Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Stefan Belic

Cache and Non-Cache Performance Evaluation of an HTTP-CoAP Proxy

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Stefan Belic

Cache and Non-Cache Performance Evaluation of an HTTP-CoAP Proxy

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Franz Korf

Abgegeben am 20.11.2019

Stefan Belic

Thema der Arbeit

Cache und Non-Cache Performance Evaluation eines HTTP-CoAP Proxys

Stichworte

HTTP, CoAP, Proxy, Cache

Kurzzusammenfassung

Angesichts der steigenden Anzahl von Geräten im Internet der Dinge (Internet of Things) und der Verwendung des Hypertext Transfer Protocol (HTTP) als wichtiger Bestandteil des modernen Internets ist es für IoT-Geräte nur sinnvoll, HTTP zu implementieren, um in der Lage zu sein mit anderen Geräten im Internet kommunizieren zu können. Da IoT-Geräte im Vergleich zu anderen Geräten über weniger Ressourcen verfügen, wurde mit dem Constrained Application Protocol (CoAP) eine weniger ressourcenintensive Alternative zu HTTP entwickelt. Obwohl CoAP mit HTTP kompatibel sein soll, wird immer noch ein Mittelsmann benötigt, der Nachrichten von einem Protokoll zum anderen zuordnen kann. Ein solcher Mittelsmann kann ein Proxy sein. Neben der reinen Zuordnung kann ein Proxy auch andere Funktionen wie Caching implementieren, um die Leistung zu steigern.

Caching in einem Netzwerk, das aus Echtzeitgeräten besteht, kann einen anderen Einfluss auf die Leistung haben als geplant. Zum Beispiel in der Antwortzeit auf eine Anfrage. Um zu sehen, wie sich das Caching auf die Antwortzeit auswirkt, basierend auf Variablen wie der Anzahl der Clients und Echtzeitserver in einem Netzwerk, wurden Experimente durchgeführt. Um diese Experimente zu ermöglichen, wurde ein HTTP-CoAP-Proxy implementiert. Dieser Proxy wurde dann in eine Emulationsumgebung gestellt, in der die Experimente ausgeführt wurden. Es wurden vier verschiedene Client-Server-Kombinationen eingerichtet: ein Client und ein Server, ein Client und fünf Server, fünf Clients und ein Server sowie fünf Clients und fünf Server. Die Antwortzeit für jede Kombination wurde mit und ohne aktivierten Cache gemessen.

Aus den Ergebnissen dieser Experimente wird der Schluss gezogen, dass Caching in jedem Fall einen Vorteil mit sich bringt. In einigen Fällen mehr als in anderen. Da die Anzahl der Clients und Server so gering ist, muss noch weiter untersucht werden, was in einem Netzwerk mit mehr Clients und Servern geschieht und welche Trends die Abnahme oder Zunahme der Leistung bestimmen.

Stefan Belic

Title of the paper

Cache and Non-Cache Performance Evaluation of an HTTP-CoAP Proxy

Keywords

HTTP, CoAP, Proxy, Cache

Abstract

With the rise in the number of Internet of Things (IoT) devices and the usage of the Hypertext Transfer Protocol (HTTP) as an important part of the modern Internet, it only makes sense for IoT devices to implement HTTP to be able to communicate with the rest of the Internet. Because IoT devices in practice have less resources available to them in comparison to other devices, a less resource intensive alternative to HTTP has been developed in the form of the Constrained Application Protocol (CoAP). Even though CoAP is meant to be compatible with HTTP there is still need for an intermediary that can map messages from one protocol to the other. Such an intermediary can be a proxy. Other than just mapping, a proxy can also implement other functionality, such as caching, to increase performance.

Caching in a network consisting of real time devices could have a different influence on performance than planned. For example, in the response time to a request. To be able to see how caching affects the response time based on variables, such as the number of clients and real-time servers in a network, experiments have been done. To make these experiments possible a HTTP-CoAP proxy was implemented. This proxy was then placed in an emulation environment where the experiments have been executed. Four different client-server combinations have been setup: one client and one server, one client and five servers, five clients and one server and five clients and five servers. The response time for each combination was measured, with and without caching enabled.

From the findings of these experiments it is concluded that in each case caching does bring a benefit with it. In some cases, more than in others. Because the number of clients and server is so small there is still a need for further research into what happens in a network consisting of more client and servers, and into the trends governing the decrease or increase of performance.

Table of Content

Table of Figures

1 Introduction

This paper will evaluate the cache and non-cache performance of an HTTP-CoAP proxy in a network consisting of real time servers. To answer why such findings are important the following chapter will present a short motivation. In it, the importance from why the protocols HTTP and CoAP were chosen, to why caching is taken into consideration, and not some other feature, are going to be shown. After the motivation, already existing work on the implementation of existing proxies and research on caching will be shortly mentioned. Lastly a look at the way in which the performance was measured will be taken, followed with the structure of the rest of this paper.

1.1 Motivation

In today's world more and more people are getting connected to the Internet. As of writing this paper, more than half of the world population possess an Internet connection (1). These people then use the Internet to access many services offered on top of it. Such services can for example be e-mail, communication services for messaging and calling, file sharing services, the world wide web, and others. One of the more important services and the service most people associate with the Internet is the World Wide Web (WWW). It is an information system for sharing resources over the Internet (2) used for accessing social media, online shopping, online banking, and other. The underlying protocol enabling the WWW is the Hypertext Transfer Protocol (HTTP) (3). Other services are also starting to use HTTP. One reason for this is that they are increasingly following the representational state transfer (REST) architecture style, aiming for faster performance and better scaling (4) for which HTTP is a perfect fit. Another very important reason is that by implementing HTTP they are becoming accessible over the WWW. These services then become web services (5). Users are already familiar with the WWW and the tools, for example a browser, used for accessing it. Therefore, making services available over the WWW makes it easier for users to access them, thereby increasing the number of users using those services.

A rising trend that can be observed is the Internet of Things (IoT) (6), where instead of people, more and more devices are connecting to the Internet. These devices can then use or offer services via the WWW. To be able to do that they must implement the HTTP protocol. Most of them are constrained devices possessing limited resources compared to other devices on the web, making HTTP not a viable option. Because of this, another protocol that is less resource intensive while still being compatible with HTTP is required. One such protocol specifically designed for this is the Constrained Application Protocol (CoAP) (7). Nevertheless, even though CoAP was designed to be easily mapped to and from HTTP, mapping between them is still necessary. One device responsible for this is called a proxy.

Proxies (8) can offer more functionality than just mapping between protocols. For example, caching requested data, load balancing towards receiving devices and more. Many of these features can be implemented in an HTTP-CoAP proxy to improve performance. However, IoT devices have different properties than traditional devices on the web. Because of that some implementations, such as caching, may not have the desired performance benefits. Caching gives performance benefits when the resources offered by the service are static, or if the cached resource is being accessed many times by many parties before it must be refreshed. However, with IoT devices this is not the case. Most IoT devices are real time devices like sensor nodes. For that reason, caching might bring only small benefits if any.

By performing experiments on caching behavior for different use cases it is possible to see what can be gained by a proxy caching system in an HTTP-CoAP proxy for real time systems. If the findings are positive and caching does bring benefits even for real time systems, then these findings only highlight the importance of caching. On the other hand, if the benefits shown are not meaningful, removing the caching system will free up resource. These resources are then either usable in other parts of the proxy or can be removed entirely, thereby reducing cost.

1.2 Related Work

Not many papers exist on the topic of HTTP-CoAP proxies with even less on cache performance evaluation of such. Two papers of interest that talk about the implementation of proxies are "HTTP-CoAP Cross Protocol Proxy: An Implementation Viewpoint" (9) and "Connecting to the Web with the Web of Things: Lessons Learned From Implementing a CoAP-HTTP Proxy" (10). Both papers discussed proxy implementations with the focus in mapping between CoAP and HTTP and vice versa. The difference between these two is that the first paper mentioned also looked at the cross-protocol security, while the second paper looked at caching. Even though the second paper discussed caching performance, it did this in a more theoretical way. Another important paper that took a detailed look on cache performance is "Performance of Caching in a Layered CoAP Proxy" (11). In this paper the researchers investigated the performance of a proxy in terms of its cache-hit ration and the response time in an IoT domain.

1.3 Structure

To be able to evaluate the difference caching can make, a proxy with which the experiments are done is necessary. Even though proxy implementations exist, a proxy will be implemented from scratch for research purposes. The caching system implemented in the proxy can be turned on and off, making it possible to test both cases on one proxy. With this proxy multiple experiments will be done in sequence. Each experiment will be measured once with caching enabled and once disabled. The changing variable between the experiments will be the number of HTTP clients and CoAP servers participating. For the metric measured in the experiments the time it takes for a response to arrive back to the client is taken. Because real time systems are used, the cache time of their resource should be low. It was chosen to be 2 seconds because of the acknowledgment timeout given by the CoAP protocol so the consequences of lost packets can be seen.

In total, the paper will consist of six chapters. An introduction to all the required theoretical knowledge necessary for understanding the rest of this paper is done in chapter two. It introduces what a proxy is, how the HTTP and CoAP protocol work, how mapping is implemented, and what caching is. The third chapter will then show the design of the proxy, which includes the requirements it must meet, and the structure and execution model used to solve them. Chapter four will talk about the tools used in the experiment and how the experiment is set up. The data measured in the experiment will be shown and analyzed in chapter five. Finally, the sixth chapter will be a conclusion of what these findings found, and the works that can be done in the future.

2 Theory

Before the design of the proxy is shown and thereafter the data measured analyzed, it is important to have a firm theoretical understanding of what is to come. For that reason, this chapter will introduce all the knowledge necessary for the rest of this paper. It starts with an introduction to what proxies are and how they are differentiated. Following that, all parts of the HTTP and CoAP protocol important for the implementation of the proxy implemented in this paper will be mentioned. Lastly, the HTTP-CoAP mapping used will be discussed and the necessary caching knowledge shown.

2.1 Proxy

A proxy is usually a computer system - a combination of hardware platforms and software applications – which serves as an intermediary in the network communication between the parties (8). When the proxy operates on the seventh layer of the Open Systems Interconnection (OSI) model (12) it is also called an application proxy. A proxy contains at least two parts; a server component for receiving requests from the clients and a client component for forwarding them to the servers. The primary use of proxies is allowing access to external sources on the web from within a firewall (13). In addition to that, proxies can also offer other functionalities, for example:

- Restricting which servers or resources can be accessed by the clients
- Filtering malicious content
- Modification of the content
- Monitoring of content
- Mapping between protocols
- Caching
- Bypassing regional restrictions

Based on the functionalities that a proxy offers and the role it plays in a network it can be categorized into two types; forward and reverse proxy.

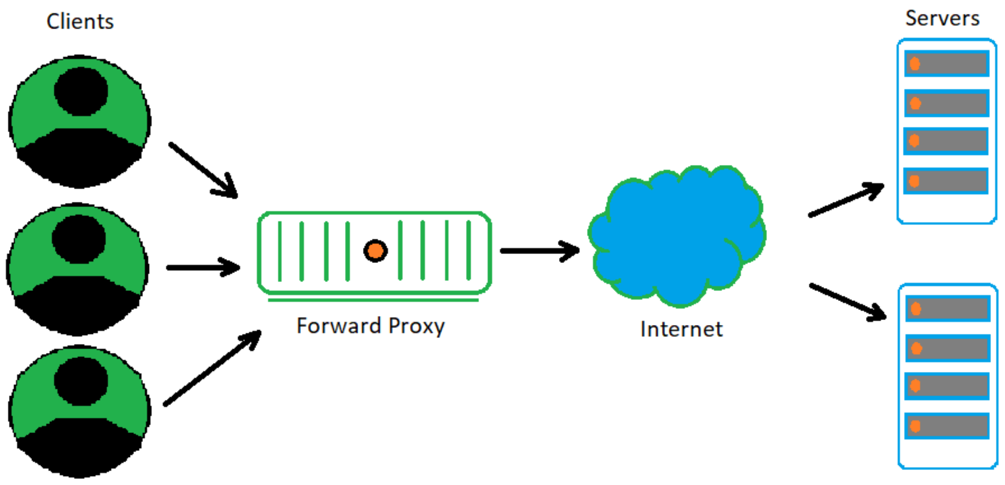

Figure 1 (Internet clients accessing resources on the Internet through a forward proxy)

Forward proxies provide services to a group of clients by representing them to the requested servers. By doing this, the IP of the clients is hidden to the servers and instead the server will see the request as if it is coming from the proxy. Because of this, a forward proxy can be used to bypass regional restrictions, or restricted content in general. This can be done by placing the proxy in a different region where the content is allowed and then accessing the content through the proxy.

Reverse proxies, as opposed to forward proxies, represent a group of servers on the Internet. A reason why this would be useful is that it makes it easy to hide the internal structure of the server network. When a client sends a request, the request is received by the reverse proxy without the knowledge of the client. The proxy then forwards the request to a server that will handle the request. This brings benefits other than just hiding the internal network. By letting reverse proxies choose which servers will handle the request, they can implement load balancing techniques by dividing the work between the accessible servers. Another benefit is increased security by filtering and modifying requests before handover.

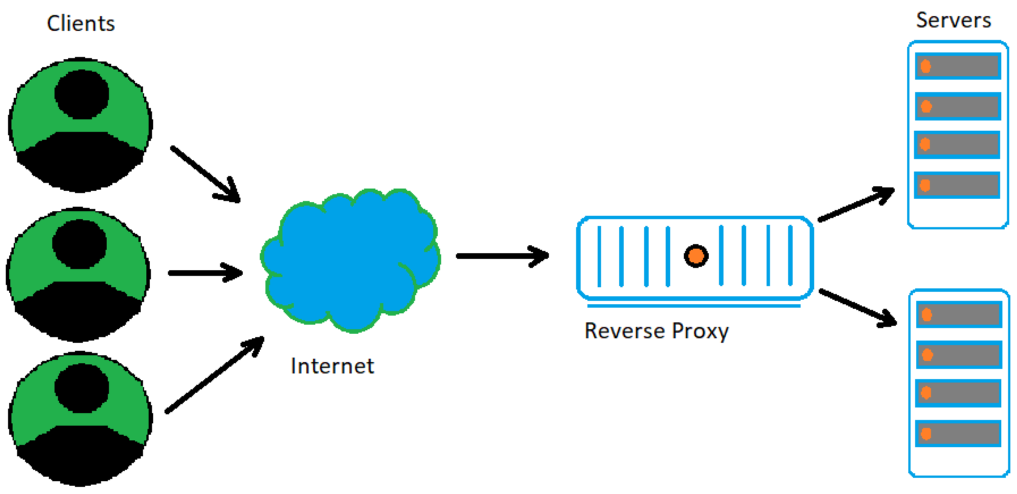

Figure 2 (Reverse proxy as intermediary between the web server and the Internet)

2.2 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems (3). It is a stateless protocol used as the underlying protocol by the World Wide Web. There are multiple versions of HTTP, the newest one being HTTP/2. Even though HTTP/2 is the newest version, this paper will focus on HTTP/1.1. The reason for this is that more libraries and devices support it. Moreover, HTTP/1.1 is simpler to use and debug because of its textual format instead of the binary format used in HTTP/2.

HTTP is a request/response protocol. As shown in Figure 3, this means that a client, such as a web browser sends request to a server. The server handles the request and sends back a response. In the rest of this chapter we will look at the structure of these messages.

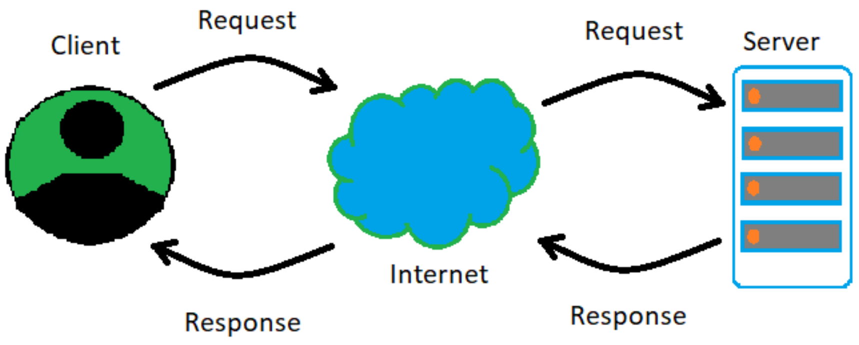

Figure 3 (Client-Server request response communication)

2.2.1 Request

A HTTP request message consists of a request line containing a request method, Uniform Resource Identifier (URI), and protocol version which is set to HTTP/1.1. The request line is then followed by headers and a message-body.

The request methods defined by HTTP/1.1 are:

OPTIONS	represents a request used to determine the options and/or requirement associated with a resource.
GET	retrieves information about the resource identified by the URI.
HEAD	is identical to GET, except that the server must not return a message-body.
POST	is used to request the server to create a new entity subordinate to the resource identified by the URI.
PUT	requests that the entity is stored under the requested URI. If no resource exists, it will be created. Otherwise it replaces the already existing resource.
DELETE	tells the server to remove the requested resource identified by the URI.
TRACE	invokes a remote, application-layer loop-back for the request message.
CONNECT	tells a proxy to dynamically switch to a tunnel.

These methods have two properties defined based on the interaction they have with the server:

Safe methods:	are those that only retrieve data and do not do any action on them. Only HEAD and GET are safe methods.
Idempotent methods:	have the property that each side-effect of N > 0 identical requests is the same as for a single request. The methods GET, HEAD, PUT and DELETE possess this property.

For a method to be cached, it is important for it to be both safe and idempotent. Thus, only GET has been implemented as a method in this paper's proxy. Even though HEAD is also safe, idempotent, and almost identical to GET, it must not return a message body. Therefore, no new findings can be made with it, thus can be safely ignored.

Uniform Resource Identifier (URI), as its name implies, is used to identify onto which resource the method is meant to be applied to. The URI can be formatted in four ways:

"*":	is used when the request is planned for the server itself and not a resource.
Absolute URI:	contains the full path to a resource and is used when the request is being sent to a proxy.
Absolute path:	is the most used URI format and is used to identify a resource on the server
Authority:	is only used by the CONNECT method

Out of these options a mix of absolute path and absolute URI will be used. Further details about its structure will be explained in the chapter HTTP to CoAP mapping.

The header fields and the message-body are used to send additional data about the client and the request to the server. We will not go into further detail because these were not used in the experiment.

2.2.2 Response

A HTTP response consists of a status line, header options and a message body.
The status line itself is made of a HTTP version, a status code and reason phrase. As mentioned before, the HTTP version will always be set to HTTP /1.1.
The status code is a three-digit integer defining the response. The first digit of the status code defines the response class. Though there are nine possible options only five classes are defined:

1 Informational:	indicates a provisional response, consisting only of the status line and optional headers.
2 Success:	indicates that the request has been successfully handled.
3 Redirection:	indicates that further actions must be taken by the client before the request can be fulfilled.
4 Client Error:	indicates that the request created by the client contains an error.
5 Server Error:	indicates that the server failed in some way or form to handle the request.

The last two digits define the status in more detail.

Depending on the status code the reason phrase is predefined, as it is meant to be a short description of the status code in human readable from. For example, the reason phrase “OK” is set when the status code is 200.

The header fields can carry additional information about the response or about the message-body. Finally, the message-body contains in most cases the data of the requested resource or the status of an action.

2.3 Constrained Application Protocol

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol
for use with constrained nodes and constrained (e.g., low-power, lossy) networks (6). It was created to easily interface with HTTP while fulfilling other needs special to constrained environments. One major difference in contrary to HTTP is that CoAP was designed to work with the user datagram protocol (UDP) (14) as its transport protocol. CoAP can be split into two layers based on the job they fulfil. The first one is a messaging layer designed to deal with UDP and its asynchronous nature of interactions, while the second one is the layer responsible for the request/response interaction comparable to HTTP. These two layers will be defined in more detail in the rest of this chapter.

2.3.1 Messaging Layer

The CoAP messaging layer is based on the exchange of messages over UDP between endpoints. CoAP messages are defined in such a way that each message must be able to fit inside of a UDP datagram. It consists of a 4 byte long binary header, followed by a token value, a compact binary option and a payload.

The header consists of five values, these are: version, message type, token length, code and message ID.
The version is a two-bit unsigned integer meant to indicate the CoAP version and it currently must be set to 1. As for the other values only the message type and message id are used by the message layer. Because of this, only they will be mentioned in this part while the rest will be discussed in the request/response layer.
The message type is used to differentiate between the four possible kinds of messages. These can be either a confirmable, non-confirmable, acknowledgment or reset message.
Confirmable messages are sent with the expectation that the server replies with an acknowledgment or reset message as a response. If none of these messages are received, the client is to resend a confirmable message until several tries have been done. The first resend happens after 2 seconds. Each following resend timeout is then doubled. This is done four times. If no response has arrived the message is not sent anymore.
Non-confirmable messages do not require to be acknowledged and are expected to be used on repeated actions, for example sensor readings.
An acknowledgment message is sent by the server to acknowledge a confirmable message, it by itself does not have to carry any response with it but it can be used to piggyback response.
Lastly, the reset message is sent by the server when either a confirmable or non-confirmable message is received, but it could not be properly processed. To differentiate between messages, a message-id is used. The message-id must be unique for each confirmable and non-confirmable message. The acknowledgment and reset message do not carry a unique ID but rather the ID of the message they are a response to.

2.3.2 Response/Request Layer

The response/request layer takes care of defining the type of request, response and how they correlate to each other. For that reason, CoAP uses tokens and a code.

The token value is a variable length identifier (0-8 bytes) and is generated by the client for the purpose of differentiating between concurrent requests. The length of the token is found in the header under the token length field. Because a token is used to match a response to a request, the response can be sent at any time within a set limit, and a confirmable request can be answered with a non-confirmable response or the other way around. If the client does not need to use a token, for example if the communication with a server is one request at a time, it can be left empty without consequences by setting the token length to 0.
The code field is used to determine if it is a response or request, and if it is a response what kind of response. It consists of two parts; the class, which is three bits long, and another five bits used for a more precise description. Four possible classes are defined:

1 – Defines the message as a request
2 – Success
3 – Client Error
4 – Server Error

Other than these two values, options can be added to a CoAP message to add additional details about the request or response. They are inserted between the token and the payload of a message and carry additional data, for example the URI, or other data that are normally carried by headers in HTTP.

2.4 HTTP to CoAP Mapping

Even though CoAP an HTTP implement similar request/response handling, there is still a need for mapping between protocol elements. The mapping can be implemented in many ways, but this papers implementation is based on the implementation guidelines from the RFC 8075 (15). The guidelines go into detail about different aspects of mapping, but only one is important for the implementation of the proxy described in this paper, which is mapping of the Uniform Resource Identifier (URI). The other mappings are not necessary because they are predefined, for example the media type used by client and server will be plain text, so no media mapping is necessary.

URI mapping is required because all major web browsers, networking libraries and command-line tools do not allow making HTTP requests using URIs with a scheme 'coap' or 'coaps'. For that reason, a HTTP URI must contain an CoAP URI in such a way to make it possible for the Proxy to redirect the request to the target server.
The RFC 8075 defines two different URI mapping types for different situations:

Null mapping is used when there is no target CoAP URI appended to the Proxy URI. This would typically be used in reverse proxies. It will not be implemented because the proxy implemented in this paper is a forward proxy, and all null mapping requests will return a service unavailable status code.

Default mapping is the case where a CoAP URI is directly appended onto a Proxy URI. The proxy must implement this mapping. For example, the proxy URI is http://proxy.com/proxy and the targeted CoAP URI is coap://coap-server.com/temp. The concatenated URI will be http://proxy.com/proxy/coap://coap-server.com/temp. The scheme coap:// can be omitted from the URI if a mutual agreement exists between the proxy and the clients, or else it must be included.

Other than the default and null mapping, other advanced template mappings are defined. They are divided into a simple and enhanced form.

2.5 Caching

Caching is a technique used to store a copy of a given resource so it can be served back when requested again (16). There are primarily two reasons for caching data: improving performance and improving reliability (12). Replication of data introduces a consistency problem whenever a replica is being updated. There are different consistency models, but in this paper only the one implemented by the web cache used on the World Wide Web will be shown.

A web cache can either be a private or shared cache. Private caches are meant to save resources for one single user, such as a browser cache. Shared cache saves responses for multiple users. As proxy caches are meant to improve the performance of multiple users, it is categorized as a shared cache.

Consistency in a web cache is easy to implement as no replica changes the data. Only the original data located on the web server is changed. Replicas revalidate data from the web server after some time. This way if the data has not changed on the web server each replica will be eventually consistent. For that reason, this form of consistency is called eventual consistency (12).

To make it possible for the web cache to know when to refresh its data headers in HTTP and options in CoAP are used. Even though both protocols define something for caching management, the implemented proxy decides its caching behavior based on options given in the CoAP protocol. Because of this only they will be mentioned.

For caching, CoAP only defines one option: max age. It is used to define how long a resource can be cached before it is not considered fresh anymore and must be requested again. If no max age option is set a standard value of 60 seconds is chosen. For any other value the max age must be set to the desired value. When a server does not want the resource to be cached it must set the max age to 0.

It is not possible for a web cache to have every web page cached because memory is limited. For that reason, it only caches data for the currently requested web servers. This makes a web cache a Client-initiated replica. When data is being requested that has not been cached a cache miss occurs. A cache miss also occurs when the data is cached but it must be refreshed. In a cache miss the data must be returned from the server. On the other hand, if the data is in the cache and is still valid a cache hit occurs. When a cache miss occurs, and the cache is full a response must be replaced. This is not important for this paper because the cache memory is big enough to cache the data from all servers used in the experiment. If this was not the case a cache replace mechanism like oldest first could have been implemented.

When implementing a cache, it is necessary to be able to identify the saved resource. Normally this is done with the help of a key that is based on the URI of the request. The response can then later be saved either on a physical medium like a hard disk or in RAM.

3 Design

After introducing all the necessary knowledge in the previous chapter, the following chapter will look on the design of the implemented proxy. But before the design is shown in more detail, requirements that the proxy could and had to fulfil are going to be mentioned first. After that, each of the components in the proxy, and the interfaces connecting them, will be described. Other than just the static model of how the components are connected, a dynamic model showing the function calls on runtime will also be shown. Both are presented with the help of a structural model.

3.1 Requirements

The proxy's main responsibility is the forwarding of HTTP request to the CoAP server and returning the CoAP responses back to the HTTP client. The reverse of it, forwarding CoAP request and HTTP responses, could also be implemented by the proxy, but because this was not going to be used it was not a requirement. The proxy should also be able to work autonomously without any user interaction other than in the initialization of the proxy. For the proxy to be able to communicate with the client and server over the network, it must implement a network stack. Each protocol can have its own network stack that is independent from the other. In the implemented proxy the first three layers of the network stack had not been implemented because it was planned for it to run as an application on top of an operating system that already has them. Because of this, only the last layer handling the corresponding protocol (HTTP or CoAP) had to be implemented, and a way to communicate with the operating system for the rest.
Each protocol must be able to run a minimum of five active connection at the same time to make it possible for all the experiments to be run. Messages of each connections must be handled and cannot be discarded. To do this, either a queueing mechanism can be implemented in the form of a buffer, or each message is handled in a separate thread.

Other than just having to handle the sending and receiving of messages, the proxy also had to implement two more important features: the mapping between the protocols and the caching of responses. The mapping is implemented as mentioned in the theory chapter. Only the necessary mapping of the URI is required. In the case of the caching implementation, because only the necessary functions are required to be able to run the experiments, they are set as a must. These includes saving and retrieving the data from the cache and managing the cache age so it can be refreshed. Other functions offered by the cache like memory management of the cached data and storing these on a different medium are not required. In each experiments the number of clients and servers are known so they can be pre allocated.

To be able to connect all these parts of the proxy together, a stable software architecture must be created. No specific architecture is required, but whatever architecture is implemented it must fulfill specific requirements. The first requirement is to make it possible that the received http message is mapped, if necessary cached, and then sent out as a CoAP request to the correct server. This is also true for the opposite direction when receiving an CoAP response. Built on top of this requirement is also that the proxy properly forwards the CoAP responses back to the correct client. CoAP messages do not contain any way to identify the original client, therefore it is the proxy's responsibility to manage this. No speed constraints are set on any of the mentioned requirements.

3.2 Proxy Design

The proxy design is represented with the help of a structural model. A structural model is used for representing the organization of the system in terms of components and the relationships between them. They can either be static or dynamic models. For representing the component architecture of the proxy and the interfaces between the components a static model is used. On the other hand, a dynamic model shows the interaction between the components on execution (17).

3.2.1 Static Model

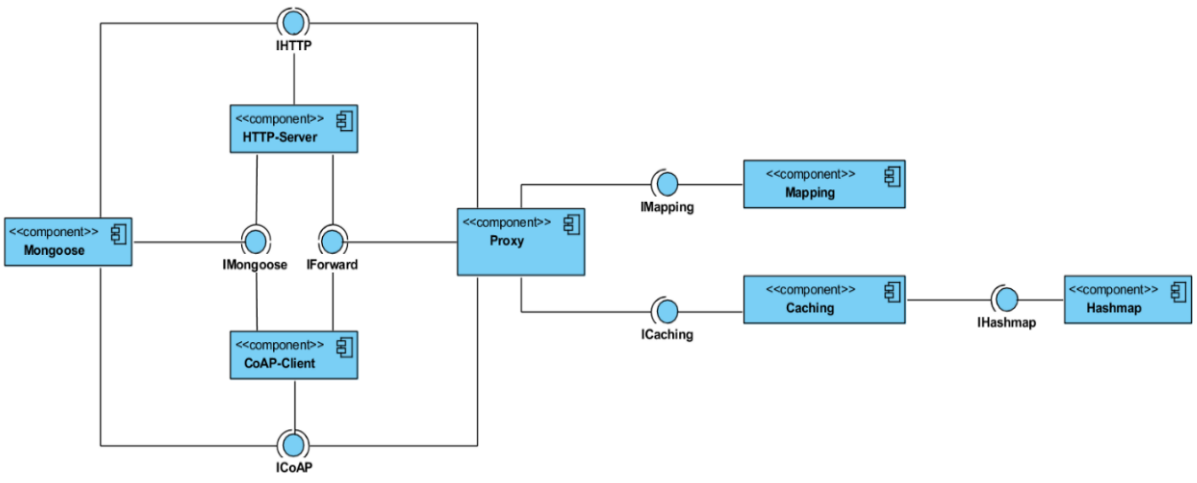

Figure 4 (All components of the proxy and the interfaces between them)

Shown in Figure 4 is the static model of the proxy architecture. As can be seen, the architecture consists of seven components. Five of these components are created from scratch, while external libraries are used for the last two. These components communicate with each other through interfaces defined between them. Each of the components will be mentioned in more detail in the rest of this chapter. The libraries used for implementing the two external components will be discussed in the next chapter.

The Mongoose component is responsible for communicating with the operating system in handling the first three network layers. Both the receiving and sending of messages. It also handles the marshalling and unmarshalling of the CoAP and HTTP messages. The HTTP-Server and CoAP-Client component can send messages through the IMongoose interface offered. For receiving messages, both the HTTP-Server and CoAP-Client component must implement a callback function that can be used by the Mongoose component. CoAP-Client defines its callback function trough the ICoAP interface while the HTTP-Server component defines its trough the IHTTP interface.

The HTTP-Server component is responsible for handling incoming HTTP requests. Request are received through the callback function defined in the IHTTP interface that the Mongoose component uses to forward HTTP messages. The component then parses the requested URI and checks if its structure is correct. If it is a valid request, the request is forwarded trough the IForward interface to the Proxy component. Otherwise it sends a response back to the client with an error status code. This is done through the IMongoose interface. Even if the URI is correct the HTTP-Server still sends a response back to the client if the Proxy component failed to forward the request. Other than just receiving request the HTTP-Server also offers functionality over the IHTTP interface to send a HTTP response to a given client. This is then used by the Proxy component to forward CoAP responses.

The CoAP-Client component handles the communication with the CoAP servers. It is responsible for managing each outgoing CoAP request. This includes matching incoming responses to their corresponding request and resending of messages in the case of lost confirmable messages. Requests are received from the Proxy component through the ICoAP interface. The ICoAP interface also offers a callback function that the Mongoose component uses to handover incoming CoAP messages.
When a received response matches an outgoing request, the CoAP component then forwards that message through the IForward interface to the Proxy component for further processing.

The Proxy component is the central part of the architecture. It communicates with all the other components to properly forward requests and responses. This includes linking CoAP responses to HTTP requests. HTTP requests that are to be forwarded are received through the IForward interface from the HTTP-Server component. These messages are then turned into CoAP messages through the help of the Mapping component. If Caching is enabled, the Proxy component also communicates with the Caching component to store and retrieve cached responses. A similar sequence is done when an CoAP response was received by the CoAP-Client component. A more detailed look on this will be shown in the dynamic models.

The Mapping component turns an incoming HTTP request into a CoAP request, or a CoAP response into an HTTP one. This is done following the mentioned mapping shown in the theory chapter. It offers all its functionality through the IMapping interface.

Lastly the caching component. It is responsible for managing the cached data inside a hashmap that is managed by the Hashmap component accessible through the IHashmap interface. This includes not only saving and retrieving the data from and to the Hashmap, but also managing the expiration date of the data and if it must be refreshed again.

3.2.2 Dynamic Model

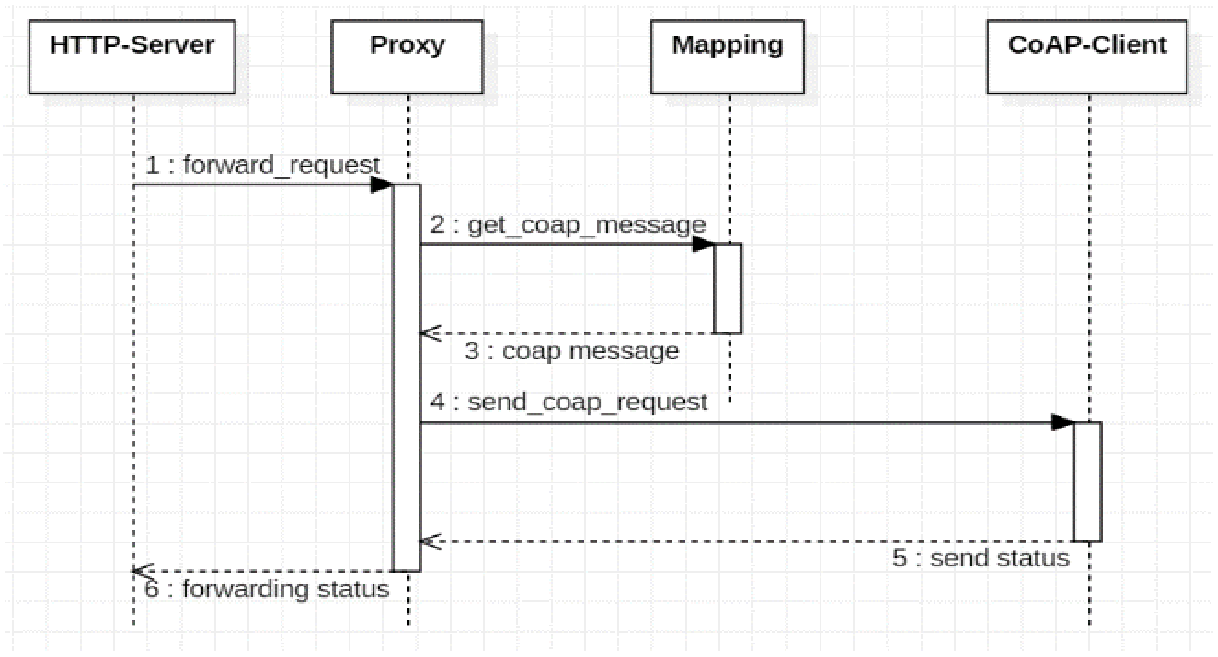

Figure 5 (Forwarding of an HTTP request in a proxy with cache disabled)

The dynamic model shows the interaction of the components during execution through function calls. All function calls shown are synchronous. Multiple sequences are possible, where a sequence represents the flow of a request or response. Because of this, multiple models will be used to represent all possible sequences. These models are categorized in two ways making it easier to understand them. The first categorization is based on the type of message being followed. It can either be a request or response sequence. The second categorization depends on if the proxy has caching enabled or disabled. Combinations of both categorizations will be observed in more detail in the rest of this chapter. In all models the communication between the Mongoose component and both the HTTP-Server and CoAP-Client components is not shown.

The first model, displayed in Figure 5, shows the sequence of a request in the proxy with caching disabled. Here the sequence starts with the HTTP-Server component calling the function “forward_request”. This function is implemented by the Proxy component and defined in the IForward interface. With it the HTTP request handled by the HTTP-Server is being handed over to the Proxy component. The Proxy component then hands it over to the Mapping component with the function call “get_coap_message”. This is done so the Mapping component turns the HTTP request into an CoAP request. After this has been done the Proxy component then calls the function “send_coap_request” defined in the ICoAP interface. Here the created CoAP request is handed over to the CoAP-Client to be sent. The CoAP-Client component does not immediately send the request, but rather changes its internal state and prepares itself for sending the CoAP request in a later moment in time. The success status of the state change is returned to the proxy, which then returns the forward success status to the HTTP-Server component.

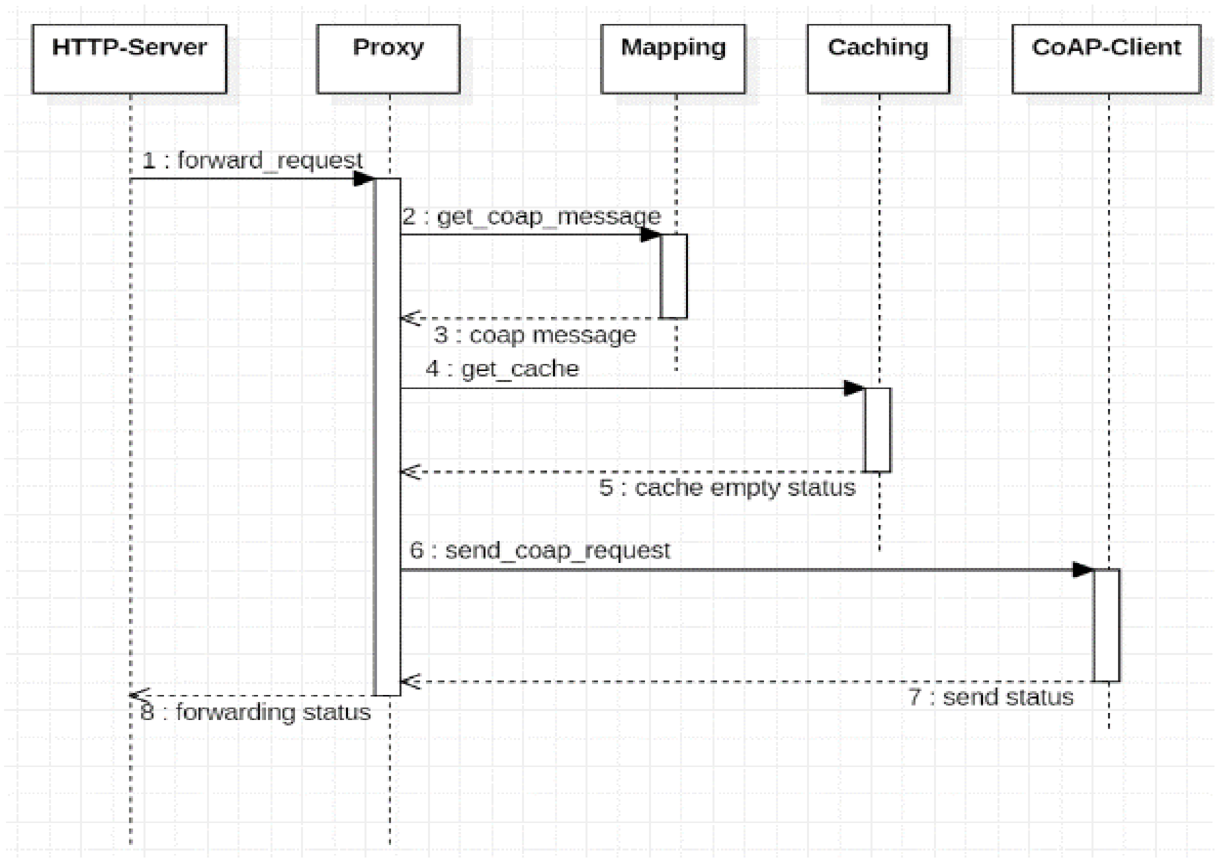

Figure 6 (Forwarding of an HTTP request in a proxy with cache enabled and a cache miss)

The models in Figure 6 and 7 both show the sequence of a request when caching is enabled. Two different models are used to represent the two different responses of the cache when reading from it. In Figure 6 the response is not cached so a cache miss occurs. In Figure 7 the response is cached. For both models the first two steps, the forwarding of the HTTP request to the Proxy component and the mapping of HTTP to CoAP, are the same as seen in Figure 5 so they will not be mentioned again.

As can be seen in Figure 6, after receiving the CoAP request, the proxy component requests the cached response from the Caching component. In this case the Caching component returns an empty object which is interpreted as a cache miss. As a result, the proxy component sends the CoAP request to the CoAP-Client in the same way as described for Figure 5.

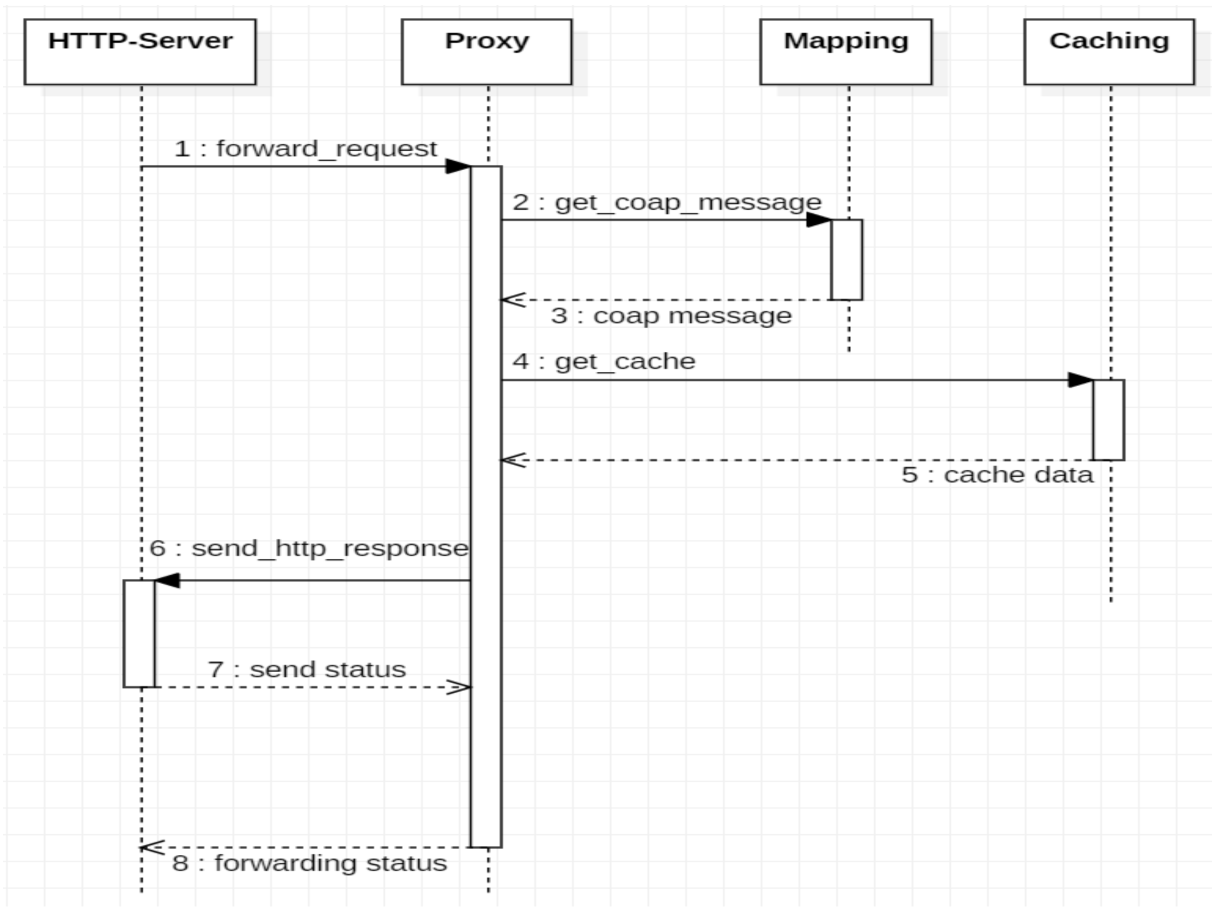

Figure 7 (Forwarding of an HTTP request in a proxy with caching enabled and a cache hit)

In Figure 7 the Proxy component also requests the cached response from the Caching component. But in comparison to the first case, a valid response has been returned. The cached response is already in the form of an HTTP response; thus, no mapping is necessary. This response is then handed over to the HTTP-Server component through a call of the "send_http_response" function. This call does not change the internal state of the HTTP-Server component in the same way "send_coap_request" changes the state of the CoAP-Client component. Instead it sends the HTTP response immediately. The HTTP-Server component then returns the send status back to the Proxy which then returns the forwarding status back to the HTTP-Server. Carrying it out this way keeps the components separated in the role they are fulfilling.

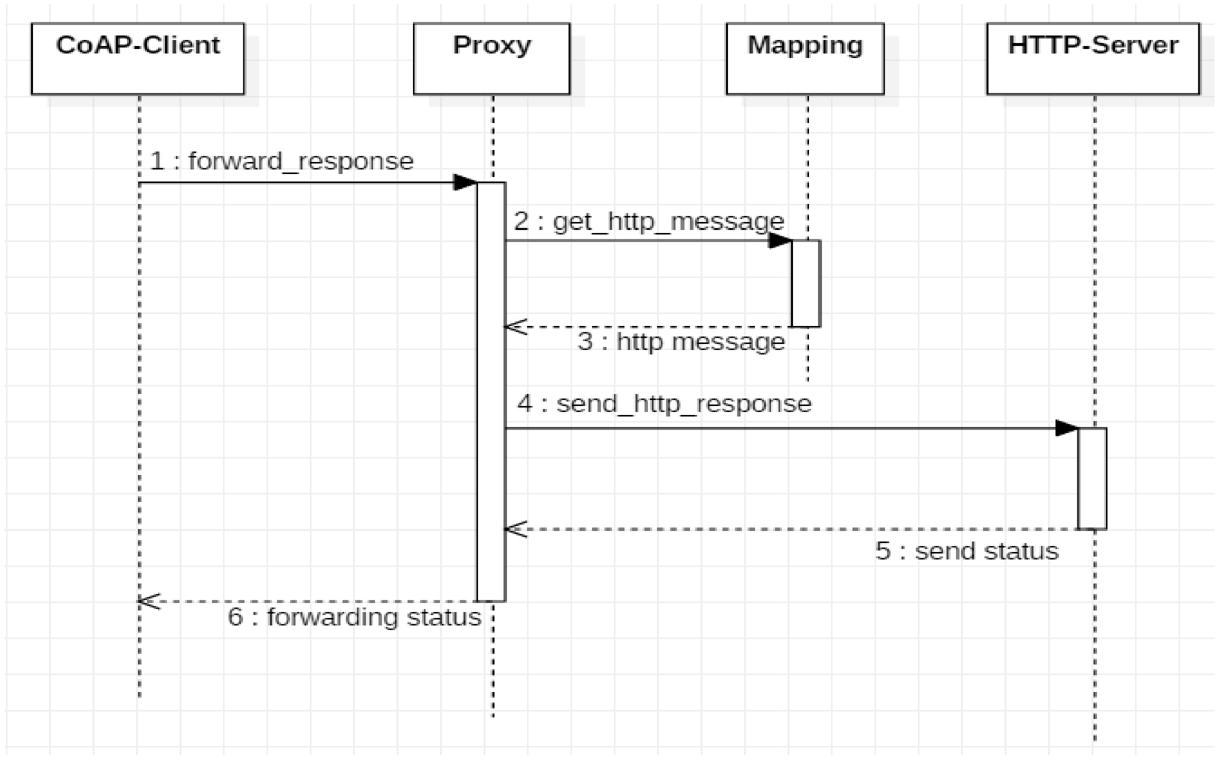

Figure 8 (Forwarding of an CoAP response in a proxy with caching disabled)

The next two models show sequences of a response message. Responses in a proxy are handled almost identically to the way request are handled. As can be observed in the difference between Figure 8 and 5, the difference between them is only that the CoAP-Client and HTTP-Server component are swapped in their location in the sequence. Following that few things change. First the function calls are different. Instead of “forward_request”, “forward_response” is used, and instead of calling “send_coap_request” on CoAP-Client, “send_http_response” on HTTP-Server is called. Another difference is that the Proxy component now receives CoAP responses that must be mapped into HTTP responses.

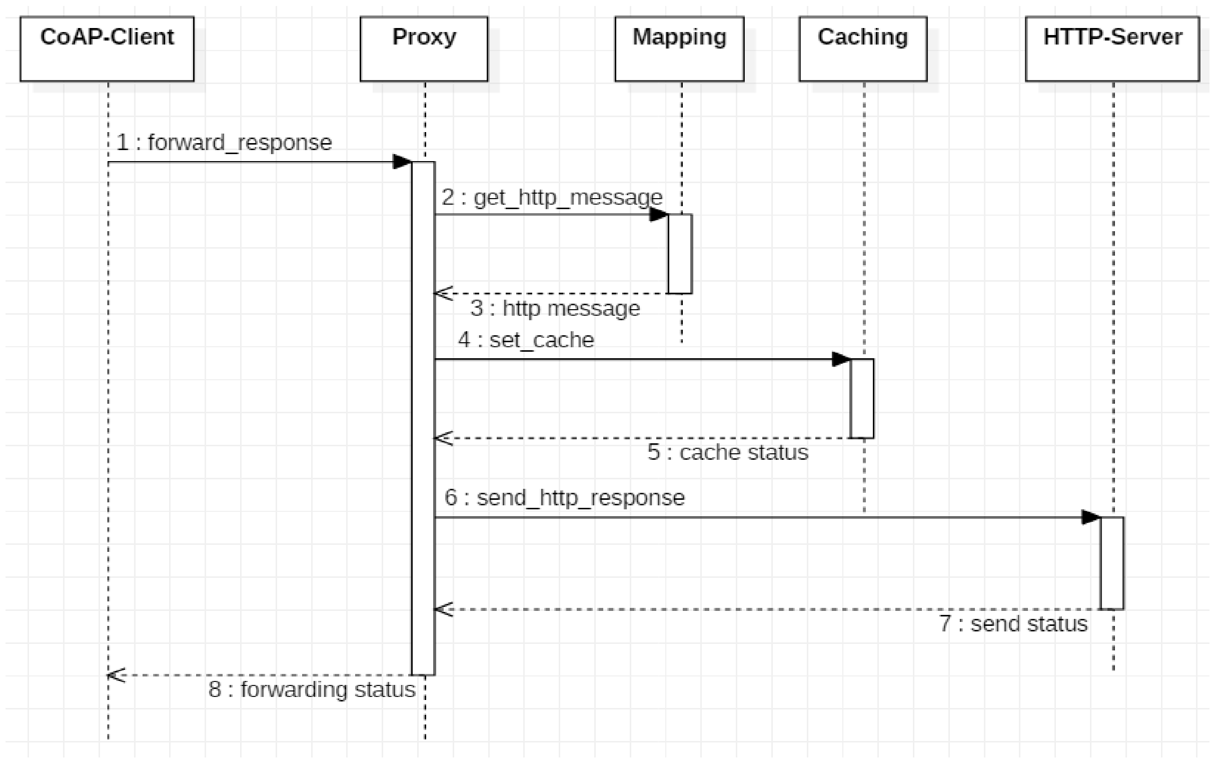

Figure 9 (Forwarding of a CoAP response in a proxy with caching enabled)

Similarly, the sequence shown in Figure 9 is almost identical to the sequence shown in Figure 6. Other than the already mention changes that occur by swapping the CoAP-Client and HTTP-Server around another difference can be seen in the way the cache is being accessed. Instead of returning the cached data from the Caching component the current response is being saved into the cache. Because of this there is no difference in the sequence if the data is already cached or not. The difference occurs only as an action in the Cache component. If no data is cached for the URI, the response is added into the Hashmap under the URI. On the other hand, if one already exists, the data will be replaced with the current response, and the expiration time set to the new time.

4 Testbed

For the experiments to be done and evaluated properly, it is important to have the correct tools. In the case of the experiments done, the tools used include software applications for emulating the environment and the HTTP client, CoAP server and proxy used. The task of each tool except of the proxy will be mentioned in the subchapter Tools. Because the design of the proxy has already been shown in the chapter design only the libraries used in the implementation will be mentioned. Other than the tools used, the experiment setup will also be shown.

4.1 Libraries

A library is a collection of files, programs, routines, scripts, or functions that can be referenced in the programming code (18). For the implementation of the proxy, three libraries were planned. These had the task of creating and managing a HTTP server, a CoAP client, and the cache. By using libraries instead of writing these software components a lot of time could be saved. Moreover, the performance and reliability of these libraries are better than if these components were implemented in the short span of the thesis.

The proxy was planned to be written in C++, so only libraries for C and C++ were considered. One library that was discovered early while researching that were to be used for most of the project was libcoap. It is a well-established C library that can create not only a CoAP client needed for the proxy, but also a CoAP server. It offers a lot of control while at the same time abstracting the exchange of CoAP messages. The search for an HTTP library was not that easy in comparison. There were different libraries based on all version of C and C++, each one of them having some positives and negatives. The library chosen at the end was pistache. It is a simple to use library that does not depend on other libraries.

While implementing a prototype many problems arose. Libcoap is written in pure C while pistache in pure C++11. Because of the object-oriented design in C++ the interface between the two libraries was not easy to implement. Another problem was in the way the libraries worked. Libcoap and pistache both required their own message loop to handle incoming and ongoing messages. Sending messages between these two components would then either require handling synchronizing of multiple threads if each message loop was in their own thread, or decreased performance if they both run on the same. Because of this problem, the choice to find a library that implements both CoAP and HTTP under one loop was made. After a little more research, the library decided on at the end was Mongoose.

Mongoose is a library created by cesanta. It offers HTTP and CoAP support. Plus, it is a simple to use library consisting only of two files that had to be included. In comparison to the other two mentioned libraries, Mongoose works on a lower level. Mongoose itself only handles the connection and the marshalling and unmarshalling of messages. Message handling of the CoAP and HTTP layer must be implemented from scratch. This brings both advantages and disadvantages. The disadvantage is that we lose some of the performance and time benefits gained by using a library, while the advantage is that because we have more control, the forwarding and caching of messages can be implemented easier. For that reason, the HTTP-Server and CoAP-Client component implemented the necessary functions, but only those necessary for this paper.

Although different caching libraries exist, they implement too much functionality irrelevant to the experiment. For that reason, it was decided that a simple caching mechanism was to be created. After some research, instead of using an existing library it was decided that a RAM cache, based on ideas derived from the paper (19) was to be implemented. Better implementations could exist, but no further research was done because all requirements were meet. By going this direction, a HashMap library was used that would handle management of the data. As a solution an opensource Hashmap library was used from Github.

4.2 Tools

Tools are necessary for the course of any experiment. For this experiment specifically, an important tool is an emulation environment in which the experiments could be done without the need of extra hardware. Other than just freeing us from extra hardware, using an emulation environment makes the experiment more replicable. The emulation tool used is the graphic network simulator-3 (GNS3) (20).Other emulation environments exist but GNS3 offers everything required, is easy to setup on windows, and has a big userbase making it easy to find solutions to problems.

Other than the emulation environment, the participants in the experiments are also important tools. These are the client and server. To be able to start the experiments a control component was also used. These have minimal requirements, because of that they have been specifically created for this paper. One big positive that arose by doing this is that a lot of evaluation components were able to be integrated into them directly for ease of measurement.

The client sends HTTP messages to the proxy for forwarding. Instead of writing it in C like the proxy it was written in the Golang language. Golang was chosen because it offers HTTP components as part of the language itself making the development process faster. The client makes it possible setting up to how many servers the requests should be sent. More than just sending requests, a client is also capable of measuring the time it takes for a response to arrive. The time is then saved in a two-dimensional array for further processing. In that array the first dimension is used to differentiate between the servers the request is being sent to, and the second to differentiate between the sent requests. Multiple clients can be used at once. It is necessary for them to be able to start at the same time so the strain on the proxy happens simultaneously. To solve this the controller is sending start messages to each client. For them to be able to receive those messages each client is also setup as an HTTP server waiting for a start request sent by the controller.

The server is a simple server written in C. All it does is responding to each received request. It does not contain any evaluation parts like the client does. The value the server responds with is not important and can be set to anything. In the case of this experiment the return value is “12C”. This was chosen to look like a response a temperature sensor could send.
First the server was written using the libcoap library, but after some problems arose where each subsequent received request increased the time required for a response to be sent back, it was decided for the server to also use mongoose.

The control component is responsible for starting the experiment. Other than that, the control component does not have any other function. Still it is necessary for the control component to exist and send a start experiment message to each of the client instead of starting them manually by hand. By doing it this way the delay between the start of the first and last client is kept at a minimum.

For these components to be used in the GNS3 environment different possibilities exist. First it was planned that each device runs in their own virtual machine. The virtual machine would be created beforehand and started multiple times with each of them running the correct software. This was not possible because only one instance of a virtual machine could run at a time. So, it was necessary to create a virtual machine for each component beforehand and then start them with GNS3 making the setup cumbersome. Another problem is that running a virtual machine is resource intensive. Thankfully GNS3 offered a better solution. Instead of running one virtual machine for each component only one virtual machine is running. On this virtual machine several docker containers are started where each container represents a component. This made the setup a lot easier while also solving the problem of using too much resources.

4.3 Experiment

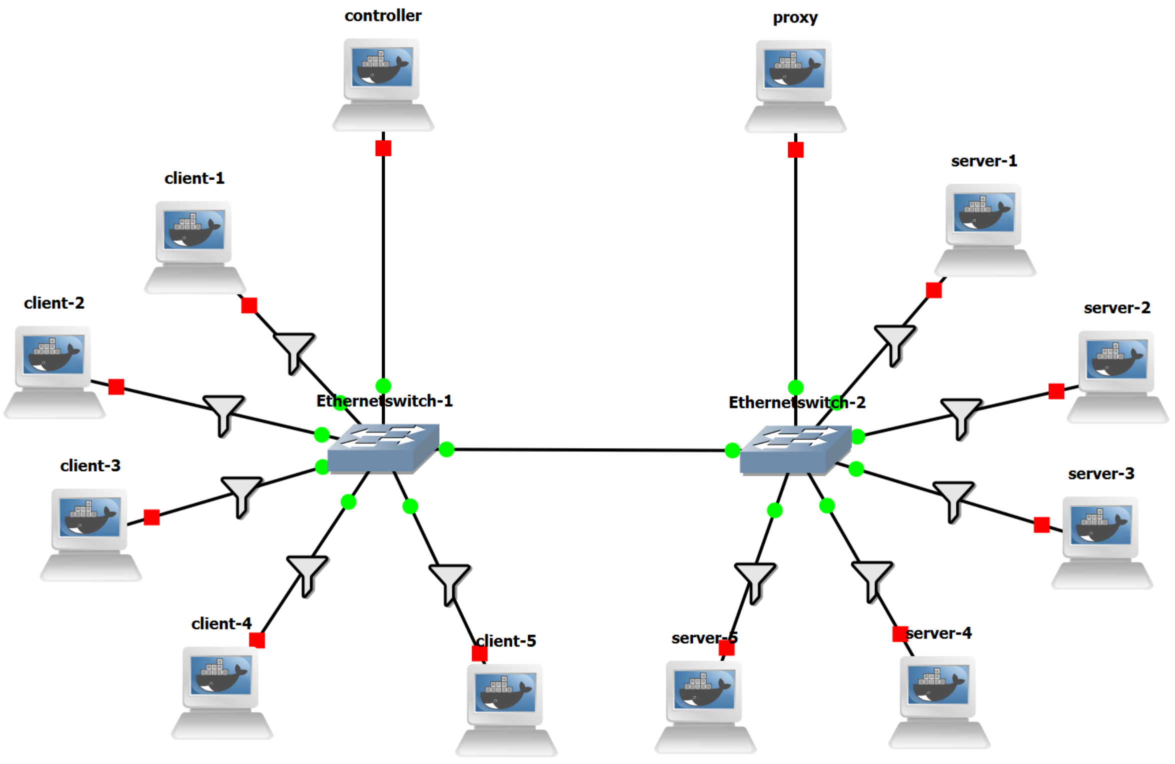

Figure 10 (Experiment setup in GNS3)

As already mentioned, the experiment setup occurred in the emulation environment GNS3. It contained one proxy, one controller, five clients, five servers and two ethernet switches. One switch had the controller and the five clients connected to it, while the other switch had the proxy and the servers connected to it. Both switches were then connected with each other so communication between all devices is possible. After connecting all devices, each device was given its own IP address. Shown in Figure 10 is the final setup. For the emulation to show real life properties, filters were implemented on the connections between devices and switches. The connection between the controller and the ethernet switch did not use any filter as the delay is not important because the controller is only used to start the experiment. A filter between the two switches was planned to represent the connection properties between the two networks, but GNS3 does not allow this. Moreover, the proxy does also not implement a filter because the client and the server implement filters that are meant to represent a connection to the proxy. In total two different filter with different values were used. The first one that will from now on be known as filter one has a delay of 10ms. This value does not mean anything specific and is only there to make it easier to see congestion in the time created by the switches.

The second filter that will be known as filter two has a delay of 100ms and a 1% packet loss rate. The 100ms has been taken from the AWS network latency map in (21) as the average latency between the USA and western Europe while the 1% packet loss is taken from (22) as an acceptable packet loss. Depending on which filter the clients implement, the first or the second, the distance to the proxy is being emulated.

With this setup multiple experiments were done with a different number of clients and servers participating. In total four combinations were used; one client and one server, one client and five servers, five clients and one server and five clients and five servers. Even though five is not a high number it is still possible to see how this influences the response time. Each of these combinations is then done with caching enabled and disabled making the total number of experiments equal to eight. In each experiment filter one is used in the connection from the clients to their switch while filter two is used in the connection between the servers and their switch. The reason why this was done is because if it were to be swapped each request would still need to travel the long distance with 100ms latency and 1% packet loss making the 10ms saved unnoticeable. To show this, in the evaluation chapter the response time difference between having caching enabled and disabled for both cases will be shown on the one client and one server combination.

On the start of an experiment, the control device sends a message to each client currently participating in the experiment. After receiving the message, the client starts sending its requests to the proxy for forwarding to the correct server. On each sent request a new connection is being established. If only one server is participating, each request is being forwarded to this server. On the other hand, if all five servers are used, the client sends a request to the first server, waits for the response to arrive, and after it has arrived it sends a request to the next server. This is done a total of 100 times. While sending a request, each client measures the time it takes for the corresponding response to arrive.

5 Evaluation

With the setup mentioned in the previous chapter experiments were done. The resulting data will be evaluated in this chapter. First, two samples of each client-server combination will be observed individually to interpret the data. One sample for when the cache is enabled, and one for when it is disabled. Data collected from the first experiment with one client and one server will be used as the base for comparison with the others. After that, the performance of all experiments will be compared in more detail with the help of the mean of multiple sample and a confidence interval. At the end of this chapter, after evaluating all data, a short look at the difference in response time between enabling and disabling caching will be shown when using filter one or filter two on the client.

5.1 Interpretation

Two samples from each client-server combination are going to be presented. The first sample is from when caching is disabled while the second one is from when it is enabled. This is done to see how the data acts to be better able to interpret how this affects the average response time. To display the data a line graph is used. On the graph the horizontal axis shows the current request sent in a row. The horizontal axis is not a time axis, therefore it is possible that the first clients sends its fifth requests before the fifth clients sends its first, even though this is highly unlikely. The vertical axis shows the required time a response takes to arrive back for the corresponding request on the horizontal axis. This is given in milliseconds. First, the data from the experiment consisting of one client and one server are shown. With this, data from the other experiments will be compared.

5.1.1 One Client and One Server

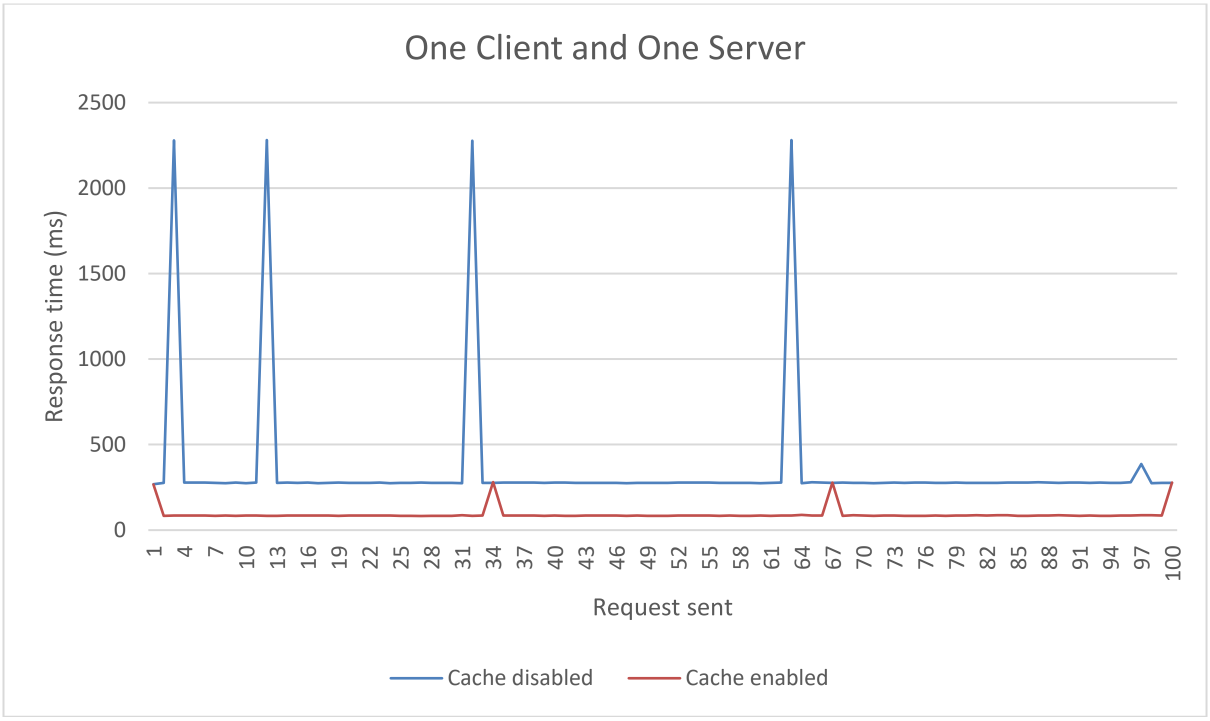

Figure 11 (One client and one servers sample comparison with cache disabled and enabled)

The average response time per request when caching is disabled for the sample shown in Figure 11 is around 357.93ms. This time would have been lower if not for requests that have been lost and had to be resent. These are represented as spikes. Each spike adds an extra two seconds to the response time. These two seconds come from the ACK timeout given by CoAP. It is the time the client must wait before another request is sent out if no acknowledgment message has arrived.
In the sample where caching is enabled the average response time is around 92.55ms. This makes the response time around 265.38ms faster. An important thing to mention in this sample no packet loss occurred when caching was enabled. By reducing the number of requests that must travel to the server, the number of requests that can be lost is also reduced. This is not always the case, for example on each cache refresh there is a possibility that a packet gets lost making the number of lost packets the same for both samples. But even if that would to happen, the average time would still be lower.

5.1.2 One Client and Five Servers

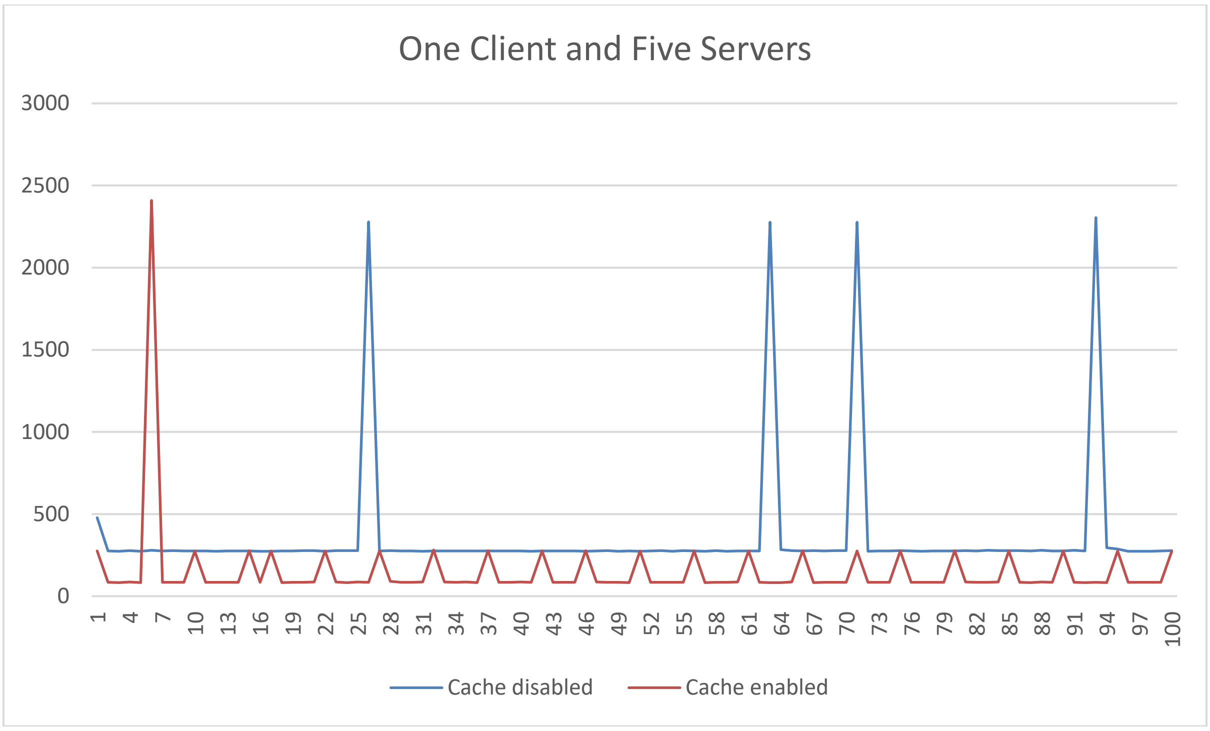

Figure 12 (One client and five servers sample comparison with cache disabled and enabled)

The sample data collected for one client and five servers are shown in Figure 12. Only the response time of one server is shown because of the similarities between them. Immediately a difference can be seen between these samples and the ones containing one client and one server. Because the client requests data from each server one after the other, the time between two consecutive requests to the same server is higher. This in return decreases the number of requests returned before the cache, when it is enabled, must be refreshed again. In this sample, the cache refreshes around every fifth request, whereby with one client and one server it happens every 34th request.

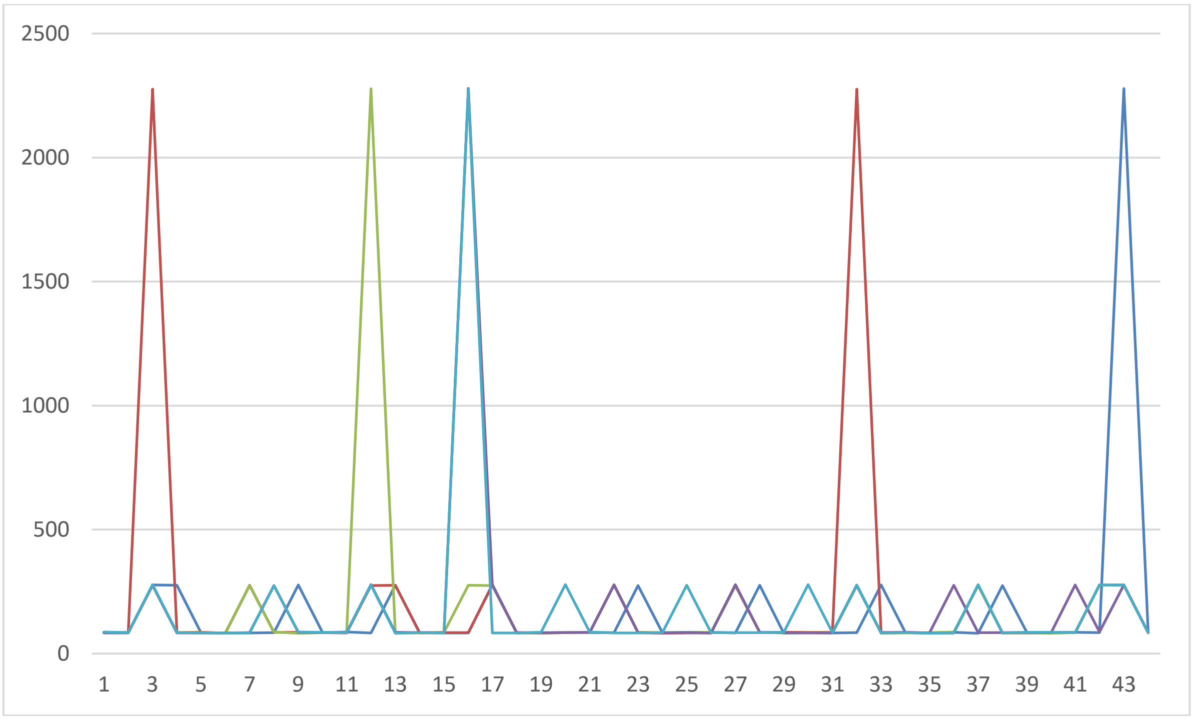

Figure 13 (One client and five servers, samples of all five servers with cache enabled in comparison)

Another noticeable detail can be seen when caching is enabled but is not visible in Figure 12. To be able to see it multiple samples for each server from the same experiment are placed in the graph shown in Figure 13 and zoomed in on a point where it is clearly visible. By placing samples of one client with all the five servers in one graph it becomes apparent that for some samples at some points the graph becomes flat from two cache refreshes in a row. This is caused by a packet loss for a message meant to a different server. Because the timeout is at two seconds, if a packet loss occurs for any of the servers the cached data for all expires. By increasing the cache time this can be avoided but only to an extent. Therefore, caching performance in case of multiple servers and one client depends not only on the number of servers but also the maximum age set.

5.1.3 Five Clients and One Server

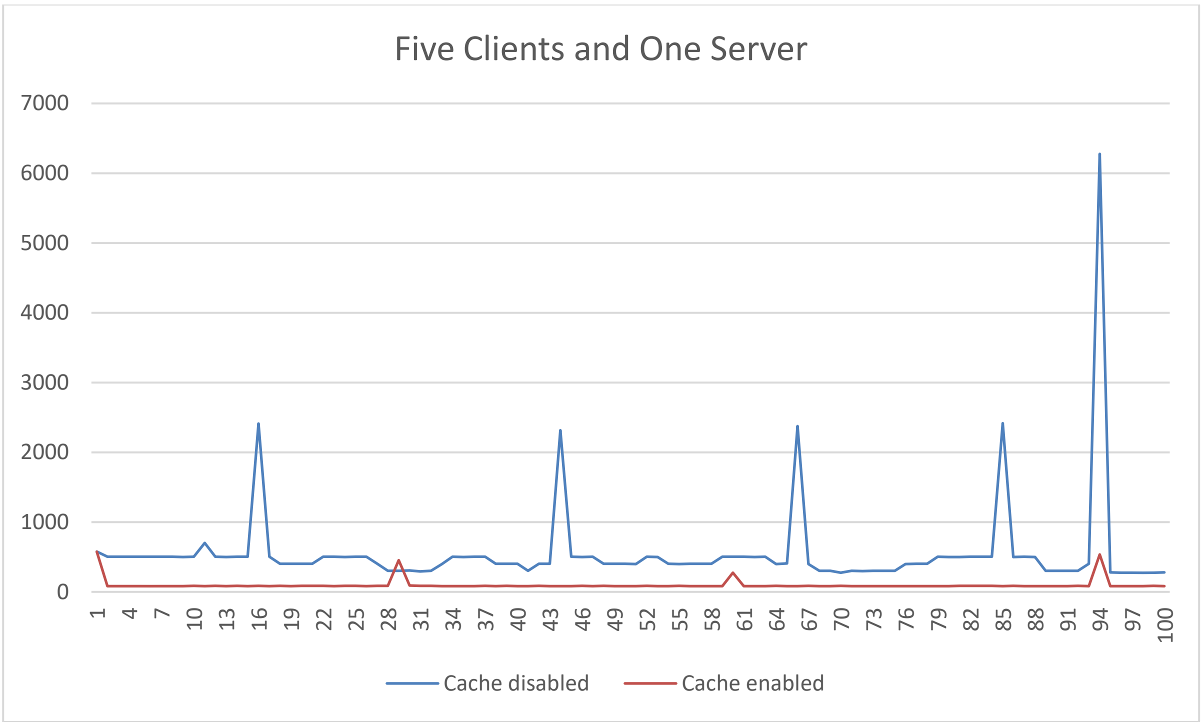

Figure 14 (Five clients and one server sample comparison with cache disabled and enabled)

Figure 14 shows the samples for five clients and one server. The data for when caching is enabled is like the data with one client and one server. During both, the cache had to be refreshed around four times. The real difference can be seen between the samples where caching is disabled. In the sample shown in Figure 14 a nonlinearity occurs with pseudo-random response times. The most likely reason for this is that because a server can only service one client at a time it delays the request of the others by the time it is required for the messages to be handled and sent back to the proxy.

5.1.4 Five Clients and Five Servers

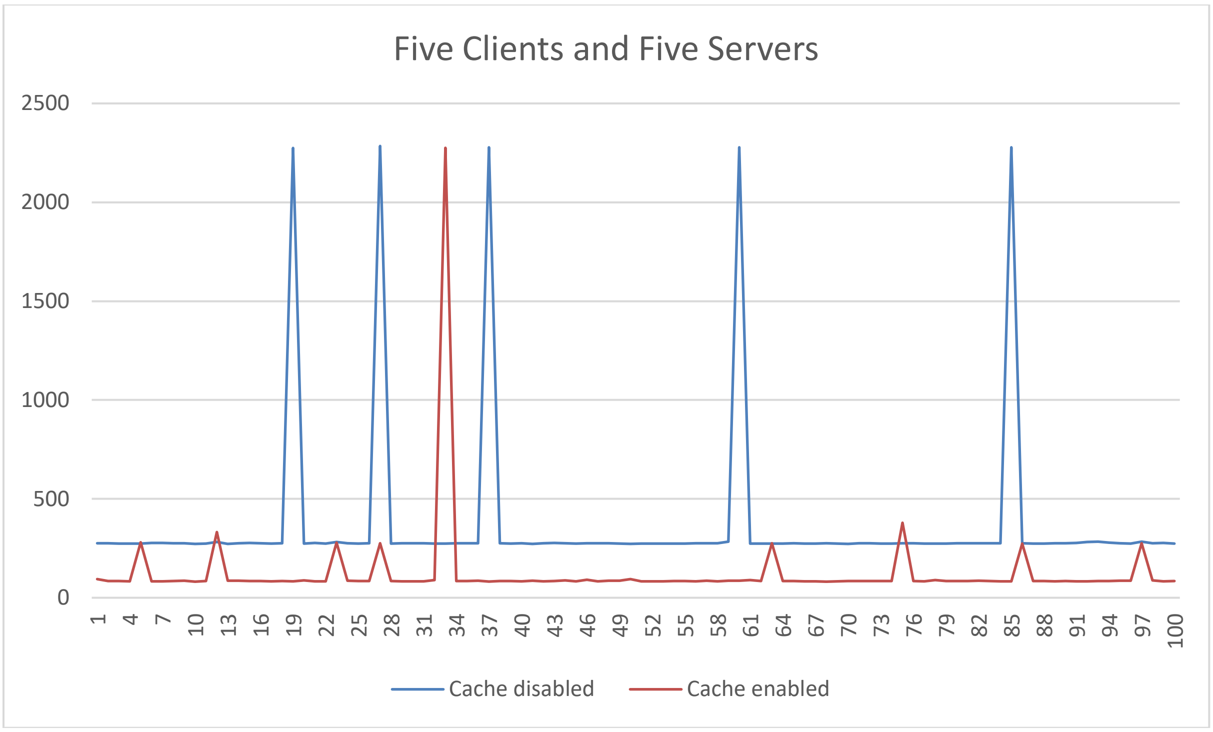

Figure 15 (Five clients and five servers sample comparison with cache disabled and enabled)

In case of the five clients and five servers, as shown in Figure 16, the data from the sample where caching was disabled was almost identical to the sample from one client and one server. In the sample where caching was enabled, there was a difference. A higher number of cache refreshes were required, but not as many as in the case of one client and five servers.

5.2 Comparison

To compare how caching affects the response times, multiple samples have been taken from each experiment and an average value were calculated for the communication between each client and server. This means that, for example, in the experiments containing one client and five servers, five different average values have been calculated. These values have then been statistically analyzed with a confidence interval. Figure 16 contains the evaluated data for when caching is disabled and Figure 17 for when it is enabled. To display the data, a bar graph is used where each bar shows the average value from all the averages collected. The error line at the top of the bar represents the deviation given by the calculation of the confidence interval. A confidence interval value of 95% has been chosen. To make it easier to differentiate between the experiments the naming format “cCsS” will be used where c is the number of clients and s the number of servers in the experiment.

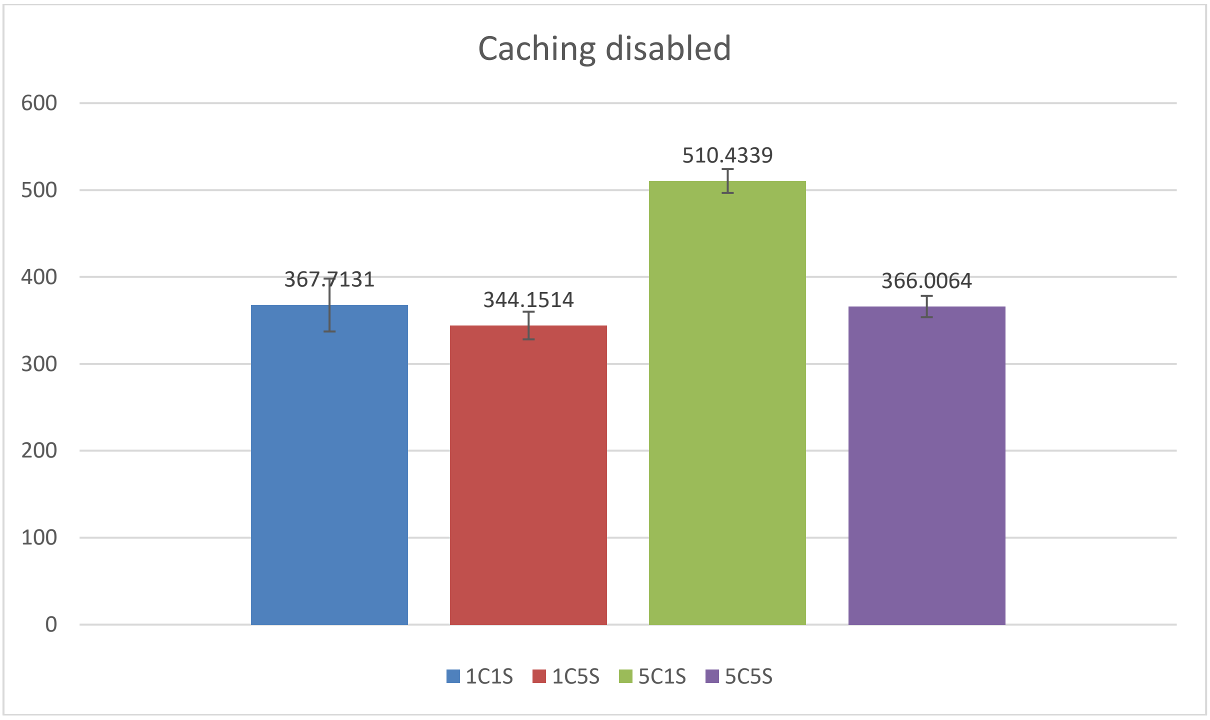

Figure 16 (Comparison of the mean response time when caching is disabled for all client server combination with a confidence interval)

When caching is disabled, the experiment with the highest response time of around 510.43ms, is the experiment 5C1S. This was to be expected as this is the only data acting unusual as seen in Figure 14. The other three experiments have better response times that are similar to each other, given their deviation, of around 350ms. Thus, it makes the difference between them and the experiment 5C1S around 160.4ms. Unexpectedly, the deviation in the experiments 1C1S are a lot higher than in the other three experiments. The deviations of the other three are around 15.88ms, 13.73ms and 12.24ms respectively. With a deviation of 30.4ms, the deviation of 1C1S is around double the value of the others. The reason for this hard to find out from the sample data and further experimentation into this is necessary if those findings are of interest.

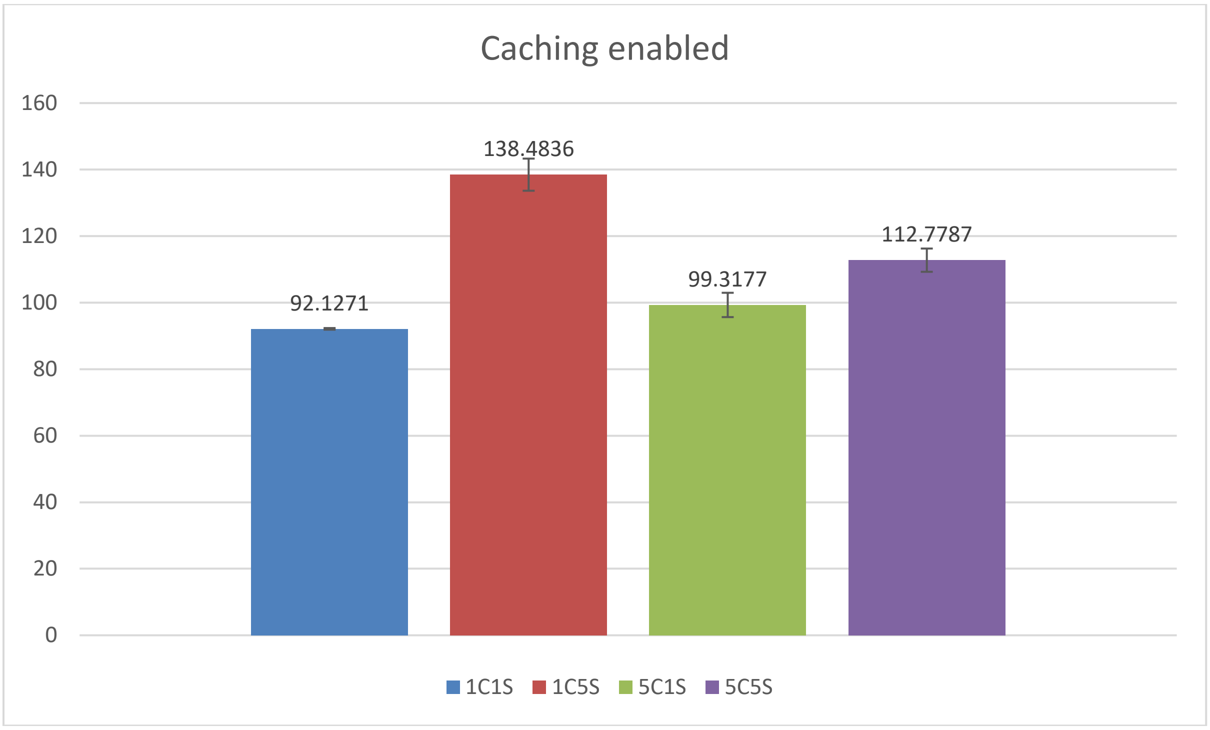

Figure 17 (Comparison of the mean response time when caching is enabled for all client server combination with a confidence interval)

The data from Figure 17 for when caching is enabled shows a different view. The highest response time here is the experiment 1C5S with a value of 138.5ms. This was also to be expected from the data seen in the samples. The second highest is 5C5S with a value of around 112.78, followed by 5C1S with 99.32ms and 1C1S with 92.13ms. Again, an unusual behavior can be seen in the deviation of the experiment 1C1S. Here it becomes smaller than the other three, with a value of 0.24ms, making it close to nonexistent.

From the data shown it is clearly visible that the best performance gained from using cache can be found in the experiments 5C1S, and the worst in the 1C5S. In the experiment 1C5S caching has only improved the response time by around 205.67ms. In comparison to this 5C1S has dropped its average value for around 411.11ms, twice the time of the other. The other two experiments also improved their response time by implementing a cache. 1C1S has dropped its for 275.58ms and 5C5S its for 253.22ms, both around the same. Not only has the average response time dropped, but the deviation as well.

5.3 Filter Difference

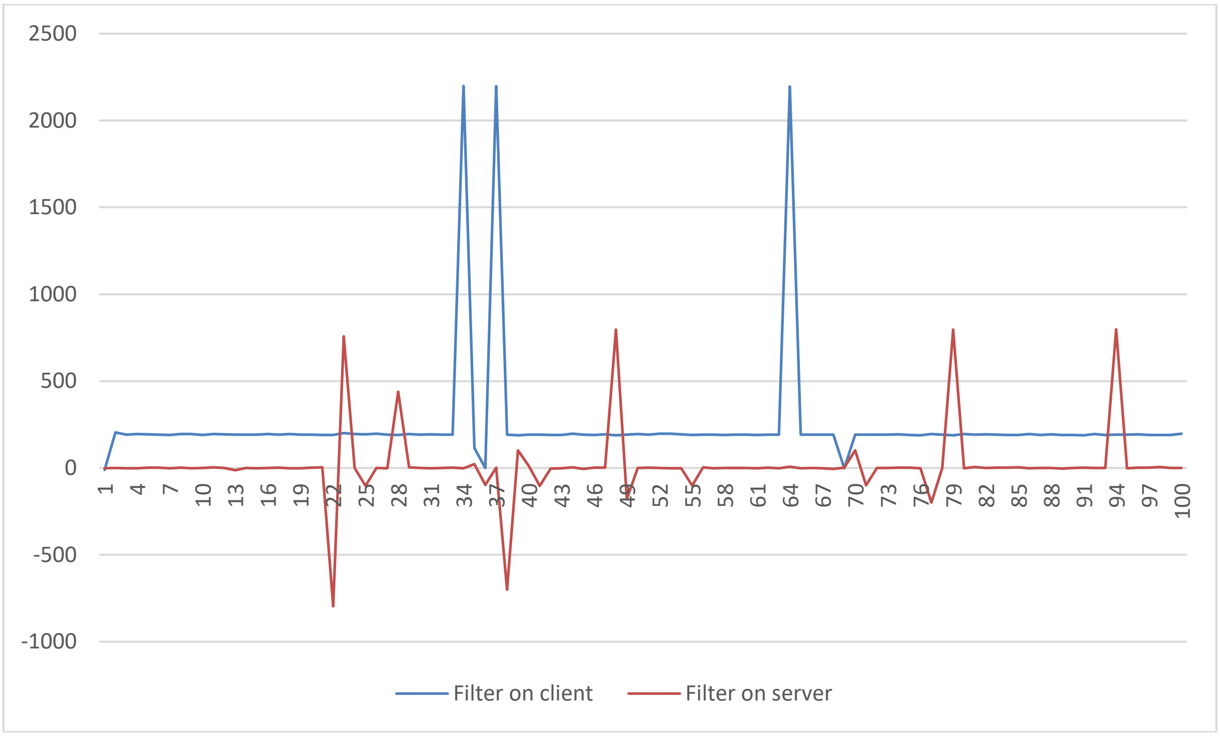

Figure 18 (One client and one server response time difference between caching enabled and disabled depending on the filter used on the client)

As already mentioned in chapter four, all experiments used in the emulation filter one on the client connection and filter two on the server connection to their switch. This was done because the performance gain from caching by putting the proxy closer to the server instead of the client is minimal. To confirm this, four experiments have been done on the one client one server combination. In the first two experiments, filter one was applied on the client connection with once disabling and once enabling the cache. On the other two, filter one was applied to the server. The difference is then taken between the response times of the first and second experiment and the third and fourth. This difference is shown in Figure 18. When filter one is put on the client the average difference is around 247ms. Compared to this, if filter one is on the server, the average difference is around 14ms. As can be seen, not much of performance is gained by putting the proxy close to the server, so this was not further done.

6 Conclusion

This paper aimed to see if caching would improve the response time of a HTTP-CoAP proxy in different use cases. After conducting the experiments and evaluating the data it is shown that in all cases caching does improve the response time.
The scenario in which caching makes the biggest difference is in the case where multiple clients talk to one server. Here the difference between the average response with caching and without is a lot higher than in the other experiments.

The worst-case scenario would be where one client talks to multiple servers. Even though the performance gain is visible and like the other experiments, by increasing the number of servers and/or lowering the max age the performance drops. In such cases it may be better if possible, to change the roles of the client and servers turning it into multiple clients sending data to one server where the data arrives to the server only when necessary therefore removing the need for cache.
In the case where one client talks to one server using a proxy cache does not only improves the performance, but by putting the cache in the proxy instead of the client more resources are available to the client.

Other future research could be built based on the findings of this paper by taking a deeper look on how caching is influenced by increasing or decreasing the number of clients or servers. Furthermore, these experiments can be done with different metrics showing other aspects that may be interesting.

Sources

1. Internet World Stats. *World Internet Usage And Population Statistics.* [Online] [Cited: February 18, 2019.] https://www.internetworldstats.com/stats.htm.
2. W3C. *Help and FAQ.* [Online] [Cited: July 20, 2010.] https://www.w3.org/Help/#webinternet.
3. *Hypertext Transfer Protocol -- HTTP/1.1.* s.l. : Internet Engineering Task Force, June 1999.
4. Fielding, Roy Thomas. Representational State Transfer (REST). *gbiv.* [Online] 2000. [Cited: July 28, 2019.] https://roy.gbiv.com/pubs/dissertation/rest_arch_style.htm.
5. Booth, David, Haas, Hugo and McCabe, Hugo. W3C. *Web Services Architecture.* [Online] W3C Working Group Note, February 11, 2004. [Cited: July 28, 2019.] https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.
6. (IDC), International Data Corporation. IDC Forecasts Worldwide Spending on the Internet of Things to Reach $745 Billion in 2019, Led by the Manufacturing, Consumer, Transportation, and Utilities Sectors. *idc.* [Online] International Data Corporation (IDC), January 3, 2019. [Cited: October 25, 2019.] https://www.idc.com/getdoc.jsp?containerId=prUS44596319.
7. Shelby, Z. *The Constrained Application Protocol (CoAP).* s.l. : Internet Engineering Task Force, June 2014. 2070-1721.
8. *An Educational HTTP Proxy Server.* Sysel, Martin and Doležal, Ondřej. s.l. : Procedia Engineering, December 2014, Vol. 69.
9. *HTTP-CoAP cross protocol proxy: an implementation viewpoint.* Castellani, Angelo P., Fossati, Thomas and Salvatore, Loreto. Las Vegas, NV, USA : IEEE, 13 January 2014. 978-1-4673-2433-5.
10. *Connecting the web with the web of things: lessons learned from implementing a CoAP-HTTP proxy.* Lerche, Christian, et al. s.l. : IEEE, 2014. 978-1-4673-2433-5.
11. *Performance of Caching in a Layered CoAP Proxy.* Mišić, Vojislav B. and Mišić, Jelena. s.l. : IEEE, 2018. 978-1-5386-2070-0.
12. Tanenbaum, Andrew S. and van Steen, Maarten. *Distributed Systems, 3rd edition.* s.l. : CreateSpace Independent Publishing Platform, February 2017. 978-1543057386.
13. Luotonen, Ari and Altis, Kevin. World-Wide Web Proxies. *Virginia Tech.* [Online] April 1994. [Cited: July 28, 2019.] http://courses.cs.vt.edu/~cs4244/spring.09/documents/Proxies.pdf.
14. Postel, J. *User Datagram Protocol.* United States : RFC Editor, 1980.
15. Castellani, A. and Loreto, S. *Guidelines for Mapping Implementations: HTTP to the Constrained Application Protocol (CoAP).* s.l. : Internet Engineering Task Force, February 2017. 2070-1721 .
16. HTTP caching. *Mozzila.* [Online] [Cited: Jun 20, 2019.] https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching.
17. Sommerville, Ian. *Software Engineering, 9th Edition.* s.l. : Pearson, 2011. 9780137035151.
18. Software library. *Computer Hope.* [Online] Computer Hope, 2019. [Cited: July 28, 2019.] https://www.computerhope.com/jargon/s/softlibr.htm.
19. *The RAM-based web proxy servers.* Lai, Ka-Chon and Wong, Kin-Yeung. 10, s.l. : WSEAS Transactions on Communications, October 2012, Vol. 11. 2224-2864.
20. *gns3.* [Online] Galaxy Technologies, LLC., 2019. [Cited: Jun 25, 2019.] https://www.gns3.com/.
21. AWS Network Latency Map. *Datapath.* [Online] Datapath.io GmbH, 2017. [Cited: July 25, 2019.] https://datapath.io/resources/blog/aws-network-latency-map/.
22. Pinger. *ICTP Science Dissemination Unit.* [Online] ICTP-SDU, March 2005. [Cited: May 31, 2015.] https://web.archive.org/web/20150531052949/http://sdu.ictp.it:80/pinger/pinger.html.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _________________ ______________________________