UiO : Department of Informatics
University of Oslo

Improving the performance of Web Services in Disconnected, Intermittent and Limited Environments

Joakim Johanson Lindquister

Master's Thesis Spring 2016

Abstract

Using Commercial off-the-shelf (COTS) software over networks that are Disconnected, Intermittent and Limited (DIL) may not perform satisfactorily, or can even break down entirely due to network disruptions. Frequent network interruptions for both shorter and longer periods, as well as long delays, low data and high packet error rates characterizes DIL networks. In this thesis, we design and implement a prototype proxy to improve the performance of Web services in DIL environments. The main idea of our design is to deploy a pair of proxies to facilitate HTTP communication between Web service applications. As an optimization technique, we evaluate the usage of alternative transport protocols to carry information across these types of networks.

By introducing a proxy pair, we were able to break the end-to-end dependency between two applications communicating in a DIL network, and thus achieve higher reliability. The proxy was designed to support different protocols for inter-proxy communication. We implement the proxy to support the Hypertext Transfer Protocol (HTTP), the Advanced Message Queuing Protocol (AMQP) and the Constrained Application Protocol (CoAP). Evaluations show that in most cases using HTTP/TCP yielded the lowest Round-Trip Time (RTT). However, with small message payloads and in networks with low data rates, CoAP had a lower RTT and network footprint than HTTP/TCP.

Acknowledgement

This master thesis was written at the Department of Informatics at the Faculty of Mathematics and Natural Sciences, University at the University of Oslo in 2015/2016. It was written in cooperation with Norwegian Defence Research Establishment (FFI), which provided the thesis topic and supervision. I would like to thank my supervisors Frank Johnsen and Trude Hafsøe Bloebaum for providing guidance and helpful feedback throughout my thesis. Their many bits of advice are much appreciated. Additionally, I would like to thank Jostein Sander for guiding me in the communication laboratory at FFI.

I would also like to thank Øyvind Tangen, Svein Petter Gjøby and Ole Kristian Rosvold for reading my thesis and providing helpful feedback. Furthermore, I would like to thank my family for always supporting me.

Finally, I would like to thank all the amazing students I've met during my five years at the University of Oslo. Thanks for all the good times, whether it was planning the next dagen@ifi, wasting countless hours talking about nothing at Assembler or drinking beer at Escape.

Contents

Chapter 1

Introduction

Today the Internet connects millions of users from all over the world. It plays an important role for both businesses and people in their everyday life by enabling the possibility to access and exchange information. The communication infrastructure provides fast and stable access to the Internet. This infrastructure might not be present in all use-cases requiring the exchange of information over the Internet. Consider for example a nature disaster damaging the communication infrastructure, which limits the quality of connections and available data rate. In such a scenario the exchange of information is critical for public health and security services. Another consideration is the development of the Internet of Things (IoT), where more and more devices are becoming connected to the Internet. Typical IoT devices are sensors with limited energy and wirelessly connected to the Internet. High packet loss rates, low data rates, and instability may characterize such wireless networks. They are often referred to as Low-Power and Lossy Networks (LLNs). In this thesis, we look into improving the performance of Web services operating in these types of conditions, with a focus on military application.

Military units may operate in areas where conditions like terrain, obstacles, and radio interference make communication difficult. They may operate far from existing communication infrastructure and rely only on wireless communication. Such communication is often characterized by unreliable connections with low data rate, long delays, and high error rates. In a military scenario, it is necessary for units to exchange information seamlessly across different types of communication systems. This ranges from remote combat units at the tactical level, to commanding officers at the operational level in a static headquarters packed with computer support. To the North Atlantic Treaty Organization (NATO), this concept is referred to as Network Enabled Capability (NEC). In a feasibility study, NATO identified the Service Oriented Architecture (SOA) paradigm and the Web Service technology as key enablers for information exchange in NATO [1].

Web service technology is well tested and in widespread use in civil

applications where the network is stable and the data rate is abundant. However, military networks may suffer from limited networks, which can leave Web services built for civilian use unusable. How to overcome these challenges are investigated in this thesis. The primary approach looks into how using alternative network transport protocols may increase speed and reliability.

1.1 Background and Motivation

NATO is a military alliance consisting of 28 member countries [2] and which primary goal is to protect the freedom and security of its members through political and military means. In joint military operations, the relatively large number of member countries can be a challenge when setting up machine-to-machine information exchange. Differences in communication systems and equipment contribute to making the integration of such systems more difficult. To address this issue, NATO has chosen the SOA concept, which when built using open standards facilitates interoperability [1].

1.1.1 Service Oriented Architecture

SOA is an architectural pattern where application components provide services to other components over a network. SOA builds on concepts such as object-orientation and distributed computing and aims to get a loose coupling between clients and services. In their reference model for SOA, the Organization for the Advancement of Structured Information Standards (OASIS) defines SOA as [3]:

Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.

In SOA, business processes are divided into smaller parts of business logic, referred to as *services*. A service can be business related, such as a patient register service, or an infrastructure service used by other services and not by a user application. OASIS defines a service as [3]:

A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.

Services are provided by *service providers* and are consumed by *service consumers* as illustrated in fig. 1.1. The service provider

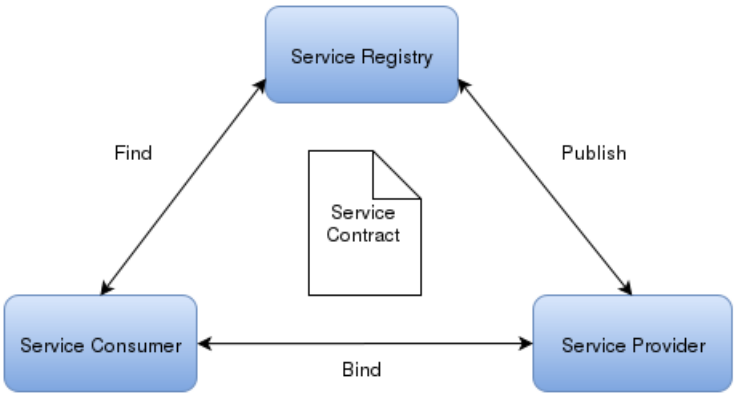


Figure 1.1: The three roles in SOA

is responsible for creating a service description, making the service available to others and implementing the service according to the service description. Services are provided to service consumers through a form of *service discovery*. This can be a static configuration, or more dynamic with a central *service registry*, where service providers publish service descriptions. Service consumers find the services they need by contacting the service registry. The communication between services occurs through the exchange of standardized messages.

Following the SOA principles dictate a very loose coupling between services and the consumers of those. This allows software systems to be more flexible, as new components can be integrated with minimal impact on the existing system. Another aspect of loose coupling is concerning time, which enables services and its consumers not to be available in the same instance of time. This allows asynchronous communication. Loose coupling with regards to location permits the location of a service to be changed without needing to reprogram, reconfigure, or restart the service consumers. This is possible through the usage of runtime service discovery, which is the dynamic retrieval of the new location of the service.

Furthermore, SOA enables service implementation neutrality. The implementation of a service is completely separated from the service description. This allows re-implementation and alteration of a service without affecting the service consumers. Thus, this can attribute to keep development costs low and avoiding proprietary solutions and vendor lock-in. Another benefit with SOA is re-usability by dividing common business processes into services, which may help cost reduction and avoids duplication.

SOA is only a pattern, and a range of technologies can realize the concepts. The most common approach used is the World Wide Web Consortium (W3C) Web service family of standards, which use the SOAP messaging protocol. To achieve interoperability between systems from

different nations and vendors, NATO has chosen this technology in order to realize the SOA principles [4]. This allows member countries to implement their technology as long as they adhere to the standards. The Web service technology is discussed in detail in section 2.2.1. Another method to realize the SOA principles is Representational State Transfer (REST), an architecture style which has gained a lot of traction in the civilian industry. We discuss REST further in section 2.2.2.

Both REST and W3C Web services are in widespread use both in the civilian and military world. However, employing Web service solutions directly for military purposes may not be so straightforward. These technologies were not specifically designed to handle the network conditions found in certain military networks. In the following sections, we present an overview of military networks, discuss characteristics of them and the possible challenges of using Web services in them.

1.1.2 Military Networks

Military networks are complex and consist of many different heterogeneous network technologies. We can group them into layers, which have different characteristics as can been seen in fig. 1.2. At the highest level, there is fixed infrastructure and a relatively static number of services. At the lower levels, there are fewer services, but the units operating at this level are much more dynamic. The lower levels are called tactical networks and are discussed in the next paragraph.

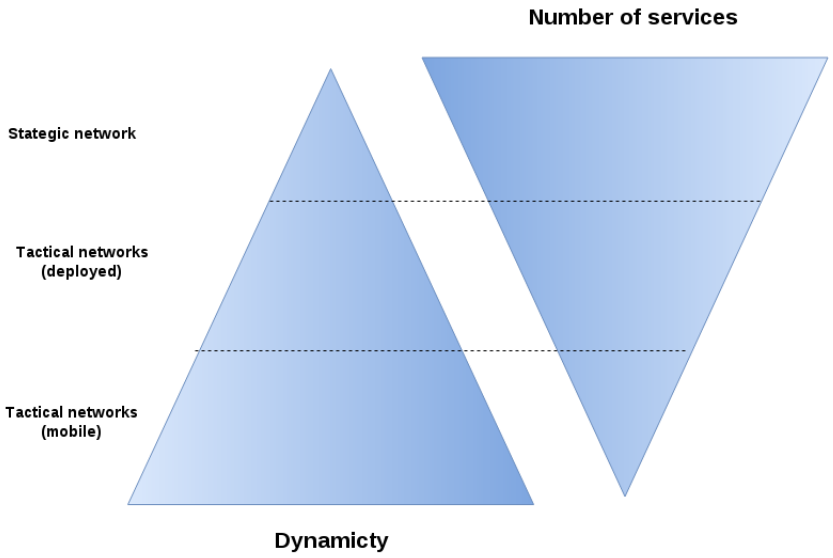


Figure 1.2: Complexity of military networks (from [5])

Tactical Networks

Tactical networks are characterized by that the units are deployed to operate on a battlefield. We distinguish between deployed and mobile tactical networks, where deployed may use existing communication infrastructure. Mobile tactical networks have no existing communication infrastructure and therefore experience the largest communication challenges.

In tactical networks, military units use tactical communication equipment, which includes technologies like VHF, UHF, HF, tactical broadband and satellites [6]. Examples of such units are mobile units like vehicles, foot soldiers and field headquarters. NATO studies[7] have identified tactical networks to have the following characteristics:

Disadvantaged grids are characterized by low bandwidth, variable throughput, unreliable connectivity, and energy constraints imposed by the wireless communications grid that link the nodes.

These types of networks are often called disadvantaged grids or Disconnected, Intermittent and Limited (DIL) environments, which is the term we use in this thesis.

1.1.3 Disconnected, Intermittent and Limited Networks

To improve the performance of Web services in tactical networks, it is important to understand the limitations of DIL networks. The DIL concept refers to three characteristics of a limited network: *Disconnected, Intermittent* and *Limited*.

Disconnected Military units that participate in a tactical network may be highly mobile and may disconnect from a network either voluntarily or not. Unplanned loss of connectivity can be due to various reasons, such as loss of signal or equipment malfunction. The disconnected term refers to that units may be disconnected for a long time, possibly for multiple hours or even days.

Intermittent Units operating in a DIL environment may lose connection temporarily before reconnecting again. The duration can range from milliseconds to seconds. As an example, consider a military vehicle that is driving on a countryside road. It may temporarily lose connection due to the signal being obstructed by trees beside the road, driving into tunnels or by having a bad radio signal.

Limited Limited refers to various ways a network can be constrained. The data rate may be low, and the network delay may be high, and the Packet Error Rate (PER) may be high. The term data rate refers

to the speed that data that can be transmitted over a network. Delay means the time it takes for data to travel from machine to machine. The packet error rate refers to the percent of packets being sent incorrectly due to the data being erroneous altered in transmission. A packet is considered as incorrect if at least one bit error in the data occurs. Section 2.1.3 elaborate on these metrics.

In addition to network limitations, other factors may also limit communication for military units. As an example, consider a military foot patrol that is operating out in the field. To communicate critical information with other units they use radios. The radio communication equipment is powered by batteries, which the soldiers have to carry with them. Running applications and the sending and receiving of data can consume a considerable amount of power. Thus, the battery could be a scarce resource for the units operating in a DIL environment. This is similar to the constraint imposed on IoT devices.

1.2 Problem Statement

The Web service technology enables interoperability between systems, but also increases the information overhead, requiring higher data rate demands. Most of the Web service solutions used today are aimed for civilian purposes and do not necessarily perform well in military environments. In contrast to civilian networks where the data rate is abundant, mobile tactical networks may suffer from high error rates and low data rate. Adapting Web service solutions meant for nonmilitary networks directly for military purposes may not be possible. Therefore, Web services need to be adapted to better handle unreliable and limited networks. However, it can be very expensive to alter existing Web service technology and incorporate proprietary solutions.

1.3 A Suggested Approach

The NATO research group titled "SOA Challenges for Real-Time and Disadvantaged Grids" (IST-090) has previously investigated which improvements that could be made to enable the usage of Commercial off-the-shelf (COTS) applications in DIL networks. COTS is a term used to describe the purchase of standard manufactured systems rather than custom made. The research group pointed out the desire to optimize Web services, but without the need of incorporating proprietary and ad-hoc solutions [6]. IST-090 did not find a magic bullet that would solve all problems with using Web services in DIL networks but identified some factors that would offer measurable improvements. The most notable findings were:

- Foundation on open-standards.

- Ease of management and configuration.
- Transparency to the user.
- The Web services should be optimized without the need to incorporate proprietary, ad hoc solutions that ensure tighter coupling between providers and consumers of services.

The last bullet point refers to the issue of when we have identified optimization techniques, where do we apply them? One approach could be to modify the Web service application itself. However, this would mean that every application deployed in a tactical network would require modification. The alteration would require a lot of resources and severely limit the flexibility of using standardized Web services.

IST-090 recommends another approach, applying optimizations in proxies without altering the Web services themselves [6]. A proxy is a node deployed somewhere in a network, which applications can tunnel their network traffic through. With this approach, the only thing required to do is to configure the applications to send and receive data through the proxies. The proxies will then handle the optimization for tactical networks. Figure 1.3 illustrates a setup like this, where clients can invoke Web services through a proxy pair over a DIL network. By placing the optimization in proxies, the Web services themselves can remain unchanged.

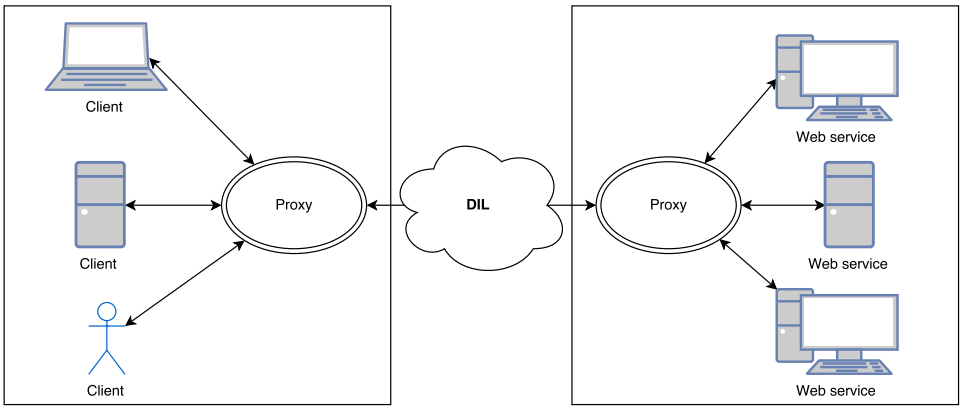


Figure 1.3: Proposed proxy solution

This approach is identified by IST-090 and is explored in this thesis. Based on this recommendation we create a proxy with the aim of facilitating Web service usage in DIL networks.

1.4 Premises of the Thesis

In this section, we define the premises for the thesis and the proxy being developed as a part of it. As we have previously discussed, W3C Web

services are in widespread use in NATO. Also, the REST architectural style has been identified as a technology of interest to NATO. As we discuss later in section 2.2, Hypertext Transfer Protocol (HTTP) is the by far most common transport protocol used by these types of services. The first and second premise are therefore that the proxy must be able to support both REST and W3C Web services deployed in a DIL network.

Next, to optimize Web services in DIL environments, the applications themselves should not be required to be customized. All optimization should be placed in proxies. By doing this, we retain the interoperability with standards-based COTS solutions. The fourth and final premise are that the proxy must work with standard security mechanisms. In our case, this means that any messages sent through the proxies must be exactly the same at the receiver as it would have been without the proxies. The reason for this is due to both the header fields and the body of the message can be part of security mechanisms, such as digital signatures and the presence of authentication header fields.

To summarize, the premises of this thesis are that the proxy solution must:

1. Support HTTP RESTful and W3C Web services.
2. Work in DIL networks.
3. Be interoperable with standards-based COTS solutions.
4. Work with security mechanisms.

1.5 Scope and Limitations

The goal of this thesis is to investigate optimization techniques for Web services in DIL environments. We limit it to techniques that can be applied to the application or the transport layer of the Internet protocol suite (see table 2.1). The reason for this is that NATO has previously decided that all data communication in NATO should occur with IP packets [1]. We, therefore, limit our optimization possibilities to the mentioned layers.

We mainly focus on the performance of Web services, yet security is of paramount importance in military networks. Hence, any optimization techniques applied should be possible to use together with common security mechanisms. Another aspect is that applications that are to be used in military networks need to be approved by security authorities. If the application is too complex, e.g. it has a very large code base or use a lot of external frameworks, the approval process can be very lengthy. It is therefore an important consideration to make the proxy as relatively simple as possible.

1.6 Research Methodology

Research is the systematic investigation of how to find answers to a particular problem. It is broadly classified into *Basic Research* and *Applied Research* [8]. Basic research also called fundamental or pure research, is research on basic principles and reasons for the occurrence of a particular event or process or phenomenon. It does not necessarily have any practical application. Applied research, on the other hand, is concerned about solving problems employing well known and accepted theories and principles. In this thesis, we set out to solve the actual real-world problem of optimizing Web services. Thus, we perform applied research. To address the problem we need a systematic approach of how to conduct the research. This is referred to as *research methodology* and says something about how the research is to be carried out.

In this thesis we are performing research in the area of Computer Science, a scientific discipline defined as [9]:

> *The systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application.*

Denning et al. have identified three main processes for the computer science discipline, *theory, abstraction* and *design* [9]. *Theory* derives from the mathematics discipline and applies to the areas of computer science that rely on underlying mathematics. Examples of this are the computer science areas of algorithms and data structures that involve complexity and graph theory. The next process, *abstraction*, deals with modeling potential implementations. The *design* process is the process of specifying a problem, transforming the problem statement into a design specification, and repeatedly inventing and investigating alternative solutions until a reliable, maintainable, documented, and tested design is achieved.

The research methodology used in this thesis is based on the design process. The four steps and the efforts undertaken in them are summarized here:

Specify the problem The primary focus of this thesis is how to improve the performance of Web services in DIL networks. We formulate a problem statement in section 1.2 and propose a possible solution in section 1.3. Moreover, we present the technical background in chapter 2 and previous related work in chapter 3.

Derive a design specification based on the requirements Based on the premises, scope of the thesis, studies of the technological background and related work, we derive a set of requirements and specifications in chapter 4.

Design and implement the system After we identify the requirements for the optimization techniques, we design and implement them. Chapter 5 elaborate this step.

Evaluate the system Finally, the solution is assessed through a series of tests. The purpose of this is to evaluate if we are in fact able to solve the problem we set out to solve. We cover the testing and evaluation in chapter 6 and draw a conclusion in chapter 7.

1.7 Contribution

The outcome of this thesis is a recommendation regarding which optimizations techniques can be used in DIL networks to increase the performance of Web services. As a part of this work, we implement a prototype DIL proxy.

1.8 Outline

The remainder of this thesis is organized as follows:

Chapter 2 presents the technical background for this thesis. We introduce computer networks in general before we dive into different communication paradigms and protocols. Then, in chapter 3, we discuss previous work done in the area. In chapter 4, we derive a specification for the proxy, before we in chapter 5 present the design and implementation details. Next, we show the testing of the proxy and how the proxy fulfilled the premises and requirements in chapter 6. Finally, in chapter 7, we summarize the discussion and provide reflections on possible future work within this field.

Chapter 2

Technical Background

In this chapter, we present the technical background of the central concepts and protocols this thesis is based. We first give an introduction to computer networks in general and how they are organized. Next, we introduce a set of network metrics used to characterize different types of DIL networks in this thesis. Then we look into two very common communication patterns. Next, we present the W3C Web service technology commonly used for exchanging data in military systems. We also introduce the REST style of services. Finally, we look into a number of protocols that we can replace HTTP/TCP with to increase the performance of Web services.

2.1 Computer Networks

A computer in a network is often referred to as a *node*. These nodes can be interconnected and form large computer networks. The most well-known network is the Internet, which is a large network of networks facilitating communication between nodes all over the world. The Internet is linked together by nodes using a set of protocols called the Internet Protocol Suite [10]. The functionality of the protocol suite is organized into four abstraction layers outlined in the following paragraphs.

2.1.1 Network Layers

The Internet Protocol Suite is organized into four layers, each one built upon the one below it as shown in table 2.1.

Application Layer
Transport Layer
Internet Layer
Link layer

Table 2.1: The layers of the Internet Protocol Suite

Link Layer

The lowest layer of the protocol suite is the link layer, where the link is the physical component used to interconnect two adjacent nodes in a network. Ethernet is an example of a link layer protocol facilitating the transfer of data between two physically connected nodes.

Internet Layer

Where the link layer is only concerned with moving data over a wire to an adjacent node, the Internet layer is concerned with how to deliver data all the way from a source to a destination, possible passing through multiple nodes on its way. It does not guarantee delivery of data since data can be lost on the way to the destination but provide a best-effort approach. Guaranteed delivery is usually handled by the higher network layers of the Internet Protocol Suite.

The Internet Protocol (IP) is the protocol that enables the transfer of messages between two nodes in a network. Messages between two nodes are sent as IP packets and are routed through possibly multiple other nodes before it reaches its destination. This routing function is fundamental for the Internet, as it allows nodes to communicate without knowing the exact network path to each other.

To provide a common transport mechanism for all types of transmissions links, NATO has decided that data communication in NATO systems should be based on IP [1].

Transport Layer

In the Internet protocol suite model, the transport layer provides end-to-end communication services to applications. It builds on top of the network layer and takes responsibility for sending data all the way from a process on a source computer to a process on the destination computer. The by far most used transport protocol is the Transmission Control Protocol (TCP), which provides reliable transport of data to applications. With reliable transport, we mean that if data in a transmission is lost or received in the wrong order, this is handled by the transport protocol. This provides a valuable abstraction for applications so that they do not need to deal with these issues themselves.

Application Layer

The top layer of the Internet Protocol Suite is the application layer. Its role is to serve communication services to applications. When we talk about application layer protocols, we usually talk about protocols that applications use to communicate with other applications. Application layer protocols use the communication services the transport layer provides. Examples of application layer protocols are HTTP and the

File Transfer Protocol (FTP), which both rely on TCP as the underlying transport protocol.

2.1.2 Messaging Patterns

A message pattern describes how applications communicate with each other. This communication is referred to as *messages*. There exist multiple messaging patterns and in this chapter we look into protocols using two very common approaches:

Request-Response

Request-response is a message pattern where a requester sends a request to a system. The system then processes the request and responds with a response message. This pattern is also known as client-server model.

Publish-Subscribe

Publish-subscribe is a message pattern where subscribers express their interest in a type of messages, often called topics or classes. Message publishers create messages of certain classes and publish them without needing to know who are subscribing to these types of messages. Many publish-subscribe systems employ a *message broker* as seen in fig. 2.1. The message broker handles published messages from publishers and receives subscriptions from subscribers. The broker can perform various tasks, such as message filtering and prioritize queuing.

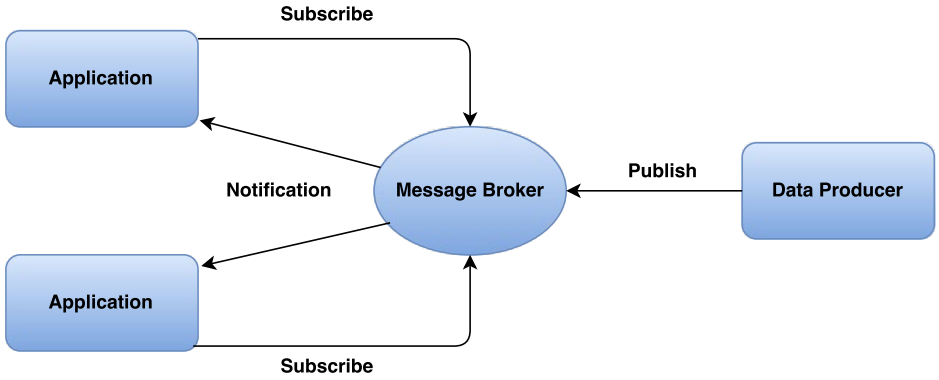


Figure 2.1: Message Brokers

2.1.3 Network Metrics

When transferring data over a network, the transfer is subject to many factors that may affect the transmission. A message sent over the Internet pass through communication infrastructure and equipment of

different quality and properties. Network metrics are used to describe various aspects of the data transfer from a node to another. In this thesis we use the following metrics:

Data Rate The data rate refers to the amount of data that can be transmitted over a network per unit of time.

Delay The delay is the time it takes for a bit of data from machine to machine to travel across the network.

Packet Error Rate PER means the number of incorrectly received packets divided by the total number of received packets. A packet is considered as incorrect if at least one bit error occurs.

2.2 Web Services

Web services are client and server applications that communicate over a network. They can be used to realize the SOA principles, and are in widespread use in both civilian and military systems. It is worth noting that the term *Web services* is a broad term and can be used to describe different types of services available over a network. The most common usage of the phrase refers to the W3C definition of SOAP-based Web services, but could also refer to more simple HTTP-based REST services.

In this thesis, we investigate optimization techniques that should support both W3C Web services and RESTful web services.

2.2.1 W3C Web Services

W3C has defined Web services as [11]:

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

This definition points out a set of standards that enable machine-to-machine interactions. Web service interfaces are described in documents called WSDL, and communication is based on sending XML-based SOAP messages. There exist many definitions of Web services where the core principles are the same, but the finer details may vary. Figure 2.2 illustrates the fundamental principles. The Web service technology is a realization of the SOA principles, and provides loose coupling and eases integration between systems.

These standards that together make W3C Web services are presented in the following sections.

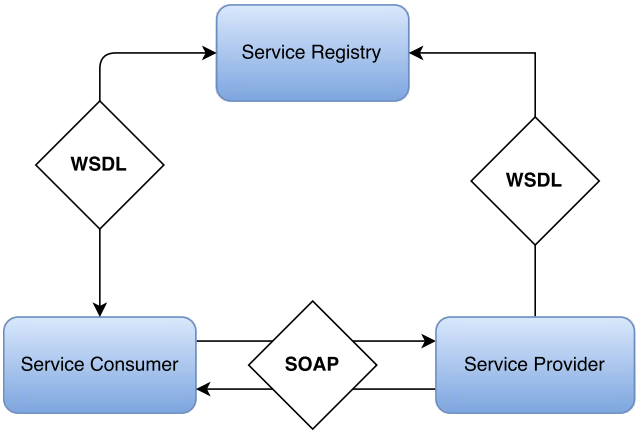


Figure 2.2: W3C Web services

Extensible Markup Language

The Extensible Markup Language (XML)[12] is considered as the base standard for Web services. An XML document consists of data surrounded by tags and is designed to be both machine and user readable. Tags describe the data they enclose. The tags can be standardized, which allows exchange and understanding of data in a standardized, machine-readable way.

Web Services Description Language

Web Services Description Language (WSDL) is an XML-based interface definition language that describes functionality offered by a Web service [13]. The interface defines available functions, data types for message requests and responses, binding information about the transport protocol, as well as address information for locating the service. The interface enables a formal, machine-readable description of Web service which clients can invoke.

SOAP

SOAP is an application level, XML-based protocol specification for information exchange in the implementation of Web services [14]. Data communication in SOAP is done by nodes sending each other SOAP messages. A SOAP message can be considered as an "envelope" consisting of an optional message header and a required message body. The header can contain information not directly related to the message such as routing information for the message and security information. The body contains the data being sent, referred to as the payload.

SOAP is transport protocol agnostic, which means it can be carried over various underlying protocols. The far most used transport protocol

is HTTP over TCP, but other protocols such as UDP and SMTP can be used as well.

2.2.2 Representational State Transfer

In the previous sections, we looked into the standards and specifications that compose W3C Web services. However, there also exist other types of Web services which do not follow these standards. In 2000, the computer scientist Roy Fielding introduced REST where he presented a model of how he thought the Web *should* work. This idealized model of interactions within a Web application is what we refer to as the REST architectural style [15]. REST attempts to minimize latency and network communication while maximizing the independence and scalability of component implementations. This is done by placing constraints on connector semantics rather than on component semantics like W3C Web services. REST is based on a client-server model where a client requests data from a server when needed.

Web services that adhere to the REST style are called RESTful Web services. They are closely associated with HTTP and use HTTP verbs (e.g. GET, POST, DELETE) to operate on information located on a server. RESTful Web services typically expose some sort of information, called resources in REST. Table 2.2 illustrates how a component exposes a set of operations of an example car resource. Resources are identified by a resource identifier. While W3C Web services are service oriented, we can look at REST as being more resource oriented.

Resource identifier	**HTTP Method**	**Meaning**
/vehicles/cars/1234	GET	Return a car with ID 1234 from the system.
/vehicles/cars/	POST	Create a new car which will be added to the list of cars.
/vehicles/cars/1234	DELETE	Delete a car with ID 1234 from the system.

Table 2.2: Example of REST operations

REST is easy to understand and has gained a lot of traction in the civil industry in the latest years. Although NATO has chosen W3C Web services as the technology to do information exchange, REST is identified as a technology of interest to certain groups in NATO [16]. One potential downside to NATO with REST, however, is that RESTful Web services lack standardization, which may cause interoperability issues.

Closely associated with REST and the most used transport protocol for W3C Web services are HTTP, which we present in detail in the next section.

2.3 Hypertext Transfer Protocol

As we have seen in the previous sections, both RESTful and W3C Web services rely on HTTP as the means of communicating with other services. The usage of HTTP is very widespread, and it is the foundation of data communication for the World Wide Web since the early 90's. It's protocol specification is coordinated by Internet Engineering Task Force (IETF) and the W3C, and is defined as [17]:

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers.

HTTP started out as a simple protocol for raw data transfer across the Internet and has since been updated in HTTP/1.0, HTTP/1.1 and most recently a major update in HTTP/2.0. It is a request-response protocol which means that all data exchanges are initiated with a client sending a HTTP request and then waits until a server responds with a HTTP response. A HTTP request consists of the request method, Uniform Resource Identifier (URI), protocol version, client information, and an optional body. The server responds with a message containing a status line, protocol version, a code indicating the success or error of the request, and an optional body. Both HTTP requests and responses use a generic message format and can contain zero or more HTTP headers. Headers are used to provide information about the request/response or about the message body, e.g. information about the encoding and caching information.

HTTP, being an application level protocol, relies on a transport protocol to transfer data to another machine. HTTP communication most often, but not necessarily, occurs over TCP/IP connections. The only requirement in the HTTP specification is that a reliable transport protocol is used.

2.3.1 HTTP Methods

Associated with all HTTP requests is a request method, which indicates the desired action to be performed on a resource located on a Web

server. The set of HTTP methods defined in HTTP/1.1 is listed in table 2.3.

HTTP Method	**Purpose**
OPTIONS	Asks the server which HTTP methods and header fields it supports.
GET	Retrieve information identified by the resource identifier (Request-URI).
HEAD	Identical to GET, except that the HTTP body is not returned from the server.
POST	Asks the server to accept the message payload from the client as a new resource.
PUT	Similar to POST but allows the client to ask the server to update a resource identified by the request URI.
DELETE	Requests that the resource identified by the request URI is deleted.
TRACE	Echoes the HTTP request. Used for debugging.
CONNECT	For use with a proxy that can dynamically switch to being a tunnel.

Table 2.3: HTTP methods

2.4 Transmission Control Protocol

TCP is called the workhorse of the Internet because it is so critical for how the Internet works. It is the primary transport protocol of the Internet Protocol Suite[10] and provides reliable in-sequence delivery of two-way traffic (full-duplex) data. TCP was defined in RFC 793[18] back in September 1981 and has since been improved in various RFC's. The main motivation behind TCP was to provide reliable end-to-end byte streams over unreliable networks. HTTP and other application layer protocols often use TCP as their transport layer protocol. In the coming sections, we therefore present TCP in detail and discuss some of the issues we may encounter using it.

2.4.1 The Protocol

TCP is a connection-oriented protocol, which means that a connection between a sender and the receiver must be established before any data can be transferred. TCP does this by using a three-way handshake to establish a connection. For each connection, TCP initializes and maintains some status information such as window size, socket information and sequence numbers.

Computers supporting TCP have a piece of software which manages TCP streams and interfaces to the Internet layer. Most often this software is a part of the kernel [19]. It accepts data streams from local processes and breaks them up into pieces, before sending them to the Internet layer. The pieces are called TCP segments, which consist of a fixed 20-byte header, followed by zero or more data bytes. The TCP software decides how big the segments should be, but for performance reasons they should not exceed the Maximum Transfer Unit (MTU) of the link (the physical network). Each segment should be so small that it can be sent in a single, unfragmented package over the entire network. This usually limits the size of each segment to the MTU of the Ethernet, which is 1500 bytes.

When the TCP software receives data from applications, it is not necessarily immediately sent as it may be buffered before its sent. At the receiving end, data is delivered to the TCP software, which reconstructs the original byte stream and delivers it to the destination application.

2.4.2 TCP Reliability

When transferring data over the Internet, the data may pass through various networks, routers, and physical networks. Some of the routers may not work correctly, a bit may be flipped when transferring data wirelessly, or some other factor may come into play. For these reasons, we have to accept that some of the data will be damaged, lost, duplicated or delivered out of order.

TCP recovers from such faults by assigning sequence numbers to each packet being sent. It then requires a positive acknowledgment from the receiver that the data was actually received. If the acknowledgment is not received within a timeout interval, the data is transmitted again. For the receiver, the sequence numbers are used to ensure that data is received in the correct order, as well as eliminating duplicates. Furthermore, to detect damaged data, TCP applies checksums to each segment transmitted. At the receiver, the checksum is then checked, and damaged segments are discarded.

2.4.3 Flow Control

If a fast receiver sends data faster than a slow receiver can process, the receiver will be swamped with data and may experience severe performance reduction. Flow control is a mechanism to manage the rate of the data transmission to avoid overflowing a receiver. TCP provides this by using a window of acceptable sequence numbers that the receiver is willing to accept. With every acknowledgment sent back to the sender, the window is specified. This allows the receiver to control which segments, and how fast, the sender can send.

2.4.4 Congestion Control

Congestion control is about controlling the entry of data traffic into a network to avoid network congestion. On its way from the sender to the receiver, an IP packet may pass through different subnets with different capabilities. Network congestion may occur if a node in a network receives more data than it is able to pass forward. The consequence of this is that an increase in network traffic to this node would only lead to a small increase, or even a decrease, of the network throughput [20].

To avoid congestion, TCP uses a number of mechanisms. These aim to control the rate of data packets entering into the networks to avoid congestion, but still get as high throughput as possible. One of these mechanisms is *slow-start*, which the general idea is to start transmitting with a low packet rate, then gradually increasing the packet rate.When TCP notice that a packet is eventually lost, it considers it as a sign of network congestion and reduces the rate it sends packets.

2.4.5 Issues Using TCP in DIL

DIL networks are characterized by their high delay, low data rate, and relatively high error rate. Since TCP's congestion control interprets this as evidence of congestion, it will back off and send with a lower packet rate. This could cause TCP to send with a lower rate than the network actually can provide. Moreover, it could also ultimately lead to the TCP connection terminating due to those effects [7].

2.5 Protocols of Interest

Since TCP may be sub-optimal or even break down entirely in DIL networks, we are in this thesis looking into alternative protocols and other optimization techniques. In networks with low data rate, protocols with low overhead per IP packet are beneficial. With frequent disconnects, protocols that are connection-less may be more suitable than connection-oriented. One important limitation is that NATO has chosen the "everything over IP", which means that all optimization must

occur on the top of the network layer. Because of this, we evaluate protocols in the transport and application layer.

There exist a vast amount of protocols developed for different use-cases and by different organizations. In this thesis, we focus on protocols that are established and/or standardized. Using proprietary solutions, or protocols which lacks standardization, may contribute to the proxy solution being more difficult to get approved by military security authorities. In the following sections, we give an introduction to the protocols we are investigating in this thesis. The protocols is because of their prevalence in the civilian and military world or their reported performance in the IoT. Protocols used in IoT are of interest to us since they are designed to work in networks which have similar characteristics as DIL networks.

We get started by discussing the User Datagram Protocol (UDP), which alongside TCP is one of the core protocols of the Internet protocol suite.

2.5.1 User Datagram Protocol

The Internet has two main protocols in the transport layer, namely UDP and TCP. They have fundamentally different characteristics and use cases, which we go through in this section. UDP was formally defined in 1980 in RFC 768[21] and is a simpler protocol than TCP. It sends messages, called datagrams, to nodes over the IP network. While TCP provides reliable transmission along with flow control and congestion control, UDP only supports the sending of IP datagrams. Furthermore, it is a connectionless protocol, which means that the protocol can send messages *without* establishing a connection first. Since UDP does not provide guaranteed delivery or in-order delivery of messages, it should only be used by applications that do not require this.

To summarize, UDP is a more lightweight protocol than TCP. It has smaller headers and less overhead. The downside is that it does not provide any mechanisms for congestion control or reliability. UDPs lack of end-to-end congestion control may result in drastic unfairness if a UDP stream is competing with a TCP stream [22]. While a TCP stream will detect congestion and back-down its traffic, a UDP stream will greedily send at full-throttle, thus causing an unfair share of the available network. UDP is therefore often referred to as not *TCP-Friendly*.

It is worth noting that UDPs lack of reliability may by handled on a higher level in the application stack on top of UDP. This is done by the next protocol we are looking into.

2.5.2 Constrained Application Protocol

Constrained Application Protocol (CoAP) is a specialized Web transfer protocol designed for use with constrained nodes and networks [23]. It is intended for machine-to-machine applications typically found in the Internet of Things. Furthermore, it is designed with a similar interface as HTTP to integrate easily with Web services. CoAP and HTTP work similarly in the way that they both use a client-server interaction model. CoAP is based on the REST style where a server makes resources available under a URI. Clients can then interact with these resources using a subset of the HTTP-verbs: GET, PUT, POST, and DELETE.

CoAP messaging is based on asynchronously exchanging CoAP messages over UDP between two endpoints. The current specification defines four types of CoAP messages where each message uses a 4-byte fixed-length binary header. Table 2.4 lists the four types of CoAP messages. A CoAP header may be followed by *options* and a payload. CoAP provides mechanisms for optional reliability since UDP itself does not guarantee reliable delivery. This is done by sending messages marked as *Confirmable*, and retransmitting using a default timeout and exponential back-off until an *Acknowledgment message* is eventually received. Basic congestion control is done by strictly limiting the number of allowed outstanding requests between a client and a server. The back-off mechanism also provides basic congestion control.

CoAP message	**Purpose**
Confirmable Message	CoAP message that requires an acknowledgement. Used to provide reliable transport.
Non-confirmable Message	Used when no acknowledgement is wanted.
Acknowledgement Message	Acknowledges that a specific Confirmable Message has arrived.
Reset Message	Indicates that a Confirmable Message or a Non-confirmable Message was received, but not understood by the client.

Table 2.4: CoAP messages

Typical CoAP messages may be small payloads from Internet of Things devices such as temperature sensors, light switches, etc. The CoAP specification states that a CoAP message *should* fit within a single IP packet to avoid IP fragmentation. However, occasionally larger messages are needed. Therefore, a blockwise transfer technique has been proposed as an extension to CoAP in an Internet Draft [24]. The

block option allows for sending larger messages in a block-wise fashion.

2.5.3 Advanced Message Queuing Protocol

The Advanced Message Queuing Protocol (AMQP) is an application layer protocol for sending messages. It supports both the request-response and the publish-subscribe communication paradigms. AMQP uses TCP as its underlying reliable transport layer protocol.

An important observation about AMQP is that it has two major versions which are fundamentally different, version 0.9.1 and 1.0. The latter has been standardized by OASIS and is a more narrow protocol as it only defines the network wire-level protocol for the exchange of messages between two endpoints [25]. The concept wire-level protocol refers to the description of the format of data sent over a network in form of bytes. Another difference between the versions is that version 1.0 does not specify the details of broker implementation. We investigate version 1.0 since it is the newest and has been standardized.

An AMQP network consists of nodes connected via *links*. Nodes can be producers, consumers, and queues. Producers generate messages, consumers process messages while queues store and forward them. These nodes live inside *containers*, which can be client applications and brokers. Each container can have multiple nodes. AMQP version 1.0 does not specify the internal workings of those nodes but defines the protocol for transferring messages between them. The basic data unit in AMQP is called a *frame* and is used to initiate, control and tear down the transmission of messages between two nodes. The 9 different frames are listed in table 2.5.

AMQP is connection-oriented since an AMQP connection must be established before to any communication. A connection is divided into independent unidirectional *channels*. An AMQP *session* correlates two unidirectional channels to form a bidirectional, sequential conversation between two containers. To establish a connection, the first operation is to create a TCP connection between the nodes. Then the protocol header is exchanged, allowing the nodes to agree on a common protocol version. This is exchanged in plaintext (not in an AMQP frame). The message itself is sent with the *transfer* frame. Larger messages can be split into multiple frames.

2.5.4 MQTT

MQTT is a publish-subscribe messaging transport protocol [26]. It emerged in 1999 and recently became an OASIS standard in 2014. MQTT is considered to be lightweight and simple to implement, making it suitable for use in networks where the data rate is limited and/or a low code footprint is needed. These properties make MQTT popular as a

AMQP Frame	Purpose
Open	Describes the capabilities and limits of the node
Begin	Begin a session on a channel
Attach	Attach a link to a session
Flow	Update link state
Transfer	Transfer a message
Disposition	Inform remote peer of delivery state changes
Detach	Detach the link endpoint from the session
End	End the session
Close	Signal a connection close

Table 2.5: AMQP Frames

IoT protocol. The protocol is broker-based and runs on top of the TCP/IP protocols.

MQTT provides message sending services to applications and offers different levels of Quality of Service (QoS), specifying the delivery policies for a message. This is beneficial in networks where messages may be lost while traveling through a network. The lowest level of QoS is *at most once*, which specifies that a message should arrive at the receiver either once or not at all. Next, the policy *at least once* ensures that the message arrives at the receiver at least once, but possible multiple times. The last and highest level of MQTT's QoS, *exactly once*, guarantees one, and only one, delivery of the message. The protocol works by sending different MQTT control packets, listed in table 2.6. Only *exactly once* QoS requires the usage of the control packets PUBREC, PUBREL and PUBCOMP.

2.5.5 Stream Control Transmission Protocol

The Stream Control Transmission Protocol (SCTP) is a transport-layer protocol, which offers functionality from both UDP and TCP [27]. The motivation behind the protocol is that many developers find TCP too limiting, but still require more reliability than UDP can provide. SCTP tries to solve these issues. It is message-oriented like UDP, but ensures reliable, in-sequence transport of messages with congestion control like TCP. SCTP is a connection-oriented protocol and provides features like multi-homing and multi-streaming. Multi-homing is the possibility to use more than one network path between two nodes. This increases reliability since if one path fails, messages can still be sent over the other links. Multi-streaming refers to SCTP ability to transmit several independent streams of data at the same time, for example sending an

MQTT Control Packet	Purpose
CONNECT	Client requests a connection to the Server
CONNACK	Acknowledge connection request
PUBLISH	Publish a message
PUBACK	Publish acknowledgement
PUBREC	Publish received
PUBREL	Response to a PUBREC Packet
PUBCOMP	Publish complete
SUBSCRIBE	Subscribe to topics
SUBACK	Subscribe acknowledgement
UNSUBSCRIBE	Unsubscribe from topics
UNSUBACK	Unsubscribe acknowledgement
PINGREQ	PING request
PINGRESP	PING response
DISCONNECT	Disconnect notification

Table 2.6: MQTT Control packets

image at the same time as a HTML Web page.

2.6 Summary

In this chapter, we presented computer networks in general, before we discussed the two most common types of Web services. Moreover, we reviewed the protocols that these Web services use to transmit messages over the Internet. We also introduced some new protocols designed to work in "Internet of things" networks, which have many of the same characteristics as DIL networks. The protocols are summarized in table 2.7.

Many of the mentioned protocols have been previously researched for use in DIL networks. In the next chapter, we present relevant work in this area.

Protocol	Network layer	Summary
TCP	Transport.	Stream-oriented transport protocol. Reliable and with congestion control.
UDP	Transport.	Message oriented. Low overhead, but lacks reliability and TCP-friendliness.
SCTP	Transport.	Similar to UDP but also provide reliable, in sequence transport of messages like TCP.
HTTP	Application. Uses TCP.	Widely used and the foundation for World Wide Web.
CoAP	Application. Uses UDP.	Low header overhead with optional reliability.
AMQP	Application. Uses TCP.	Messaging middleware with store-and-forward capabilities.
MQTT	Application. Uses TCP	Light weight and simple pub/-sub protocol.

Table 2.7: Summary of protocols

Chapter 3

Related Work

In this chapter, we discuss earlier relevant work in the area of improving the performance of Web services in DIL environments. Improving Web services is important for both civil and military users as increasing the performance means that applications can become faster and more reliable.

In the following sections, we identify studies and recommendations that apply to this thesis. We get started by looking into the work of the NATO research groups IST-090 and "SOA Recommendations for Disadvantaged Grids in the Tactical Domain" (IST-118). IST-118 is an ongoing follow-on to the work of IST-090, with the goal of creating a recommendation for a tactical profile for using SOA in disadvantaged grids. Next, based on these recommendations, we investigate work done in the area of alternative transport protocols and existing proxy implementations. Finally, we summarize the findings that are applicable with regards to the scope and premises of this thesis.

3.1 Making SOA Applicable at the Tactical Level

IST-118 has published a paper[28] where they summarized the findings of IST-090. Although the paper only looked into W3C Web services, many of their recommendations are also applicable to RESTful Web services. They identified three key issues that need to be addressed to adopt Web services in tactical networks:

1. End-to-end Connections

Web services mostly use transport protocols that depend on a direct, end-to-end connection between a client and the service. Attempting to establish and maintaining a connection in a DIL environment can lead to increased communication overhead and the possible complete breakdown of communication. Most Web services use TCP as their transport protocol, which relies on an uninterrupted connection. In DIL environments with high error rates and high latencies, the congestion

control of TCP can cause sub-optimal utilization of the network as previously discussed in section 2.4.5. Similar, HTTP, which is the application layer protocol most often used together with TCP, struggles in such environments. HTTP is a synchronous protocol, which means that the HTTP connection is kept open until a response is received. Long response times could cause timeouts. IST-090 points out the possible solution of replacing HTTP and TCP with other, more suitable protocols.

The IST-90 report mentions two approaches to replace HTTP/TCP. The clients and services themselves can be modified to support other protocols, or proxies which support alternative protocols can be used [6]. Moreover, they pointed out that if using a proxy approach, standards compliance can be retained.

2. Network Heterogeneity

Another issue is when heterogeneous networks are interconnected. Different performance in networks may lead to the buildup of data in buffers, risking the loss of information. A proposed solution to this is to have store-and-forward support which can support that messages are not dropped, but rather stored and forwarded when possible. Another important usage of this technique is to overcome network disruptions because messages can be stored until the network connection is reestablished.

3. Web Service Overhead

W3C Web services are associated with a considerable amount of overhead. Web Service technology is based on SOAP, which uses XML-based messages. It is a textual data format and produces much larger messages than binary formats. Optimization approaches should seek to reduce the network traffic generated by Web services by using techniques as compression to reduce the size of messages. Another method is to decrease the number of messages being sent, which was looked into in IST-090 [6]. In their work they investigated three different ways to do this:

1. Employing caching near the client in order to reuse older messages.
2. Using the publish-subscribe paradigm, which allows clients to subscribe to information instead of requesting it. This allows the same message to be sent to multiple clients.
3. Employing content filtering which filters out unnecessary data.

The scope of this thesis is to optimize for request-response type of clients and Web services. Furthermore, since we are investigating

general-purpose optimization techniques without knowledge of the payload, some of these recommendations does not quite apply to us. However, to reduce Web service overhead, we can apply the well-known technique of compression.

Compression

Data compression is the technique of encoding information using fewer bits than the original representation. In a network with limited data rate, the reduction would significantly reduce time used to send the data. The reduction of data is often expressed in the term *compression rate*, which represents the ratio between the uncompressed size and compressed size of the payload. Moreover, there exist two types of compression, *lossy* and *lossless compression*. Lossy compression is used to compress data such as images and movies where the consequence of losing some of the data is not critical. Lossless compression utilizes repeating patterns in the data in order to represents the same information in a more efficient way.

XML is the data format used by W3C Web services and has a significant overhead. A previous study evaluated different lossless compressions techniques for exchanging XML documents using W3C Web services [29]. They evaluated both XML-specific and general purpose (payload agnostic) compression techniques. There exist a great number of different compression techniques, so the authors focused on a few they saw as promising for use in tactical communication networks. The first one, Efficient XML (EFX), encodes XML documents in a binary instead of textual format. The two other were the XML-specific XMLPPM and general-purpose compression tool GZIP.

In their evaluation, they saw that for all techniques, larger XML documents achieved a higher compression ratio than smaller documents. As the average, EFX applied with a built-in proprietary ZIP enabled had the highest compression ratio followed by GZIP. However, they concluded that all evaluated techniques provided a significant reduction of payload size, so the specific technique was of less importance.

3.2 Previous Evaluations of Alternative Protocols

A previous study has investigated potential gains from replacing HTTP/TCP with alternative protocols [30]. Johnsen et al. looked into TCP, UDP, SCTP and AMQP for conveying Web services traffic under typical military networking conditions. The researchers found that SCTP had the highest success rate in military tactical communication. However, on links with the lowest data rate, the protocol tended to generate more overhead than TCP. They pointed out that this was due to SCTP

having a more complex connection handshake procedure and besides using heartbeat packets.

Another study has compared the performance of MQTT and CoAP concerning end-to-end delay and network usage [31]. Thangavel et al. performed experiments in different emulated networks with varying message sizes and loss rates. They saw that both MQTT and CoAP were successfully able to handle packet losses of up to 25 %. With lower loss rates, messages sent with MQTT had the least delay, but as the loss rate increased, CoAP had a lower delay. They identified the reason for this being that the TCP transmission of MQTT had a larger overhead than CoAP's UDP transmission. In their experiments with small message sizes and for all tested loss rates, CoAP had less network overhead than MQTT. However, when the message size grew, MQTT had less overhead than CoAP.

Another comparison of CoAP and MQTT was done in a study using the protocols for sensing applications running on a smartphone [32]. This study also confirmed CoAP as having lower network usage and a lower RTT. However, the study pointed out that MQTT has more advanced QoS services, since it can guarantee exactly-once delivery of messages. Since CoAP does not have this feature, applications which require this functionality should consider using MQTT.

CoAP has also been compared against HTTP in work done by Colitti et al., where they performed an evaluation with regards to response time and energy consumption by a sensor node [33]. They found that using CoAP consumed significantly lower energy than using HTTP and that CoAP also had a lower response time.

3.3 Proxy Optimization

One of the recommendations of IST-090 was the usage of proxies. This recommendation has been picked-up by other research groups and a set of proxies for optimizing Web services in DIL networks already exist. However, many of them do not fulfill all the requirements we have for our proxy. Some of them do only support SOAP Web services, and others are unusable due to security reasons. This section lists and discusses previous implementations of such proxies.

3.3.1 Types of Proxies

A proxy is a node deployed somewhere in a network, which through network traffic can pass. Proxies have many use cases such caching, firewalling and security. To adopt Web services into tactical networks, we mainly talk about two types of proxies. Edge proxies act as gateways between different networks as illustrated in fig. 3.1 and can perform adaptations on network traffic passing through it.

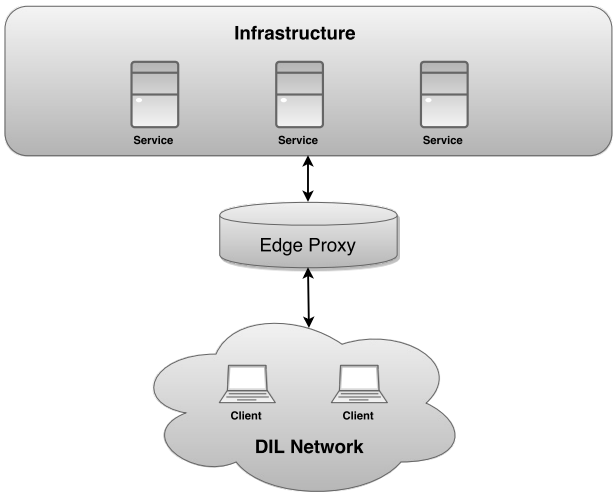


Figure 3.1: Edge proxy

Another type, is point-to-point proxies in a network as illustrated in fig. 3.2, which involves using a proxy-pair to facilitate communication between two or more applications.

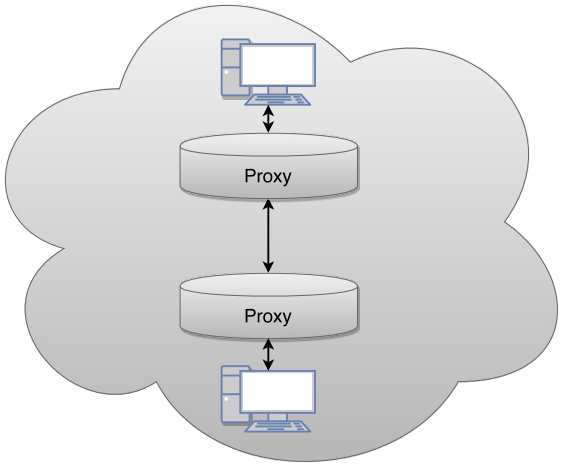


Figure 3.2: Point-to-point proxy

3.3.2 Delay and disruption tolerant SOAP Proxy

The Delay and disruption tolerant SOAP Proxy (DSProxy) is a point-to-point proxy solution developed by Norwegian Defence Research Establishment (FFI) [34][35]. Its goal is to enable the usage of unmodified standard W3C Web services (SOAP over HTTP/TCP) in DIL environments. The concept is to route all SOAP messages through the

proxy. When the proxy receive a message, it is stored locally before it is forwarded. If the forwarding fails for some reason, it is retries the request until it eventually succeeds. This ability, called *store-and-forward*, is one of the fundamental core functionalities of DSProxy. When a request eventually succeeds, the response is returned to the client on the original TCP connection initiated by a client. By doing this, Web service invocations is made possible over unreliable networks by hiding any network disruptions from the client.

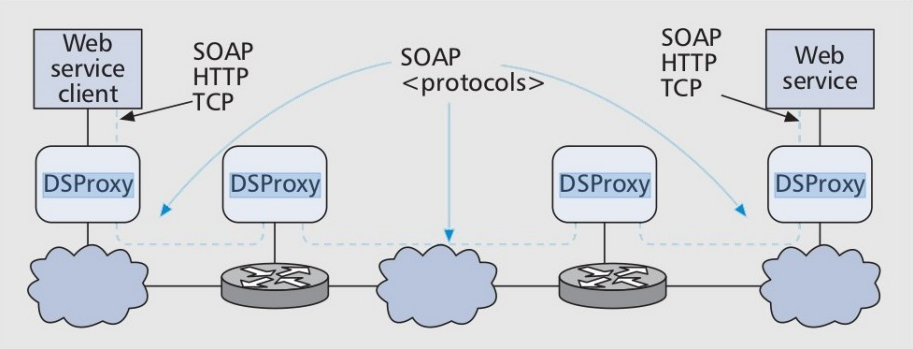


Figure 3.3: DSProxy overlay network (from [35])

Another core functionality of DSProxy is mechanisms for organizing an overlay network consisting of multiple proxy instances as seen in fig. 3.3. The overlay network enabled the ability to traverse multiple and heterogeneous networks, but also add a lot of complexity to the proxy application. Apart from the mentioned core functionalities, DSProxy support a set of pluggable features such as GZIP compression and caching.

After performing experiments using the DSProxy, the researchers identified the store-and-forward ability as important in unreliable networks to avoid having to re-establish end-to-end connections each time the network connections was lost [34]. One of the downsides with DSProxy was that it only supported W3C Web services. Moreover, it became very complicated due to its mechanisms for building overlay networks and supporting different configurations and plugins.

3.3.3 NetProxy

NetProxy is another network point-to-point proxy solution aiming at enabling SOA applications for use in DIL environments [36]. The proxy is a component of the Agile Computing Middleware (ACM), a set of components that satisfy many of the communications requirements found in challenged networks. The work is being carried out by researchers at the Florida Institute for Human & Machine Cognition.

Like DSProxy, NetProxy is a transparent proxy providing integration between SOA systems without requiring modification of applications

themselves. It works by first intercepting all network traffic from applications and then do an analysis of it. Together with information about the network, NetProxy then decides which appropriate action to take. It can be configured to support protocol remapping by using other protocols than HTTP/TCP. Integrated with NetProxy is the message-oriented transport protocol Mobile Sockets (Mockets), which is designed to replace TCP and UDP and is targeted for DIL networks [36]. Mockets substitutes the congestion control and reliable transmission algorithms of TCP with other alternate implementations designed for DIL networks. It is configurable for different types of networks and offers various QoS levels.

Performance testing of W3C Web services showed that using NetProxy with Mockets as the transport protocol yielded a significant increase in the performance compared to plain TCP [36]. The researcher attributed this to several factors:

- Mockets handles packet loss much better than TCP since TCP attributes packet loss to congestion and triggers its congestion control.
- NetProxy multiplexes all network traffic directed to a single node onto the same connection and holds it open instead of closing it after a finishing request. This allows reusing the connections across consecutive requests, also from other applications.
- Less overhead due to the fact that NetProxy buffers data until it fills an entire packet before sending it over the network.
- Enabling compression gave a very high gain in the measured network throughput partly due to the messages subject for compression was XML documents, which have a relatively high compression rate.

Although the testing shows some promising results, there are some issues regarding using the Mockets protocol in our proxy solution. Applications meant for usage in military systems, must be approved by military security authorities. Mockets has not been standardized and lacks usage outside the experiments conducted in the mentioned research. Because of this, it may be difficult to get applications using Mockets approved.

3.3.4 AFRO

Adaption Framework foR Web Services prOvision (AFRO) is an edge proxy which offers different levels of QoS to Web services through performance monitoring and usage of the context-aware service provision paradigm [6]. It performs so-called adaption actions, which modifies the

SOAP XML messages by changing their encoding to more efficient data representation. AFRO also removes information that is acceptable to be removed by the service requester.

However, since the proxy modifies the data being sent, the digital signature of the data is also changed. In applications where we want to be sure that no one has tampered with the data before arriving, digital signatures are often used. Consequently, this solution would not work for such applications.

3.3.5 TACTICS TSI Architecture

Another ongoing effort to overcome issues using standard Web services in tactical networks is the TACTICS project supported by the European Defence Agency (EDA) [37]. The goal of the project is to propose a reference architecture for a Tactical Service Infrastructure (TSI) suited for establishing a tactical SOA of defence-related information systems. The architecture features a service middleware meant to run on devices with different capabilities. The purpose of the middleware is to receive standard Web service invocations or responses and ensure that they complete. The middleware can forward IP packets between different radio networks and store and forward messages. The concept is based on the point-to-point proxy approach.

3.4 Tuning Application Server Parameters

Another approach to improve the performance of Web services is to configure the way they are deployed. Web services can be deployed in applications servers, which is a software framework that provides an environment where the Web services can run. When setting up an application server, several parameters which can affect the performance of running applications can be configured. A wrong or bad configuration may cause inaccurate timeouts and congestion in the network. In a paper written by researchers at Norwegian University of Science and Technology (NTNU) and FFI [16], they investigated how tuning the server parameters of the application server Glassfish affected the performance of both REST and SOAP Web services. They identified a number of key HTTP and TCP tuning parameters:

HTTP Timeout Controls how long a HTTP connection can be deemed as idle and kept in the "keep-alive" state. Having a too low timeout on networks with low data rate, can potentially flood the network with packets that have timed out. Consideration should therefore be taken when setting this parameter for mobile tactical networks.

HTTP Compression Enables HTTP/1.1 GZIP compression.

HTTP Chunking Allows the server to send data in dynamic chunks.

HTTP Header and Send Buffer Sizes Vary the size of the buffers that hold the request and send the data.

TCP Idle Key Timeout Sets the time before an idle TCP channel closes.

TCP Read and Write Timeouts Set the timeout for TCP read and write operations, respectively.

TCP Selector Poll Timeout Sets the time a Java new/non-blocking I/O (NIO) selector will block waiting for user requests.

TCP Buffer Size Sets the size of the buffer that holds input streams created by the network listener.

TCP Batching/TCP NO_DELAY Batches together small TCP packets into larger packets.

MTU Size The maximum transmission unit size regulates the largest data unit that can be passed onwards. In tactical military communication the MTU size can be very low (down to 128 bytes).

After running their experiments, they concluded that few of the parameters actually had any significant impact on the performance of the Web Service. However, they identified HTTP Chunking configuration as having the most impact on the performance. It significantly improved the performance for both SOAP and RESTful Web services in different types of networks.

While tuning application server parameters may help improve the performance of Web services in DIL environments, it is not directly related to the proxy solution investigated in this thesis. When deploying Web services, this optimization technique should be considered, but are not explored further in this thesis.

3.5 Summary

In this chapter, we looked into efforts previously undertaken in order to improve the performance of Web services in networks with the DIL characteristics. We first investigated the work of the research groups IST-090 and IST-118, and saw how they identified end-to-end connections and Web service overhead as major issues for enabling Web services in DIL environments. To overcome these problems, they

recommended the usage of proxies and several techniques for reducing the overhead. We identified GZIP and EFX with zipping as important compression techniques to reduce the size of Web service messages sent over a network. Next, we looked into previously developed proxies for DIL networks. Although many of them showed promising results, some of their properties did not fulfill the premises for this thesis. They were either limited to SOAP-based Web services or are inadequate to be used due to security reasons. However, we identified some of their techniques that we carry on in the proxy developed in this thesis.

Finally, we investigated previous attempts with the usage of alternative transport protocols, before we looked into previous efforts in the area of tuning application server parameters. Based on recommendations and studies of previous work, we are in the next chapter deriving a set of requirements for our proxy.

Chapter 4

Requirement Analysis

In this chapter, we discuss the requirements for optimization techniques aiming at enabling Web services in DIL environments. These requirements build on the scope and premises discussed in the introduction. The defining premises were that the proxy should:

1. Support HTTP RESTful and W3C Web services.
2. Work in DIL networks.
3. Be interoperable with standards-based COTS solutions.
4. Work with security mechanisms.

Based on previous studies, in particular the work of the NATO research groups IST-090 and IST-118, we are in this thesis developing a point-to-point proxy solution supporting these premises. In the following sections, we discuss the specific requirements for the proxies.

4.1 HTTP Proxy

The first premise is that the proxy should support HTTP RESTful and W3C Web services. While the majority of W3C Web services use HTTP to transport their SOAP messages, some use for example UDP or TCP. To avoid the proxy becoming to complex and while still supporting the majority of Web services, we chose to only support HTTP Web services. Our first premise therefore implies that the proxy must be able to forward HTTP messages. Furthermore, the third and fourth premises have some important implications for our proxy. Our proxy must be able to accept HTTP requests from a Web service, forward it to the other proxy, which in turn delivers it to the intended receiver. The communication between the proxies is not required to be with HTTP, but rather using a protocol that deals with DIL networks in a better way. However, since ultimately a HTTP request should be delivered to the intended receiver, the HTTP properties must be retained. This means

that the proxy must preserve the HTTP method and headers. Also, since REST is payload agnostic, the proxy must be able to support different types of data being sent through it (XML, JSON etc.).

Furthermore, the proxy must be able to handle the challenging network conditions of DIL. The specific requirements are outlined in the following sections.

4.2 Cope with DIL Networks

The DIL term refers to three aspects of a network, *disconnected, intermittent* and *limited*. The proxy should be able to overcome the implications of these issues. In the following sections, we discuss the requirements each aspect implies.

4.2.1 Disconnected

The Disconnected aspect of DIL refers to disconnects for a longer period of time. As we saw in the previous chapter, earlier work has identified the removal of end-to-end dependencies as important to overcome this aspect. Without proxies, a disconnect for a longer period of time would cause a timeout exception at the Web service, leaving it up to Web service itself to deal with the exception. By employing a proxy pair, the end-to-end dependency is instead moved from between a client and a Web service, and to between the client and the locally deployed proxy. As a result, the connection between the proxies over a DIL network can be lost, while still maintaining the connection between the client and local proxy. When the connection is reestablished, the proxy must be able to continue transmission of messages on behalf of clients.

This requires the proxy to have some redelivery mechanism. When a proxy detects that it unable to transmit messages to the other proxy, it should ideally wait until the connection is reestablished before trying to send more messages. However, the only way to know if the connection is restored is to try and send more messages and see if they succeed. The first, and maybe naive approach, could be just to retransmit the message again and again. But by doing this, we could risk overflowing a slow receiver, as well as causing congestion in a possibly overloaded network. Different types of networks and different use cases for the applications involved may require different redelivery mechanisms. Therefore, at deployment, the proxy should support a set of configurable redelivery mechanism properties:

Redelivery Delay The proxy should support the retransmission of sending messages with a fixed delay between each attempt.

Exponential Backoff If exponential backoff is configured, the proxy should gradually try resending more and more seldom.

Maximum Redeliveries The proxy should support user configuration of how many times a retransmission should be attempted before giving up.

4.2.2 Intermittent

The proxy should handle brief, temporary disconnects that can occur in a DIL network. It is comparable to the disconnect aspect, as intermittent refers to a shorter disconnect. A "long" intermittent disconnect triggers a timeout at the application layer and should be dealt with by the proxy retransmission mechanisms. With shorter intermittent disconnects, the transport protocol should be able to deal with it. This requires using a reliable transport protocol or handling it in the application layer.

4.2.3 Limited

Limited refers to different ways a network can be limited. Accordingly, the proxy must cope with very low data rates, possible high error rates, and long delays. This implies that reducing Web service overhead in order to lower the amount of bytes that need to be sent over a limited network is important. Moreover, the proxy may run on machines with restricted resources (battery capacity), which means that a low CPU overhead is desired.

4.3 Support Optimization Techniques

The proxy should support a set of optimization techniques to improve the performance of Web services in DIL environments. As we discussed in the related works chapter, there exist many approaches to optimizing Web services. Reducing Web service overhead by using compression was identified as a technique that yields a significant improvement. Another method was the usage of alternative transport protocols. In this thesis, we focus on compression and the usage of alternative protocols as the means of optimizing Web services. However, the prototype proxy should be designed to easily support additional optimization techniques in the future.

4.3.1 Compression

Compression reduces the size of a message sent over a network. To perform compression, the proxy must be able to modify the payload of the message. Due to security mechanisms that detect changes to the payload (digital signatures), the payload must be restored back to its original form before being forwarded to the final receiver. One of our premises is that we must support both RESTful and W3C Web services. RESTful services do not put any restrictions on the data format of a

message. Thus, we cannot use XML-specific compression, but rather we need to use general-purpose techniques.

Based on previous work we identify GZIP as the best approach for general purpose compression.

4.3.2 Inter-Proxy Communication

One of the optimization techniques we identified is the usage of alternative transport protocols between the proxy pair. The proxy should support different protocols to allow us to evaluate how they affect the performance of Web services in DIL networks. Furthermore, if additional transport protocols are identified in the future, the prototype proxy should be easily extendable to support these.

We introduced a set of protocols in the technical background chapter and discussed previous evaluations using them in DIL networks in the last chapter. In the following paragraphs, we analyze them for usage in the context of inter-proxy communication in a DIL network.

HTTP The by far most used protocol for Web services is HTTP over TCP. TCP is an old and proven protocol and was originally designed to provide reliable end-to-end communication over unreliable networks. The less intrusive optimization technique would therefore be that the proxies simply forward HTTP-requests without using an alternative protocol. Although proxying Web service requests through proxies would cause some overhead from processing time and custom proxy headers, we still get the benefit of breaking the end-to-end dependency and the possibility of using compression. Furthermore, using HTTP allows us to compare the "standard" protocol against other protocols. Therefore, we recommend HTTP as a possible proxy pair communication method.

UDP UDP has less overhead than TCP but lacks mechanisms for reliability and congestion control. The lack of reliability could be handled at the application level instead, but would require a library on top of it. Furthermore, UDP is not TCP-friendly. For these reasons, we conclude that UDP is unfit for proxy communication as part of this thesis.

CoAP CoAP is a relatively new protocol intended for use in the Internet of Things. It is designed to have low overhead, low code footprint and be easily mapped to and from HTTP. These properties make the protocol interesting as the means of communication between a proxy pair.

AMQP AMQP is in widespread use and offers reliable message transmission. Therefore, we recommend AMQP as a possible proxy pair communication method. AMQP supports both the request-response and publish-subscribe message paradigms.

MQTT MQTT is a publish-subscribe messaging protocol and is considered as lightweight and simple to implement. However, the inter-proxy communication requires a request-response type of messaging. MQTT does not facilitate this type of communication. With that said, it is possible to have a request-response paradigm on top of publish-subscribe by organizing queues and by using some application logic. However, since MQTT does not natively support request-response, we do not recommend this protocol for inter-proxy communication.

SCTP SCTP offers functionality from both UDP and TCP. It is reliable and has been identified in previous related work as an interesting protocol for DIL networks. Therefore, we recommend it as a possible proxy communication method.

The proxy should support the identified protocols found suitable for communication between proxies over a DIL network. Table 4.1 summarize our recommendations. For evaluation purposes, the proxy should be easily configured with which protocol to use.

Protocol	**Recommendation**
HTTP	Yes
UDP	No
CoAP	Yes
AMQP	Yes
MQTT	No
SCTP	Yes

Table 4.1: Protocols recommended as possible proxy communication protocol.

4.4 Summary

In this chapter, we have discussed the requirements for our proxy, which we summarize here:

1. Receive and forward HTTP requests.
2. Retain HTTP request and response headers.
3. Support GZIP compression of payload.
4. Handle frequent network disruptions.
5. Handle disconnects over longer periods of time.
6. Handle low data rates, high delays and high packet error rates.

7. Allow for configuration of redelivery delay and maximal number of retransmissions.
8. Support usage of different transport protocols between the proxies.
9. Easy configuration of which protocol to use.
10. Be easily extendable to include other protocols and other optimization techniques.

Next, we discuss the design and implementation of our proxy supporting the premises and identified requirements.

Chapter 5

Design and Implementation

Based on the premises and requirements identified in the previous chapter, we are in this chapter introducing the design and implementation details of our proposed proxy solution. Before we start looking into the specific design and implementation, we present the primary area of use for our point-to-point proxy solution.

5.1 Area of Use

The purpose of our point-to-point proxy solution is to facilitate communication between two nodes separated by a DIL network. Military networks typically consist of several interconnected tactical networks and nodes. Within each tactical network, the network conditions normally are satisfactory, while networks spanning between the different tactical networks may have the DIL characteristics. Consider for example a military patrol consisting of a vehicle and a group of soldiers. Within the tactical network consisting of the vehicle and the soldiers, the network conditions are good. However, the communication link to the Headquarter has the DIL characteristics. A proxy pair can then be deployed at the vehicle and at the Headquarters to facilitate communication over the DIL network. Figure 5.1 illustrates this example. This concept, where a local network is connected back to existing communication infrastructure, is referred to as a *reach-back link*.

While use cases similar to the previously mentioned example are the primary area of use for our proxies, they can be deployed between any two nodes in a network to facilitate communication over a DIL network.

5.2 Design of Solution

In previous chapters, we have argued that all optimization techniques should be placed in proxies to retain interoperability for COTS applications, as well as to break their end-to-end dependency. Therefore,

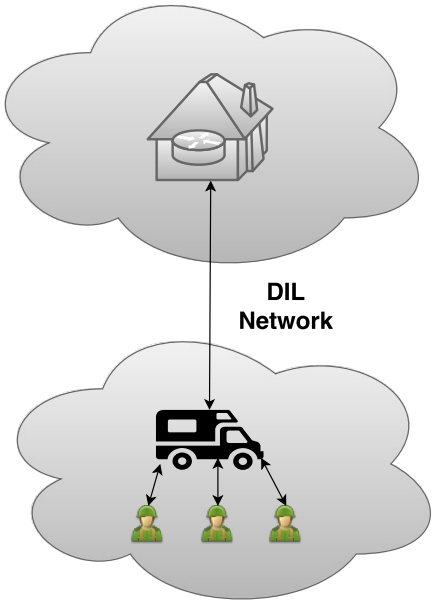


Figure 5.1: Example area of use for point-to-point proxies

our design involves deploying a proxy pair to facilitate Web communication over a DIL network. The idea is to deploy the proxy pair in two different locations separated by a DIL network. Through a locally deployed proxy, Web applications can then proxy all their data communication. The proxy then applies different optimization techniques and forwards requests over a DIL network to the other proxy and finally return a response. Ideally, the proxy should be deployed as close to its intended users as possible, as the communication between an application and its proxy is not subject to any optimization for DIL environments. Since we design the proxies accept HTTP requests, they can support any applications that use HTTP. Besides, the solution is designed to be bi-directional, meaning that requests can originate from either side.

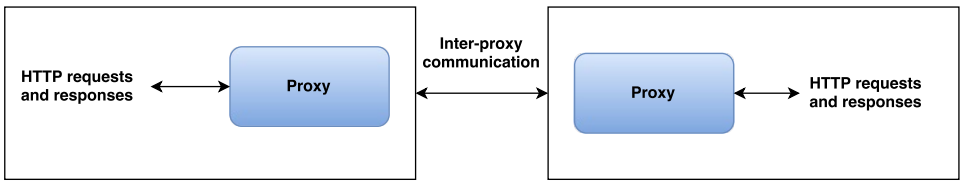


Figure 5.2: Solution concept

5.2.1 Design of Proxy

A deployed proxy is designed to accept arbitrary HTTP requests, possibly originating from multiple clients, and forward them to the other proxy as seen in fig. 5.2. The communication between a proxy

pair is subject to optimizations. The proxies are therefore designed to support the optimization techniques we have identified. Those are primarily concerned about using different transport protocols as the inter-proxy communication, as illustrated in fig. 5.3. The purpose of this is to give recommendations about the usage of transport protocols in DIL networks. Which protocol to use as the means of inter-proxy communication is therefore designed to be easily configurable at start-up by the users of the proxy.

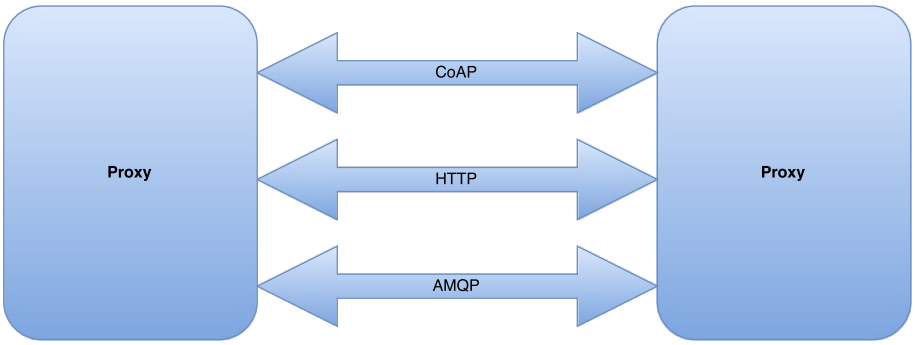


Figure 5.3: The proxies were designed to support multiple protocols for inter-proxy communication

5.3 Choosing a Framework

Requirement 1 implies creating a HTTP proxy which accepts HTTP requests, forwards them, and finally returns a HTTP response. We identified some approaches to do this:

1. Build a HTTP proxy from scratch ourselves.

2. Use an existing HTTP proxy.

Building a HTTP proxy ourselves would allow us to customize our solutions as we wanted, but would require a considerable amount of implementation efforts. Therefore, we concluded that best use of our resources was to use an existing configurable proxy. Using an existing solution allows us to focus on the optimization techniques, rather than on the specific low-level details of HTTP. There are numerous HTTP proxies available for use, for example, Nginx[38] and Squid[39]. Requirement 8 states that the solution must support different communication protocols between proxies. Because of this, we looked for software that could easily map a HTTP message to other protocols. Based on recommendations from the community at FFI we found the Apache Camel framework.

5.3.1 Apache Camel

Apache Camel is an open source Java framework developed by the Apache Software Foundation for rule-based routing and mediation [40]. It has a wide range of use-cases and focuses on making integration between different enterprise communication systems easier. It supports a large set of different communication transports (transport protocols). We chose to use Apache Camel as our HTTP proxy due to its simplicity and support for various transport protocols.

Routing is a central concept in Apache Camel and consists of defining a *from route*. The from route is an endpoint from which Camel consumes messages. Messages received by Camel are converted into generic Camel *exchanges*. *Processors* can then be invoked on these exchanges, which allow for modification of message headers and payload. After consuming a message, Camel can forward the message to a *to route*, which can be an application running somewhere else. When a response is received, Camel can invoke a new set of processors on the message before it is finally returned to the origin. Figure 5.4 shows the concept of routing in Camel.

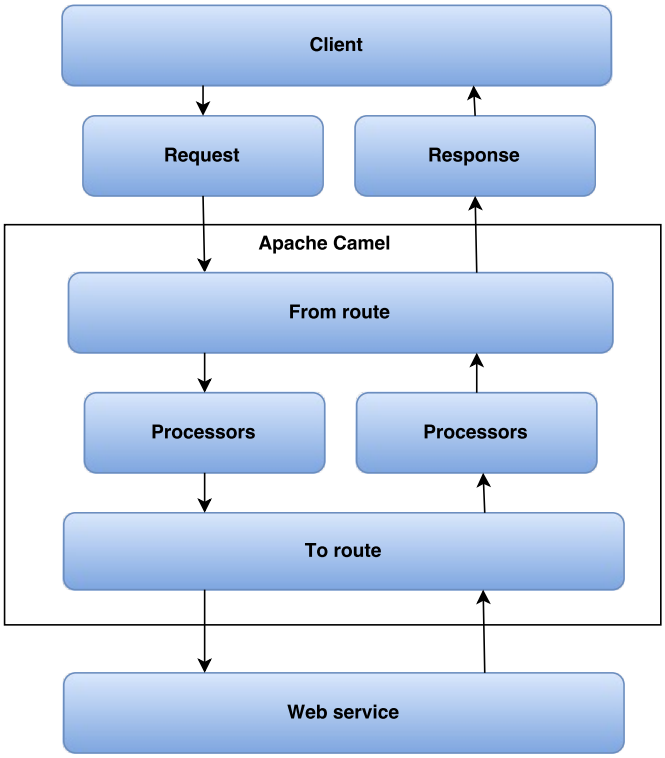


Figure 5.4: Example of a Camel route

To consume and produce messages from different protocols, databases and other sources of messages, Camel offers a set of *components*. These can be considered as factories for endpoint instances. For example, the AMQP component allows Camel to route messages us-

ing the AMQP protocol. Camel includes numerous components and is designed to support user-written components as well.

5.4 Implementation

The proxy is implemented as a Java 1.8 application using the Apache Camel framework. A large part of the implementation is concerned about reading user configuration and setting up *routing rules* for Camel. Information about the source code of the proxy is listed in chapter D of the appendix and table 5.4 lists the software used as part of the implementation.

We can divide the implementation into four stages:

1. Reading and parsing user configuration.
2. Initializing Camel components.
3. Setting up routes.
4. Runtime.

5.4.1 Parsing Configuration

The first stage involves reading a user provided configuration file. Details about the configuration are explained in section 5.5.1.

5.4.2 Initializing Components

Depending on which protocol the user has selected for usage as inter-proxy communication, at startup the respective Camel component is initiated and added to the Camel context. Due to the time available, we did not implement support for all of the recommended protocols from the last chapter. The prototype proxy currently supports the protocols HTTP, AMQP, and CoAP. However, we design the proxy to be easily extendable to include additional protocols in the future.

HTTP Component

We made use of the Camel component Jetty to consume and produce HTTP requests. The component is based on the Jetty Web server[41] and is used for two purposes. One of them is to consume HTTP requests from applications. The other is to, if HTTP was configured as the selected protocol, consume and produce HTTP messages as part of the inter-proxy communication.

AMQP Component

Apache Camel's AMQP component supports the AMQP 1.0 protocol using the JMS Client API of the Qpid project. JMS is a Java Message Oriented Middleware for sending messages between two or more clients. In the proxy component initialization phase, the AMQP component is initialized to connect to the configured message broker. Besides, the request timeout value of an AMQP request is set either to the default value of 20 seconds or the configured value.

CoAP Component

At the time of writing this thesis, there is no Camel component available for the CoAP protocol. Therefore, we implement our own custom component, supporting the transport of CoAP messages. Section 5.6 presents this component in detail. The component utilizes Californium, which is a Java framework supporting the CoAP protocol [42]. Californium is open source and is part of the IoT ecosystem of Eclipse. The component is initialized with the port the CoAP server should listen for requests. Also, an optional timeout value for a CoAP request can be added.

5.4.3 Routes

A running proxy listens on two *routes*. It can either receive messages from an application or it can receive messages from the other proxy. This setup can be seen in fig. 5.5. The routing logic is different for these two cases. We define a request originating from an outside application as the *application route*, and a request originating from another proxy as the *proxy route*. We discuss these routes shortly, but first, we need to introduce what we have chosen to call the *proxy message format*. Requirement 2 says that we need to retain all the original HTTP headers from the original request. Consider if the proxy receives a HTTP request and forwards it to the other protocol using AMQP. The message itself will arrive correctly, but the original HTTP headers and method would be lost. Our approach to this was to introduce a custom *proxy message format*, which is discussed in section 5.4.4.

5.4.4 Proxy Message Format

The proxy message format was developed to retain HTTP headers and other necessary information about the request. Our solution was to wrap all messages in a JavaScript Object Notation (JSON) document and include required information as properties in the JSON document. JSON is a lightweight, text-based data format [43]. We chose this data format due to its compactness, simplicity and the wide support for libraries for generating and parsing JSON. Due to a HTTP request and response

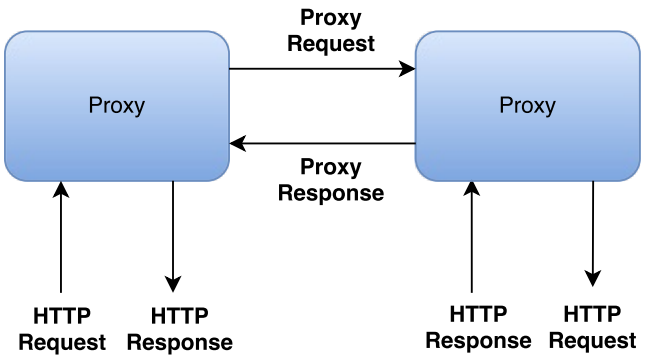


Figure 5.5: Proxy routes

having slightly different semantics, we used the same format, but with different properties for a request and response. The request format is defined in table 5.1, and the response format in table 5.2.

Field	Purpose	Required
path	The original request URL from the application. Specifies the intended final destination of the original HTTP request.	Yes
method	HTTP method of the request.	Yes
query	Query string associated with the original HTTP request.	No
headers	JSON object containing all the original HTTP headers of the request.	Yes
body	The original payload of the message.	No

Table 5.1: Proxy message request fields

Field	Purpose	Required
headers	JSON object containing the HTTP response headers.	Yes
responsecode	The HTTP response code.	Yes
body	Response body of the HTTP request.	No

Table 5.2: Proxy message response fields

Listing B.6 includes an example proxy request message. The listing illustrates a HTTP request originating from an outside application. It is a HTTP POST to the intended destination http://myservice.com, with a XML message as payload.

Listing 5.1: "Example proxy request"

```
1 {
2         "path" : "http://myservce.com:8080/",
```

```
3         "method" : "POST",
4         "query" : "?hello",
5         "headers" : {
6                 "Accept" : "Accept",
7                 "User-Agent" : "myuseragent",
8                 "Authorization" : "Basic
                     QWxhZGRpbjpvcGVuIHNlc2FtZQ=="
9         },
10        "body" : {
11                "<note>
12                        <to>Tove</to>
13                        <from>Jani</from>
14                        <heading>Reminder</heading>
15                        <body>
16                                Don't forget me this
                                     weekend!
17                        </body>
18                </note>"
19        }
20}
```

5.4.5 Application Route

The purpose of the application route is to consume HTTP requests from an outside HTTP request, transform it to a proxy request message and deliver it to the other proxy using the configured protocol. The semantics of the protocol specific routes are explained in section 5.4.7. When a response from the other proxy is received, it is returned to the application which made the request. The route consists of the following steps:

1. Defining a HTTP endpoint to consume HTTP requests from. This is read from the configuration which specifies which hostname and port to listen on.

2. Consume HTTP request from an outside application

3. Apply the *ProxyRequestPreProcessor*. This processor converts the message into a Proxy Request Message.

4. If compression is enabled, compress the entire message.

5. Forward the request to the other proxy using the configured transport protocol.

6. Receive a response from the other proxy.
7. If compression is enabled, de-compress the message.
8. Restore the HTTP response from Proxy Response Message.
9. Return the response to the application.

5.4.6 Proxy Route

The purpose of the proxy route is to listen for messages from the other proxy, de-serialize it, and deliver it to its intended receiver. When a response is received, transform it into a Proxy Response Message and return it to the other proxy. The route consists of the following steps:

1. Defining an endpoint depending on the configured protocol.
2. Consume requests from the other proxy.
3. If compression is enabled, de-compress the message.
4. Transform the message into the original HTTP request.
5. Forward the HTTP request to its intended destination.
6. Receive a HTTP response from the intended destination.
7. Transform it into a Proxy Response Message.
8. If compression is enabled, compress the message.
9. Return the response to the other proxy.

5.4.7 Protocol Specific Routes

One of the requirements for the proxy solution was that it should be easily extendable to support possible optimization techniques and transport protocols identified in the future. To realize this, we designed the setup of routes as generic as possible. Each implemented protocol derive from the abstract class DilRouteBuilder and implements two required methods. Listing 5.2 shows the DilRouteBuilder class. At initialization, to easily include additional optimizations, preprocessors and postprocessors can be added to the routes.

Listing 5.2: "Abstract class that all protocols derive from"

```
1 public abstract class DilRouteBuilder {
2
3     private final ArrayList<Processor> preProcessors = new
          ArrayList<>();
4     private final ArrayList<Processor> postProcessors = new
          ArrayList<>();
```

```
5
6       public abstract String getToUri();
7       public abstract String getFromUri();
8
9       public void addPreprocessor(Processor processor) {
10          preProcessors.add(processor);
11      }
12
13      public ArrayList<Processor> getPreProcessors() {
14          return preProcessors;
15      }
16
17      public void addPostProcessor(Processor processor) {
18          postProcessors.add(processor);
19      }
20
21      public ArrayList<Processor> getPostProcessors() {
22          return postProcessors;
23      }
24  }
```

On startup, depending on which protocol the user has configured for inter-proxy communication, the endpoint defining the interface between the proxies is defined. Table 5.3 lists example endpoints of a deployed proxy located at the IP address 192.168.10.10. The address 0.0.0.0 means that the proxy binds to all known network interfaces.

Component	**From endpoint**	**To endpoint**
HTTP	http://0.0.0.0:3001 proxy	http:// 192.168.10.10:4001/proxy
AMQP	amqp:queue:uniquename1	amqp:queue:uniquename2
CoAP	coap://0.0.0.0:3001/proxy	coap:// 192.168.10.10:4001/proxy

Table 5.3: Example endpoints for a deployed proxy.

The following paragraphs discuss the protocol specific routing.

HTTP Route

Two HTTP endpoints are defined if HTTP is configured as the inter-proxy communication protocol. The first is used to consume from, while the second is used to produce to. From the user provided configuration, the *hostname* and *port* are retrieved, and the proxy starts listening on this endpoint. Note that the HTTP component is also used to listen for requests from other applications. Therefore, only requests with a URI starting with a *proxy* prefix will be treated as an incoming proxy

message. In the same way, the produce endpoint is defined. The target hostname of the other proxy is retrieved from the configuration, and the *proxy* prefix is appended.

Including a message body with some of the HTTP methods like GET and OPTIONS have no semantic meaning according to the HTTP specification [17]. However, for inter-proxy communication a proxy message must be included in the request even if the original request was without a message body. Therefore, all HTTP requests between the proxies are designed to be a POST request in order to include a message body. Upon receiving a HTTP request from another proxy, the original HTTP method is restored before the request is delivered to the destination application.

AMQP Route

AMQP messaging is based on the concept of queues. For the routing of messages between the proxies, we define two queues. One queue for incoming messages to one of the proxy, and one queue for incoming messages to the other proxy. A proxy then consumes messages from its incoming messages queue and produces to the queue for incoming messages of the other.

CoAP Route

The CoAP route is similar to the HTTP route. It listens on the provided hostname and port and produces messages to the configured hostname of the other proxy.

5.4.8 Dealing with Errors

If an error occurs during the routing of a message, for example, a timeout exception, the default Camel error handling is to propagate the error back to the requester. One of our requirements is that the proxy should be able to deal with disconnects. Therefore, we need to handle exceptions that occur during routing in a more elegant way. Note that this applies to the routing between the proxies, the *proxy route*.

We implement this by using the *DeadLetterChannel* error handler rather than the default error handler. The DeadLetterChannel allows us to configure the redelivery policy according to the configuration of the proxy. The policy can either be set with an exponential delay or with a fixed delay. Finally, the maximum number of redelivery attempts is set. This number can be set to infinity.

5.4.9 Runtime

In the running stage, the proxy listens on the defined routes and forwards requests according to the previously configured routes. All

requests passing through the proxy are logged to the console.

5.5 Functionality

The proxy prototype is packaged as a JAR file and can be started from the command line as seen in listing 5.3. The path to a valid configuration file must be passed as a command line argument.

Listing 5.3: "How to start the proxy"

```
1 java -jar proxy.jar configfile.conf
```

5.5.1 Configuration of Proxy

The configuration of a proxy is done by passing a configuration file as an argument to the proxy at startup. In the configuration, the user can specify settings such as which protocol to use for inter-proxy communication and compression settings. We use the typesafe[44] configuration library to parse configuration files. The supported configuration options of the proxy are listed in chapter A in the appendix.

Listing 5.4 displays an example configuration of a proxy. The proxy is configured to listen on port 3001 for messages from applications and forward them using the AMQP protocol. Messages sent to the other proxy are set up to be sent uncompressed. At initialization, the proxy connects to the broker at the given location. It will consume messages on the given *consumeQueue* and produce messages to the *produceQueue*.

Listing 5.4: "Example proxy configuration file"

```
1  proxy {
2    useCompression = false
3    protocol = "amqp"
4    hostname = "0.0.0.0"
5    port = 3001
6    timeout = 40000
7    targetProxyHostname = "192.168.11.10:4001"
8  }
9
10 amqp {
11   produceQueue = 4001
12   consumeQueue = 3001
13   brokerConnectionUri = "amqp://vetur:5672"
14 }
```

5.5.2 Proxy Setup

To enable the applications to tunnel all their HTTP traffic through our proxy, we need a way to set a proxy without altering the applications themselves. Fortunately, Java provides mechanisms to deal with proxies [45]. We configured the Java Virtual Machine (JVM) to get the applications to tunnel all HTTP traffic through our proxy. This is done by setting properties to the JVM:

Listing 5.5: "Setting a proxy on the JVM"

```
1 java -Dhttp.proxyHost=localhost \
2 -Dhttp.proxyPort=3001 \
3 -Dhttp.nonProxyHosts= \
4 -jar target/client.jar
```

In listing 5.5, the application **client.jar** is started, and all HTTP traffic will go through the proxy server at localhost on port 3001.

5.6 Custom Camel CoAP Component

Since no Apache Camel CoAP component was available at the time of implementing this proxy, we developed our own. We followed a tutorial[46] for writing custom components by Apache Camel and made the component available as open source at GitHub [47].

The component exposes a Camel endpoint, which can be added to the Camel Context at startup. This allows the user to use CoAP to both consume and produce messages. The CoAP component extends Camel's DefaultEndpoint class and implements two methods: CreateProducer and CreateConsumer.

5.6.1 CoAP Producer

The purpose of the Camel CoAP producer is to send CoAP messages to a CoAP server. When Camel routes a message using the CoAP component, the producer is invoked with a generic Camel Exchange message. The producer then retrieves the payload from the exchange and sends it to the CoAP server specified at component initialization. We use the Californium[42] implementation of CoAP to send and receive CoAP messages.

When a CoAP response is returned from the CoAP server, the producer copies the returned message into the response of the Exchange. The CoAP message response code is mapped into a HTTP status code according to guidelines for mapping HTTP to CoAP [48]. For the purpose of creating the proxy prototype, we only support mapping a subset of all possible mappings.

If a response is not received within a specified timeout, the producer sets a HTTP Timeout status on the exchange. When Camel continues the routing process, this will, in turn, invoke Camels error handler allowing the redelivery mechanism to start retransmitting.

5.6.2 CoAP Consumer

The purpose of the Camel CoAP consumer is to consume CoAP messages and convert them into Camel Exchanges. At startup of a Camel application using the CoAP component, a Californium CoAP server is started. It starts listening for CoAP messages on a user-specified port. When a CoAP message is received, the message is converted into a Camel exchange. Then Camel continues the routing of the message according to the route configuration. In our proxy implementation, messages are forwarded to a HTTP application. When a response is received, the CoAP consumer converts the message into a CoAP response. The HTTP status response code is mapped into a CoAP response code using guidelines for mapping HTTP to CoAP [48].

5.7 Software Used

The proxy is implemented in Java using the Apache Camel framework. Table 5.4 lists the software versions used in the implementation.

Software	**Version**
Java	1.8
Apache Camel	2.16.1
camel-amqp	2.16.1
camel-jetty	2.16.1
javax.jms-api	2.0
Californium	1.0.0
typesafe	1.3.0

Table 5.4: Software used in the proxy implementation

5.8 Summary

In this chapter, we presented the design and implementation details of the proxy. We introduced the Apache Camel Framework and how we utilized it to compress messages and facilitate mapping of HTTP messages to other protocols. Furthermore, we presented how the proxy could be used and configured by users. We also introduced a custom open source Camel component we implemented to map messages between HTTP and CoAP. In the next chapter, we present how the implemented proxy solution is tested and evaluated.

Chapter 6

Testing and Evaluation

In this chapter, we outline how the testing and evaluation of the proxy is performed and present the results obtained. The goal is to validate that the proxy fulfills the premises and requirements and to measure any possible improvements (or deteriorations) of the performance of Web services. Based on the measurements we then give a recommendation about which adaptations to make in different types of DIL networks. Since the proxy is being developed as a prototype for military usage, we use test scenarios that resemble actual military usage. We also included one civilian usage based on Enhanced Data rates for GSM Evolution (EDGE) since civilian mobile phone technology is currently being considered for military use both by NATO and several nations. For the purpose of testing, we develop two sets of test applications, one W3C Web service, and one RESTful Web service. These applications are then used to test the proxy in networks with different DIL characteristics.

We start this chapter by introducing different types of DIL networks we can encounter working with tactical networks. Then we present how these networks can be emulated using the Linux network traffic tools before we introduce evaluation tools and the two test applications. Next, we put the proxy to the test in an unlimited network to verify that it behaves correctly. We call this the function test. Then we start introducing the DIL characteristics into the tests and measure if we can improve the performance by using proxies. We introduce the *disconnected* and *intermittent* aspects first before we test the proxy in six different *limited* networks. We also test in a setup using actual military communication equipment. This testing was done to validate the results from the software emulated networks.

Finally, we discuss the results, their underlying causes, and which implications they have. Ultimately, we give a recommendation about the usage of proxies in DIL networks.

6.1 Types of DIL Networks

Military communication can occur over a wide range of different technologies and environments. An infinite number of possible network combinations exist, so we have chosen to focus on five different network types identified by the task group IST-118 for DIL-testing [16]. The networks were identified because they represent typical networks typically found in military communication. These include Satellite Communication (SATCOM), Line of Sight (LOS), Combat Net Radio (CNR) and WiFi. WiFi is divided into two types, one indicating operation in the "sweet spot" and one in the edge of the network. Some communication technologies, such as satellite communication, are characterized by long communication delay while others may be by their low data rate. An overview of selected military communication technologies can be seen in fig. 6.1.

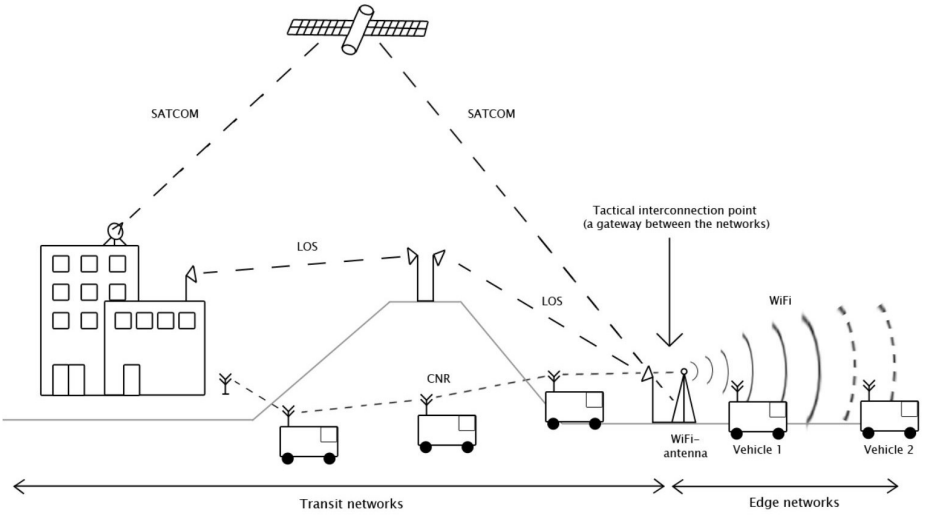


Figure 6.1: Overview of tested networks (from [49])

We also investigated Long-Term Evolution (LTE), commonly known as 4G, a network technology which has become in widespread use in the latest years. The reason for including LTE in addition to the ones from IST-118 is that the Norwegian Defense is looking into the possibility of using LTE. This fact makes it interesting for us to investigate the performance of Web services in this type of network as well. However, LTE has gotten so fast and reliable that it is not really relevant from a DIL perspective. We therefore instead looked into EDGE, which is used as a fallback in geographical areas where LTE and 3G are not available. Of the networks we evaluate for, EDGE is the only one with asymmetrical down- and upload speed: 50 kbps up and 200 kbps down [30].

Table 6.1 summarizes the identified networks and their propererties.

Network	Data Rate	Delay	PER
SATCOM	250 kbps	550 ms	0 %
LOS	2 mbps	5 ms	0 %
WiFi 1	2 mbps	100 ms	1 %
WiFi 2	2 mbps	100 ms	20 %
CNR	9.6 kbps	100 ms	1 %
EDGE	50 kbps up/200 kbps down	200 ms	0 %

Table 6.1: Different network types

6.2 Testing and Evaluation Tools

To evaluate how using the proxies impacts the performance of Web services in DIL environments, we needed some way of emulating limited and constrained networks. Obviously, we would have got the most realistic test environment by testing "out in the field" ourselves. However, this would require a considerable amount of effort, and it would be difficult to reproduce the exact same environment and test results. Therefore, we chose to emulate DIL networks by using a setup consisting of interconnecting two machines through a third machine. The third machine is used as a link emulator and controls the network traffic passing between the two other machines. This setup is illustrated in fig. 6.2. To emulate DIL networks, the link emulator uses components in the Linux kernel to control the flow of the network traffic going through it.

Additionally, we performed experiments using actual military communication equipment. The purpose was to confirm the results from the emulated network tests. These experiments are presented in section 6.8.

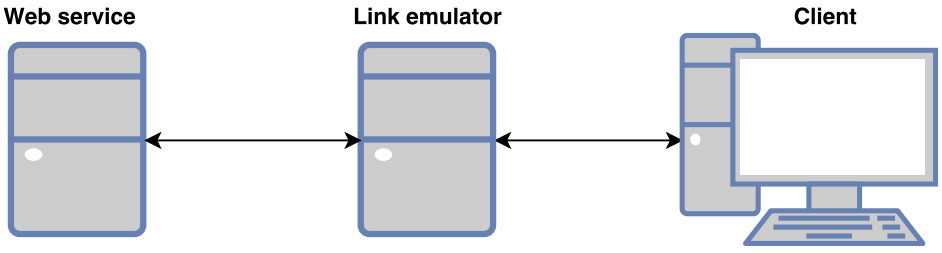


Figure 6.2: Test setup

6.2.1 Linux Network Traffic Control

The Linux kernel offers a rich set of tools for managing and manipulating the transmission of packets, referred to as network traffic control. The central concept in traffic controlling is the concept of queues, which collects entering packets and dequeues them as fast as the network

hardware can accept them. **tc** (traffic control) is a Linux program to configure and control the Linux kernels network scheduler. The Network Emulator (NetEm) is an enhancement of the traffic control facilities that allows us to control delay, packet loss and other characteristics of packets outgoing from a selected network interface [50]. These tools together enable us to emulate the networks listed in table 6.1.

How to configure NetEm and the Linux traffic control tools is outlined in the following paragraphs.

Emulating Network Delays

With NetEm, we can emulate delays on outgoing packets on a specific link. In listing 6.1, we show an example configuration where a fixed delay of 100 ms to all packets going out of local Ethernet connection.

Listing 6.1: "Emulating the delay of outgoing packets"

```
1  tc qdisc add dev eth0 parent 1:1 handle 10: \
2    netem delay 100ms
```

Emulating the Data Rate

To emulate different data rates, we use a part of the Linux traffic control tool called Token Bucket Filter (TBF). TBF can be used to shape network traffic and ensures that the configured rate is not exceeded. It shapes traffic based on the concept of *tokens* and *buckets*. Tokens are generated at the desired data rate and are collected into buckets, which have a maximum number of tokens they can store. When TBF receives a packet, it checks if it has a sufficient number of tokens to send the packet. If not, it is deferred, thus causing an artificial delay for the packet.

Listing 6.2 shows an example configuration where we configure the maximum data rate of 50 kilobits per second. The burst value is the size of the bucket in bytes and describes the maximum amount of bytes that tokens can be available for instantaneously. The limit is the number of bytes that can be queued waiting for available tokens.

Listing 6.2: "Emulating the data rate"

```
1  tc qdisc add dev eth0 handle 1: \
2    root tbf rate 50kbit burst 15000 limit 15000
```

Emulating the Corruption Rate

The corruption rate allows us to insert random data into a chosen percent of packets. In listing 6.3, we show how the corruption rate can be set to 20 percent.

Listing 6.3: "Emulating the corruption rate"

```
1  tc qdisc add dev eth0 parent 1:1 handle 10: \
2    netem delay 100ms corrupt 20%
```

6.2.2 iPerf 3

iPerf 3 is a tool for measurement of maximum achievable date rate on a network [51]. Since we in this thesis are *emulating* different DIL networks, it is critical that the emulation is as correct and realistic as possible. Misconfiguration or wrongful emulation could cause us to draw invalid conclusions. IPerf was one of the recommended tools in a previous study which explored different network monitoring tools for use in limited capacity networks [52].

To confirm and validate our network emulations we use iPerf 3 alongside the Linux tool *ping*. The measurements are performed between the machine hosting the client and the machine hosting the Web service. They are performed before starting the test cases so that the network traffic it generates do not interfere. The measurements are included in chapter C of the appendix.

6.2.3 Wireshark

Wireshark is a packet analyzer and allows for performing network usage analysis [53]. As an example, this tool allows a user to see all IP packets sent from a machine over the Ethernet interface.

When performing the testing, we use Wireshark to monitor the network traffic on the machine hosting the client and its proxy. This is called a packet capture and allows us to investigate the behavior of the evaluated protocols in the different types of networks. In particular, we use it to see how many packets that are sent, as well as the total number of bytes that are sent over the network.

6.3 Test Sets

For each test network, we perform tests with both a W3C Web service test case and a RESTful Web service test case. Each test set consists of a Java client and a Web service. Information about the source code of the test applications is included in chapter D of the appendix.

In this thesis, we look into ways of improving the performance of Web services. The purpose of the test sets is to imitate network traffic generated by real Web services. From evaluating the performance increase or decrease of the test services when using the proxies, we can deduce that this applies to applications in actual use as well. As performance indicators, we use the average Round-Trip Time (RTT) as perceived by the client application and how the network is utilized.

6.3.1 NFFI W3C Web Service

For the purpose of testing W3C Web service applications, we created a mock system which allows a client to request a service to report positions of friendly forces. The position report uses the NATO Friendly Force Information (NFFI) format, which has an associated XML schema with it. We refer to this test case as the "NFFI" test case.

One test run is illustrated in fig. 6.3 and consists of a client making a HTTP POST request to the Web service. Associated with the request is an XML payload which tells the Web service which operation to invoke. In our case, the service then returns an XML message containing a large number of positions in the NFFI format.

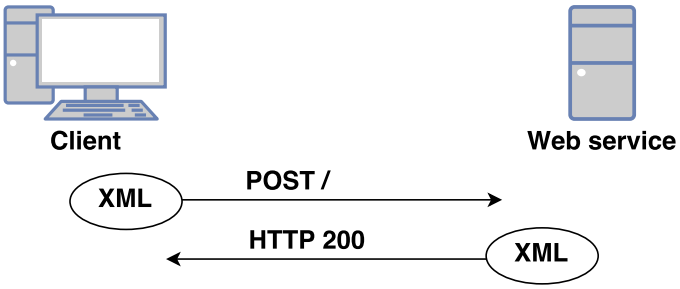


Figure 6.3: NFFI Web service

Request URI	HTTP Method	Bytes sent	Bytes received
?wsdl	GET	192	3527
?wsdl=1	GET	194	4331
/	POST	829	40631
Total:	**3**	**1215**	**48489**

Table 6.2: NFFI Web service HTTP requests

6.3.2 RESTful Car Service

We originally wanted to use a service resembling a military scenario like the NFFI service. However, no such applications were easily available for testing at the time of writing the thesis. For the purpose of testing RESTful services, we chose to develop a small example service ourselves. The RESTful Car service is a service keeping order of cars in a car registry. It is a simple system keeping track of the registration number and type description of multiple cars. We refer to this test case as the "Car system" test case.

The service exposes an Application Program Interface (API) which offers different actions to manage the car system. Clients can invoke these operations by using HTTP requests and utilizing the associated HTTP method to indicate what to do with a resource. Since RESTful

services are payload agnostic, we chose JSON to represent the data being sent between the server and the client. An example of usage of the Car system is illustrated in fig. 6.4.

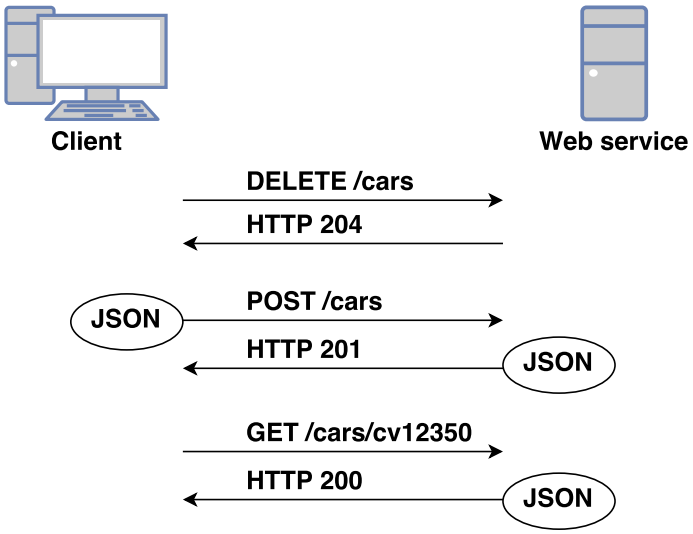


Figure 6.4: Example usage of the REST Car system

Each test run of the Car system consists of a client sequentially invoking the server with different API requests, listed in table 6.3. The most common HTTP-methods GET, PUT, POST, and DELETE are all part of the tests. To test that custom HTTP headers are retained when the HTTP messages are forwarded through the proxy, both the client and service set a custom header. When a request or response is received, the application validates that the custom header is present.

Request URI	**HTTP Method**	**Bytes sent**	**Bytes received**
/cars	DELETE	233	243
/cars	POST	293	353
/cars	POST	298	358
/cars	POST	294	354
/cars	POST	299	359
/cars	POST	296	356
/cars	GET	198	538
/cars/id	GET	209	348
/cars/id	PUT	309	243
/cars/id	GET	209	354
/cars/id	DELETE	244	243
/cars/	GET	198	495
Total:	**12**	**3080**	**4244**

Table 6.3: REST Car system HTTP requests

6.3.3 Test Applications Summary

A test run of both the NFFI test case and the Car system test case consists of sequentially sending HTTP requests to their respective Web service. However, they have some fundamental differences. The Car system test involves running 12 HTTP requests while the NFFI request only invokes three. Also, the payload of each request and response is generally much smaller for the Car system tests. Moreover, the response message of the third request of the NFFI service is significantly larger than any other request or response.

6.4 Test Setup

Both test sets consist of one client and one Web service, where the client would request the service for some sort of action. The client is running on one computer while the Web service is deployed in the Glassfish 4 application server on another computer. The specifications of the machines used in the testing are listed in table 6.4.

Machine	**Client machine**	**Web service machine**	**Link emulator**
Model	Asus UX 31A Notebook	HP EliteBook 6930p	HP Compaq Elite 8000
OS	Debian 8.2	Ubuntu 14.04	Ubuntu 14.04
Kernel	3.16.0-4-amd64	3.13.0-79-generic	3.19.0-25-generic
CPU	Intel i7 @ 1.90GHz	Intel Duo T95550	Intel Quad Q9500 @ 2.83GHz
Cores	4	2	4
Memory	4 GB	4 GB	12 GB
Network hardware	ASIX AX88772 USB 2.0	82567LM Gigabit	82567LM-3 Gigabit
Network interface capacity	100 Mbit/s	1 Gbit/s	1 Gbit/s

Table 6.4: Machines involved in the testing

6.4.1 Network Setup

The client and Web service machines are connected to each other through a third computer acting as a link emulator. The link emulator machine has two Ethernet network cards and interconnects the two other machines. This setup can be seen in fig. 6.5. For the link emulator

to forward IP packets back and forth between the client and server, IP forwarding is enabled in the kernel.

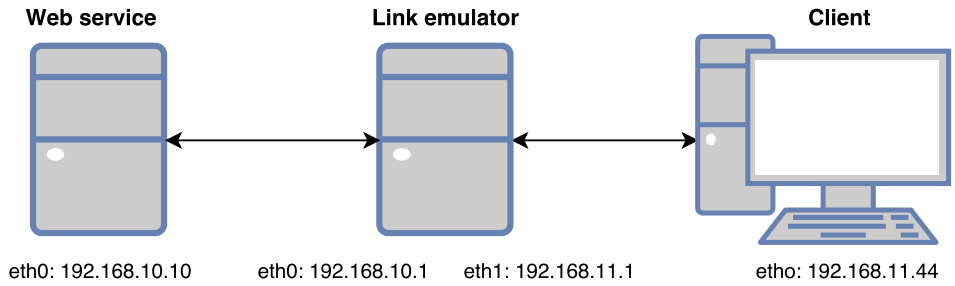


Figure 6.5: Network used for testing

The client and Web service machine are assigned an IP address in two different subnets. This is done by the Linux network interface administration program *ifconfig*. In listing 6.4, the client machine is assigned the IP address 192.168.11.44.

Listing 6.4: "Setting the IP address a network interface"

```
1 ifconfig eth0 192.168.2.1 up
```

After setting up the IP addresses, we configure the routing so that the kernel knows where to route the network traffic. In this case we want all traffic to go through the link emulator. In listing 6.5 we configure all IP traffic bound for the subnet 192.168.10.0/24 to be routed through the link emulator with the IP 192.168.11.1.

Listing 6.5: "Configuring routing rules"

```
1 ip route add unicast 192.168.10.0/24 via 192.168.11.1
```

After configuring the IP address and setting up IP routing on both the client and Web service machine, we start emulating different DIL networks.

Emulating Different Types of Networks

Since all network traffic passes through the routing machine, we can control the flow of IP packets here. As we previously discussed, we make use of the network traffic control tools of Linux. For each network configuration, a bash script is run. This script configures the network interface to get the correct network behavior. Both interfaces are configured so the network is symmetrical in both directions, except EDGE which has asymmetrical data rates. The bash scripts used to emulate the DIL networks are included in chapter B of the appendix.

6.4.2 Test Execution

The tests were executed with the setup illustrated in fig. 6.6. Machine 2 hosts a test client and a proxy, while machine 1 hosts a proxy and the Web services. The Web services are deployed in a Glassfish 4 application server. To facilitate AMQP communication between proxies, a message broker is also run at the Web service machine. In our tests we use version 5-13.2 of Apache ActiveMQ [54], an open source message broker supporting messaging protocols like AMQP and MQTT.

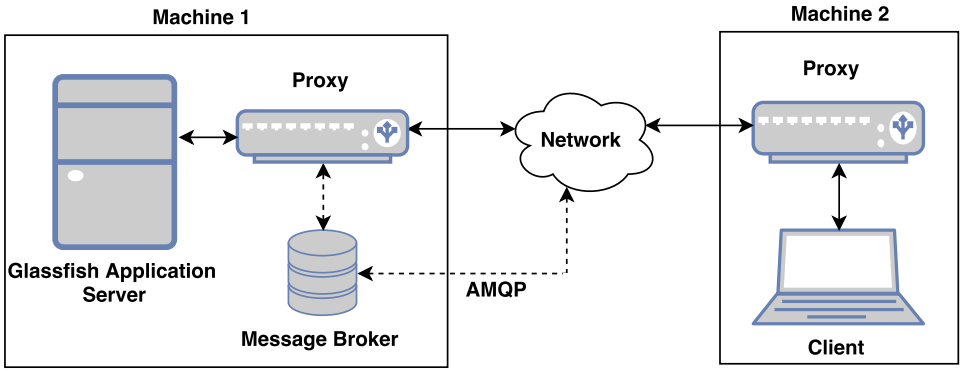


Figure 6.6: Test setup

Each test execution is initiated from a Java client on machine 2. We then measure how long it takes to complete all requests part of the test, thus allowing us to measure the Route Trip Time (RTT) as perceived by the client. All tests are performed at least 10 times to calculate the mean, standard deviation, and variance. The results used to create the graphs presented in the following sections are included in the appendix, chapter C.

Moreover, for each test we perform a packet capture with Wireshark of one sample test run. The capture allows us to get an indicator of the network usage of the proxies and the different inter-proxy communication protocols. It is worth noting that this was only performed on one test case for each test. Thus, any variance between test runs may not have detected. This is especially true for the networks with a chance of packet errors. However, it gives us an idea of the network traffic during that test.

The tests are performed with the following parameters:

- Without and with proxies.
- GZIP compression on/off. When testing without proxies, the messages are *not* compressed.
- The protocol used to communicate between the proxies. We refer to proxies using HTTP as the inter-proxy communication protocol as a HTTP proxy, using AMQP as an AMQP proxy and so on.

6.5 Function Tests

The first part of the testing is performed without any intended limitations to the network. The objective of the function tests is to validate that the proxy fulfills the functional requirements we sat for our proxy:

- Receive and forward HTTP requests.
- Retain HTTP request and response headers.
- Support GZIP compression of payload.
- Support usage of different transport protocols between the proxies.
- Work with security mechanisms.

In addition, the results from the function tests can be used to benchmark against other tests. We run both with and without proxies, allowing us to investigate the overhead associated with the usage of proxies. We use the test setup described in the last section, although without any intended limitations of the network. The tests are performed for both test applications and repeated multiple times to get the average RTT.

6.5.1 Results

Both the NFFI and Car system test cases finish successfully within the average of 200 ms when not using proxies. Figure 6.7 shows the results from the function tests and reveals the impact of introducing the usage of proxies. When using proxies, all test cases still completes successfully, but their average RTT varies depending on the protocol. The clients HTTP requests are forwarded through the proxies to the Web service, which successfully returns a HTTP response back.

We also verified that the proxy solution works with standard security mechanisms. The HTTP header added by the Car system client and Web service is successfully retained. Moreover, we verified the integrity of the payload of the NFFI test case when using the proxies and compression. Using Wireshark, we compared the HTTP response body from one of the requests in the NFFI test using proxies, with one run where proxies were not used. Diffing the two payloads, we verified that the proxy did not alter the payload in any way. This verifies that the digital signature of the payload is intact.

Together with the successful GZIP compression, the functional requirements for the proxy is therefore identified as fulfilled.

Analysis

Even in an unlimited network, we still see a significant difference between the protocols. These trends may be the same or perhaps reinforced when the protocols are used in DIL networks. Therefore, in the coming paragraphs, we investigate and discuss the possible underlying reasons for the results we obtained in the function tests.

In test cases without compression, using proxies results in a longer RTT. The longer RTT can be due to the overhead of sending requests through proxies, which includes initializing TCP connections and the time the proxies use processing requests. When using a HTTP proxy with compression, we see a decrease in the RTT for the NFFI test case. Inspecting the network traffic with Wireshark reveals the probable cause for this. Compressing the relatively large XML documents sent in NFFI test case yield a very high compression rate. The compression rates of the rather small JSON documents in the Car System test case are relatively small in comparison.

Furthermore, we observe that the transport protocol used by the proxies has a significant impact on the RTT and packets sent over the network. Table 6.5 and table 6.6 list the IP packets sent over the networks of one sample run of the two test cases. To better understand the reasons for the difference in average RTT and network usage, we are in the following sections investigating the behavior of each protocol separately.

HTTP Proxy

The HTTP proxy generally performed well. For the NFFI test case with compression, the proxy is marginally faster than when not using a proxy. The reduction of data sent and received over the network is reflected by the reduced number of IP packets. Without compression and for the Car system test case, the RTT of the HTTP proxy is marginally longer. The reason could be the overhead associated with the proxy. Using Wireshark we analyzed one sample run of the Car system test using a HTTP proxy and found the following network activities:

1. The Car system client starts invoking its first HTTP request. This is a DELETE request without a message body. Since requests are proxied through the HTTP proxy, a TCP connection between the client application and the proxy is established.

2. After receiving the request from the client, the proxy establishes a TCP connection with the other proxy.

3. The HTTP request is sent from the proxy to the other proxy. The DELETE request has now been converted to a POST request, and the message body contains the proxy message. Since the

request now has a message body, two HTTP headers have been appended: Content-Type and Content-Length. Besides, the HTTP-header breadcrumbID has been added, a header used by Camel. Including the proxy message, the size of the original HTTP request has now grown from 243 to 635 bytes.

4. A HTTP response is returned from the other proxy. It consists of two reassembled TCP segments with the total size of 974 bytes. Comparing with the response from when not using a proxy, indicates a few things. Without the proxy, the response to the DELETE request is without a message body, while with, the HTTP body contains a proxy message. In addition, the response has additional HTTP headers than the original request. For this examined response, using the proxy introduced a message overhead of 756 bytes.

5. The response if forwarded from the proxy to the client.

6. The client starts its next request and repeats the mentioned steps. However, since the TCP connections now are initialized and open, both between the client and proxy and between the proxies, the TCP connections are reused.

Using a HTTP proxy successfully forwarded messages, but introduced some overhead. HTTP headers are added and possibly duplicated, as the proxy encapsulates the original headers inside the proxy message and then adds its own headers to the HTTP request between the proxies.

AMQP Proxy

AMQP had the worst average RTT of the proxy protocols, especially for the Car system test case. As seen in table 6.5 and table 6.6, AMQP sends a lot more IP packets through the network than HTTP. Since AMQP is broker based, communication occurs through a message broker and not directly between the proxies. Using Wireshark, we dived down into the details:

1. A TCP connection between the test client and proxy is first established.

2. The proxy establishes a TCP connection with the message broker.

3. The proxy and message broker agree on an AMQP version by exchanging the AMQP protocol header.

4. Next, to forward the first request, the proxy initiates an AMQP connection. The connection initialization consists of numerous AMQP frames being sent between the proxy and message broker. It includes sending the AMQP frames Open, Begin, Attach and Flow.

First after sending these frames, the first *transfer* frame carrying the message is sent.

5. Finally, when a response is returned, both the AMQP and TCP connection between the proxy and message broker are closed.
6. The proxy returns the response back to the test client.
7. When the next test HTTP request is initiated by the client, these steps are repeated.

Although all requests are successfully forwarded, the AMQP proxy cause a significant overhead due to its complex connection procedures. For every request that is forwarded through the proxy, a new AMQP and TCP connection had to be established.

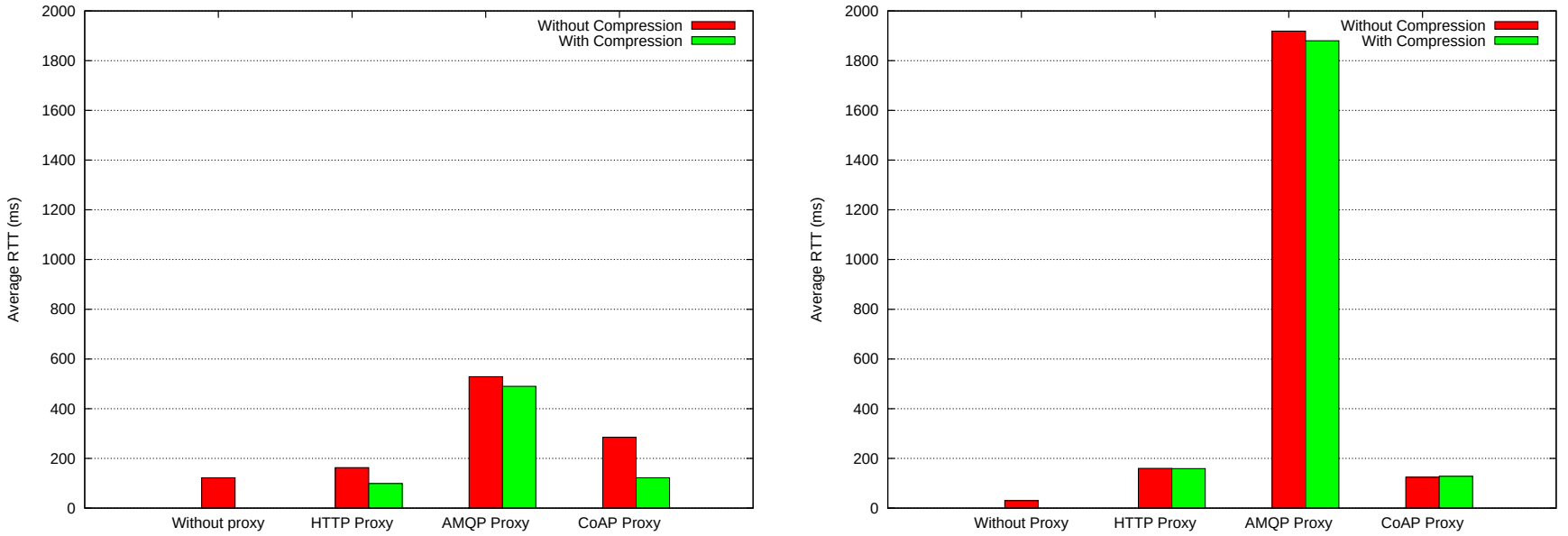
CoAP Proxy

Using CoAP as the inter-proxy communication protocol had roughly the same average RTT as the HTTP proxy, with one exception. In the uncompressed NFFI test case, it had a longer RTT and sent an unreasonable higher amount of packets. We discuss the possible reasons for this in detail in section 6.9. For the test cases not involving large messages, however, CoAP sent significantly *fewer* IP packets than the other proxy protocols. Using Wireshark we looked into the network traffic of the Car system test:

1. The test client first establishes a TCP connection with the proxy and sends the first message.
2. The proxy forwards the request in a UDP message to the other proxy.
3. The other proxy returns the response and acknowledgment in one UDP message.
4. The proxy returns the response to the test client.
5. The test client invokes a new request, and the steps are repeated.

As we see, CoAP has a more simple messaging pattern compared to AMQP and partly also HTTP/TCP. CoAP uses UDP, which is a connectionless protocol, which means that no packets have to be sent to establish a connection. For the function tests, we see that the CoAP proxy is the proxy with the least network footprint.

The function tests were done in an unlimited network, so our findings are not necessarily applicable to DIL networks. In section 6.7, we put the proxy and protocols to the test in more limited networks. However, first, in the next section we see how the proxies cope with the disconnect and intermittent aspect of the DIL.



(a) NFFI

(b) REST

Figure 6.7: Function tests - Average RTT Time for the client application.

Test	Packets sent	Packets received
Without Proxy	51	46
Proxy with HTTP	45	44
Proxy with HTTP & GZIP	13	13
Proxy with AMQP	73	94
Proxy with AMQP & GZIP	57	62
Proxy with CoAP	101	101
Proxy with CoAP & GZIP	11	11

Table 6.5: NFFI Function test - IP Packets sent and received by the client application.

Test	Packets sent	Packets received
Without Proxy	25	21
Proxy with HTTP	28	26
Proxy with HTTP & GZIP	28	28
Proxy with AMQP	180	203
Proxy with AMQP & GZIP	190	207
Proxy with CoAP	12	12
Proxy with CoAP & GZIP	12	12

Table 6.6: REST Function test - IP Packets sent and received by the client application.

6.6 DIL Tests - Intermittent and Disconnected

Intermittent and *disconnected* refers to the network connection being lost for some period, but then regained again. Disconnected refers to the loss of connection over a longer period, while intermittent is a special case of disconnected and refers to shorter disruptions. The requirements we set for our proxy were that it should:

- Handle frequent network disruptions.
- Handle disconnects over longer periods of time.

In our testing, we focus on the loss of connections for longer periods of time. The objective of this testing is to evaluate how the proxy manages disconnects. We define the success criteria to be that a client can eventually process his request after the connection is reestablished. The client's HTTP request should not be interrupted in any way, other than it taking a longer time to process the request.

6.6.1 Execution

The tests are performed over an unlimited network and for both the NFFI and Car system test. The proxy redelivery delay is configured to be with a fixed 20-second retransmission. The tests are executed by starting the test applications and then immediately removing the Ethernet cable between the client machine and the link emulator as illustrated in fig. 6.8. We then wait around 60 seconds, allowing requests to trigger timeouts and thus invoking the proxy redelivery mechanism. Finally, we connect the cable again and observe if the test application is able to finish its requests successfully.

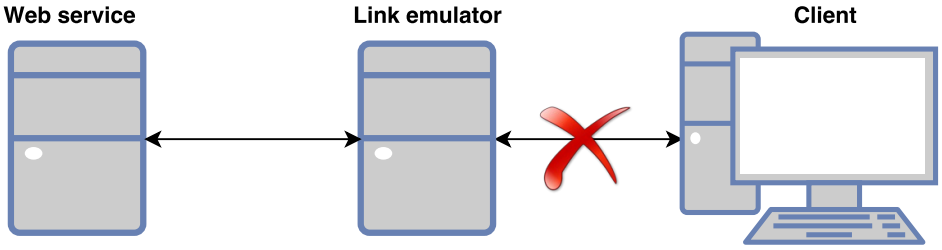


Figure 6.8: Emulating a disconnect

6.6.2 Results and Analysis

For both the REST and W3C Web service test scenarios, the results were identical. Without using proxies, the connection timed out, and the applications were unable to continue as shown in table 6.7 and table 6.8. With proxies, the connection did not time out, and the protocols retransmission mechanisms were able to continue transmission when the connection was reestablished.

Test	**Result**
Without proxy	Connection timeout
Proxy with HTTP	Success
Proxy with AMQP	Success
Proxy with CoAP	Success

Table 6.7: NFFI Web service results

6.7 DIL Tests - Limited

The third DIL characteristic, *limited*, refers to various ways a network can be constrained. The limited characteristic includes long delays, packet loss, and low data rate. In this section, we present the testing performed for the different types of networks identified in table 6.1.

Test	Result
Without proxy	Connection timeout
Proxy with HTTP	Success
Proxy with AMQP	Success
Proxy with CoAP	Success

Table 6.8: RESTful Web service results

Through this testing we evaluate how the proxy performs with regards to requirement 6, stating that the proxy should be able to:

- Handle low data rates, long delays, and high packet error rates.

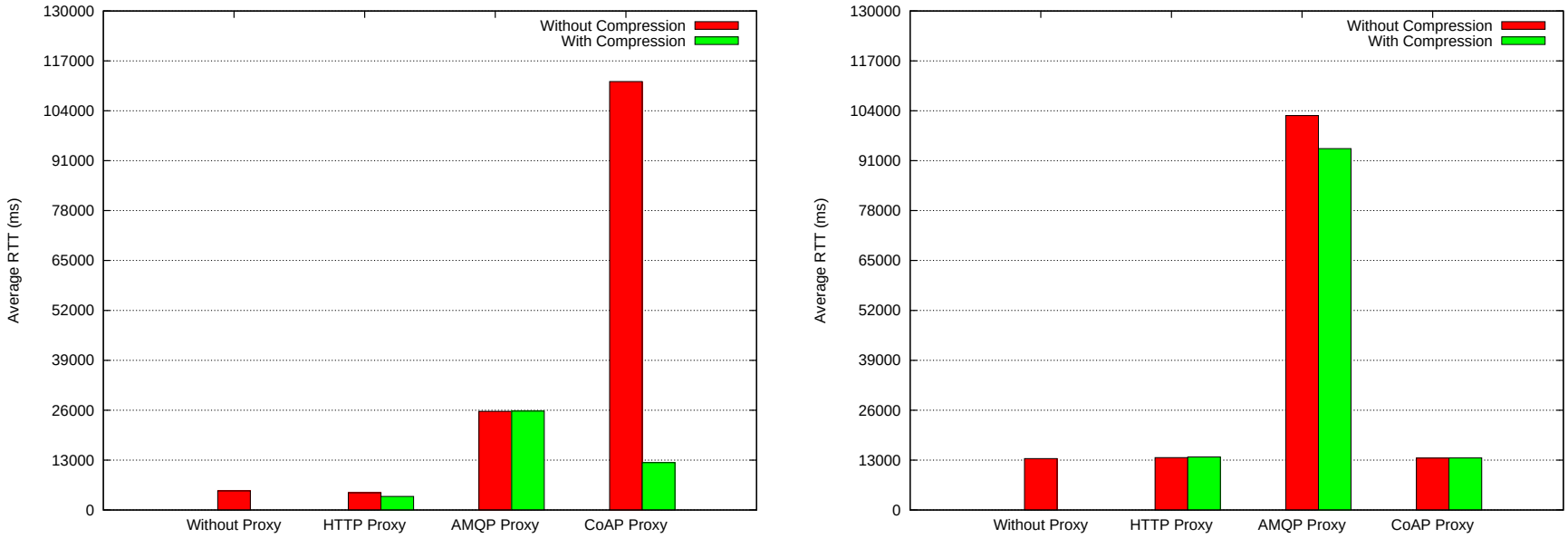
6.7.1 Satellite Communication

In this test network, we emulate SATCOM. With satellite communication, all data is relayed through a communication satellite in orbit around the earth. This type of communication is characterized by its low data rate and high delay.

Results and Analysis

From the SATCOM testing results presented in fig. 6.9, table 6.9 and table 6.10, we observe the following:

- The HTTP proxy with compression has the overall best RTT.
- With one exception, AMQP has significantly higher RTT than the other protocols. For the Car system tests with many subsequent HTTP requests, we see that AMQP triggers the sending of many IP packets. In a sample test run, the Wireshark capture revealed that AMQP sends twenty times the amount of IP packets compared to CoAP.
- CoAP struggles with large uncompressed messages of NFFI test case. For the Car system test, however, the CoAP proxy has almost equal average RTT as the HTTP proxy. A Wireshark capture during the Car system test shows that CoAP proxy sends very few of IP packets compared to other protocols.
- Compression can be of less importance in networks with high data rates and where the long delay is the limiting factor.



(a) NFFI

(b) REST

Figure 6.9: SATCOM tests - Average RTT Time for the client application.

Test	Packets sent	Packets received
Without Proxy	54	47
Proxy with HTTP	47	45
Proxy with HTTP & GZIP	16	14
Proxy with AMQP	88	102
Proxy with AMQP & GZIP	71	68
Proxy with CoAP	101	101
Proxy with CoAP & GZIP	11	11

Table 6.9: NFFI SATCOM test - IP Packets sent and received by the client application.

Test	Packets sent	Packets received
Without Proxy	27	22
Proxy with HTTP	26	25
Proxy with HTTP & GZIP	30	28
Proxy with AMQP	244	238
Proxy with AMQP & GZIP	240	240
Proxy with CoAP	12	12
Proxy with CoAP & GZIP	12	12

Table 6.10: REST SATCOM test - IP Packets sent and received by the client application.

6.7.2 Line-of-Sight

In this test scenario, we emulate LOS networks which are characterized by being a radio-based type of network with no physical obstacles between the nodes in the network. LOS has high data rate, low delay, and zero error rate.

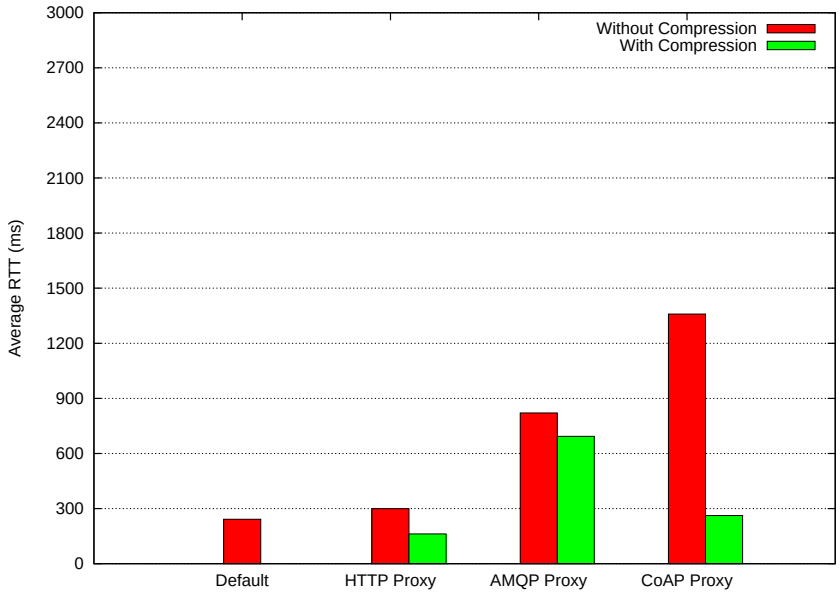
Results and Analysis

The average RTT of the LOS tests is shown in fig. 6.10. IP packets sent and received in a sample run of the NFFI and Car system test cases are listed in table 6.11 and table 6.12. The significant findings are summarized here:

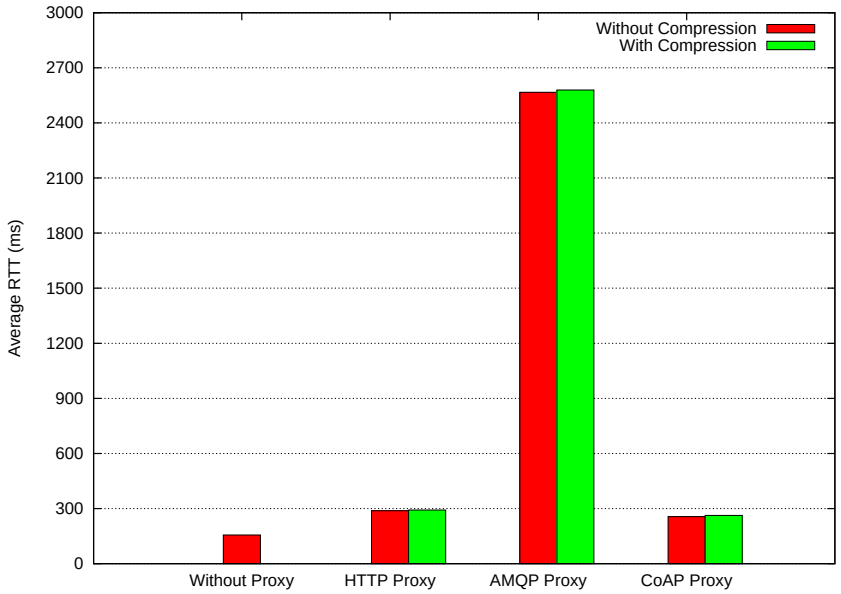
- HTTP proxy yielded the lowest average RTT in the NFFI test case, while not using a proxy had the best RTT in the Car system test. In the Car system tests, the CoAP proxy is marginally faster than a HTTP proxy.
- We observe the same trends regarding CoAP and AMQP as in the function testing. The LOS type of network is a relatively unlimited

network. The results have the same characteristics as the results from the function tests.

- For the Car system test, we see that enabling compression yields a slightly longer average RTT. The reason for this can be the time used to compress the payload is larger than the time saved by reducing the size of the message.



(a) NFFI



(b) REST

Figure 6.10: LOS tests - Average RTT Time for the client application.

Test	Packets sent	Packets received
Without Proxy	46	43
Proxy with HTTP	43	44
Proxy with HTTP & GZIP	14	13
Proxy with AMQP	68	91
Proxy with AMQP & GZIP	54	59
Proxy with CoAP	101	101
Proxy with CoAP & GZIP	11	11

Table 6.11: NFFI LOS test - IP Packets sent and received by the client application.

Test	Packets sent	Packets received
Without Proxy	25	21
Proxy with HTTP	28	26
Proxy with HTTP & GZIP	24	24
Proxy with AMQP	189	201
Proxy with AMQP & GZIP	187	201
Proxy with CoAP	12	12
Proxy with CoAP & GZIP	12	12

Table 6.12: REST LOS test - IP Packets sent and received by the client application.

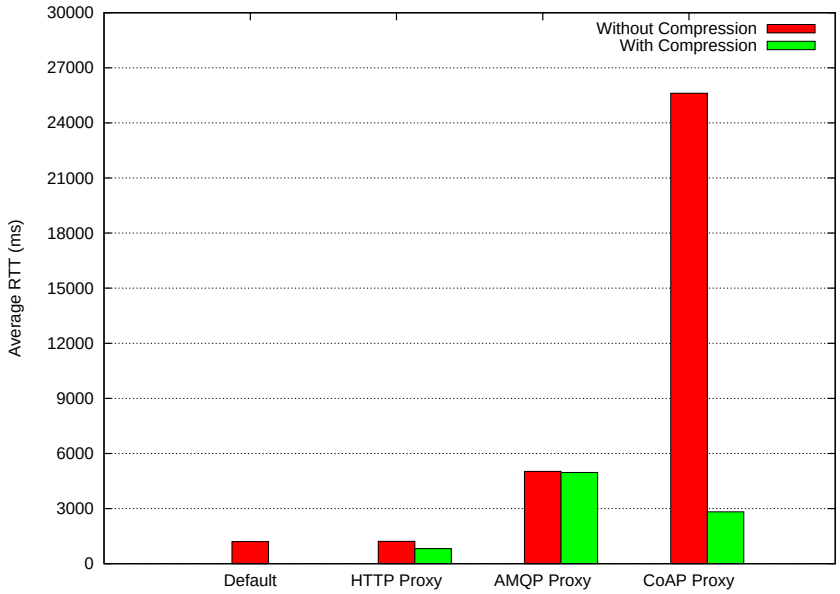
6.7.3 WiFi 1

With this type of network, we emulate communication over WiFi where the conditions are relatively good. The data rate is high, the delay is moderate, and the packet error rate is 1 %.

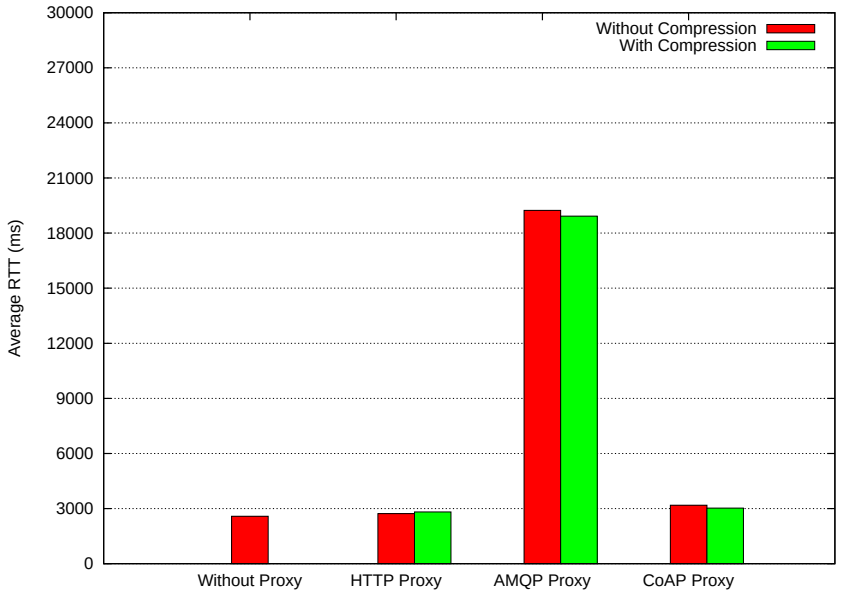
Results and Analysis

The results of the tests in this type of network are presented in fig. 6.11, table 6.13 and table 6.14. We see the following:

- Again we observe the same trends from previous tests. AMQP has the longest average RTT while CoAP struggle with large messages.
- For the NFFI test, HTTP proxy with compression yields the lowest average RTT.
- For the Car system tests, running without using proxies have the lowest average RTT.



(a) NFFI



(b) REST

Figure 6.11: WiFi 1 tests - Average RTT Time for the client application.

Test	Packets sent	Packets received
Without Proxy	50	45
Proxy with HTTP	45	45
Proxy with HTTP & GZIP	13	14
Proxy with AMQP	76	93
Proxy with AMQP & GZIP	60	60
Proxy with CoAP	104	104
Proxy with CoAP & GZIP	11	11

Table 6.13: NFFI WiFi 1 test - IP Packets sent and received by the client application.

Test	Packets sent	Packets received
Without Proxy	28	22
Proxy with HTTP	26	24
Proxy with HTTP & GZIP	30	27
Proxy with AMQP	192	211
Proxy with AMQP & GZIP	198	208
Proxy with CoAP	12	12
Proxy with CoAP & GZIP	12	12

Table 6.14: REST WiFi 1 test - IP Packets sent and received by the client application.

6.7.4 WiFi 2

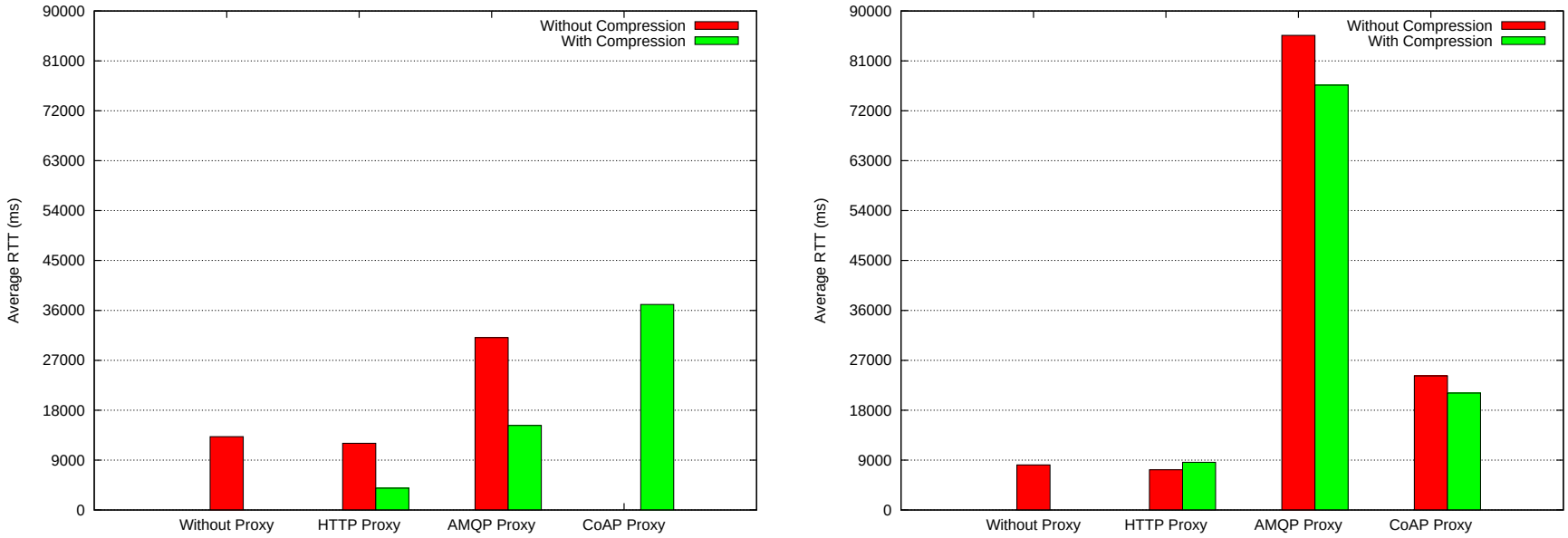
This type of network also emulates wireless communication, but instead in the “outer” areas of the wireless range. It has good data rate, moderate delay, and very high packet error rate (20 %).

Results and Analysis

Figure 6.12 shows the average response times of the WiFi 2 test cases. Table 6.15 and table 6.16 list the packets sent and received from the test applications in a sample test run. For the tests ran in an emulated WiFi 2 network, we see the following:

- A significantly longer average RTT for all test cases. The variance of the test results has increased compared to the other test networks. The high variance can be attributed to the high probability of packet errors, since some test runs may experience few errors, while other more.
- The HTTP proxy with compression had the overall best average RTT.

- In the NFFI test case with a CoAP proxy without compression, the proxy was not able to forward the request. The reason for this is that the CoAP request between the proxies timed out. The retransmission mechanism of the proxy was invoked, but the consecutive attempts were unsuccessfully as well. Furthermore, we observe that even with compression did the CoAP proxy have a longer average RTT than the other proxies protocols.
- We also see that for the NFFI test cases, compressing the messages yields a substantial performance increase with regards to the average RTT. This is probably due to since fewer IP packets need to be sent over the network, it is a less chance for packet errors.



(a) NFFI

(b) REST

Figure 6.12: WiFi 2 tests - Average RTT Time for the client application.

Test	Packets sent	Packets received
Without Proxy	51	54
Proxy with HTTP	45	52
Proxy with HTTP & GZIP	15	13
Proxy with AMQP	101	111
Proxy with AMQP & GZIP	76	71
Proxy with CoAP	0	0
Proxy with CoAP & GZIP	14	12

Table 6.15: NFFI WiFi 2 test - IP Packets sent and received by the client application.

Test	Packets sent	Packets received
Without Proxy	32	39
Proxy with HTTP	37	30
Proxy with HTTP & GZIP	31	28
Proxy with AMQP	332	317
Proxy with AMQP & GZIP	231	243
Proxy with CoAP	18	15
Proxy with CoAP & GZIP	24	17

Table 6.16: REST WiFi 2 test - IP Packets sent and received by the client application.

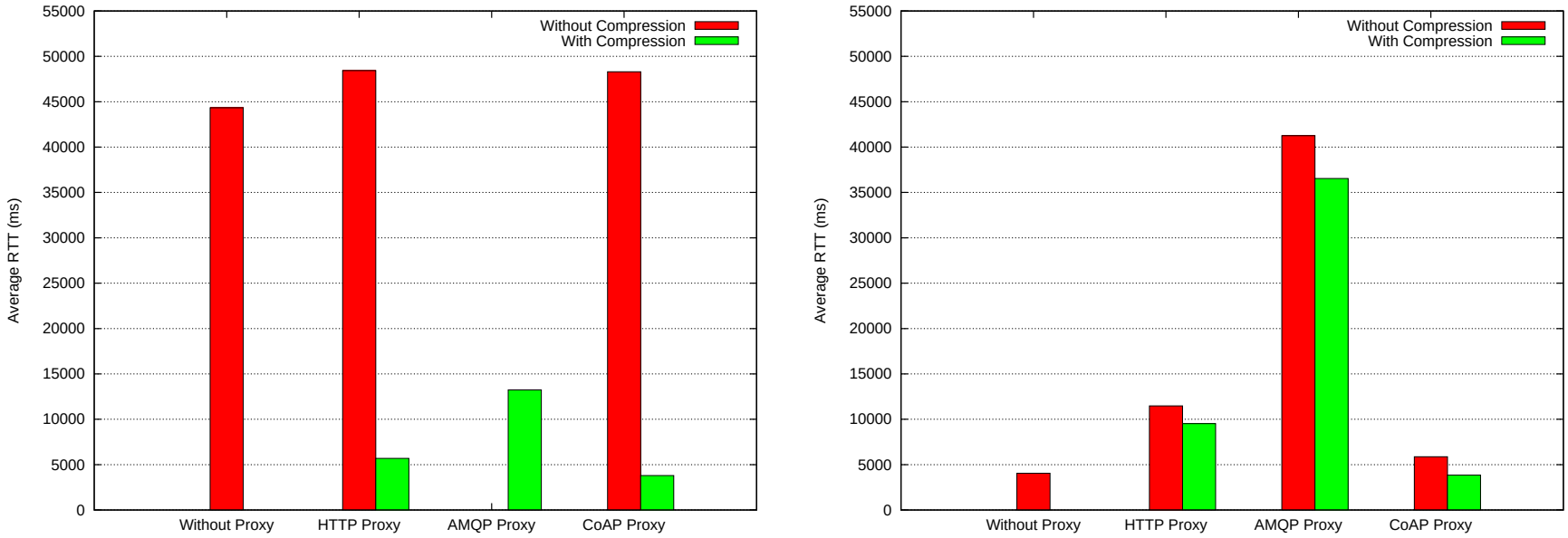
6.7.5 Combat Net Radio

CNR is characterized by its very low data rate, moderate timeout and packet error rate of 1 %.

Results and Analysis

In fig. 6.13 we show the average RTT of the tests for the emulated CNR network. Table 6.17 table 6.18 shows the IP packets sent/received in a sample run of the test cases. We observe the following:

- CoAP proxy with compression had the best average RTT and sent the fewest number of IP packets.
- The NFFI tests without compression have a very high average RTT.
- The AMQP test without compression was not able to complete before it timed out.
- If we compare the test cases without proxy and proxy with HTTP, we can see the overhead caused by using proxies. The increased HTTP message size caused by the proxy leads to a higher average RTT.



(a) NFFI

(b) REST

Figure 6.13: CNR tests - Average RTT Time for the client application.

Test	Packets sent	Packets received
Without Proxy	70	71
Proxy with HTTP	66	67
Proxy with HTTP & GZIP	14	13
Proxy with AMQP	0	0
Proxy with AMQP & GZIP	56	62
Proxy with CoAP	103	103
Proxy with CoAP & GZIP	11	11

Table 6.17: NFFI CNR test - IP Packets sent and received by the client application.

Test	Packets sent	Packets received
Without Proxy	25	21
Proxy with HTTP	28	27
Proxy with HTTP & GZIP	24	24
Proxy with AMQP	233	240
Proxy with AMQP & GZIP	220	225
Proxy with CoAP	14	13
Proxy with CoAP & GZIP	12	12

Table 6.18: REST CNR test - IP Packets sent and received by the client application.

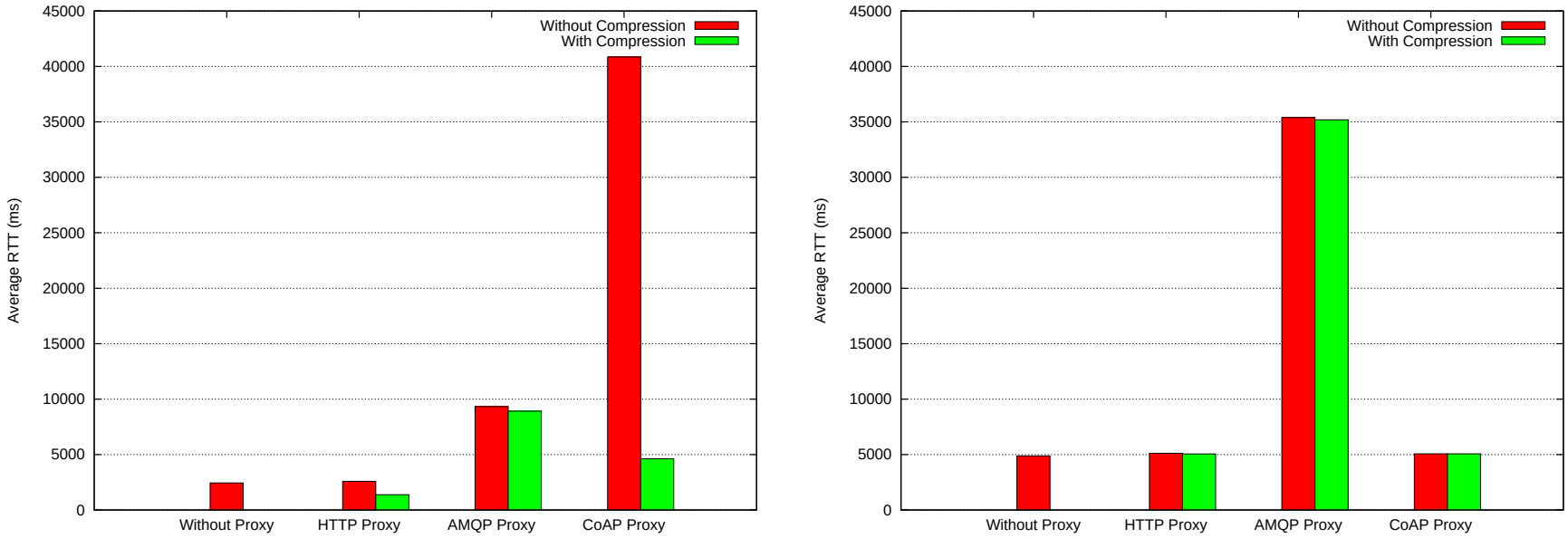
6.7.6 EDGE

EDGE is characterized by a low upload data rate and a moderately low download rate. We emulate EDGE with a moderate delay and zero packet loss.

Results and Analysis

Figure 6.14 shows the average response times of the EDGE test cases. Table 6.19 and table 6.20 list the packets sent and received from the test applications in a sample test run. We observe the following:

- HTTP proxy with compression has the overall lowest average RTT.
- Again we see that CoAP struggles with large messages.



(a) NFFI

(b) REST

Figure 6.14: EDGE tests - Average RTT Time for the client application.

Test	Packets sent	Packets received
Without Proxy	50	45
Proxy with HTTP	45	44
Proxy with HTTP & GZIP	14	13
Proxy with AMQP	78	95
Proxy with AMQP & GZIP	59	59
Proxy with CoAP	101	101
Proxy with CoAP & GZIP	11	11

Table 6.19: NFFI CNR test - IP Packets sent and received by the client application.

Test	Packets sent	Packets received
Without Proxy	27	23
Proxy with HTTP	28	27
Proxy with HTTP & GZIP	29	27
Proxy with AMQP	194	201
Proxy with AMQP & GZIP	201	212
Proxy with CoAP	12	12
Proxy with CoAP & GZIP	12	12

Table 6.20: REST EDGE test - IP Packets sent and received by the client application.

6.8 Experiments with Tactical Broadband

The majority of the testing was performed over software emulated networks. To validate these results, we performed experiments with military communication equipment. We used two WM600 radios developed by Kongsberg Defence & Aerospace (KDA), intended for users "on-the-move". WM600 can be used as IP radios through the Ethernet interface and support data rates up to 2500 kbit/s [55]. A picture of the radio can be seen in fig. 6.15.

We performed the testing in a communication laboratory located at FFI with the setup illustrated in fig. 6.16. It is a point-to-point setup with two radios and without any multi-hop functionality. The radios have the capacity to work as a multi-hop Mobile ad hoc network (MANET), but this was not tested in this thesis. The radios were attached to configurable attenuators, which could reduce the power of a signal by distorting its waveform. The purpose of the attenuators is to facilitate radio experiments with varying signal strength.

During our experiments, the attenuators were set to 30 DB. The measured data rate of the network was around 90 kbit/s and with a ping response time of 23 ms.

Figure 6.15: The KDA WM600 radio (from [55])

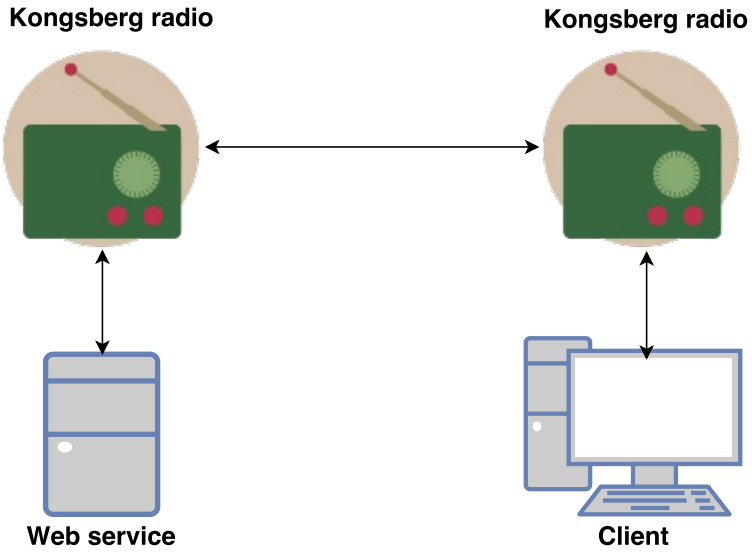
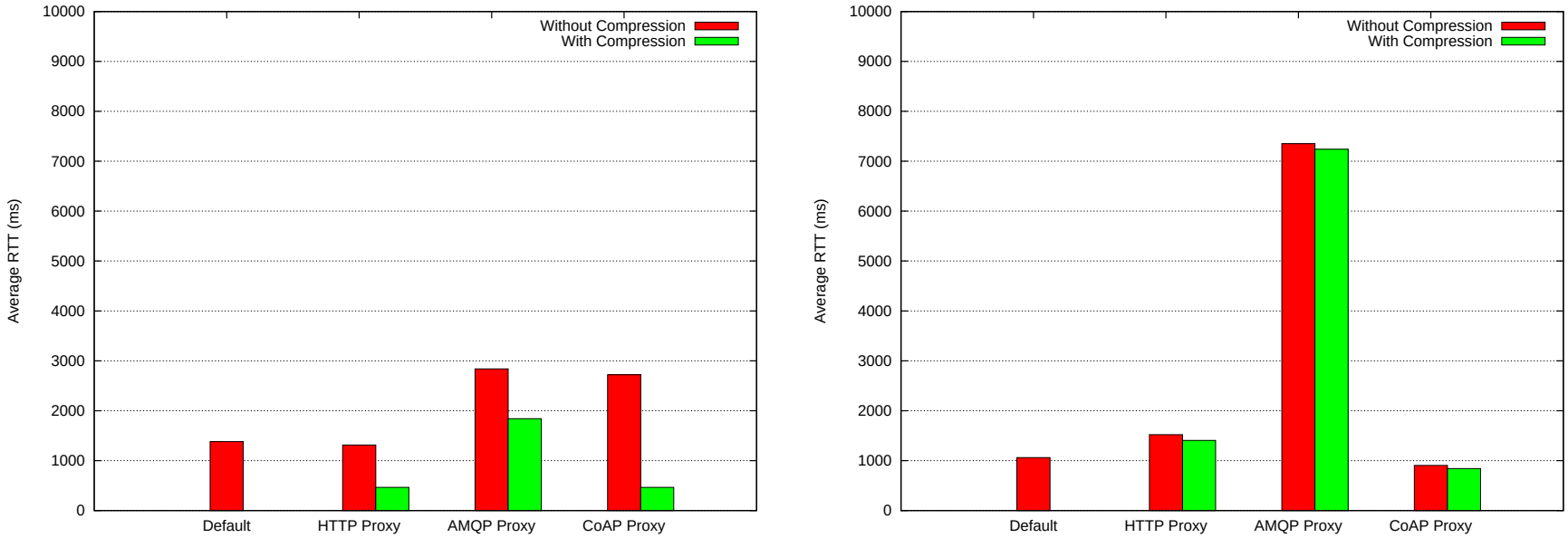


Figure 6.16: Tactical broadband testing environment

Results and analysis

Figure 6.17 shows the average RTT of the test cases performed over tactical broadband. We make the following observations:

- We see the same trends as in the software emulated networks.
- Compression yields a significantly lower RTT for the NFFI tests and a small decrease for the Car system tests.
- The CoAP proxy struggles with large messages but otherwise has the overall best RTT.



(a) NFFI

(b) REST

Figure 6.17: Tactical Broadband tests - Average RTT Time for the client application.

6.9 Discussion

In all emulated networks we see some consistent trends. Compressing the messages between the proxies generally lowered the RTT, especially for large messages of the NFFI test case. The exception is in some of the DIL networks with relatively high data rates, the time spent to compress and decompress a message was longer than the time saved by sending a compressed message.

In all test cases, the AMQP proxy had a significant overhead for each HTTP request forwarded by the proxy. We believe this is due to the laborious connection initialization of the AMQP protocol. This is especially noticeable to the many subsequent HTTP requests of the Car system test. We saw that for each HTTP request, a new AMQP connection was established, which generated a lot of network traffic. It is possible to avoid this by reusing connections over multiple requests, often referred to as *connection pooling*. However, the Camel AMQP component did not offer this functionality at the time of the implementation of this proxy. Regardless of this, compared against HTTP/TCP and CoAP, AMQP generates more network traffic.

Another consistent trend was that the CoAP proxy struggled with large messages in the NFFI test case. A Wireshark capture reveals that the CoAP proxy could utilize the Ethernet link in a better way. The maximum size of an IP packet sent over Ethernet is 1500 bytes [56], while the packet capture shows that CoAP splits larger messages into CoAP messages of only 512 bytes. Sending more than necessary packets over the network introduces some overhead:

- The minimum size of an IP packet header is 20 bytes. For each additional unnecessary packet sent, at least 20 more bytes are therefore sent over the network. Furthermore, since the receiver acknowledges each message, an *additional* packet is sent over the network.
- The more IP packets sent, the greater is the chance of packet loss. This especially applies to networks with high error rate.
- The messages have to be splitt at the sender and then reassembled at the receiver, consuming CPU power.

The maximum size of an IP packet is 65535 bytes [57] while the underlying transmission links usually have a much lower maximum size on its packets. Thus, if an IP packet larger than 1500 bytes is sent over Ethernet, it has to be *fragmented* into smaller fragments before they are sent. The maximum size of a packet that can be transmitted over a network without causing fragmentation is called the Path MTU. We generally want to avoid causing IP fragmentation due to the overhead associated with it [58].

To avoid IP fragmentation, and to support sending messages larger than 65 535 bytes, CoAP supports the block-wise feature splitting larger messages into smaller *blocks*. At the receiving end, these blocks are reassembled before they are delivered to the higher layers. The implementation we used for CoAP, Californium, supports the block-wise transfer feature. According to the specification of the feature, the byte size of each block must be of a power-of-two [24]. When we looked into the source code of Californium, we saw that the default size of a block is 512 bytes. This may be reasonable in some cases where the path MTU is not known or simply is low. However, in our case with a path MTU of 1500 bytes, the block size could have been set to 1024 to reduce the number of packets sent by the CoAP proxy.

Regardless of this, the CoAP proxy still performed equal to, or even better than, the HTTP proxy in some of the emulated networks. This can be due to CoAP's low overhead by having a small binary header and a simple messaging model.

6.10 Summary

In this chapter, we introduced six types of DIL networks and presented two test cases. We performed a function test of the proxy and saw that the premises and requirements were fulfilled. Then we showed how disconnects would cause clients not using a proxy to fail while the ones using a proxy were eventually successful. Finally, we evaluated the proxy solution with regards to the average RTT perceived by a client and network usage.

Overall, we saw that the HTTP proxy or not using a proxy yielded the lowest average RTT in the limited networks. However, not using a proxy is vulnerable to disconnects which the HTTP proxy handles better. Therefore, as the general recommendation, we recommend using a HTTP proxy in limited networks. In some special cases, however, the CoAP proxy may be a viable option. When the data rate of a network is low, and the message size is low, the CoAP proxy proved itself with a lower average RTT and less network usage than the HTTP proxy. Table 6.21 summarize our recommendations.

Network	**NFFI Web service recommendation**	**REST recommendation**
SATCOM	HTTP proxy with GZIP	HTTP proxy with GZIP
LOS	HTTP proxy with GZIP	HTTP proxy with GZIP
WiFi 1	HTTP proxy with GZIP	HTTP proxy with GZIP
WiFi 2	HTTP proxy with GZIP	HTTP proxy with GZIP
CNR	CoAP proxy with GZIP	CoAP proxy with GZIP
Edge	HTTP proxy with GZIP	HTTP proxy with GZIP

Table 6.21: Recommendations

Chapter 7

Conclusion and Future Work

In this chapter, we conclude the thesis and suggest potential future work.

7.1 Conclusion

The goal of this thesis was to investigate different ways to improve the performance of Web services in networks characterized by unreliable connects, high error rates, and low data rate. Web services enable interoperability between systems, but adapting Web services meant for civilian use into limited networks may not be feasible due to the network limitations. To adapt standard Web services into DIL networks without requiring incorporating proprietary solutions, based on previous research we introduced the usage of proxies. The proxy applies optimization techniques to facilitate the usage and to increase the performance of Web services in DIL. As a part of the thesis, we specified a set of requirements for the proxy and implemented it using the Apache Camel framework.Table 7.1 lists the premises and requirements.

In our evaluation, we tested whether our proxy solution fulfilled these premises and the more detailed requirements we specified in section 4.4. Through the function testing, we were able to prove that the proxy worked with a test set of Web service applications. The Web services successfully forwarded the requests through a deployed proxy pair, without requiring modifications except configuring the usage of proxies. This fulfilled premises one and three, as well as requirement 1 and 2.

Furthermore, we tested how the proxies facilitated the usage of Web services in DIL networks. We verified that the proxy was able to overcome the disconnect aspect of DIL by implementing a redelivery mechanism. This fulfilled requirement 5 and the disconnect aspect of premise 2. Requirement 4 regarding frequent network disruptions was not explicitly tested, but should be achieved by design since the proxy employs reliable protocols and an application layer redelivery

Premise 1	Support HTTP RESTful and W3C Web services.
Premise 2	Work in DIL networks.
Premise 3	Be interoperable with standards-based COTS solutions.
Premise 4	Work with security mechanisms.
Requirement 1	Receive and forward HTTP requests.
Requirement 2	Retain HTTP request and response headers.
Requirement 3	Support GZIP compression of payload.
Requirement 4	Handle frequent network disruptions.
Requirement 5	Handle disconnects over longer periods of time.
Requirement 6	Handle low data rates, high delays and high packet error rates.
Requirement 7	Allow for configuration of redelivery delay and maximal number of retransmissions.
Requirement 8	Support usage of different transport protocols between the proxies.
Requirement 9	Easy configuration of which protocol to use.
Requirement 10	Be easily extendable to include other protocols and other optimization techniques.

Table 7.1: Premises and requirements

mechanism.

Requirement 3 was fulfilled by implementing GZIP compression on messages between proxies. Optimization was identified to yield a significant performance increase with regards the RTT time perceived by Web service clients.

We also validated that the proxy could overcome the limited aspect of DIL as the test cases were successful in all emulated DIL networks. This fulfilled premises 2 and requirement 6. Furthermore, we supported a set of transport protocols as the means of transporting data between proxies and with that fulfilled requirement 8. We saw how different transport protocol affected the performance of Web services. In most of the various types of DIL environments, using HTTP/TCP as the inter-proxy protocol was identified as the best transport. However, we saw that in cases where the message payload was low and in networks with low data rates, using a CoAP proxy was the best option. We also discovered how the Californium implementation of CoAP with default configuration caused a sub-optimal utilization of an Ethernet link. Tuning the block-size configuration could improve the CoAP's proxy performance also for larger payloads.

Next, we were able to show that the proxy works with security mechanisms by verifying that the digital signature of the payload is intact, by diffing the payload when not using proxies with the payload when using proxies.

Finally, the proxy implements a configuration setup that allows the user to specify different properties of the proxy. A user of the proxy can easily configure properties of the redelivery mechanism and change the transport protocol used between proxies. The proxy has been designed to be easily extendable to include other protocols and optimization techniques. The Apache Camel framework used in the implementation facilitates this by supporting a component based transport mechanism, as well as easily allowing customization of the payload. This satisfies requirement 7, 9 and 10.

All in all, the goal of the thesis was reached. We developed a prototype proxy and gave a recommendation regarding optimization techniques for Web services in DIL environments. Table 7.2 summarize our recommendations.

DIL Network	**Optimization recommendation**
SATCOM	HTTP proxy with GZIP
LOS	HTTP proxy with GZIP
WiFi 1	HTTP proxy with GZIP
WiFi 2	HTTP proxy with GZIP
CNR	CoAP proxy with GZIP
Edge	HTTP proxy with GZIP

Table 7.2: Optimization recommendations for DIL networks

Further possible investigations in this area and improvements to the proxy are discussed in the next section.

7.2 Future Work

The proxy solution developed as a part of this thesis primarily focus on alternative transport protocols and compression as the means of optimization. Further investigations should consider other optimization techniques as well, such as caching and content filtering. Moreover, due to the time available, we were not able to test the SCTP protocol for use in the proxies. We identified SCTP as a potential protocol for DIL networks, so further optimization investigations should consider this protocol as well.

Security is of vital importance in military communication. Another area that could be investigated further is to perform tests with IPSEC enabled. IPSEC is a protocol suite designed to provide interoperable cryptographic security on the IP layer [59]. Performing tests with IPsec enabled would give knowledge of how this security mechanism affects the performance of Web services.

7.2.1 Improving the proxy

In our evaluation, we saw that which transport protocol had the best performance depended on the message size and the current network conditions. Future optimization of the proxy solution may include a runtime selection of the transport protocol instead of selection at start-up. Furthermore, we saw that the CoAP implementation we used, sent large messages split over small Ethernet frames. Setting a higher CoAP block size may yield a potential performance increase when sending large messages with the CoAP protocol.

To reduce the overhead of the proxy, further optimization may improve the proxy message format implementation. Instead of using a textual message format, a compact binary header could be used.

Known Bugs

During our Wireshark analysis of the test results, we discovered that when using the proxy, some of the HTTP request headers was propagated into the HTTP response headers. This happened for all inter-proxy protocols and caused by the way Camel exchanges are implemented. This could be fixed by explicitly removing the headers when routing the message back to the requester. Although this implies that the proxies are not completely transparent, we do not believe this had any significant impact on the testing. The payload of messages is still exactly the same, and all headers are present. However, for each unnecessary header sent, additional bytes has to be transferred over the network. Although the number of additional bytes is relatively small, fixing this bug would reduce the overhead of using the proxy.

Bibliography

[1] P. Bartolomasi et al. *NATO network enabled capability feasibility study*. 2005.

[2] NATO. *NATO - Member Countries*. http://www.nato.int/cps/en/natohq/nato_countries.htm. Accessed: 2015-05-04.

[3] OASIS et al. *Reference Model for Service Oriented Architecture 1.0 OASIS standard*. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf. Accessed: 06. 10. 2015. Oct. 2006.

[4] NATO C3 Board. *Core Enterprise Services Standards Recommendations - The SOA Baseline Profile*. 1.7. 2011.

[5] Frank T. Johnsen. "Pervasive Web Services Discovery and Invocation in Military Networks". In: *FFI-rapport 2011/00257* (2011).

[6] F. Annunziata et al. *IST-090 SOA challenges for disadvantaged grids*. https://www.cso.nato.int/pubs/rdp.asp?RDP=STO-TR-IST-090. Apr. 2014.

[7] A. Gibb et al. "Information Management over Disadvantaged Grids". In: *Task Group IST-030/ RTG-012, RTO-TR-IST-030* (2007). Final report of the RTO Information Systems Technology Panel.

[8] S Rajasekar, P Philominathan, and V Chinnathambi. "Research methodology". In: *arXiv preprint physics/0601009* (2006).

[9] Peter J. Denning et al. "Computing as a dicipline". In: *Communications of the ACM* (1989).

[10] R. Braden. *RFC 1122 – Requirements for Internet Hosts – Communication Layers*. https://tools.ietf.org/html/rfc1122. Accessed: 06. 01. 2016. Oct. 1989.

[11] Hugo Haas and Allen Brown. *Web Services Glossary*. http://www.w3.org/TR/ws-gloss/\#webservice. Accessed: 2015-05-06.

[12] W3C. *Extensible markup language (XML) 1.0*. Nov. 2008. URL: https://www.w3.org/TR/REC-xml/ (visited on 02/25/2016).

[13] Erik Christensen et al. *W3C - Web service definition language (WSDL)*. Mar. 2001. URL: https://www.w3.org/TR/wsdl (visited on 02/27/2016).

[14] Martin Gudgin et al. *W3C - SOAP version 1.2 part 1: Messaging framework (Second edition)*. Apr. 2007. URL: https://www.w3.org/TR/soap12-part1/ (visited on 02/27/2016).

[15] Roy T. Fielding and Richard N. Taylor. "Principled Design of the Modern Web Architecture". In: *Proceedings of the 22Nd International Conference on Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 407–416. ISBN: 1-58113-206-9. DOI: 10.1145/337180.337228. URL: http://doi.acm.org/10.1145/337180.337228.

[16] Frank. T Johnsen, Trude Bloebaum, and Kristoffer R. Karud. "Recommendations for increased efficiency of Web services in the tactical domain". In: International Conference on Military Communications and Information Systems (ICMCIS). Krakow, Poland, May 2015.

[17] R. Fielding et al. *RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1*. https://tools.ietf.org/html/rfc2616. Accessed: 10. 02. 2016. June 1999.

[18] Information Sciences Institute - University of Southern California. *RFC 793 – Transmission Control Protocol*. https://tools.ietf.org/html/rfc793. Accessed: 10. 02. 2016. Sept. 1981.

[19] David J. Wetherall Andrew S. Tanenbaum. *Computer Networks*. Fifth Edition. Pearson New International Edition.

[20] Hussein Al-Bahadili. *Simulation in computer network design and modeling: Use and analysis*. IGI Global, Feb. 2012.

[21] J Postel. *RFC 768 - User Datagram protocol*. Aug. 1980. URL: https://tools.ietf.org/html/rfc768 (visited on 02/28/2016).

[22] S. Floyd and K. Fall. "Promoting the use of end-to-end congestion control in the Internet". In: *IEEE/ACM Transactions on Networking* 7.4 (Aug. 1999), pp. 458–472. ISSN: 1063-6692. DOI: 10.1109/90.793002.

[23] Z. Shelby et al. *RFC 7252 – The Constrained Application Protocol (CoAP)*. https://tools.ietf.org/html/rfc7252. Accessed: 10. 02. 2016. June 2014.

[24] C. Bormann. *Block-wise transfers in CoAP – draft-ietf-core-block-19*. https://tools.ietf.org/html/draft-ietf-core-block-19. Accessed: 23. 04. 2016. Mar. 2016.

[25] OASIS. *Advanced message queuing protocol (AMQP) version 1.0*. Oct. 2012. URL: http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html%5C#toc (visited on 02/28/2016).

[26] OASIS, Andrew Banks, and Rahul Gupta. *MQTT Version 3.1.1 Specification*. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. Accessed: 06. 01. 2016. Oct. 2014.

[27] R Stewart. *RFC 4960 - Stream control transmission protocol*. Sept. 2007. URL: https://tools.ietf.org/html/rfc4960 (visited on 02/29/2016).

[28] Frank T. Johnsen et al. "IST-118 - SOA recommendations for Disadvantaged Grids in the Tactical Domain". In: *18th ICCRTS* (2013).

[29] Frank T. Johnsen and Trude Bloebaum. "Using NFFI Web Services on the tactical level: An evaluation of compression techniques". In: 13th International Command and Control Research and Technology Symposium (ICCRTS). Seatle, WA, USA, 2008.

[30] Frank T. Johnsen et al. "Evaluation of Transport Protocols for Web Services". In: *MCC 2013* (2013).

[31] D. Thangavel et al. "Performance evaluation of MQTT and CoAP via a common middleware". In: *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*. Apr. 2014, pp. 1–6. DOI: 10.1109/ISSNIP.2014.6827678.

[32] N. De Caro et al. "Comparison of two lightweight protocols for smartphone-based sensing". In: *2013 IEEE 20th Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*. Nov. 2013, pp. 1–6. DOI: 10.1109/SCVT.2013.6735994.

[33] W. Colitti et al. "Evaluation of constrained application protocol for wireless sensor networks". In: *Local Metropolitan Area Networks (LANMAN), 2011 18th IEEE Workshop on*. Oct. 2011, pp. 1–6. DOI: 10.1109/LANMAN.2011.6076934.

[34] Ketil Lund et al. "Information exchange in heterogeneous military networks". In: *FFI-rapport 2009/02289* (2009).

[35] Ketil Lund et al. "Robust Web Services in Heterogeneous Military Networks". In: *IEEE Communications Magazine, Special Issue on Military Communications* (Oct. 2010).

[36] N. Suri et al. "Agile Computing Middleware Support for Service-Oriented Computing over Tactical Networks". In: *Vehicular Technology Conference (VTC Spring), 2015 IEEE 81st*. May 2015, pp. 1–5. DOI: 10.1109/VTCSpring.2015.7145672.

[37] Anne Diefenbach et al. "TACTTICS TSI Architecture – A European Reference Architecture for Tactical SOA". In: International Conference on Military Communications and Information Systems (ICMCIS). To appear. May 2016.

[38] *nginx home page*. URL: http://nginx.org/.

[39] *squid home page*. URL: http://www.squid-cache.org/.

[40] *Apache Camel home page*. URL: http://camel.apache.org/.

[41] *jetty home page*. URL: http://www.eclipse.org/jetty/.

[42] *Californium home page*. URL: http://www.eclipse.org/californium/.

[43] T. Bray. *RFC 7159 – The JavaScript Object Notation (JSON) Data Interchange Format*. https://tools.ietf.org/html/rfc7159. Accessed: 17. 04. 2016. Mar. 2014.

[44] *Github – typesafehub/config*. URL: https://github.com/typesafehub/config.

[45] Oracle. *Java Networking and Proxies*. https://docs.oracle.com/javase/8/docs/technotes/guides/net/proxies.html.

[46] *Apache camel - Writing Components*. URL: http://camel.apache.org/writing-components.html.

[47] *Github – camel-coap*. URL: https://github.com/Quist/camel-coap.

[48] Esko Dijk et al. *Guidelines for HTTP-CoAP mapping Implementations*. URL: https://tools.ietf.org/html/draft-ietf-core-http-mapping-09 (visited on 05/01/2016).

[49] Michael A. Krog et al. "PISA: Platform Independent Sensor Application". In: 20th International Command and Control Research and Technology Symposium (ICCRTS). 2015.

[50] Fabio Ludovici and Hagen Paul Pfeifer. *Tc-netem(8) - Linux manual page*. Nov. 2011. URL: http://man7.org/linux/man-pages/man8/tc-netem.8.html (visited on 03/29/2016).

[51] *iPerf 3 home page*. URL: https://iperf.fr/.

[52] T.H Bloebaum, Frank T. Johnsen, and Gunnar Salberg. "Monitoring in Disadvantaged Grids". In: *18th ICCRTS* (2013).

[53] *Wireshark home page*. URL: https://iperf.fr/.

[54] *ActiveMQ home page*. URL: http://activemq.apache.org/.

[55] Kongsberg Defence & Aerospace. *TacLAN UHF radio*. http://www.kongsberg.com/en/kds/products/defencecommunications/taclan/. Accessed: 18. 04. 2016.

[56] Charles Hornig. *RFC 894 – A Standard for the Transmission of IP Datagrams over Ethernet Networks*. https://tools.ietf.org/html/rfc894. Accessed: 23. 04. 2016. Sept. 1984.

[57] Information Sciences Institute - University of Southern California. *RFC 793 – Internet Protocol – DARPA Internet Program – Protocol Specification*. https://tools.ietf.org/html/rfc791. Accessed: 23. 04. 2016. Sept. 1981.

[58] Delian Genkov and R Ilarionov. “Avoiding IP Fragmentation at the Transport Layer of the OSI Reference Model”. In: *Proceedings of the international conference on computer systems and technologies–CompSysTech, University of Veliko Tarnovo, Bulgaria* (2006).

[59] S. Kent and R. Atkinson. *RFC 2401 – Security Architecture for the Internet Protocol*. Nov. 1998. URL: `https://www.ietf.org/rfc/rfc2401.txt` (visited on 05/02/2016).

[60] *The Apache Commons Mathematics Library home page*. URL: `https://commons.apache.org/proper/commons-math/`.

Acronyms

ACM Agile Computing Middleware. 40

AFRO Adaption Framework foR Web Services prOvision. 41, 42

AMQP Advanced Message Queuing Protocol. 31, 48, 99

API Application Program Interface. 70

CNR Combat Net Radio. 66, 67, 92, 101, 105, 122

CoAP Constrained Application Protocol. 30, 38, 48, 63, 64, 99, 100, 104

COTS Commercial off-the-shelf. 14

DIL Disconnected, Intermittent and Limited. 13, 14, 16, 35, 46

DSProxy Delay and disruption tolerant SOAP Proxy. 39, 40

EDA European Defence Agency. 42

EDGE Enhanced Data rates for GSM Evolution. 65–67, 94, 123

EFX Efficient XML. 37

FFI Norwegian Defence Research Establishment. 5, 39, 96

FTP File Transfer Protocol. 21

HTTP Hypertext Transfer Protocol. 16, 20, 25, 48, 55

IETF Internet Engineering Task Force. 25

IoT Internet of Things. 9, 14, 29, 32, 56

IP Internet Protocol. 20, 29

JSON JavaScript Object Notation. 56, 71

JVM Java Virtual Machine. 63

KDA Kongsberg Defence & Aerospace. 96

LOS Line of Sight. 66, 67, 84, 101, 105, 121

LTE Long-Term Evolution. 66

MANET Mobile ad hoc network. 96

Mockets Mobile Sockets. 41

MTU Maximum Transfer Unit. 27, 100

NATO North Atlantic Treaty Organization. 9, 10, 45

NEC Network Enabled Capability. 9

NetEm Network Emulator. 68

NFFI NATO Friendly Force Information. 70

NIO Java new/non-blocking I/O. 43

NTNU Norwegian University of Science and Technology. 42

OASIS Organization for the Advancement of Structured Information Standards. 10, 31

PER Packet Error Rate. 13, 22

QoS Quality of Service. 32, 38, 41

REST Representational State Transfer. 12, 22, 24, 42

RTT Round-Trip Time. 69, 75

SATCOM Satellite Communication. 66, 67, 82, 101, 105

SCTP Stream Control Transmission Protocol. 32, 37, 49

SOA Service Oriented Architecture. 9–12, 22, 42

STD Standard Deviation. 125, 126

TBF Token Bucket Filter. 68

TCP Transmission Control Protocol. 20, 21, 26, 28, 29, 31, 32, 41

TSI Tactical Service Infrastructure. 42

UDP User Datagram Protocol. 29, 30, 32, 41, 48

URI Uniform Resource Identifier. 25, 30, 60

W3C World Wide Web Consortium. 11, 22, 25

WSDL Web Services Description Language. 23

XML Extensible Markup Language. 23, 37

Appendices

Appendix A

Configuration

A.1 Proxy Configuration

The configuration fields of the proxy is listed in table A.1. In addition, some protocols require additional configurations. The AMQP protocol uses a broker, which must be configured. See table x.

Field	Purpose	Required	Type
useCompression	Turn GZIP compression on/off	Yes	boolean
protocol	Inter-proxy communication protocol	Yes	String
hostname	Hostname to listen on	Yes	String
port	Which port the proxy should listen for messages	Yes	Integer
targetProxyHostname	The hostname and the port of the opposite proxy	Yes	String
timeout	The timeout value of a request between proxies	No	Integer
useExponentialBackoff	Turn on/off exponential backoff of retransmission	No	Integer
redeliveryDelay	Number of milliseconds to wait before trying redelivery	No	Integer
maximumRetransmissions	Maximum number of attempted retransmissions. -1 indicates infinity	No	Integer

Table A.1: Configuration fields of the Proxy

Appendix B

Network emulating

This appendix lists the different scripts that was used to emulate the different types of networks.

Listing B.1: "Emulating SATCOM"

```
1 #!/bin/bash
2
3 sudo tc qdisc add dev eth0 handle 1: root tbf rate  250kbit
    burst 15000 limit 15000
4 sudo tc qdisc add dev eth0 parent 1:1 handle 10: netem
    delay 550ms
5 sudo tc qdisc add dev eth1 handle 1: root tbf rate  250kbit
    burst 15000 limit 15000
6 sudo tc qdisc add dev eth1 parent 1:1 handle 10: netem
    delay 550ms
```

B.1 LOS

Listing B.2: "Emulating LOS"

```
1 #!/bin/bash
2
3 sudo tc qdisc add dev eth0 handle 1: root tbf rate  2mbit
    burst 15000 limit 15000
4 sudo tc qdisc add dev eth0 parent 1:1 handle 10: netem
    delay 5ms
5 sudo tc qdisc add dev eth1 handle 1: root tbf rate  2mbit
    burst 15000 limit 15000
6 sudo tc qdisc add dev eth1 parent 1:1 handle 10: netem
    delay 5ms
```

B.2 WiFi 1

Listing B.3: "Emulating WiFi 1"

```
#!/bin/bash

sudo tc qdisc add dev eth0 handle 1: root tbf rate  2mbit
    burst 15000 limit 15000
sudo tc qdisc add dev eth0 parent 1:1 handle 10: netem
    delay 100ms corrupt 1%
sudo tc qdisc add dev eth1 handle 1: root tbf rate  2mbit
    burst 15000 limit 15000
sudo tc qdisc add dev eth1 parent 1:1 handle 10: netem
    delay 100ms corrupt 1%
```

B.3 WiFi 2

Listing B.4: "Emulating WiFi 2"

```
#!/bin/bash

sudo tc qdisc add dev eth0 handle 1: root tbf rate  2mbit
    burst 15000 limit 15000
sudo tc qdisc add dev eth0 parent 1:1 handle 10: netem
    delay 100ms corrupt 20%
sudo tc qdisc add dev eth1 handle 1: root tbf rate  2mbit
    burst 15000 limit 15000
sudo tc qdisc add dev eth1 parent 1:1 handle 10: netem
    delay 100ms corrupt 20%
```

B.4 CNR

Listing B.5: "Emulating CNR"

```
#!/bin/bash

sudo tc qdisc add dev eth0 handle 1: root tbf rate  9.6kbit
    burst 15000 limit 15000
sudo tc qdisc add dev eth0 parent 1:1 handle 10: netem
    delay 100ms corrupt 1%
sudo tc qdisc add dev eth1 handle 1: root tbf rate  9.6kbit
    burst 15000 limit 15000
sudo tc qdisc add dev eth1 parent 1:1 handle 10: netem
    delay 100ms corrupt 1%
```

B.5 EDGE

Listing B.6: "Emulating EDGE"

```
#!/bin/bash

sudo tc qdisc add dev eth0 handle 1: root tbf rate  200kbit
    burst 15000 limit 15000
sudo tc qdisc add dev eth0 parent 1:1 handle 10: netem
    delay 200ms
sudo tc qdisc add dev eth1 handle 1: root tbf rate  50kbit
    burst 15000 limit 15000
sudo tc qdisc add dev eth1 parent 1:1 handle 10: netem
    delay 200ms
```

Appendix C

Results

In this appendix the data material from the evaluations is presented. Each test case was run a number of times, ranging from 10 to 100 runs. Then the mean, Standard Deviation (STD) and variance was calculated by using the Apache Commons Mathematics Library[60]. An example of how this was done running NFFI Web service tests can be seen in listing C.1.

Listing C.1: "Calculating statistic values"

```
DescriptiveStatistics stats = new DescriptiveStatistics();

for (int i=0; i<antall; ++i) {
    long ts1 = System.currentTimeMillis();
    NFFIDataResponse response = pullDataOperation(null);
    long ts2 = System.currentTimeMillis();
    stats.addValue(ts2-ts1);
}

System.out.println("Mean: " + stats.getMean());
System.out.println("Standard Deviation: " +
   stats.getStandardDeviation());
System.out.println("Variance: " + stats.getVariance());
System.out.println("Min: " + stats.getMin());
System.out.println("Max: " + stats.getMax());
System.out.println("Median: " + stats.getPercentile(50));
```

We also performed an analysis of the network utilization using Wireshark. This was done by starting a packet capture, running one test run and inspecting the packet capture. The calculation of bytes sent and received was done by:

1. Starting Wireshark on the same machine as the client.
2. Filtering traffic to only show traffic between the IP addresses of the client and Web service.

3. Using the TCP/UDP conversation view of Wireshark.

C.1 Function Tests

- Ping measured to ~1 ms.
- Iperf3 measured data rate: 7.76 Mbits/sec.

C.1.1 NFFI Web Service

Test	Mean	STD	Variance	Test runs
Without proxy	122 ms	29	869	300
Proxy with HTTP	163 ms	25	601	300
Proxy with HTTP & GZIP	99 ms	19	346	300
Proxy with AMQP	529 ms	60	3690	300
Proxy with AMQP & GZIP	490 ms	62	3847	300
Proxy with CoAP	285 ms	33	1122	300
Proxy with CoAP & GZIP	122 ms	33	1091	300

Table C.1: Mean response times of NFFI Web Service - Function Test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	51	4609	46	51706
Proxy with HTTP	45	5392	44	55489
Proxy with HTTP & GZIP	13	2781	13	2585
Proxy with AMQP	73	10284	94	64472
Proxy with AMQP & GZIP	57	8731	62	15244
Proxy with CoAP	101	8120	101	57137
Proxy with CoAP & GZIP	11	1680	11	5502

Table C.2: Wireshark analysis of NFFI Web Service - Function Test

C.1.2 RESTful Car System

Test	Mean	STD	Variance	Test runs
Without proxy	30 ms	12	147	100
Proxy with HTTP	160 ms	97	9486	100
Proxy with HTTP & GZIP	159 ms	76	5822	100
Proxy with AMQP	1919 ms	128	16388	100
Proxy with AMQP & GZIP	1880 ms	109	11919	100
Proxy with CoAP	124 ms	64	4079	100
Proxy with CoAP & GZIP	128 ms	64	4109	100

Table C.3: Mean response times of RESTful Car System - Function Test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	25	4738	21	5638
Proxy with HTTP	28	9677	26	15147
Proxy with HTTP & GZIP	28	8735	28	12993
Proxy with AMQP	180	30366	203	47484
Proxy with AMQP & GZIP	190	30224	207	42314
Proxy with CoAP	12	4757	12	8369
Proxy with CoAP & GZIP	12	3943	12	6053

Table C.4: Wireshark analysis of RESTful Car System - Function Test

C.2 Satellite Tests

- Ping measured to ~1100 ms.
- Iperf3 measured data rate: 402/291 Kbits/sec.

C.2.1 NFFI Web Service

Test	Mean	STD	Variance	Test runs
Without proxy	4978 ms	378	142762	10
Proxy with HTTP	4511 ms	71	5009	10
Proxy with HTTP & GZIP	3530 ms	50	2472	10
Proxy with AMQP	25709 ms	793	628112	10
Proxy with AMQP & GZIP	25780 ms	1159	1343947	10
Proxy with CoAP	111636 ms	59	3437	10
Proxy with CoAP & GZIP	12347 ms	41	1652	10

Table C.5: Mean response times of NFFI Web Service - Satellite test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	54	4811	47	51623
Proxy with HTTP	47	5532	45	55563
Proxy with HTTP & GZIP	16	2987	14	7177
Proxy with AMQP	88	11342	102	65040
Proxy with AMQP & GZIP	71	9731	68	15679
Proxy with CoAP	101	7810	101	56827
Proxy with CoAP & GZIP	11	1668	11	5486

Table C.6: Wireshark analysis of NFFI Web Service - Satellite test

C.2.2 RESTful Car System

Test	Mean	STD	Variance	Test runs
Without proxy	13386 ms	401	160523	10
Proxy with HTTP	13643 ms	427	182464	10
Proxy with HTTP & GZIP	13825 ms	897	804893	10
Proxy with AMQP	102748 ms	3065	9396423	10
Proxy with AMQP & GZIP	94163 ms	568	322659	10
Proxy with CoAP	13545 ms	217	47260	10
Proxy with CoAP & GZIP	13562 ms	223	49522	10

Table C.7: Mean response times of RESTful Car System - Satellite test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	27	4878	22	5712
Proxy with HTTP	26	9538	25	15075
Proxy with HTTP & GZIP	30	8873	28	13010
Proxy with AMQP	244	34841	238	49914
Proxy with AMQP & GZIP	240	33739	240	44625
Proxy with CoAP	12	4751	12	8380
Proxy with CoAP & GZIP	12	3940	12	6063

Table C.8: Wireshark analysis of RESTful Car System - Satellite test

C.3 Line-of-Sight Tests

- Ping measured to ~11 ms.
- Iperf3 measured data rate: 2.34/2.15 Mbits/sec.

C.3.1 NFFI Web Service

Test	Mean	STD	Variance	Test runs
Without proxy	242 ms	26	663	100
Proxy with HTTP	299 ms	40	1577	100
Proxy with HTTP & GZIP	162 ms	34	1177	100
Proxy with AMQP	821 ms	60	3588	100
Proxy with AMQP & GZIP	693 ms	75	5632	100
Proxy with CoAP	1359 ms	45	1988	100
Proxy with CoAP & GZIP	262 ms	36	1314	100

Table C.9: Mean response times of NFFI Web Service - LOS test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	46	4267	43	51343
Proxy with HTTP	43	5260	44	55489
Proxy with HTTP & GZIP	14	2847	13	7103
Proxy with AMQP	68	9950	91	64274
Proxy with AMQP & GZIP	54	8529	59	15044
Proxy with CoAP	101	7565	101	56582
Proxy with CoAP & GZIP	11	1647	11	5466

Table C.10: Wireshark analysis of NFFI Web Service - LOS test

C.3.2 RESTful Car System

Test	Mean	STD	Variance	Test runs
Without proxy	156 ms	15	214	100
Proxy with HTTP	288 ms	77	6000	100
Proxy with HTTP & GZIP	292 ms	86	7382	100
Proxy with AMQP	2567 ms	102	10333	100
Proxy with AMQP & GZIP	2579 ms	129	16595	100
Proxy with CoAP	256 ms	69	4775	100
Proxy with CoAP & GZIP	263 ms	69	4693	100

Table C.11: Mean response times of RESTful Car System - LOS test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	25	4738	21	5638
Proxy with HTTP	28	9704	26	15201
Proxy with HTTP & GZIP	24	8486	24	8486
Proxy with AMQP	189	30968	201	47352
Proxy with AMQP & GZIP	187	29979	201	41927
Proxy with CoAP	12	4756	12	8397
Proxy with CoAP & GZIP	12	3934	12	6059

Table C.12: Wireshark analysis of RESTful Car System - LOS test

C.4 WiFi 1 tests

- Ping measured to ~200 ms.
- Iperf3 measured data rate: 1.72/1.67 Mbits/sec.

C.4.1 NFFI Web Service

Test	Mean	STD	Variance	Test runs
Without proxy	1202 ms	162	26326	100
Proxy with HTTP	1213 ms	354	125628	100
Proxy with HTTP & GZIP	820 ms	154	23586	100
Proxy with AMQP	5026 ms	460	211385	100
Proxy with AMQP & GZIP	4964 ms	637	405390	100
Proxy with CoAP	25615 ms	3185	10142866	10
Proxy with CoAP & GZIP	2823 ms	1425	2031770	100

Table C.13: Mean response times of NFFI Web Service - WiFi 1 test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	50	4531	45	51475
Proxy with HTTP	45	5560	45	57003
Proxy with HTTP & GZIP	13	2793	14	8297
Proxy with AMQP	76	10494	93	64406
Proxy with AMQP & GZIP	60	8941	60	15126
Proxy with CoAP	104	9214	104	58817
Proxy with CoAP & GZIP	11	1682	11	5491

Table C.14: Wireshark analysis of NFFI Web Service - WiFi 1 test

C.4.2 RESTful Car System

Test	Mean	STD	Variance	Test runs
Without proxy	2581 ms	265	70406	100
Proxy with HTTP	2728 ms	270	73000	100
Proxy with HTTP & GZIP	2818 ms	369	136307	100
Proxy with AMQP	19236 ms	490	240174	10
Proxy with AMQP & GZIP	18925 ms	722	521008	10
Proxy with CoAP	3184 ms	1565	2447810	100
Proxy with CoAP & GZIP	3024 ms	946	894686	100

Table C.15: Mean response times of RESTful Car System - WiFi 1 test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	28	5146	22	6060
Proxy with HTTP	26	9564	24	15061
Proxy with HTTP & GZIP	30	9476	27	12925
Proxy with AMQP	192	31450	211	49663
Proxy with AMQP & GZIP	198	30730	208	42380
Proxy with CoAP	12	4754	12	8366
Proxy with CoAP & GZIP	12	3945	12	6035

Table C.16: Wireshark analysis of RESTful Car System - WiFi 1 test

C.5 WiFI 2 Tests

- Ping measured to ~200 ms.
- Iperf3 measured data rate: 125/99.6 Kbits/sec.

C.5.1 NFFI Web Service

Test	Mean	STD	Variance	Test runs
Without proxy	13235 ms	9070	82266227	10
Proxy with HTTP	12042 ms	6908	47717943	10
Proxy with HTTP & GZIP	3938 ms	4793	22970668	20
Proxy with AMQP	31096 ms	20578	423443967	10
Proxy with AMQP & GZIP	15243 ms	9267	85874508	10
Proxy with CoAP	0 ms	-	-	1
Proxy with CoAP & GZIP	37073 ms	46459	2158462617	20

Table C.17: Mean response times of NFFI Web Service - WiFi 2 test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	51	5198	54	64805
Proxy with HTTP	45	5736	52	67172
Proxy with HTTP & GZIP	15	3846	13	8548
Proxy with AMQP	101	12862	111	78455
Proxy with AMQP & GZIP	76	10653	71	16773
Proxy with CoAP	0	0	0	0
Proxy with CoAP & GZIP	14	1863	12	6061

Table C.18: Wireshark analysis of NFFI Web Service - WiFi 2 test

C.5.2 RESTful Car System

Test	Mean	STD	Variance	Test runs
Without proxy	8132 ms	7853	61661813	20
Proxy with HTTP	7259 ms	1764	3111671	20
Proxy with HTTP & GZIP	8611 ms	2815	7924419	20
Proxy with AMQP	85609 ms	26355	694606921	10
Proxy with AMQP & GZIP	76636 ms	34666	1201698634	10
Proxy with CoAP	24183 ms	14067	197893185	10
Proxy with CoAP & GZIP	21096 ms	11300	127698638	10

Table C.19: Mean response times of RESTful Car System - WiFi 2 test

	Client -> Web service		Web service -> Client	
Protocol	Packets sent	Bytes sent	Packets sent	Bytes sent
Without proxy	32	6 136	39	11 065
Proxy with HTTP	37	12 434	30	16 596
Proxy with HTTP & GZIP	31	9 575	28	13 901
Proxy with AMQP	332	49 793	317	65 154
Proxy with AMQP & GZIP	231	34 501	243	54 626
Proxy with CoAP	18	6 895	15	10 640
Proxy with CoAP & GZIP	24	7 730	17	8 566

Table C.20: Wireshark analysis of RESTful Car System - WiFi 2 test

C.6 Combat Net Radio Tests

- Ping measured to ~200 ms.
- Iperf3 measured data rate: 41/36 Kbits/sec.

C.6.1 NFFI Web service

Test	Mean	STD	Variance	Test runs
Without proxy	44332 ms	773	597167	10
Proxy with HTTP	48434 ms	3255	10595445	10
Proxy with HTTP & GZIP	5696 ms	522	272157	10
Proxy with AMQP	0 ms	-	-	1
Proxy with AMQP & GZIP	13241 ms	1071	1147182	10
Proxy with CoAP	48302 ms	1046	1095139	10
Proxy with CoAP & GZIP	3803 ms	1218	1482324	10

Table C.21: Mean response times of NFFI Web Service - CNR test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	70	5831	71	64836
Proxy with HTTP	66	6670	67	71551
Proxy with HTTP & GZIP	14	2847	13	7104
Proxy with AMQP	0	0	0	0
Proxy with AMQP & GZIP	56	8697	62	15253
Proxy with CoAP	103	7718	103	57745
Proxy with CoAP & GZIP	11	1652	11	5741

Table C.22: Wireshark analysis of NFFI Web Service - CNR test

C.6.2 RESTful Car System

Test	Mean	STD	Variance	Test runs
Without proxy	4055 ms	960	921629	20
Proxy with HTTP	11478 ms	2842	8077362	10
Proxy with HTTP & GZIP	9526 ms	2701	7292955	10
Proxy with AMQP	41255 ms	3171	10057224	10
Proxy with AMQP & GZIP	36540 ms	3281	10767443	10
Proxy with CoAP	5872 ms	2056	4226612	10
Proxy with CoAP & GZIP	3840 ms	1366	1865202	10

Table C.23: Mean response times of RESTful Car System - CNR test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	25	4738	21	5638
Proxy with HTTP	28	9677	27	15213
Proxy with HTTP & GZIP	24	8473	24	12762
Proxy with AMQP	233	34279	240	54257
Proxy with AMQP & GZIP	220	32420	225	45833
Proxy with CoAP	14	5435	13	9065
Proxy with CoAP & GZIP	12	3919	12	6023

Table C.24: Wireshark analysis of RESTful Car System - CNR 1 test

C.7 EDGE Tests

- Ping measured to ~400 ms.
- Iperf3 measured data rate: 140/97 Kbits/sec.

C.7.1 NFFI Web service

Test	Mean	STD	Variance	Test runs
Without proxy	2437 ms	18	340	20
Proxy with HTTP	2587 ms	40	1583	20
Proxy with HTTP & GZIP	1381 ms	38	1477	20
Proxy with AMQP	9334 ms	65	4216	20
Proxy with AMQP & GZIP	8909 ms	158	24930	20
Proxy with CoAP	40855 ms	46	2151	20
Proxy with CoAP & GZIP	4630 ms	38	1481	20

Table C.25: Mean response times of NFFI Web Service - EDGE test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	50	4531	45	51475
Proxy with HTTP	45	5404	44	55489
Proxy with HTTP & GZIP	14	2847	13	7101
Proxy with AMQP	78	10630	95	64538
Proxy with AMQP & GZIP	59	8871	59	15050
Proxy with CoAP	101	7948	101	56965
Proxy with CoAP & GZIP	11	1660	11	5480

Table C.26: Wireshark analysis of NFFI Web Service - EDGE test

C.7.2 RESTful Car System

Test	Mean	STD	Variance	Test runs
Without proxy	4884 ms	132	17328	20
Proxy with HTTP	5116 ms	139	19459	20
Proxy with HTTP & GZIP	5061 ms	138	18960	20
Proxy with AMQP	35393 ms	764	583712	20
Proxy with AMQP & GZIP	35192 ms	446	199015	20
Proxy with CoAP	5063 ms	59	3488	20
Proxy with CoAP & GZIP	5064 ms	60	3604	20

Table C.27: Mean response times of RESTful Car System - EDGE test

	Client -> Web service		Web service -> Client	
Test	Packets sent	Bytes sent	Packets sent	Bytes sent
Without Proxy	27	4886	23	5570
Proxy with HTTP	28	9677	27	15213
Proxy with HTTP & GZIP	29	8798	27	12941
Proxy with AMQP	194	31292	201	47325
Proxy with AMQP & GZIP	201	31006	212	42611
Proxy with CoAP	12	4761	12	8375
Proxy with CoAP & GZIP	12	3943	12	6068

Table C.28: Wireshark analysis of RESTful Car System - EDGE 1 test

C.8 Tactical Broadband Tests

- Ping measured to ~23 ms.
- Iperf3 measured data rate: 99/82 Kbits/sec.

C.8.1 NFFI Web service

Test	Mean	STD	Variance	Test runs
Without proxy	1379 ms	230	52988	100
Proxy with HTTP	1313 ms	139	19430	100
Proxy with HTTP & GZIP	464 ms	77	5874	100
Proxy with AMQP	2838 ms	318	101162	100
Proxy with AMQP & GZIP	1841 ms	220	48240	100
Proxy with CoAP	2720 ms	120	14457	100
Proxy with CoAP & GZIP	463 ms	25	618	100

Table C.29: Mean response times of NFFI Web Service - Tactical Broadband test

C.8.2 RESTful Car System

Test	Mean	STD	Variance	Test runs
Without proxy	1061 ms	X	X	100
Proxy with HTTP	1522 ms	X	X	100
Proxy with HTTP & GZIP	1404 ms	X	X	100
Proxy with AMQP	7353 ms	X	X	100
Proxy with AMQP & GZIP	7241 ms	X	X	100
Proxy with CoAP	906 ms	X	X	100
Proxy with CoAP & GZIP	840 ms	X	X	100

Table C.30: Mean response times of RESTful Car System - Tactical Broadband test

Appendix D

Source Code

The source code of the different projects are available at the URL provided in table D.1.

Project	**URL**
Proxy	https://github.com/Quist/dil-proxy
Camel CoAP Component	https://github.com/Quist/camel-coap
Car System Service	https://github.com/Quist/master-car-backend
Car System Client	https://github.com/Quist/master-car-client
NFFI Service	https://github.com/Quist/master-nffi-service
NFFI Client	https://github.com/Quist/master-nffi-client

Table D.1: Source code repositories