of an HTTP application suite including a client, server and proxy server has been presented. In addition, the code has been described and the teamwork aspects of the project discussed and reflected on. The use of the Go language allowed for a simple and elegant solution with its `goroutines`, powerful high level features such as `map`s and garbage collection, and fast run time. A subset of HTTP large enough to perform basic requests and responses in HTTP/1.1 was implemented. However, there is a large room for improvements in this aspect as well as in the the limited features of the proxy server. The web client was able to access pages hosted on both the Internet and the server. Testing with Wireshark revealed that all of the systems worked as expected. In addition, the various configurations of connection types, protocols and proxy usage behaved as expect.

Key words: HTTP, Client-Server, Proxy Server, TCP, UDP, Go programming language

1 INTRODUCTION

Hyper Text Transfer Protocol (HTTP) is a fundamental component of the World Wide Web [1]. Understanding HTTP is vital for gaining a well rounded and deep understanding of not only the web and Internet but networking in general. Through understanding HTTP, an understanding of protocols as well as the Internet Protocol (IP) networking stack can also be acquired. A collection of web applications, including a client, a server and a proxy server, is to be implemented using a limited subset of HTTP. The goal of this project is to, as closely as possible, match the real world use case for HTTP. The design, implementation and testing of the web application suite is presented in this report along with a critical analysis and recommendations for future improvement of the system. The group work aspects of the project are also discussed.

2 HTTP DESCRIPTION

The Hyper Text Transfer Protocol has been in existence since 1990 [2]. It is used by the World Wide Web as an application level protocol for transfer of information in a distributed system. It consists of two programs: a client and a server. The client, also commonly referred to as the browser, communicates with the server by sending it an HTTP request message. The server then responds with an HTTP response message. HTTP is a stateless protocol which means that the server has no knowledge of the client other than the information contained in the request. This is a limitation of HTTP which is overcome with the use of cookies [1]. HTTP uses Transmission Control Protocol (TCP) as its underlying transport layer protocol [1]. This means that HTTP does not need to worry about reliable data transfer issues such as out of sequence packets or packet loss.

A typical HTTP communication can be described as follows:

1. A server sets up a TCP listener on one of its ports (an end point for network communication). The default port for HTTP is 80.
2. A client initiates a TCP connection with the server (via the server's url) on the appropriate port.
3. The client and server complete a 'three way handshake' after which each of them have a socket (a virtual data connection between processes) which can be used to send and receive messages. Note that the port associated with these sockets will not be 80.
4. The client sends an HTTP request message to the server using its socket. This request is received by the server on its matching socket.
5. The server processes the request and sends an appropriate HTTP response to the client. This communication once again makes use of the sockets that have been set up.
6. The server closes the TCP connection.
7. The client receives the HTTP response before the connection is closed.

2.1 Request and Response Messages

HTTP is based on communication via request and response messages, both of which have specific formats and can take on a number of values. Both request and response messages contain three parts: the request/ response line, zero or more header lines and an optional body. The format of an HTTP request is very specific, as shown in *Figure 1*. A number of common HTTP request methods are described in *Table 1*. The format of an HTTP response is once again specific, as shown in *Figure 2*. *Table 2* describes number of common HTTP response codes. The responses fall into a few categories:

- 1xx responses indicate that the client must wait for the server to finish processing the request
- 2xx responses indicate a success
- 3xx responses indicate that client must submit a new request
- 4xx responses indicate that the client is in error
- 5xx responses indicate that the server is in error

2.2 Persistent and Non-persistent

In HTTP version 1.0 (HTTP/1.0), all communication takes the form shown above [3]. In other words, a new TCP connection is made every time the client wants to make a

Table 1: Commonly used HTTP request methods

Method	Description
GET	Request data from the specified URL
HEAD	Request a response identical to the GET response but without the body so that the client can get the header values without retrieving the body
POST	Request that the web server accept the data in body, this is often used for filling in forms
PUT	Request that the web server store the data in the body at the specified URL
DELETE	Request that the web server remove the resource stored at the specified URL
TRACE	Request that the web server echo the request so that the client can detect any changes made to the original request
OPTIONS	Request that the web server inform the client which methods are valid at the specified URL
PATCH	Request that the web server apply a partial change to the resource at the given URL

Table 2: Commonly used HTTP response methods

Code	Phrase	Meaning
200	OK	HTTP request successful
201	Created	A new resource was created
202	Accepted	Request has been accepted for processing
301	Moved Permanently	The requested resource is now located elsewhere
302	Found	The requested resource is temporarily located elsewhere
304	Not Modified	The requested resource has not been modified since last requested - used for caching
400	Bad Request	The request was not understood by the server
404	Not Found	The requested resource could not be found
503	Service Unavailable	The server is temporarily unavailable
505	HTTP Version Not Supported	The server does not support the HTTP version of the request

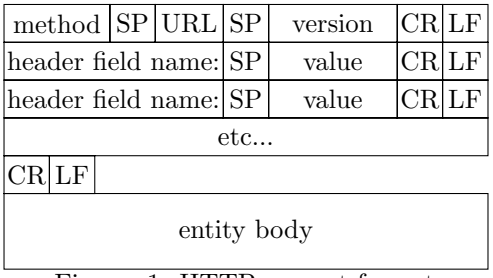

Figure 1: HTTP request format

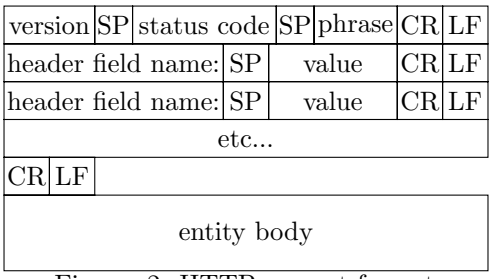

Figure 2: HTTP request format

request to the server. However, this approach has its disadvantages. The primary disadvantage is that it is wasteful to repeatedly make new connections when more than one request is going to be made. This is because setting up the TCP connection requires the three way handshake which is a time consuming operation. The solution to this, which was implemented in HTTP version 1.1 (HTTP/1.1), was to allow connections to persist for multiple request-response pairs [2]. This is accomplished using the *connection* header field, which can take on the values *close* and *keep-alive* for non-persistent and persistent connections respectively.

2.3 Proxy Servers and Caching

Another limitation of HTTP that results from its statelessness is that it does not have a mechanism for 'smart' communication. Consider the following situation:

1. A client requests the file `Foo.bar`.
2. The server responds by sending the file to the client by encapsulating it withing an HTTP response message.
3. The same client receives the file and immediately requests the same file, `Foo.bar`, again.
4. The server responds by sending the exact same file to the client again.

The problem with the above situation is that the server is wasting time sending the client the file `Foo.bar` again because no changes have been made to the file. As a result, the server could be slower to respond to other clients. This also increases congestion on the uplink of the Local Area Network of the client. The solution is to make use of a Proxy server (proxy), also known as a web cache. In this situation, all HTTP requests are sent to the proxy by default. The proxy then forwards requests to the destination servers. The proxy acting as an intermediary between the clients and server can now store files sent from a server to a client. Now when a client requests the same file in quick succession, the server need only send it once. This solves both problems: the server has a reduced load and the uplink traffic is also reduced. Additionally, because LANs usually have network speeds orders of magnitude larger than the uplink, the client gets the data faster. By making use of the `Last-Modified` and `If-Modified-Since` headers, the proxy can make sure that it always serves the client with the most up-to-date version of the requested file. This

is accomplished with the *Conditional-Get* request.

2.4 User Datagram Protocol

Although HTTP makes use of TCP as it's transport layer protocol, an HTTP-like communication system can be implemented with the User Datagram Protocol (UDP). This would have the disadvantage that it provides unreliable communication. However, it could be faster than if TCP were used.

3 SYSTEM OVERVIEW

The system is implemented using HTTP/1.1. Some HTTP/1.0 features are still implemented for backwards compatibility and not all HTTP/1.1 features have been implemented.

3.1 System Architecture

As mentioned, the HTTP protocol follows the Client-Server model. This means that to properly demonstrate the implementation of HTTP, both a client and server are required. Additionally, in order to more extensively represent a real world application of HTTP, a proxy is required. All three of these systems have been implemented in the Go programming language developed by Google. It is a statically typed, imperative, garbage collected, compiled language. It was designed as a systems programming language with specific support for networking and multi-threading.

Client: The client is responsible for sending HTTP requests to the server. These requests are generated based on user input. This process involves creating the HTTP request in the correct format, as discussed in *Section 2.1*, creating a TCP or UDP connection and then sending the request in byte format. The client then waits for a response from the server, decomposing the response into its elements various elements, mentioned in *Section 2.1*, and takes an appropriate action based on the response type and contents.

Server: The server is responsible for listening for both TCP and UDP connections. Once a connection of either kind is made, the server must handle the client which involves, in the case of TCP, creating a new socket for the communication, and in the case of either TCP or UDP, receiving the HTTP request. The HTTP request must then be decomposed into its elements, discussed in *Section 2.1.* Based on the type of request and its contents, the server must take an appropriate action as well as compose and send an HTTP response. Then, in the case of non-persistent TCP connections only, the sever must close the connection.

Proxy Server: The proxy is responsible for taking all incoming requests from the client and forwarding them to the destination server. The proxy server must also cache web objects sent, via itself, from the server to the client. This makes use of the `If-Modified-Since` and `Last-Modified` header fields to allow the proxy to have the most up-to-date version of any object in its cache. The proxy acts as both a client and a server in the sense that it is viewed as a server from the perspective of the client and vice versa. As a result, it has many of the responsibilities listed above.

3.2 Features

A number of features were implemented. The ability to use both TCP and UDP was implemented for the client and server. The ability to use both persistent and non-persistent TCP was implemented on the client, server and proxy. Multi-threading was implemented on the server and proxy allowing them to server multiple clients. The ability to track the round-trip-time (RTT) of a message was implemented on the client.

TCP vs UDP: This feature allows the user of the client to select the underlying transport layer protocol to use. TCP has the advantage that it offers some robustness in the form of protection against packet loss and delay. On the other hand, UDP offers a higher speed than TCP due to its lack of data reliability services. The server implements this feature in terms of the ability to listen for both TCP and UDP connections and handle each appropriately.

Persistent vs Non-persistent: The choice of persistent TCP over non-persistent TCP allows the communication between the client and the server to be sped up for sessions with more than one request. This speed up is a result of not having to perform the three-way handshake every time a request message is sent from the client to the server. This is implemented using the header fields discussed in *Section 2.2* as well as Multi-threading discussed in the section below.

Multi-threading: Multi-threading, implemented via `goroutines`, allows the server (as well as the proxy) to be parallelized. Effectively, this feature allows the server to handle multiple TCP connections at the same time (while also serving clients via UDP connections). This makes the use of persistent TCP feasible as it allows the server to keep listening for new TCP connections even while handling a long persistent communication with the client.

RTT timer: This feature allows the client to time how long it takes to receive the response to a message. This could be useful for time out periods or for comparing different types and combinations of communication schemes i.e. TCP vs UDP, persistent vs non-persistent, and proxy vs no proxy.

4 DETAILED IMPLEMENTATION

HTTP is a large, complex protocol that has evolved over many years. As a result, only a subset of the features were implemented in this system. The implemented features will be discussed in detail and important unimplemented features will be mentioned. Additionally, the Go language

and the implications of implementing the system in this language will be discussed. For example Go is a garbage collected language which greatly simplified the code due to the lack of manual memory management. Additionally, because Go is compiled and not interpreted it has a very fast run time.

4.1 Client

The client must be capable of dealing with a number of scenarios when making requests to a server. If the server responds with a `503 Service Unavailable` then the client must resend the request as this is usually a temporary state. On the other hand if the server responds with a `301 Moved Permanently` or a `302 Found`, the client must resend the the request to the new location specified by the `location` header. If neither of those cases are true, then the client must get the header of the response message and using the `Content-Length` and `Transfer-Encoding` headers take an appropriate action to ensure that the whole response is received. Following this, if the response is a `200 OK`, the client must search through the HTML for additional objects which must also be requested. This complex process is managed by recursively calling the `handleRequest` function which takes into account all of the above behaviors. *Algorithm* 1 describes this behavior. Additionally the client must also be aware of whether or not the connection is set to `keep-alive` or `close` and keep the connection alive or close it respectively after the message has been received. When the response to a GET message is received, the contents are written to disk. Note that a `map` is used to store the sources host names and the files to retrieve from them. Go `map`s are implemented using Hash tables which means that they have a search time of $\mathcal{O}(1)$ [4].

Algorithm 1 `handleRequest` Pseudo Code

```
handleRequest(method, url, body, host)
conn ← Dial(host)
request ← createRequest(method, url, body, host)
conn.Write(request)
response ← conn.Read()
if responseMethod == 503 then
    handleRequest(method, url, body, host)
    return
end if
if responseMethod == 301 or 302 then
    newURl ← responseLocation
    handleRequest(method, newUrl, body, host)
    return
end if
if responseTransferEncoding == chunked then
    keep reading until EoF marker
else if responseContentLength ≥ bodySize then
    keep reading until responseContentLength == bodySize
end if
for each source in body do
    handleRequest(GET, srcURL, , srcHost)
end for
```

The client allows the user some flexibility with regards to how to communicate. The user is able to set the following settings:

- protocol: TCP or UDP
- connection: persistent or non-persistent
- proxy: on or off

The settings data is stored in a text file. When the UDP protocol is selected, the connection used will be non-persistent regardless of the user's choice. When selecting the proxy the user must either specify 'off' or the host address and port of the proxy server.

The client is missing some important functionality. The most important of which is HTTP Secure (HTTPS), which uses Secure Socket Layer (SSL). A large amount of websites on the Internet now require SSL such as Google and Wikipedia. Without HTTPS the client is unable to access these websites. The main advantage of HTTPS is the improved security and authentication of the visited website which improves privacy and integrity in the exchange of data. This is an increasingly important aspect of Internet communication. This was not added due to time constraints.

Another important feature that was not implemented was timeout periods. The client does not have a time out period after which it will cease to wait for a response. This means that if the server's response does not reach the client, it will continue to listen for it. This could be easily implemented using the RTT timing code, however, due to time constraints this was not completed.

4.2 Server

As mentioned the server uses multi-threading to support multiple TCP connections at a time alongside a UDP connection. This is done by creating two threads when the server starts - one for the UDP listener and one for the TCP listener. Each of these then listens for a connection and when one is made it handles the client appropriately. The UDP server communicates with the client in a connectionless manner and as a result can conduct all of the communication on the port used for listening. The TCP server, on the other hand, uses connections for communication and as a result must create a new port to communicate with the client on. In order to prevent this from locking up the system for long or persistent communication sessions, every time a new client makes a connection, a new thread is created with the purpose of handling the communication on a new port. `goroutines` are very lightweight which means that they scale well with volume and will allow the server to support thousands of simultaneous connections [6]. Note that `goroutines` are executed concurrently which is not necessarily in parallel. What this means is that although the system will not lock out connections while one is being handled, the system will slow down with too many connections. `goroutines` also have the advantage that they are trivial to use when code makes proper use of function calls, the function is simply called with the `go` keyword and the Go compiler takes care of the rest of the underlying complexity. Although not used in this project, Go supports

communication across threads using `channels`.

The server accepts the following requests: GET, HEAD, POST, PUT and DELTE. These were considered to be the most important for demonstrating the HTTP protocol. The GET, HEAD and POST methods were the original three methods in HTTP/1.0 [3], while the PUT and DELTE methods provide the missing functionality for the CRUD operations, which are very important for many applications which make use of RESTful APIs [5]. Other methods that would be useful are those specified in *Table 1.* The PATCH method in particular would be an important addition as it allows much more flexible Update operations by facilitating partial updates of files. This means that when a client wishes to overwrite part of a file they do not need to send the whole file to the server, rather they send only the part they wish to update which saves bandwidth. The OPTIONS method is very useful for allowing clients to query which methods the server accepts. This would reduce the amount of `400 Bad Request` responses as a result of requests not being understood. The OPTIONS and PATCH methods were not added due to time constraints. The POST and GET methods implemented by the server cannot be used to fill in forms using the `application/x-www-form-urlencoded` content type. This was not implemented due to time constraints.

The server is capable of using the following responses: `505 HTTP Version Not Supported`, `404 Not Found`, `400 Bad Request`, `304 Not Modified`, `301 Moved Permanently`, and `200 OK`. These methods cover the majority of common situations. However, some key methods are missing. The `302 Found` method is useful in real world systems where resources are temporarily moved. Additionally, a wider range of 2xx responses are also useful as they give the client more information about the action that has been taken by the server in addition to the fact that their request was successful. Once again, these were not added due to time constraints. The `403 Forbidden` response would be useful to differentiate between invalid and forbidden requests from the client.

The server implements a very limited subset of headers. These are: `Content-Length`, `Last-Modified`, `Location`, `Content-Language`, `Date` and `Server`. The `Content-Length` header is for HTTP/1.0 backwards compatibility. The `Last-Modified` header, along side the `304 Not Modified` response, is for use with proxy servers. The `Location` header is used in conjunction with the `301 Moved Permanently` response for redirections. The server can also accept a range of headers. Of particular interest here is the `If-Modified-Since` header which is sent by a proxy to check if it has the most up-to-date version of a file. An important header that needs to be added is `Content-Type`, which gives the client important information about the body of the message.

Another feature that the server does not support is the use of chunked encoding which allows for data to be streamed. This is particularly useful for dynamically generated content where the length of the content is not known before hand and the `Content-Length` header cannot be used. Instead the `Transfer-Encoding` header is used with the value `chunked`. This lets the client know that they should continue receiving data until the End-Of-File marker is received. This was not implemented because the server does not have any dynamically generated pages and can always use `Content-Length` as would have been done in HTTP/1.0.

To facilitate the `301 Moved Permanently` response, the server maintains a text file that keeps track of moves. When the server receives a request for a URL that does not exist it loads the file into a `map` data structure and checks if the URL can be found in the `map`. If it is found, the server returns a `301 Moved Permanently` response with the location retrieved from the `map`, otherwise the server returns a `404 Not Found` response.

HTTPS should also be implemented on the server for security reasons. However, HTTPS is fairly complicated as it requires the use of SSL, this is the reason it was not implemented.

4.3 Proxy

The proxy, much like the server, is multi-threaded using `goroutines`. This allows the proxy to serve multiple clients at a time. The proxy works with both persistent and non-persistent TCP, however, it is not implemented with UDP.

In its most basic sense the proxy's behavior can be described as follows:

1. The proxy listens for TCP connections
2. The proxy receives a connection from the client
3. The proxy creates a new thread to handle this new connection
4. The proxy receives a request from the client
5. The proxy creates a connection to the destination server
6. The proxy forwards the request to the destination server
7. The proxy receives a response from the destination server
8. The proxy forwards the response to the client
9. The communication continues as is appropriate, closing connections when finished

In addition to the above, if the request is a GET, the proxy also acts as a web cache. The proxy maintains a list of files it has cached and the last modified date for the file. This is stored in a text file and loaded into a `map`, of file to last modified date, whenever it is searched. Whenever the proxy receives a request from the client, in step 4, it checks its `map` to see if it has the file cached. If it does not, then the procedure above continues, with and added step 7.5, in which the proxy caches the file and saves its last modified time. However, if it does, it sends the server a conditional get message containing the `If-Modified-Since` header with the value loaded from the `map`. The server will now send back either one of, a `304 Not Modified` response with a blank body to indicate that the cached file is up to date, or a response containing the most up to date file and its last

modified date. This date will be formatted as described by RFC1123. If the server sends back the newly modified file, the proxy caches it before forwarding it to the client in step 8. Otherwise the proxy simply composes a new response containing the cached file and sends that to the client as though it was from the server instead of completing step 8.

The proxy employs the same checks as the client for making sure that it receives all the data using the `Content-Length` and `Transfer-Encoding` headers.

5 DIVISION OF WORK

5.1 Overall Project Management

The overall project was managed through the use of time management application called Trello. Trello allows job cards to be created and assigned to a member of the group. It has multiple features such as allocating cards as bugs, providing deadlines for cards and allowing for files to be uploaded and shared. All these aspects of the application were used continuously throughout the project. A screen shot of the board that was used for the project can be seen in *Figure* 18 of Appendix C.

The code aspect of the project was managed using git and GitHub. The integration capabilities offered by git meant that the team did not necessarily need to work together all the time, which increased the rate at which work was developed and ultimately lead to the deadline being met.

5.2 Division of Code

The division of work was such that each member of the group contributed equally to the outcome of the entire project. This was achieved by designating specific subsystems to each member of the group. Devin Taylor was responsible for handling the client code and all associated common functionality, while James Allignham was responsible for handling all the sever code and associated common functionality. From these two bases both engineers contributed equally to the proxy as the majority of the proxy was composed of the subsystems from the client and server. *Table* 4 in Appendix B.5 details the specific aspects of the code that each engineer wrote/contributed to. A screen shot of the GitHub contributions to the project can be seen in *Figure* 19 of Appendix B.5, this figure emphasises the equal contributions of each engineer.

5.3 Division of Report Writing

The report writing was divided in the same manor as the code, each engineer was responsible for writing about the sections of code that they implemented. The theoretical aspect of the report was divided equally by the aspects that each engineer felt most comfortable contributing to. The introduction, HTTP description, system overview were all done by James Allignham while the results, critical analysis, conclusion and division of work were all done by Devin Taylor.

6 RESULTS

In order to test the integrity of the system it was necessary to observer the behaviour of the system between two computers over the same network. The primary criteria for testing the system was to ensure the following:

- The client request message was received by the server and interpreted correctly.
- The server response message was received by the client, in conjunction with the necessary data, and interpreted correctly.
- The client was able to respond to the contents of the servers message by either following a link to the actual site or requesting further information in the form of sources.
- The server was capable of returning conditional error codes if necessary.
- The client was able to interact with the proxy.
- The proxy was able to interact with the client.
- The proxy was able to interact with the cache and communicate the corresponding information with the client and server respectively.

6.1 Basic Client Server HTTP Requests and Responses

As mentioned in *Table* 1, there are a set of commonly used HTTP requests. For the basis of the project only GET, HEAD, POST, PUT and DELETE were required to be implemented. The use of Wireshark allowed for the message trace between the client and server to be followed. It also provides detailed insight into the communications by facilitating the analysis of the messages being sent and received. For all the examples provided below assume that the client IP address is 10.0.0.148 and the server IP address is 10.0.0.243. These IP addresses will provide insight into the trace observed between client and server when they communicate.

The GET and HEAD Methods: With reference to *Figure* 3 of Appendix A.1 the client sends a GET request to the server, to which the server responds with an OK message containing the contents of the file that was requested. The OK message symbolises that the server was able to access the requested file. The second GET message sent for `/test.jpg` is due to the fact that the returned file contained an image and as a result the client is able to detect the presence of sources and request them from the applicable server. In this example the image was hosted on the same server.

An example of the GET message sent can be seen in *Figure* 4 of Appendix A.1. The aspects of interest are the inclusion of the required URI in headers as well as the method (GET) in the request line. This is contrary to what is observed in *Figure* 5 of Appendix A.1 whereby there is no mention of a URI but there is the inclusion of the `Data` section, symbolising the entity body, which contains the requested content. These two figures highlight the fundamental differences between the HTTP messages sent by the

client and the HTTP messages sent by the server.

Fundamentally the communication between the client and the server for the HEAD request is the same as the GET request. The only difference observed is that the response message from the server contains no entity body (`Data` section) but rather only the header information.

The POST and PUT Methods: The post method differs from the above mentioned methods in the sense that the client includes an entity body in the request message sent to the server. As mentioned in *Table* 1 the POST method is commonly used for interacting with forms, hence the body that's included in the client request message contains the information that needs to be submitted to the form. An example of the trace between the client and the server takes on the same form as mentioned above, this can be seen in *Figure* 6 of Appendix A.1.

The PUT method has the same face value structure as the POST method with regards to the trace observed between the client and server. The two methods only differ in how the client handles the requested URI as well as the entity body included in the client's request message. These differences are detailed in *Table* 1.

The DELETE Method: The DELETE method also has the same trace structure as the above mentioned methods, see *Figure* 7 of Appendix A.1, whereby the client makes a request and the server responds to the request. The difference with the DELETE method is that the URI included in the request message links to the file on the server that the client wishes to remove. The server subsequently responds to the client whether or not it was able to remove the file.

6.2 Client Server HTTP Requests and Responses Via Proxy

The communication observed between the client and server, via the proxy, follows the same basic structure in the sense that a request message is sent by the client and a response message is received back. The communication differs in the sense that the proxy acts as an intermediary relaying the message from the client to the server and the server to the client. The details pertaining to how this is accomplished are detailed in Section 2.3. A test was conducted on three different computers on the same network in order to recreate the trace sequence observed with a proxy being used. The client was represented on `192.168.1.103`, the proxy on `192.168.1.104` and the server of `192.168.1.105`.

The trace sequence observed for a GET message can be seen if *Figure* 9 of Appendix A.1. An initial GET request was sent by the client to the proxy for the `/index.html` file. The message sent to the proxy can be seen in *Figure* 10 of Appendix A.1, the important aspect of this message to note is the inclusion of the servers IP address under the host heading. Once the proxy receives the request it checks if the requested file was saved in its cache, which it was not, and forwarded the request to server. The server then retrieves the requested file, which is passed by to the client via the proxy. The response from the server contains an `Last-Modified` header line which is fundamental for the caching of the proxy, this can be seen in *Figure* 11 of Appendix A.1.

The above experiment was repeated again expect this time the proxy has added the `/index.html` file into its cache. The trace sequence can be observed in *Figure* 12 of Appendix A.1. It can be noted that the sequence is the same as described above except the response code from the server is a `304 Not Modified`. This occurs because the proxy registered that the requested `/index.html` file was in its cache and forwarded an `If-Modified-Since` header line in its request to the server, this date corresponds to the date returned from the server in the previous example. A comparison between the proxy GET request before and after caching can be seen in *Figure* 13 of Appendix A.1. Due to the `304` response from the server the proxy then retrieves the requested file from its cache and forwards a copy of this to the client.

6.3 Server Error Codes

Additional tests were also conducted between the client and server to ensure that all the server error codes were implemented correctly. The specific errors that were implemented were provoked as follows:

Table 3: Different methods for invoking server response errors

Status Code	**Provocation**
301	Request a file that had been moved to a new location on the local server
400	Make a request that did not fall within the range of accepted/implemented requests
404	Request a file that did not exist on the server
505	Alter the HTTP version used in the request message

The corresponding server responses can be seen in the trace sequences in *Figure* 14 of Appendix A.2. It can be noted that the `301` response message invoked a new request to be generated by the client to the location that the file had been moved to. The client is able to make the new request as a result of the `Location` header line returned by the server in the header field, this can be seen in *Figure* 15 of Appendix A.2.

6.4 Round Trip Time Calculations

A timer class was created in order to determine the Round Trip Times (RTT) observed for different settings on the system. The different systems settings are detailed in Section 4.1.

The results obtained from these calculations were written

to a text file and plotted using a graphing library, the results can be seen in *Figure* 16 of Appendix A.2. The tests were conducted on retrieving the same `/index.html` for each situation. The results obtained are as expected, whereby the fastest time was TCP with a persistent connection and no proxy while the slowest time was a TCP non-persistent connection going through the proxy.

The same experiment was conducted to determine the effectiveness of the cache on the rate at which the information is retrieved by the client. The results can be seen in *Figure* 17 of Appendix A.2, it can be noted that the once the `/index.html` file had been cached the client was able to retrieve the information faster.

7 CRITICAL ANALYSIS

The system is a fully functional, stable and well constructed solution. The presented system implements all functionality stipulated in project brief, that being the following: web server, web client, proxy server, multithreading, persistent and non-persistent connection types, TCP and UDP, the use of Wireshark to obtained results and the use of a timer to obtain RTT values for different connection types.

Despite the solution being fully functional there are three weaknesses. Firstly, the primary weakness of the system is that once the client establishes a connection with the proxy, all future computation (i.e. proxy communication with the server) until the response reaches the client takes place in a thread established by the proxy. Despite Go multithreading being able to handle thousands of threads at the same time with very little computational overhead, this is not the ideal implementation if the infrastructure utilising the proxy were to be on the level of a large scale business. Secondly, another design constraint was accepted when it was decided to only implemented TCP communication when the client was communicating with the server via the proxy. This was primarily due to the above mentioned problem.

Thirdly, another minor weakness, which was primarily due to time constraints, was the inability to interact with web forms. This is due to the fact that JavaScript is often implemented in order to auto-generate the correctly structured POST message for the form. As a result of this a well rounded implementation of this could not be presented and a design decision was taken to omit this aspect. Despite this, the POST method still works when generating a basic example between the client and server.

There are multiple aspect of the system that are subject to improvements. Initially, creating a smart way of mapping client and sever IP addresses on the proxy will allow for the above mentioned thread problem to be fixed. This would be the primary priority of the project. Also, allowing the proxy to implement UDP communication would better the functionality range of the system. The other aspect of improvement would be increasing the overall scope of the project. Expansions to the system could be in the form of adding support for https and not only http and increasing the range of requests and responses to all those mentioned in *Tables* 1 and 2.

8 DESCRIPTION OF CODE

8.1 Overall Code Structure

The project was structured such that all code was localised to a common root directory called `src/`. By localising the code in a source root directory it allowed for the client, server and proxy associated code to be located in this folder under their own subdirectories (i.e. `src/client/`). Due to this specific project layout, a feature of Go is that all common functionality between the client, server and proxy code can be abstracted to a further subdirectory `src/lib/`, thus creating a library of common functionality for the project. This library can be sourced in any project script allowing that script access to all common functionality.

The majority of the code was centralised around the functionality of structs. The main reason for doing this was to provide each struct with its own methods, a feature of the Go programming language. An example of this implementation can be seen in the listings excerpt from the `request_message.go` file (Listings 1).

Listing 1: Struct method setup

```
1  type RequestMessage struct {
2      Method string
3      Url string
4      Version string
5      HeaderLines map[string]string
6      EntityBody string
7  }
8
9  func (rm *RequestMessage) SetRequestLine(Method ↩
       string, Url string, Version string) {
10     rm.Method = Method
11     rm.Url = Url
12     rm.Version = Version
13 }
```

The function seen in Listings 1 can be invoked on a `RequestMessage` object, `rm`, by calling `rm.RequestMessage (arg1, arg2, arg3)` on the object. This proved beneficial when dealing with multiple objects of the same type as the information was localised to each struct object.

8.2 Configuration Settings

As there are multiple options available for the types of protocols, connections and proxy, it was necessary to provide the user with the ability to change these settings in a 'user friendly' manner. The configurations were set-up in a struct as this allowed for easy manipulation of the values as well as being able to implement read and write methods on a configuration file easily. The struct was defined as seen is Listing 2 of Appendix B.1 while the code used to edit the configuration settings can be seen in Listing 3 of Appendix B.1.

8.3 Client Code

The client code was mainly centralised around establishing a connection with either the web server or the proxy server. Once this connection was established, it needed to ensure that all information was read from the server before analysing the received information. Upon analysis of this information, the client could be required to establish further connections to retrieve images and other sources before saving the file to a temporary directory to be launched.

Establishing The Connection: The type of connection that is established by the client is dependant on what protocol configuration setting was specified by the client. Despite this the `net` package in Go is able to accept either TCP or UDP when creating a network connection [7]. The connection was established as can be seen in Listing 4 of Appendix B.2.

Retrieving Information: In order to determine whether or not all the information had been obtained from the server, it was required that initially the client would read a large packet from the connection. In this packet, the client code would search for either the presence of either the header line `Transfer-Encoding:Chunked` or `Content-Length:<bytes>`. The presence of either of these two header lines meant that the server had divided the sent file into multiple packets. The fact that all the headers are parsed into a map of header name to details it makes the look up fast and effective.

In order to read multiple packets it was necessary to continuously loop over the read function until all the information had been obtained. Determining when to exit this loop was aided by the above mentioned header lines. Firstly, the `Transfer-Encoding` is a header line specific to HTTP/1.1 which ensures that an End-Of-File character, `\r\n0\r\n\r\n`, will appear in the last packet. Therefore, searching each packet for this character provided the required knowledge. Secondly, the `Content-Length` provided knowledge of the size of the attached body. Therefore, utilising this and being able to manually calculate the size of the header lines meant that the packets could be read from the connection and continually subtracted from the running tally of remaining bytes until such time that the running tally became negative. The implementation of these two concepts can be seen in Listing 5 of Appendix B.2.

Analysing Received Information: The next important aspect of the client was determining whether or not the received file contained any references to other files that were imperative to the received file forming correctly. It was assumed that all references to other files would begin with 'src=http://' in the base HTML file. All other file references were ignored as they would be beyond the scope of the project.

String manipulation was used to determine the presence of these sources. If there were any sources present the `retrieveSources()` function, see Listing 6 of Appendix B.2, would deconstruct the referenced strings through the use of regular expressions and the occurrence of special characters. From the deconstruction the function constructs a map of host name to file extension on the hosts server. The map would then be used to iterate over all elements in the map and retrieve them as described in the above sections.

8.4 Server Code

As in the case of the client code the server code was primarily revolved around certain concepts. These concepts are: confirming and establishing the connection with the client, processing the clients request, generating the appropriate response to the client's request and sending the generated information to the client. The server has additional functionality in two aspects, namely: when interfacing with the proxy it has to determine if a cached file has been recently updated and when interfacing with the client it has to prove the new location of a page if it has been moved.

Listening For and Establishing a Connection: Unlike the case for the client, the server side calls for the implementation of TCP and UDP differ. Thus, in order to account for this, two separate functions were implemented to establish the individual connections. The code used to implement these connection can be seen in Listing 7 of Appendix B.3. Two aspects of Listing 7 need to be noted. Firstly, the listening function calls differ (lines 20 and 40) and secondly in lines 5 and 6 there is the implementation of the `go <function>` command which is the Go programming language implementation of establishing a new thread. Therefore, each time a new connection request is received by the server, the request is handled in a new thread.

Processing The Client Request: Processing the client request requires decomposing the client request message to obtain the HTTP method, the required URL, the HTTP version being used by the client, the header field lines and the body (if one is attached). Depending on this information the server is able to perform certain conditional checks in order to generate the correct response message.

Taking Action and Generating Response Message: The HTTP method will determine the course of action that the server follows, these are detailed in *Table* 1. The method of interest is the GET method. The GET method firstly requires the server to check if the file exists and secondly, if it does exist, to retrieve a copy of the file and parse it into the response message to be sent to the client. If the file does not exist or there was another problem with the client request message, the server would generate the appropriate error message detailed in *Table* 2.

Server Proxy Interfacing: The proxy interaction will make specific reference to the GET request as this encompasses

all functionality. The implementation of the GET response code can be seen in Listing 8 of Appendix B.3.

If the proxy contains a copy of the requested file it appends a `Last-Modified` header line to the request message, this is what the server looks for. If there is a date mentioned the server then compares if the file has been modified since that date (the last modified date can be obtained from the function call in line 7 of the listing), this is achieved through the use of the `time.After()` function in the `time` Go package (see line 11 of the listing). If the file has been modified then the server returns the latest copy of the file and if it hasn't it returns the `304 Not Modified` response.

Moved Files: In order to determine whether or not a file has been moved the server keeps a table of file previous location and current location for all moved files, this is stored as a text file on the server. When a request for a file is made, if the location appears in this file the server composes a `301` response message and appends a `Location` header with the new location to the header field.

8.5 Proxy

The proxy sending and receiving of information is identical in nature to the corresponding implementations in the client and server, therefore the only aspect of the proxy that will be analysed is caching. Two characteristics of proxy caching is the storing of files and the conditional gets.

File Storing: File storing is only implemented for GET requests. When the response message is received from the server the proxy creates a directory in the cache corresponding to the full path of the file received from the server. A design decision was taken to 'mimic' the server in the proxy as opposed to mapping file names to a different location in the cache. The code used to create a directory with full read write permission can be seen in line 3 of Listing 9 of Appendix B.4. Once the directory is created the code then writes the corresponding file to the location, this can be seen in line 6 of the listing.

Conditional Get: The conditional GET revolves primarily around the aforementioned `304` response code from the server. When the proxy receives the clients request message it checks if the file occurs in the proxy cache, see line 1 of Listing 10. This is achieved by using the look-up table stored on the proxy of URL to last-modified times. If the file does occur in cache the client request message is modified by including the last-modified date in the look-up table. This is simple to do as the headers are stored as a map already. The proxy then obtains the response message from the server which either specifies that the file was modified or it was not.

If the file is not modified then the proxy retrieves the file from cache and forwards a copy of it to the client. If the file has been modified then new file is written to cache as mentioned above and the cache map is modified to contain the new date, this implementation can be seen in lines 12-16 of Listing 10.

8.6 All Other Common Functionality

The above mentioned sections were described in detail as they are fundamental to the functionality of the system. The other common functions implemented in the project are described in brief in *Table* 4 of Appendix B.5.

8.7 How To Run Project

The details as to how to run the project can be seen in the README file in Appendix B.6.

9 CONCLUSION

The use of the Go language allowed for a simple and elegant solution with its `goroutines`, which allowed for seamless concurrency, powerful high level features such as `maps` and garbage collection, which promoted clean and concise code, and fast run time. A subset of HTTP large enough to perform basic requests and responses in HTTP/1.1 was implemented. However, there is a large room for improvements in this aspect as well as in the limited features of the proxy server. The web client was able to access pages hosted on both the Internet and the server. Testing with Wireshark revealed that all of the systems worked as expected. In addition, the various configurations of connection types, protocols and proxy usage behaved as expect.

REFERENCES

[1] Kurose J F, Ross K W. *Computer networking: a top-down approach.* Boston, Pearson. 2013. pp 83 - 115.

[2] Fielding R, Reschke J. 'Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.' IETF, RFC June 2014. [online] Available: `https://tools.ietf.org/html/rfc7230` Last Accessed: 3 April 2016

[3] Berners-Lee T, Fielding R, Frystyk H. 'Hypertext Transfer Protocol – HTTP/1.0' IETF, RFC May 1996. [online] Available: `https://tools.ietf.org/html/rfc1945` Last Accessed: 3 April 2016

[4] Gerrand A. *The Go Blog - Go maps in action.* The Go Programming Language. 6 February 2013 . [online] Available: `https://blog.golang.org/go-maps-in-action` Last Accessed: 3 April 2016

[5] Guinard D, Trifa V (2009). *Towards the Web of Things: Web Mashups for Embedded Devices.* Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences). Madrid, Spain. April 2009.

[6] Doxsey C. *An Introduction to Programming in Go.* 2012. ch 10. pp 108 - 111.

[7] The Go Programming Language. *Package net.* The Go Programming Language. Available: `https://golang.org/pkg/net/#Dial` Last Accessed: 3 April 2016

Appendix

A Results supporting information

This appendix details the results obtained when running the application one two separate computers on the same network. The results include excerpts of the results obtained from Wireshark as well as the Round Trip Times (RTT) obtained for different implementations of the project.

A.1 Wireshark results

No.	Time	Source	Destination	Protocol	Length	Info
66	24.551451000	10.0.0.148	10.0.0.243	HTTP	168	GET /index.html HTTP/1.1
68	24.551727000	10.0.0.243	10.0.0.148	HTTP	394	HTTP/1.1 200 OK
72	24.567964000	10.0.0.148	10.0.0.243	HTTP	166	GET /test.jpg HTTP/1.1
94	24.580730000	10.0.0.243	10.0.0.148	HTTP	1013	HTTP/1.1 200 OK (JPEG JFIF image)

Figure 3: Computer interaction for GET request

```
▶Frame 66: 168 bytes on wire (1344 bits), 168 bytes captured (1344 bits) on interface 0
▶Ethernet II, Src: IntelCor_83:0f:90 (60:36:dd:83:0f:90), Dst: LiteonTe_6b:e7:a9 (9c:b7:0d:6b:e7:a9)
▶Internet Protocol Version 4, Src: 10.0.0.148 (10.0.0.148), Dst: 10.0.0.243 (10.0.0.243)
▶Transmission Control Protocol, Src Port: 59704 (59704), Dst Port: mosaicsyssvc1 (1235), Seq: 1, Ack: 1, Len: 114
▼Hypertext Transfer Protocol
 ▼GET /index.html HTTP/1.1\r\n
  ▼[Expert Info (Chat/Sequence): GET /index.html HTTP/1.1\r\n]
     [Message: GET /index.html HTTP/1.1\r\n]
     [Severity level: Chat]
     [Group: Sequence]
    Request Method: GET
    Request URI: /index.html
    Request Version: HTTP/1.1
  User-agent: Mozilla/5.0\r\n
  language: en\r\n
  Host: 10.0.0.243:1235\r\n
  Connection: keep-alive\r\n
  \r\n
  [Full request URI: http://10.0.0.243:1235/index.html]
  [HTTP request 1/1]
  [Response in frame: 68]
```

Figure 4: GET request message

```
▶Frame 68: 394 bytes on wire (3152 bits), 394 bytes captured (3152 bits) on interface 0
▶Ethernet II, Src: LiteonTe_6b:e7:a9 (9c:b7:0d:6b:e7:a9), Dst: IntelCor_83:0f:90 (60:36:dd:83:0f:90)
▶Internet Protocol Version 4, Src: 10.0.0.243 (10.0.0.243), Dst: 10.0.0.148 (10.0.0.148)
▶Transmission Control Protocol, Src Port: mosaicsyssvc1 (1235), Dst Port: 59704 (59704), Seq: 1, Ack: 115, Len: 340
▼Hypertext Transfer Protocol
 ▼HTTP/1.1 200 OK\r\n
  ▼[Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
     [Message: HTTP/1.1 200 OK\r\n]
     [Severity level: Chat]
     [Group: Sequence]
    Request Version: HTTP/1.1
    Status Code: 200
    Response Phrase: OK
  Server: FooBar\r\n
  Date: Sat, 02 Apr 2016 19:26:47 +0200\r\n
  Content-Language: en\r\n
▶Content-Length: 175\r\n
  Last-Modified: Sat, 02 Apr 2016 19:00:20 +0200\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.000276000 seconds]
  [Request in frame: 66]
▶Data (175 bytes)
```

Figure 5: GET response message

No.	Time	Source	Destination	Protocol	Length	Info
27	10.376887000	10.0.0.148	10.0.0.243	HTTP	172	POST /test.html HTTP/1.1
29	10.377240000	10.0.0.243	10.0.0.148	HTTP	240	HTTP/1.1 200 OK

Figure 6: POST trace between client and server

No.	Time	Source	Destination	Protocol	Length	Info
49	22.219627000	10.0.0.148	10.0.0.243	HTTP	170	DELETE /temp.html HTTP/1.1
51	22.220017000	10.0.0.243	10.0.0.148	HTTP	221	HTTP/1.1 200 OK

Figure 7: DELETE trace between client and server

```
▶Frame 49: 170 bytes on wire (1360 bits), 170 bytes captured (1360 bits) on interface 0
▶Ethernet II, Src: IntelCor_83:0f:90 (60:36:dd:83:0f:90), Dst: LiteonTe_6b:e7:a9 (9c:b7:0d:6b:e7:a9)
▶Internet Protocol Version 4, Src: 10.0.0.148 (10.0.0.148), Dst: 10.0.0.243 (10.0.0.243)
▶Transmission Control Protocol, Src Port: 59821 (59821), Dst Port: mosaicsyssvc1 (1235), Seq: 1, Ack: 1, Len: 116
▼Hypertext Transfer Protocol
 ▼DELETE /temp.html HTTP/1.1\r\n
  ▼[Expert Info (Chat/Sequence): DELETE /temp.html HTTP/1.1\r\n]
     [Message: DELETE /temp.html HTTP/1.1\r\n]
     [Severity level: Chat]
     [Group: Sequence]
    Request Method: DELETE
    Request URI: /temp.html
    Request Version: HTTP/1.1
  Host: 10.0.0.243:1235\r\n
  Connection: keep-alive\r\n
  User-agent: Mozilla/5.0\r\n
  language: en\r\n
  \r\n
  [Full request URI: http://10.0.0.243:1235/temp.html]
  [HTTP request 1/1]
  [Response in frame: 51]
```

Figure 8: DELETE request message

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000136000	192.168.1.103	192.168.1.104	HTTP	183	GET /index.html HTTP/1.1
9	0.000420000	192.168.1.104	192.168.1.105	HTTP	1090	GET /index.html HTTP/1.1
11	0.000858000	192.168.1.105	192.168.1.104	HTTP	409	HTTP/1.1 200 OK
16	0.001264000	192.168.1.104	192.168.1.103	HTTP	409	HTTP/1.1 200 OK

Figure 9: GET request for proxy with no cache saved

```
▶Frame 4: 183 bytes on wire (1464 bits), 183 bytes captured (1464 bits) on interface 0
▶Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▶Internet Protocol Version 4, Src: 192.168.1.103 (192.168.1.103), Dst: 192.168.1.104 (192.168.1.104)
▶Transmission Control Protocol, Src Port: 34786 (34786), Dst Port: bvcontrol (1236), Seq: 1, Ack: 1, Len: 117
▼Hypertext Transfer Protocol
 ▼GET /index.html HTTP/1.1\r\n
  ▶[Expert Info (Chat/Sequence): GET /index.html HTTP/1.1\r\n]
    Request Method: GET
    Request URI: /index.html
    Request Version: HTTP/1.1
  Connection: keep-alive\r\n
  User-agent: Mozilla/5.0\r\n
  language: en\r\n
  Host: 192.168.1.105:1235\r\n
  \r\n
  [Full request URI: http://192.168.1.105:1235/index.html]
  [HTTP request 1/1]
  [Response in frame: 16]
```

Figure 10: GET request message for client to proxy with no cache saved

```
▶Frame 11: 409 bytes on wire (3272 bits), 409 bytes captured (3272 bits) on interface 0
▶Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▶Internet Protocol Version 4, Src: 192.168.1.105 (192.168.1.105), Dst: 192.168.1.103 (192.168.1.103)
▶Transmission Control Protocol, Src Port: mosaicsyssvc1 (1235), Dst Port: 49515 (49515), Seq: 1, Ack: 1025, Len: 343
▼Hypertext Transfer Protocol
 ▼HTTP/1.1 200 OK\r\n
  ▶[Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
    Request Version: HTTP/1.1
    Status Code: 200
    Response Phrase: OK
  Content-Language: en\r\n
 ▶Content-Length: 178\r\n
  Last-Modified: Sun, 03 Apr 2016 12:06:13 +0200\r\n
  Server: FooBar\r\n
  Date: Sun, 03 Apr 2016 13:46:14 +0200\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.000438000 seconds]
  [Request in frame: 9]
▶Data (178 bytes)
```

Figure 11: GET response message from server to proxy

No.	Time	Source	Destination	Protocol	Length	Info
501	399.159545000	192.168.1.103	192.168.1.104	HTTP	183	GET /index.html HTTP/1.1
506	399.160119000	192.168.1.104	192.168.1.105	HTTP	1142	GET /index.html HTTP/1.1
508	399.160412000	192.168.1.105	192.168.1.104	HTTP	191	HTTP/1.1 304 Not Modified
513	399.160557000	192.168.1.104	192.168.1.103	HTTP	359	HTTP/1.1 200 OK Continuation or non-HTTP traffic

Figure 12: GET request message for client to proxy with saved cache

```
▶Frame 9: 1090 bytes on wire (8720 bits), 1090 bytes captured (8720 bits) on interface 0
▶Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▶Internet Protocol Version 4, Src: 192.168.1.103 (192.168.1.103), Dst: 192.168.1.105 (192.168.1.105)
▶Transmission Control Protocol, Src Port: 49515 (49515), Dst Port: mosaicsyssvc1 (1235), Seq: 1, Ack: 1, Len: 1024
▼Hypertext Transfer Protocol
 ▼GET /index.html HTTP/1.1\r\n
  ▶[Expert Info (Chat/Sequence): GET /index.html HTTP/1.1\r\n]
    Request Method: GET
    Request URI: /index.html
    Request Version: HTTP/1.1
  Connection: keep-alive\r\n
  User-agent: Mozilla/5.0\r\n
  language: en\r\n
  Host: 192.168.1.105:1235\r\n
  \r\n
  [Full request URI: http://192.168.1.105:1235/index.html]
  [HTTP request 1/1]
  [Response in frame: 11]
```

(a) Proxy GET before caching

```
▶Frame 506: 1142 bytes on wire (9136 bits), 1142 bytes captured (9136 bits) on interface 0
▶Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▶Internet Protocol Version 4, Src: 192.168.1.103 (192.168.1.103), Dst: 192.168.1.105 (192.168.1.105)
▶Transmission Control Protocol, Src Port: 49560 (49560), Dst Port: mosaicsyssvc1 (1235), Seq: 1, Ack: 1, Len: 1076
▼Hypertext Transfer Protocol
 ▼GET /index.html HTTP/1.1\r\n
  ▶[Expert Info (Chat/Sequence): GET /index.html HTTP/1.1\r\n]
    Request Method: GET
    Request URI: /index.html
    Request Version: HTTP/1.1
  Host: 192.168.1.105:1235\r\n
  Connection: keep-alive\r\n
  User-agent: Mozilla/5.0\r\n
  language: en\r\n
  If-Modified-Since: Sun, 03 Apr 2016 12:06:13 +0200\r\n
  \r\n
  [Full request URI: http://192.168.1.105:1235/index.html]
  [HTTP request 1/1]
  [Response in frame: 508]
```

(b) Proxy GET after caching

Figure 13: Difference in proxy request messages depending on status of cache

No.	Time	Source	Destination	Protocol	Length	Info
12	6.820318000	10.0.0.148	10.0.0.243	HTTP	173	GET /test/index.html HTTP/1.1
14	6.820542000	10.0.0.243	10.0.0.148	HTTP	452	HTTP/1.1 301 Moved Permanently
18	6.833755000	10.0.0.148	10.0.0.243	HTTP	168	GET /index.html HTTP/1.1
20	6.833980000	10.0.0.243	10.0.0.148	HTTP	394	HTTP/1.1 200 OK
24	6.852919000	10.0.0.148	10.0.0.243	HTTP	166	GET /test.jpg HTTP/1.1
58	6.944204000	10.0.0.243	10.0.0.148	HTTP	1013	HTTP/1.1 200 OK (JPEG JFIF image)

(a) 301

No.	Time	Source	Destination	Protocol	Length	Info
77	28.461217000	10.0.0.148	10.0.0.243	HTTP	172	OPTIONS /index.html HTTP/1.1
79	28.461515000	10.0.0.243	10.0.0.148	HTTP	485	HTTP/1.1 400 Bad Request

(b) 400

No.	Time	Source	Destination	Protocol	Length	Info
47	15.456507000	10.0.0.148	10.0.0.243	HTTP	175	GET /filewontfind.html HTTP/1.1
49	15.456701000	10.0.0.243	10.0.0.148	HTTP	395	HTTP/1.1 404 Not Found

(c) 404

No.	Time	Source	Destination	Protocol	Length	Info
1368	165.398234000	10.0.0.148	10.0.0.243	HTTP	168	GET /index.html HTTP/1.0
1369	165.398251000	10.0.0.243	10.0.0.148	TCP	54	mosaicsyssvc1 > 60239 [ACK] Seq=1 Ack=115 Win=29312 Len=0
1370	165.398483000	10.0.0.243	10.0.0.148	HTTP	441	HTTP/1.1 505 HTTP Version Not Supported

(d) 505

Figure 14: Different server error response codes

```
▶Frame 14: 452 bytes on wire (3616 bits), 452 bytes captured (3616 bits) on interface 0
▶Ethernet II, Src: LiteonTe_6b:e7:a9 (9c:b7:0d:6b:e7:a9), Dst: IntelCor_83:0f:90 (60:36:dd:83:0f:90)
▶Internet Protocol Version 4, Src: 10.0.0.243 (10.0.0.243), Dst: 10.0.0.148 (10.0.0.148)
▶Transmission Control Protocol, Src Port: mosaicsyssvc1 (1235), Dst Port: 59932 (59932), Seq: 1, Ack: 120, Len: 398
▼Hypertext Transfer Protocol
 ▼HTTP/1.1 301 Moved Permanently\r\n
  ▼[Expert Info (Chat/Sequence): HTTP/1.1 301 Moved Permanently\r\n]
    [Message: HTTP/1.1 301 Moved Permanently\r\n]
    [Severity level: Chat]
    [Group: Sequence]
   Request Version: HTTP/1.1
   Status Code: 301
   Response Phrase: Moved Permanently
  Server: FooBar\r\n
  Date: Sat, 02 Apr 2016 20:29:46 +0200\r\n
  Content-Language: en\r\n
  Location: http://10.0.0.243/index.html\r\n
 ▶Content-Length: 226\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.000224000 seconds]
  [Request in frame: 12]
 ▶Data (226 bytes)
```

Figure 15: 301 Server response message with location header field

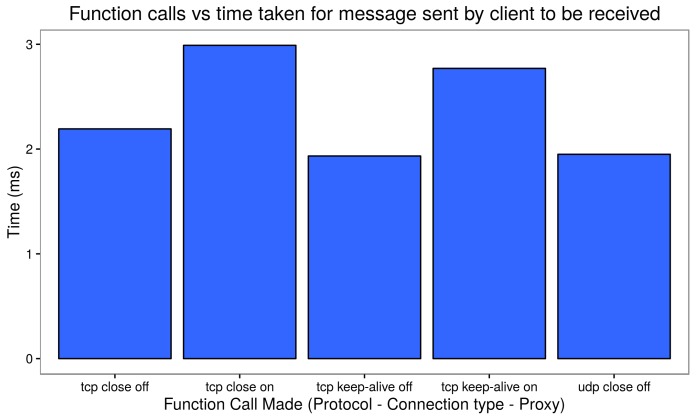

Figure 16: RTT times for different connection type settings

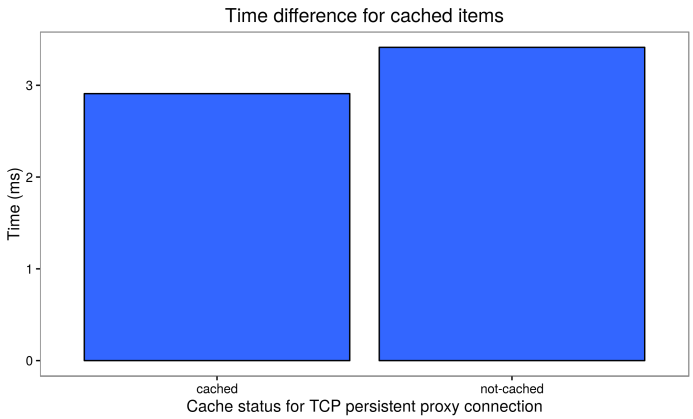

Figure 17: RTT times for different cache statuses on a TCP persistent connection

B Code Description Supporting Information

This appendix serves to provide additional information for the detailed analysis of the code implemented in the project.

B.1 Configuration Settings

Listing 2: Struct for configuration settings

```
type ConfigSettings struct {
   Protocol string
   Connection string
   Proxy string
}
```

Listing 3: Manipulation of configuratuion settings

```
func (config *ConfigSettings) WriteConfig() error {
   writeLines := config.Protocol + "\n"
   writeLines += config.Connection + "\n"
   writeLines += config.Proxy
   err := ioutil.WriteFile("../../config/connection_config.txt", []byte(writeLines), 0644)
   return err
}
// function responsible for checking the user inputs and edditing the configuration folder
// inputs - config: the configuration settings read in from a folder
//        configStatement: a string representing the configuration setting that must be changed
//        connectionType: the change that must be changed to configStatement
func (config *ConfigSettings) CheckInput(configStatement string, connectionType string) {
   switch configStatement {
      case "protocol":
         config.Protocol = connectionType
         break
      case "connection":
         config.Connection = connectionType
         break
      case "proxy":
         config.Proxy = connectionType
         break
      default:
   }
   // write the new configuration settings to the configuration file
   config.WriteConfig()
}
```

B.2 Client

Listing 4: Establishing client sever initial connection

```
   conn, err := net.Dial(config.Protocol, dialHost)
   lib.CheckError(err)
   defer conn.Close()
```

Listing 5: Client read from connection

```
   headerSize := getHeaderSize(version, code, status, headers)
   lengthDiff := 0
   // get content length from headers in reply from server
   contentLen, err := strconv.Atoi(headers["Content-Length"])
   if err == nil {
      // get the remainer of data that needs to be read
      lengthDiff = contentLen + headerSize - 65000
   } else {
      lengthDiff = -1
   }
   // if the header mentions chunked then need to read more data (HTTP/1.1)
   if strings.ToUpper(headers["Transfer-Encoding"]) == "CHUNKED" {
      // itterate until all data is read in
      for {
         var buf [65000]byte
         // read input
         n, err = conn.Read(buf[0:])
         lib.CheckError(err)
         response += string(buf[0:n])
         // break if EOF character is read in or if there is not more data to read in
         if strings.Contains(response, "\r\n0\r\n\r\n") || n == 0 {
               break
         }
      }
```

```
    } else {
        // itterate until no more data to read in
        for lengthDiff > 0 {
            var buf [65000]byte
            // read input
            n, err = conn.Read(buf[0:])
            lib.CheckError(err)
            response += string(buf[0:n])
            lengthDiff -= 65000
        }

    }
```

Listing 6: Client retrieve sources function

```
func retrieveSources(body string) map[string]string {
    // find all occurances of src in the file
    reg := regexp.MustCompile("src=\"(.*?)\"")
    allMatches := reg.FindAllStringIndex(body, -1)

    var sourceStrings []string

    urlToFileMap := make(map[string]string)
    // for all the occurances of src extract everything the is referenced
    for _, value := range allMatches {
        sourceStrings = append(sourceStrings, body[value[0]:value[1]])
    }
    // for all sources extract the new host and url and parse it into the map
    for _, value := range sourceStrings {
        withoutHttp := strings.Split(value, "//")
        splitURL := strings.SplitAfterN(withoutHttp[1], "/", 2)
        urlToFileMap[strings.Replace(splitURL[0], "/", "", 2)] = "/" + strings.Replace(splitURL[1], "\"↩
            ", "", 2)
    }
    return urlToFileMap
}
```

B.3 Server

Listing 7: Server TCP and DUP handing

```
func main() {
    // set the port on which to listen
    service := ":1235"

    go startTCPServer(service)

    go startUDPServer(service)

    // keep server running
    for {

    }
}

// function responsible for starting and running the TCP server
// inputs - a string containing the port on which to run the server
func startTCPServer(service string) {
    defer fmt.Println("closing TCP server")
    //create a listener for a TCP connection on the given port
    listener, err := net.Listen("tcp", service)
    lib.CheckError(err)

    for {
        // make a new socket for any TCP connection that is accepted
        conn, err := listener.Accept()
        if err != nil {
            continue
        }

        // handle the TCP connection on a new thread
        fmt.Println("New connection for ", conn.RemoteAddr())
        go  handleTCPClient(conn)
    }
}

// function responsible for starting and running the UDP server
// inputs - a string containing the port on which to run the server
func startUDPServer(service string) {
    defer fmt.Println("closing UDP server")
    packetConn, err := net.ListenPacket("udp", service)
```

```
    lib.CheckError(err)

    for {
        // handle any UDP connection
        handleUDPClient(packetConn)
    }
}
```

Listing 8: Server proxy GET interfaction

```
            case "GET":
                // get last modified time
                stat, err := os.Stat(path + url)
                if err != nil {
                    fmt.Println(err)
                }
                serverTime, _ := time.Parse(time.RFC1123Z, stat.ModTime().Format(time.RFC1123Z))
                proxyTime, _ := time.Parse(time.RFC1123Z, headers["If-Modified-Since"])

                // check if modified time is after a last modified time
                if headers["If-Modified-Since"] == "" || serverTime.After(proxyTime){
                    fmt.Println("200")
                    // compose 200
                        response.StatusCode = "200"
                    response.Phrase = "OK"

                    // load html file
                    file, err := os.Open(path + url)
                    if err != nil {
                        //need to figure out how to handle this
                    }
                    defer file.Close()
                    // read from file and convert to string
                    b, err := ioutil.ReadAll(file)
                    html := string(b)

                    response.EntityBody = html
                    response.HeaderLines["Content-Length"] = strconv.Itoa(len([]byte(response.↩
                        EntityBody)))

                    // add last modified header
                    response.HeaderLines["Last-Modified"] = serverTime.Format(time.RFC1123Z)
                } else {
                    fmt.Println("304")
                    // compose 304
                        response.StatusCode = "304"
                    response.Phrase = "Not Modified"

                    response.EntityBody = ""
                    response.HeaderLines["Content-Length"] = strconv.Itoa(len([]byte(response.↩
                        EntityBody)))
                }
```

B.4 Proxy

Listing 9: Server proxy GET interfaction

```
    exists, _ := lib.FileExists("../../cache/"+host)
    if !exists {
        os.MkdirAll("../../cache/"+host, 0777)
    }

    ioutil.WriteFile("../../cache/"+host+newUrl, []byte(body), 0777)
```

Listing 10: Conditional GET

```
        isInCache, lastModified, locationMap := checkInCache(url, strings.Split(host, ":")[0])
        // if is in cache then modify the request message to include the last modified date and ↩
            recompile message
        if isInCache {
            headers = modifyHeaders(lastModified, headers)
            message = compileNewRequest(method, url, version, headers, body)
        }
        // get the response message from the server
        serverResponse := handleServer(message, host)
        // check the server for the file and if it has been modified
        isUpdated, newResponse, newTime := getNewResponse(serverResponse, strings.Split(host, ":")↩
            [0], url)
        // if file has been modified then write new file to cache
        if isUpdated {
```

```
            destination := strings.Split(host, ":")[0]+url
            locationMap[destination] = newTime
            saveMap(locationMap, "../../cache/cache_map.txt")
        }
        // write the response message back to the client
        _, err = conn.Write(newResponse.ToBytes())
        lib.CheckError(err)
```

B.5 Other Functionality

Table 4: Different methods for invoking server response errors

Function Name	File	Author	Description
newRoundTripTimer	timer.go	DT	Create new timer struct object
loadTimerMap	timer.go	DT	Load pre-existing timer values
startTimer	timer.go	DT	Start the timer
stopTimer	timer.go	DT	Stop the timer and calculate duration
addToTimer	timer.go	DT	Add the new values to the timer map
writeTimerToFile	timer.go	DT	Write new timer values to the timer map file
CheckError	check_error.go	JA	Interpret system error
ReadConfig	config_settings.go	DT	Read configuration settings from file
InitializeConfig	config_settings.go	DT	Initialize the configuration settings to last known settings
WriteConfig	config_settings.go	DT	Write configuration settings to configuration file
CheckInput	config_settings.go	DT	Interpret the user input for changing a specified configuration
DecomposeRequest	decompose_request.go	JA	Extract the HTTP method, desired URL, HTTP version, header lines, and body from the client request message
DecomposeResponse	decompose_response.go	DT	Extract the HTTP version, status code, status message, header lines, and body from the server response message
FileExists	file_exists.go	JA	Determine if the file at the desired path exists
NewRequestMessage	request_message.go	DT	Create new request message struct object
SetRequestLine	request_message.go	DT	Set the request line of the request message object
SetHeaders	request_message.go	DT	Set the header lines of the request message object
SetEntityBody	request_message.go	DT	Set the entity body of the request message object
SetRequestMessage	request_message.go	DT	Call the above three set methods on the request message object
ToString	request_message.go	DT	Convert the request message object into a string
ToBytes	request_message.go	DT	Convert the request message string into bytes
NewReponseMessage	response_message.go	JA	Create a new response message struct object
ToString	response_message.go	JA	Convert the response message object to a string
ToBytes	response_message.go	JA	Convert response message string to bytes
loadMap	map_handler.go	DT	Load text file from server into a map
saveMap	map_handler.go	DT	Save new map to the server as a text file
handleRequest	client.go	DT	Responsible for handling the request to the server dependant on configuration settings
writeReceivedToFile	client.go	DT	Write the received file from the server to a temporary file for launching
getRedirectionLocation	client.go	DT	If the server response is a 301/302 then determine the new location of the file from the response message
getUserInputs	client.go	DT	Get the user to input the HTTP method, URL and body if the method was PUT or POST
getFileName	client.go	DT	Retrieve the file name from the entire URL string
retrieveSources	client.go	DT	Extract all the sources from the body of the response message and place them in a map
printToConsole	client.go	DT	Print the server response message to console
getHeaderSize	client.go	DT	Get the size of the response message headers in bytes
startTCPServer	server.go	JA	Establish a TCP connection with the client
startUDPServer	server.go	JA	Establish a UDP connection with the client
handleUDPClient	server.go	JA	Perform read and write functionality for UDP client
handleTCPClient	server.go	JA	Perform read and write functionality for TCP client
composeResponse	server.go	JA	Compose a new server response message based on the HTTP method and URL supplied by the client request message
loadMovesMap	server.go	JA	Load The map containing the previous and current location of all objects that have been moved
handleClient	proxy.go	JA	Handle all read and write functionality for proxy-client interface
getNewResponse	proxy.go	JA & DT	Compose a new response message depending of the status of the cache
checkInCache	proxy.go	JA	Checking if the file exists in the proxy cache
modifyHeaders	proxy.go	DT	Modify the headers based on last modified information from cache
compileNewRequest	proxy.go	DT	Compile a new request message
handleServer	proxy.go	DT	Handle all read and write functionality for proxy-server interface
getHeaderSize	proxy.go	JA	Get the size of the message header field

ELEN4017_project

The project entails the implementation of a client, server and proxy intereaction through the use of the Go programming language

Prerequisits

Follow the instructions on the Go Webpage to install Go.

NOTE: When setting the `$GOPATH` as mentioned in the above link, set it to the project root directory (i.e. one directory above the `src/` folder)

Directory Structure

- Project Root
  - cache: proxy cached files and maps
  - config: system configuration settings
  - documentation: report and timer related data files
  - objects: all server objects
  - src: all source code
  - temp: temporary directory for launching client files

Using The Application

In order to run the server, proxy and client the user is required to navigate to the respective folder, for example `/src/server/` from this directory the user can execute the command `go run *.go`. By default the server and proxy do not accept any arguements. The client is configured to accept two different type of aguements, these are:

- `go run *.go <config> <new setting>`
  - `go run *.go protocol upd/tcp`: sets the protocol to either TCP or UDP
  - `go run *.go connection close/keep-alive`: sets the connection type to either non-persistent or persistent
  - `go run *.go proxy off/proxyIP:port`: sets the proxy to on or to connection to the proxy on the specified address
- `go run *.go <options>`
  - `go run *.go destinationIP:port`: dials into the server on the specified address
  - `go run *.go print-config`: prints the current configuration settings to screen

NOTE: The proxy and server must be running before the client can be run. If the proxy is set to off then only the server needs to be running

User Inputs

When dialing into the server the client must specify the host IP or DNS address: > `go run *.go localhost:1235` or `go run *.go www.amazon.com:80` (port :1235 was specified as the localhost port for the server and :1236 was specified as the localhost port for the proxy)

Once the user runs the client connection to the server the user will be prompted to enter the method, this can be of the form of one of the following: GET, HEAD, PUT, POST, DELETE. The user will then be prompted to enter the URL, this is the location of the desired file of the server. > localhost: `/index.html` or for amazon home page: `/`

NOTE: The URL **MUST** begin with a single forwardslash

If the method that was specified was either PUT or POST the user will then be prompted to enter the body of the message, this can be in the form of anything BUT if the user desires it to be an HTML page then the user must enter the text in full HTML format.

1

C Time Management

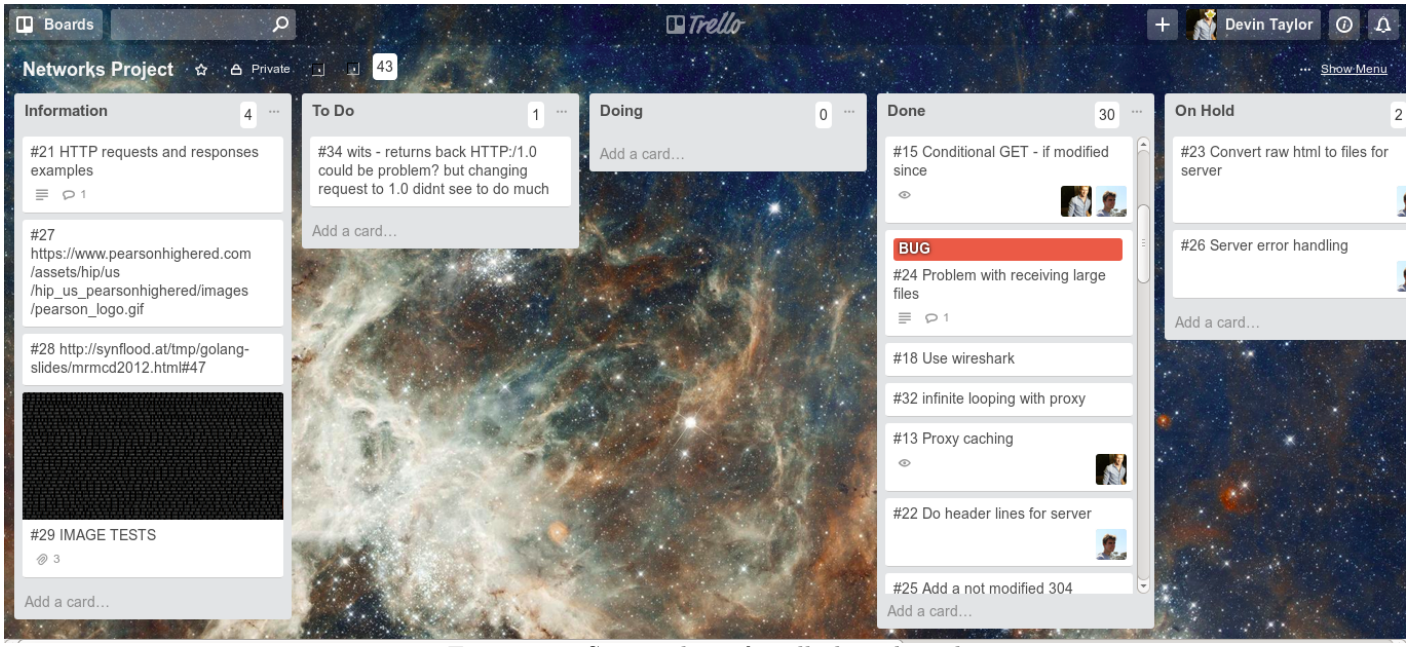

Figure 18: Screen shot of Trello board used

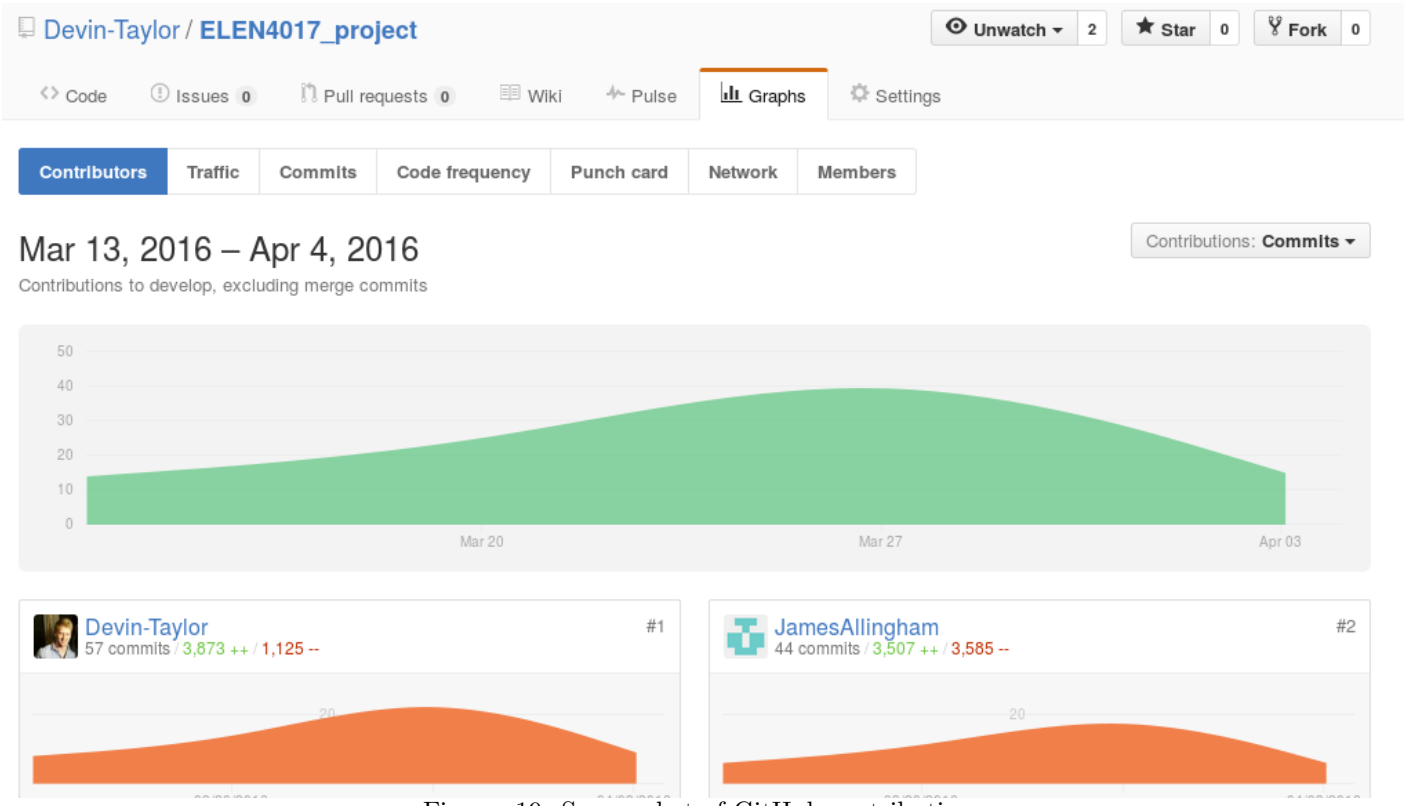

Figure 19: Screen shot of GitHub contributions