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

Lorem ipsum dolor sit amet, cu sed suas apeirian, decore iudicabit at per, pro ne lorem dicit dictas. Cu quo aeque maiorum gubergren, principes complectitur ei ius, numquam veritus minimum mel id. Ea ius vidit soleat. Mel timeam laoreet tractatos no.Pro an sadipscing efficiantur, esse ludus diceret nam in. Vis percipit probatus in. Est noster moderatius dissentiet te. Eirmod latine dissentias in sea, perfecto omittantur at duo, mea vide exerci ut. Nec euismod vocibus consectetuer eu.

Et fierent delectus sapientem eam, id eum dolore nullam. Cu his quod possit utamur, mel offendit copiosae forensibus ut, ius fabulas fierent sapientem an. Sed at vidit mentitum expetendis, utamur insolens ad cum, dicat dicta salutatus ei duo. Est tnce e numquam explicari posidonium. Vim amet nostrud at, ea nam graece mediocritatem, cu fabulas maiorum nostrum vix. Ius id zril nullam aperiam, at sint corpora repudiandae eam.

Preface

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 supervison.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Military units operate under conditions where the reliability of the network connection may be low. They can operate far from existing communication infrastructure and rely only on wireless communication. Such networks are often characterized by unreliable connections with low date rate and high error rates making data communication difficult. In a military scenario it is necessary for units at all levels to seamlessly exchange information across different types of communication systems. This ranges from remote combat units at tactical level, to commanding officers at 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, certain military networks suffer from high error rates and very low date rate, which can leave Web services built for civilian use unusable. This thesis investigates how these challenges can be overcome by applying different optimization techniques. The main 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 attribute to making the integration of such systems more difficult. In order 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 is built 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) define 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 chunks of business logic, referred to as *services*. A service can be business related, e.g a patient register service, or a infrastructure service used by other services and not by a user application. OASIS define 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

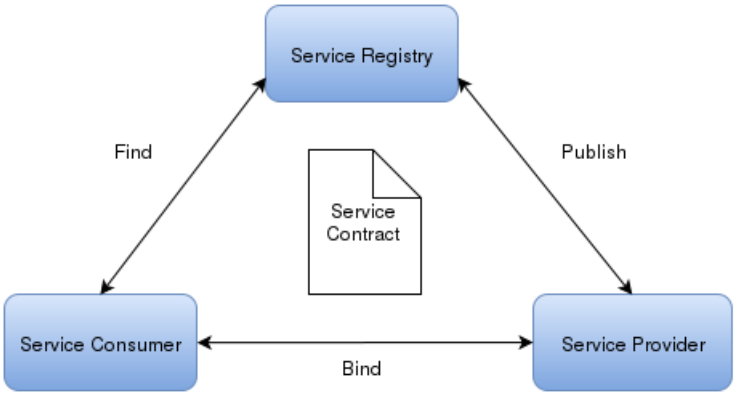

Figure 1.1: The three roles in SOA

Services are provided by *service providers* and are consumed by *service consumers* as illustrated in fig. 1.1. The service provider is responsible for creating a service description, making the service available to others and implementing the service according to the

service description. Services are made available 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 fin d the services they need by contacting the service registry. The communication between services occur through the exchange of standardized messages.

Following the SOA principles dictates 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 with regard to time, which enable services and its consumers to not be available at the same instance of time. This enables asynchronous communication. Loose coupling with regards to location allows 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 dynamic retrieval of the new location of the service.

Furthermore SOA enables service implementation neutrality. The implementation of 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 the concepts can be realized by a range of technologies. The most common used approach is the Web service family of standards, using 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 nations to implement their own technology as long as they adhere to the standards. The Web service technology is discussed in detail in section 2.3. Another approach to realize the SOA principles is Representational State Transfer (REST), an architecture style which has gained a lot of traction in the civil industry and is discussed further in section section 2.3.2.

The mentioned Web service technologies, both REST and W3C Web services, are in widespread use, both in the civil and military world. However, employing Web service solutions directly into military use may not be so straight forward. These technologies were not specifically designed to handle conditions found in certain military networks. In the following sections we discuss characteristics of such networks 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 relatively static users, meaning that they seldom move around or disconnect. At the lower levels, there are fewer units, but they are much more dynamic. The lower level is called tactical networks, which is discussed in the next paragraph.

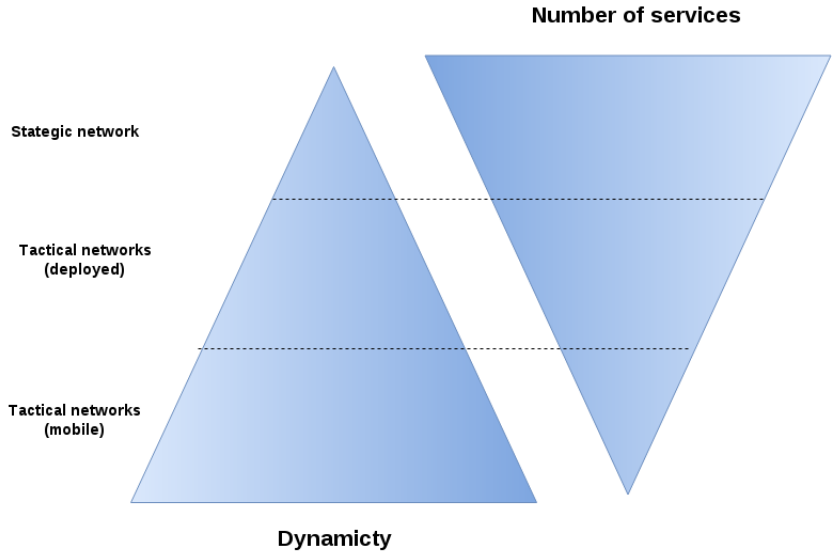

Figure 1.2: Complexity of military networks(from [5])

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 therefor have the largest communication challenges.

In tactical networks in general the users 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. These types of networks are unpredictable and may have very low date rate, possibly high delay, high error rates and frequent disconnections. NATO studies[7] have identified such 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 limited military networks, it is important to understand the limitations we're dealing with. 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 nodes in the network may be disconnected for a long time, possibly for multiple hours and even days.

Intermittent Nodes 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 driving on a countryside road. It may temporary lose connection due to the signal being obstructed by trees beside the road, driving into tunnels or simply having a bad radio signal.

Limited Limited refers to various ways a network can be limited. The Data rate may be low, the network delay may be high and the Packet Error Rate (PER) may be high. The term data rate refers to the amount of data that can be transmitted over a network per unit of time. Delay refers to the time it takes for a bit of data to travel across the network from machine to machine. 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 is transmitted erroneous.

In addition to network limitations, other factors may also restrict the communication for military units. These are discussed in the next section.

Other constraints

As well as being restricted by the communication link itself, military units may have other limitations as well. Consider that military foot patrols have limited battery capacity as they have to carry it with them in their backpacks. The transmission range of the communication equipment for mobile units may also be limited. Another factor that comes into play for military units is that in some cases they are required

to enter radio silence in order to avoid being detected by the enemy. During such circumstances the soldiers may only receive data, but not send any.

1.2 A suggested solution

The Web service technology enables interoperability between systems, but also increase the information overhead, requiring higher data rate demands. Employing Web services developed for use in civilian networks directly into a DIL environment may not perform satisfactorily. To increase the performance we can apply different optimization techniques. The NATO research group with the title "SOA Challenges for Real-Time and Disadvantaged Grids"[6] (IST-090) have previously investigated which improvements that could be made in order to get SOA applicable at the tactical level. They did not find a magic bullet that would solve all problems, but identified factors that would offer measurable improvements. The most important findings were:

- Foundation on open-standards.
- Ease of management and configuration.
- Transparency to the user.
- The Web services should 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 that 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 tactical network would require modification. This would require a lot of resources and severely limit the flexibility of using Web services. Another possible solution is, by using proxies, applying the optimization in them without altering the Web services themselves. 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. This approach is identified by IST-090 and is explored in this thesis. Figure 1.3 illustrates a setup like this, where the client invoke Web services through a proxy pair over a DIL network.

1.2.1 Proxies

A proxy is an application which acts as an intermediary between a client and a server. Proxies are widely in use and their usage and type varies. Example of proxy usage is for load balancing, caching and security. Web

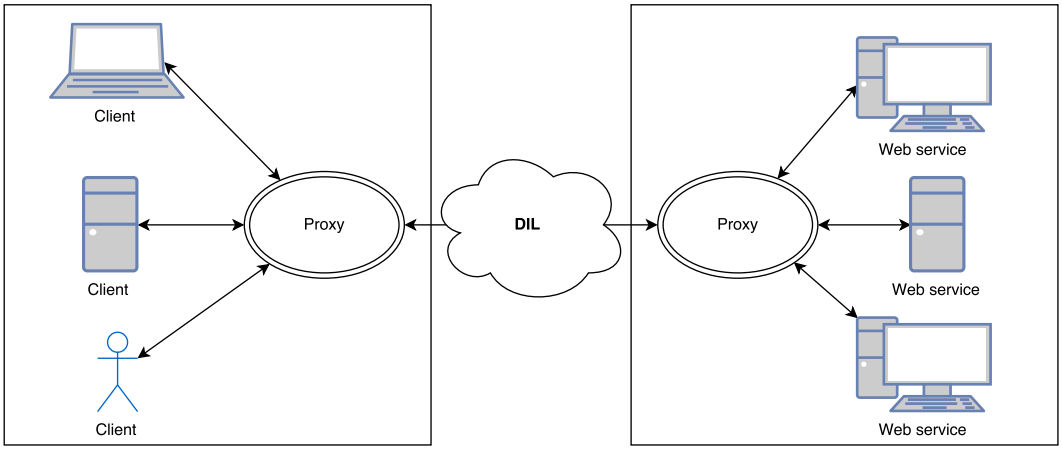

Figure 1.3: Proposed proxy solution

proxies are proxies that forward Hypertext Transfer Protocol (HTTP) requests, which is what we are investigating in this thesis. This proxy will support features for compression, delay tolerance and overcome network disruptions.

1.3 Problem Statement

Most of the Web service solutions used today are aimed for civilian use and do not necessarily perform well in military environments. In contrast to civilian networks where the date rate is abundant, mobile tactical networks may suffer from high error rates and low date rate. Adapting Web service solutions meant for civil networks directly for military purposes may not be possible. Therefore, Web services needs to be adapted in order to handle network challenges. However, it can be very expensive to alter existing Web service technology and incorporate proprietary solutions. A NATO research task group has previously identified the foundation on open standards to avoid tighter coupling between service providers and consumers[6]. It is much better to use Commercial off-the-shelf (COTS) software. By placing the optimization in proxies, the Web services can remain unchanged.

The goal of this thesis is to investigate different optimization techniques that can be applied in order to improve Web service performance in DIL networks. In order for the clients and services to remain interoperable the optimization techniques will be placed in proxies. The Web services will communicate as normal, while all network traffic is tunneled through a proxy. The Web service itself does not need to pay attention to the bad connectivity, the proxy will choose the appropriate protocol and configuration.

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 2.3, HTTP is the by far most common transport protocol used by these type of services. The first and second premise is therefore that the proxy must be able to support both HTTP REST and Web service communication between machines connected in a DIL network.

Next, in order to optimize Web services in DIL environments, the applications themselves should not be required to be customized, all optimization should be placed in proxies. This retains the interoperability with standardized solutions (COTS). The forth and final premise is 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. 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 listed here:

1. Support HTTP RESTful and W3C Web services.
2. Work in DIL networks.
3. Interoperable with standardized solutions(COTS).
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 at 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 "everything over IP", a statement describing that all data communication in NATO should occur with IP packets[1]. We therefore limit our optimization possibilities to the mentioned layers.

Since we focus on performance of Web services, security is not the main focus in this thesis. However, 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 approvement process can be very lengthy. It

is therefore a 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 specific 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 occurrence of a particular event or process or phenomenon. It does not necessarily have any practical use. 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 an actual real-world problem of optimizing Web services, thus we performed applied science. To solve this problem we needed a systematic approach of how to perform 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're performing research in the area of Computer Science, which has been 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 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 involves complexity and graph theory. The next process, *abstraction*, deals with modeling potential implementations. *Design* is about 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 problem The main focus of this thesis was how to improve the performance of Web services in DIL networks. In order to do this it was important to get a deep understanding of what the problem was really about, and formulate a hypothesis. We formulated a problem statement in section 1.3 and proposed a solution in section 1.2.

Derive a design specification based on the requirements Based on the problem at hand and with the premises and scope of this thesis, we derived a set of requirements and specifications in chapter 4.

Design and implement the system When we had identified the requirements for the optimization techniques, we designed and implemented them. This step is elaborated in chapter 5.

Evaluate the system When the optimization techniques were implemented as a part of a prototype proxy, it was tested through a series of tests. The purpose of this was to evaluate if we in fact were able to solve the problem we set out to solve. This testing and evaluation is covered in chapter 6 and a conclusion is drawn in chapter 7.

1.7 Contribution

The outcome of this thesis is a recommendation regarding which optimizations techniques can be used in DIL to enhance the performance of Web services. As well as a prototype implementation of a DIL proxy.

1.8 Outline

Hvordan er resten av oppgaven strukturert.

Chapter 2

Technical Background

Before diving into the design and implementation of the proxy developed in this thesis, in this chapter we present the technical background of the central concepts and protocols. We first give an introduction to computer networks in general and how they are organized. Then we look into two very common communication patterns. Next, we discuss common Web services used for exchanging data in military systems. Then we look into a number of protocols that we can replace HTTP/TCP with in order to increase the performance of Web services. Finally, we introduce the concept of performance testing and network metrics.

2.1 Network layers

To reduce design complexity, networks are organized into layers, each one built upon the one below it. In the Internet Protocol Suite[10], networks is divided into 4 layers. As stated in the scope of this thesis, we only look into optimization techniques for the application and transport layer.

Application Layer
Transport Layer
Internet Layer
Link layer

Table 2.1: The layers of the Internet Protocol Suite

Link layer

The lowest layer is the link layer, where link refers to the physical network component used to interconnect nodes in a network. Link layer protocols operate between adjacent network nodes. An example of a link layer protocol is Ethernet.

Internet Layer

Where the link layer is only concerned of moving data over a wire to an adjacent node, the Internet layer is concerned of 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. Guaranteed delivery is usually handled on the higher levels of the Internet Protocol Suite.

The core protocol of the Internet layer is Internet Protocol (IP) and its routing function enables sending data over interconnected networks.

Transport layer

In the Internet protocol suite model, the transport layer provides end-to-end communication services for applications. It builds on top on the network layer, and takes responsibility of sending data all the way from a process on a source machine to a process on the destination machine. 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 transmission is lost or received in the wrong order, this is all handled by the transport protocol. This provides an important abstraction for applications so that they don't need to deal with the characteristics of the physical network itself.

Application layer

The top layer is the application layer and is where applications real user use reside. The other layers provide transport services to applications found in this layer. When we talk about application layer protocols, we 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 is HTTP and File Transfer Protocol (FTP).

2.2 Communication patterns

Request-reply

Request-reply is a common message pattern where a requester sends a request to a system. The system then process the request and responds with a response message.

Publish-subscribe

Publish-subscribe is a message pattern where subscribers express their interest in a type of messages, often called topics or classes. A message publisher then creates messages of a certain class and publishes them

without knowing who is actually subscribing to these types of messages. Many publish-subscribe system employs 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.

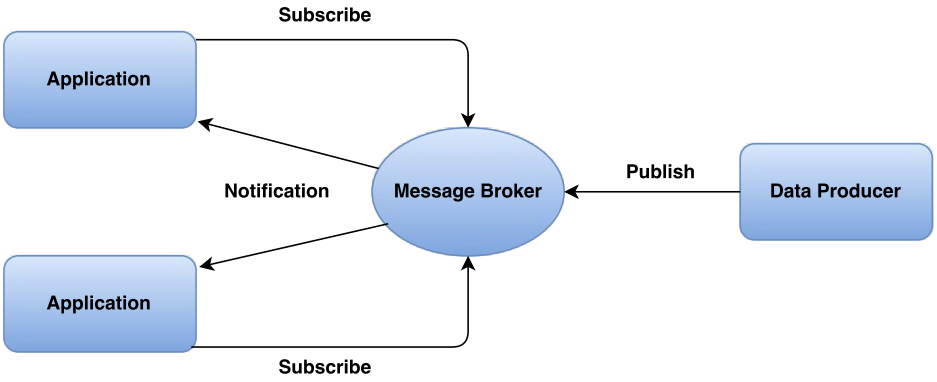

Figure 2.1: Message Brokers

2.3 Web services

Web services are client and server applications that communicate over a network and can be used to realize a SOA. Web services are critical in any data systems and are in widespread use in both civilian and military systems. It is a broad term and can be used to describe different types of services that are available over a network. The most common usage of the term refers to the World Wide Web Consortium (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.

2.3.1 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 enables machine-to-machine interactions. Web service interfaces is described in documents called WSDL, and communication is based on sending XML-based SOAP messages. There exists many definitions of Web services where the core principles are the same, but the finer details may vary. Figure 2.2 illustrates these fundamental principles. Web service technology is a realization of the SOA principles, which provides loose coupling and eases integration between systems.

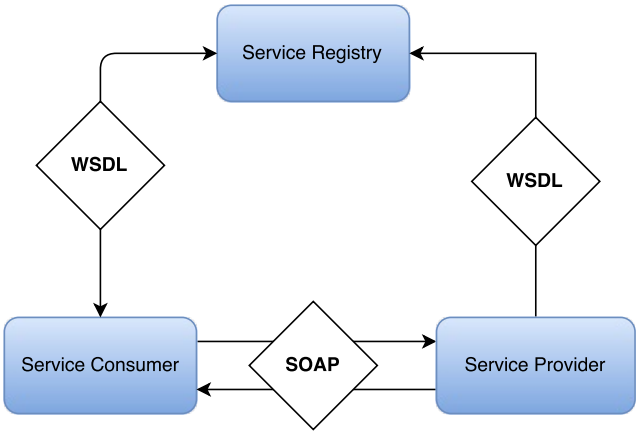

Figure 2.2: W3C Web services

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

Extensible Markup Language

The Extensible Markup Language (XML)[12] is considered as the base standard for Web services. An XML document consist 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 describes available functions, data types for message requests and responses, binding information about the transport protocol, as well as address information for locating the service. This 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[14] in the implementation of Web services. Data communication in SOAP is done by nodes sending each other whats called 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, known 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.

2.3.2 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 does not follow the 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[15] is what we refer to as the REST architectural style. 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. While W3C Web services is service oriented, we can look at REST as being more data oriented.

Web services that adhere to the REST style is 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. Resources are identified by a resource identifier. Table 2.2 illustrates how a component exposes a set of operations of the car resource.

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 downside to NATO with REST is that it lacks standardization, which may cause interoperability issues.

In the next section we will look into HTTP, which is closely associated with REST and is the most used transport protocol for W3C Web

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

services.

2.4 Hypertext Transfer Protocol

As we have seen in the previous sections, both RESTful and W3C Web services utilize the HTTP as their way to communicate 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 with HTTP/2.0. It is a request-reply protocol which means that all data exchanges are initiated with a client invoking a HTTP-request and then waits until a server responds with a HTTP response. A HTTP-request consist of the request method, Uniform Resource Identifier (URI), protocol version, client information, and a 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 a 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/reply 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 actually transfer data to an 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.

2.4.1 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 field it supports.
GET	Retrieve information identified by the resource indentifier(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

2.5 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 most often uses TCP as its transport protocol. In this section we present the characteristics of TCP and some of the issues we may encounter working with it.

2.5.1 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 transfered. A connection is specified by a pair of sockets identifying its two sides. Associated with each connection TCP initialize and maintains some status information for each connection. This includes window size, socket information and sequence numbers.

Computers supporting TCP have a piece of software which manages TCP streams and interfaces to the IP 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 IP layer. The pieces are called TCP segments, which consist of a fixed 20 byte header, followed by zero or more data byte. 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 sent immediately as it may be buffered before its sent. At the receiver, data is delivered to the TCP software, which reconstructs the original byte streams and deliver them to the target application.

2.5.2 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 in to play. For those 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 number to each packet being sent. It then requires a positive acknowledgement from the receiver that the data was actually received. If the acknowledgement 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.

2.5.3 Flow Control

If a fast receiver sends data faster than a slow receiver is able to process, the receiver will be swamped with data and may experience serious 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 acknowledgement sent back to the sender, the window is specified. This allows the receiver to control which segments, and how fast, the sender can send.

2.5.4 Congestion Control

Congestion control is about controlling the data traffic entry into a network in order 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 then 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 aims 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 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 its packet rate.

2.5.5 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 use a lower data rate. This cause that TCP sends 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].

2.6 Protocols of Interest

Since TCP may be sub-optimal or even break down entirely in DIL networks, we're in this thesis looking into alternative protocols and other optimization techniques. In networks with low data rate,

protocols with low overhead per IP packet is beneficial. With frequents 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. Of this follows that we will evaluate protocols in the transport and application layer of the Internet Protocol Suite.

In the following sections we will give a short introduction to the protocols we're investigating in this thesis. The protocols have been selected because of their prevalence in the civil and military world or their reported performance in the "Internet of Things". We'll get started by discussing User Datagram Protocol (UDP), which alongside TCP is one of the core protocols of the Internet protocol suite.

2.6.1 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 more 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, does UDP only support the sending of IP datagrams. Furthermore it is a connectionless protocol, which means that the protocol can send messages *without* first establishing a connection. Since UDP does not provide guaranteed delivery or in-order delivery of messages, it should only be used by applications that does not require this.

To summarize, UDP is a more lightweight protocol than TCP. It has smaller headers and less overhead, which makes it a faster protocol. 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 an UDP stream are competing with a TCP stream[22]. While a TCP stream will detect congestion and back-down its traffic, an UDP stream will greedly 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're looking into.

2.6.2 The Constrained Application Protocol

The Constrained Application Protocol (CoAP) is a specialized Web transfer protocol designed for use with constrained nodes and networks[23].

It is designed for machine-to-machine applications, typically in the Internet of Things. Furthermore it is designed with a similar interface as HTTP, in order to easily integrate with Web services. CoAP is based on the REST model, where the server makes resources available under a resource identifier(URI). Clients access these resources using the HTTP-verbs GET, PUT, POST and DELETE. CoAP main features includes:

- UDP transport with optional reliability supporting unicast and multicast requests.
- Asynchronous message exchanges.
- A stateless HTTP mapping, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP.
- Low header overhead and parsing complexity.

CoAP works similar to HTTP in the way that they use a client-server interaction model. CoAP requests is sent from a client to request an action on a resource located on a server. The server then responds with a response code and a possible response body. Unlike HTTP which uses TCP as its transport protocol, CoAP uses UDP. Since UDP does not guarantee delivery, CoAP provide mechanisms for optional reliability.

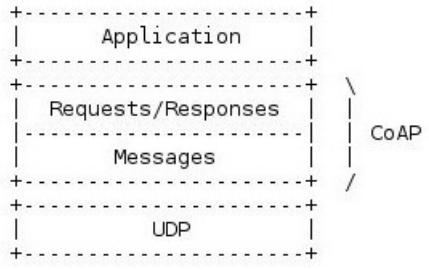

Figure 2.3: Overview of CoAP

2.6.3 Advanced Message Queuing Protocol

The Advanced Message Queuing Protocol (AMQP) is an application layer protocol for sending messages. It support both 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[24], and is a more narrow protocol as it only defines the network wire-level protocol for the exchange of messages between two endpoints. Wire-level protocols

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 consist 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 lives inside *containers*, which can be client applications and brokers. Each container can have multiple nodes. AMQP version 1.0 is does not specify the internal workings of those nodes, but defines the protocol for transferring messages between them. The basis data unit in AMQP is called a *frame* and is used to initiate, control and tear down the transfer of a message between two nodes. The 9 different frames are listen in table 2.4.

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

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 2.4: AMQP Frames

2.6.4 MQTT

MQTT is like AMQP also a publish/subscribe messaging transport protocol [25]. It emerged in 1999 and recently became an OASIS

standard in 2014. MQTT is considered to be light weight 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. With the emerge of "The Internet of Things", these properties have caused regained interest in MQTT. The protocol is broker-based and runs on top of the TCP/IP protocols.

The protocol provide 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 2.5. Only *exactly once* QoS requires the usage of the control packets PUBREC, PUBREL and PUBCOMP.

MQTT Control Packet	**Purpose**
CONNECT	Client request a connection with 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 2.5: MQTT Control packets

2.6.5 Stream Control Transmission Protocol

The Stream Control Transmission Protocol is a transport-layer protocol, which offers functionality from both UDP and TCP[26]. The motivation behind the protocol was that many developers found TCP too limiting, but still required more reliability that UDP could provide. Stream Control Transmission Protocol (SCTP) tries to solve these issues. It is

message-oriented like UDP, but ensure reliable, in sequence transport of messages with congestion control like TCP. SCTP is a connection-oriented protocol and provide 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 link(s). Multi-streaming refers to SCTP ability to transmit several independent streams of data at the same time, for example sending an image at the same time as a HTML Web page.

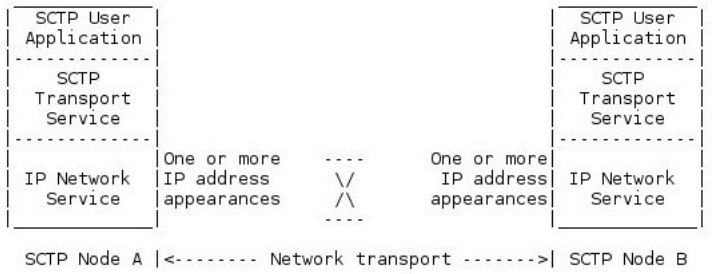

Figure 2.4: Overview of SCTP

2.7 Performance testing

To determine which optimization techniques that have a positive effect on the performance of Web services in DIL environments, we do performance testing.

2.7.1 Network metrics

Network metrics are used to describe various aspects of data transfer from a point to another.

Data throughput The data throughput is influenced by how large distance there is between the nodes communicating.

Reliability How much of the arriving data that is correct. This is called *bit error rate* or *packet error rate*. With high error rates, more data to be transmitted again due to the data arriving being incorrect. This contributes to longer transmission time. In a military setting, an enemy may deliberate sabotage the network with jamming, causing higher error rates.

Latency The communication technology in use influences how fast data transmission can be done. Long delay may cause that the application sending data times out.

2.8 Summary

In this chapter we have presented computer networks in general, before we discussed the two most common type of Web services. Moreover, we have discussed the protocols that these Web services use in order 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 2.6. Finally, we introduced the concept of performance testing and important network metrics when doing such testing.

Protocol	**Network layer**	**Summary**
TCP	Transport	Core transport protocol.
UDP	Transport	Low overhead, but lacks reliability.
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.	Designed for use in the Internet of Things.
AMQP	Application. Uses TCP.	Messaging middleware with store-and-forward capabilities.
MQTT	Application. Uses TCP	Light weight and simple pub/-sub protocol.

Table 2.6: Summary of protocols

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

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 critical for both civil and military users as increasing the performance means that applications become faster and more reliable. For these reasons quite amount of research has been done in the area of optimizing network applications, both for "regular" networks and DIL networks.

In the following sections we identify results and recommendations that are applicable 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 a 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(DIL). 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 to this thesis with regards to the scope and premises.

3.1 Making SOA applicable at the tactical level

IST-118 has published a report 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 in order to apply Web services in tactical networks[6, 27]:

1. End-to-end connections

Web services mostly use transport protocols that depend on a direct, end-to-end connection between the client and the service. Attempting to establish and maintaining connections in DIL environments can lead to increased communication overhead and possible complete breakdown of communication. Most Web services use TCP as their transport protocol,

which requires an uninterrupted connection in order to function. 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 2.5.5. 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 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 solution, standards compliance can be retained.

2. Network heterogeneity

Another issue is when heterogeneous networks are interconnected. Different performance in networks may lead to buildup of data in buffers, risking loss of information. A proposed solution to this is to have store-and-forward support, which can support that messages are not dropped, but stored and forwarded when possible.

3. Web service overhead

W3C Web services are associated with a considerable amount of overhead. Web Service technology is based on SOAP, which use XML-based messages. It is a textual data format and produce 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 approach is to reduce 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-reply type of clients and Web services. Furthermore, since we are investigating general-purpose optimization techniques without knowledge of the payload,

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. Reducing the amount of bits we need to transmit would greatly increase performance, especially in a limited network. The reduction is often expressed in the term *compression rate*, which express 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 loosing some of the data is not critical. Lossless compression utilize repeating patterns in the data in order to represents the same data 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[28]. 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 was the XML-specific XMLPPM and general-purpose GZIP.

In their evaluation they saw that for all techniques, larger XML documents granted 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

Previous studies have investigated potential gains from replacing HTTP/TCP with alternative protocols [29]. They 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 the lower bandwidth links the protocols tends to generate more overhead than TCP. They pointed out that this was due to SCTP having a more complex connection handshake procedure and in addition use heartbeat packets.

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 group and a set of proxies for optimizing Web services in DIL networks already exist. However, many of them does not fulfill all the requirements we have for our proxy. Some of them does only support SOAP Web services and others are unusable due to security reasons. This section lists and discuss previous implementations of such proxies.

3.3.1 Delay and disruption tolerant SOAP Proxy

Delay and disruption tolerant SOAP Proxy (DSProxy) is a proxy solution developed by FFI[30][31]. Its goal was to enable the usage of unmodified standard W3C Web services (SOAP over HTTP/TCP) in DIL environments. The concept was to route all SOAP messages through the proxy. When the proxy received a message, it was stored locally before it was forwarded. If the forwarding failed, it could retry the request until it eventually succeeded. This ability, called *store-and-forward*, was one of the fundamental core functionalities of DSProxy. When a request eventually succeeded, the response was returned to the client on the original TCP connection initiated by the client. By doing this, Web service invocations was made possible over unreliable networks by hiding any network disruptions from the client.

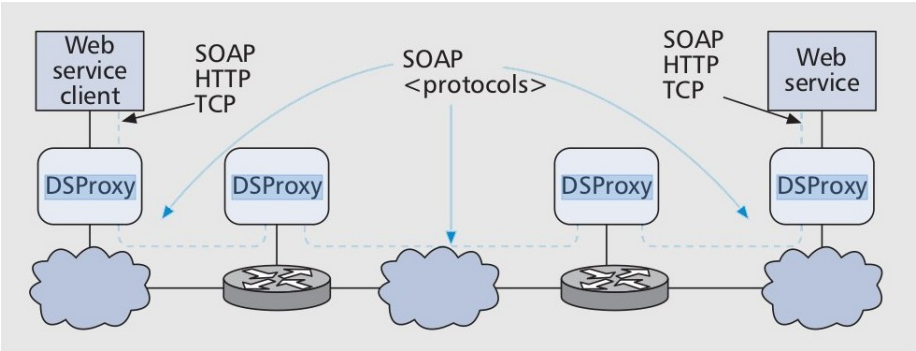

Figure 3.1: DSProxy overlay network (from [31])

Another core functionality of DSProxy was mechanisms for organizing an overlay network consisting of multiple proxy instances as seen in fig. 3.1. This enabled the ability to traverse multiple and heterogeneous networks, but also added a lot of complexity to the proxy application. Apart from the mentioned core functionalities, DSProxy supported a set of pluggable functionalities such as GZIP compression and caching.

After performing experiments using the DSProxy, the researcher identified the store-and-forward ability as very important in unreliable networks in order to avoid having to re-establish end-to-end connections

each time the network connections was lost[30]. The downsides with DSProxy was that it only supported W3C Web services, which leaved other types of Web services out of the picture. Moreover, it became very complex due to its mechanisms for building overlay networks and supporting different configurations and plugins.

3.3.2 NetProxy

NetProxy is an another network proxy solution aiming at enabling SOA applications for use in DIL environments[32]. 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, is NetProxy a transparent proxy providing integration between SOA systems without requiring modification of applications themselves. It works by first intercepting all network traffic from the applications and then analysis. Together with this information and information about available data rate, network delay etc, NetProxy decide which appropriate action to take. It can be configured to support protocol remapping by using other protocols than HTTP/TCP. Integrated with NetProxy is a the message-oriented transport protocol Mobile Sockets (Mockets), which is designed to replace TCP and UDP and is targeted for DIL networks[32]. Mockets replaces 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[32]. 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 allow reusing the connections across consecutive requests from various applications.
- Less overhead due to 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 were XML documents, which have a relatively high compression rate.

3.3.3 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 perform 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, checksums are often used. Consequently this solution would not work for such applications.

3.4 Tuning application server parameters

When setting up an application server, several parameters which can affect the performance of Web services running on the application server can be configured. 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 to low timeout on networks with low bandwidth, can potentially flood the network with packets that have timed out. Consideration should therefor be taken when setting 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 Can vary the size of the buffers that hold the request and send data, respectively.

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 in different types of networks and for both SOAP and RESTful Web services.

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. The most important findings are summarized in table 3.1. We first looked into 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 this 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.

DIL Issue	**Findings**
Reduce Web service overhead	Use compression techniques like GZIP or EFX with zip.
End-to-end connection dependency	Use proxies.
Alternate transport protocols	Summary here.

Table 3.1: Related work summary.

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. To recap, the defining premises were:

1. Support HTTP RESTful and W3C Web services.
2. Work in DIL networks.
3. Interoperable with standardized solutions(COTS).
4. Work with security mechanisms.

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

4.1 HTTP Proxy

The first premise implies that our proxy must accept HTTP, as this is the far most used Web service protocol. Furthermore, the third and forth premises have some important important implications for our proxy. Our proxy must be able to accept a 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 are not required to be with HTTP, but rather 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 difficult 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*. Any optimization techniques must be able to handle the implications of these aspects. In the following sections we discuss the requirements each aspect enforces.

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 handle this aspect. By employing proxies, 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. Followingly, the connection between the proxies over a DIL network can still be lost, while still maintaining the connection between the client and local proxy. This means that the proxy must maintain the connection with the local application, while managing loss of connection with the other proxy. When the connection are reestablished, the proxy should continue sending the data and finally delivering a response back to the client.

This requires the proxy to have some sort of redeliver mechanism, which after a configurable amount of time tries to retransmit the data. If still unsuccessful, the proxy waits an amount of time before attempting again. This can be done a configurable amount of times, or indefinitely until the connection is reestablished. One consideration about this is the risk of overflowing the receiver. It is therefore common to use an exponential back off. This mechanisms grants that the proxy tries frequently immediately after loss of connection, but then gradually tries more and more seldom. Different use cases may require different parameters, so back-off and redeliver delay should be able to alter via proxy configuration.

4.2.2 Intermittent

The proxy should handle brief, temporary disconnects that can ocour 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.

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

To improve the performance of Web services in DIL environments, the proxy should support a set of optimization techniques.

4.3.1 Compression

Compression reduces the size of a message sent over a network. In order 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 was that we must support both RESTful and W3C Web services. RESTful services does 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 identified GZIP as the best approach for compression.

4.3.2 Proxy protocol communication

One of the optimization techniques we have identified is the usage of alternative transport protocols between the proxy pair. We introduced a set of protocols in the technical background chapter and discussed previous evaluations using them in DIL networks in last chapter. In the following paragraphs we analyse them for usage in the context of 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-top-end communication over unreliable networks. The less intrusive optimization technique would therefore be that the proxies simply forwards HTTP-requests without using an alternative protocol. Since HTTP is in extensive use in many different type of networks. We therefore recommend HTTP as a possible proxy pair communication method.

UDP UDP is a fast protocol with less overhead than TCP, but lack mechanisms for reliability and congestion control. The lack of reliability could be done at the application level instead, but would require a library on top of it. Furthermore it 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 designed 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 makes the protocol very interesting as the means of communication between a proxy pair.

AMQP AMQP is in widespread use, and offers reliable message transmission. It support both the request-reply and publish-subscribe message paradigms. We therefore recommend AMQP as a possible proxy pair communication method.

MQTT MQTT is a publish-subscribe messaging protocol and is considered as light weigh and simple to implement. However, due to the inter-proxy communication requires an request-reply 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 proxy pair communication.

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

The proxy should support the identified protocols suitable for communication over a DIL network. The recommendations are summarized in table 4.1. For evaluation purposes it should be easily configured which protocols to use.

4.4 Summary

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

1. Receive and forward HTTP requests
2. Retain HTTP request and response headers.
3. Support GZIP compression of payload.

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. Handle frequent disconnects.
5. Handle disconnects over longer periodes of time.
6. Handle very low data rate.
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.

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

Chapter 5

Design and Implementation

Based on the requirements identified in the previous chapter, we're in this chapter introducing the design and implementation details of the of our proposed proxy solution. We get started by discussing the overall design, before we dive into the implementations details.

5.1 Overall Design

In previous chapters we have argued that all optimization techniques should be placed in proxies in order to retains interoperability for COTS applications, as well as to break their end-to-end dependency. Our overall design therefore deploying a proxy pair to facilitate Web communication. The idea is to deploy the proxy pair in two different locations separated by a DIL network. Through the locally deployed proxy can then Web applications proxy all their communication. The proxy will then apply different optimization techniques, and over over a DIL network forward the request to the other proxy, and finally return a response. Ideally should a proxy be deployed as close to its intended user applications as possible, preferable at the same machine.

It is worth noting that since the proxies is designed to accept *all* HTTP requests, they can support any applications that utilize HTTP, including request- response and publish-subscribe applications.

5.2 Design of Solution

5.2.1 Design of HTTP proxy

A deployed proxy is designed to accept arbitrary HTTP requests, possible from multiple clients, and forward them to the other proxy as seen in fig. 5.1. 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.

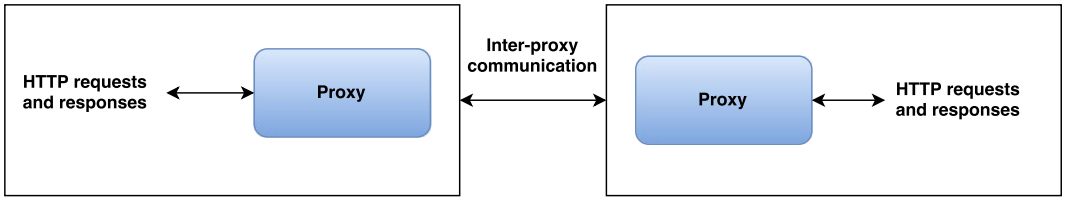

Figure 5.1: Design of Solution

It is the communication between a proxy pair that 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. 5.2. The purpose of this is to evaluate the performance of the transport protocols in DIL networks. Which protocol to use as the means of inter-proxy communication is therefore designed to be easily configured by the user of the proxy.

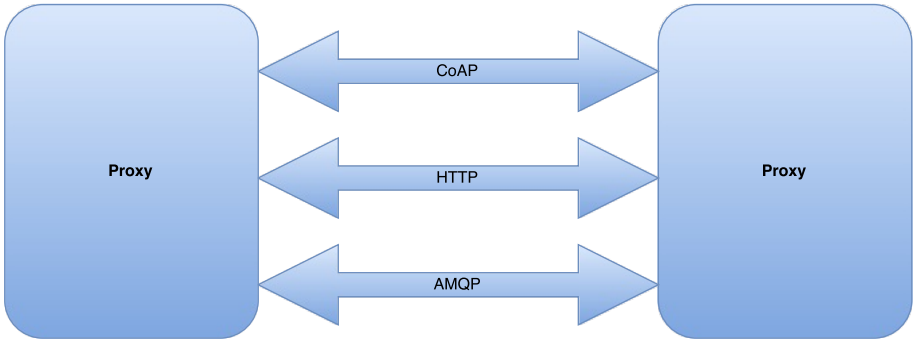

Figure 5.2: The proxies were designed to support multiple protocols for inter-proxy communication

5.3 Choosing a framework

Requirement one implies creating a HTTP proxy which accepts HTTP requests, forward them, and finally return a HTTP response. We identified some approaches:

1. Build a HTTP proxy from scratch ourself.
2. Use an existing HTTP proxy.

Building a HTTP proxy ourself would allow us to customize our solutions as we wanted, but would require a lot of implementation. We therefore concluded that best use of our resources was to use an existing configurable proxy. Building on state-of-art existing solutions allowing us to focus on the optimization techniques, rather than on the specific

low-level details of HTTP. After some research we found three possible HTTP-proxies:

1. Squid
2. Nginx.
3. Apache Camel

Apache Camel is an open source Java framework developed by the Apache Software Foundation for rule-based routing and mediation[33]. It has a wide range of use-cases and focuses on making integration between different enterprise communication system easier. It support 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 different transport protocols.

5.3.1 Apache Camel

Routing is a central concept in Apache Camel and consist of defining a *from route*. This is an endpoint from which Camel consumes messages. It can then invoke a series of *processors*, which can modify the headers, payload etc. of the message. Then, Camel forwards 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. An overview of this can be seen in fig. 5.3.

Describe Camel Components.

5.4 Implementation

The proxy was 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 Apache camel. The stages of the program are as follows:

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

5.4.1 Parsing Configuration

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

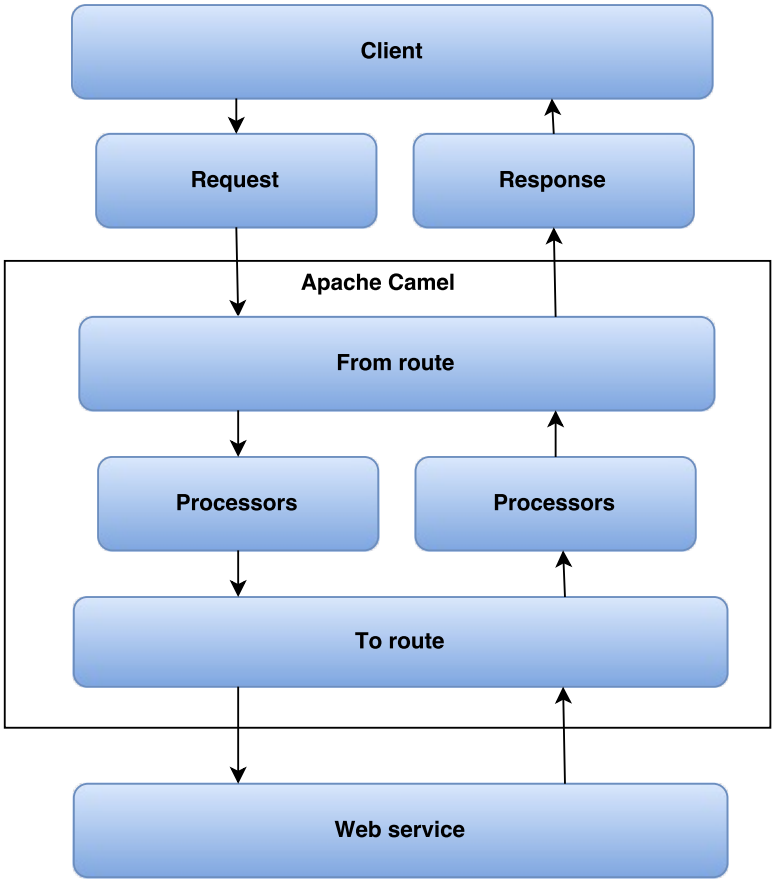

Figure 5.3: Example of a Camel route

5.4.2 Initializing Components

Depending on which protocol the user has selected for usage as inter-proxy communication, the respective Camel component is initiated and added to the Camel context.

HTTP Component

We made use of the Camel component Jetty in order to consume and produce HTTP requests. The component is based on the Jetty Web server[34]. It was used for two purposes: to consume HTTP requests from applications and if HTTP was configured,as the selected protocol, to consume/produce HTTP messages as part of the inter-proxy communication.

AMQP Component

Describe AMQP component.

CoAP Component

Describe CoAP component

5.4.3 Routes

A running proxy listens on two *routes*. It can either receive messages from an application, or it can receive a message from the other proxy. This setup can be seen in fig. 5.4. The routing logic is different for these two cases. We define a request origination from an outside application as the *application route*, and a request origination from another proxy as a *proxy route*. We discuss these routes, but first we need to introduce what we have chose 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 the next section.

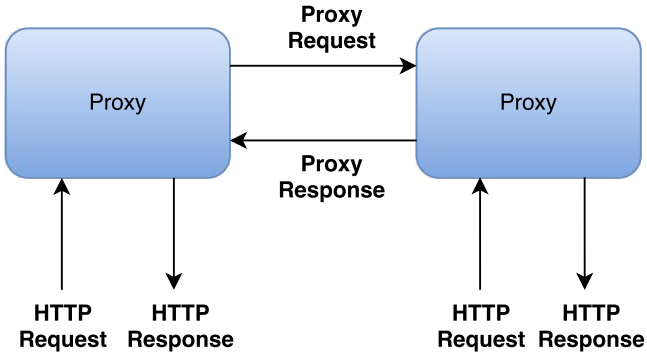

Figure 5.4: Proxy routes

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 JSON document and include necessary information as properties in the JSON document. An HTTP request and response had slightly different requirements, so we used the same format, but with different properties for a request and response.

Proxy Message Request

A proxy request message contains the following JSON fields:

path Contains the original request URL from the application. This describes the intended final destination of the original HTTP request.

method The HTTP method of the request.

query The query string associated with the original HTTP request.

headers JSON object containing all the original HTTP headers of the request.

body Contains the original payload of the message.

An example proxy request message is included in listing 5.1. The listing illustrates a HTTP request originating from an outside application. It was a POST to the intended target was http://myservice.com. The payload was an XML message.

Listing 5.1: "Example proxy request"

```
{
        "path" : "http://myservce.com:8080/",
        "method" : "POST",
        "query" : "?wsdl",
        "headers" : {
                "Accept" : "Accept",
                "User-Agent" : "myuseragent",
                "Authorization" : "Basic
                   QWxhZGRpbjpvcGVuIHNlc2FtZQ=="
        },
        "body" : {
                "<note><to>Tove</to><from>Jani</from><heading>Reminder</headi
                   forget me this weekend!</body></note>"
        }
}
```

Proxy message response

Similar to the request, but contains less fields:

headers JSON object containing the HTTP response headers.

responsecode Contains the HTTP response code.

body The response body of the HTTP request.

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. When a response is received, return it to the application. The route consist 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.
6. Receive an 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 consist of the following steps:

1. Defining a endpoint depending on which 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 application.

5.4.7 Dealing with errors

Discuss retransmission mechanisms

5.4.8 Runtime

In the running stage, the proxy listens on the defined routes and forward them according to the previously configured routes. All requests are logged.

5.5 Functionality

The prototype is packaged as an JAR an can be started from the command line. A path to a valid configuration file must be passed as a command line argument.

5.5.1 Configuration

Listing 5.2: "Example proxy configuration file"

```
proxy {
  useCompression = false
  protocol = "amqp"
  hostname = "0.0.0.0"
  port = 3001
  timeout = 40000
  targetProxyHostname = "192.168.11.10:4001"

}

amqp {
  produceQueue = 4001
  consumeQueue = 3001
  brokerConnectionUri = "amqp://vetur:5672"
}
```

5.5.2 Proxy setup

In order to enable the applications to tunnel all their HTTP traffic through our proxy, we needed a way to setup a proxy without altering the applications themselves. Fortunately, Java provide mechanisms to deal with proxies[35]. 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 5.3: "Setting a proxy on the JVM"

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

In listing 5.3 the application **client.jar** is started and all HTTP traffic will go through the proxy server at localhost on port 3001.

5.6 CoAP component

Elaborate about the custom CoAP component here.

5.7 Summary

In this chapter we presented the design and implementation details of the proxy

Chapter 6

Testing and Evaluation

In this chapter we present how the testing and evaluation of the proxy was performed and present the results we obtained. The goal is to measure any possible improvements (or deterioration) of the performance of Web services when the proxy developed as a part of this thesis is being used. Since the proxy was developed as a prototype for military usage, we wanted to use test scenarios that resemble actual military and civilian usage. For the purpose of testing, we therefore originally developed two sets of applications, one W3C Web service and one RESTful Web service. These applications were then put to test in networks with different characteristics. During testing, we discovered that some of the protocols were very sensitive to the size of the messages being sent. We therefore also developed a complementary test service which allowed us to test sending messages of different sizes.

We'll get started by discussing the test and evaluation tools used, before we introduce the different test applications, test cases and the different types of networks used for testing. Then we present the test results for each of the three aspects of DIL, *disconnected, intermittent* and *limited.* The aspects were tested separately. We started with the disconnected and intermittent tests, where we investigated the behaviour when connection was lost. For the limited tests, we saw how different types of networks influenced the performance of Web services. The base case was to test without any intentional limitations to the network and without the actual usage of the proxy. Then we introduced usage of the proxy and evaluated it in different types of limited networks.

Furthermore we performed the tests with two different setups. First with an emulator that emulates DIL networks, then we supplemented with testing over actual military communication equipment. The usage of actual military equipment allowed us to validate the results testing from the emulation testing.

6.1 Types of DIL networks

Military communication can occur over a wide range of different technologies and environments. These include SATCOM, LOS, CNR and WiFi. WiFi is divided into two types to illustrate both with good connection and one with less. Some communication technology, such as satellite communication, is 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.

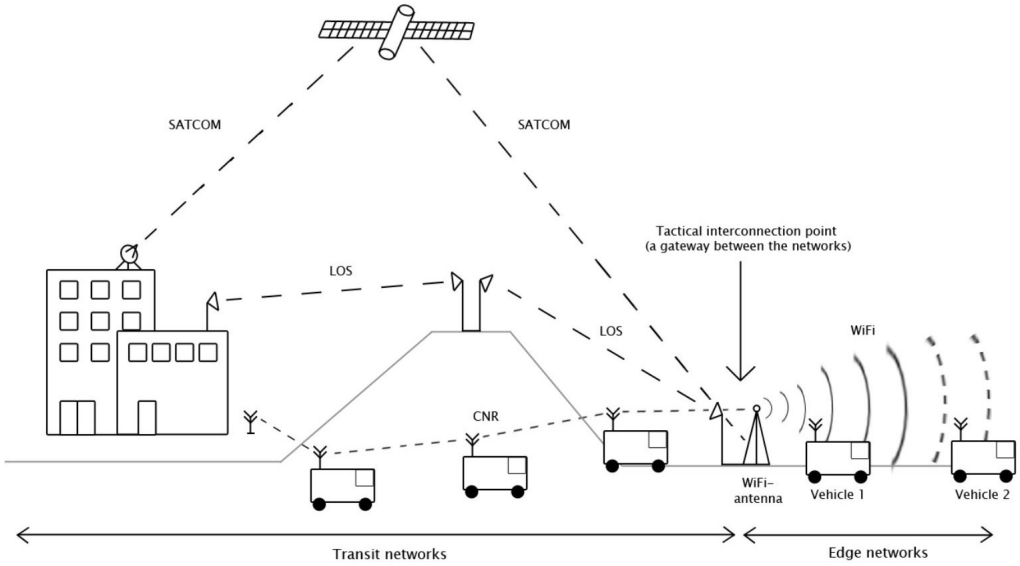

Figure 6.1: Overview of tested networks (from [36])

An infinite number of possible network combinations exists, so we have chosen to focus on five different network types identified by the task group IST-118 for DIL-testing. The networks were identified because they represent typical networks typically found in military communication. 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. Thus making it interesting for us to investigate the performance under this type of network as well. However, we eventually found out that 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 fall back in geographical areas where LTE and 3G is not available. Of the networks we evaluate for, is EDGE the only one with asymmetrical down- and upload speed: 50 kbps up and 200 kbps down.

The different networks and their properties are summarized in

table 6.1.

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

In order to evaluate how our solution impacts the performance of Web services in DIL environments, we needed some way of simulating such environments. Obviously, we would have got the most realistic test environment by testing "out in the field" ourself. However, this would require of a considerable amount of effort and it would be difficult to reproduce the exact same environment and test results. We therefore choose to emulate DIL networks instead. For testing we used two approaches, the first one connecting two machines through a third machine. The third machine used a component in the Linux kernel to control the flow of the network traffic flowing through it, allowing us to simulate DIL networks. The second approach involved using actual military equipment in a laboratory at FFI. The benefit of using actual equipment, is that we got as realistic tests as possible.

6.2.1 Linux Network Traffic Control

The Linux kernel offers a rich set of tools for managing and manipulating the transmission of packets. **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 to packets outgoing from a selected network interface[37]. These tools allow us to emulate different network characteristics, which we use to emulate the networks listed in table 6.1.

How we configure NetEm and the linux traffic control to do this is outlined in the following paragraphs.

Emulating network delays

NetEm can emulate delays on packets on a specific link. In listing A.1 we add a fixed delay on 100 ms to all packets going out of local Ethernet.

Listing 6.1: "Emulating delay"

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

Emulating the data rate

Listing 6.2: "Emulating data rate"

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

Emulating the corruption rate

The corrupt rate allows us to insert random data into a chosen percent of packets.

Listing 6.3: "Emulating corruption rate"

```
tc qdisc add dev eth0 parent 1:1 handle 10: \
  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[38]. 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 in the worst case lead to us drawing invalid conclusions. To confirm and validate our network emulations, we therefore used iPerf 3 alongside the Linux tool *ping* to confirm that the NetEm scripts worked as expected. For reference, we included the bash scripts we used to configure NetEm in the appendix, chapter A.

Cite at iPerf 3 har blitt evaluert til beste måleverktøy. Vurdere å ta med graf som viser de ulike målingene med iPerf.

6.2.3 Wireshark

Wireshark is a packet analyzer and allows for performing network analysis[39]. As an example, this tools allows a user to see all IP packets sent from a machine over the Ethernet interface.

When performing the testing we used Wireshark to monitor the network traffic on the machine hosting the client and the local proxy. This allowed us to investigate the behaviour of the evaluated protocols in the different types of networks. In particular we used it to see how many packets that were sent, as well as total number of bytes that were sent over the network. Moreover, we used it to see how long a HTTP request

from a client was processed by the proxy before it was forwarded. This enabled us to calculate how much processing time the proxy used on a request, thus saying something about the CPU overhead of the proxy

6.3 Test Setup

The majority of testing was performed at the FFI-lab at Kjeller. All the test applications consisted of one client and one Web service, where the client would request the service for some sort of data. The client where hosted on one computer and the service on an another computer. The majority of testing was done using NetEm to emulate DIL networks, and some testing was done using actual military radios. The machines used for testing are listed in table 6.2.

Machine	**Client**	**Application server**	**Router**
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 6.2: Machines involved in the testing

6.3.1 NetEm Setup

In this setup, the client and Web service machines were connected to each other through a third computer, acting as a router. This router machine had two network cards and networked together the other machines by Ethernet cables. The setup can been seen in fig. 6.2. In order for the router machine to forward IP packets back and forth between the client and server, IP forwarding was enabled on the kernel.

The server and client are assigned an IP address in two different subnets. This is done by the Linux network interface administration

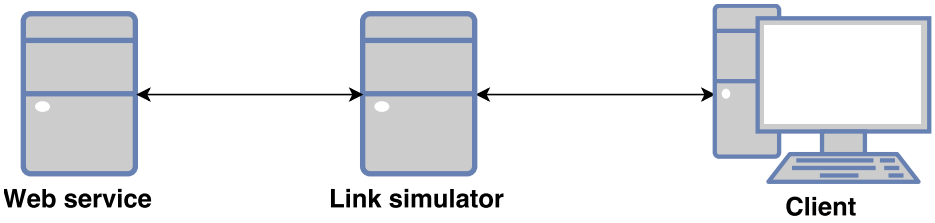

Figure 6.2: Testing environment

program *ifconfig*. In listing 6.4 the client machine is assigned the IP address 192.168.2.44.

Listing 6.4: "Configuring a network interface of the router"

```
ifconfig eth0 192.168.2.1 up
```

After setting up the IP addresses we need to 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 routing machine. In listing 6.5 we configure all IP traffic bound for the subnet 192.168.1.X to be routed through the router machine with IP 192.168.2.1.

Listing 6.5: "Configuring routing rules for the client"

```
ip route add unicast 192.168.1.0/24 via 192.168.2.1
```

Emulating different types of networks

Since all network traffic passes through the routing machine, we can control the flow of IP packets here. As previously discussed, we use NetEm. For each network configuration, a bash script is run. This script configures the network interfaces in order to get the correct network behaviour. Both interfaces are configured so the network is symmetrical in both directions, with the exception of EDGE which has asymmetrical data rates.

6.3.2 Tactical Broadband Setup

The majority of testing was performed over emulated networks. To validate these results we also performed tests on military communication equipment. The setup is illustrated in fig. 6.3. It is point-to-point setup with two radios, without any multi hop functionality. The radios have capacity to work as a multi hop Mobile ad hoc network (MANET), but this was not tested in this thesis.

Skrive kort om kongsberg radioen.

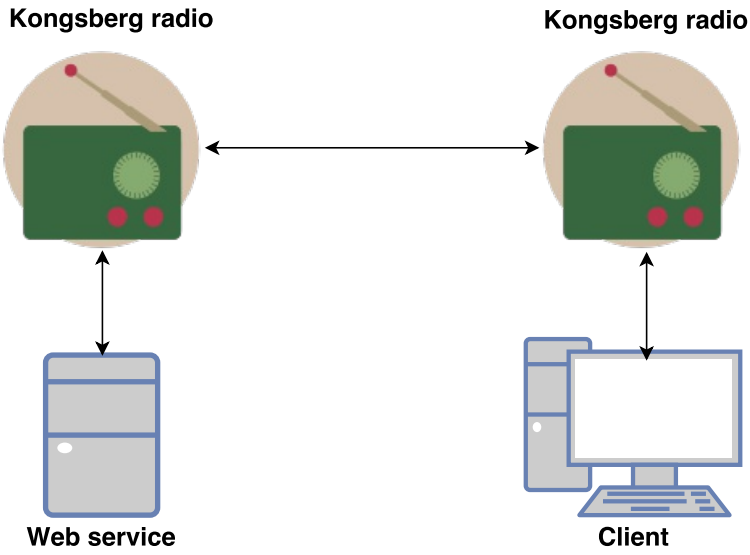

Figure 6.3: Testing environment

6.3.3 Proxy setup

For all test cases, the proxies ran on the same machine as their intended users. The client and its proxy ran on one machine, while the Web service and its proxy ran on another machine. Running on the Web service machine was also the message broker, used by test cases involving AMQP.

6.4 Test Execution

Each test scenario was performed with both a W3C Web service application and a RESTful Web service application. Each service is deployed in Glassfish 4, while the client is executed either from the command line or directly from the Netbeans IDE. Data being sent between the client and server is by default sent uncompressed.

During testing we discovered that especially CoAP was very sensitive to the size of the messages being sent. We therefore also developed a test application that allowed the client to request a number of bytes from the server. This allowed us to see how CoAP performed with different message sizes.

6.4.1 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 reports uses the NATO Friendly Force Information (NFFI) format, which has an associated XML schema

with it. One test run is illustrated in fig. 6.4 and consists of the 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.

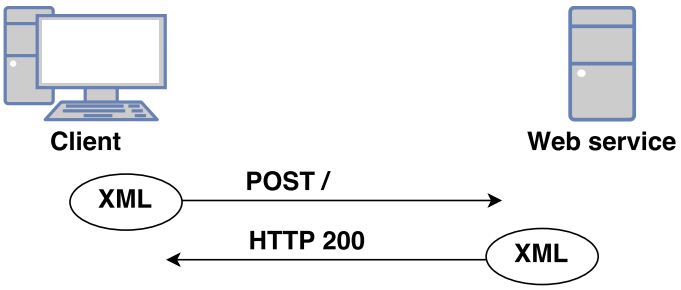

Figure 6.4: NFFI Web service

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 6.3: NFFI Web service HTTP requests

6.4.2 RESTful Car Service

The RESTful Car service is an example service keeping order of cars in a "car system". The service exposes an Application Program Interface (API) which offers different operations 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 an resource. Since RESTful services are payload agnostic, we chose JSON to represent the data being sent between the server and the client. JSON is a lightweight data-format. Each test run consist of a client sequentially invoking the server with different API requests. The most common HTTP-methods GET, PUT, POST, and DELETE are all part of the testing. An example of usage of the Car system is illustrated in fig. 6.5.

6.4.3 Request Size Application

This application allowed us to test with different message sizes.

6.4.4 Test parameters

The tests was performed with the following parameters.

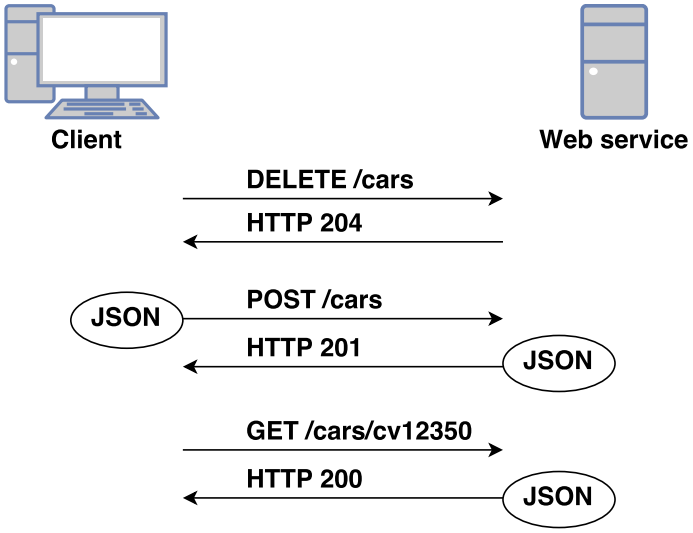

Figure 6.5: RESTful car service

- GZIP compression on/off.
- Without and with proxies.
- Transport protocol used.

6.4.5 Test Applications summary

6.5 Function tests

The first phase of the testing was performed without any actual intended limitations to the network. The objective of this testing is to validate that the proxy is working correctly and have a benchmark to compare other results with. This phase was again divided into two phases, one without the usage of proxy and one with. Doing this allowed us to investigate any potential overhead associated with the usage of the proxy. We used the NetEm setup with a third machine acting as a router, although without any NetEm limitations turned on.

6.5.1 Results and Analysis

Enabling compression yields an improvement in the performance, especially for W3C Web services which had much larger messages. We also notice that HTTP and CoAP has a almost identical performance, while AMQP has significant longer average response time. Furthermore we can observe that the default solution without proxies has the best performance in this unlimited network.

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 6.4: RESTful Car Service HTTP requests

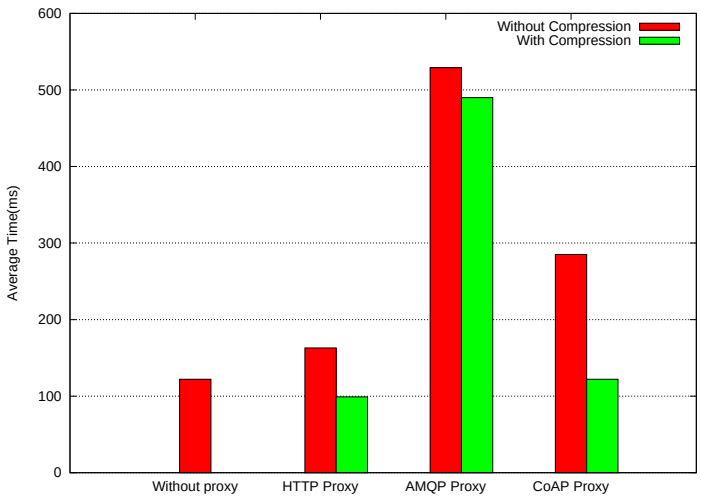

Figure 6.6: Function tests - NFFI Web service

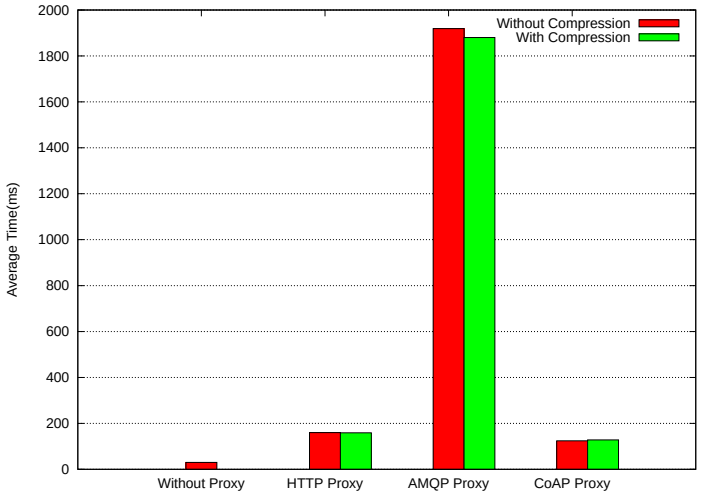

Figure 6.7: Function tests - REST

6.6 DIL Tests - Disconnected

In this scenario we evaluate the performance with the DIL characteristic *disconnected*, which refers to the network suddenly going down when the application is sending data. The objective of this testing is to evaluate how the proxy manages disconnects over longer periods of time. We define the success criteria for this test to be that the client is able to eventually process his request after the connection is reestablished. The client HTTP request should not be interrupted in any way, other than it taking longer time to process the request.

6.6.1 Execution

The tests are performed on an unlimited network. During testing the Ethernet cable between the client machine and the router was removed for about 60 seconds. It was then reconnected.

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. With proxies the connection did not time out, and the protocols retransmission mechanism were able to continue transmission when connection was reestablished.

6.7 DIL Tests - Intermittent

Intermittent refers to the network connection being lost, but then regained again. The objective of this testing is to evaluate how the proxy

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

Table 6.5: NFFI Web service results

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

Table 6.6: RESTful Web service results

manages frequent temporary loss of connections. The success criteria is the same as for disconnected, the client should not notice any disruption of service.

6.7.1 Execution

Not done yet. Similar to disconnect.

6.7.2 Results

Test	Result
Without proxy	X
Proxy with HTTP	X
Proxy with AMQP	X
Proxy with CoAP	X

Table 6.7: W3C Web service results

Test	Result
Without proxy	X
Proxy with HTTP	X
Proxy with AMQP	X
Proxy with CoAP	X

Table 6.8: RESTful Web service results

6.8 DIL Tests - Limited

The third DIL characteristic, *limited*, refers to different ways a network can be limited. This includes high delays, packet loss and low

bandwidth. In this section we present the testing performed for the different types of networks identified in table 6.1.

6.8.1 Satellite communication

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

Results and analysis

AMQP has a very long response time for both test scenarios, while also CoAP struggles with large uncompressed XML messages of the NFFI service. For both with and without compression, we observe that employing HTTP proxies yields a small improvement of performance, compared to the default. We can also notice that for the RESTful service, CoAP has better performance than default, and similar performance to HTTP proxies.

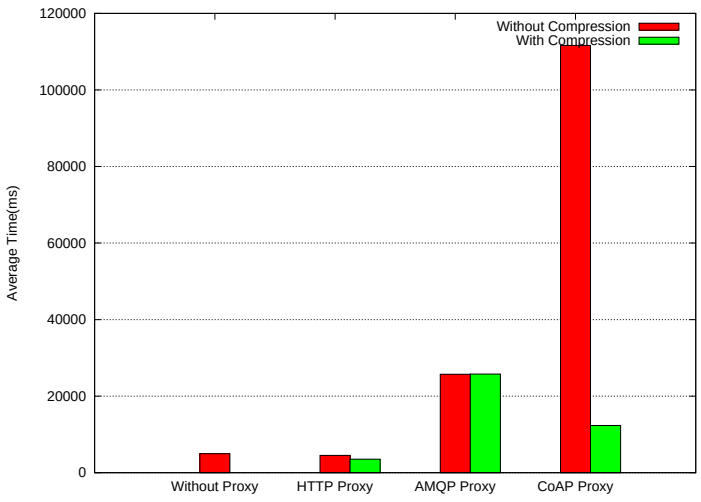

Figure 6.8: Satellite tests - NFFI Web service

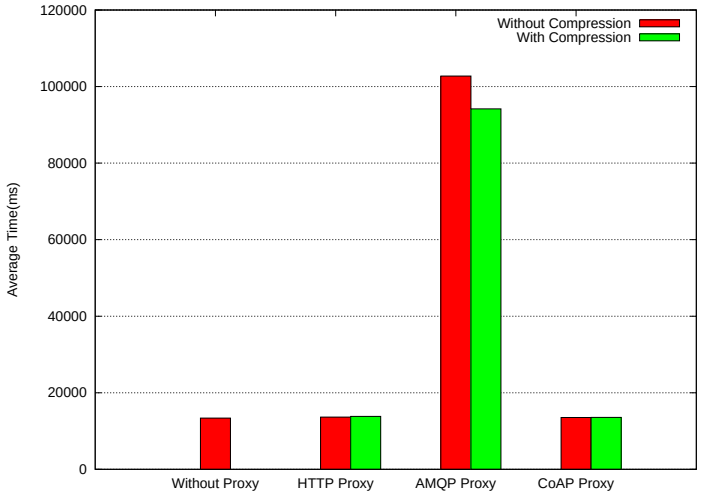

Figure 6.9: Satellite tests - REST

6.8.2 Line-of-Sight

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

Results and analysis

Again we notice CoAP really struggling with uncompressed XML messages, as well as AMQP performing significantly poorer than the other protocols. However, when the message is compressed CoAP has roughly equal performance as the default. When we look on the RESTful test application results, we see that CoAP performs better than default and HTTP with compressed messages.

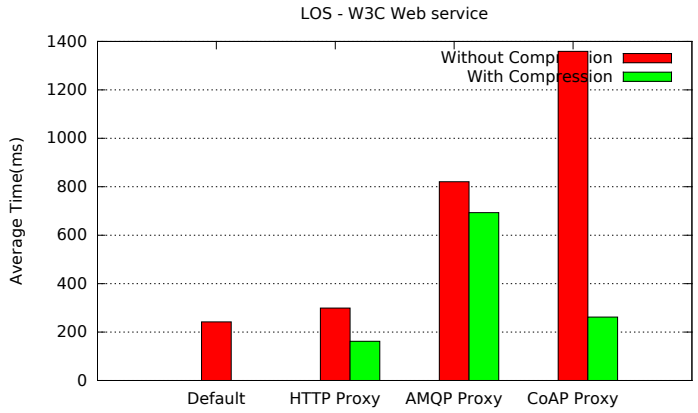

Figure 6.10: LOS tests - NFFI Web service

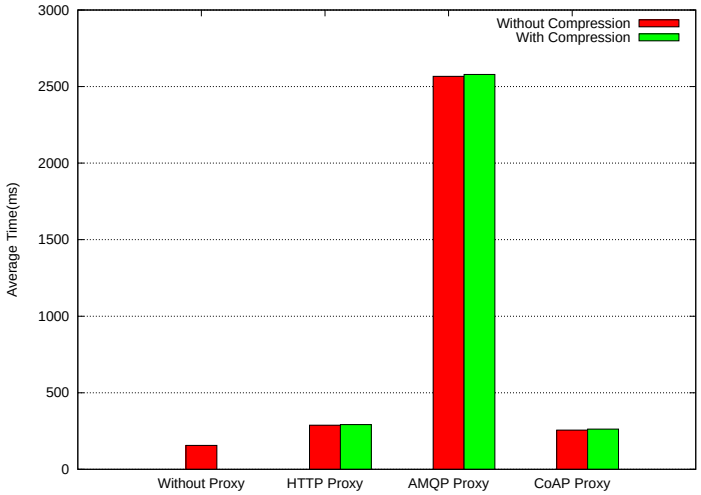

Figure 6.11: LOS tests - REST

6.8.3 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 around 1 %.

Results and analysis

HTTP proxies with compression enabled yields the best performance. CoAP has worse performance than the HTTP proxies, but roughly equal for the compressed REST tests.

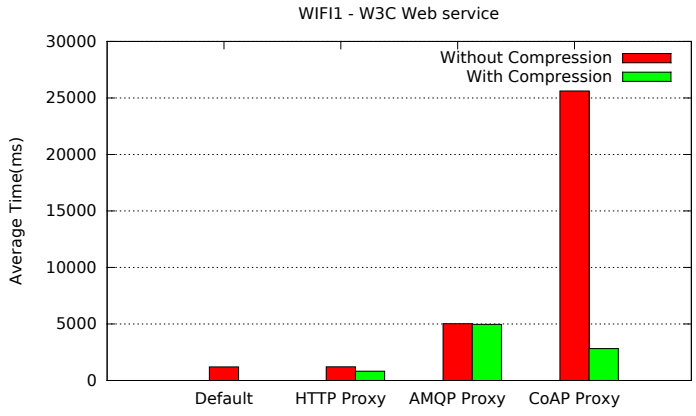

Figure 6.12: WiFi 1 tests - NFFI Web service

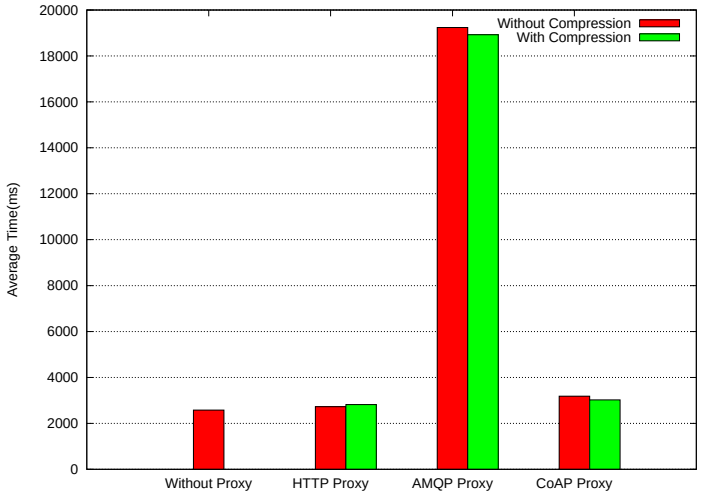

Figure 6.13: WiFi 1 tests - REST

6.8.4 WiFi 2

This type of network also emulate 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

Compared to WiFi 1, we see that all response times has increased significantly. The importance of compression has increased, the tests with compression turned on yields a large performance increase. In the uncompressed NFFI test scenario, CoAP reached it’s time out, and were unable to finish. In the scenarios where CoAP did finish it still performed worse than default and HTTP. HTTP proxies with compression yielded the best results.

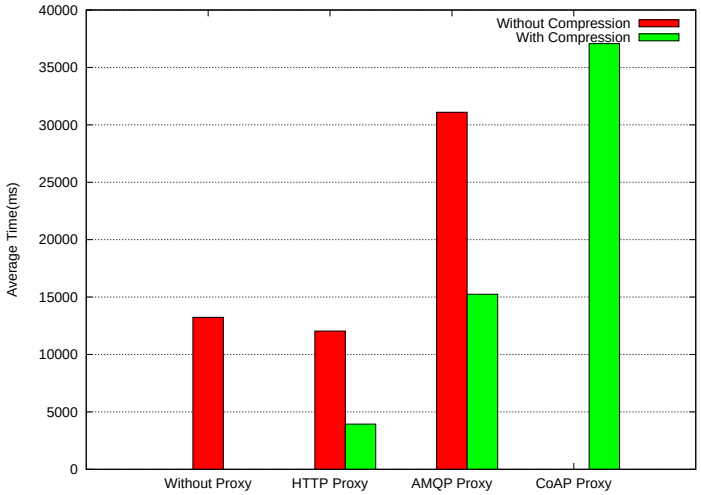

Figure 6.14: WiFi 2 - NFFI Web service

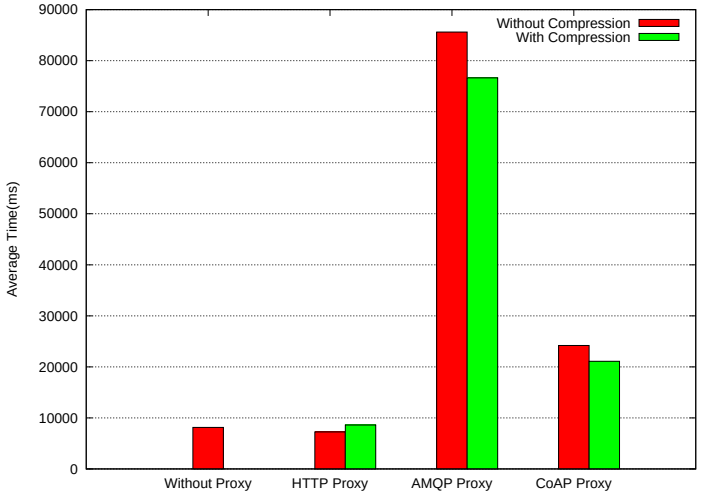

Figure 6.15: WiFi 2 - REST

6.8.5 Combat Net Radio with Forward Error Correction

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

Results and analysis

Again we can observe the importance of compression in this type of networks. CoAP has the best performance.

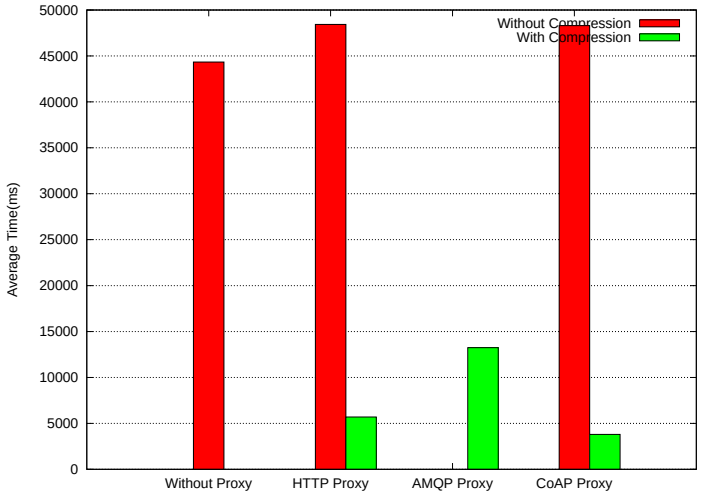

Figure 6.16: CNR tests - NFFI Web service

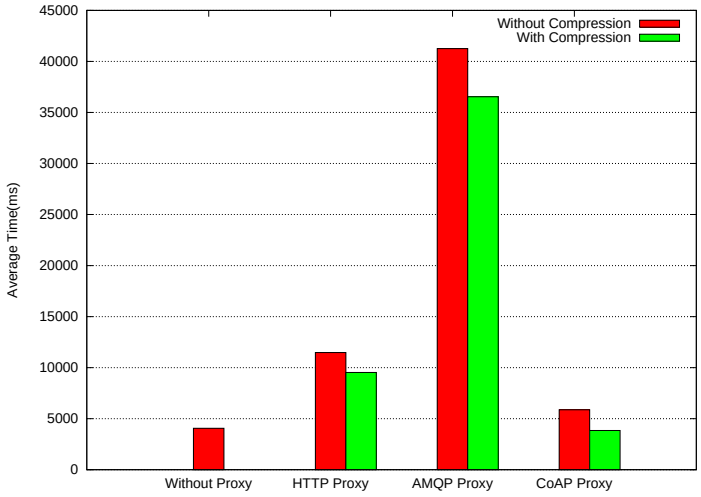

Figure 6.17: CNR tests - REST

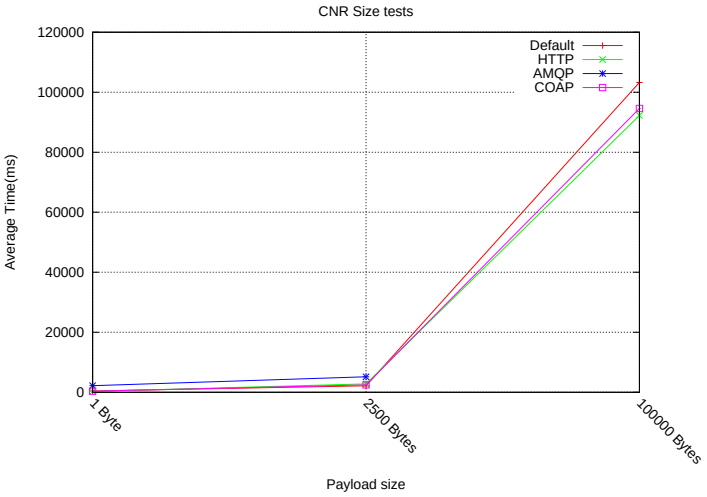

Figure 6.18: Size request results

6.8.6 EDGE

About this type of network.

Results and analysis

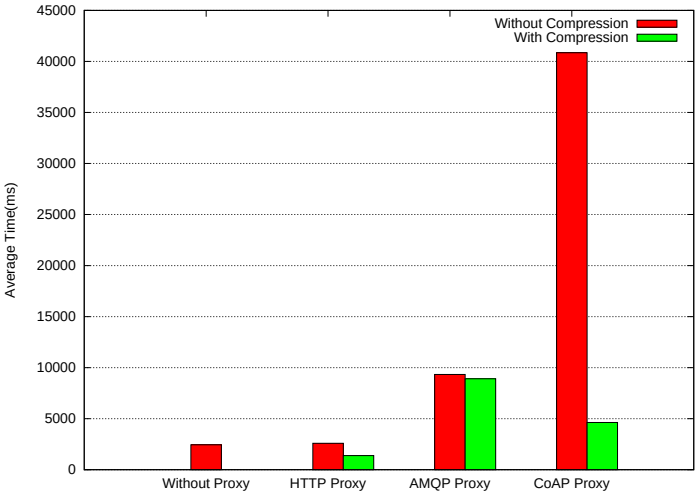

Figure 6.19: EDGE tests - NFFI Web service

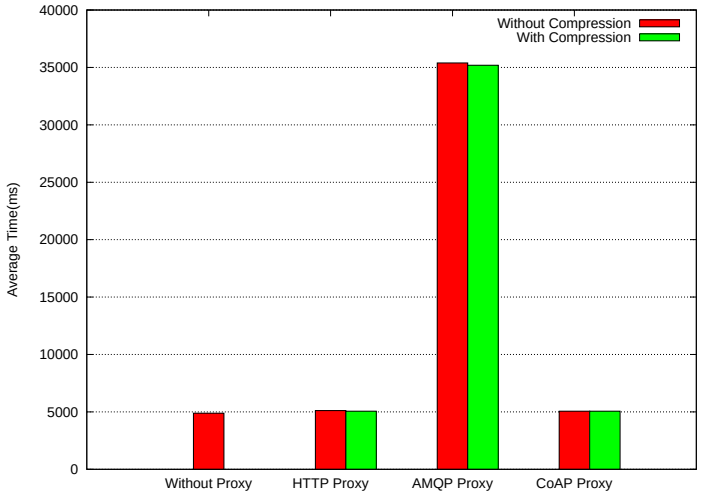

Figure 6.20: EDGE tests - REST

6.8.7 Kongsberg Radio

Two KDA WM 600.

Results and analysis

For NFFI tests, compression yields a lot of increase in performance.

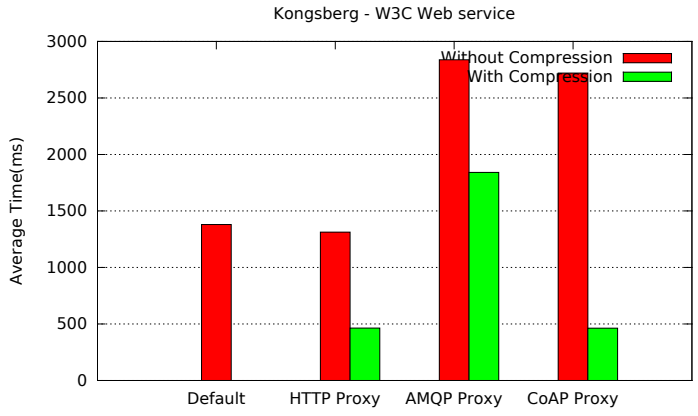

Figure 6.21: Tactical Broadband - NFFI Web service

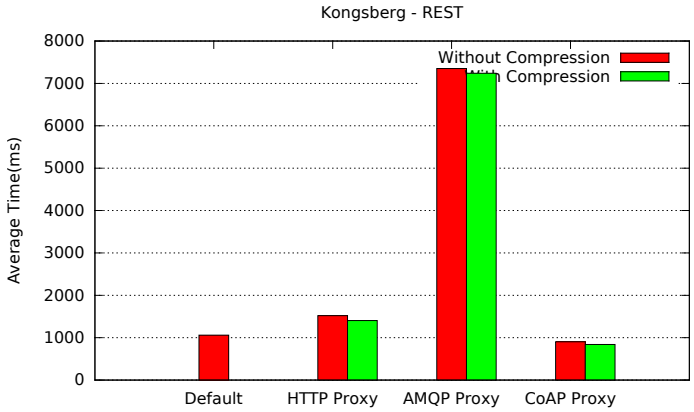

Figure 6.22: Tactical Broadband - REST

6.9 Summary

In this section the results from the tests are presented. These results lead up to the discussion and conclusion in the next chapter.

- AMQP has the worst performance in almost every test scenario.
- Compression almost always yields an performance increase.
- CoAP struggles with larger messages.
- CoAP has the best performance with smaller messages in networks with low data rate.

Network	**NFFI Web service recommendation**	**REST recommendation**
SATCOM	AMQP Proxy	550 ms
LOS	X	X
WiFi 1	X	X
WiFi 2	X	X
CNR	X	X
Edge	X	X

Table 6.9: Recommendations

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Revisit problem statement.

7.2 Future Work

IPSEC.

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] 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).

[25] 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.

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

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

[28] 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.

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

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

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

[32] 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.

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

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

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

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

[37] 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).

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

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

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

Acronyms

ACM Agile Computing Middleware. 45

AFRO Adaption Framework foR Web Services prOvision. 46

AMQP Advanced Message Queuing Protocol. 35, 52

API Application Program Interface. 72

CNR Combat Net Radio. 10, 66, 67, 81, 85, 97

CoAP The Constrained Application Protocol. 34, 35, 52, 71

COTS Commercial off-the-shelf. 21

DIL Disconnected, Intermittent and Limited. 19–22, 41, 50

DSProxy Delay and disruption tolerant SOAP Proxy. 44

EDGE Enhanced Data rates for GSM Evolution. 10, 66, 67, 97

EFX Efficient XML. 43

FFI Norwegian Defence Research Establishment. 5, 44

FTP File Transfer Protocol. 26

HTTP Hypertext Transfer Protocol. 21, 22, 26, 29, 30, 49, 51, 58

IETF Internet Engineering Task Force. 30

IP Internet Protocol. 26, 34

JVM Java Virtual Machine. 62, 63

LOS Line of Sight. 9, 66, 67, 78, 85, 97

LTE Long-Term Evolution. 66

MANET Mobile ad hoc network. 70

Mockets Mobile Sockets. 45

MTU Maximum Transfer Unit. 32

NATO North Atlantic Treaty Organization. 15, 49

NEC Network Enabled Capability. 15

NetEm Network Emulator. 67, 68

NFFI NATO Friendly Force Information. 71

NIO Java new/non-blocking I/O. 47

NTNU Norwegian University of Science and Technology. 46

OASIS Organization for the Advancement of Structured Information Standards. 16, 35, 36

PER Packet Error Rate. 19

QoS Quality of Service. 37, 45, 46

REST Representational State Transfer. 17, 27, 29, 46

SATCOM Satellite Communication. 9, 66, 67, 77, 85, 97

SCTP Stream Control Transmission Protocol. 37, 38, 43, 52

SOA Service Oriented Architecture. 15–17, 27, 28

STD Standard Deviation. 99, 100

TCP Transmission Control Protocol. 26, 31–35, 37, 45

UDP User Datagram Protocol. 34, 35, 37, 45, 52

URI Uniform Resource Identifier. 30

W3C World Wide Web Consortium. 27, 30

WSDL Web Services Description Language. 28

XML Extensible Markup Language. 28, 43

Appendices

Appendix A

Network emulating

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

A.1 SATCOM

Listing A.1: "Emulating SATCOM"

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

A.2 LOS

Placeholder

A.3 WiFi 1

Placeholder

A.4 WiFi 2

Placeholder

A.5 CNR

Placeholder

A.6 EDGE

Placeholder

Appendix B

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[40]. An example of how this was done running NFFI Web service tests can be seen in listing B.1.

Listing B.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.

B.1 Function Tests

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

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

B.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 B.2: Mean response times of RESTful Car System - Function Test

	Client -> Web service		Web service -> Client	
Test	IP Packets sent	Bytes sent	IP Packets sent	Bytes sent
Without proxy	25	4 738	21	5 638
Proxy with HTTP	28	9 677	26	15 147
Proxy with HTTP & GZIP	X	X	X	X

Table B.3: Wireshark analysis of RESTful Car System - Function Test

B.1.3 Request Message

Test	1 byte	2 500 bytes	100 000 bytes
Without proxy	43 ms	8 ms	112 ms
HTTP	53 ms	54 ms	121 ms
AMQP	155 ms	199 ms	283 ms

Table B.4: Mean response times of Request Message - Function Test

B.2 Satellite Tests

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

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

B.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 B.6: Mean response times of RESTful Car System - Satellite test

	Client -> Web service		Web service -> Client	
Test	IP Packets sent	Bytes sent	IP 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 B.7: Wireshark analysis of RESTful Car System - Satellite test

B.2.3 Request Message

Test	1 byte	2 500 bytes	100 000 bytes
Without proxy	2246 ms	2210 ms	3987 ms
HTTP	1121 ms	1121 ms	4035 ms
AMQP	7939 ms	8210 ms	9388 ms

Table B.8: Request message results

B.3 Line-of-Sight Tests

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

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

B.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 B.10: 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 B.11: Wireshark analysis of RESTful Car System - LOS test

B.3.3 Request Message

Test	1 byte	2 500 bytes	100 000 bytes
Without proxy	64 ms	27 ms	420 ms
HTTP	62 ms	68 ms	423 ms
AMQP	214 ms	274 ms	592 ms

Table B.12: Mean response times of Request Message - LOS Test

B.4 WiFi 1 tests

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

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

B.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 B.14: 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 B.15: Wireshark analysis of RESTful Car System - WiFi 1 test

B.4.3 Request Message

Test	1 byte	2 500 bytes	100 000 bytes
Without proxy	467 ms	410 ms	962 ms
HTTP	221 ms	244 ms	1372 ms
AMQP	1605 ms	1666 ms	2295 ms

Table B.16: Mean response times of Request Message - WiFi 1 test

B.5 WiFI 2 tests

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

B.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 B.17: NFFI Web service results

B.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 B.18: REST Web service results

	Client -> Web service		Web service -> Client	
Protocol	IP Packets sent	Bytes sent	IP 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 B.19: RESTful Car System Wireshark analysis

Test	1 byte	2 500 bytes	100 000 bytes
Without proxy	3426 ms	4637 ms	- ms
HTTP	441 ms	1254 ms	19515 ms
AMQP	5490 ms	5186 ms	- ms

Table B.20: Request message results

B.6 Combat Net Radio tests

- Ping measured to ~200 ms.

- Iperf3 measured data rate: 41/36 Kbits/sec.

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 B.21: NFFI Web service results

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 B.22: REST Web service results

Test	1 byte	2 500 bytes	100 000 bytes
Without proxy	447 ms	2179 ms	103260 ms
HTTP	312 ms	2805 ms	92259 ms
AMQP	2204 ms	5183 ms	- ms

Table B.23: Request message results

B.7 EDGE

- Ping measured to ~400 ms.

- Iperf3 measured data rate: 140/97 Kbits/sec.

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 B.24: NFFI Web service results

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 B.25: REST Web service results

Test	1 byte	2 500 bytes	100 000 bytes
Without proxy	847 ms	810 ms	4201 ms
HTTP	427 ms	426 ms	4229 ms
AMQP	2925 ms	2972 ms	5963 ms

Table B.26: Request message results

B.8 Military radio tests

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

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 B.27: NFFI Web service results

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 B.28: REST Web service results