Hovedopgave
Master i Informationsteknologi – Softwarekonstruktion

An upgrade of a legacy web service infrastructure

Af
Morten Poulsen
Studiekort nr. 20080582
porten@cs.au.dk

9. maj 2013

Morten Poulsen, studerende
Anders Møller, vejleder
Abstract

In this project, I examine the existing Telenor DK web service infrastructure. This infrastructure is in many areas outdated, and has known issues.

Relevant aspects of architecture and associated issues are described, and solutions for the issues suggested.

Two main upgrade-paths are laid out: upgrading the existing legacy SOAP frameworks to more-recent editions, or using REST instead.

In order to understand and describe the REST approach, a literary study of REST and a concrete REST implementation, JAX-RS, is carried out.

The project uses methods from my previous courses at the Master of IT line at Aarhus University, which are XML- and Web technologies (2010), Software Architecture in practise (2011) and IT-security (2012).
1 Introduction

1.1 Motivation

At Telenor DK, we have a complex and ever-growing set of internally maintained systems collectively dubbed “middleware”. As the moniker implies, these systems act as intermediaries between other systems, typically large 3rd party systems such as internal billing engines or external data providers like the Danish CPR registry. The purpose of these middleware-systems vary from implementing business processes, to exposing business-rule hardened custom API layers on top of other systems, when these do not natively expose a usable API.

Over time, communication between these systems has changed from platform-dependent binary protocols to WS-I SOAP web services. This transition began approximately 10 years ago, and many of the decisions made in those early days still leave their mark on the Telenor system stack today.

Specifically, we still use some of the same discontinued 10-year-old web service frameworks to expose and consume web services today, as we did then. Efforts to upgrade these frameworks to more modern (and maintained) alternatives have been put on hold, since it is unclear which, if any, framework best support our intended future use of web services, while still providing backwards compatibility with the services we for the time being are unable to upgrade. Rather than risking an un-maintainable web of ad-hoc chosen frameworks, we have a standing decision to use the already in-place frameworks, until a proper analysis of framework alternatives can be carried out.

This leaves the current system stack in an undesirable state with known communication issues, and no resolution in sight.

In light of the known SOAP issues, competing technologies – REST in particular – are making their way into the Telenor system stack, adding further complexity to an already complicated infrastructure.
1.2 Problem statement

The purpose of this project is to identify and describe solutions to the problematic communication infrastructure outlined in the Motivation section.

Concretely:

- I will describe the current architecture, its shortcomings, and the requirements that have led to the current web service based architecture.
- I will show how alternative SOAP web service frameworks can solve the problems of the current architecture. This will be done through experimental prototyping of the proposed alternatives.
- I will show if and how a REST based architecture can be used instead of the current SOAP based architecture. This entails a literary study of REST, and a comparison of REST versus SOAP for the Telenor-specific use. A REST-based exploratory prototype will be built, for comparison with the SOAP prototype.
- I will summarize the proposed architectural changes, and finally produce a recommendation of how to evolve the current Telenor middleware.

1.2.1 Scope

The project scope is deliberately limited in the following areas:

Representative Telenor systems subset

Telenor has a vast collection of systems performing diverse tasks. Many of these systems could rightfully be called “middleware” systems as per the [WIKIMIDL] definition, since their primary task is often not providing functionality to end-users, but to act as “software glue” between other systems. In this project, I will only concern myself with the web service based subset of middleware systems belonging to the Business Software Solutions (BSS) department. This subset consists of less than 10 systems, with 3-4 primary server systems, and the rest being secondary or client-only systems.

The issues and solutions found for this area will be representative for other systems relying on the same technologies elsewhere in Telenor, since most other WS-* / SOAP communication in Telenor is based on the BSS-area architecture and framework choices.

IT security concerns

All communication issues investigated in this project are between Telenor-internal systems with no connection to public networks. Security concerns are limited to identification of valid users. All systems mentioned, and all users accessing these systems, are considered part of the trusted computing base [ITSIKNOTE].
1.3 Method

The overall project goals are described in the Problem statement section. I will be using the following method to accomplish these goals.

1.3.1 Describe the as-is architecture

Firstly, I will present the as-is architecture, using architectural description methods from the “Software Architecture in Practise” course [SAIPCS]. This description will focus on the runtime communication protocols employed by the individual systems (the Allocation and Component & Connector viewpoints [SWDESC]), with the purpose of mapping interaction-types between the relevant web service frameworks and applications. The individual services, their business purpose and functionality is not relevant in this context.

1.3.2 Describe the Telenor web service issues

Based on the as-is architectural description, I will detail the known problems and incompatibilities. We do not currently have much in-house knowledge about web service protocol specifics at Telenor, so this problem description will serve a secondary purpose of knowledge build-up, allowing us to make informed decisions about web service configurations in the future, rather than just using whichever default settings the framework provides.

1.3.3 Define a representative environment

From the as-is architecture and web service issue listing I will select key technologies (platforms, frameworks and protocols), and define a small-scale environment using these technologies in a way similar to the as-is architecture.

The technologies in this environment will be selected so key problem areas can be prototyped, using both SOAP and REST alternatives to as-is. The goal is to be able to “plug in” alternative SOAP frameworks, or re-write communication from SOAP to REST, while still being able to relate the outcome to the as-is architecture.

This setup will be runnable on a regular desktop PC. Scalability and performance under high load will not be testable by prototypes in such a simulated environment.

1.3.4 Prototype SOAP framework replacements

With Telenors as-is SOAP architecture, and the costs associated with major architecture rewrites in mind, the first part of the solution analysis will focus on how to progress with the current SOAP technologies.

This means investigating how the concrete web service framework issues described in the Telenor SOAP problems section can be solved via framework upgrades. To this end, I will use experimental prototyping to determine which of the framework candidates provide the best alternative to our current outdated Axis framework.

As the web service issues section will show, Axis is in itself only part of the problem. Axis is used in multiple roles across several systems today, and a complete Axis phase-out is not necessarily required. Therefore, the SOAP prototyping will not have “remove Axis” as a goal, as much as “remove problems (which are often caused by Axis)”.
1.3.5 REST as a SOAP alternative

REST-advocates often claim that much of the overhead and complexity inherent in WS-* / SOAP does not exist in REST.

To determine if REST is a viable SOAP-alternative (for Telenor specifically), I will first perform a literary study of REST. Through this study, I will produce a description of what REST is, how it compares to SOAP, and which key elements of SOAP and REST are actually relevant to Telenor. The literary study will be based on Roy Fielding’s REST dissertation [RFREST], derived material by others, and commercial REST implementations such as JBoss RESTEasy [RESTEASY].

This study leads to an experimental prototype demonstration of how REST would handle certain problematic areas described in the Telenor SOAP problem breakdown.

1.3.6 Choosing a solution

With an (expected) list of multiple possible technical solutions to the issues plaguing the as-is architecture, I intend to use quality attributes [SAIPBOOK] to evaluate a) if REST is viable, or we should stick to SOAP, and b) which SOAP framework is the best candidate as an Axis replacement.

1.3.7 To-be architectural description

Finally, once the recommended architecture candidate(s) are identified, I will present an updated architectural description detailing what changes are to be made compared to the as-is architecture.

If deemed necessary, this architectural description will be supplemented with an upgrade-path from as-is to to-be.
2 As-is architectural description

As with many existing software systems in the industry, the Telenor DK middleware architecture is constantly evolving, well known (by a lucky few), but never fully and thoroughly described.

The existing architectural descriptions are primarily purpose-specific tools used during projects, and are discarded (or at least, not updated) when the project completes. These descriptions bring little value to this project, and will not be used.

Instead, this architectural description describes the relevant areas from scratch, using methods from “An approach to Software Architecture Description using UML” [SWDESC]. This approach is a concrete specification of the architectural description recommended practices from IEEE Std 1471-2000. The viewpoints defined in this article are (quote):

- “A Module viewpoint concerned with how functionality of the system maps to static development units,
- a Component & Connector viewpoint concerned with the runtime mapping of functionality to components of the architecture, and
- an Allocation viewpoint concerned with how software entities are mapped to environmental entities”

2.1 Purpose and stakeholders

The purpose of this architectural description is to enable discussion about the problems present in the current Telenor web service setup, not to convey a deep understanding of the Telenor system stack itself.

The intended recipients (stakeholders) for the architectural description are architects or developers with an understanding of service-oriented architectures (SOA) [WIKISOA], but no prior knowledge of the Telenor system stack.

This applies to all viewpoints.

2.2 Viewpoints used in this description

The focus of this project is on communication/protocol issues, not the functionality of the applications themselves. Because of this, most of the Module viewpoint – such as class & package hierarchy descriptions – is irrelevant. Individual classes, packages and similar static model elements are not described.

The Component & Connector viewpoint describes runtime interactions between components including the protocols in use, which is a central topic. Typically, the Component & Connector viewpoint will also describe how/when components interact (for instance via sequence diagrams). Such interaction descriptions are vital for understanding how a system functions, but understanding the individual systems is not the purpose of this description. Therefore, such
diagrams are not included, and the Component & Connector viewpoint will only describe the components, and their interconnecting protocols.

The Allocation viewpoint describes the platforms/hardware on which the individual components reside. This determines available technologies (f.ex. Java 4, Java 6, or .NET?), and is thus also relevant for building the simulated prototype environment.

2.3 Allocation viewpoint

2.3.1 Allocation viewpoint concerns
The Allocation viewpoint can describe the physical hardware entities on which software components (systems) are deployed, or other relevant environment factors.

Rather than going into detail with the hardware, this viewpoint describes the platforms on which systems are running.

A comment on physical hardware
All platforms are running on clustered or loadbalanced hardware with sufficient resources. Because of this, the physical hardware of the entire as-is architecture is considered a secondary concern, and is not described in detail.

Previous analysis has determined that performance bottlenecks are caused by poorly scaling software, not lack of hardware. As a concrete example of “sufficient hardware”, one of the oldest key components (NewDelfiAPI, on the WLS platform) is running on 2 loadbalanced 64-core SUN UltraSparc-T2 servers, and rarely manages a load-average above 3 on either server (i.e., 6 of 128 available CPU cores in use). Architectural design flaws such as repeated (serial) transformation of large XML documents is not realistically solved by additional hardware processing power.

2.3.2 Deployment diagram
The following diagram shows platforms and systems as UML Nodes and Components, respectively. The diagram contains the relevant system stack, as defined in the Scope section. Each individual platform and system is described in more detail in the next section.

We are interested in the direction of SOAP communication – i.e., which systems consume services from which other systems. Most systems have a clear client/server relationship, but in some cases systems will act as both client and server – though not necessarily during the same call sequence. The sequence of calls is (as per the Component & Connector viewpoint scoping) not interesting in itself – the fact that systems communicate at all is enough to warrant a look at the protocols and technologies involved.
Most systems communicate exclusively via SOAP web services. For sake of readability, the “SOAP” label is omitted on the connectors, except the places where not only SOAP is used.

In the following sections, all mentioned platforms and systems are briefly described. The purpose of this description is to show how the entire infrastructure relies on SOAP web services, but the individual systems have different frameworks available for utilizing SOAP.

### 2.3.3 Platforms types

A “platform” in this context is an application deployment environment, typically a commercial implementation of an industry standard such as Java JEE [WIKIJE]. Each platform hosts at least a single “system” (or “application”). Certain platforms host multiple systems, but only a few are relevant/depicted above.

The modelled platforms are divided into two main types: Java JEE and .NET (IIS), with Java JEE platforms further divided by vendor and version.
Java Enterprise Edition - JEE

Java (2) Enterprise Edition is a server platform based on Oracle's Java Virtual Machine (JVM). The JEE platform can be seen as an extension of the regular Java Standard Edition (JSE) most people have installed on their desktop PC's. JSE allows execution of Java programs within a local Java Virtual Machine (JVM), both as web browser plugins and desktop applications.

A JEE server/platform is in itself a Java application that allows other applications to be deployed and executed within the confines of this platform-application. Various Java JVM language versions and JEE specification versions allow for many different combinations of available language- and platform features between implementations.

The JEE platform provides application-server oriented capabilities such as database connection pooling, transaction handling and message queues. Applications deployed to a JEE platform are often developed using Enterprise Java Beans (EJBs). The JEE specification is huge, and only a handful competing business-grade implementations exist.

.NET / Internet Information Services

The .NET platforms are full-blowed application servers as well, similar in purpose and features to the JEE servers. In general, JEE is considered more backend-centric with great support for large-scale backend applications, whereas the .NET platforms are considered superior for web-application hosting.

In the scope of this project, the .NET platforms can be considered pure client-systems (to the rest of the middleware, at least), and are only included for sake of context. The web service client frameworks in use from .NET are capable of consuming the troublesome RPC/encoded services, so there are no known “.NET-specific” issues.

Therefore, the .NET platforms and applications will be described in less detail, and not be part of prototyping setups.

As for JEE vendors and versions, Telenor use Oracle WebLogic 8 (a quite old release) and JBoss Application Server 5 (a more recent release). A transition from WebLogic to JBoss is on-going, expected to complete during 2013.

Because of this on-going transition, limitations on the older WebLogic platforms (such as Java 4 / JEE 1.3) should not influence choice of solutions.

2.3.4 Platforms

The individual platforms in Figure 1: Deployment diagram are:

WLS: WebLogic Server

Oracle BEA WebLogic 8.1 J2EE 1.3 Application Server (2003).

The WLS (short for “WebLogic Server”) is a J2EE 1.3 compliant application server, currently owned and maintained by Oracle. It is an old platform (2003), with very limited built-in web service tooling. The Java language version is Java 4 (more precisely; Java 2 version 1.4). This version is ancient by software development standards, and lack language features such as generics and auto-boxing. This outdated language version greatly impacts 3rd party tool availability.
**WLI: WebLogic Integration**  
*Oracle BEA WebLogic 8.1 J2EE 1.3 Application Server (2003).*

The WLI (short for “WebLogic Integration”) is a proprietary workflow engine with an accompanying GUI for modelling workflows. The WLI workflow engine is itself a J2EE application running on top of WLS, but has its own dedicated hardware and is considered an entirely separate platform in Telenor.

**EAP: JBoss Enterprise Application Platform**  
*Red Hat JBoss Enterprise Application Platform 5.1 (2009).*

The EAP is a JEE 5 compliant application server, hosting applications that have been migrated from the WLS platform. The EAP platform provides a wide range of web service related tools supporting the JAX-WS and JAX-RS specifications.

**SOA-P: JBoss Enterprise SOA Platform**  
*Red Hat JBoss Enterprise SOA Platform 5.1 (2009).*

Like the WLI/WLS, the Enterprise SOA Platform or “SOA-P” (not to be confused with “SOAP”) builds on EAP and provides functionality beyond that of the JEE standard, most notably a workflow engine (“jBPM”) and an Enterprise Service Bus (ESB) implementation.

These are “inner workings” of the SOA-P platform – any exposed web service relies on the same technologies as the EAP, so from a web service prototyping point of view, there is no perceivable difference between EAP and SOA-P.

**IIS: Internet Information Server**  
*A Windows Server 2003-based .NET 3.5 Application Server using MS Internet Information Services 6 to host web applications. As mentioned in the Platforms types section, although this is a complex platform in itself, it will be considered a simple client throughout this project.*

**Windows PC**  
*A regular Microsoft Windows XP or Windows 7 desktop PC in various configurations, on which the only direct GUI client (“Opus”) to the middleware is running. This client is a Visual Basic (VS2005) application, relying on the Microsoft Web Service Enhancements (WSE) 3.0 framework to consume web services. This is essentially the same framework used from the IIS platform, so these two client platforms – although quite different – share all relevant web service client system characteristics.*

### 2.3.5 Systems

A description of the role, background and technology for each system mentioned in *Figure 1: Deployment diagram:*

**NewDelfiAPI**

Originally designed as a J2EE application for BEA WebLogic 6.1 (2001) this is one of the oldest applications in the system stack. The purpose of this application is to expose a business-rule hardened API layer on top of the Tuxedo (ATMI) services provided by the Telenor-
customized Amdocs Ensemble billing engine called “New Delfi”. This system contains most of the data in use throughout the architecture – data such as GSM subscriber phone numbers, subscription types, name and address etc.

As such, NewDelfiAPI is at the “bottom” of the middleware architecture, having no dependencies to other middleware systems.

It is a SOA application consisting of around 80 Stateless Session Beans, with a total of ~150 services.

The Stateless Session Beans are exposed via RMI, and as SOAP web services, using Apache Axis 1.4 [AXIS1].

All SOAP web services use the RPC/encoded binding style.

**WLI**

The WLI system as a whole is the combined set of workflows running on the WLI platform. These workflows were originally intended as integration services for clients requiring combined data from various backend systems, first and foremost NewDelfiAPI.

There are two distinct categories of WLI workflows: “Channel” and “Order Engine”. As they share identical technology (only differ in purpose), they are not considered distinct systems. This is contrasted by the JBoss-migrated edition of Channel and Order Engine workflows, which are based on separate technologies.

Each workflow is implemented as an XML-document describing the workflow decision points, with interleaved Java snippets. These workflow definitions are compiled by a custom BEA/Oracle-proprietary compiler into ordinary Java JEE Stateless Session Beans.

These EJBs are in turn exposed as SOAP web services by the WLI platform itself, using the RPC/literal binding style. Consequently, all workflows are started via web service requests.

**Business Core**

A pure JEE application, which has usurped the “integration” role from the WLI. The Business Core is the now-central component responsible for combining datasets from multiple backend systems into a single Domain Model. All updates to data in backend systems must also pass through the Business Core, which enforce all relevant business rules.

The Business Core contacts a wide variety of backend data-suppliers – databases, web services, FTP servers, flat files etc. For all web service consumption, the Business Core relies on Axis 1.4 to produce client proxy classes.

The Business Core exposes its combined data services as web services, also using Axis 1.4. The binding style used is RPC/encoded.

**Order Engine Workflows**

The successor to Order Engine WLI workflows. All WLI workflows are actively being re-written from the aging WLI technology into the SOA-P supplied JBPM engine.

SOA-P workflows are typically started asynchronously via Java JMS queues, but “direct” control is also made possible via JBossWS-exposed Document/literal SOAP web services.
These services primarily communicate with the Business Core (RPC/encoded SOAP web services). In certain scenarios, some workflows on this platform rely on still un-migrated WLI workflows, and thus contact the WLI via the RPC/literal SOAP web services exposed there.

Web service communication originating from this platform rely on Axis 1.4 for web service proxy code generation.

**Channel Services**

The other half of SOA-P (the first being Order Engine Workflows) are the synchronous workflows “channel” systems. “Channels” cover web clients, apps for handsets, dial-in Interactive Voice Response (IVR) systems and others.

The purpose of these channel services is to expose a tailor-made, simpler-to-use subset of the internally available middleware services for specific clients.

All SOA-P channel workflows are exposed as SOAP web services, but the particulars differ from workflow to workflow:

As with the SOA-P Order Engine Workflows, some (but not all) of these channel services are re-written version of old WLI workflows. These migrated workflows are all “contract-first” flows, their interface 100% based on the WSDL originally produced by the old WLI platform. The reason behind this “legacy interface” preservation is that some of the clients are difficult, expensive or time-consuming to alter. It was deemed simpler to preserve the interface for the migrated services, and “just” re-write the inner workings of the workflows.

The migrated contract-first workflows all expose “WLI-style” RPC/literal WSDLs.

New channel services developed on the SOA-P are contract-last, with the WSDL auto-generated by the JBossWS toolkit supplied by the JBoss platform. Contract-last services use the Document/literal binding style.

**Web Channels**

A collection of distinct web applications, such as the [www.telenor.dk](http://www.telenor.dk) website, brand sites like [www.selvhenter.dk](http://www.selvhenter.dk), and intranet/extranet applications.

These web applications all rely on Channel Services for their backend communication, consuming a mix of “old WLI” RPC/literal and “new SOA-P” Document/literal SOAP services. These web channels do not directly interact with the Business Core or NewDelfiAPI-exposed RPC/encoded services.

**Opus GUI**

A Visual Basic (not .NET) GUI application communicating only with the Business Core. This is the primary interface for all maintenance operations performed by customer services.
2.4 Component & Connector viewpoint

2.4.1 Component & Connector viewpoint concerns

The SOAP protocol variants in use by the individual components are the focus of this Component & Connector viewpoint. Each component connects to another component via UML Ports and Interfaces. A Port in this context is the client framework, and an Interface the server framework. Each Port-to-Interface connector further details the SOAP binding style used. This is always determined by the Interface, but described on each connector for clarity.

The SOA-P systems “Channel Services” and “Order Engine Workflows” both expose services with two different binding styles (for contract-last and contract-first services), as described in the Allocation Viewpoint, Systems section.

![Component diagram]

Figure 2: Component diagram

A couple of Ports are highlighted on Figure 2, all for components residing on the SOA-P platform:
- **The Axis 1.4 Ports connecting from “Channel Services” and “Order Engine Workflows” (SOA-P systems) to Axis 1.4 services on the Business Core.** These are one of the central issues (described later) in this project – we have to use outdated Axis 1.4 technology on these (much newer) platforms, since the platforms themselves only provide the JAX-WS compliant JBossWS framework, which is unable to communicate with non-JAX-WS compliant RPC/encoded services.

- **The self-referencing JBossWS Ports from Channel Services to Channel Services and Order Engine Workflows to Order Engine Workflows.** This connection (indicated by grey connectors) is currently only used for integration testing, and is not part of the production codebase. JBossWS is the preferred client framework, but since it is incompatible with the RPC/encoded services exposed by Business Core, using JBossWS for “the real” production connectors is not an option.

### 2.4.2 Client and Server web service frameworks

The individual Ports and Interfaces (client and server frameworks) are:

**Axis 1.4**

The Apache Axis framework has two roles: client and server.

**Client**

Using the “wsdl2java” tool, Axis can produce Java proxy classes that encapsulate the SOAP communication / transformation from XML to Java objects. As the tool name implies, this generation of Java classes is based on the WSDL file. The output Java classes can be used to invoke SOAP services as if they were local methods, except for some initial server-endpoint configuration and handling of potential IOExceptions.

**Server**

The Axis server part is essentially a Java Servlet that accepts all POST requests to the URL specified in the servlets deployment description. The request URL determines which service (class) is invoked, and the posted XML SOAP document determines the operation and input. Operation/method name and input parameters are mapped from the received XML document according to the binding style used, which is always RPC/encoded for the Telenor services.

Which Java “service” classes to expose, and how, is determined by the Axis-proprietary Web Service Deployment Descriptor (WSDD) XML file. This file identifies classes to expose, which methods on the classes, which objects to allow, namespaces, etc. – and also the SOAP binding style to use.

The WSDL for each service is auto-generated by the Axis Servlet the first time an individual service is used or the WSDL explicitly requested.

**JBossWS**

JBossWS is essentially a wrapper around Apache CXF [CXF]. The JBossWS layer on top of Apache CXF provides close integration with the JBoss server as well as JBoss-specific configuration and management of CXF.
CXF (and thus, JBossWS) implements the JAX-WS specification, allowing web services to be built using Java Annotations. Such services can be built very quickly using @WebService and @WebMethod annotations on any class/method deployed in a WAR or EAR file.

Similar to the Axis clients, JBossWS is also capable of producing client proxy classes used to call the exposed web services. Unlike Axis, JBossWS “only” supports SOAP services conforming to the JAX-WS specification – which does not include the RPC/encoded binding style.

WLS 8.1 WS
The BEA-developed JAX-RPC (precursor to JAX-WS) implementation used by WLI workflow EJBs. The WSDL for such web services is generated deployment time.

The early version (from WLS 6.1) of this implementation was very limited in configuration options, and was discarded in favour of Axis. The only place WLS 8.1 WS is in use is in the situations where it is forced, ie. WLI workflow exposure.

Client proxy generation capabilities (to clients outside of WLI workflows) exist, but are not used anywhere in Telenor.

WSE 3.0
Web Service Enhancements for .NET 3.0 is a Visual Studio (2005+) add-on that allows easy consumption of web services. This is a client-only framework, allowing aging .NET 2.0+ platforms to make SOAP web service calls. Generated proxy classes are very similar to the Java classes produced by Axis or JBossWS; classes named after the services the WSDL represents, with local operations that handle all the SOAP/XML protocol tasks transparently for the user.

2.5 WS-* usage
“WS-*” refers to a vaguely defined group of web service specifications including SOAP and WSDL. There are many individual WS-* specifications in varying degrees of maturity, maintained by different organizations. These specifications (typically) complement each other, and when combined they provide a very large set of features. A common point of critique of the WS-* suite of specifications is that they are overly elaborate, providing detailed specifications for features that are near-useless or in the real world.

Telenor use of the WS-* “death star” is limited to SOAP enveloping, WSDL service interface specifications and WS-I Basic Security Profile for authentication.

This is a very basic use of the complex WS-* suite. If our use is limited to sending validate-able XML documents over the HTTP protocol, the same could be accomplished by just using XSD Schemas for interface specifications, and regular HTTP Basic Authentication for identification of users. This limited use of the complex WS-* specifications could indicate that WS-*/SOAP is not a good match for our needs. This will be explored in further detail in the SOAP/REST prototypes.
2.6 Representative minimal environment

The middleware production environment is built using large and costly server platforms. For prototyping purposes, it will not be necessary to use a setup consisting of the same platforms – instead, a small-scale environment using similar or identical technologies can be used for the individual prototypes.

This environment must support the different protocol-to-protocol variants – or at least the interactions with known issues, or interactions that will be impacted by architecture upgrades.

It is obvious that a complete REST re-write will impact every single interaction between components, so the “impacted” statement should be understood in a SOAP framework-upgrade context.

A setup with a single JEE application server, and multiple stand-alone desktop applications will be easy to configure, and support all required interaction types. For SOAP, this setup is used:

And for REST, the same setup will be used, only with different contents:
2.7 Current Telenor web service issues

The Telenor middleware has grown considerably, both size and complexity, over the last 10 years. Solutions that worked well in smaller scale and simpler circumstances have become difficult and time-consuming to maintain, or scale poorly.

Here, the key issues that must be addressed are described. For each of these issues I describe possible anticipated solutions.

Experimental prototyping will later be used to test if the solution suggestions are valid, or if other/better solutions than those expected up-front can be found.

2.7.1 Axis framework

The middleware infrastructure as described in the previous section rely heavily on the Apache Axis 1.4 [AXIS1] framework, to enable both client and server SOAP communication. Axis gained widespread use with version 1.1 from 2003, but was eventually discontinued in 2006 with version 1.4. Axis supports the WS-I Basic Profile 1.0, and various SOAP binding styles.

At Telenor, we use Apache Axis 1.4 (upgraded from the initial use of 1.1) for exposing web services, as well as for service consumption. Web service consumption in this context means that we use the Axis framework to produce native Java client proxy classes based on the available WSDL files. The systems relying Axis for web service exposure are mostly applications on older platforms (WLS) with no native SOAP web service exposure mechanisms.

During our years with Axis we have identified several problems – some relate to server-use, and some to client-use.

Axis servers

Our main use of Axis for web service exposure is through the “RPC/encoded” WSDL binding style. This binding was not included in the JAX-WS API [WIKIWS], and is as such not supported by most modern web service frameworks. This essentially means that the services exposed by Axis are incompatible with most SOAP web service client frameworks available today, leaving us only one possible choice for client framework: Axis itself.

An attempt to use a JAX-WS tool to generate local proxies for an RPC/encoded WSDL will produce errors such as

```
[ERROR] rpc/encoded wsdls are not supported in JAXWS 2.0.
```

or

```
Rpc/encoded wsdls are not supported with CXF
```

… depending on JAX-WS implementation used. As an example of the latter, I’ve tried producing SOAP clients using Apache CXF 2.7.3 against a (Axis1) demo-service deployed on my local JBoss instance, as laid out in the Representative minimal environment section. Using Maven, and the CXF-supplied plugin “cxf-codegen-plugin”, as shown below:

---

1 Unless explicitly noted “Axis” refers to Apache Axis 1.x [AXIS1], not Apache Axis 2.x [AXIS2].
Use cxf-codegen-plugin to produce jax-ws clients -->

<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>2.7.3</version>
  <executions>
    <execution>
      <id>axis-rpc-encoded-client</id>
      <phase>generate-sources</phase>
      <configuration>
        <wsdlOptions>
          <wsdl>
            http://localhost:8080/services-axis1/services/HelloWorldRPCEncoded?wsdl
          </wsdl>
          <extraargs>
            <extraarg>-p</extraarg>
            dk.au.cs.wsinfra.cxfclient.axioservice.rpencoded
          </extraargs>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>

(Build-file snippet is from the “client-cxf-to-axis1-rpc-encoded” project. Prototyping projects are described in full detail later.)

Above, http://localhost:8080/services-axis1/services/HelloWorldRPCEncoded?wsdl targets an Axis-based RPC/encoded “hello world” service developed for prototyping purposes. Running a Maven “mvn install” build produces this output:


**Axis clients**

Axis is used as the client-framework of choice, partly because it is the only known framework capable of consuming our existing RPC/encoded services, but also because it was the dominant – if not only – mature Java web service framework available when SOAP web services were introduced at Telenor. Its age and years without maintenance does show, however: the Axis clients performs poorly when handling large XML documents, and are not completely thread-safe, leading to unpredictable errors when under high load.

Axis-client performance was tested on large XML documents (~1MB) and deserialization times around 30 seconds were not uncommon. Deserialization in this context refers to parsing the XML SOAP document, and creating Plain Old Java Object (POJO) instances populated with the XML data.
For services with end-users browsing a website as the recipients of such data, a 30 second wait is clearly unacceptable. As a consequence of this poor performance (coupled with the *Unnecessary transformations* issue), data-intensive services which must respond quickly\(^2\) have been switched back from SOAP web services to RMI.

When under heavy load, Axis clients have also been known to produce Java NullPointerExceptions in the internal library classes. Since Axis is an open-source project, the source was debugged and found to be thread-unsafe. Heavy multithreaded use of Axis client libraries in the same JVM (such as on the 64-core servers hosting our WLS applications) would in certain race-conditions cause failures. Although unlikely and uncommon, they do occur. Telenor-local patches for some of these threading issues have been created, fixing some – but not all – of these problems.

**Possible solutions**

*Axis client framework replacement*

Based on past research, there are no Java SOAP Java web service client tools capable of generating Java client proxies for Axis-exposed RPC/encoded services, except Axis. As part of a large Telenor project in 2009, we searched for possible Axis replacements, but came up empty handed. An offshot of this investigation can still be seen at [stackoverflow]\(^3\), posted by an Oracle consultant while working on the before-mentioned project. This was back in 2009 though, and it is possible new frameworks, or just updated version of existing frameworks can handle the outdated Axis services now.

So, framework-alternatives should be tested against Axis RPC/encoded services to see if any of these can replace Axis as client-library.

Generating Java proxy classes based on a WSDL is not magic. It would be possible to produce homegrown tools to create such java bindings against RPC/encoded services, but even if the effort required to do so is acceptable, additional customization it is not the direction we wish for the middleware area. Any client framework should rely on open, established standards as much as possible, to make future upgrades less painful than the current Axis-upgrade is likely to be.

*New SOAP service layers*

It should be possible to expose a secondary set of SOAP web services on alongside the current Axis services. Axis is just a Servlet listening on a given URL pattern (/services/*), and redirecting posted documents to the appropriate Java class. The Axis-exposed WSDLs for these services/classes are generated by the Axis framework itself, based on an external configuration file (WSDD). The target “actual” Java code has no knowledge about Axis (or web services in general) at all, and would not require any change. Therefore, there is nothing preventing an additional web service framework from accepting SOAP documents posted to another URL (f.ex. /new-services) and invoking the same backend methods.

\(^2\) “quickly” as in “fast enough for a webpage”. No specific SLA exist for such services in general, but more than 2-3 seconds is generally considered too slow.

\(^3\) StackOverflow may not exactly be an appropriate resource for scientific research, but the answer to a good many everyday-problems can be found there nonetheless ☺️.
This could be achieved with any framework supported by the host platform (i.e., a Java 4 – compatible framework for WLS applications, or Java 6 for JBoss EAP/SOA-P platforms).

This would of course mean that two distinct WSDLs, and two distinct service-endpoint URLs, would exist for every single service. And more importantly, that existing clients actively have to update their already-generated client proxies against the new set of WSDLs. This could be an on-going transition where touched/modified clients shift towards the new service layer.

REST (or other) additional layer

The additional web service layer mentioned above need not be a SOAP layer. REST could be used as well, provided a reasonable mapping of REST path/parameters can be made against the existing SOAP (SOA) services.

All existing backend logic is built following the SOA pattern with services such as getSubscriberInfo(subscriberId) and addService(subscriberId, serviceId). If and how such existing SOA services can be mapped to REST will be explored in the REST literary study and prototypes.

2.7.2 Interface lockdown

The architectural description did not cover system functionality, but a bit of detail about the functionality is required to describe this issue. All individual systems follow the same SOA paradigm. All define (at least) one system-specific data Domain Model [WIKIDM], typically a basic POJO class hierarchy. These POJOs are used for input/output across all web services.

With most services being contract-last (WSDL generated based on the method signatures), this is an easy way to quickly establish a complex interface – a simple Java method signature becomes a WSDL complete with all input/output types pre-defined by the internal Domain Model.

As the number of services using the same interface objects grow, more and more individual WSDLs now describe the same Java objects. These WSDLs are in turn used by client systems to generate their own language-native proxy classes. If the system requirements or functionality change in a way that affects this domain model even the slightest, it is potentially a problem.

As a concrete (though simplified) example, consider an Account object. This object describes a billing account entity, complete with name/address, services to bill, amounts etc. Each such Account is further capable of containing a single InstallmentPlan object, which describes a set of payment-terms / periods:

![Account class hierarchy example](image)

Figure 5: Account class hierarchy example

Originally developed as a “one per account” type object, business requirements changed, and support for “multiple Installment Plans per account” was required. Simple:
Figure 6: Updated Account class hierarchy example

… except not simple at all. This change breaks the interface of any WSDL currently describing the Account object – forcing all (or most) client systems using the Account object to be updated, even if they do not use the InstallmentPlan property at all.

It is not even the case that interface changes always requires updating in the above scenario. That depends on the concrete change, on how the client system use the Account object, and the combination of client and server frameworks. If, for instance, a new property was added, instead of altering an existing property:

Figure 7: New Property on the Account object

This new property will turn up in output-documents from any service returning Account objects. Since the WSDL is generated by the server (“contract-last”), all WSDLs will correctly describe this new property as soon as the services using it are deployed. Using a non-updated client (i.e., a client based on a WSDL without “newProperty” in it) will result in the client receiving a SOAP XML document containing a new, unknown property. Some client frameworks (such as Axis) will simply ignore such unknown content. Other client frameworks (such as WebLogic-WS) will fail due to unknown contents in the received document.

These issues mean that any interface change has to be considered carefully, and the default response to any change affecting any existing I/O objects ends up being “update all client systems!” With multiple client systems, some external to Telenor (with a 3-month notice for any system change), that quickly becomes something you dread. To avoid that, a typical workaround is to isolate the required changes into new objects, and add new methods with extended method-signatures. Hardly a desirable evolution of an already complex system stack, and not even possible for the single-object-turned-array example in Figure 1 (Figure 6 above).

In fairness, this issue is not specific to SOAP, Axis nor web services in general. Changes to un-versioned method/RPC interfaces is likely to require client-side updates, regardless of technology used.

**Possible solutions**

*Versioned interfaces*

The most direct way to handle this is issue to preserve the existing interfaces, and only add new interfaces as new versions. A primitive implementation of this is simply to incorporate version into the service name / URI such as

http://localhost:8080/demoservices/GetSubscriberInfo_1_0,
http://localhost:8080/demoservices/GetSubscriberInfo_1_1
Instead of simply

http://localhost:8080/demoservices/services/GetSubscriberInfo

This allows simultaneous deployment of multiple versions of the same service. Clients can then change to newer editions at their own pace. Changing the interface of an individual service alone might not be possible though (due to shared I/O objects used across multiple services) – leading to the next approach:

**Versioned applications**

With the case of JAX-WS web service applications, they will typically be bundled in WAR (Web Archive) files for deployment. Under such circumstances it might be preferable to version the entire application instead:

http://localhost:8080/demoservices_1_0/GetSubscriberInfo

That would be a more elegant handling, at least in the case of Telenor-applications, where the interface objects are often used across multiple services, and updates to objects are far more frequent than updates to method signatures. An update to such an object would impact all services using this object, not just the single intended service. So, you’d have to release new versions of all services, or version the objects as well.

Thus, having the full application versioned instead becomes easier the larger the application and I/O object hierarchy becomes. This can prove quite expensive though, since each application-deployment (version) will require additional server resources.

**Generic interfaces**

The WSDL for any SOAP service acts as the contract for said service. With WSDLs auto-generated based on Java method signatures (JAX-WS contract-last approach) the WSDL is quite specific on field names and data types.

It would be possible to define a SOAP service simply producing (and receiving) an XML document. These documents could then contain both data and definition – such as (versioned) XSD Schemas describing the current content. Doing that would eliminate much of the benefit from using SOAP services in the first place, especially considering the very limited WS-* use at Telenor.

Such unrestricted services would obviously also require major interface updates to all client systems currently relying on the auto-generated WSDL interfaces. So in essence, it would be replacing one interface issue with a vastly larger one.

### 2.7.3 Unnecessary transformations

A typical use-case for the middleware systems is to fetch information, to display on the www.telenor.dk website. A flow of calls from frontend to backend, and back, would look like this:

The web user initiates the request by browsing to a URL. The IIS web service activates the C# code-behind page, which is responsible for generating the HTML content. Part of what this page does is to invoke middleware web services for fetching data to display. This is done via a Channel Service, which in turn calls a Business Core service, which then again calls a
NewDelfiAPI service. The output from NewDelfiAPI is ultimately received in the web page and presented to the user.

A lot of Channel Services actually rely on multiple Business Core services to collect data. Similarly every Business Core service will often call multiple NewDelfiAPI services. So a single Channel Service operation could easily end up as 5-10 NewDelfiAPI service calls. For simplicity, only a 1-to-1 path is shown here:

![Figure 8: Web page fetch-data sequence diagram](image)

Above, systems are colored light-green, while the web service frameworks used for SOAP communication are colored light-red.

For every Frame transition (f.ex. IIS ↔ JBoss Channel Services) there will be an initiating and receiving web service framework. For connector B it is the MS WSE framework posting a SOAP XML document to JBossWS over HTTP. Each such Frame transition involves:

a) Native-language object to XML serialization in the client system.

b) XML to native-language object deserialization in the server system.

… where a) takes place in the client system/framework, and b) in the server system/framework.

So, from connectors A to I there are three de/serialization pairs, before the request object reaches the bottom-layer backend system, NewDelfiAPI. During this voyage, the data format/content will change to match the Domain Model representation in each system, but the meaning/purpose is unchanged.

Once in NewDelfiAPI, this system fetches data from the database (not shown), and the whole ordeal begins again in reverse order. Again, three de/serialization pairs, this time of the full data-payload fetched from the backend system.

In extreme cases, such as retrieving full call-detail listing for a given GSM subscriber, the data-payload returned from NewDelfiAPI can be several MB of data (in XML form). These rather large documents then have to undergo the same three layers of de/serialization as the (typically very small) input/request object did. Even layers that do nothing except pass-through the data still have to perform the XML-to-native de/serialization steps.
For such large documents, Axis performs poorly, and each single serialization or de-serialization could take 30 seconds or more, adding up to a several-minute response time. But even if the XML de/serialization processes were done using high-performance SAX parsers it would still be a very significant overhead.

It is also worth noting that these *unnecessary* transformations are not by definition *bad* thing. The setup with multiple completely independent Domain Models (one per system, modelling the world as perceived by the respective system), and repeated transformations from one model to another, ensures loose coupling between systems. This allows the “bottom” systems such as NewDelfiAPI to maintain a datamodel matching the underlying backend system with which it is interacting. Similarly, it allows the middle-tier Business Core system to model data gathered from many distinct backend systems (NewDelfiAPI being one of them) into a single coherent structure. Loose coupling and performance is a typical example of Quality Attributes that are at odds with one another – designing for one hurts the other. At Telenor we have designed for loose coupling rather than performance, leaving us with hacks and workarounds in situations where performance is too poor to be acceptable.

**Possible solutions**

Within the existing architecture server and client constraints of “Java-frameworks exposing Java methods as SOAP services” and “Java-frameworks producing Java-proxies based on WSDLs” there is not much to do about this issue.

Any service must serialize its Java object hierarchy into XML for SOAP transport. And any client must deserialize the received XML into Java objects.

**Transport model**

If the *Generic interface* approach to the *Interface lockdown* issue was followed, a raw XML document could be passed as a String value through the intermediary systems (Business Core and Channel Service, in Figure 8) without further processing.

That would open up for using a shared transport-level data-model, and let it be optional for the individual services if they need to map data from the transport-level model into the application specific internal (Java) domain model or not.

The combination of the “Generic interface” approach and such a transport-level model is a candidate for REST prototyping.

**2.7.4 Run-time interface discovery**

A less-critical issue than the previous three, but a daily problem nonetheless, is the availability of WSDLs.

For a client system to generate proxies it needs the WSDL file describing the services. Changes to the services means changes to the WSDL file, so in order to ensure interface compatibility, the client system should preferably re-generate the local proxies against current WSDLs every time the server system is rebuilt. This requires the client system to be rebuilt also.

Such a build-process would ensure that (most) interface changes become visible to clients, most likely in the form of un-compilable client code referencing properties in the WSDL that no longer exist, or have changed datatypes. Such continuous re-generation of proxies also ensures...
that added properties (which are “hidden” since they are not referenced) do not become a problem during XML de/serialization. It does however require all systems to be built and re-built, in server → client order.

That is cumbersome, but doable. It becomes even more cumbersome when some systems cannot build-time produce their WSDLs, but require the application to be deployed to a server before WSDLs can be generated. That is the case for Axis services, as exposed by the WLS-applications NewDelfiAPI and Business Core. The WSDLs for all services in these two systems are generated by the Axis Servlet when the first client requests a WSDL from the running system. Build-time generation of WSDLs is actually possible using Axis, but requires the build-script to supply a lot of configuration-properties defining how the services are exposed. This configuration does not necessarily match the actual WSDD configuration used by the servlet, leading to pre-generated WSDLs that are not identical with the Servlets run-time generated WSDLs.

Possible solutions

Build time WSDL generation
The outdated Axis framework does not support generating services prior to deployment (well), but several JAX-WS implementations can do this. So, for any JAX-WS service it will be possible to build-time generate the WSDL. Given the JAX-WS standard, it should furthermore not matter which JAX-WS implementation is used to produce the WSDLs – they should all produce identical WSDLs for the same services.

Manual WSDL, contract-first
Deploying Java code to a server and telling it to “turn it into web services” is a fast and easy way to create web services. But this process only leaves as much control over the final WSDL as the tooling allows – including when the WSDL can be produced.

Some (most) Java SOAP frameworks allow you to supply the WSDL yourself and simply point the framework in the direction of Java classes matching the WSDL specification. This approach is often used when detailed control of the WSDL is required, or – as the case with Telenor WLS to JBoss migrations – the WSDL is pre-existing.

For JAX-WS, the ability to use a pre-built WSDL is part of the specification.
3 SOAP framework prototyping

3.1 Purpose and method

For prototyping to provide any value it is important to be clear about the purpose of the prototype(s). To this end, multiple prototypes variants are described in the CS literature. The article “Architectural Prototyping: An Approach for Grounding Architectural Design and Learning” [PROTO] classifies prototypes as either Explorative or Experimental (quote):

- Explorer prototypes used to clarify requirements to the architecture together with its stakeholders, to explore aspects of the target system, and used to discuss alternative architectural solutions.
- Experimental prototypes used to gauge the adequacy of a proposed architecture, or details hereof, before investing in a large-scale implementation of it. Experimental prototypes are typically used to measure software architecture qualities; e.g., to make quantitative measurement of performance or portability.”

All the SOAP problems in the current architecture have known solution-candidates. Uncertainties in these solutions primarily revolve around implementation/API specification details. Thus, the following prototypes will be experimental, with the purpose of verifying that the suggested approaches work as intended.

A secondary purpose of the prototypes is to demonstrate some of the known issues described in the Current Telenor web service issues section.

3.2 Problems to prototype

The Current Telenor web service issues section describes these issues:

1. Axis1 RPC/encoded services
2. Versioned interfaces
3. Unnecessary transformations
4. Run-time interface discovery

For the “Axis1 RPC/encoded services” problem, I will use prototypes to demonstrate the problem, and to try the “Additional layer” approach: exposing multiple web service layers on top of the same service.

The 4th issue, “Run-time interface discovery”, has a simple solution, provided the frameworks support it: build-time WSDL generation. This is tested for the Axis framework replacement candidate(s).

The issues described in “Versioned interfaces” are inherent in any Client-Server application, and not SOAP specific. So no SOAP prototypes will be made to cover this issue.
The “Transport model” solution to “Unnessecary transformations” is well aligned with the REST idea of resources and representations (having a shared transport-representation, understandable by the various applications), so that issue will not be SOAP prototyped either, but left for REST prototypes.

Therefore, the SOAP prototypes are scoped to experimental prototyping of 1\textsuperscript{st} and 4\textsuperscript{th} problems above only.

3.3 Prototype projects

3.3.1 Apache Axis 1 framework alternatives
The frameworks I have chosen for evaluation as Axis replacements are:

- Apache Axis1
- Apache Axis2
- Apache CXF

CXF is intended to cover all “pure” JAX-WS implementations, since there should be little difference between various implementations, apart from tooling and performance. Since CXF is the internal JAX-WS implementation used by JBossWS, it is already in-house and is as such the first choice among Axis-alternatives.

Axis2 is, like CXF, a JAX-WS implementation, but may (due to its lineage) be able to consume the old RPC/Encoded Axis1 services, which makes it very interesting indeed. CXF and Axis2 both implement the JAX-WS standard and are interchangeable on paper. CXF is implemented using primarily other standard frameworks (such as JAXB for Java to XML mapping), where Axis2 relies on homegrown tools to accomplish the same tasks. This gives Axis2 and edge performance-wise, in some situations at least, but also makes it more likely to get “left out” on future benefits from standard improvements.

Consumption of RPC/encoded services
To test if Axis1 services can be consumed by either CXF or Axis2, a setup with two server projects and four client projects has been created.

The two services-\textsuperscript{cxf} projects are self-contained WAR (Web Archive) files which can be drop-in deployed on any JEE6 server. The four client-\textsuperscript{cxf} projects attempt to generate local proxies against a specified type of services (expected to be available on the localhost:8080 JBoss AS7 server, described in the Representative minimal environment section). More detail and purpose-description of each individual project follows below:

- \texttt{services-cxf-\textsuperscript{jaxws}}: A collection of JAX-WS SOAP web services. The purpose of this project is to expose a few basic JAX-WS services as reference point for functional/correct services. All client frameworks must be able to consume such services.
- **services-axis1**: A single RPC/encoded service. This demonstrate how existing services from the NewDelfiAPI and Business Core systems expose services. It is expected that only Axis1 is capable of consuming this service.

- **client-cxf-to-cxf-jaxws**: A set of clients connecting to the CXF-exposed JAX-WS services. The purpose is a reference-point for client systems – they should be capable of consuming JAX-WS exposed services.

- **client-cxf-to-axis1-rpc-encoded**: A CXF client project attempting to generate proxies against the RPC/encoded Axis service. This is expected not to be possible. The purpose of this prototype project is to verify if that assumption is (still) true.

- **client-axis2-to-axis1-rpc-encoded**: An Axis2 client project attempting to generate proxies against the RPC/encoded Axis(1) service. Since Axis2 is the successor to Axis1, and is known to favour homegrown code over existing standards, it is possible that Axis2 is capable of consuming Axis1 services. Being JAX-WS compliant does not necessarily exclude Axis2 from going “above and beyond” the JAX-WS API specification.

- **client-axis2-to-cxf-jaxws**: An Axis2 client project generating proxies against the CXF JAX-WS services, which should be possible since both frameworks implement JAX-WS. This project serves to demonstrate interoperability between various JAX-WS implementations.

The full source of these projects is available in the Appendix section – only excerpts are shown here.

Exposing contract-last JAX-WS services is very simple, as shown in the **services-cxf-jaxws** project. Any Java class with the `@WebService` annotation will be exposed as a SOAP service, with the `@WebMethod` marked methods as available operations. The binding style to use is optionally set in the `@SOAPBinding` annotation.

The **services-cxf-jaxws** project contains four HelloWorld services, with the binding styles Document/literal, Document/encoded, RPC/literal and RPC/encoded. Each service is a class as shown below, with just the `@SOAPBinding` changing from class to class:
package dk.au.cs.wsinfra.cxf.services;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
import javax.jws.soap.SOAPBinding.Use;

@WebService
@SOAPBinding(style = Style.DOCUMENT, use = Use.LITERAL)
public class HelloWorldDocumentLiteral {
    @WebMethod
    public String helloWorld(String name) {
        return String.format("Hi there %s!", name);
    }
}

(As a curiosity: when instructed to expose an RPC/encoded service (not possible in JAX-WS), the CXF framework actually exposed an RPC/literal service instead, with no warnings or error messages to indicate this.)

When deployed to the JEE server, the @WebService annotation uses the available JAX-WS implementation (JBossWS-CXF) to generate WSDL and handle the request/response chain.

From the client project client-cxf-to-jaxws-cxf, the service is consumed by using the CXF-supplied Maven cxf-codegen-plugin, which generates local proxy-sources from a supplied WSDL file:

```xml
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>2.7.3</version>
  <executions>
    <execution>
      <id>document-literal-client</id>
      <phase>generate-sources</phase>
      <configuration>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>
            <extraargs>
              <extraarg>-p</extraarg>
              <extraarg>dk.au.cs.wsinfra.cxf.client.document.literal</extraarg>
            </extraargs>
          </wsdlOption>
        </wsdlOptions>
        <goals>
          <goal>wsdl2java</goal>
        </goals>
      </configuration>
    </execution>
  </executions>
</plugin>
```
This produces local Java classes which require a minimal initial configuration before being used as if the services were methods on local objects (excerpt from a unit test):

```java
private HelloWorldDocumentLiteral service;

@Before
public void before() {
    HelloWorldDocumentLiteralService serviceLocator = new HelloWorldDocumentLiteralService();
    service = serviceLocator.getHelloWorldDocumentLiteralPort();
}

@Test
public void testHelloWorld() {
    String response = service.helloWorld("Morten");
    assertThat(response).isEqualTo("Hi there Morten!");
}
```

This all works well, as it should with CXF-server + CXF-client.

For verification purposes, an Axis2 client is generated against the same CXF-services. Surprisingly, I was unable to get the generated Axis2 clients to function against the CXF JAX-WS service. The clients were generated without a hitch (although the produced client-code was extremely messy!), but execution kept producing ClassNotFoundExceptions for some (supposedly) Axis2-generated type definition classes. Example partial stacktrace:

```
Caused by: java.lang.RuntimeException: Cannot load SchemaTypeSystem. Unable to load class with name
schemaorg_apache_xmlbeans.system.s880E2883A158DEEBBA55943EEF7192DA.TypeSystemHolder. Make sure the generated binary files are on the classpath.
```

These errors look more related to my usage of Axis2 rather than actual JAX-WS compliance issues on the part of Axis2. Nonetheless, “hard to use” is not a positive label to earn during framework evaluation. My inability to get Axis2 clients working for standard JAX-WS services also means that Axis2-prototyping against Axis1 RPC/encoded services is not performed.

CXF consumption of Axis1 RPC/encoded services was not expected to work, and did not work. The client-cxf-to-axis1-rpc-encoded project fails during build with message:

```
```

These tests confirm the initial assumptions: that JAX-WS tools are unable to consume RPC/encoded services, and that CXF is a better choice than Axis2 as replacement framework.

**Multiple SOAP service layers for a single service**

One approach for replacing Axis1 as server framework is to expose multiple web services for the same resource at the same time; one legacy Axis1, and one modern CXF JAX-WS.
This is demonstrated in project services-multi, which is essentially a combination of the services-axis1 and services-cxf-jaxws.

The project exposes a single “helloWorld” service on these 2 URLs:
/services-multi/HelloService
/services-multi/legacy/HelloService

The /legacy/ URL targets the Axis 1 service. Both operation endpoints will, when invoked, execute the same Java class. Such multi-exposure of the same services is simply a matter of configuration.

**Build-time generation of WSDL**

Apache CXF supplies the “cxf-java2ws-plugin” Maven plugin, which is capable of build-time producing WSDL files. This is also demonstrated in the “services-multi” project pom.xml file:

```xml
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-java2ws-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>generate-wsdl</id>
      <phase>process-classes</phase>
      <goals>
        <goal>java2ws</goal>
      </goals>
      <configuration>
        <outputFile>target/multiservice.wsdl</outputFile>
        <className>dk.au.cs.wsinfra.multi.services.MultiService</className>
        <verbose>true</verbose>
        <genWsdl>true</genWsdl>
      </configuration>
    </execution>
  </executions>
</plugin>
```

This enables easy build-time WSDL generation, with no external service configuration required. All required configuration parameters for this generation (such as SOAP Binding style) is supplied via code-annotations, ensuring that the build-time generated WSDL will be identical to the run-time generated WSDL.

### 3.4 SOAP prototype conclusions

In general, the SOAP prototypes confirmed my expectations:

- Axis1 RPC/encoded services cannot be consumed by other frameworks than Axis1 (at least not CXF or Axis2).
- It is possible to expose multiple SOAP web service layers for the same resource.
- The CXF framework allows build-time generation of WSDL files.

The Axis2 framework proved much more troublesome than expected, especially given my years of experience with the Axis1 framework. This difficulty, plus the “favor standards over
customization” approach of CXF, and the fact that JBoss uses CXF internally, makes the choice between CXF or Axis2 fairly simple: CXF.

In conclusion, the best SOAP upgrade path for Telenor, is to use Apache CFX to expose a secondary set of web services, using a standard JAX-WS binding style, such as Document/literal.
4 REST study and prototyping

4.1 REST literary study

4.1.1 Background
In recent years, I have often heard REST mentioned as an alternative to SOAP / WS-* web services. Details about what is actually meant with “REST” are often scarce, and limited to comparisons along the lines of “like SOAP, only resource-oriented” and “free-for-all homemade services responding to HTTP GET and PUT requests”. More often than not, REST is perceived as an alternative, underspecified, kind of RPC.

To consider if and how REST can be utilized in the Telenor middleware, I have performed a literary study of REST, to gain an understanding of what REST is – and is not.

4.1.2 Literature
The literary study is based primarily on Roy Fielding’s “Architectural Styles and the Design of Network-based Software Architectures” [RFREST] dissertation, also known as the “REST dissertation”.

Fielding’s dissertation defines REST in general. For information on how REST can be applied to concrete programming problems, particularly the Telenor SOA middleware, I read “RESTful Java with JAX-RS” [JAXRSBOOK] (Part I), which covers the “Java API for RESTful Web Services” JAX-RS 1.0 API specification. And as a follow-up note, the “JAX-RS 2.0: What’s New in JSR 339?” [JAXRS2SLIDE] presentation from a 2012 JavaOne conference, to see what had changed since Bill Burke wrote the “Java API for RESTful Web Services” book.

4.1.3 REST dissertation summary
REST – short for Representational State Transfer – is an architectural style, defined by the set of constraints limiting how hypermedia based network-applications should operate. Specifically, REST is the architectural style that describes how web pages and resources are made available to clients.

These REST constraints were not fully described during development of the basic web standards (HTTP, HTML and URI), but were the underlying guideline used by Internet Engineering Taskforce (IETF) nonetheless. This relied on a mutual understanding of how the web should evolve. Such an understanding may have been present among the architects internally at IETF, but as external (and commercial) entities began contributing to the web as well, not all extensions conformed to these undescribed constraints.

To guide the future development of the web, and assess which extensions were to be elevated to standards, Roy Fielding (IETF member and central contributor to several web protocol specifications) distilled the architectural constraints already in place into the REST dissertation[RFREST]. This dissertation describes various network-centric architectural styles (Client-Server, Peer-to-Peer etc.), and extracts the benefits and drawbacks (“properties”) of each styles.
The REST style itself is defined by the set of constraints that provide the desired properties, or rather by the combination of the pre-described styles that convey these properties. Thus, the REST architectural style ends up with this style-derivation diagram (from [RFREST]):

![REST Style Derivation Diagram](image)

**Figure 9: REST style derivation diagram**

The diagram is read from top to bottom, with the top row being basic architectural styles such as Client-Server (CS). The middle and bottom row are derivations of these basic styles, formed by adding constraints. F.ex. Client-Server (CS) gains the Stateless constraint (Client-Stateless-Server, CSS) and the Cacheability constraint, turning it into Client-Cache-Stateless-Server (C$SS$).

The full list of acronyms used above is:

- **RR:** Replicated Repository.
- **CS:** Client-Server
- **LS:** Layered-System
- **VM:** Virtual-Machine
- **U:** Uniformity
- **$:** Cacheability
- **CSS:** Client-Stateless-Server
- **LCS:** Layered-Client-Server
- **COD:** Code-on-Demand
- **C$SS$:** Client-Cache-Stateless-Server
- **L$CSS$:** Layered-Client-Cache-Stateless-Server
- **L$CODC$SS$:** Layered-Code-on-Demand-Client-Cache-Stateless-Server

Thus, ultimately, REST is defined as the Layered-Code-on-Demand-Client-Cache-Stateless-Server architectural style, with the added constraint of a Uniform interface.
REST styles and constraints applied to SOAP

Since REST is an architectural style and SOAP is protocol specification, they are not directly comparable. However, it is possible to evaluate if each of the base styles that define REST as a derived architectural style apply to SOAP as well. After all, if SOAP is a derivation of the same styles, it would mean that SOAP is as much a “REST implementation” as f.ex. HTTP. Such comparisons between SOAP and HTTP are likely to get muddled, since SOAP more often than not rely on HTTP as the transport-layer. By reducing HTTP to a simple transport layer, however, SOAP is likely to reduce the style-inherited “REST benefits” of the HTTP protocol as well, so I believe the comparison is valid.

In the following section I will describe the individual styles from the REST style derivation diagram, and consider how well SOAP conforms to each base style. When applicable, the style-conformance of SOAP and HTTP (used as a representative REST implementation, not as a SOAP transport layer) will be directly compared.

This should result in a better understanding of how SOAP and REST (or RESTful services) differ.

Replicated Repository (RR), Cacheability ($)

Replicated Repository is the ability to replicate parts of the origin data to a location closer to the client, or locally at the client itself. This is done to improve user-perceived performance of network applications, since network bandwidth is often a scarce resource. It allows for other benefits (reduced origin server load, scalability) as well. Replicated Repository is often implemented as client-side caching, or caching in intermediary layers. In HTTP, the Cache-Control header allows both clients and servers (and intermediaries) to act according to the caching-settings of the resource being manipulated. This allows for flexible caching strategies per individual resource.

SOAP does not disallow caching per se, but the protocol provides no easy mechanisms for utilizing caching either. When SOAP is discussed, it is most often “SOAP over HTTP” since SOAP does not rely on any specific transport layer. HTTP is the default choice because it fits the request-response call patterns of SOAP well, and is universally available. Being transport-layer agnostic may sound like a good thing, but actually prevents SOAP from making use of the built-in HTTP Cache-Control headers. SOAP documents will typically be sent as HTTP POST documents containing the SOAP XML document as the HTTP body. This message body then translates to a concrete method call on the SOAP server. HTTP GET is rarely used, and pure HTTP proxy implementations will not be able to distinguish a HTTP POST’ed SOAP document targeting a cacheable read operation from a non-cacheable write operation.

The Replicated Repository-style is not inherent in the SOAP protocol, as it is in the HTTP protocol, leaving application developers to implement their own in-application caching strategies. So, SOAP is at a disadvantage compared to HTTP/REST.

Client-Server (CS), Client-Stateless-Server (CSS)

Client-Server is a very common style; a central server exposes resources or operations, and multiple clients initiate contact to these server-resources. Needs no further introduction. In HTTP, the client is typically a web browser, GET’ing resources from the web server, or POST’ing input form content to the server.
The same call pattern is used by SOAP, with the only difference being the HTTP methods utilized and the HTTP body contents.

Where SOAP and HTTP differ is in the Stateless style. The HTTP method set only allows clients to PUT new resources on the server, or POST updates to existing resources. In both cases, the transmitted operation must not rely on any existing server state. It would of course be possible to implement an HTTP-driven server which makes use of server state (“shopping baskets” being a notable example of this happening in real life), but such behaviour is not aligned with the HTTP intentions, or the REST principles. So, server statefulness could be said to be “possible, but frowned upon” for HTTP. Actual state should be pushed from the HTTP server, for example down into a database, to again make the HTTP server stateless.

SOAP on the other hand make no assumptions about server state. In RPC client-server systems, it is common for clients to manipulate the same server resource over repeated fine-grained operations, often relying on server-state and “sticky routing” to the server with the active session in between calls. Many SOAP implementations directly provide tools for server statefulness (JAX-WS @WebMethods on Stateful EJBs, for instance).

Statelessness provide scalability and flexibility, by allowing any compatible server to receive and process any request, without prior contact with the client.

Statelessness is a grey area; possible but discouraged in HTTP, possible and accepted in SOAP. Again, SOAP cannot be said to truly conform to the REST architectural style, which discourages server state.

Layered-System (LS), Layered-Client-Server (LCS)

By organizing applications in layers, each layer can abstract away the implementation details of the layers below. This allows for looser coupling between the client and origin server, since any number of layers can exist between the two, providing translations between otherwise incompatible nodes. It is also possible for a single layer to loadbalance the requests across any number of systems behind it, allowing for great scalability.

As a concrete example of the Layered style from this project scope, the Telenor middleware stack is layered both in terms of systems (the Channel Services system being an aggregator-layer above the Business Core, and again NewDelfiAPI), and inside the individual systems themselves (Web service layer, Business Logic layer, Data Access layer etc.). It is a very common architectural style.

Both HTTP and SOAP allow any number of layers between client and origin server. The major difference is that the HTTP protocol allows a uniform handling of messages passing through all intermediate/proxy layers, since all control information for the message is located in the well-defined HTTP headers which are understood by all layers. Intermediary layers do not have to concern themselves with the body content. With SOAP, it becomes harder for intermediaries such as proxies or firewalls to make informed decisions about the message, since the HTTP headers will often not contain much information about the intention of the message. This does not prevent SOAP systems from being Layered-Client-Server, it just means that the interface between adjacent layers becomes part of the application-design and not a protocol-specification, as it is for HTTP.
SOAP, and the SOA architectural style, is a derivation of the Layered-Client-Server style as well as HTTP/REST is. The uniform interface restriction of REST/HTTP just makes layering easier.

**Virtual-Machine (VM), Code-on-Demand (CoD)**

Code-on-Demand is the possibility of transferring executable code from one location in a network-application to another. A common example for the web is Javascript stored on the server, transmitted and executed on the client. The same is the case for Java Applets; the difference being that Javascript is transferred as source code to be interpreted client side, whereas Java Applets are transferred as binary code (although not machine-code, but byte-code for the JVM).

Code-on-Demand provides flexibility (like almost all other REST styles) and allows network-applications to offload computations to where the result is needed – the client. This can increase the user-perceived responsiveness of applications as well as reduce load on central servers.

Adding a Virtual Machine to the client also ensures that the server producing/storing the code to execute does not have to worry about a host of unknown client execution environments. Aging server code will still work on future clients, as long as the client provides a compatible VM.

Neither REST/HTTP nor SOA/SOAP rely in particular on VM and Code-on-Demand as they in themselves define just the transport protocols, not the target execution environment. Adding VM and CoD to large-scale network applications is a logical choice regardless of transport protocol.

**SOAP web services do not utilize CoD as much as hypermedia applications (the web) does over HTTP. RESTful services (described in more detail later) are likely to make more use of CoD than SOAP applications, due to REST exposing resources where SOAP expose operations. A REST “resource” is more likely to be Javascript code, XSL stylesheets or similar, than the outcome of a SOAP RPC method call is.**

**Uniformity (U)**

The REST “uniformity” restriction dictates that all components should share a uniform interface, allowing communication between components with different purpose and implementation details. That is a very general requirement to make, and easy to misinterpret. For instance, as long as the components all rely on TCP packets, does that mean they share a uniform interface? Ultimately, each component is required to understand the data it receives, if it is to process it. This requires the components to agree on data structures, semantics etc. This common agreement then becomes the uniform interface.

For the web (which is the original “REST application”), the uniform interface is defined by the HTTP protocol/version, specifically by the headers and methods. The body content is not part of the uniform interface, and is only required to be understood by the client and origin server, not the “uniform interface intermediaries”.

SOAP does not have the same kind of uniform interface. True, SOAP utilize HTTP as the primary transport layer, but as described in the Layered-System comparison above, SOAP reduce HTTP to a dumbed-down transport layer by moving information about the message from the headers into the body.
On top of this dumbed-down HTTP, SOAP adds its own uniform interface – the SOAP envelopes. There is not a lot of protocol-defined general information about the message itself in such an envelope. The message body is concerned with mapping the transmitted values to RPC method calls in the receiving end (aka. the binding), and requires application-specific knowledge in both the client and server ends, exactly as if the client and server were interacting via local function calls.

Addition of “higher-level” WS-* protocols such as WS-Security does not alleviate the requirement for application-specific knowledge about the operation being called. It just adds additional information about other aspects of the request/message, such as user-authentication method (LDAP, HTTP Basic Auth, …) to use.

*The interface uniformity of REST becomes possible by REST having only pre-defined operations (the HTTP protocol methods/headers) that treat all resources the same way. SOAP on the other hand has user-defined operations, which makes implementation of generic intermediaries much more difficult.*

*Keeping resource-manipulations within the pre-defined HTTP operations allows construction of generic intermediaries that can intelligently make decisions about the current request, rather than just be simple proxies. This is a clear advantage of REST over SOA / SOAP.*

In summary, the REST architectural style has several advantages that the SOAP protocol does not. Thus, SOAP cannot be said to be an implementation of REST, the way that HTTP is.

### 4.1.4 REST concepts

The architectural-style study above sheds some light on how REST is defined, but little on the practicalities of REST when applied to HTTP and RESTful web services.

**Resources and representations**

Two central concepts in REST are those of the *resource* and *representation*. In HTTP, any URI identifies a resource (or collection of resources). Any resource can have multiple representations. The distinguishing between resource and representation of a resource allows heterogeneous systems to expose native resources (files etc.) via representations understandable to the current client.

When you request a resource (GET operation), you also identify the desired representation type of the resource – or allow the server to make the choice for you. Representation choice will often be done via MIME-types such as “text/html” or “text/xml”. Thus, the representation a client sees for any given resource can be independent of actual implementation technologies on the origin server.

**Non-REST web extensions**

Deviations from the REST style exist; extensions which break one or more constraints. One of the examples from [RFREST], quite relevant to the context of this project, is the Java vs JavaScript situation. Java, while in certain aspects a platform and language considered superior to JavaScript, never gained widespread use in the web, at least not when compared to JavaScript.
Fielding attributes this to the fact that JavaScript fits the deployment model of the web better: Java applets break the Layered constraint by allowing the applet to communicate directly with the origin server, instead of piping requests through all intermediaries in a uniform manner. Furthermore, the code is not visible for reuse/alteration, and cannot be partially rendered by the client, which increase user-perceived performance.

Another example of a web extension that is a poor match for the REST constraints is Cookies, which potentially break the Stateless constraint by sending client-session information along with each request.

One of the purposes of the “REST dissertation” is to help identify such mismatches for the web architecture, in order to better guide the future evolution of the web.

4.1.5 JAX-RS and RESTful services
In the architectural-style description I evaluated how well SOAP conformed to the REST architectural style (which it did not fully). There exist web service implementations based directly on the REST architectural style – collectively called “RESTful services”.

RESTful services
“RESTfulness” is not a concrete API specification but merely an indication of conformance with the constraints imposed by the REST architectural style.

If REST was originally intended as the architectural style for distributing web resources to human end-users, RESTful services are an implementation of that style, meant for programmatic access to resources using the same distribution model.

In both cases, a resource (content or service) is accessed and possibly manipulated by a client capable of understanding the (representation of the) resource. RESTful services is thus an extension of the existing web, adding capabilities for network based program-to-program interactions beyond those supplied by the HTTP protocol itself.

WS-* services are essentially RPC over HTTP, exposing operations as you would in a local programming language. This naturally lends itself towards the Service Oriented Architecture style, with focus on the operation: you expose many fine-grained services and combine these into applications. However, many of the positive properties of HTTP are lost when HTTP is reduced to a transport-layer for what is essentially program control messages with interleaved data (XML SOAP documents). Fielding describes such misuse of HTTP in chapters 6.5.2 and 6.5.3. Although his writings precede WS-*, it applies to WS-* very well.

The RESTful web service approach focus on the resource, instead of the operation. As with any “typical” web resource, you define a URI (scheme) for the resource(s) you wish to expose, and let the basic HTTP GET/PUT/POST/DELETE operations act as your operations on those resources. The HTTP methods are to be interpreted as operations on resources according to this scheme (from Wikipedia [WIKIREST]):
<table>
<thead>
<tr>
<th>Resource</th>
<th>GET</th>
<th>PUT</th>
<th>POST</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection URI, such as <a href="http://example.com/resources/">http://example.com/resources/</a></td>
<td>List the URIs and perhaps other details of the collection's members.</td>
<td>Replace the entire collection with another collection.</td>
<td>Create a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.</td>
<td>Delete the entire collection.</td>
</tr>
<tr>
<td>Element URI, such as <a href="http://example.com/resources/item">http://example.com/resources/item</a> 17</td>
<td>Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.</td>
<td>Replace the addressed member of the collection, or if it doesn't exist, create it.</td>
<td>Not generally used. Treat the addressed member as a collection in its own right and create a new entry in it.</td>
<td>Delete the addressed member of the collection.</td>
</tr>
</tbody>
</table>

**JAX-RS**

One important aspect of RESTful services, is that no fixed cross-platform API specification (such as SOAP) for RESTful services in general exist. It is up to individual implementers to define how they interpret “RESTfulness”, and provide APIs/tools to support that definition.

One such implementation of RESTful services is the JAX-RS API specification. This API specification is part of Java EE, but can be used in simpler Servlet environments as well.

JAX-RS is, as the name lets on, a sibling of JAX-WS. JAX-WS allows developers to create web services by adding simple Java annotations such as @WebService and @WebMethod to JEE applications.

JAX-RS provides similar annotations, but instead with with focus on the URI on which a given resource is available. Any resource in a system of RESTful services should have a unique id, or a combined key resulting in such a unique id.

As an example, I return to the Account object on Figure 5: Account class hierarchy example, this time with an added accountNumber property:

![Account class hierarchy example](image)

The Account object itself is still a simple Java POJO, and can be made accessible on URLs such as
The first URL produce a collection of Account resources (assumed to be the full collection of all account resources), the second an individual Account specified by the \{accountNumber\} URL segment.

To achieve this via JAX-RS would require a Java class like this:

```java
@Path("/account")
public class AccountService {
    @GET
    @Path("accountNumber")
    @Produces("text/xml")
    public Account getAccount(@PathParam("accountNumber") int accountNumber) {
        Account account = ... // find account
        return account;
    }
}
```

The @Path annotation used on the class directs all requests on the /account URL (relative from the applications root context-URL) towards this class. GET requests on /account with an additional integer-segment, such as /account/111 will match the getAccount() method, which then gets the accountNumber parameter injected.

Returning a “text/xml” representation of “Account” object like shown above requires the object to be XML-serializable, which is not a REST-specific concern, and typically handled by the “Java Architecture for XML Binding”, JAXB, framework.

There are many additional details to the JAX-RS specification, allowing developers much flexibility in how they construct applications, but the above examples shows the basics.

**Conneg**

Of particular interest to me is the Content Negotiation (“Conneg”) capabilities of JAX-RS. Conneg is not unique to JAX-RS – it is the same basic HTTP mechanisms used by browsers & web servers to settle on HTTP version, as well as other areas like supported mime-types and compression algorithms.

JAX-RS allows the same mime-type content negotiation (illustrated above by the getAccount() method producing text/xml) content.

For programmatic access to resources, you can add application-defined media types, with built-in versioning, so it would be possible to define an application-specific versioned media type such as

```
Accept: application/vnd.telenor.account+xml;version=1.2
```

If a client requests such a media type, the server would know it is safe to return any version 1.x of the Account object, but the client will only be capable of understanding features from version 1.0 through 1.2. It would not be allowed for the server to return version 2.x or greater of the...
Account object, since the major-version change indicates non-backwards compatibility. The server would have to fall back to previous version 1-editions of the Account object, or respond with HTTP status code 406 “Not Acceptable” if it is not possible to produce version 1.x of the requested resource.

That provides clients/servers with a flexible way of negotiating which version of a resource is being operated on, but does not provide ways for the server-application to internally organize/handle the different versions of objects.

For minor-version increments, Bill Burke [JAXRSBOOK] suggests using XML Schemas with room for additions, assuming you use XML as primary data format (JSON is also a common transport format for RESTful services). Such an XML Schema should from version 1.0 allow any additional elements and attributes as valid (but unused) content. Later editions of the same type will then add details specifying which attributes or elements are now part of the type definition:

```
<schema targetNamespace="http://www.telenor.dk/account"
   xmlns="http://www.w3.org/2001/XMLSchema">
  <complexType name="accountType">
    <attribute name="accountNumber" use="required" type="integer"/>
    <anyAttribute/>
    <any/>
  </complexType>
</schema>
```

The HATEOAS principle

“The web”, as in human-browsable Internet, is navigable largely due to the Hypermedia As The Engine Of Application State (HATEOAS) principle. Every page rendered by a browser is considered an application state, and the application can transition to other states by following links to other resources/states. It is the responsibility of each resource (web page) to provide meaningful state-transition links. This works well for humans who are capable of interpreting the current content and making decisions on which state to transition to next – which link to click on.

The same does not necessarily apply to programmatic clients consuming RESTful service content rather than web pages. Client programs need pre-programmed knowledge about the resources with which they are interacting. It is, however, possible to generalize this knowledge, so client programs do not need resource-specific knowledge. This can be done by providing links with specific metadata, so the client only has to understand this metadata in order to make a selection of which resource (application state) to transition to.

An approach described in “RESTful Java with JAX-RS” [JAXRSBOOK] is to embed Atom[^4] links in the content provided by a resource, or provide the “Link” HTTP header parameter. An example response document for a get-resource request which retrieved order details could supply a link for how to cancel such an order:

The client application would have to understand the concept of “cancelling orders”, and when
to make use of that, but would not need pre-programmed knowledge of resource URI locations
or syntax for cancelling orders, since that information is provided by each individual
(cancellable) resource. Similarly, Links for all other conceivable application state transfers can
be provided as needed.

This Link-header building has ascended from “good idea” to “part of the JAX-RS API
specification” as per JAX-RS 2.0, providing easy mechanisms for servers to set the links, and
for clients to discover and traverse the links.

4.2 REST / JAX-RS prototyping

REST and JAX-RS is undoubtedly a mature programming API, but one I have little-to-no
experience with.

The prototypes of REST should, like the SOAP prototypes, attempt to solve the known issues
in the Telenor middleware, and not alone be REST-experiments for the sake of REST. Given
what I now know about REST, based on the literary study of Roy Fieldings REST dissertation,
and the JAX-RS book by Bill Burke, I intend to prototype the following issues:

1. Axis1 RPC/encoded services
2. Versioned interfaces
3. Unnessecary tranformations
4. Run-time interface discovery

Issues 1 and 4 relate entirely to SOAP and the concrete SOAP tooling in use. Issues 2 and 3
appears to have elegant solutions using REST, using Content Negotiation for versioned
interfaces, and passing raw XML instead of relying on “forced transformations” a la the
generated Java SOAP proxies.

4.2.1 Prototype projects

As with the SOAP prototypes, these are experimental prototypes. I have constructed two simple
projects:

- services-jaxrs
- client-jaxrs

Issue: Versioned interfaces

To experiment with Content Negotiation, the following simple class hierarchy is created, in
project services-jaxrs:
This server-application provides three methods (in AccountService), which implement all the HTTP operations supported by the application. Interface versioning is tested via the `getAccount(HttpHeaders, Integer)` method.

JAX-RS will be responsible for providing mechanisms for content negotiation, used to determine the clients preferred interface version.

The application itself must internally support multiple version of the same resources – in this case the Account + InstallmentPlan classes, and their relationship. To keep the application simple, the entire object model is replicated in versioned packages with the changes from version 1.0, through 2.0 being:

1. Version 1.1 adds an “owner” String property to Account. This is a minor change, since it is backwards compatible (so long as the original XML Schema describing Account contains `<anyElement/>`).

2. Version 2.0 changes the InstallmentPlan cardinality from 1-1 to 1-*, allowing one Account to contain multiple InstallmentPlan objects. This is not backwards compatible, so the major version number is increased.

Replication and manual versioning of the entire domain model is obviously not a very scalable or practical way for the application to handle interface versioning. These application-internal versioning concerns are not the focus of the prototype (the REST interface is), so this simple approach works fine for the prototype.

The `getAccount(HttpMethod, Integer)` method is implemented as follows:

---

**Figure 10: Versioned interfaces prototype class hierarchy**
public class AccountService {

    @GET
    @Path("{accountNumber}")
    @Produces("application/vnd.telenor.account+xml")
    public Object getAccount(
        @Context HttpHeaders headers,
        @PathParam("accountNumber") Integer accountNumber) {

        // Extract accepted version (if any)
        Integer[] acceptedVersion = getVersionFromHeader(
            MediaType.valueOf("application/vnd.telenor.account+xml"), headers);

        // Switch on major requested version
        switch (acceptedVersion[0]) {
            case 1:
                model.v1_1.Account account1 = new model.v1_1.Account();
                account1.accountNumber = accountNumber;
                account1.owner = "Morten";
                account1.installmentPlan = new model.v1_1.InstallmentPlan(1001);
                return account1;
            case 2: {
                model.v2_0.Account account2 = new model.v2_0.Account();
                account2.accountNumber = accountNumber;
                account2.owner = "Morten";
                account2.installmentPlans = new model.v2_0.InstallmentPlan[] {
                    new model.v2_0.InstallmentPlan(2001),
                    new model.v2_0.InstallmentPlan(2002)};
                return account2;
            }
            default:
                throw new WebApplicationException(Response.Status.NOT_ACCEPTABLE);
        }
    }

    private Integer[] getVersionFromHeader(MediaType producedType,
        HttpHeaders headers) {
        for (MediaType acceptedType: headers.getAcceptableMediaTypes()) {
            // For the supported (produced) type...
            if (acceptedType.isCompatible(producedType)) {
                // ... extract version, if present
                Map<String, String> param = acceptedType.getParameters();
                if (param != null && param.containsKey("version")) {
                    String[] versions = param.get("version").split("\.");
                    Integer major = Integer.parseInt(versions[0]);
                    Integer minor = Integer.parseInt(versions[1]);
                    return new Integer[] {major, minor};
                }
            }
        } // Default version, if not specified/found, is 2.0
        return new Integer[] {2,0};
    }
}

The highlighted JAX-RS ensures that HTTP GET requests which…

- Contains the “Accept: application/vnd.telenor.account+xml” header and
- Targets /account/{id}
... are directed to the getAccount(HttpMethod, Integer) method. This method then inspects the HttpMethod for optional parameters, specifying which version of “vnd.telenor.account+xml” the client accepts. If no parameter is specified, the latest version will be returned. In real-life situations it might be more appropriate to return version 1.0 unless clients explicitly tell you they understand later versions.

A simple JAX-RS (2.0) client program, from project jaxrs-client, then use this method:

```java
public static void main(String[] args) {
    ClientBuilder clientBuilder = ClientBuilder.newBuilder();
    Client client = clientBuilder.build();

    Builder builder = client
        .target("http://localhost:8080/services-jaxrs/account")
        .path("10")
        .request("application/vnd.telenor.account+xml;version=1.4");

    Invocation i = builder.buildGet();
    String s = i.invoke(String.class);
    System.out.println("RESPONSE: \n" + s);
}
```

The printed response is:

RESPONSE:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><account accountNumber="10">
    <installmentPlan planId="1001"/>
    <owner>Morten</owner>
</account>
```

The client claims to understand version 1.4 in the accept header, but the server only knows 1.1 within the same major version.

Requesting version 3.0 produces a “HTTP 406: Not Acceptable” response. And no version specified produces the version 2.0 response:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><account accountNumber="10">
    <installmentPlans planId="2001"/>
    <installmentPlans planId="2001"/>
    <owner>Morten</owner>
</account>
```

**Issue: Unnecessary transformations**

The examples above also highlight another interesting feature, allowing REST applications to bypass unwanted transformations from XML to Java on-demand.

The exact same client code snippet as above, with the specification for accepted resource representation highlighted:
```java
public static void main(String[] args) {
    ClientBuilder clientBuilder = ClientBuilder.newBuilder();
    Client client = clientBuilder.build();
    Builder builder = client
        .target("http://localhost:8080/services-jaxrs/account")
        .path("10")
        .request("application/vnd.telenor.account+xml;version=1.4");
    Invocation i = builder.buildGet();
    String s = i.invoke(String.class);
    System.out.println("RESPONSE:\n" + s);
}
```

By explicitly informing the JAX-RS client framework to produce a String from an XML resource, I get the raw XML, not XML serialized to an Account object.

In fact, auto-serializing to an Account object would only be possible if I had created XML Schemas (or WADLs) for the service, and client-side produced a Java-representation of the Account schema definition. With a few additions (and JAXB) this would have been possible:

```java
public static void main(String[] args) {
    ClientBuilder clientBuilder = ClientBuilder.newBuilder();
    Client client = clientBuilder.build();
    Builder builder = client
        .target("http://localhost:8080/services-jaxrs/account")
        .path("10")
        .request("application/vnd.telenor.account+xml;version=1.4");
    Invocation i = builder.buildGet();
    Account a = i.invoke(Account.class);
    System.out.println("RESPONSE:\n" + s);
}
```

In other words, the client gets to runtime decide which representation of the resource it wishes to retrieve, and how the representation should be mapped to Java (raw XML or serialized to Java objects).

This would allow for pass-through methods where middle-tiers of the architecture are not forced to transform the same data over and over again, but can act as pure proxies.

In fact, it would be possible for the middle tiers to perform this kind of proxying by only looking at the headers, not content, thus being very effective indeed.

### 4.3 REST conclusions

#### 4.3.1 REST versus SOAP in general

REST as an architectural style appears to be very scalable and flexible, and well suited for network-application infrastructure. That this is true has been demonstrated in full by the internet, which serves as a reference-implementation / breeding ground for the REST architectural style.
RESTful services could, if the REST principles are properly followed, be superior to WS-*/SOAP in many ways: for example, a uniform interface which allows for looser coupling and on-the-fly content negotiation using resource-representations.

Where WS-* is the sprawling sum of many commercial interests and (somewhat over-elaborated) theoretical uses, REST is defined as the constraints a network-based application architecture should adhere to if, it wishes to have the same positive architectural properties described in Fielding’s dissertation.

For WS-* applications that make use of the more advanced functionality of WS-*, such as distributed transactions (Java Transaction Architecture, JTA + WS-Transaction), RESTful services might be too “basic” or under-specified if you will, forcing you to build home-grown solutions to problems already solved in WS-*. However, in many cases, the more advanced WS-* functionality could (and probably should) be altogether avoided by better application design. Taking distributed transactions as an example, both Bill Burke and Roy Fielding commented on how/if REST could support distributed transactions in this JSR311 (JAX-RS proposal) discussion:

https://java.net/projects/jsr311/lists/dev/archive/2008-06/message/124

The gist is, that while REST does not explicitly prohibit distributed transactions (it would not require session state, only resource state transitions), it would often be a better solution to have the client build the complete set of modified resources locally and submit in a single message. So, for now at least, that particular feature is not part of the JAX-RS API.

Obviously, avoiding advanced features such as distributed transactions is not always possible or desirable, so REST cannot be said to be generally preferable in all situations.

I believe the choice should ultimately depend mostly on your applications environment and clients. With few (or simply well-known and controlled) clients, SOAP provides strong tooling and clean (static) interfaces. With large-scale applications used by unknown clients, particularly applications with interfaces exposed to the public internet, the more flexible REST approach seems a far better choice.

### 4.3.2 Telenor issues and REST

For the concrete Telenor issues of versioned interfaces and unnecessary transformations, REST provides very clean solutions as demonstrated in the prototypes:

- Versioned interfaces can be handled via content negotiation, custom Accept-header datatypes and flexible XML Schemas.
- Unnecessary transformations is solved as a side-effect of the content negotiation – if a client wishes to pass-through the retrieved data, it can just choose to not deserialize the received XML into Java objects.

REST (and JAX-RS) itself does not provide mechanisms for an application to organize/version its resources – it only allows the client and server to easily negotiate which resource-version is to be used. Internal representation and storage of the resource is a platform- and application-specific design issue.
5 To-be architecture recommendation

Based on learnings from the SOAP prototyping and the REST literary study plus prototyping, it should be possible to determine the best course of action for the Telenor middleware, in order to solve known issues.

I had originally intended to use quality attributes as defined by Len Bass et al [SAIPBOOK] to clearly describe the architectural goals of any future changes to the middleware infrastructure. These quality attributes would factor in both technical attributes like performance and scalability, but also business attributes like available developer knowledge (buildability).

However, during the course of this project (late April 2013), it has been decided to initiate a major architectural re-invention at Telenor DK, not limited to “just” the middleware area. This suddenly leaves all systems mentioned in this project in their end-of-life phase, expected to be terminated 2015-2016 at the latest – earlier if in any way possible. The scope of this “new stack” is vast, encompassing around 70 distinct large-scale systems such as “billing engine”, “customer service management”, “network provisioning”, “sales channels” and “website”, just to name a few. The goal is to entirely replace more or less all business-related software, leaving only the network as-is. This is a massive undertaking with total costs estimated to be well beyond 1 billion DKK\(^5\). It will require the attention of all available resources (as well as a swarm of consultants), immediately resulting in halted development on the existing system stack – only small-scale projects and necessary maintenance remains.

With this in mind, a few simple decision guidelines will serve instead of actual quality attributes:

- Work-effort: only few development resources will be available.
- Development time: long-term development is out of the question.

It is obvious that expanded use of an existing SOAP framework (Apache CXF) within the current architecture is vastly cheaper (and faster) than a complete rewrite of all existing infrastructure to REST. Any potential cost-savings associated with switching to REST (potentially leading to faster development cycles and fewer of the known issues) are highly unlikely to ever match the costs incurred by the rewrite itself.

Therefore, REST is automatically relegated to a thought-experiment in the context of how to upgrade the current middleware.

REST is, however, a very interesting candidate for future systems integration (of which there is suddenly likely to be quite a bit), and would be my personal choice over SOAP in many situations – particularly situations where clients are not easily updated to match server-changes, i.e. are unknown or not under direct control.

---

\(^5\) “1 milliard KR”, in Danish – to avoid confusion between a DK billion and a UK billion.
5.1 To-be architecture

Within the constraints of SOAP upgrades, the prototyping showed that CXF could be implemented alongside Axis by adding a CXF web service layer to the systems that currently rely on Axis to expose web services. This applies to Business Core and NewDelfiAPI.

That allows client-systems currently forced to use Axis as client-framework to switch to their preferred (JAX-WS compatible) web service client framework. This applies to Channel Services and Order Engine Workflows.

On the updated Component diagram below, the Axis interfaces and Axis clients have been replaced with CXF and JBossWS respectively:

![To-be component diagram](image)

The above figure shows the final setup, where Axis is fully replaced by CXF. For a long period of time, the two would co-exist, as shown in this smaller component diagram only displaying the Channel Services and Business Core:
Figure 12: Dual web service frameworks

Given the “new stack” timeframe, and the time it would take to update all the 100’s of interactions, it is doubtful if a full transition from Axis to CXF could be carried out, but addition of CXF would allow a seamless switch away from Axis in the few critical areas with high performance requirements, or known concurrency issues.
6 Summary and conclusions

6.1 Summary

In this project I have described the current Telenor web service infrastructure architecture and its problems. I have analysed and prototyped how SOAP framework upgrades could solve these issues – specifically the incompatibility issues caused by the outdated Apache Axis 1 framework and its use of the “RPC/encoded” SOAP binding style.

Via prototyping I found that the best approach is to expose a secondary (contemporary / standard) set of SOAP web services, using the Apache CXF framework. With clients migrating from the old Axis1 towards the CXF services, Axis1 would in time become redundant and be removed.

Not all issues are easily solvable within the SOAP architecture – the multi-tiered architecture with each tier having its own Domain Model means that SOAP introduces several unnecessary transformations from XML to native Java. That, and hard-to-change interfaces have no elegant solutions within the SOAP architecture.


REST was found to provide several architectural benefits, as well as concrete solutions to the transformation and interface issues, primarily relying on the HTTP-inherent Content Negotiation mechanisms.

6.2 Conclusions

It is possible to upgrade the current Telenor web service middleware infrastructure, but not feasible to do so in a manner which solves all the known issues. Only the CXF-and-Axis dual web service layer approach is realistically possible, and that will not in itself solve the “Unnecessary transformations” problem. However, it may lessen the problem significantly by utilizing much newer XML parser implementations, thus reducing XML transformation execution time.

RESTful services, as implemented by JAX-RS is found to be a very interesting alternative to SOAP (JAX-WS), with great flexibility and scalability provided by the REST architectural style. The full rewrite required to switch from SOAP to REST(ful services) is costly, and not feasible for the Telenor-case. In general, the benefits provided by REST must be compared against the feature-completeness of WS-*; so there is no “best” approach – it depends on the concrete application, and its client systems. REST favors very loose coupling between systems, which is good for internet applications (where you are unlikely to control or know the clients), whereas SOAP favor strict predefined interface agreements, which is good for local-network applications, or applications with controlled clients.
7 Related work

7.1 Consumption of Axis1 RPC/encoded services

Others have addressed some of the issues we have with Axis services, as multiple discussions on the Stack Overflow forum shows [SORPC]. A common understanding in these discussions is that Axis has simply outlived its purpose, and should be replaced by another framework.
8 References


By Ivan Damgård.


By Bill Burke.

By Jakob Eyvind Bardram, Henrik Bærbak Christensen and Klaus Marius Hansen.


By Roy Thomas Fielding

By Len Bass, Paul Clements and Rick Kazman.

[SAIPCS] Course: Software Architecture in Practice course at Aarhus University.
Stackoverflow discussion on consumption of RPC/encoded WSDLs: http://stackoverflow.com/questions/7284126/best-way-to-consume-rpc-encoded-webservice


Description of the term “Middleware” on Wikipedia: http://en.wikipedia.org/wiki/Middleware

Representational state transfer on Wikipedia: https://en.wikipedia.org/wiki/Representational_state_transfer


9 Appendix

9.1 Prototype project sources

All prototype projects are developed using:

- JBoss Application Server 7 as JEE container: [http://www.jboss.org/jbossas](http://www.jboss.org/jbossas)

The zipped source for all Eclipse projects are embedded in this PDF document, here:

[prototype-src.zip](#)