MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21b/2017/03.20.11.44-TDI

MODELO ARQUITETURAL PARA GERENCIAMENTO DE VERSÕES DE CONTRATOS DE SERVIÇOS WEB

David de Souza França

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Eduardo Martins Guerra, aprovada em 23 de fevereiro de 2017.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34P/3NHNH9E>

INPE
São José dos Campos
2017

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/6921
Fax: (012) 3208-6919
E-mail: pubtc@inpe.br

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21b/2017/03.20.11.44-TDI

MODELO ARQUITETURAL PARA GERENCIAMENTO DE VERSÕES DE CONTRATOS DE SERVIÇOS WEB

David de Souza França

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Eduardo Martins Guerra, aprovada em 23 de fevereiro de 2017.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34P/3NHNH9E>

INPE
São José dos Campos
2017

Dados Internacionais de Catalogação na Publicação (CIP)

França, David de Souza.

F844m Modelo arquitetural para gerenciamento de versões de contratos de serviços web / David de Souza França. – São José dos Campos : INPE, 2017.

xxii + 98 p. ; (sid.inpe.br/mtc-m21b/2017/03.20.11.44-TDI)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2017.

Orientador : Dr. Eduardo Martins Guerra.

1. Interoperabilidade. 2. Contrato. 3. Mensagem XML. 4. Scripts. 5. Conversão. I.Título.

CDU 004.777

Aluno (a): *David de Souza França*

Título: "MODELO ARQUITETURAL PARA GERENCIAMENTO DE VERSÕES DE CONTRATOS DE SERVIÇOS WEB"

Aprovado (a) pela Banca Examinadora em cumprimento ao requisito exigido para obtenção do Título de *Mestre* em *Computação Aplicada*

Dr. Nandamudi Lankalapalli Vijaykumar

Presidente / INPE / SJCampos - SP

Dr. Eduardo Martins Guerra

Orientador(a) / INPE / São José dos Campos - SP

Dra. Karine Reis Ferreira

Membro da Banca / INPE / São José dos Campos - SP

Dr. Nilson Sant'Anna

Membro da Banca / INPE / SJCampos - SP

Dr. Fábio Fagundes Silveira

Convidado(a) / UNIFESP / São José dos Campos - SP

Este trabalho foi aprovado por:

( ) maioria simples

(X) unanimidade

São José dos Campos, 23 de fevereiro de 2017

"Se eu vi mais longe foi por estar de pé sobre ombros de gigantes.".

Isaac Newton
em *"Cartas a R. Hooke"*, 1676

A minha família e amigos.

AGRADECIMENTOS

Durante todo o desenvolvimento deste trabalho tive o apoio e ajuda de muitas pessoas e quem devo meus sinceros agradecimentos:

Ao Instituto Nacional de Pesquisas Espaciais (INPE) e à Coordenação da Computação Aplicada (CAP) pela oportunidade de realização deste estudo e apoio em todas as necessidades surgidas durante sua realização.

Ao professor Dr. Eduardo Martins Guerra pelas orientações, ensinamentos e conselhos transmitidos durante o desenvolvimento deste trabalho e o fazendo sem medir esforços para me ajudar.

Ao professor Me Fernando Mauro de Souza pela sempre presente procupação com meu desenvolvimento, conselhos e incentivos.

a minha família que sempre me apoiou, em especial a minha esposa Giulianna pelo incentivo e compreensão antes e durante todo o desenvolvimento deste trabalho.

Aos professores do curso de pós-graduação do INPE pelos ensinamentos.

Aos membros da banca examinadora que contribuiram para a melhora do conteúdo deste trabalho.

RESUMO

Na tentativa de prover serviços de interoperabilidade e difusão de informações pertinentes a muitos ramos da computação, muitas aplicações *web* são baseadas ou consomem dados de outras aplicações. Isto é um problema quando as aplicações evoluem em velocidades diferentes e compartilham formatos definidos nas trocas de mensagens. Este trabalho apresenta uma pesquisa para o desenvolvimento de um modelo arquitetural que permite a evolução no formato das mensagens utilizado pelas aplicações sem a quebra da interoperabilidade, permitindo que cada uma evolua de forma independente. A solução proposta é gerar, a partir das mudanças nos contratos das mensagens XML, *scripts* para refatoração de tais mensagens para que estas sigam um modelo de contrato ou outro, tornando transparente às aplicações envolvidas qual a versão do contrato o destinatário da mensagem espera receber. As informações sobre as versões e transformações são gerenciadas por um serviço intermediário que centraliza as funções de gerenciamento de *scripts*, transformação de dados e entrega de mensagens. Esta pesquisa foi avaliada em um estudo abrangendo diversos cenários onde foram avaliadas a capacidade de gerar transformações de forma transparente às aplicações, o impacto do uso da arquitetura sobre aplicações em questões de alterações de linhas de código e qual o impacto sobre o desempenho da tramitação das mensagens entre as aplicações. Como principais conclusões desse estudo, foi possível perceber que a arquitetura é capaz de gerar transformações necessárias para conversão das mensagens e de forma transparente às aplicações que a utilizam, que necessita de poucas modificações nas aplicações e que é utilizável quando as aplicações não necessitam de respostas rápidas às requisições. Espera-se com essa pesquisa prover uma alternativa para resolver o problema de versões de contratos diferentes de uma forma transparente às aplicações, de fácil implantação e que independa do tipo de domínio das aplicações envolvidas.

Palavras-chave: Interoperabilidade. Contrato. Mensagens XML. Scripts. Conversão.

ARCHITECTURAL MODEL FOR CONTRACT MANAGEMENT OF WEB SERVICES

ABSTRACT

In an attempt to provide interoperability services and dissemination of information relevant to many branches of computing, many web applications are based or consume data from other applications. This is a problem when applications evolve at different speeds and share defined formats in the message exchanges. This paper presents a study for the development of an architectural model that allows developments in the message format used by applications without breaking interoperability, allowing each to evolve independently. The solution proposed is to generate from the changes in the contracts of refactoring scripts XML messages for such messages so that they follow a model contract or other, making transparent to applications which involved the contract version the message receiver expects to receive. The information about the versions and changes are managed by an intermediary service that centralizes scripts management functions, data transformation and delivery of messages. The proposal was evaluated in a study covering several scenarios, which were evaluated the ability to generate transformations transparently to applications, the impact of the use of architecture for applications in lines of code changes issues and the impact on the performance of the processing of messages between applications. The main conclusions of this study, it was revealed that the architecture is able to generate the necessary transformations for conversion of messages and transparently to applications that use it, it needs a few modifications in the applications and is usable when applications do not require rapid response requests. It is hoped that this research provide an alternative to solve the problem of different contract versions transparently to applications, easy to deploy and is independent of the type of field of applications involved.

Keywords: Interoperability. Contracts. XML messages. Scripts. Conversion.

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

SUMÁRIO

Pág.

1 INTRODUÇÃO

Na era da Internet, o sucesso das aplicações baseadas na *web* desempenhou um papel vital em disponibilizar informações em diferentes sistemas e localizações a todo tempo. Como um próximo passo evolutivo, serviços da *web* permitiram uma nova geração de aplicativos que abordam o novo fenômeno da construção de uma plataforma de propósitos gerais para criar eficientemente a integração entre processos de negócio, aplicações, empresas, parceiros e clientes.

Sobre *web service*, Kalin (2013) discorre que é raro que um sistema de *software* funcione perfeitamente de modo isolado. O típico sistema de *software* deve interagir com os outros sistemas que podem estar em diferentes máquinas, serem escritos em diferentes linguagens e utilizarem diferentes plataformas. A interoperabilidade não é apenas um desafio em longo prazo, mas também uma exigência atual de produção de *software*.

"Hoje, uma parte dos serviços computacionais que são disponibilizados para acesso e consumo é criada sob *Service Oriented Architecture* (SOA)" (FRANÇA; GUERRA, 2013), cujo princípio é tornar disponíveis em uma rede de computadores serviços ou funções de certas aplicações utilizando para isso um contrato entre o fornecedor dos serviços e seus consumidores. Nesse contrato estão estipuladas características descritivas informando ao consumidor do serviço as necessidades semânticas e sintáticas da sua utilização.

Esse tipo de arquitetura traz a vantagem de aplicações poderem consumir vários serviços de uma ou mais aplicações fornecedoras apenas conhecendo o contrato estipulado para isso.

Com o contrato descrevendo como devem ser acessadas e utilizadas as funções disponibilizadas pelo fornecedor, os consumidores utilizam arquivos sob essa estrutura para enviar e receber informações. O contrato também é o descritor estrutural dessas mensagens trocadas entre as aplicações, utilizando modelo para formalização dos arquivos de mensagens, onde se pode informar, por exemplo, nome dos nós (ou entidades) que uma mensagem obrigatoriamente deve possuir, suas multiplicidades, tipos de dados e atributos. A partir dessa descrição é possível perceber que a relação entre fornecedores e consumidores é fortemente baseada no contrato do serviço.

1.1 Motivação

Uma desvantagem desse modelo baseado em contratos é que a evolução de aplicações fornecedoras, aquelas que disponibilizam os serviços, podem alterar o contrato por elas utilizadas forçando as aplicações cosumidoras a se adequarem ao novo contrato do serviço para que possam continuar se comunicando. Nestes termos, uma evolução do contrato de serviço não é feito de uma forma fácil, pois devem ser feitas adaptações tanto no fornecedor do serviço como nos consumidores.

Serviços podem ser disponibilizados por diversos fornecedores ao mesmo tempo em que diversas aplicações também podem consumir estes serviços formando uma complexa estrutura de consumidores e fornecedores. Manter todos os fornecedores e consumidores atualizados para continuarem a consumir e fornecer serviços mesmo após a evolução de un contrato pode se tornar uma tarefa muito complexa, lenta e até mesmo inviável.

Para entender com que frequência esses arquivos de contrato são alterados França et al. (2015) faz um estudo sobre as alterações encontradas em contratos em aplicações industriais e *open source*. No estudo com aplicações industriais a pesquisa foi feita em um repositório gerenciado pelo controlador de versões Subversion (2016) que possui 3 projetos da Força Aérea Brasileira (FAB) compartilhando um mesmo contrato de serviços. Este modelo sofreu 31 alterações em 17 meses de projeto fazendo com que as aplicações que consumiam o serviço não pudessem mais utilizá-lo enquanto não se adequassem ao novo contrato cada vez que este era alterado.

Cada projeto envolvido era composto por equipes diferentes e que nem sempre estavam disponíveis para adequar cada aplicação consumidora ao novo contrato toda vez que ele era alterado. Isto significa que todas as aplicações que consumiam o serviço, nesses 17 meses, estavam constantemente necessitando de alterações para se adequarem ao novo contrato e nem sempre isso era possível fazendo com que essas aplicações deixassem de se comunicar

Outro estudo abrangeu projetos *open source* escritos na linguagem Java presentes em uma base de dados disponibilizada pela Universidade de São Paulo (USP). Nesta base de dados havia 67 projetos, gerenciados pelo controlador de versões Git (2016). Destes 67 projetos, 22 utilizam contratos para descreverem os arquivos de mensagens e em média para cada alteração em um modelo 13 arquivos de código fonte (nesse caso arquivos .java) também foram alterados. Isto é uma evidência de que pode haver um forte acoplamento entre os contratos de mensagens e o código fonte das

aplicações fornecedoras e consumidoras. Essa evidência mostra um impacto em todas as aplicações envolvidas quando ocorre uma mudança em um contrato pois gera uma demanda por alterações em todos os consumidores.

Para entender a causa dessas alterações Almeida e Guerra (2016) demonstraram essa rastreabilidade de entidades dentro de um contrato descrevendo a frequência de alterações de certos tipos de elementos, se projetos distintos possuem frequências de alterações similares para os mesmos tipos de entidades e quando ocorrem estas alterações.

Uma possível solução para este problema da evoluções de contratos é a utilização de modelo semântico para tratar as conversões entre os dados de diferentes versões. Ele é capaz de fazer uma inferência e apontar a melhor correspondência de certo tipo de dado em ambas as versões de um contrato. Para utilizar essa solução é necessário que cada campo de domínio utilizado em um serviço seja mapeado e incluído neste modelo semântico, o que não é trivial. Poucas aplicações na indústria utilizam tais modelos e em nenhum caso no conjunto de projetos pesquisados foi encontrado sua utilização. Sendo assim, considera-se como requisito deste trabalho que a solução seja aplicável a serviços que não possuam um modelo semântico associado.

1.2 Objetivo

Propor um modelo arquitetural que minimiza o impacto de mudanças efetuadas no contrato de serviços que não possuam modelos semânticos associados para permitir que aplicações continuem trocando mensagens sem que precisem migrar para o novo contrato.

1.3 Abordagem de solução

Na tentativa de se criar uma solução para o problema de mundaças em contratos este trabalho propõe a utilização de uma arquitetura que integrada às aplicações fornecedora e consumidora do serviço gere, de modo transparente à elas, uma forma de continuarem se comunicando mesmo que estejam utilizando diferentes versões de um contrato. Para isso, a arquitetura prevê um repositório de versões, um gerenciador dessas versões e uma ferramenta para a geração de *scripts* necessários para serem aplicados as mensagens trocadas pelas aplicações.

A solução proposta por esse trabalho é a utilização de *scripts eXtensible Stylesheet Language for Transformation* (XSLT) gerados a partir das mudanças entre versões de um mesmo contrato. Esses *scripts* são gerados pela ferramenta Chrysalis (DUARTE,

2010) que deve ser utilizada para a modificação do contrato. Nesse caso, as aplicações que utilizam versões de contratos diferentes podem aplicar aos seus dados os *scripts* de mudanças e adequá-los a uma versão em comum. Para tanto, esses *scripts* são disponibilizados em um repositório juntamente com outros dados sobre os contratos utilizados pelas aplicações para que se possa obter informações acerca de como se alterar os dados de certa versão de contrato para outra.

Um intermediador entre as aplicações fará o gerenciamento das informações entre mudanças de versões de um mesmo contrato, se responsabilizando em receber uma mensagem em uma dada versão de contrato, aplicar os *scripts* de mudança e enviar a mesma mensagem na estrutura esperada pela aplicação consumidora da mensagem.

O único ponto de mudança não está nas aplicações que se utilizam de um contrato de serviços *web*, mas na arquitetura onde estão inseridas, passando a utilizar esse intermediador como ponto de envio e recebimento de mensagens, deixando a cargo dele o gerenciamento das versões de contratos, bem como a aplicação dos *scripts* de conversão sobre os arquivos de mensagens.

1.4 Abordagem de avaliação

Para se avaliar a solução proposta, seguiu-se um roteiro onde foram criados vários cenários entre duas aplicações que desejam se comunicar, uma fornecedora de um serviço e outra consumidora.

Esses cenários abrangem os principais casos em que a solução pode ser aplicada. Primeiro cenário: quando não é implementado o modelo de arquitetura e a comunicação não necessita de mudanças na mensagem. Segundo cenário: quando é implementado o modelo de arquitetura, porém a mensagem não necessita de alterações. Terceiro cenário: quando o modelo de arquitetura é implementado e a mensagem necessita de alterações para ser entregue à aplicação que irá consumí-la.

Para cada cenário criou-se pontos de registros para a medição do desempenho da arquitetura. Primeiramente mediu-se a comunicação entre as aplicações sem a utilização da arquitetura proposta e depois utilizando a arquitetura para uma posterior comparação. Comparou-se os registro de desempenho dos cenários em que a mensagem não necessita de alterações utilizando e não utilizando o modelo proposto; e comparou-se os cenários onde utilizando o modelo proposto a mensagem necessita de alterações com o cenário onde ela não necessita de alterações.

Por fim, ainda sobre a avaliação, computou-se o impacto da adoção da arquitetura

sobre as aplicações, isto é, qual é a dificuldade de se adotar tal arquitetura em termos de linhas de código tanto para a aplicação consumidora do serviço quanto para a aplicação fornecedora do serviço.

1.5 Relevância

Uma dificuldade encontrada em interoperabilidade de aplicações é o desenvolvimento e manutenção das aplicações por várias equipes em diferentes lugares e períodos. Este modelo contribui com o abrandamento deste problema já que fornece uma forma das aplicações evoluírem em tempos diferentes e em muitos casos continuarem a se comunicar.

Cada equipe desenvolvedora não necessita mais estar em constante comunicação com as equipes de outras aplicações para saber se houve alterações nos contratos das mensagens. Este trabalho contribui com a facilidade de manutenção por parte de cada equipe não necessitando a todo o momento em que o contrato é alterado evoluir sua aplicação para se adequar ao novo contrato de mensagens.

A adoção de *web services* é visto como uma prática comum para a troca de informações e este modelo de arquitetura proposto que trata de problemas encontrados nestes tipos de serviço é de relevância pois vem ao encontro com a crescente necessidade de interligação de informações e a contínua comunicação entre aplicações.

Manter a interoperabilidade entre aplicações pode ser uma tarefa complexa pois necessita que as aplicações estejam utilizando sempre a mesma versão de contrato. Para isso devem sempre estarem atualizadas o que também exije equipes de desenvolvimento disponíveis o que nem sempre é possível. Esta pesquisa contribui com o desenvolvimento de formas de manutenção da interoperabilidade entre sistemas de forma relativamente simples, diminuindo o esforço de manutenção necessário quando a comunicação entre aplicações é interrompida. Além disso, esta pesquisa não utilizada modelos particulares a cada ramo de domínio de uma aplicação, como é o caso dos modelos semânticos, tornando-se uma solução mais abrangente e de ampla utilização.

Este modelo de arquitetura é importante no sentido de não gerar acoplamento entre as aplicações envolvidas na comunicação para se chegar a solução proposta. O modelo fornece uma forma das aplicações trocarem informações sem se comunicarem diretamente. Isso é um ganho quando há múltiplos fornecedores e consumidores pois a adição de uma aplicação não altera a arquitetura.

1.6 Originalidade

Existem publicações acerca de possíveis soluções para o problema da quebra de interoperabilidade entre serviços *web*, abordando vários aspectos do desenvolvimento e infraestrutura, porém nenhuma delas traz uma abordagem onde existe a conversão das mensagens por *scripts* com um gerenciamento dessas informações (versões de contrato, *scripts* e aplicações) pela infraestrutura do modelo. O gerenciamento dessas informações das versões de contratos e aplicações introduz um novo aspecto a solução deste tipo de problema, pois gerenciar manualmente essas informações pode ser difícil ou até mesmo inviável.

Outro aspecto original neste trabalho é a necessidade de apenas uma interface de acesso com apenas uma implementação para cada serviço de um fornecedor. Outras soluções propostas trazem múltiplas instâncias ativas de um serviço ou muitas interfaces de acesso, tornando dificil sua manutenção. Este trabalho simplifica esta questão, eliminando múltiplas instâncias e com apenas uma implementação e que não é gerenciada de forma manual.

Outra abordagem é a utilização de *web services* semânticos, onde deve-se modelar o conhecimento a fim de criar uma composição baseada na própria semântica dos dados, porém esse tipo de abordagem pode falhar quando a descrição semântica não é bem feita ou o algoritmo não reconhece a equivalência. Vale ressaltar que essa solução não se aplica ao contexto atual da indústria, onde raramente serviços semânticos são utilizados. No presente trabalho optou-se por não utiliza-los, mas a proposta cria um modelo genérico que pode ser aplicado a qualquer domínio, não necessitando de mapeamentos. Por esse modelo ser genérico o suficiente, não é necessário alterá-lo a cada vez que um novo domínio de uma aplicação é introduzido, dando mais um aspecto original ao modelo. Em outras publicações notou-se que na intrdução de um novo domínio há sempre alterações na solução proposta.

Existem também publicações que propõem soluções que utilizam *scripts* de conversão e intermediadores para a troca de mensagens, como o ESB (*Enterprise Service Bus*). Estas soluções seguem o padrão SOA(*Service-Oriented Architecture*) *Data Model Transformation*, que propõem esse padrão de solução com a geração de *script*, porém é feita de forma manual e o gerenciamento das versões de contratos e *scripts* também é feita de forma manual.

A presente pesquisa visa abordar um novo aspecto, introduzindo uma arquitetura que integra um componente desacoplador de aplicações intermediando as mensagens.

Também é feita a geração de *scripts* de conversões e o gerenciamento das distintas versões dos contratos e os *scripts* relacionados. Com isso, essa pesquisa propõe uma solução de baixa intrusão onde as aplicações não precisam ser migradas todas as mesmo tempo sempre que uma delas sofrer alterações.

1.7 Organização do trabalho

Esta monografia está estruturada da seguinte forma:

No capítulo 2 são apresentados os conceitos fundamentais para a compreensão do trabalho, com a respeito de arquivos XML, arquivos XSD, refatoração de arquivos XML, *scripts* XSLT, SOA. Serão apresentados também os conceitos sobre ESB e versionamento de serviços.

No capítulo 3 são apresentados os principais trabalhos relacionados a esta pesquisa que são encontrados na literatura.

No capítulo 4 é descrita a arquitetura da solução proposta, passando por uma visão geral e posteriomente detalhando cada componente envolvido, a dinâmica dos processos de troca e conversão de mensagens.

No capítulo 5 é descrita a implementação da solução proposta, quais tecnologias foram utilizadas e como cada componente foi criado.

No capítulo 6 é descrita a metodologia de avaliação utilizada no modelo arquitetural proposto, quais os itens a serem avaliados nos estudos de caso, como foi montado o ambiente de avaliação, quais dados foram obtidos e uma análise sobre estes dados.

No capítulo 7 são apresentadas as considerações finais sobre este trabalho, sua pesquisa, desenvolvimento e resultados de avaliações feitas. São apresentados também possíveis trabalhos futuros para dar continuidade a esta pesquisa.

2 FUNDAMENTAÇÃO TEÓRICA

O objetivo desse capítulo é apresentar as principais tecnologias e conceitos relacionados com o trabalho desenvolvido. Serão abordados conceitos de arquivos XML, sua relação com o modelo XSD, bem como tipos de refatoração de arquivos XML e a utilização de *scripts* XSLT como ferramenta de refatoração. São apresentados conceitos relativos à serviços *web*, protocolos SOAP e REST e suas diferenças, modelos WSDL, arquitetura SOA e o funcionamento de um *Enterprise Service Bus*. Serão ainda apresentados conceitos sobre versionamento de serviços.

2.1 Documentos XML e XSD

Extensible Markup Language, abreviado para XML, descreve uma classe de dados chamado documentos XML e parcialmente descreve o comportamento dos programas de computador que os processam. XML é um perfil de aplicação ou uma forma restrita de SGML, o Standard Generalized Markup Language [ISO 8879]. Por construção, documentos XML estão em conformidade com documentos SGML. (BRAY et al., 1998)

Na Figura 2.1 é apresentado um exemplo de um documento XML com uma *tag* inicial, uma final e uma de conteúdo. Cada elemento deve sempre ter uma *tag* de fechamento para cada uma aberta e pode-se opcionalmente ter subelementos e atributos.

Figura 2.1 - Exemplo de documento XML.

```
1 <?xml version="1.0" standalone="yes"?>
2 <BankAccount>
3     <Number>1234</Number>
4     <Type>Checking</Type>
5     <OpenDate>11/04/1974</OpenDate>
6     <Balance>25382.20</Balance>
7     <AccountHolder>
8         <LastName>Singh</LastName>
9         <FirstName>Darshan</FirstName>
10    </AccountHolder>
11 </BankAccount>
```

Fonte: Produção do autor

Apesar de um documento XML aceitar qualquer nome para definir uma tag (ele-

mento ou atributo), é possível definir um formato válido para elas. Esse tipo de validação de estrutura é utilizado regularmente, por exemplo, por aplicações que recebem mensagens via XML e desejam confirmar a integridade dos documentos recebidos avaliando se todo o conteúdo está de acordo com o domínio em que trabalha.

> "A definição de quais *tags* são válidas, bem como sua multiplicidade e opcionalidade podem ser definidas através de documentos XML *Schema Definition* (XSD)" (VALENTINE et al., 2002)

Estes arquivos XSD são correntemente o padrão mais utilizado para essa finalidade. Nele são descritos, com sua linguagem própria, qual a estrutura que um XML deve ter para ser considerado válido.

> "XSDs expressam um vocabulário comum e permite às máquinas seguir uma série de regras impostas por pessoas. Elas proveem uma forma de definir a estrutura, conteúdo e semântica de documentos XML." (W3C, 2016).

Na Figura 2.2 é apresentado um exemplo de documento XSD que valida a estrutura do XML da Figura 2.1.

Cada elemento do tipo `complexType` descreve um elemento que pode existir no XML a ser validado assim como os atributos que possui, sua obrigatoriedade e tipo de dado. Isso garante que um arquivo XML possa seguir um padrão estipulado apenas utilizando um arquivo XSD para validação.

2.2 *Scripts* XSLT

"XSLT, o *Extensible Stylesheet Language for Transformations*, é uma recomendação oficial do *World Wide Web Consortium* (W3C). Ele fornece de forma flexível, uma linguagem para transformar documentos XML em um documento HTML, outro documento XML, um arquivo *Portable Document Format*(PDF), um arquivo *Scalable Vector Graphics* (SVG), um arquivo *Virtual Reality Modeling Language* (VRML), código Java, um arquivo simples de texto, um arquivo JPEG, ou qualquer outro tipo de documento. Escreve-se uma folha de estilo XSLT para definir as regras para transformar um documento XML e o processador XSLT faz a transformação" (TIDWELL, 2008).

Figura 2.2 - XSD validador do exemplo de XML.

```
1 <?xml version="1.0"encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified" attributeFormDefault="unqualified
      ">
3     <xs:element name="BankAccount">
4         <xs:complexType>
5             <xs:sequence>
6                 <xs:element name="Number" type=="xs:integer"/>
7                 <xs:element name="Type" type=="xs:string"/>
8                 <xs:element name="OpenData" type=="xs:date"/> <
                      xs:element name="Balance" type=="xs:float"/> <
                      xs:element name="AccountHolder">
9                     <xs:complexType>
10                        <xs:sequence>
11                            <xs:element name="LastName" type==
                                  xs:string"/>
12                            <xs:element name="FirstName" type==
                                  xs:string"/>
13                        </xs:sequence>
14                    </xs:complexType>
15                </xs:element>
16            </xs:sequence>
17        </xs:complexType>
18    </xs:element>
19 </xs:schema>
```

Fonte: Produção do autor

Scripts XSLT são aplicáveis a um arquivo XML para alterar sua estrutura. Tem-se o exemplo da Figura 2.3, onde um XML foi alterado estruturalmente sem perda de dados. Isso pode ser feito manualmente por se tratar de uma pequena amostra de dados, porém em grandes volumes é impraticável. Existem casos em que os arquivos XML são gerados em tempo de execução e não é possível alterá-los manualmente, então nesses casos os *scripts* de conversão entre modelos auxiliam deste processo.

Para a geração desse tipo de *script* pode se fazer das seguintes formas: é possível criar o *script* manualmente, especificando cada alteração a ser aplicada ao XML alvo; ou alterar o XSD validador para a nova estrutura e posteriormente utilizar uma ferramenta para geração automática dos *scripts* de conversão entre o XSD original e o alterado.

Na Figura 2.4 é mostrado um trecho do *script* de conversão do XML da Figura 2.1. Ele executa da linha 13 à 22 a criação de um novo elemento chamado `FullName` que será a copia dos elementos `FirstName` e `LastName`. Esse tipo de cópia inclui também

Figura 2.3 - XML depois da refatoração.

```
1 <?xml version="1.0" standalone="yes"?>
2 <BankAccount>
3     <Number>1234</Number>
4     <Type>Checking</Type>
5     <OpenDate>11/04/1974</OpenDate>
6     <Balance>25382.20</Balance>
7     <AccountHolder>
8         <FullName>Singh Darshan</LastName>
9     </AccountHolder>
10 </BankAccount>
```

Fonte: Produção do autor

os atributos que os elementos possuem.

Utilizando essa técnica pode-se alterar a estrutura dos dados apenas com *scripts* XSLT sem alterar seu conteúdo. Para a aplicação dos *scripts* sobre os arquivos XML é necessário um processador da linguagem XSLT. Hoje no mercado pode-se encontrar algumas opções como, por exemplo: Saxon (2015).

2.3 Refatoração de XML

De acordo com Fowler e Beck (1999), refatoração é o processo de mudança de um sistema de *software* de tal maneira que não se altera o comportamento externo do código, porém melhora a sua estrutura interna. É uma maneira disciplinada para limpar código que minimiza as chances de introdução de *bugs*. Em essência, quando se refatora, está se melhorando o *design* do código depois de ter sido escrito.

Segundo Ambler e Sadalage (2006), a refatoração de banco de dados é uma simples mudança para um esquema de banco de dados que melhora a sua concepção, mantendo tanto a sua semântica comportamental quanto os dados. Um esquema de banco de dados inclui tanto aspectos estruturais, tais como definições de *tables* e *views*, como aspectos funcionais, tais como *procedures* e *triggers*. Uma coisa interessante a se notar é que uma refatoração de banco de dados é conceitualmente mais difícil do que uma refatoração de código; refatorações de código só precisam manter a semântica comportamental, enquanto refatorações de banco de dados também devem manter a denotação dos dados.

No caso da refatoração de XML, Salerno et al. (2010) define como uma mudança

Figura 2.4 - *Script* XSLT de refatoração.

```
1        <xsl:apply-templates />
2    </xsl:copy>
3 </xsl:template>
4    <xsl:template match="//element()[(namespace-uri(.)='')and (local-
         name(.)='AccountHolder')]">
5        <xsl:variable name="firstId" select="generate-id(./element()[
             (namespace-uro(.)='')and(local-name(.)='FirstName') or (
             namespace-uro(.)='')and(local-name(.)='LastName')][1])"/>
6        <xls:copy>
7            <xls:copy-of select="@*"/>
8            <xls:apply-templates>
9                <xsl:with-param name="id" select="\$firstId"/>
10           </xls:apply-templates>
11       </xls:copy>
12   </xsl:template>
13   <xsl:template match="//element()[(namespace-uri(.)='')and (local-
         name(.)='AccountHolder')]/element()[(namespace-uri(.)='')and (
         local-name(.)='Firstname') or (namespace-uri(.)='')and (local-
         name(.)='LastName')]">
14       <xsl:param name="id"/>
15       <xsl:if test="\$id= generate-id()">
16           <xsl:element name="FullName namespace="">
17               <xsl:copy-of select "../element()[(namespace-uro(.)='')
                     and(local-name(.)='FirstName') or (namespace-uro(.)=
                     '')and(local-name(.)='LastName')]"/>
18           </xsl:element>
19       </xsl:if>
20   </xsl:template>
21   <xsl:template match="//element()[(namespace-uri(.)='')and (local-
         name(.)='BankAccount')]/element()[(namespace-uri(.)='')and (
         local-name(.)='AccountHolder')]">
22       <xsl:variable name="firstId" select="generate-id(./element()
             [(namespace-uro(.)='')and(local-name(.)='FirstName') or (
             namespace-uro(.)='')and(local-name(.)='LastName')][1])">
23       <xsl:copy>
24           <xsl:copy-of select="@*"/>
25           <xls:apply-templates>
26
27               <xsl:with-param name="id" select="\$firstId"/>
28
29           </xls:apply-templates>
30
31       </xls:copy>
32   </xsl:template>
```

Fonte: Produção do autor

estrutural dentro de um documento que preserva o significado da informação armazenada. Em outras palavras, a informação recebida pela aplicação consumidora não muda.

Essa alteração deve ser apenas estrutural como agrupamento de elementos, alteração do nome de um atributo, alteração de um tipo de dado, criação de uma nova entidade ou atributo, etc. Pode ser feita através do XSD que descreve o formato do documento XML. A Figura 2.2 apresentou a estrutura que um documento XML deve seguir para ser considerado válido, já a Figura 2.5 apresenta uma nova estrutura para este documento e que deve ser aplicado aos XMLs que estavam em conformidade com a estrutura antiga.

Figura 2.5 - Modelo depois da refatoração.

```
<?xml version="1.0"encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified
    ">
    <xs:element name="BankAccount">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Number" type=="xs:integer"/>
                <xs:element name="Type" type=="xs:string"/>
                <xs:element name="OpenData" type=="xs:date"/> <
                    xs:element name="Balance" type=="xs:float"/> <
                    xs:element name="AccountHolder">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="FullName" type==""
                                xs:string"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Fonte: Produção do autor

Nesta nova estrutura houve um agrupamento de duas entidades, `FirstName` e `LastName`, que foi mostrado na Figura 2.1 e que agora chama-se `FullName`. Nenhuma informação foi perdida ou sofreu alteração em sua semântica, apenas uma nova estrutura foi gerada. Com esta nova estrutura todos os arquivos XML que foram produzidos seguindo a estrutura inicial não são mais considerados válidos, pois não possuem a entidade `FullName`. Isto é um problema quando se tem aplicações que consomem dados via XML de serviços *web* pois se o fornecedor do serviço altera a estrutura dos XMLs enviados a aplicação consumidora não os entende mais

causando problemas ou inviabilizando a comunicação entre as aplicações.

Para que um documento XML seja novamente considerado válido, ele deve se adequar a nova estrutura descrita no arquivo XSD. A Figura 2.3 apresenta um arquivo XML com os mesmo dados do XML anterior, porém seguindo a nova estrutura descrita no XSD.

Para a refatoração de um documento XML é necessário algum tipo de conversor que saiba tratar os dados e os transformar pra um outro formato qualquer. Assim, segundo Tidwell (2008), a linguagem XSLT foi criada para prover flexibilidade na transformação de documentos XML. Esta linguagem possibilita a criação de *scripts* que são aplicados ao documento XML para alterar sua estrutura.

2.4 Ferramenta Chrysalis

Para o processo de criação de *scripts* XSLT que refatoram a estrutura de um documento XML utilizou-se a ferramenta Chrysalis (DUARTE, 2010) que possui uma série de funcionalidades que auxiliam nessa tarefa. Para cada alteração feita num documento XSD ela gera *scripts* XSLT de conversão de dados para serem aplicados aos documentos XML para que possam refletir a nova estrutura válida.

O Chrysalis conta com um conjunto de tipos de refatorações implementadas para que após a alteração do documento XSD possa gerar os *scripts* de conversão. A seguir têm-se os tipos de refatorações disponíveis na ferramenta Chrysalis:

- Adicionar Elemento: Adiciona um novo elemento a estrutura do arquivo.
- Remover Elemento: Remove um elemento existente na estrutura do arquivo, removendo também todas as referências a ele no restante do documento.
- Renomear Elemento: faz a troca do nome de um elemento por outro nome à escolha do usuário, substituindo também todas as suas referências à esse elemento no restante do documento.
- Renomear Atributo: faz a troca do nome de um atributo por outro nome à escolha do usuário, substituindo também todas as suas referências a esse atributo no restante do documento.
- Agrupar Elementos: consiste em unir um conjunto de elementos já existentes, transformando-os em um único elemento nomeado à escolha do

usuário.

- Desagrupar Elementos: consiste em separar um conjunto de elementos filhos de outro elemento já existente, removendo cada um destes elementos filho do pai e adicionando-os ao mesmo nível de hierarquia do pai.

- Atributo em Elemento: visa promover um atributo já existente de um elemento a um elemento em si.

- Elemento em Atributo: visa promover um elemento já existente a um atributo de outro elemento existente.

- Elemento em Referência: visa transformar um elemento já existente em uma entidade global e alterar quem o utilizava, passando a utilizar apenas sua referência.

- Referência em Elemento: visa transformar uma referência a um elemento global em um elemento em si.

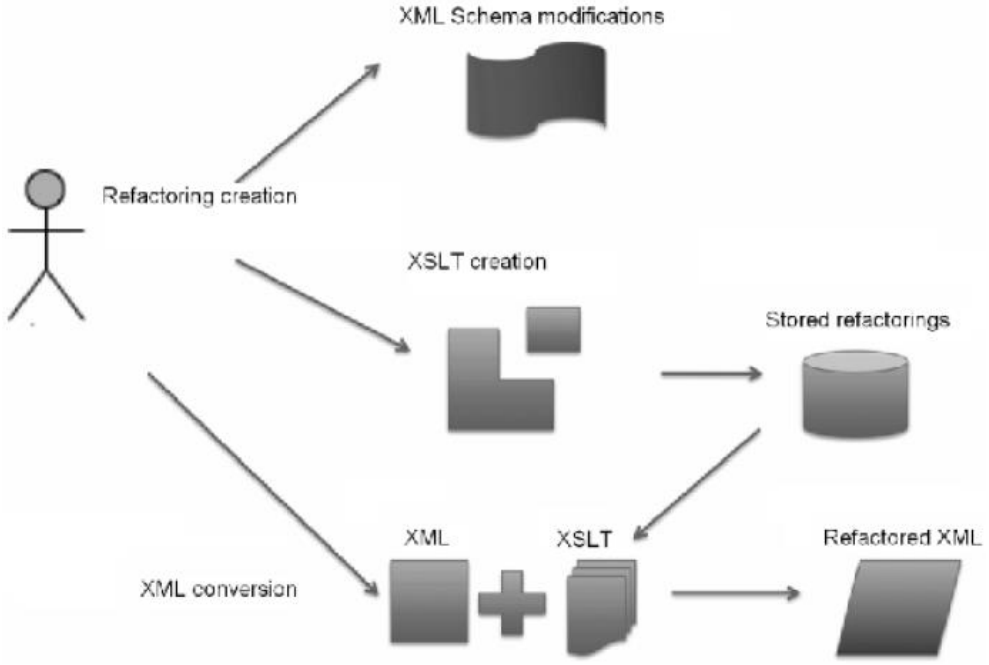
- Mover Elemento: visa alterar o elemento pai do elemento a ser movido, passando como filho a outro elemento já existente.

Dentre as opções de refatoração da ferramenta Chrysalis, existem dois tipos de mudanças: a que altera estruturalmente os dados que é o foco deste trabalho e as alterações que podem mudar a semântica do XML, o que inviabiliza a conversão de uma estrutura para outra. Por exemplo, no caso do agrupamento de entidades exemplificado anteriormente, os dados abrigados pela entidade `FullName` continuam os mesmos, não perdendo seu sentido, como foi explicitado.

Já no caso de uma alteração semântica, como é o caso da refatoração que move um elemento de posição dentro da hierarquia de entidades, essa alteração pode causar uma mundaça no sentido do dado existente no documento. No exemplo da Figura 2.1, se a a entidade `Number` for movida para dentro da entidade `AccountHolder`, este valor numérico perde o sentido na estrutura, pois este valor não tem significado para os dados do titular da conta, mas sim para a conta. Pode ocorrer uma alteração semântica mesmo num agrupamento de entidade, por exemplo, caso sejam agrupados os valores das entidades `Number` e `Type`, estes perdem o sentido. Logo, é necessário saber que um mesmo tipo de alteração, dependendo do seu contexto, pode causar uma alteração estrutural ou semântica nos dados.

É possível alterar o documento XSD através da ferramenta para uma nova estrutura e ainda gerar os *scripts* de conversão dos XML para essas mudanças. Essa geração de *scripts* é bidirecional, isto é, são gerados *scripts* de conversão de dados da antiga estrutura para a nova como da nova estrutura para a antiga. Na atual versão de implementação do Chrysalis, os arquivo de *script* são salvos localmente ficando a cargo do usuário o gerenciamento deles. A Figura 2.6 descreve o modelo de refatoração utilizado pela ferramenta Chrysalis.

Figura 2.6 - Modelo de refatoração da ferramenta Chrysalis.



Fonte: Castro (2012)

Após a confirmação das alterações efetuadas no documento XSD a ferramenta cria um diretório local com os *scripts* gerados. A desvantagem é que esses diretórios são locais e devem ser distribuídos manualmente para quem queira os aplicar sobre seus documentos XML. Para cada nova versão gerada de um documento XSD todos os *scripts* devem ser enviados para as aplicações que desejam utilizá-los. Isto pode se tornar inviável uma vez que se pode não saber quais aplicações necessitam de destes *scripts*.

2.5 Serviços *web*

De acordo com a *Wide Web Consortium* (W3C), um serviço *web* pode ser entendido como um sistema de *software* projetado para suportar interações máquina-máquina através de uma rede. Ele possui uma interface descrita em um formato que é processável por outras máquinas que é especificado em um padrão chamado WSDL (*Web Services Description Language*). Outros sistemas devem interagir com o *web service* da maneira prescrita na descrição do serviço.

"Um serviço *web* fazendo o recolhimento de informações pode consistir em vários componentes de código, cada um hospedado em um servidor separado; e o serviço *web* pode ser consumido em PCs e outros dispositivos" (KALIN, 2013). Isso mostra que um serviço *web* deve estar disponível a consumidores sem se importar quem são eles, da mesma forma que o fornecedor do serviço deve prover serviços aos consumidores sem que eles necessitem saber como o serviço é processado.

"A orientação a serviços é um paradigma arquitetural, onde uma aplicação é desenhada e desenvolvida para oferecer serviços sem se importar quem o consumirá, seguindo o conceito de solicitação e resposta." (ERL, 2008).

Existem muitas tecnologias para a criação, publicação, fornecimento e consumo de serviços. Cada tipo de tecnologia provê características peculiares e dá ao arquiteto de *software* opções que melhor se encaixam às necessidades da aplicação. Pode-se citar como tecnologias de orientação a serviços: *Web Services* (KALIN, 2013), CORBA (*Common Object Request Broker Architecture*) (OTTE et al., 1995), OSGi (*Open Services Gateway Initiative*) (HALL et al., 2011) e *Java Beans* (ROMAN et al., 2005).

2.6 SOAP, REST e WSDL

Entre os protocolos de comunicação para serviços *web* destacam-se o *Simple Object Access Protocol*(SOAP) e *Representational State Transfer*(REST), cada um seguindo uma abordagem diferente, sendo utilizados em grande parte das aplicações e envolvendo a maioria das discussões a respeito das arquiteturas desse tipo de comunicação.

O protocolo SOAP, baseado na troca de informações através de mensagem utilizando XML, é uma recomendação da W3C para a criação de serviços *web*. As mensagens XML utilizadas pelo protocolo SOAP possuem três partes: o envelope que identifica o documento XML e a mensagem SOAP, um cabeçalho que contém informações relacionadas ao aplicativo que vai processa-la e por fim tem-se a terceira parte,

o corpo da mensagem que contém as informações a serem transmitidas em sí. Os elementos do corpo da mensagem devem ser definidos pelo serviço *web* que o fornece, indicando quais elementos devem existir na mensagem, sua ordem, multiplicidade, etc.

Juntamente com as definições de mensagem, o protocolo SOAP também descreve os padrões do serviço implementado, dizendo quais as especificações de *endpoint* que devem ser seguidas e que normalmente são descritas através de um arquivo WSDL. O padrão WSDL possui um conjunto de elementos que descrevem as características de um serviço *web*, especificando dentro de um contêiner de serviço a porta utilizada, protocolo, o nome das operações que este serviço possui e a estrutura de mensagem que a operação espera receber e qual ela deve retornar. A estrutura da mensagem contém elementos que são descritos no formato XSD e que são vinculados ao WSDL do serviço. O contrato de mensagem a que este trabalho faz referência é justamente este arquivo XSD que o WSDL utiliza para criar a estrutura das mensagens de envio e recebimento de informações pelas operações.

Já o protocolo REST segue uma outra linha arquitetural onde não é necessária uma descrição formal da interface de acesso ao serviço pois ele usa sempre a mesma interface, utilizando toda a estrutura do protocolo HTTP (métodos *GET*, *POST*, *PUT* e *DELETE*). Este protocolo não precisa utilizar documentos XML para criar a requisição e a resposta, podendo gerar dados em vários formatos como *Command Separated Value* (CSV), *JavaScript Object Notation* (JSON) e *Really Simple Syndication* (RSS). Com este conceito pode-se utilizar um serviço arquitetado com o protocolo REST apenas enviando um comando do tipo PUT e receber como resposta os dados no formato JSON, por exemplo, muito utilizado hoje em dia.

O protocolo REST obtém melhores resultados em relação à flexibilidade e ao controle, mas requer codificação de baixo nível. WS oferece melhor suporte a ferramentas e a conveniência de uma interface de programação, mas introduz uma dependência a fornecedores e projetos de código-fonte aberto. (PAUTASSO et al., 2008). A decisão de qual protocolo utilizar deve estar de acordo com a necessidade de cada serviço criado e qual protocolo melhor lhe atende. A seguir tem-se uma comparação entre os protocolos SOAP e REST:

SOAP: apesar de ser uma opção mais pesada que o REST, possui linguagem, plataforma e transporte independente, isto é, não necessita do uso do protocolo HTTP como base. Segue uma padronização das mensagens previamente definida informando a estrutura das requisições e das respostas. Funciona muito bem em

ambientes corporativos distribuídos.

REST: um protocolo mais leve que o SOAP, possui vantagens como sua flexibilidade, não definindo os formatos das mensagens, muito eficiente pois não necessita sempre utilizar XML para a troca informações, podendo utilizar outros formatos que atendem ao necessitado com um tamanho menor de mensagem e utiliza o protocolo HTTP como base sendo amplamente conhecido e utilizado, além de não necessitar de protocolos adicionais para a comunicação.

2.7 Arquitetura SOA

Service Oriented Architecture (SOA) é uma arquitetura de referência baseada na orientação a serviços onde o princípio é que toda funcionalidade é oferecida em forma de serviço. Esse tipo de arquitetura pode abranger um conjunto de tecnologias e ferramentas para disponibilizar seus serviços concatenando muitos tipos de serviços e tornando-se muito flexível.

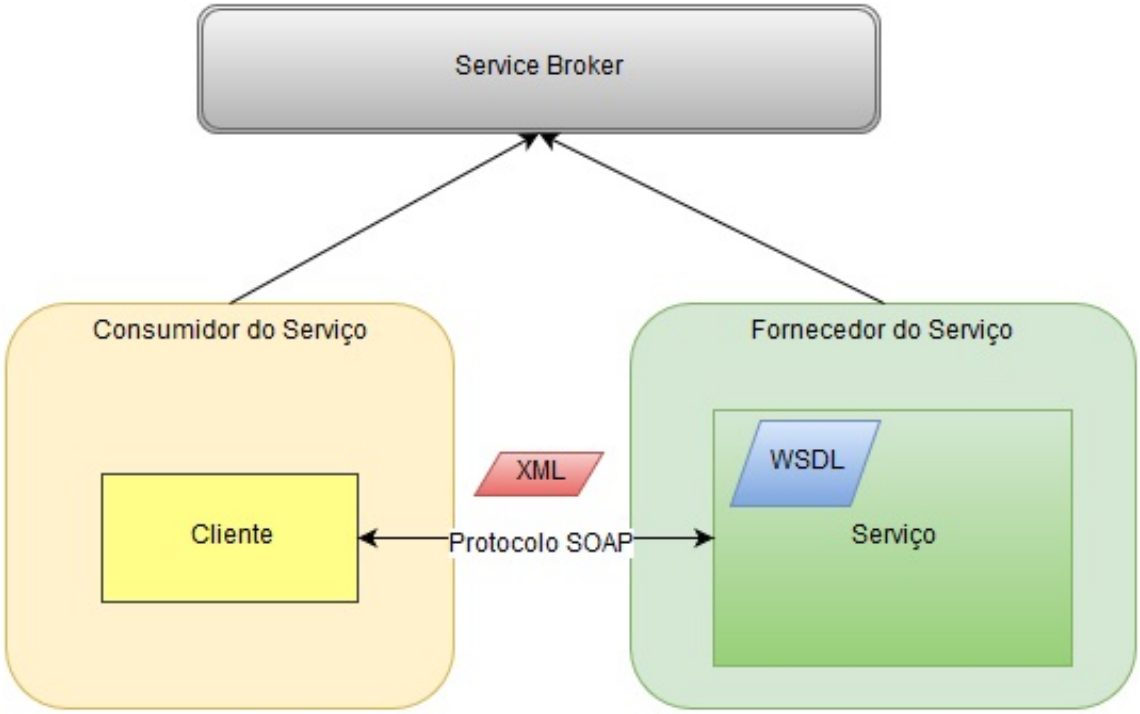
Segundo Erl (2008), "Arquitetura orientada a serviços representa um modelo de arquitetura que visa melhorar a agilidade e custo-efetividade de uma empresa, reduzindo a carga de TI em toda a organização".

Na Figura 2.7 é descrita a arquitetura SOA clássica com seus componentes e a forma de troca de mensagens. Ele possui um componente chamado Fornecedor do Serviço que representa o fornecedor do serviço disponibilizado. É ele quem executa as funções ou ações do serviço. Seus serviços são formalizados através de um arquivo WSDL que contém a interface do serviço, quais operações estão disponíveis, endereços de *endPoint* (endereços na rede) e a forma de consumi-los. Esse documento descreve o contrato do serviço.

Com a publicação dos serviços através do WSDL os consumidores podem utilizar as operações providas pelo fornecedor, se adequando aos parâmetros para consumo descritos em tal documento. Os dados transmitidos pelas operações devem seguir um contrato de mensagem que é descrito em um documento XSD específico que faz parte do contrato definido no WSDL.

O componente Consumidor do serviço da Figura 2.7 representa os consumidores do serviço disponibilizado pelo Fornecedor do Serviço. Através do protocolo SOAP é possível transmitir mensagens entre fornecedores e consumidores utilizando documentos XML.

Figura 2.7 - Arquitetura SOA



Fonte: produção do autor

O *Service Broker* representa o componente responsável por dizer aos consumidores do serviço onde encontrar o contrato descritor da interface do serviço, o WSDL. UDDI (*Universal Description, Discovery and Integration*) representa o padrão de publicação e descobrimento de diretórios de serviços comumente utilizada para publicar serviços *web*.

Esse padrão para registro, publicação e listagens de serviços para os usuários foi concebido como parte da arquitetura de serviços, porém hoje em dia não é mais utilizado, mostrando que existem diferenças da utilização do SOAP como ele foi concebido e como é realmente utilizado.

Extol International (2009) comenta que este padrão foi descontinuado pela Microsoft a partir do Sistema Operacional Windows Server 8 R2 com o encerramento do projeto UDDI Business Registry (UBR) nos anos 2000 onde participavam Microsoft, IBM e SAP.

Uma discussão sobre isso pode também ser encontrado em InnoQ (2010) que atenta

para o fato do uso de outras tecnologias que suplantam sua utilização. Deixando de lado que alguém poderia ter feito a mesma coisa usando um simples protocolo REST e Paul Prescod mostrou como fazer isso em 2002, UDDI pode ser usado para algo útil, porém deve ser personalizado de uma forma não interoperável." (INNOQ, 2010)

Outra discussão a respeito da utilização do UDDI é visto em InfoWorld (2005) que apresenta outras formas mais eficientes de descobrimento de serviços como, por exemplo, LDAP. De fato, UDDI é um bom diretório de serviços, como um modelo. E caras como sistnet estão fazendo um bom negócio vendendo UDDI dentro de empresas. No entanto, serviços de diretório como Active e LDAP não mais eficazes, embora não sejam especificamente para SOA. (INFOWORLD, 2005)

Este modelo de arquitetura SOA apresentado na Figura 2.7, tida como a clássica, pode sofrer variações dependendo da tecnologia ou padrão utilizados. Hoje é comum a utilização de uma aplicação intermediadora para as aplicações que se comunicam fornecendo um ponto central para a troca de todas as informações entre as aplicações e serviços. Essa arquitetura será apresentada na seção 2.9 deste capítulo.

2.8 SOA *patterns*

"Com a difusão do *design* SOA, muitas soluções de problemas recorrentes foram utilizadas naturalmente e documentadas individualmente em cada projeto. Assim, a esses padrões comuns de construção, publicação e fornecimento de serviços deu-se o nome de SOA *Patterns*". (ERL, 2008).

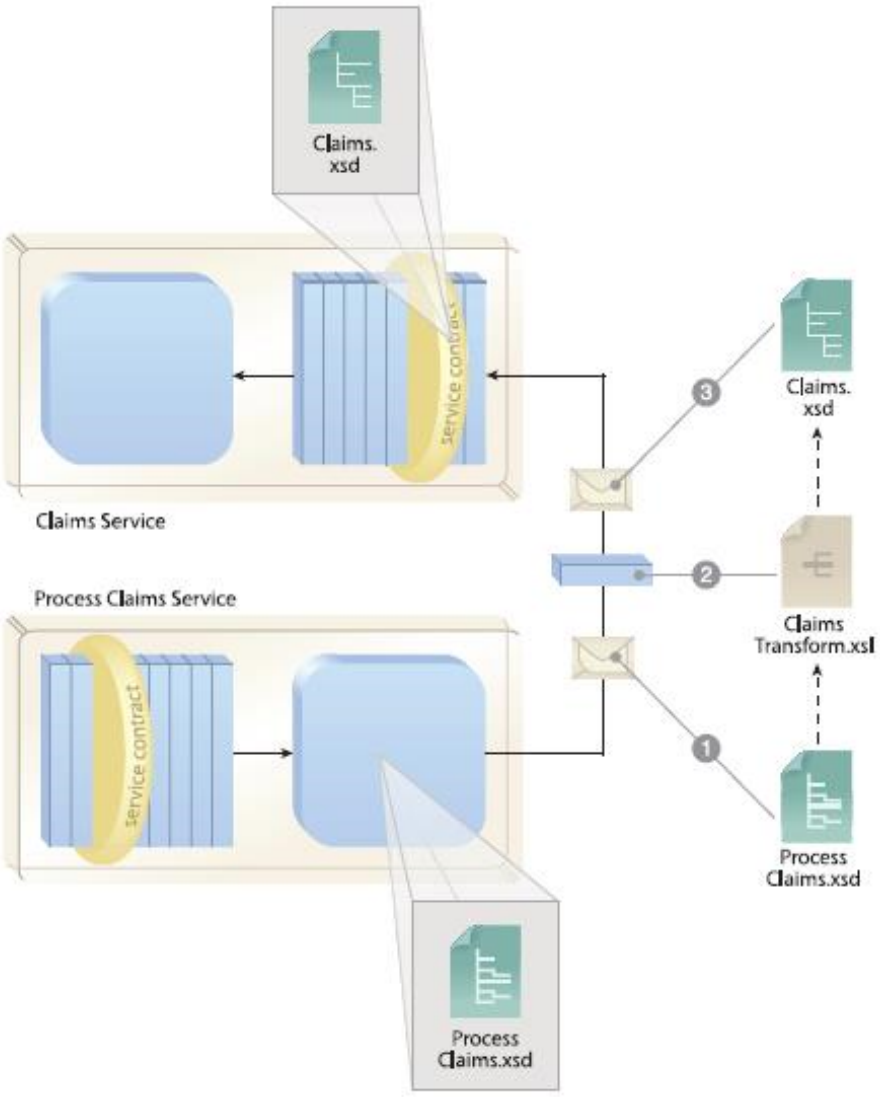
Existe um conjunto de padrões desenvolvidos para auxiliar na modelagem dos serviços chamado de SOA *Patters*. São classificados em 17 grupos de acordo com o tipo de problema que se propõem a solucionar. Por exemplo, existe um grupo especifico de padrões que se destina a tratar de transformações de arquivos, o grupo *Transformation*. Outro grupo é o *Service Security* que une padrões a respeito de como tratar problemas de segurança nos serviços.

Sobre o padrão *Data Model Transformation*, que foi reunido ao grupo *Transformation*, podemos dizer que é aplicável quando se tem um problema de serviços usando modelos diferentes para representar os mesmos tipos de dados.

Na Figura 2.8 está ilustrado esse padrão: um serviço chamado *Claim Process*, que utiliza um modelo de XML chamado *ProcessClaim.xsd*, envia uma mensagem ao receptor que espera receber mensagens no modelo *Claim.xsd*. Para isso o padrão

Data Model Trasformation aplica um *script* XSLT de transformação de dados entre o envio e a recepção marcado com o número 2 na Figura 2.8.

Figura 2.8 - *Data Model Transformation.*



Fonte: Erl (2008)

Erl (2008) comenta que: quando os serviços são implementados como serviços *web*, XSLT é geralmente usado para definir a lógica de mapeamento que é posteriormente executada para realizar a transformação em tempo de execução. Na verdade, o uso de *scripts* XSLT representa a aplicação mais comum desse padrão, que descreve a estrutura para a aplicação de transformações baseadas em XSLT, porém a forma como essas transformações serão efetuadas ou como será o gerenciamento dessas

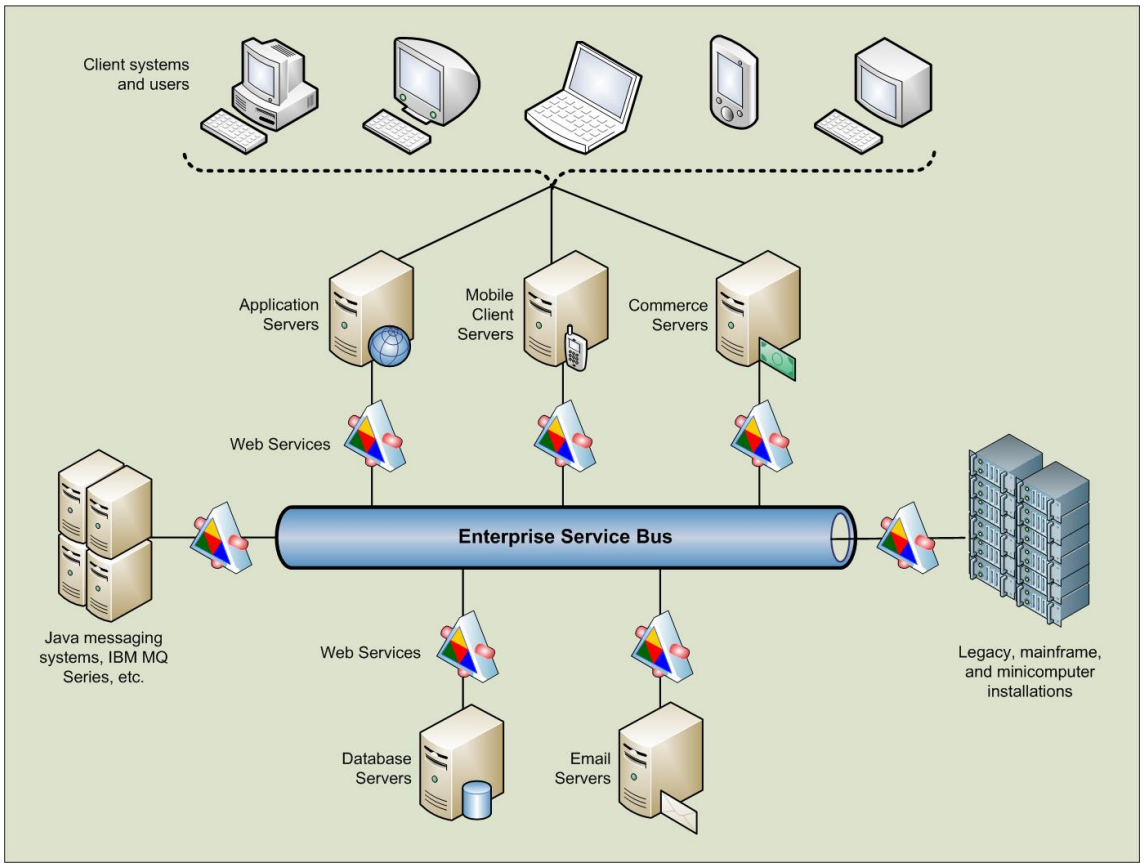
versões não faz parte do padrão apresentado.

2.9 *Enterprise Service Bus*

"Um *Enterprise Service Bus* (ESB) é uma plataforma de integração baseada em padrões que combina mensagem, serviços *web*, transformação de dados e roteamento inteligente para se conectar de forma confiável e coordenar a interação de um número significativo de diversas aplicações em empresas estendidas com integridade transacional." (SCHULTE, 2002)

Essa plataforma age como um *middleware* entre as aplicações que desejam se comunicar, resolvendo os endereços para onde as mensagens devem ser encaminhadas, além de possuir uma infraestrutura para agir sobre as mensagens podendo ser aplicado algumas regras ou ações.

Figura 2.9 - Arquitetura ESB.



Fonte: Delfim (2009)

A Figura 2.9 descreve a arquitetura de funcionamento de um ESB com as aplicações que trocam mensagens. Ela funciona como um canal de comunicação entre todas as aplicações destinando as mensagens recebidas de acordo com o destinatário, ficando transparente às aplicações como essas mensagens são entregues. Cada aplicação envia diretamente suas mensagens ao ESB que se encarrega de enviar ao destinatário. Isso traz a vantagem de uma aplicação que consome vários serviços *web*, por exemplo, conhecer apenas um endereço de comunicação, desacoplando fornecedores e consumidores de serviços.

A função do ESB como *Message Routers* é encaminhar solicitações para serviços e o translado das respostas de volta para os clientes. A intenção é proporcionar aos clientes um meio de chamar os serviços, minimizando dependências do cliente em implementações de serviços específicos. ESBs fornecem uma camada que permite que os serviços sejam adicionados, atualizados, substituídos, ou removidos, minimizando o impacto sobre aplicativos clientes. Isto é possível porque os clientes enviam mensagens para o barramento em vez de comunicar diretamente com os serviços de destino.

"Em outras palavras, as dependências do cliente mudam para o barramento. Os clientes frequentemente enviam mensagens através de *Virtual Services* [*Patterns* IBM, ESB]. Estes se parecem com os serviços de destino reais para os clientes, mas simplesmente fornecer um *endpoint* comum que recebe mensagens. Os clientes também podem usar *Channel Adapters* para ligar para o barramento. ESBs roteiam mensagens para o serviço adequado, de acordo com regras pré-definidas. Há muitas maneiras de construir estas regras. Administradores ESB podem, por exemplo, usar *SOAPAction* ou cabeçalhos *WS-Addressing*, modelos URI, ou conteúdo encontrado no corpo das mensagens." (DAIGNEAU, 2011).

Além das funções já descritas que são executadas por um ESB, ele possui funções de garantia de entrega de mensagens, que mesmo que uma aplicação destinatária esteja incomunicável no momento o ESB tenta enviar novamente a mensagem até que a aplicação receba quando estiver novamente comunicável. Há funções de *log* ou integração com ferramentas de monitoramento, aplicação de politicas e autenticação.

2.10 Versionamento de serviços

A disponibilização de funcionalidades em forma de serviços *web* traz consigo a mesma necessidade de qualquer outro tipo de *software* que é sua evolução, seja para se adequar a novas tecnologias presentes no mercado, seja para se adequar a alterações

nas informações tramitadas na comunicação. O fato é que evoluções em serviços *web* quase sempre alteram o contrato do serviço fazendo com que existam múltiplas versões de um mesmo serviços para atender diferentes consumidores. Esta multiplicidade de versões do mesmo serviço deve ser transparente às aplicações que as consomem pois cada aplicação consumidora necessita apenas da versão do serviço que lhe atende e não precisa conhecer as demais versões.

Versionamento de serviços é a sobrecarga de um serviço, isto é, quando existe mais de uma versão de um serviço e elas estão disponíveis para consumo, dependendo do contexto ou de como ele é requisitado. Podem existir diferentes interfaces que utilizam a mesma implementação fazendo com que o serviço possua várias formas de ser acessado e existe o caso em que cada interface aponta para uma implementação diferente. Dependendo do tipo de arquitetura adotada uma mesma interface pode apontar para diferentes implementações de acordo com algum parâmetro passado na mensagem do requisitante.

Por um lado, os provedores de serviço querem fornecer várias versões em paralelo, oferecendo variantes específicas para alguns clientes ou versões de serviço mais antigas para aplicativos legados. Por outro lado, alguns solicitadores de serviço podem querer acessar diferentes versões de serviço de uma maneira uniforme ou até mesmo alternar entre eles em tempo de execução, enquanto outros não querem tratar explicitamente diferentes versões de serviço. (LEITNER et al., 2008). Isto mostra que muitas vezes solicitadores não só podem, mas precisam de várias versões de um serviço depedendo do contexto a ser utilizado.

No contexto SOA, uma alteração em um serviço é a evolução do WSDL fazendo com que a interface de acesso ao serviço possa mudar. Existem vários tipos de mudanças que um WSDL pode sofrer acarretando ou não na alteração de sua interface de acesso e que consequentemente altera a forma com que o consumidor do serviço o utilize. Dependendo do tipo de alteração a comunicação entre consumidores e fornecedor é quebrada. Nas Listagens 2.10 e 2.11 são mostradas um exemplo de alteração de um WSDL que altera a interface de acesso para o consumidor. Na linha 8 das listagens é possível ver que o nome da operação foi alterado de `AccountDetails` para `AccountInformation`, fazendo com que a interface de acesso para o consumidor mude quebrando a comunicação entre as aplicações.

Figura 2.10 - WSDL com a operação AccountDetails.

```
<wsdl:message name="AccountDetailsRequest">
    <wsdl:part element="schema:AccountDetailsRequest" name="
        AccountDetailsRequest"/>
  </wsdl:message>
  <wsdl:message name="AccountDetailsResponse">
    <wsdl:part element="schema:AccountDetailsResponse" name="
        AccountDetailsResponse"/>
  </wsdl:message>
  <wsdl:portType name="accountService">
    <wsdl:operation name="AccountDetails">
      <wsdl:input message="schema:AccountDetailsRequest" name="
          MemberDetailsRequest"/>
      <wsdl:output message="schema:AccountDetailsResponse" name=
          "MemberDetailsResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="accountserviceBinding"
    type="schema:accountService">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="AccountDetails">
      <soap:operation soapAction="" />
      <wsdl:input name="MemberDetailsRequest">
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="MemberDetailsResponse">
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="AccountService">
    <wsdl:port binding="schema:accountserviceBinding" name="
        accountServicePort">
      <soap:address location="http://localhost:8080/
          accountservice/services"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Fonte: Produção do autor

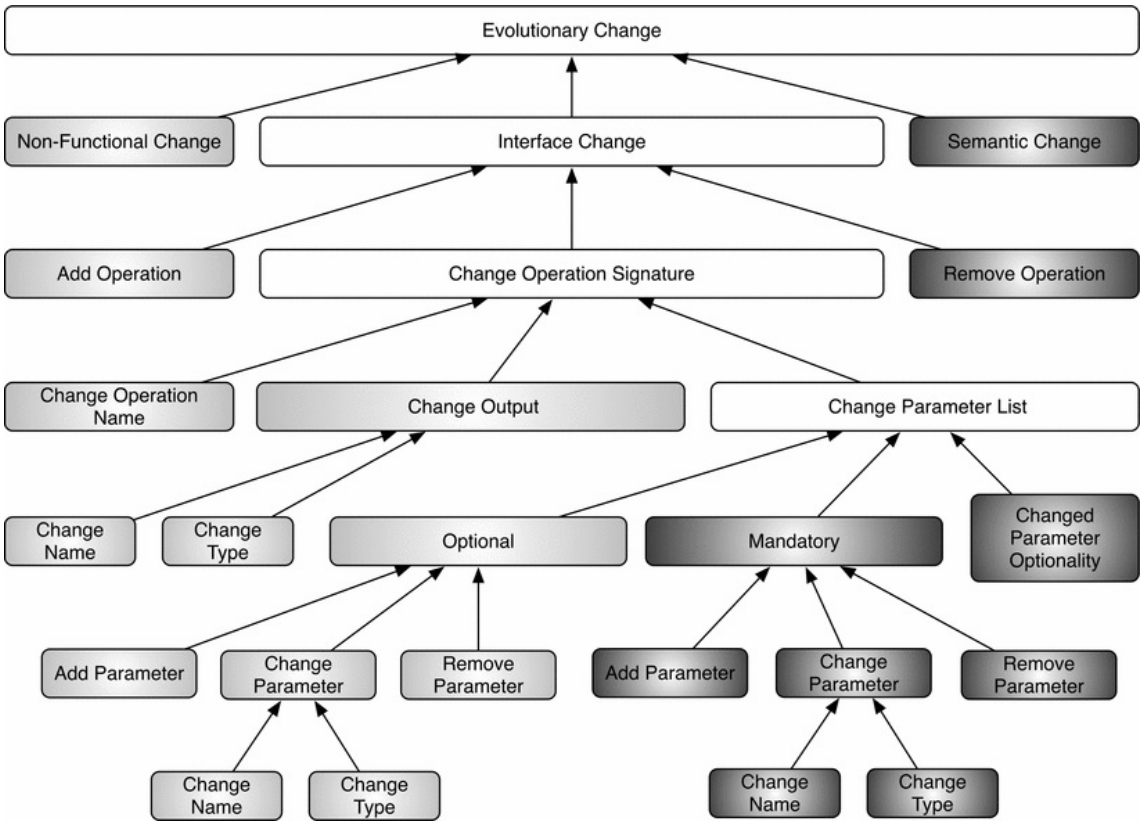
Figura 2.11 - WSDL com a operação AccountInformation.

```
1  <wsdl:message name="AccountInformationRequest">
2      <wsdl:part element="schema:AccountInformationRequest" name="
           AccountInformationRequest"/>
3    </wsdl:message>
4    <wsdl:message name="AccountInformationResponse">
5      <wsdl:part element="schema:AccountInformationResponse" name="
           AccountInformationResponse"/>
6    </wsdl:message>
7    <wsdl:portType name="accountService">
8      <wsdl:operation name="AccountInformation">
9        <wsdl:input message="schema:AccountInformationRequest"
             name="MemberDetailsRequest"/>
10       <wsdl:output message="schema:AccountInformationResponse"
             name="MemberDetailsResponse"/>
11     </wsdl:operation>
12   </wsdl:portType>
13   <wsdl:binding name="accountserviceBinding"
14       type="schema:accountService">
15     <soap:binding style="document"
16       transport="http://schemas.xmlsoap.org/soap/http" />
17     <wsdl:operation name="AccountInformation">
18       <soap:operation soapAction="" />
19       <wsdl:input name="MemberDetailsRequest">
20         <soap:body use="literal" />
21       </wsdl:input>
22       <wsdl:output name="MemberDetailsResponse">
23         <soap:body use="literal" />
24       </wsdl:output>
25     </wsdl:operation>
26   </wsdl:binding>
27   <wsdl:service name="AccountService">
28     <wsdl:port binding="schema:accountserviceBinding" name="
           accountServicePort">
29       <soap:address location="http://localhost:8080/
             accountservice/services"/>
30     </wsdl:port>
31   </wsdl:service>
32 </wsdl:definitions>
```

Fonte: Produção do autor

A Figura 2.12 proposta por Leitner et al. (2008) descreve os tipos de alterações que se pode encontrar na evolução de um contrato de serviço. Nota-se que nem todas as alterações em um serviço *web* causam uma alteração na interface para os consumidores como é o caso das alterações não funcionais. Para todos os outros casos o contrato de serviço entre fornecedor e consumidores é alterado gerando uma nova versão do serviço.

Figura 2.12 - Esquema de classificação de alteração de versão.



Fonte: Leitner et al. (2008)

Para se tentar solucionar este problema, diversos aspecto já foram abordados buscando ao máximo diminuir o impacto sobre as aplicações consumidoras ou sobre a complexa gerência das múltiplas versões de um mesmo serviço. Quando não há mais compatibilidade entre a interface de acesso utilizada por um consumidor e a implementada pelo fornecedor há a quebra de comunicação e para tanto muitas soluções já foram propostas como, por exemplo, Leitner et al. (2008), que utiliza um *Service proxy* que recebe as requisições dos consumidores e os relaciona a alguma versão do serviço disponível. Essa abordagem requer que todas as versões do serviço estejam funcionando e acessíveis e que também o *proxy* seja atualizado toda vez que uma nova versão do serviço for criada, que ele chama de *dynamic rebinding*.

Esta mesma abordagem é também apresentada por Frank et al. (2008) que introduz um novo componente chamado *service interface proxy* que faz a intermediação entre os clientes e as diferentes implementações do serviço. A cada nova implementação de um mesmo serviço o WSDL é clonado e sofre alteração apenas no *service endpoint*

para se diferenciar das outras versões.

Uma solução comumente utilizada para evitar alterar os cosumidores é hospedar múltiplas versões de um serviço que, todavia, é menos ideal porque exige mais recursos para manter todas ass versões do mesmo serviço disponíveis com *namespaces* separados. (CHIPONGA et al., 2014)

Chiponga et al. (2014) numa outra abordagem utiliza um ESB para tornar transparente aos consumidores as várias versões de um serviço. A mensagem enviada do consumidor passa pelo ESB e é transformada para a versão esperada pelo fornecedor do serviço por um componente interno. Após a transformação ela é enviada ao fornecedor do serviço que irá responder a requisição.

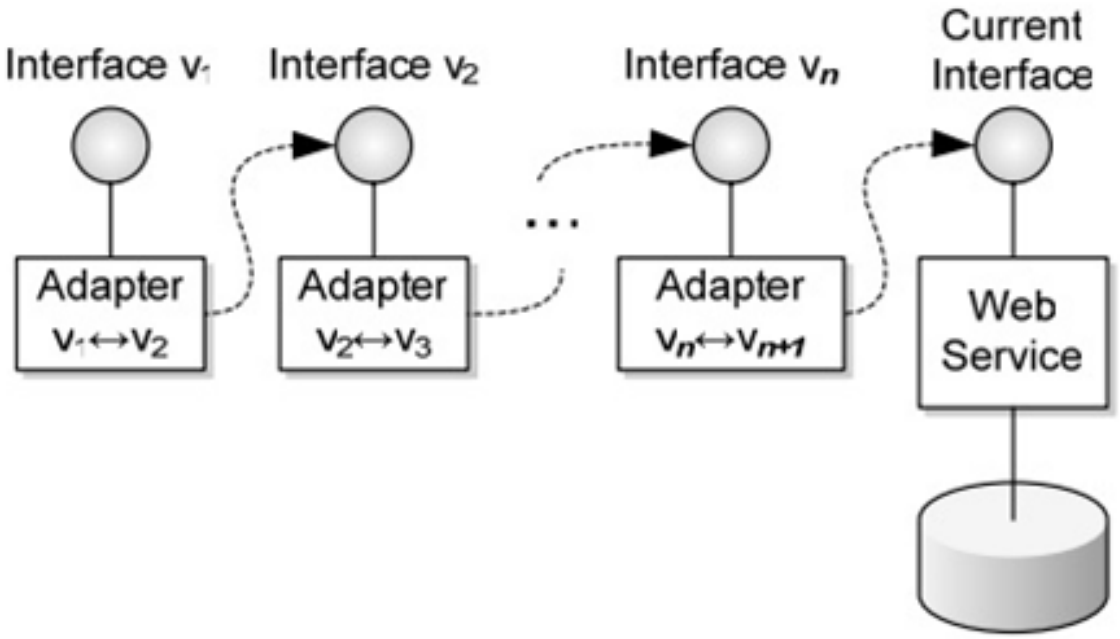
3 TRABALHOS RELACIONADOS

Esta seção apresenta alguns trabalhos relacionados ao problema a ser solucionado e como seus autores popuseram alternativas para a manutenção da interoperabilidade de serviços *web* mesmo com o versionamento do serviço e a geração de novas interfaces de acesso.

3.1 Chain of adapter

Um trabalho encontrado na literatura é proposto por Kaminski et al. (2006) que tenta resolver o problema da interoperabilidade de versões de serviços *web* criando um conjunto de interfaces interligadas, formando uma corrente de adaptações. Esta proposta de solução mantém uma interface ativa para cada versão do serviço existente e se comunicam até chegar a versão de interface realmente implementada. Na Figura 3.1 é mostrada a proposta chamada de *Chain of Adapter*.

Figura 3.1 - Chain of Adapter



Fonte: Kaminski et al. (2006)

Quando um consumidor faz uma requisição, ele envia uma mensagem ao fornecedor do serviço através da interface que conhece, mas que não necessariamente é a interface que possui a implementação do serviço. Caso a interface acessada não seja

a que possui a implementação, a solução proposta faz as alterações na mensagem para o formato da interface adjascente e assim por diante até que chegue na última e o fornecedor do serviço receba a mensagem no formato esperado. Neste trabalho as interfaces são chamadas de *Adapters* e a versão de interface que possui a implementação do serviço é chamada de *Current Interface*.

Para cada nova versão criada do serviço quandoa a interface é alterada é necessário adicionar a forma com que a mensagem deve ser transformada para a nova versão, seja em *hard code* ou através de *scripts*. O consumidor não necessita saber que existe uma nova versão implementada, pois essa transformação é feita de forma transparente a ele. Neste tipo de solução, quando um serviço possui muitas alterações é necessário manter sempre todas as interfaces ativas podendo gerar muito processamento de transformações já que uma mensagem não é transformada diretamente para a vesão final e sim de uma versão adjascente para outra. Um ganho com essa solução é a retrocompatibilidade com versões anteriores e sem duplicação de código para várias versões.

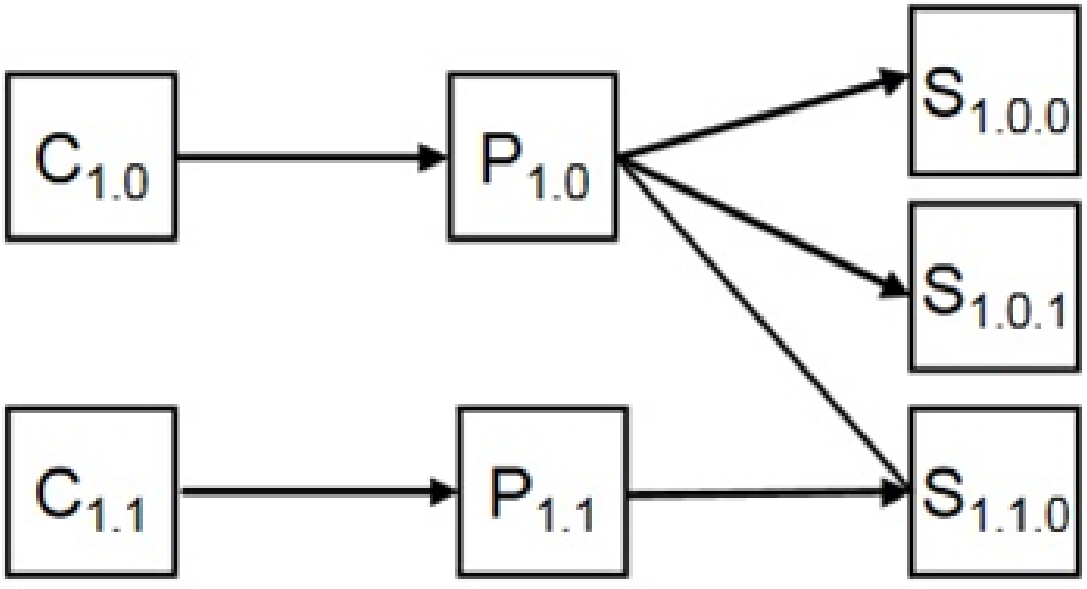
3.2 Proxy Interface

Outro trabalho encontrado na literatura e proposto por Frank et al. (2008) trata de uma outra forma o versionamento de serviços. Chamado de *interface proxy*, a solução propõe um *proxy* que faz a intermediação entre os consumidores do serviço e a interface de implementação do serviço.

No modelo de versionamento deste trabalho é proposto um *proxy* entre os clientes e a atual implementação do serviço. Atual, pois podem existir várias interfaces de acesso, mas apenas um implementação ativa. O *proxy* que faz intermediação pode dinamicamente selecionar a rota para a versão de implementação que é distinguida pela requisição do cliente.

Na Figura 3.2 são mostrados dois *proxies*, um para cada serviço disponibilizado pelo fornecedor. Cada *proxy* compartilha a mesma interface que o próprio serviço que está intermediando. O *proxy* filtra a requisição através de algum tipo de metadata na mensagem e procura na tabela de configuração de *proxies* que possui qual rota deve criar para que a mensagem chegue a implementação correta. Essa escolha de rota é feita pelo *proxy* que altera o endereço de *endpoint* para a implementação escolhida. Para a resposta de retorno, a mensagem é novamente enviada ao *proxy* que através de um componente *Response Generator* transforma a mensagem e envia de volta a aplicação cliente que fez a requisição.

Figura 3.2 - Interface Proxy



Fonte: Frank et al. (2008)

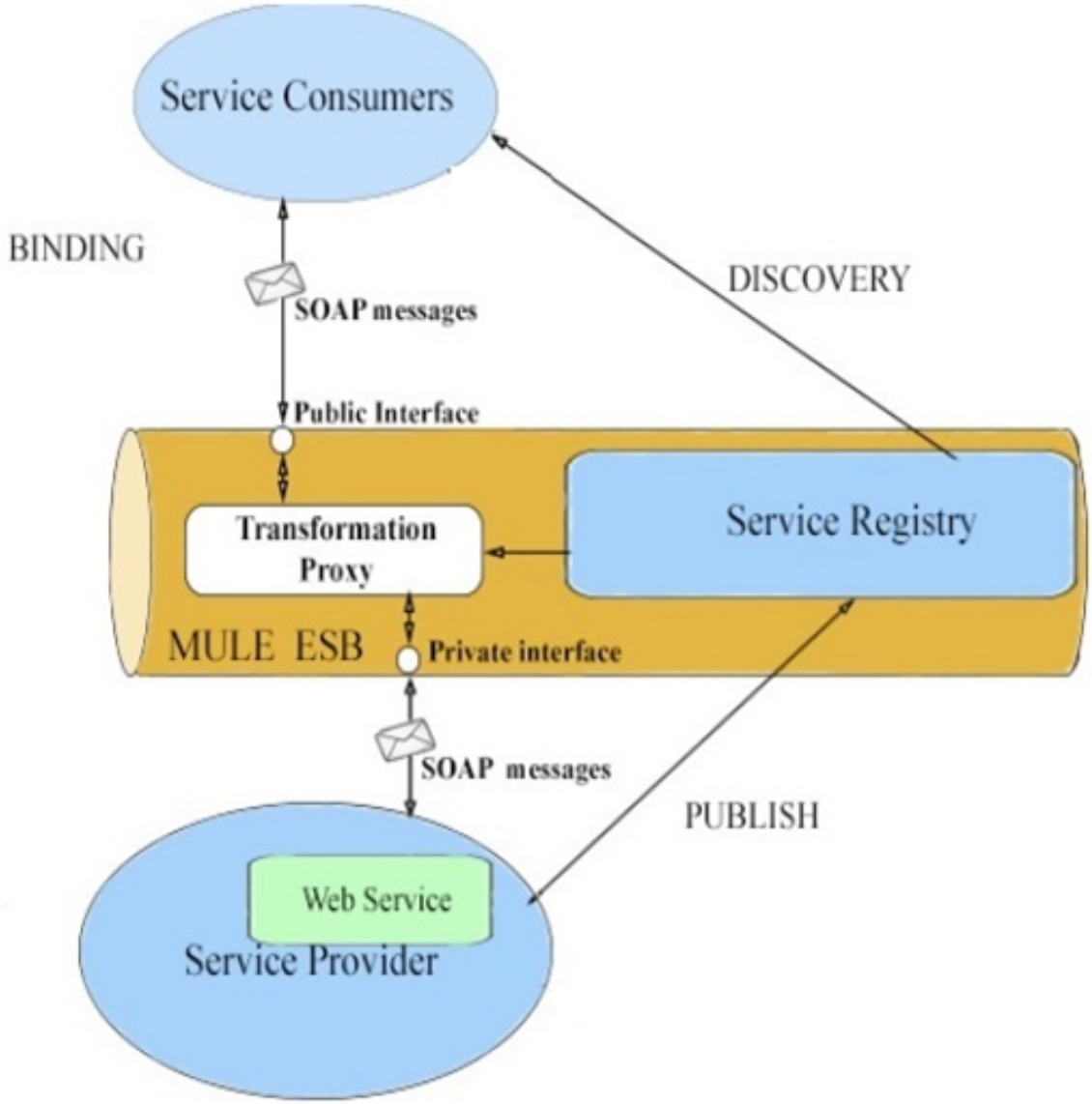
Nesta solução a geração do *proxy* é clonando a interface de acesso ao serviço. O *proxy* obtém o arquivo WSDL do arquivo EAR do serviço, clona sua interface e o torna disponível para acesso fazendo intermediação entre as aplicações. Uma vantagem da utilização do WSDL para isso é que ela separa a interface de acesso da implementação podendo se criar versões de um serviço.

3.3 Proxy Transformation

O modelo proposto por Chiponga et al. (2014), que foi chamado de *Proxy Transformation* também possui um intermediador entre as aplicações, porém mantém apenas uma interface de acesso mesmo que o serviço do cliente possua várias versões. É mantido também apenas uma implementação do serviço. Esse modelo pode ter um bom desempenho, pois apenas uma versão da interface e da inmplentação estão ativos.

Na Figura 3.3 é descrito como o modelo foi criado. O intermediador das mensagens é feito por um Enterprise Service Bus(ESB). Ele recebe as mensagens do cliente, faz as transformações necessárias na mensagem utilizando alguma implementação que foram previamente inseridas no proprio ESB e entrega a mensagem ao destinatário. Uma desvantagem deste modelo é a inserção em *hard-code* do procedimento de

Figura 3.3 - Proxy Transformation



Fonte: Chiponga et al. (2014)

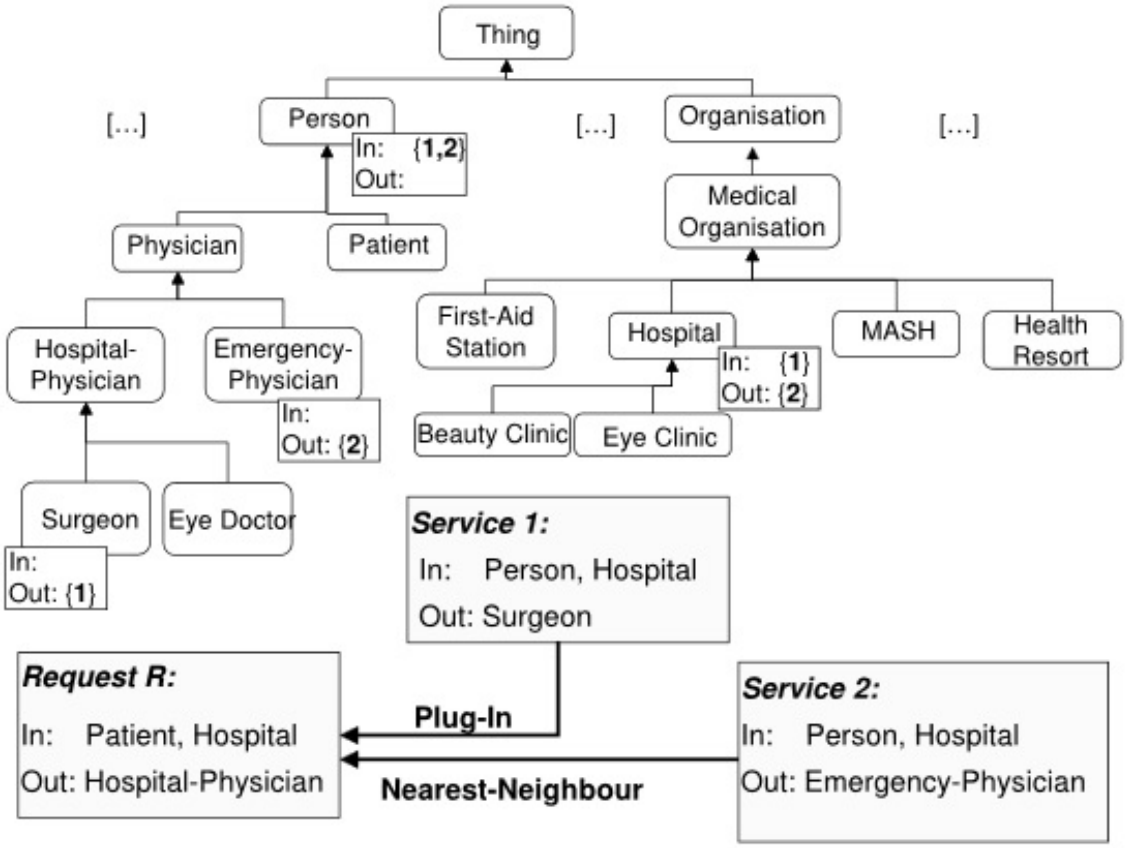
como transformar uma mensagem de uma versão para outra. A cada nova versão de contrato é necessário inserir no intermediador.

Outro ponto deste modelo que deve-se levar em consideração é que para ele definir em qual versão de um contrato está uma mensagem recebida é necessário compará-la co todos as versões de contrato disponíveis. Isso pode ser muito custoso quando se tem muitas versões de contratos e pode gerar ambiguidades, pois mais de uma versão de contrato pode ser compatível com a mensagem recebida.

3.4 Hybrid Matchmaking OWSL-MX

Klusch et al. (2006) propõe um modelo utilizando um *web service* semântico híbrido chamado de OWSL-MX. O modelo é aplicado para tentar compatibilizar a implementação do serviço com as requisições através de *matching* de entidades. Para tanto, toda as entidades do domínio devem ser mapeadas em um modelo ontológico. A Figura 3.4 apresenta um exemplo deste mapeamento juntamente com o serviço OWSL-MX.

Figura 3.4 - OWLS-MX



Fonte: Klusch et al. (2006)

Uma requisição R é enviada com os parâmetros `Patient` e `Hospital` e espera receber a entidade `Hospital-Physician` e modelo tenta achar a equivalência utilizando o modelo ontológico, pois o serviço espera receber outros tipos de parâmetros, neste caso, `Person` e `Hospital` e tem um retorno do tipo `Emergency-Physician`. O modelo então tenta fazer uma equivalência da requisição **R** com os serviços **S1** e **S2** e não

consegue criar essa equivalência retornando uma falha.

Utilizando o modelo ontológico, as entidades possuem uma característica semântica. O modelo compara cada parâmetro de entrada com suas entidades mais primitivas, ou seja, que estão acima na árvore ontológica dando valores a suas similaridades. Deste modo:

- `Patient` deriva de `Person`.
- `Hospital` deriva de `Medical Organization`.
- `HospitalPhysician` deriva de `Physician` e `Person`.
- `Surgeon` deriva de `HospitalPhysician`, `Physician` e `Person`.
- `EmergencyPhysician` deriva de `Physician` e `Person`.

Como resultado destas compatibilides e derivações, o modelo dá pesos como forma de similaridade a cada serviço disponível, como é o caso de **S1** e **S2**. O trabalho mostrou que modelos ontológicos podem ser utilizados para fazer equivalência de requisições e serviços através de atributos semânticos.

4 MODELO ARQUITETURAL PARA GERENCIAMENTO DE VERSÕES DE CONTRATOS DE SERVIÇOS WEB

Este capítulo descreve o modelo arquitetural para o gerenciamento de versões de contratos. Para isso são mostradas as principais características que o modelo a ser desenvolvido deve possuir para tentar solucionar o problema da interoperabilidade descrita na primeira seção. Dessa forma, são enumerados os requisitos que o modelo deve possuir para atender a proposta de solução. Os requisitos descrevem como a arquitetura deve se comportar diante de vários tipos de cenários de comunicação entre fornecedor e consumidor de serviços e como o modelo deve automatizar essa comunicação.

Ainda nesta capítulo são mostradas visões acerca do modelo arquitetural proposto, passando por uma visão geral da arquitetura e apresentados dois cenários de utilização com aplicações reais testadas em um ambiente controlado com a descrição de cada componente do modelo e como é sua dinâmica quando utilizada. As seções deste capítulo descrevem também os protocolos necessários na comunicação entre cada componente e as motivações de seu papel na arquitetura.

4.1 Requisitos do modelo

O modelo deve atender a alguns requisitos básicos que foram presumidos para que ele possa suprir o problema de evolução de aplicações que compartilham um mesmo contrato. Esses requisitos serão avaliados posteriormente e verificados se apenas com eles se pode chegar a uma solução para este problema. Abaixo seguem os requisitos bem como sua justificativa:

(R1) A solução deve gerenciar os vários contratos existentes entre as aplicações, devendo gerenciar os contratos e versões a fim de poder consultar, armazenar e disponibilizar, caso seja necessária uma transformação, os *scripts* para conversão e deve conseguir fazer consultas sobre versões de aplicações e contratos. Essa gerência deve ser executada totalmente pelo modelo de arquitetura para que não haja envolvimento humano.

(R2) A solução deve executar as transformações de forma transparente às aplicações. Elas não devem participar do processo em sí de transformação da mensagem de um formato para outro, isto é, devem apenas enviar sua mensagem sem saber como ou quem fará estas alterações. Esta transparência é necessária para envolver o mínimo possível as aplicações nas transformações geradas desacoplando os com-

ponentes da arquitetura.

(R3) A solução deve ser capaz de aplicar as transformações necessárias nas mensagens. De acordo com o estudo efetuado no artigo de França et al. (2015), existem indícios que as transformações mais encontradas em mudanças de contratos são: adição, modificação ou remoção de um elemento, alteração de multiplicidade, adições ou alterações em *enumerations* e modificação ou remoção de um atributo. Para tanto, a solução deve ser capaz de realizar esses tipos de transformações nas mensagens. Esse requisito deve ser atendido para que uma aplicação consiga enviar uma mensagem em certa versão e chegue a outra aplicação em uma outra versão.

(R4) A solução deve ser capaz de executar uma transformação de forma bidirecional, por exemplo, transformar uma mensagem da versão 1 de certo contrato para a versão 2 e também da versão 2 para a versão 1 deste mesmo contrato. Isso garante que uma aplicação interagindo com outra aplicação em uma versão diferente de contrato possa tanto enviar como receber mensagens.

(R5) A solução deve ser pouco intrusiva às aplicações existentes, isto é, a sua adoção deve impactar o mínimo possível nas aplicações existentes que já se comunicam. Muitas vezes as aplicações envolvidas na comunicação estão em plataformas e linguagens diferentes e este baixo impacto de alterações torna a transição para a nova arquitetura mais simples.

(R6) A solução deve gerar informações quando um contrato é alterado para que se possa converter as mensagens de uma versão de contrato para outra. É importante a geração de tais informações no momento da alteração de um contrato pois gerá-las manualmente é um trabalho difícil e quando se tem contratos com grande volume de entidades a comparação entre uma versão e outra muitas vezes é inviável.

(R7) A solução deve armazenar as informações geradas a partir da alteração de um contrato para que se possa utilizá-las posteriormente para conversões das mensagens. Uma das funções do modelo proposto é a recuperação e aplicação de conversões sobre as mensagens em versões diferentes de um contrato, para isso o armazenamento das informações sobre as conversões é de suma importância, pois o modelo necessita de tais dados para as conversões.

(R8) A solução deve prover uma alternativa para a possibilidade de não ser possível converter uma mensagem recebida. Quando versões de uma mensagem não podem ser convertidas, seja por falta de informações sobre a conversão ou por incompati-

bilidade de versões, a solução deve retornar uma mensagem de erro ao produtor da mensagem dizendo que não foi possível converter a mensagem.

A partir de um modelo que atenda os requisitos descritos acima pode-se criar uma arquitetura possível de transformar mensagens para manter a interoperabilidade entre aplicações que utilizam contratos em versões diferentes e que gerencie as informações sobre conversões de mensagens, versões de contratos e versões de aplicações. Ela também irá dar suporte ao versionamento de contratos que é um dos objetivos desta arquitetura. Este versionamento traz a vantagem da não intervenção ou gerenciamento de versões por pessoas que pode se tornar muito difícil quando se tem muitos contratos e versões para serem gerenciados.

4.2 Visão geral

O objetivo dessa seção é apresentar uma visão geral de como o modelo proposto funciona. As seções seguintes irão detalhar como cada um dos passos que serão descritos funcionam.

Quando aplicações desejam trocar informações através de um serviço *web* elas compartilham um contrato onde é descrito o formato das mensagens,porém, quando há necessidade de evoluir o contrato as aplicações devem se adaptar a este novo formato para que possam continuar se comunicando. Se apenas umas das aplicações envolvidas se adapta a este novo contrato de mensagens existe a necessidade de a cada toda troca de informações entre as aplicações a mensagem ser transformada para a versão de contrato da aplicação destinatária. Para isso, o modelo considera a utilização de uma ferramenta de refatoração de XML para a evolução dos contratos pois ela gera *scripts* XSLT que convertem a mensagem de uma versão para outra.

Esta ferramenta auxilia na alteração de contratos disponibilizando refatorações para alterações de elementos e atributos. A cada refatoração feita no contrato a partir da ferramenta são gerados *scripts* XSLT que podem ser aplicados ao XML da mensagem para sua conversão. A ferramenta também gera *scripts* para o caso contrário onde a mensagem que está no formato de contrato mais atual deve ser convertida para a versão mais antiga.

Essas versões de contratos e seus respectivos *scripts* são armazenados em um repositório através de um componente chamado Serviço de Gerência de Versões. Esse componente que gerencia todos os arquivos gerados pela ferramenta de refatoração disponibiliza serviços para consulta e recuperação de arquivos para serem aplicados

às mensagens. Este componente é disponibilizado em forma de serviço e que pode ser acessado por qualquer aplicação que implemente sua interface, isto é, por outro *middleware* ou diretamente por outras aplicações.

Quando umas das aplicações migra de uma versão para outra e deseja continuar se comunicando com aplicações em outras versões de contratos, ela envia sua mensagem a um *Middleware*, para que ele entregue a mensagem ao destinatário (que está em outra versão). O *Middleware* acessa os Serviços de Gerência de Versões e recupera do repositório os arquivos necessários para converter a mensagem para o formato esperado e entregar à aplicação que irá consumir a mensagem.

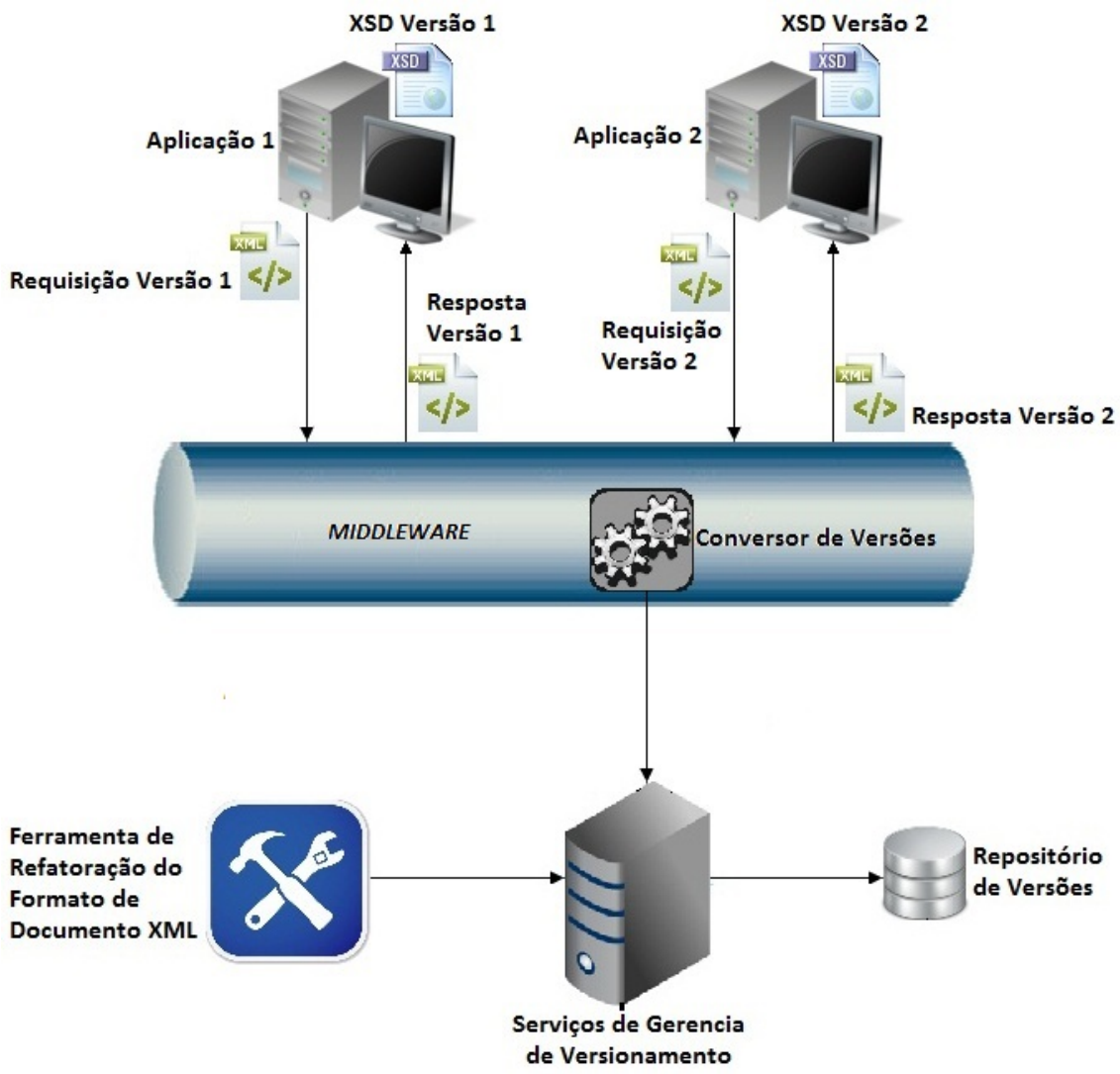
O mesmo acontece com a resposta do destinatário que requisitou o serviço: a mensagem passa pelo *Middleware* que converte a mensagem utilizando os Serviços de Gerência de Versões e entrega a mensagem ao requisitante.

Na Figura 4.1, é representado o modelo arquitetural proposto. Para ilustrar seu funcionamento, duas aplicações chamadas de Aplicação 1, que fará o papel de uma aplicação consumidora de um serviço e Aplicação 2 que fará o papel de uma aplicação fornecedora de um serviço são mostradas trocando informações no formato XML. As duas aplicações não conseguem se comunicar diretamente pois cada uma delas está utilizando uma versão diferente do contrato de mensagens. A Aplicação 1 utiliza a versão de contrato 1 e a Aplicação 2 utiliza a versão de contrato 2.

Ainda na Figura 4.1 é visto que as aplicações 1 e 2 e estão se comunicando através do *Middleware* e não mais diretamente entre si. Quando uma aplicação deseja enviar uma mensagem a outra aplicação ela envia sua mensagem ao *Middleware* que utiliza o Conversor de Versões para consultar se as aplicações envolvidas na troca de informações compartilham uma mesma versão de contrato de serviços.

Caso não compartilhem a mesma versão do contrato, o Conversor de Versões recupera os arquivos de transformações entre a versão do contrato da aplicação produtora da mensagem para a versão do contrato utilizado pelo consumidor da mensagem através dos Serviços de Gerência de Versionamento. São aplicados os *scripts* sobre a mensagem XML e enviadas ao consumidor da mensagem, tornando assim, transparente às duas aplicações envolvidas que a mensagem necessitava de algum tipo de conversão. O mesmo se aplica à resposta do consumidor à aplicação produtora da mensagem: o consumidor envia a resposta ao *Middleware* que faz novamente uma conversão para que o produtor receba a mensagem no mesmo formato em que enviou.

Figura 4.1 - Visão Geral do Modelo de Arquitetura



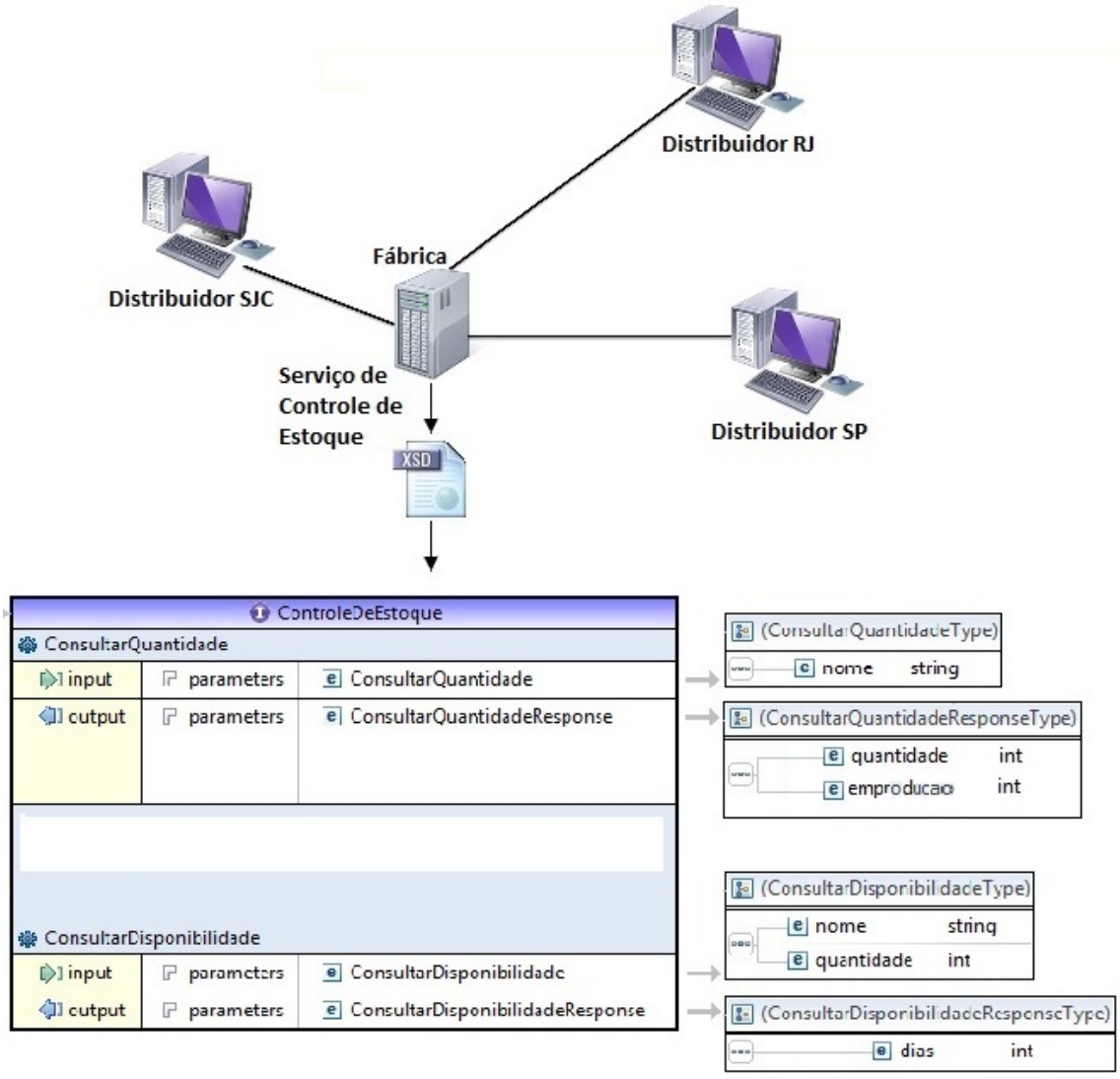
Fonte: Produção do autor

Existe o caso em que não é possível converter uma mensagem de uma versão de contrato para outra por não existirem *scripts* de conversão no repositório de versões. Caso não seja possível converter a mensagem, o *Middleware* retorna ao produtor da requisição uma mensagem de erro informando que não foi possível convertê-la. Posteriormente, caso os *scripts* sejam adicionados ao repositório via Serviços de Gerência de Versões, o *Middleware* passa a estar apto a converter a mesma mensagem. Vale ressaltar que a inexistência dos *scripts* de conversão pode ser proposital em alguns casos significando que a modificação feita no contrato não é compatível com a versão anterior.

4.3 Exemplo de utilização

Supõe-se que exista uma fábrica produtora de componentes automobilísticos que disponibiliza aos seus distribuidores um serviço *web* para consulta de estoque de peças. Este serviço segue uma padronização definida por um contrato e todas as distribuidoras que queiram utilizar o serviço devem implementar o contrato disponibilizado pela fábrica. Na Figura 4.2 está representado esse serviço bem como o detalhamento da interface descrita no contrato.

Figura 4.2 - Serviço disponibilizado pela fábrica.



Fonte: Produção do autor

Com esse serviço, todas as distribuidoras são capazes de obter através da operação `ConsultarQuantidade` quantas peças existem disponíveis e quantas estão em produção no momento da consulta. Através da operação `ConsultarDisponibilidade` podem saber em quantos dias estará disponível a quantidade desejada de certa peça da fábrica.

A fábrica notou que um pedido de uma distribuidora leva ao menos 24 horas para ser processado não podendo ser entregue no mesmo dia. Por uma evolução do serviço o contrato foi alterado para melhor prover os dados disponíveis. O contrato alterou o retorno da operação `ConsultarQuantidade` que retornava dois valores, `quantidade` e `emproducao`, como pode ser visto na Figura 4.2.

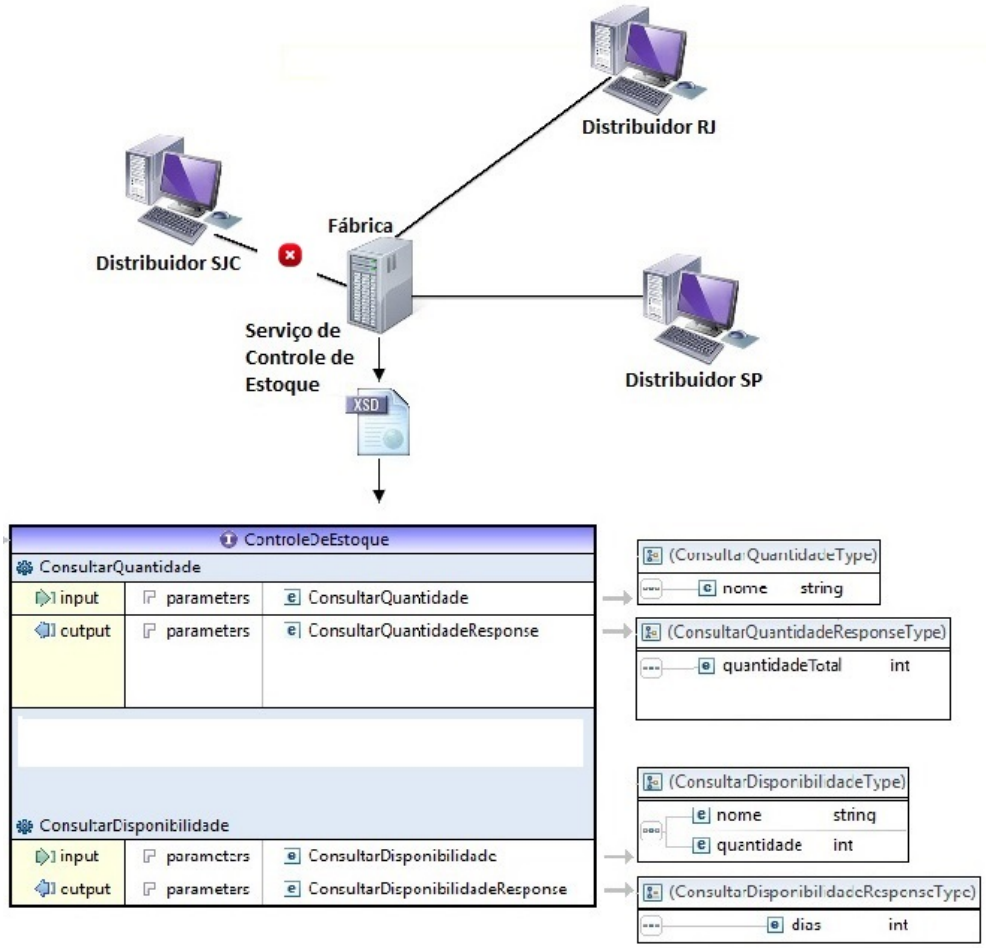
O novo contrato retorna apenas um inteiro chamado `quantidadeTotal` indicando as peças disponíveis mais as que estão em produção no mesmo dia e que não pertencem a nenhum pedido de produção. Essa alteração de retorno pode ser vista na Figura 4.3 onde o método `ConsultarQuantidade` agora possui apenas um parâmetro do tipo inteiro.

As distribuidoras de SP e RJ adaptaram suas aplicações para o novo contrato, mas a distribuidora de SJC ainda não conseguiu se adaptar fazendo com que apenas ela não consiga utilizar o serviço. Para que a distribuidora de SJC consiga voltar a se comunicar é necessário que ela também se adapte ao novo contrato de serviço.

Com a arquitetura apresentada esse problema pode ser resolvido deixando todas as distribuidoras aptas a utilizar o serviço mesmo que elas não alterem suas aplicações para o novo contrato. Este modelo prevê que para se alterar o contrato de serviço é necessário utilizar uma ferramenta de refatoração de XML que gera os *scripts* XSLT de conversão de uma versão para outra. Após serem feitas todas as alterações no contrato a nova versão e os *scripts* XSLT são armazenadas pela própria ferramenta no repositório de versões através do Serviço de Gerência de Versões.

A Figura 4.1 descreve como ficaria a troca de mensagens entre a Fábrica e a distribuidora que ainda não alterou sua aplicação para o novo contrato. A Distribuidora SJC envia a requisição da operação no formato antigo, porém envia ao *Middleware*. Ele reconhece que a mensagem não está no formato mais novo e trata de convertê-la para a versão mais recente. Em seguida, repassa a mensagem ao serviço da Fábrica e ela responde no novo formato. Por fim, o *Middleware* converte para o formato antigo para que a Distribuidora SJC entenda.

Figura 4.3 - Disbribuidora SJC utilizando o contrato antigo.



Fonte: Produção do autor

A medida que as distribuidoras vão adaptando seu código para o novo contrato, o *Middleware* deixa de fazer a conversão da mensagem para quem requisitou o serviço e já está utilizando o novo contrato. Essa geração de *scripts* e a gerência deles é bastante útil quando tem-se muitas versões de contratos com muitas aplicações utilizando vários serviços. Realizar manualmente a gerência de várias versões de contratos e a geração de *scripts* de conversão pode ser uma tarefa bem trabalhosa.

4.4 Descrição dos componentes do modelo

Com o exemplo dado anteriormente fez-se menção ao *Middleware* e sua tramitação. Essa seção descreve com mais detalhe cada componente e suas entradas e saídas. A Figura 4.1 pode ser consultada para situar o componente dentro do contexto do modelo.

Aplicações Consumidora e Fornecedora: esses componentes representam as aplicações que desejam trocar informações via serviços *web* e que podem ter serviços em diferentes versões. Pode existir o caso das aplicações compartilharem a mesma versão de contrato, não precisando de conversões para se comunicar. Outro caso é quando ambas as aplicações não utilizam a mesma versão de um contrato sendo necessário efetuar conversões nas mensagens para que consigam se comunicar. Para isso a arquitetura prevê que as aplicações não mais enviem mensagens diretamente de uma para outra, mas através do componente ***Middleware***.

Versões do Formato XSD: representam as diferentes versões de contratos que cada aplicação pode estar utilizando. Quando uma aplicação cria uma mensagem de requisição ou de resposta deve obedecer ao contrato para que seja considerada válida.

XMLs de Requisição e Resposta: representam as mensagens geradas para envio e recebimento de informações entre as aplicações que desejam se comunicar. As mensagens de requisições e respostas são estruturas XML formatadas de acordo com a versão do contrato utilizado.

Middleware: Para o recebimento e envio de informações entre as aplicações de forma transparente, a arquitetura prevê uma aplicação, chamada de *Middleware*, que provê serviços de recebimento de mensagens, conversão entre versões e entrega de mensagens. Este componente possui um subcomponente chamado **Conversor de Versões** que tem acesso aos **Serviços de Gerência de Versionamento**. Quando uma mensagem é recebida pelo ***Middleware*** é repassada ao **Conversor de Versões** que irá processá-la. Para que o ***Middleware*** possa processar a mensagem e verificar se é possível entregar a mensagem recebida no formato correto ele deve conhecer quais aplicações está intermediando e quais as versões de contrato utilizadas.

Conversor de Versões: deve verificar na mensagem recebida qual a versão de seu contrato e qual a versão do contrato do consumidor. Para se descobrir em qual versão está a mensagem recebida, o conversor lê o cabeçalho da mensagem SOAP que irá conter a informação da versão de contrato utilizado, neste caso, o consumidor do serviço. No caso das aplicações fornecedoras de serviços, suas versões de contratos devem ser armazenadas previamente em um repositório para que o **Conversor de Versões** possa consultá-las.

Caso as versões do contrato da mensagem do produtor e do consumidor sejam a mesma o conversor solicita ao ***Middleware*** que apenas encaminhe a mensagem.

Caso não tenham as mesmas versões ele deve verificar se existem *scripts* de conversão utilizando os **Serviços de Gerência de Versionamento** que se encarrega de verificar a existência dos *scripts*.

É possível que não existam *scripts* para uma dada versão no **Repositório de Versões** por ainda não terem sido adicionados ou por não ser possível a conversão. O ***Middleware*** neste caso enviará uma mensagem de resposta informando que houve um erro ao se tentar converter a mensagem. Nota-se que a equipe de desenvolvimento que alterou o XSD utilizado deve adicionar os *scripts* de conversão ao **Repositório de Versões** para que todos possam converter suas mensagens para a versão de XSD alterada utilizando as operações de envio de *scripts* e contratos disponibilizados pelo **Serviço de Gerência de Versionamento**.

Serviços de Gerência de Versionamento: este componente, que disponibiliza suas funcionalidades através de serviços *web* irá prover operações para:

- Consulta sobre a versão atual do contrato de uma aplicação que será utilizado pelo **Conversor de Versões** para verificar se a versão do contrato do consumidor da mensagem é a mesma do produtor.

- Verificação da existência de *scripts* de uma dada versão para outra que acessa o **Repositório de Versões** e verifica se existem *scripts* de transformação de uma versão de contrato para outra.

- Obtenção de *scripts* de conversão que recupera os *scripts* de conversão de uma versão de contrato e os aplica sobre a mensagem devolvendo ao **Conversor de Versões** já na versão do consumidor da mensagem.

- Adição de novos contratos e *scripts* que disponibiliza uma forma de se adicionar novos contratos de aplicações ou novos *scripts* gerados pela **Ferramenta de Refatoração do Formato do Documento XML**.

Este componente possui como entrada de dados requisições em formato XML feitas pelo **Conversor de Versões** ou pela **Ferramenta de Refatoração de Formato de Documentos XML** e possui como saída uma resposta também em formato XML para quem o solicitou.

Repositório de Versões: representa o repositório de contratos e *scripts* de conversões entre os contratos onde estão armazenadas informações acerca dos arquivos XSD que as aplicações utilizam, quais as versões em que existem transformações e

os próprios *scripts* de transformação de uma versão de contrato para outra. Esse repositório fica disponível ao **Serviço de Gerência de Versionamento** para que ele acesse os dados necessários para fazer as conversões das mensagens.

Ferramenta de Refatoração do Formato do Documento XML: este componente deve permitir alterar o formato do contrato em XSD e gerar os *scripts* XSLT para conversão de XMLs entre as versões do formato do arquivo XSD alterado. Para cada alteração feita do documento XSD a ferramenta deve gerar *scripts* que possam ser aplicados ao documento XML para se adequar ao novo formato do XSD.

A ferramenta deve armazenar no **Repositório de Versões** através dos **Serviços de Gerência de Versionamento** as novas versões do contrato e *scripts* gerados. O acesso aos serviços de gerenciamento desacopla a ferramenta de geração de *scripts* ao repositório utilizado deixando a cargo deste serviço a gerência de todos os documentos que são gerados.

Este componente possui como entrada de dados um contrato no formato XSD que se deseja alterar e possui como saída uma mensagem em formato XML enviada ao Serviço de Gerência de Versionamento contendo o novo contrato e os *scripts* de conversão em formato XSLT.

4.5 Dinâmica do processo

O objetivo dessa subseção é apresentar os principais processos que serão executados por esse modelo arquitetural. Ele envolve a descrição dos processos no gerenciamento das versões de um documento e a troca de mensagens entre as aplicações.

4.5.1 Processo de troca de mensagens

Na Figura 4.4 é representada a dinâmica de troca de mensagens segundo o modelo arquitetural, dividida em cinco componentes distintos, descrevendo como a mensagem é criada até o momento em que ela chega ao consumidor.

Na primeira fase a Aplicação A cria uma mensagem em formato XML com os dados que necessita enviar ao consumidor. Esta mensagem deve passar por um formatador para que se adeque ao modelo de XML utilizado. Por fim, é necessário adicionar um cabeçalho à mensagem SOAP com a versão do XSD utilizado como modelo da mensagem. Com esses passos a mensagem já pode ser enviada ao *Middleware*.

Quando o *Middleware* recebe uma mensagem, repassa-a ao Conversor de Versões

que deve verificar a quem se destina e se é necessário algum processamento. Ao chegar uma mensagem ele deve verificar no cabeçalho da mensagem se a versão do XSD utilizado é a mesma versão da utilizada na aplicação consumidora. Caso a versão seja a mesma não há processamento extra e o Conversor de Versões solicita ao *Middleware* que entregue a mensagem ao consumidor.

Para essa verificação, todas as versões de contratos das aplicações fornecedoras de serviços são previamente inseridas no Repositório de Versões através de uma operação no Serviço de Gerência de Versionamento. Apenas as aplicações consumidoras necessitam adicionar ao cabeçalho da mensagem a versão de contrato utilizado pois ao chegar ao conversor de versões ele compara a versão enviada no cabeçalho da mensagem com a versão do fornecedor do serviço que está armazenada no Repositório de Versões.

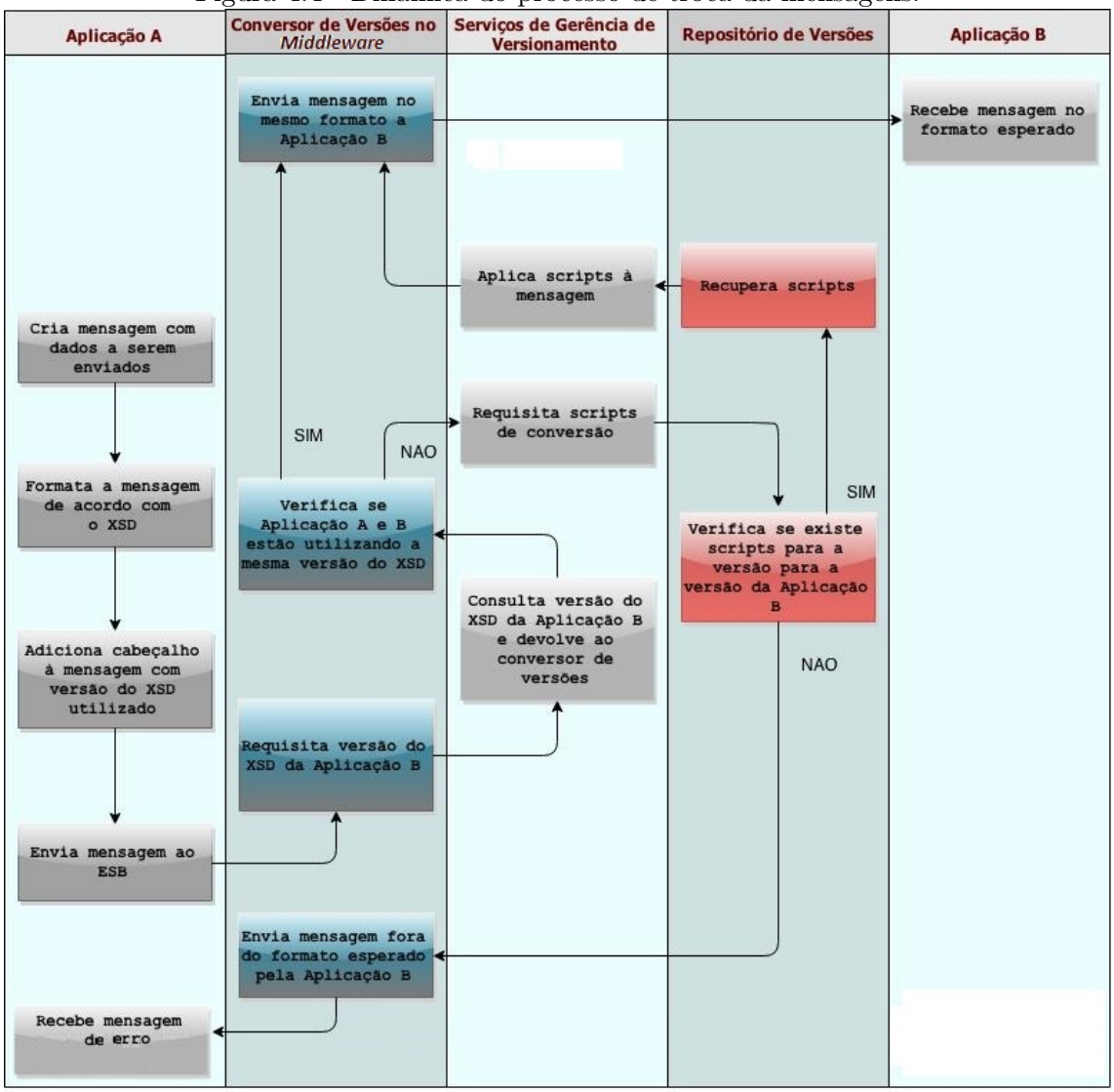
Caso a versão do XSD utilizado na Aplicação A não seja a mesma versão da utilizada na Aplicação B o Conversor de Versões deve gerar um processamento para transformar a mensagem de uma versão para outra. Para isso devem existir *scripts* de transformação entre as versões armazenadas no Repositório de Versões. O Conversor de Versões requisita aos Serviços de Gerência de Versionamento que verifique se existem *scripts* de conversão entre as versões do contrato da Aplicação A e da Aplicação B.

Caso não existam os *scripts* no repositório de versões uma mensagem é retornada informando que não foi possível converter a mensagem para a versão do consumidor da mensagem já que o Serviços de Gerência de Versionamento não consegue transformar para o formato esperado. Caso existam os *scripts* de transformação, eles são obtidos do Repositório de Versões pelo componente Serviços de Gerência de Versionamento e os aplica à mensagem utilizando um processador XSLT.

A mensagem já transformada pela aplicação dos *scripts* XSLT é devolvida ao Conversor de Versões que faz a validação da mensagem com o XSD para qual os *scripts* foram aplicados. Se a mensagem foi transformada corretamente e não houve nenhum tipo de erro, ela deve ser válida para o XSD utilizado pela aplicação destinatária. Só então a mensagem é encaminhada ao *Middleware* que entrega a mensagem ao consumidor da mensagem. O mesmo ocorre na resposta da requisição de uma aplicação que deve ser convertida para a versão de contrato utilizado na aplicação que enviou a requisição.

Por questões de eficiência do serviço o *Middleware* pode criar um *cache* dos últimos

Figura 4.4 - Dinâmica do processo de troca da mensagens.



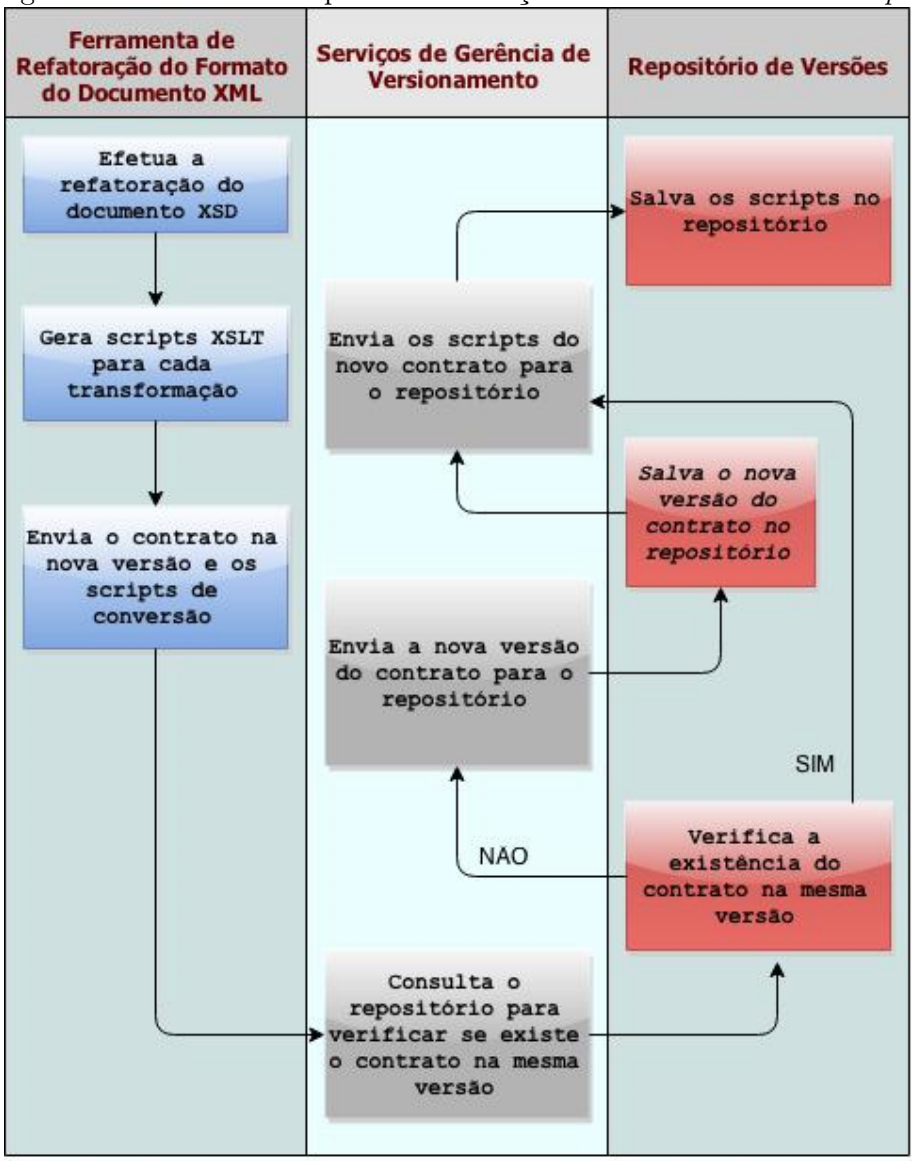
Fonte: Produção do autor

scripts recuperados do Repositório de Versões para não necessitar recuperá-lo mais de uma vez num curto espaço de tempo. Esse comportamento não interfere em nada o seu funcionamento e fica a cargo da implementação sua utilização e os períodos de validade deste *cache*.

4.5.2 Gerenciamento das versões de um documento

Na Figura 4.5 é representada a dinâmica da adição de novo contratos e *scripts* de conversão entre versões de um mesmo contrato no Repositório de Versões.

Figura 4.5 - Dinâmica do processo de adição de novos contratos e *scripts*



Fonte: Produção do autor

A primeira etapa consiste na refatoração do documento XSD utilizando a ferramenta prevista no modelo de arquitetura proposto. Após as alterações, são gerados os *scripts* XSLT de conversão da versão antiga do documento XSD para nova versão, bem como da nova versão para a versão antiga. Através das funcionalidades disponibilizadas a ferramenta envia a nova versão do documento XSD e os *scripts* de conversão para o Serviço de Gerência de Versionamento.

Nesta etapa é necessário consultar o Repositório de Versões para verificar se já existe um contrato na versão especificada, então o Serviço de Gerência de Versionamento

consulta o Repositório de versões. Caso não exista nenhum contrato na versão especificada o Serviço de Gerência de Versionamento envia o ao repositório para que seja armazenado.

São enviados do Serviço de Gerência para o Repositório de Versões os*scripts* gerados, nome do contrato, a versão atual e para qual versão os *scripts* transformam. Assim, a refatoração, a geração de *scripts* e sua adição ao Repositório de Versões são efetuadas.

4.6 *Rationale*

Esta subseção trata do porquê da utilização de cada componente do modelo de arquitetura com uma explicação não do seu uso, mas de sua necessidade para a troca de mensagens e gerenciamento de contratos. Após as seções anteriores com uma visão geral do modelo arquitetural, explicação de cada componente e seus processos na troca de informações, esta seção explicará a necessidade de cada componente. Durante as explicações, os requisitos apresentados para o modelo na seção 4.1 serão referenciados para uma melhor compreensão.

O *Middleware* foi criado de modo a ser independente das aplicações, desacoplando das aplicações consumidoras todos os serviços disponibilizados pelos fornecedores. Criá-lo como um serviço *web* traz a vantagem desse desacoplamento e como os fornecedores e consumidores já são baseados numa tecnologia SOA o impacto de mudanças nas aplicações é mínimo, reduzindo a necessidade de novos módulos dentro das aplicações. **(R5)**. As transformações das mensagens ocorrem internamente no *Middleware*, não necessitando que as aplicações envolvidas na comunicação precisem saber como elas funcionam, trabalhando de forma transparente.

O Conversor de Mensagem foi criado como parte da implementação do *Middleware* já que apenas ele tem a necessidade de receber as mensagens das aplicações que desejam trocar informações. E como as aplicações que enviam e recebem mensagem tratam direito com o *Middleware* achou-se interessante manter o Conversor de Mensagem como parte da implementação. As conversões são bidirecionais, isto é, pode-se converter uma mensagem de uma versão para outra e a sua volta também, tudo de forma transparente às aplicações envolvidas, pois elas não participam de qualquer processo durante as transformações das mensagens. **(R3), (R2) e (R4)**.

O Serviço de Gerência de Versionamento foi criado como um serviço *web* pois, não só o *Middleware* necessita dos seus serviços, mas também a ferramenta que faz a

refatoração quando necessita armazenar novos contratos e *scripts*. **(R1)**. Criá-lo como um serviço *web* garante quatro coisas:

- Desacopla a Ferramenta de Refatoração do Formato de Documentos XML do *Middleware* deixando-o com uma interface de acesso mais limpa, com operações apenas na troca de mensagens, sem se preocupar com armazenamento de contratos e *scripts*
- Desacopla o próprio *Middleware* do Serviço de Gerência de Versionamento deixando independente se estão todos implementados no mesmo computador ou em locais distintos
- Faz com que a Ferramenta de Refatoração do Formato de Documentos XML não necessite acessar o *Middleware* pois é uma ferramenta independente da tramitação das mensagens
- O desacoplamento do Serviço de Gerência de Versionamento permite que possam existir várias instâncias do *Middleware* acessando o, inclusive por outras aplicações que queiram recuperar os *scripts* de conversões para outros fins.

Esse componente em forma de serviço deixa o modelo mais flexível, pois cada componente que o acessa pode ser modificado à necessidade do ambiente e uso e provê uma única interface para quem quer que o acesse, seja um *Middleware* ou outra aplicação qualquer.

Como foi dito, a Ferramenta de Refatoração do Formato de Documentos XML acessa o Serviço de Gerência de Versionamento através do serviço *web*, o desacoplando de quem irá gerar as mudanças nos contratos e seus respectivos *scripts* **(R6)**. Novas ferramentas de geração de *scripts* XSLT com mais funcionalidades podem surgir no mercado ou serem desenvolvidas por equipes internas e serem substituídas para acessar o Serviço de Gerência de Versionamento. Não é necessário utilizar o Serviço de Gerência de Versionamento para acessar o Repositório de Versões, pois pode ser feito manualmente, porém não é o ideal, já que o Serviço de Gerência de Versionamento conhece todas as regras de negócio para adição, modificação e consulta dos *scripts* armazenados.

Por fim, tem-se um Repositório de Versões, independente no modelo, já que em muitos locais, Sistemas Gerenciadores de Banco de Dados (SGBDs) são instalados

e mantidos em computadores distintos dos que executam alguma aplicação **(R7)**. Essa boa prática auxilia na geração de *backups* dos dados, além de poder se manter os dados ativos mesmo que a aplicação não esteja em operação.

Com aplicações rodando no mesmo computador que um SGBD pode-se comprometer o desempenho de ambos, pois compartilham o mesmo processador. Deste modo, o modelo de arquitetura foi proposto a definir o Repositório de Versões como um componente a parte e não como parte dos Serviços de Gerência de Versionamento.

4.7 Requisitos de ambiente para implantação do modelo

Esta subseção trata dos requisitos necessários para que o modelo de arquitetura proposto possa ser aplicado a um conjunto de aplicações. Serão descritos onde cada componente da arquitetura deve ou pode estar, bem como qual a interface o *Middleware* de escolha deve implementar.

Para a utilização do modelo arquitetural proposto é necessário que se tenha um ambiente onde existam aplicações se comunicando através de *web services* onde deve haver ao menos uma aplicação fornecedora de serviços e ao menos uma aplicação consumidora desse serviço. Elas devem trocar informações através de documentos XML como é comumente utilizado em aplicações SOA.

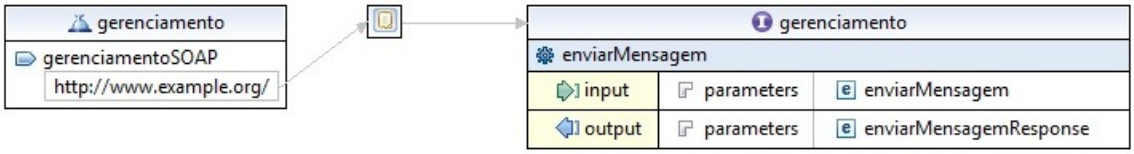
Todas as aplicações envolvidas na troca de mensagens devem ser capazes de se comunicar com uma aplicação intermediária, tratada aqui como um *Middleware* dos serviços disponíveis. É necessário que exista um computador acessível à aplicação *Middleware* que possua um SGBD para que se possa armazenar, consultar e recuperar dados necessários ao funcionamento do modelo proposto. Este computador com o SGBD instalado não precisa necessariamente ser um diferente do computador que possui a aplicação de *Middleware*.

Quanto a geração dos arquivos necessários para o funcionamento das conversões que o modelo arquitetural se propõe a fazer é necessário que se tenha uma aplicação que gere documento XSLT de conversão para cada transformação efetuada sobre o XML *Schema* utilizado no contrato de serviços das aplicações fornecedoras envolvidas. É possível gerar esses arquivos manualmente, mas seria muito trabalhoso criar estes *scripts* para cada mudança efetuada no XSD, então recomenda-se utilizar alguma ferramenta que automatize essa geração.

Na Figura 4.6 são descritos os serviços que devem ser implementados no *Middleware* para a comunicação das aplicações fornecedoras. A interface apresentada é a descri-

ção de seus serviços que contém apenas uma operação: `enviarMensagem`.

Figura 4.6 - Interface de comunicação com a aplicação *Middleware*



Fonte: Produção do autor

4.8 Diferenciais do modelo

Este modelo proposto mostrou alguns diferencias em relação aos outros modelos propostos já citados anteriormente. Uma nova visão sobre o mesmo problema, trazendo alguns ganhos de gerenciamento, desempenho e simplificação de processos. A seguir serão listados estes diferencias:

- **Apenas uma implementação do serviço**: Neste modelo não é feito o roteamento da requisição para alguma implementação do serviço, é redirecionada a mensagem sempre para a mesma implementação. Esta forma traz a facilidade de não haver a necessidade de saber qual versão de contrato é vinculada a qual implementação, pois todas as mensagens, independentemente da versão do contrato, são enviadas ao mesmo serviço ativo e que possui apenas uma implementação.

- **Gerenciamento de várias versões de contratos**: com a criação de várias versões de um contrato de serviço existe a possibilidade de criação de várias interfaces de acesso ao serviço. A gerência desses contratos e versões de aplicações pode ser inviável quando feita manualmente. Este modelo proposto faz essa gerência de forma automatizada, não necessitando da interveção humana no processo. Além disso, os *scripts* de conversão de mensagens, que podem ser dezenas entre uma versão de contrato e outra também são gerenciados pelo modelo. Deste modo, a utilização deste modelo de arquitetura se difere no sentido de que não existe o papel de um administrador responsável por aplicar mudanças de contratos ao modelo, elas são feitas de forma automática.

- **As transformações não são *hard-code***: Em outros trabalhos, a aplica-

ção de conversões de mensagens é implementada no código do componente intermediário. Isto é um problema pois a cada alteração, a implementação do intermediador deve ser alterada. Neste modelo uma vez implantado o componente intermediário, não há mais alterações em seu código-fonte, mas apenas a adição de novos contratos e *scripts* ao repositório de contratos. Remover esses dados em *hard-code* significa que não existe mais o papel de uma equipe de implementação disponível para alterar o intermediador a cada mudança de contrato.

- **Pode ser aplicado à XMLs gerados em *runtime***: Um diferencial deste modelo é a aplicação das transformações à mensagens que são geradas em tempo de execução. Não é necessário que o arquivo da mensagem exista previamente para que os *scripts* sejam aplicados à ela. Muitas vezes as aplicações consomem serviços *web* sem possuirem os dados para requisição até o momento da chamada do serviço ou o serviço gera a resposta em *runtime*. Mesmo que isso ocorra, este modelo de arquitetura proposto consgue aplicar as transformações necessárias para a entrega da mensagem de forma correta.

- **Adição de novas versões não geram novas interfaces de acesso**: Um grande diferencial é a não geração de novas interfaces a cada mudança de contrato. Todas as aplicações consumidoras de serviços, mesmo que utilizando versões diferentes de um mesmo contrato, acessam o serviço pela mesma interface. Este modelo de acesso livra o fornecedor de serviços de manter múltiplas interfaces para um mesmo serviço, o que gera um ganho de desempenho e simplificação no processo de tramitação de mensagens.

- **Novos serviços não geram alterações**: Quando um novo serviço é adicionado ao modelo de arquitetura não há a necessidade de alteração na interface de uso do intermediador, isto é, adição de novos serviços não gera nenhum tipo de alteração na interface do intermediador. Isso garante que novos serviços sejam inclusos ao modelo apenas enviando os dados necessário ao repositório de contratos para que ele entenda como fazer as transformações. Essa característica permite que a composição de serviços utilizados e fornecidos pelas aplicações que estão sob esta arquitura seja mudada de forma muito simples e dinâmica.

5 PROVA DE CONCEITO

Esta seção trata sobre como foi criada a implementação do modelo de arquitetura proposto, especificando cada componente e as ferramentas utilizadas. O modelo proposto descreve componentes que interagem entre si e necessitam de entradas e saídas de dados diferentes. Serão descritos como foram implementados todos os componentes do modelo de arquitetura.

Essa implementação tem por objetivo tornar-se uma prova de conceito para assegurar de que é possível utilizar o conceito da arquitetura proposta, isso é, mostrar que é factível de ser implementado. Essa prova de conceito também será utilizada na avaliação descrita no próximo capítulo.

5.1 Visão geral

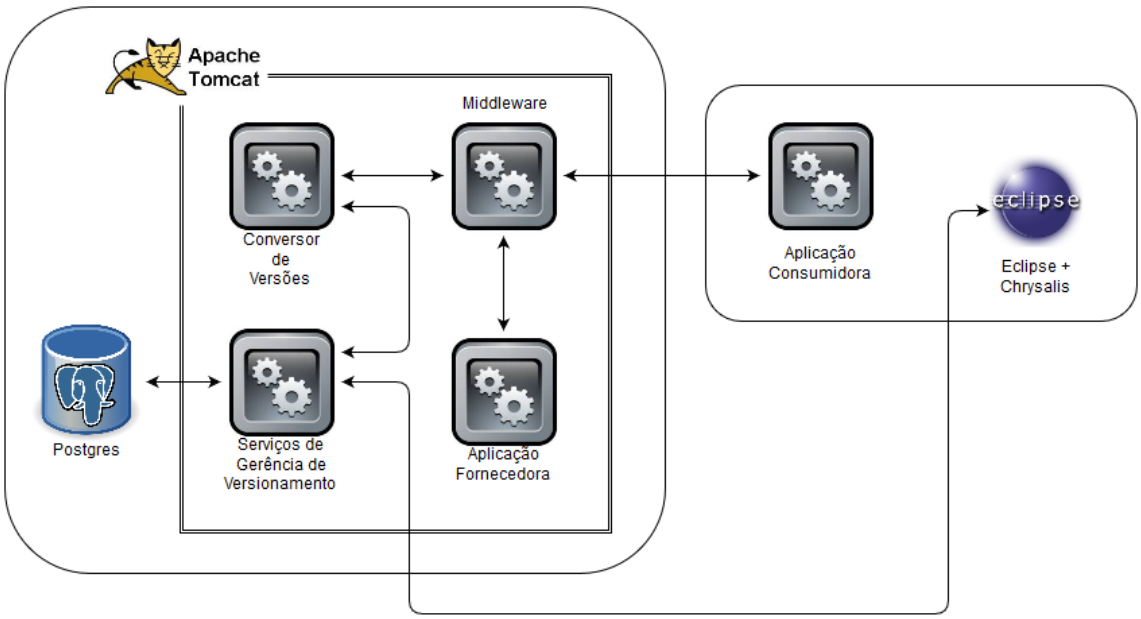
Para a prova de conceito o ambiente foi montado da seguinte forma: a aplicação consumidora do serviço foi executada em uma máquina e em outra foi implantado um servidor de aplicações Tomcat contendo as aplicações fornecedora do serviço, o *Middleware*, o Serviço de Gerência de Versionamento e o Postgres como SGBD para ser utilizado como repositório de *scripts* e contratos.

O código fonte do *Middleware*, o *plugin* Chrysalis com as modificações para acessar o Gerenciador de Versões, o repositório utilizado para salvar os contratos e *scripts*, o repositório para salvar os registros dos tempos medidos nos experimentos e o *software* para gerar essas métricas estão disponíveis no repositório Github [1].

Com este ambiente montado o WSDL selecionado foi alterado utilizando a ferramenta Eclipse com o *plugin* Chrysalis. Essas alterações foram as mesmas que o projeto a qual esse WSDL faz parte sofreu em umas de suas evoluções. Após a geração dos *scripts* XSLT das modificações, eles foram enviados ao repositório de versões através da aplicação Serviços de Gerência de Versionamento. Na Figura 5.1 é mostrado como ficou o ambiente para a prova de conceito.

[1] https://github.com/davidsfranca/dadosprojetosmestrado

Figura 5.1 - Ambiente para a prova de conceito.



Fonte: Produção do autor

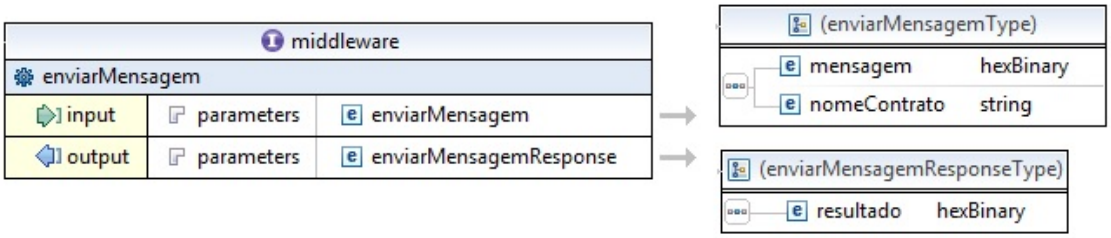
5.2 *Middleware*

Para a implementação e avaliação do modelo de arquitetura proposto foi criada uma aplicação chamada *Middleware*. Ela faz o papel de um ESB (*Enterprise Service Bus*), comumente utilizado na distribuição de mensagens e aplicações de diretivas e políticas de acesso. Para este trabalhou optou-se utilizar esta aplicação mediadora ao invés do ESB, pois apenas será utilizado o serviço de entrega de mensagens e o acesso ao código fonte desta aplicação facilitou a inserção de pontos de registro para as medições da avaliação. Esta aplicação mediadora poderia ser substituída por um ESB que tenha funções de entrega de mensagens e aplicações de funções sobre as requisições.

Utilizando a ferramenta Altova (2010) para modelagem de documentos XML *Schema* foi criada uma interface de serviços que provê um serviço contendo apenas uma operação chamada de `enviarMensagem`. A aplicação foi implementada seguindo esta interface através da ferramenta Eclipse (2016), importando o arquivo *Schema* produzido. O Eclipse gera o *skeleton* do serviço deixando a cargo do desenvolvedor apenas criar a lógica de negócios de cada operação dentro do serviço.

Na Figura 5.2 tem-se a estrutura do documento XML *Schema* criado para ser o contrato do serviço do *Middleware*. A operação `enviarMensagem` necessita de um conjunto de dados de entrada que contém a mensagem propriamente dita e o nome do contrato utilizado pela aplicação.

Figura 5.2 - XML *Schema* para criação do *Middleware*.

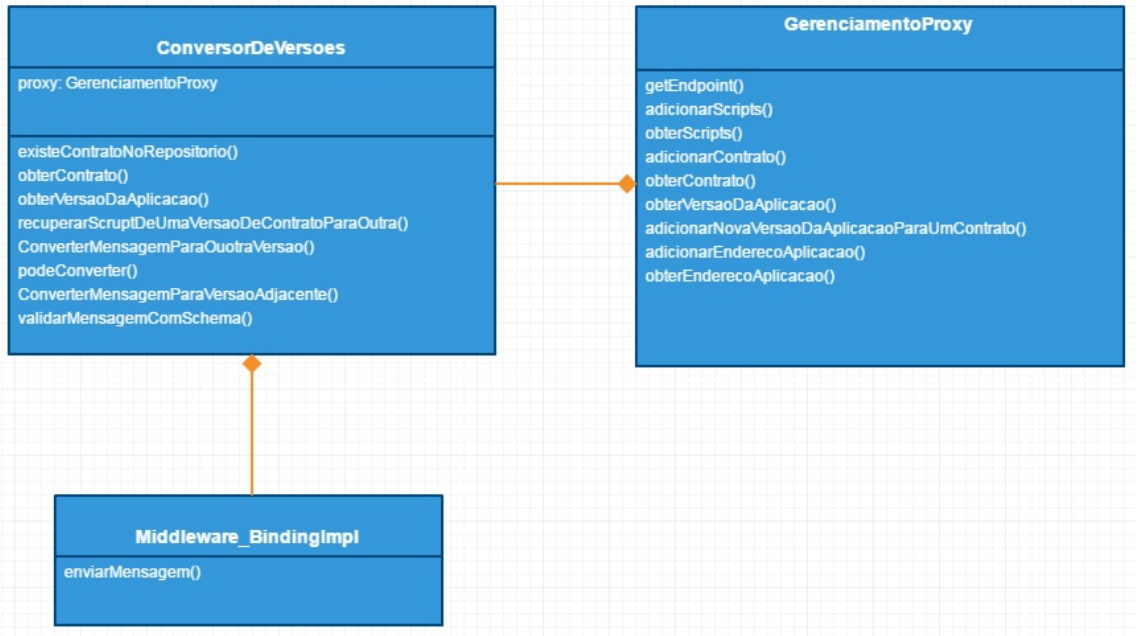


Fonte: Produção do autor

Com o contrato de serviços definido é possível gerar a estrutura do serviço e colocá-lo em funcionamento. Na Figura 5.3 são apresentadas as classes de serviço do *Middleware* e do seu componente interno o Conversor de Versões que será detalhado na seção a seguir.

A classe `MiddlewareBindingImpl` é responsável por receber a mensagem, acessar o conversor de versões para possíveis conversões e enviar a mensagem para o destinatário. A Classe `ConversorDeVersoes` é responsável por executar os métodos de conversão sobre os dados da mensagem. Ele possui métodos para verificar versões de aplicações, contratos e aplicar os *scripts* nas mensagens. Ainda existe a classe `GerenciadorProxy` que é responsável por acessar o serviço *web* do Gerenciador de Versões e obter todos os dados armazenados no repositório de versões. Também possui métodos para incluir novos *scripts* e contratos no repositório.

Figura 5.3 - Diagrama de classes de serviço do *Middleware*.



Fonte: Produção do autor

Na Figura 5.4 é possível ver parte do trecho de código gerado pelo Eclipse com a implementação da lógica de negócio do método `converterMensagem` do *Middleware*. Da linha 4 à 19 foi criada a implementação do serviço que recebe uma mensagem, faz as conversões, se necessário, e envia a mensagem ao seu destinatário. A implementação do *Middleware* possui uma instância do Conversor de Versões pois é ele quem sabe tratar se a mensagem precisa ser convertida ou não e como acessar os serviços de Gerenciamento de Versões.

5.3 Conversor de Versões

Este componente que fica junto ao *Middleware* possui toda a lógica de conversões entre versões de um contrato. Ele foi implementado para ter a capacidade de acessar os Serviços de Gerência de Versionamento para recuperar *scripts* de conversões caso seja necessário. Ele também é capaz de aplicar esses *scripts* sobre a mensagem e devolver ao *Middleware* para que ele entregue ao consumidor.

Para leitura da mensagem foi utilizada a biblioteca XSL Transformations API. Na Figura 5.5, das linhas 11 à 16, é onde acontece a aplicação dos *scripts* recuperados do repositório. Ao final, na linha 18, a mensagem é retornada ao *Middleware* com

Figura 5.4 - Implementação do *skeleton* do serviço gerado pelo Eclipse.

```
1  private ConversorDeVersoes conversorDeVersoes = new
       ConversorDeVersoes();
2
3      public byte[] enviarMensagem(byte[] mensagem, String
           nomeContrato, int versaoRementente) throws java.rmi.
           RemoteException {
4          RequestWs requestWs = new RequestWs();
5          ValidarEstruturaDasEntidadesRequest request = new
               ValidarEstruturaDasEntidadesRequest();
6          int versaoDestinatario = conversorDeVersoes.
               obterVersaoDaAplicacao("aplicacaoB", nomeContrato);
7          if(!conversorDeVersoes.podeConverter(nomeContrato,
               versaoRementente, versaoDestinatario)){
8              String erro = "Nao foi possivel entregar a mensagem.
                   Faltam scripts de conversao.";
9              try {
10                 return erro.getBytes("UTF-8");
11             } catch (UnsupportedEncodingException e) {
12                 e.printStackTrace();
13             }
14         }
15         mensagem = conversorDeVersoes.
               converterMensagemParaOutraVersao(mensagem, nomeContrato,
               versaoRementente, versaoDestinatario);
16         request.setMundo(mensagem);
17         requestWs.setEndpoint(conversorDeVersoes.
               obterEnderecoDaAplicacao("aplicacaoB"));
18         ValidarEstruturaDasEntidadesResponse response = requestWs.
               execute(request);
19         return response.getResult().getBytes();
20     }
21 }
```

Fonte: Produção do autor

as devidas modificações.

Além deste método, o Conversor de Versões possui outros como `converterMensagemParaOutraVersao`, que converte uma mensagem de certa versão para qualquer outra; `podeConverter`, que verifica se existem *scripts* para conversão; `validarMensagemComSchema`, que após a aplicação dos *scripts* verifica se a mensagem está no formato esperado. Possui ainda métodos de acesso aos Serviços de Gerência de Versionamento.

Figura 5.5 - Implementação do Conversor de Versões.

```
1  public byte[] converterMensagemParaVersaoAdjacente(byte[] mensagem,
       String nomeContrato, int versaoAnterior,
2          int versaoAtual) {
3        byte[][] scripts;
4        try {
5           scripts = proxy.obterScripts(nomeContrato, versaoAnterior,
                versaoAtual);
6           TransformerFactory tFactory = TransformerFactory.
               newInstance("net.sf.saxon.TransformerFactoryImpl", null
               );
7           byte[] dados = null;
8           InputStream entrada = new ByteArrayInputStream(mensagem);
9           ByteArrayOutputStream saida = new ByteArrayOutputStream();
10          for (int i = 0; i < scripts.length; i++) {
11             Transformer transformer = tFactory
12                   .newTransformer(new StreamSource(new
                        ByteArrayInputStream(scripts[i])));
13             transformer.transform(new StreamSource(entrada), new
                   StreamResult(saida));
14             dados = saida.toByteArray();
15             entrada = new ByteArrayInputStream(dados);
16             saida = new ByteArrayOutputStream();
17          }
18          return dados;
19       } catch (TransformerException | RemoteException e) {
20          e.printStackTrace();
21       }
22       throw new RuntimeException();
23 }
```

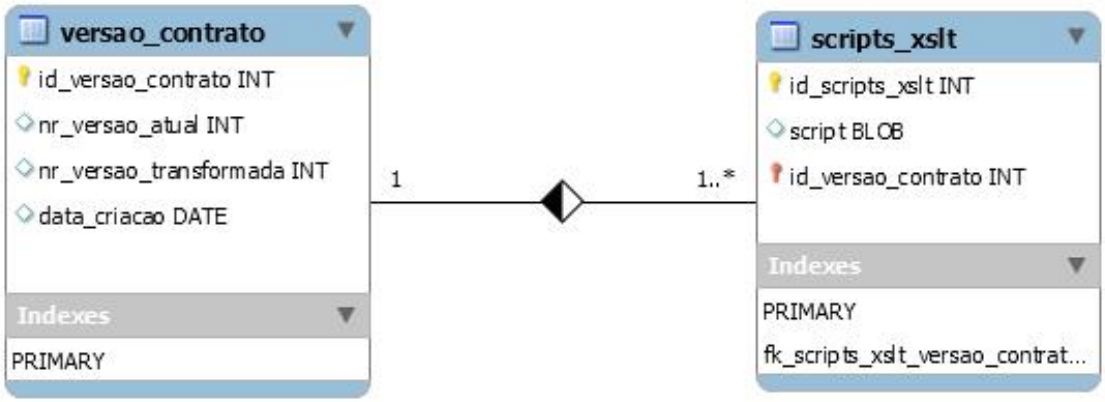
Fonte: Produção do autor

5.4 Repositório de Versões

Este componente é o responsável por armazenar os arquivos de contratos e *scripts* de conversão de cada versão gerada. Para uma conversão entre duas versões é necessário armazenar o contrato das duas versões existentes, os *scripts* da versão anterior para a atual (pode ser que existam mais de um arquivo de *script* para ser aplicado) e os *scripts* da versão atual para a anterior. Para isso utilizou-se o modelo de banco de dados representado na Figura 5.6.

Este modelo foi implementado sobre o SGBD PostgresSQL (2015) que é facilmente acessível pela linguagem Java utilizando um conector disponibilizado pelos implementadores. Não há qualquer requisito do modelo que obrigue o uso deste banco de dados podendo ser substituído por qualquer outro a escolha do implementador. O Uso do Postgres foi apenas para a prova de conceito Na Figura 5.6 está descrito o

Figura 5.6 - Modelo Entidade Relacionamento do Repositório de Versões.



Fonte: Produção do autor

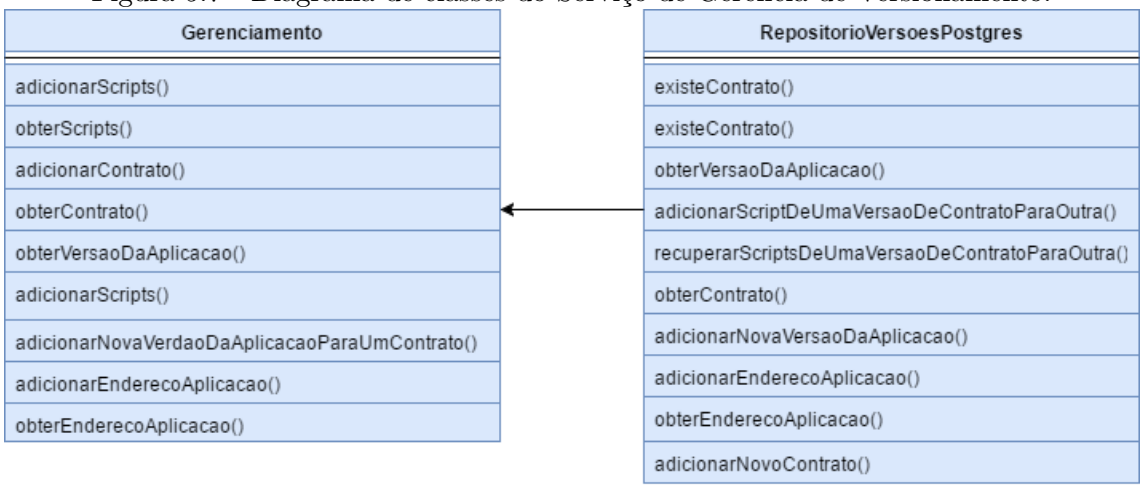
Modelo Entidade-Relacional implementado onde a tabela **versao_contrato** armazena registros de contratos como nome do contrato, a sua versão atual e para qual versão é a transformação. Esta se liga à tabela **scripts_xslt** que armazena os arquivos de *scripts* de conversão através de uma chave estrangeira. Pode haver vários registros na tabela **scripts_xslt** ligadas a uma mesma versão do contrato. Isto é justificado porque uma mudança no contrato poderá gerar vários arquivos diferentes dependendo da ferramenta que se utiliza para a mudança no XML *Schema*.

5.5 Serviços de Gerência de Versionamento

Este componente é responsável por receber, via serviço *web*, requisições da Ferramenta de Refatoração do Formato de Documento XML e acessar o Repositório de Versões. A implementação deste serviço abrange um conjunto de funções capazes de reconhecer as novas requisições e verificar se é possível incluir no repositório novas versões de contratos e *scripts*. Ele também recupera os *scripts* necessários para conversão de versões entre mensagens. A Figura 5.7 mostra o diagrama de classes do componente Serviço de Gerência de Versionamento.

Na Figura 5.8 é apresentado o método do Serviço de Gerência de Versionamento `adicionarScriptDeUmaVersaoDeContratoParaOutra` que é utilizado para enviar ao Repositório de Versões novos *scripts* gerados na mudança de um contrato. Este método adiciona os *scripts* sem se preocupar se já existem tais arquivos no Repositório de Versões. Esta verificação é feita por outros métodos implementados neste serviço e que também verificam se existe uma cópia do contrato em questão para

Figura 5.7 - Diagrama de classes do Serviço de Gerência de Versionamento.



Fonte: Produção do autor

posterior comparação quando uma mensagem é convertida.

Figura 5.8 - Implementação do método de adição de *scripts* da Gerência de Versionamento.

```
1  public String adicionarScriptDeUmaVersaoDeContratoParaOutra(
2          Transformacao transformacao) {
3      try {
4          con.setAutoCommit(false);
5          String  sql = "INSERT INTO script (nome_contrato,
               versao_anterior, versao_atual, arquivo)" + "VALUES
               (?,?,?,?)";
6          for(byte[] file :transformacao.getScripts()){
7              PreparedStatement pstmt = con.prepareStatement(sql);
8              pstmt.setString(1, transformacao.getNomeContrato());
9              pstmt.setInt(2, transformacao.getVersaoAnterior());
10             pstmt.setInt(3, transformacao.getVersaoAtual());
11             LargeObjectManager lobj = ((org.postgresql.PGConnection
                   )con).getLargeObjectAPI();
12             int oid = lobj.create(LargeObjectManager.READ |
                   LargeObjectManager.WRITE);
13             LargeObject obj = lobj.open(oid, LargeObjectManager.
                   WRITE);
14             byte buf[] = new byte[2048];
15             obj.write(file);
16             obj.close();
17             pstmt.setInt(4, oid);
18             pstmt.executeUpdate();
19             con.commit();
20             pstmt.close();
21         }
22     } catch (SQLException e) {
23         e.printStackTrace();
24     }
25     return "Scripts adicionados com sucesso";
26 }
```

Fonte: Produção do autor

Na Figura 5.9 temos a implementação do método responsável por obter os *scripts* quando solicitado pelo Conversões de Versões. Este método acessa o repositório de Versões e verifica se existem *scripts* armazenados para um dado contrato de uma versão para sua posterior versão. É possível que uma conversão feita pelo Conversor de Versões seja entre versões não adjacentes, por exemplo, conversão entre a versão 1 e a versão 4 de um contrato.

Figura 5.9 - Implementação do método de recuperação de *scripts* da Gerência de Versionamento.

```
1  public List<byte[]> recuperarScriptsDeUmaVersaoDeContratoParaOutra(
2          String nomeContrato, int versaoAnterior, int versaoAtual)
               {
3      List<byte[]> arquivos = new ArrayList<byte[]>();
4      try {
5          con.setAutoCommit(false);
6          LargeObjectManager lobj = ((org.postgresql.PGConnection)
               con).getLargeObjectAPI();
7          PreparedStatement ps = con.prepareStatement("select * from
               script " +
8                "where nome_contrato=? and versao_anterior=? and
                    versao_atual=?");
9          ps.setString(1,nomeContrato);
10         ps.setInt(2, versaoAnterior);
11         ps.setInt(3, versaoAtual);
12         ResultSet rs = ps.executeQuery();
13         while (rs.next()) {
14             long oid = rs.getLong("arquivo");
15             LargeObject obj = lobj.open(oid, LargeObjectManager.
                   READ);
16             byte buf[] = new byte[obj.size()];
17             obj.read(buf, 0, obj.size());
18             arquivos.add(buf);
19             obj.close();
20         }
21         rs.close();
22         ps.close();
23         con.commit();
24         return arquivos;
25     } catch (SQLException e) {
26         e.printStackTrace();
27     }
28     return arquivos;
29 }
```

Fonte: Produção do autor

No total, o Conversor de Versões faz 3 solicitações ao Serviço de Gerência de Versionamento, requisitando os *scripts* da versão 1 para 2, da 2 para 3 e da 3 para 4. Aplica-os sobre a mensagem da produtora da mensagem e envia a quem se destina.

5.6 Ferramenta de Refatoração do Formato do Documento XML

Como dito anteriormente, este componente da arquitetura deve gerar os *scripts* XSLT para conversão de XMLs. A cada mudança efetuada em um documento XSD

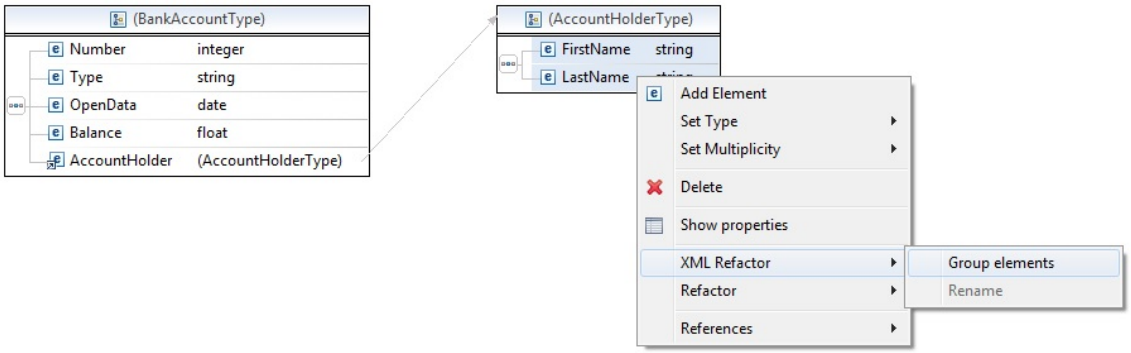
os *scripts* correspondentes para a conversão dos XMLs também devem ser gerados. Esse componente deve fornece um conjunto de tipos de refatoração que refletem possíveis evoluções de um documento XSD.

Para a implementação utilizou-se a ferramenta Chrysalis (CASTRO, 2012; SALERNO et al., 2010; DUARTE, 2010; SALERNO, 2009), um *plugin* para a plataforma de desenvolvimento Eclipse. Ela provê uma forma de refatorar um documento XSD de forma simples e ainda gera os *scripts* XSLT para serem aplicados sobre os arquivos XML para que continuem válidos.

Para a implementação deste modelo foi necessário alterar a ferramenta Chrysalis de modo que seja possível utilizar os Serviços de Gerência de Versionamento para armazenar os *scripts* gerados, pois inicialmente a ferramenta salvava os arquivos localmente, não tendo a opção de se escolher a forma ou local onde serão salvos.

Na Figura 5.10 é possível ver a interface de geração de mudanças da ferramenta Chrysalis acoplada como *plugin* ao Eclipse. O Chrysalis cria um item de menu dentro do menu de propriedades do Eclipse com o nome de XML Refactor. Dentro desta opção, dependendo dos itens selecionados no XSD *Schema*, é possível escolher algumas opções. Neste caso, duas entidades foram selecionadas e foi apresentada a opção de agrupar os elementos com um novo nome.

Figura 5.10 - Interface de alterações do Chrysalis.

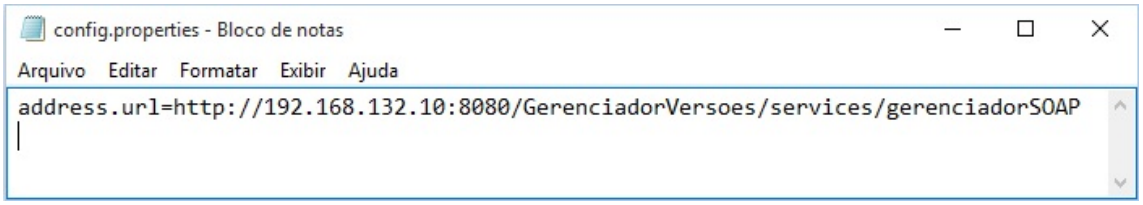


Fonte: Produção do autor

Também é possível configurar o endereço do Serviço de Gerência de Versionamento no *plug-in* do Chrysalis através de um arquivo de propriedades localizado no diretório

raiz do eclipse chamado config.properties. A Figura 5.11 exemplifica este arquivo que contém o endereço.

Figura 5.11 - Arquivo de propriedades para configuração do Serviço de Gerência de Versionamento.



Fonte: Produção do autor

6 AVALIAÇÃO DO MODELO

Esta seção tem por objetivo apresentar uma avaliação do modelo arquitetural proposto, determinando o impacto nas aplicações, a viabilidade do modelo e o impacto de desempenho de sua adoção. Para isso serão descritos a metodologia de avaliação utilizada, a configuração do experimento e os dados que foram utilizados. Uma análise também será feita sobre os resultados obtidos frente às questões de pesquisa levantadas.

6.1 Perguntas de pesquisa

Segundo Choi et al. (2007) em seu trabalho sobre qualidade de métricas de serviço, são indicados 6 pontos em que um serviço SOA pode ser medido e avaliado que demonstram a sua qualidade como provedor de um serviço e que são: Disponibilidade, Desempenho, Confiabilidade, Descoberta dinâmica, Adaptabilidade dinâmica e Modularidade dinâmica.

Para a avaliação do modelo selecionou-se 3 questões relevantes para a validade do modelo e sua aplicação, de acordo com os pontos levantados por Choi et al. (2007):

- **(Q1)** A arquitetura é capaz de gerar as transformações necessárias nas mensagens e de forma transparente para as aplicações que a utilizam (Confiabilidade)?

Ao se tentar responder esta pergunta poderá se saber se a arquitetura apresentada é realmente eficaz em certos casos e quais são os casos em que ela não é aplicável.

- **(Q2)** O que é preciso mudar nas aplicações para que possa se adotar a arquitetura (Adaptabilidade dinâmica)?

Essa pergunta poderá responder o que é necessário se alterar, em termos de código fonte, para que as aplicações consumidora e fornecedora do serviço possam utilizar a arquitetura proposta.

- **(Q3)** Qual o impacto no desempenho das aplicações (Desempenho)?

A resposta a esta pergunta pode revelar o quão custoso é para efetuar as verificações e alterações nas mensagens antes de serem enviadas a quem irá consumi-las. Esse ponto é importante para que um arquiteto possa avaliar se em sua aplicação o ganho em flexibilidade e interoperabilidade compensa a perda em desempenho.

6.2 Estudo de caso

Para este estudo de caso, três cenários foram montados, dois deles utilizando o modelo proposto e um cenário sem a utilização do modelo para servir de base de comparação.

Como requisito para o estudo de caso buscou-se uma aplicação real onde fosse possível o acesso ao código fonte de um fornecedor e de um consumidor do serviço onde também se tivesse acesso ao histórico de mudanças do contrato do serviço. Dessa forma, foi utilizado um projeto real da Força Aérea Brasileira(FAB), que possui duas aplicações que se comunicam via serviços *web*, porém este estudo de caso e sua avaliação foram feitos em um ambiente controlado.

A alteração efetuada no XSD realmente ocorreu durante a evolução das aplicações e foi reproduzida para simular um cenário real de mudanças. Uma aplicação que possui um serviço chamado `MundoExportadoDoCenario` que recebe como parâmetro uma *String* informando a versão da entidade `Mundo` solicitada e tem como retorno esta entidade que possui 77 subentidades.

A alteração efetuada consistiu em dois *renames* em uma dessas 77 entidades que passou de `DesdobramentoSMOAType` para `DesdobramentoEsquadraoType` e de `TipoDesdobramentoSMOA` para `TipoDesdobramentoEsquadrao`. Com estas mudanças o resultado retornado ao consumidor do serviço não é mais válido, pois ele ainda espera receber a entidade com o nome `DesdobramentoSMOAType` e `TipoDesdobramentoSMOA`. Tem-se assim um exemplo real de mudanças que ocorreram em aplicações reais, onde uma alteração inviabiliza a continuidade de consumo de um serviço. As Figuras 6.1 e 6.2 mostram como era e como ficou o trecho desse XSD alterado pelo fornecedor do serviço.

A seguir tem-se a descrição dos três cenários montados para se verificar a efetividade, viabilidade e desempenho do modelo de arquitetura proposto bem como alguns exemplos de XML enviados pelas aplicações.

6.2.1 Cenário 1: aplicação consumidora acessando diretamente o fornecedor (sem o modelo).

Neste cenário a aplicação consumidora acessa diretamente o serviço disponibilizado pelo fornecedor, enviando uma *String* com a versão do `Mundo`. As aplicações compartilham o mesmo contrato. É apresentado na Figura 6.3 como o cenário foi criado.

Figura 6.1 - XSD da FAB antes da alteração.

```
<xs:complexType name="DesdobramentoSMOAType">
    <xs:complexContent>
        <xs:extension base="DesdobramentoType">
            <xs:sequence>
                <xs:element name="TipoDesdobramentoSMOA" type="TipoDesdobramentoSMOAEnum"/>
                <xs:element name="IdAerodromoDestino" type="xs:string"/>
                <xs:element name="IdEsquadraoASerCriado" type="xs:string" minOccurs="0"/>
                <xs:element name="IdDesdobramentoTripulacao" type="xs:string" minOccurs="0"/>
                <xs:element name="QuantidadeAeronaves" type="xs:int"/>
                <xs:element name="TripulacoesAdicionais" type="xs:boolean">
                    <xs:annotation>
                        <xs:documentation>boolean</xs:documentation>
                    </xs:annotation>
                </xs:element>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Fonte: Produção do autor

Figura 6.2 - XSD da FAB depois da alteração.

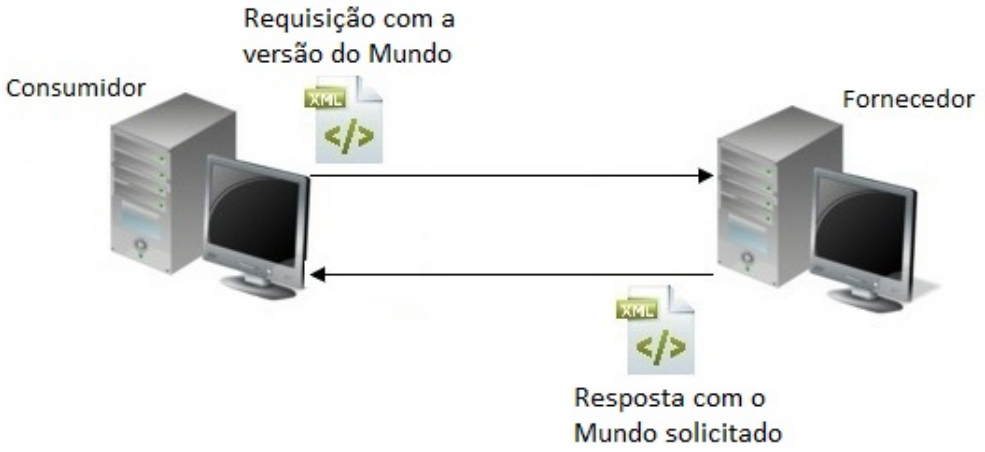
```
<xs:complexType name="DesdobramentoEsquadraoType">
    <xs:complexContent>
        <xs:extension base="DesdobramentoType">
            <xs:sequence>
                <xs:element name="TipoDesdobramentoEsquadrao" type="TipoDesdobramentoEsquadraoEnum"/>
                <xs:element name="IdAerodromoDestino" type="xs:string"/>
                <xs:element name="IdEsquadraoASerCriado" type="xs:string" minOccurs="0"/>
                <xs:element name="IdDesdobramentoTripulacao" type="xs:string" minOccurs="0"/>
                <xs:element name="QuantidadeAeronaves" type="xs:int"/>
                <xs:element name="TripulacoesAdicionais" type="xs:boolean">
                    <xs:annotation>
                        <xs:documentation>boolean</xs:documentation>
                    </xs:annotation>
                </xs:element>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Fonte: Produção do autor

6.2.2 Cenário 2: aplicações utilizando o modelo proposto, mas sem conversões de mensagens.

Neste cenário a aplicação consumidora envia sua requisição ao *Middleware* que verifica a necessidade de transformação da mensagem antes de enviar ao fornecedor do serviço. Como os parâmetros de requisição não mudaram, continuando apenas uma *String*, não há a necessidade de transformações, isto quer dizer que o *Middleware*

Figura 6.3 - Cenario 1: consumidor e fornecedor trocando informações de forma direta.



Fonte: Produção do autor

apenas repassa a mensagem ao fornecedor do serviço.

O fornecedor recebe a mensagem com a versão do `Mundo` e envia este `Mundo` de volta ao *Middleware* como resposta. Novamente a mensagem é analisada para verificar se necessita de transformações e é devolvida ao consumidor.

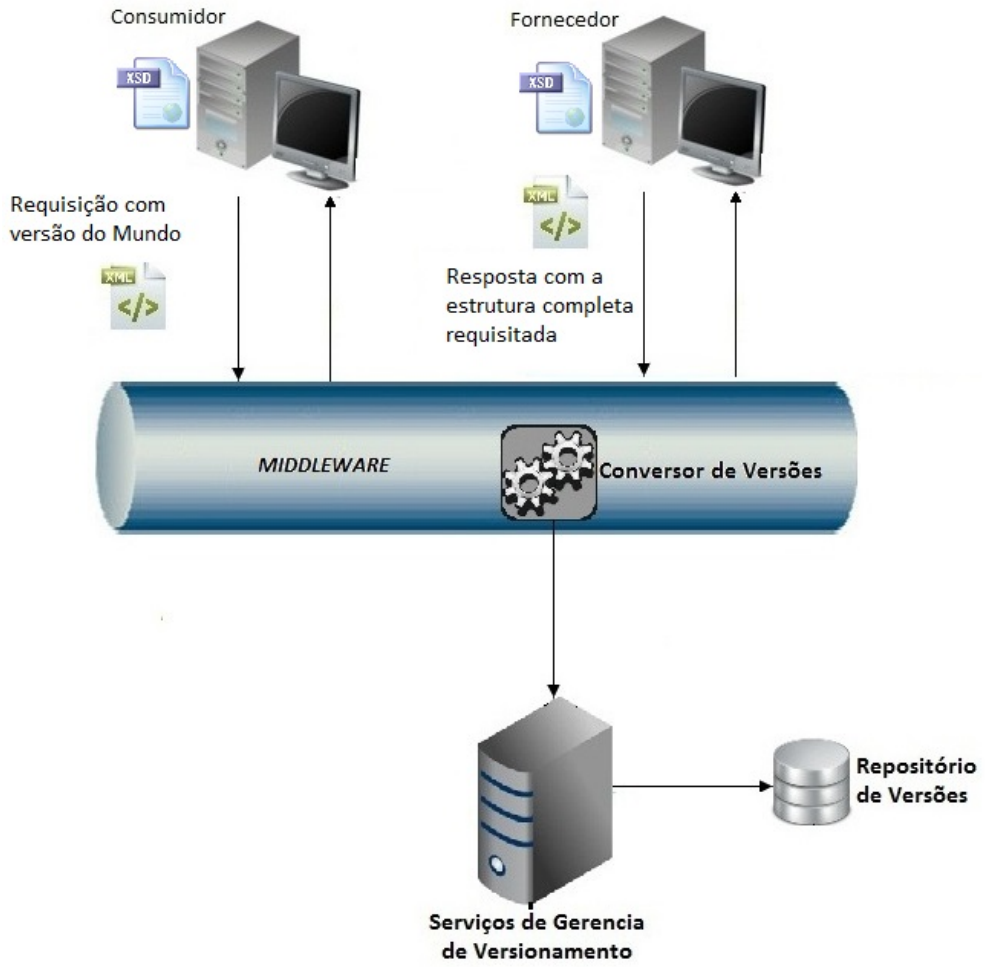
Neste cenário, o arquivo XML enviado pelo fornecedor do serviço como resposta está no mesmo formato que o consumidor espera receber, isto é, o *Middleware* compara a versão de contrato utilizada pelo fornecedor e pelo consumidor e verifica que pode enviar a mensagem sem transformações. É apresentado na figura 6.4 como o cenário foi criado.

6.2.3 Cenário 3: aplicações utilizando o modelo proposto com conversões de mensagens.

Neste cenário a aplicação consumidora envia sua requisição ao *Middleware* que verifica a necessidade de transformação da mensagem antes de enviar ao fornecedor do serviço. Essa verificação é feita comparando as versões de contrato utilizadas pelas aplicações.

Neste cenário as aplicações estão utilizando versões de contratos diferentes e necessitam de transformações na troca de mensagens. O *Middleware* recebe a requisição do consumidor do serviço, descobre que utiliza uma versão de contrato diferente do fornecedor através do Serviço de Gerência de Versionamento e busca os *scripts* de transformações através deste mesmo componente. Como os parâmetros de requisição

Figura 6.4 - Cenario 2: consumidor e fornecedor trocando informações através do *Middleware* sem conversão.



Fonte: Produção do autor

não mudaram, continuando a ser apenas uma *String*, não há transformações a serem feitas, repassando a mensagem ao fornecedor do serviço.

A resposta do fornecedor volta ao *Middleware* e novamente é comparada as versões de contratos e como são diferentes é necessário aplicar os *scripts* pelo Conversor de Versões. Posteriormente a mensagem é devolvida ao consumidor. A Figura 6.5 mostra um trecho da mensagem que é enviada do fornecedor do serviço para o *Middleware*.

Após a aplicação dos *scripts* a mensagem deve seguir o modelo de contrato da aplicação consumidora, que ainda utiliza as entidades com os nomes de `DesdobramentoSMOAType` e `TipoDesdobramentoSMOA`. A Figura 6.6 mostra como

Figura 6.5 - Trecho da mensagem antes da conversão.

```
<DesdobramentoEsquadrao>
  <Id>ID-26/10/2016-13:56:34.527638099-R0,40036348368770747000</Id>
  <IdPartido>AZUL</IdPartido>
  <Status>FINALIZADA</Status>
  <Tipo>DESDOBRAMENTO</Tipo>
  <VersaoOrigem>ID-24/10/2016-15:50:04.944576573-R0,05118299856610187000</VersaoOrigem>
  <DataHoraCriacao>2016-10-26T13:56:34.524Z</DataHoraCriacao>
  <DataHoraSolicitacao>2016-10-27T00:51:03.676Z</DataHoraSolicitacao>
  <DataHoraAprovacao>2016-10-27T13:40:22.314Z</DataHoraAprovacao>
  <IdUsuarioCriacao>ID-24/10/2011-08:20:03.382255025-R0,56919214556159250000</IdUsuarioCriacao>
  <IdUsuarioSolicitacao>ID-24/10/2011-08:20:03.382255025-R0,56919214556159250000</IdUsuarioSolicitacao>
  <IdUsuarioAprovacao>JUIZ_A4</IdUsuarioAprovacao>
  <ObservacoesDaSimulacao>1/39 Gav/SBRP/F39 desdobrado para SBRP.</ObservacoesDaSimulacao>
  <IdItemCenarioOrigem>ID-13/10/2014-15:45:27.896996605-R0,26775645161036343000</IdItemCenarioOrigem>
  <TipoDesdobramento>ESQUADRAO</TipoDesdobramento>
  <FormaDesdobramento>IMEDIATA</FormaDesdobramento>
  <InicioDesdobramento>2016-10-27T00:00:00.000Z</InicioDesdobramento>
  <TerminoDesdobramento>2016-10-27T00:00:00.000Z</TerminoDesdobramento>
  <UtilizarTodasAsMissoesDaJanela>false</UtilizarTodasAsMissoesDaJanela>
  <TipoDesdobramentoEsquadrao>COMPLETO</TipoDesdobramentoEsquadrao>
  <IdAerodromoDestino>ID-18/09/2013-16:00:11.485849437-R0,56843253209604250000</IdAerodromoDestino>
  <QuantidadeAeronaves>0</QuantidadeAeronaves>
  <TripulacoesAdicionais>false</TripulacoesAdicionais>
</DesdobramentoEsquadrao>
```

Fonte: Produção do autor

ficou a mensagem após a aplicação dos *scripts* de conversão.

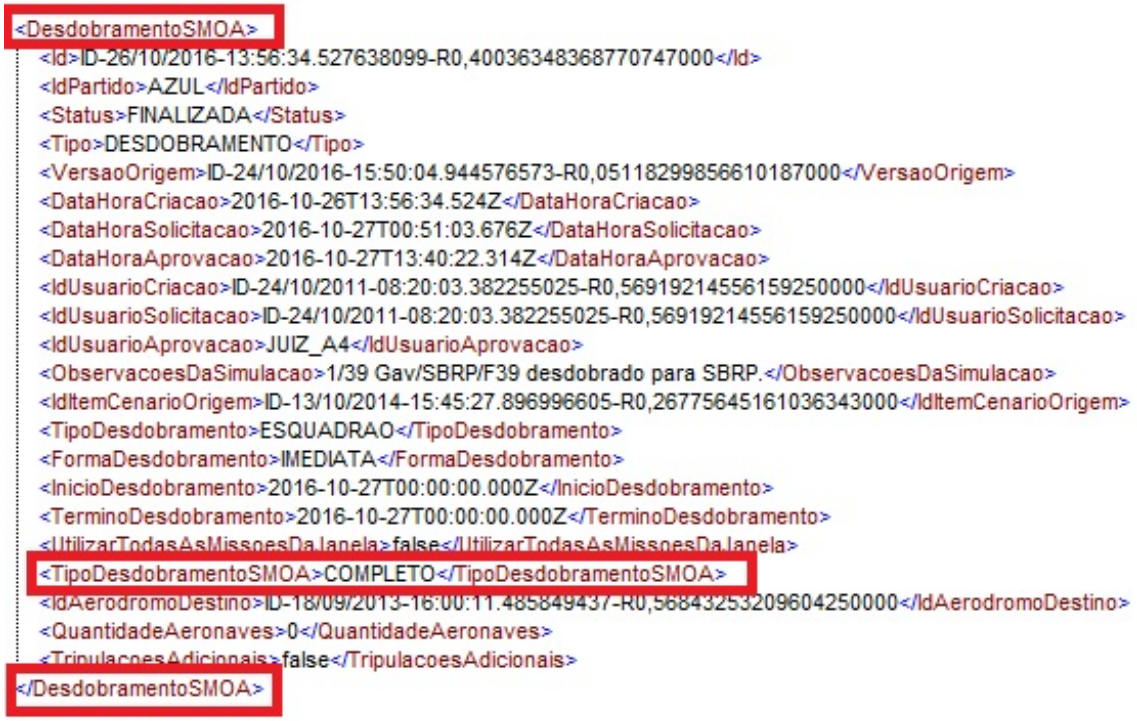
Nas linhas marcadas na Figura 6.6 os nomes das entidades foram alteradas para seguirem a versão de contrato do consumidor do serviço. Agora que a mensagem já está no formato esperado pelo consumidor o *Middleware* lhe envia a mensagem terminando o processo de requisição e resposta de forma transparente às aplicações. É apresentado na Figura 6.7 como o cenário foi criado.

6.3 Metodologia

Inicialmente cada cenário foi montado utilizando duas máquinas distintas contendo em uma delas a aplicação consumidora e na outra estão a aplicação fornecedora do serviço e a aplicação responsável pelo *Middleware* do processo. No primeiro cenário as duas aplicações trocam informações diretamente e nos outros cenários trocam informações través do *middleware* como foi explicitado nas subseções anteriores.

Para a avaliação foi utilizado um arquivo XML real baseado no XSD *Schema* da FAB e sem alterações como descrito na seção 6.2. Ele foi enviado como resposta à requisição do consumidor. Esse XSD utilizado conta com 77 tipos de entidades

Figura 6.6 - Trecho da mensagem depois da conversão.



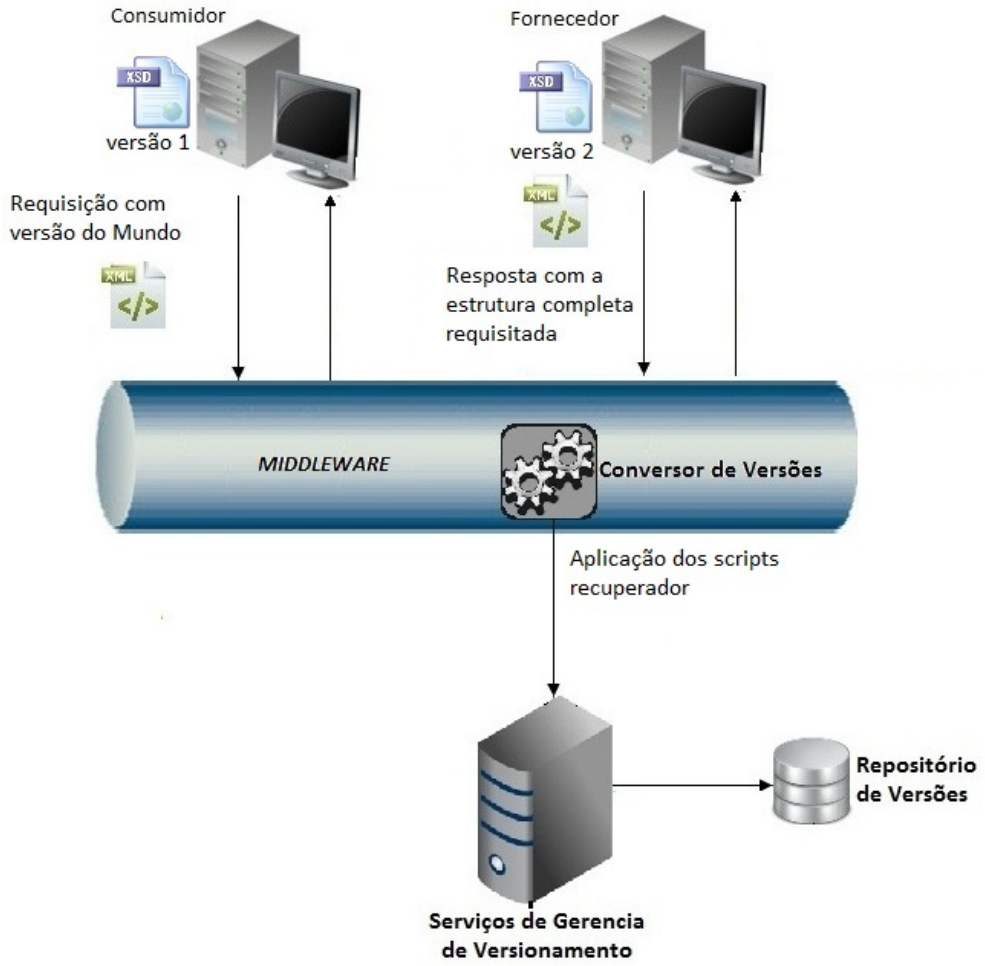
Fonte: Produção do autor

diferentes, o que resultou em um XML com aproximadamente 8000 entidades. Dessas entidades do XML existem 41 do tipo `DesdobramentoSMOAType` que foram o foco das transformações. Cada uma dessas entidades sofreu dois *rename*, resultando em 82 alterações no arquivo XML.

Através da Ferramenta de Refatoração do Formato do Documento XML foi criada uma segunda versão deste XSD *Schema* utilizado anteriormente, alterando o arquivo XSD como mostram as Figuras 6.1 e 6.2. Comparando essas figuras é possível notar que houve duas refatorações do tipo *rename* onde o nome da entidade `DesdobramentoSMOAType` foi alterada para `DesdobramentoEsquadraoType` e `TipoDesdobramentoSMOA` para `TipoDesdobramentoSEsquadrao` gerando 4 *scripts* de conversão: 1 para cada renomeação e 1 para cada ação inversa. A Figura 6.8 mostra os *scripts* gerados pela ferramenta e que foram enviados ao Serviço de Gerência de Versionamento para ser armazenenado no repositório.

Nota: Ao iniciar a avaliação de cada cenário foram armazenados previamente no Repositórios de Versões os *scripts* de transformação gerados pela alteração do XSD

Figura 6.7 - Cenario 3: consumidor e fornecedor trocando informações através do *Middleware* com conversão de mensagens.



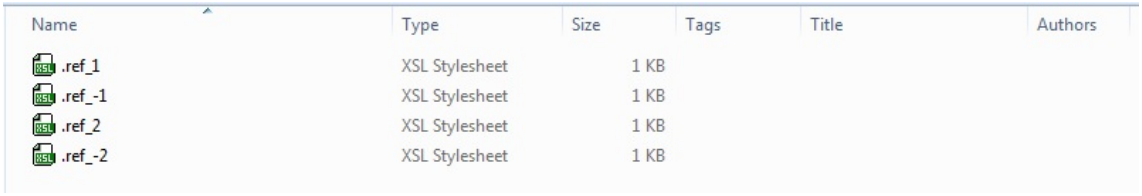
Fonte: Produção do autor

utilizado pelas aplicações.

Para o primeiro cenário onde existem apenas as aplicações consumidora e fornecedora a mensagem de requisição é enviada diretamente ao fornecedor do serviço, este processa a mensagem e retorna o resultado contendo um XML do `Mundo` à aplicação consumidora. Para a coleta de dados foram criados pontos de registro nas aplicações que marcam o momento em que a execução passou por eles:

- quando a mensagem sai do consumidor.
- quando a mensagem chega ao fornecedor.
- quando a resposta sai do fornecedor.

Figura 6.8 - Arquivos gerados pela ferramenta de refatoração.



Fonte: Produção do autor

- quando a mensagem chega ao consumidor.

Para os outros dois cenários que possuem um componente a mais, o *Middleware*, foi necessário adicionar mais os seguintes pontos de registro:

- quando a mensagem chega ao *Middleware* vinda do consumidor.
- quando a mensagem sai do *Middleware* para o fornecedor.
- quando a mensagem resposta chega ao *Middleware* vinda do fornecedor.
- quando a mensagem de resposta sai do *Middleware* para o consumidor.

Com estes pontos de registros tem-se:

- o tempo de envio da mensagem entre consumidor e *Middleware*.
- o tempo de processamento no *Middleware*.
- o tempo de envio da mensagem entre *Middleware* e fornecedor.
- o tempo de envio da resposta do fornecedor ao *Middleware*.
- o tempo de processamento da resposta do *Middleware*.
- o tempo de envio da resposta do *Middleware* ao consumidor.

Os registros obtidos foram enviados a um repositório de dados e um pequeno *software* foi construído para gerar as métricas desejadas.

Utilizando o NTP (Protocolo de Tempo para Redes) foram sincronizados os relógios das máquinas que estavam com as aplicações, permitindo uma maior acuidade

nas medições. Para isso, antes do início da avaliação, cada máquina foi sincronizada com o relógio atômico do projeto NTPBr (2016) através de sua ferramenta de sincronização.

6.4 Ambiente experimental

Para montagem do ambiente em que os cenários de avaliação foram testados utilizou-se a configuração apresentada nas próximas seções.

6.4.1 Configurações do ambiente de avaliação

Configuração do computador onde foi implantada a aplicação consumidora:

- Processador intel core i5-3210M CPU @2.50GHz 2.50GHz
- 4GB de memória RAM
- 1GB de memória RAM dedicada a aplicação

Configuração do computador onde foram implantadas as aplicações fornecedora e o *Middleware*:

- Processador Xeon W3530 @ 2.80HGZ 2.80GHZ
- 6GB de memória RAM
- 2GB de memória RAM dedicada a aplicação

6.4.2 Versões dos *softwares*

Foram utilizadas as seguintes versões de *softwares* para desenvolvimento e execução das aplicações:

- Eclipse Mars - release 4.5.0 - Implementação das aplicações.
- Eclipse Ganymede - release 3.4.2 - Implementação de novas *features* no *plugin* Chrysalis
- XMLSpy 2005 versão 2005 sp1 - Criação ou alteração dos XSD utilizados na avaliação.
- Windows 7 64 bits - Sistema operacional das máquinas utilizadas

- Postgres versão 9.4 - SGBD utilizado como repositório de contratos e *scripts*

- Axis 2 - release 1.6.3 - *Engine* utilizada para a criação de estruturas de *web services*.

- Apache Tomcat release 7.0.68 - Servidor de aplicações utilizado para fazer o *deploy* das aplicações envolvidas na avaliação.

- Java Development Kit (JDK) versão 1.7.0.79 - Utilizado como máquina virtual Java para a criação e *deploy* de todas as aplicações.

6.4.3 Rede

As máquinas estavam conectadas a um *switch* Cisco 2500 Series a uma velocidade de de 1 Gbps. A máquina que estava executando a aplicação cliente utilizava um adaptador de rede Broadcom netExtreme 57xx e a máquina que executava as aplicações *Middleware* e fornecedora do serviço estava utilizando um adaptador de rede Qualcomm Atheros ar8161/8165 Gigabit.

6.5 Dados obtidos

Nesta subseção serão apresentados os dados sobre a execução de cada cenário testado passando pela execução de cada caso, como foram as medições de desempenho, a quantificação do impacto da adoção da arquitetura proposta sobre as aplicações que consomem os serviços e uma análise sobre os dados e medições obtidos. Por fim são apresentadas algumas ameaças à validade do experimento.

6.5.1 Execução Preliminar

Primeiramente todos os cenários foram executados sem qualquer tipo de medição para verificar se as aplicações conseguiam realmente se comunicar. Posteriormente foram adicionados os pontos de medições e registros.

Foi visto que nos três cenários testados, como esperado, as aplicações conseguiram se comunicar, seja no cenário 1 onde consumidor e fornecedor de serviços se comunicam diretamente, seja nos cenários 2 e 3 em que as aplicações consumidora e fornecedora de serviços se comunicam através do *Middleware*.

6.5.2 Medições de desempenho

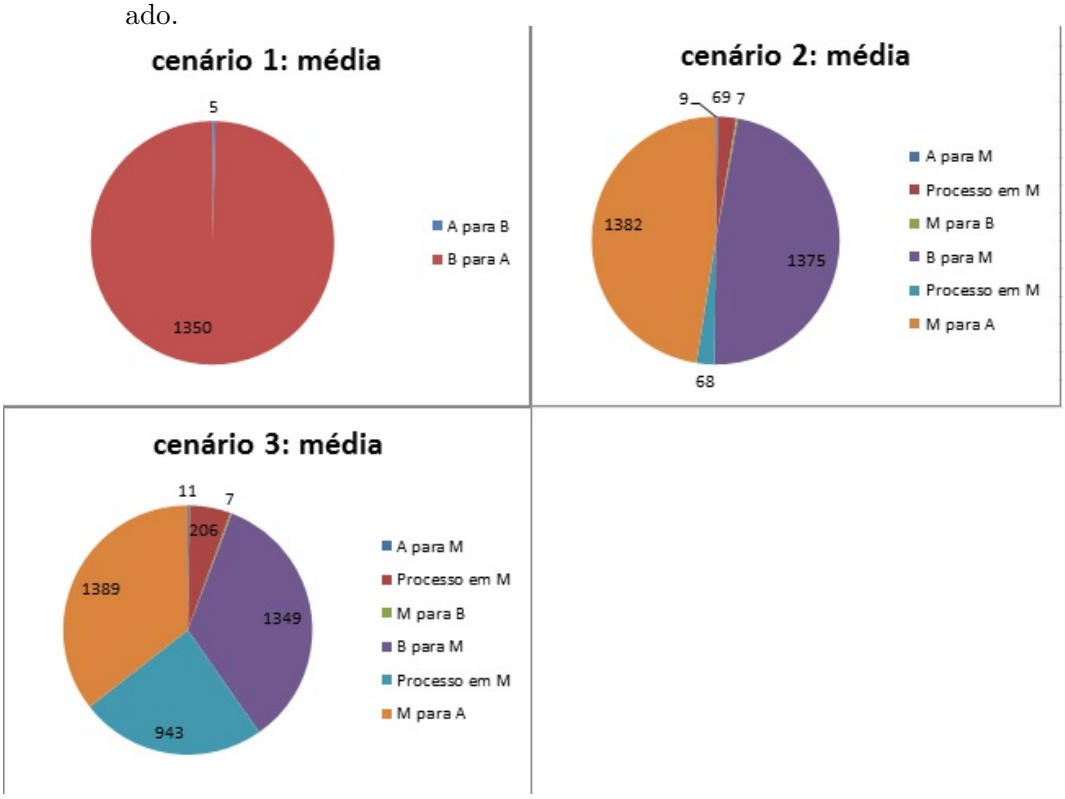
Durante os experimentos, os registros de tempo capturados foram armazenados para posterior análise. Cada um dos cenários foi executado 1000 vezes para se ter uma média dos tempos e evitar possíveis discrepâncias devido a fatores externos como oscilações na rede.

Nas figuras 6.9, 6.10 e 6.11 são apresentados os gráficos referentes aos tempos obtidos em cada fase da avaliação de cada cenário. Nos gráficos apresentados optou-se por chamar cada componente por:

- **A**: refere-se a aplicação consumidora
- **M**: refere-se a aplicação de *Middleware*
- **B**: refere-se a aplicação fornecedora do serviço

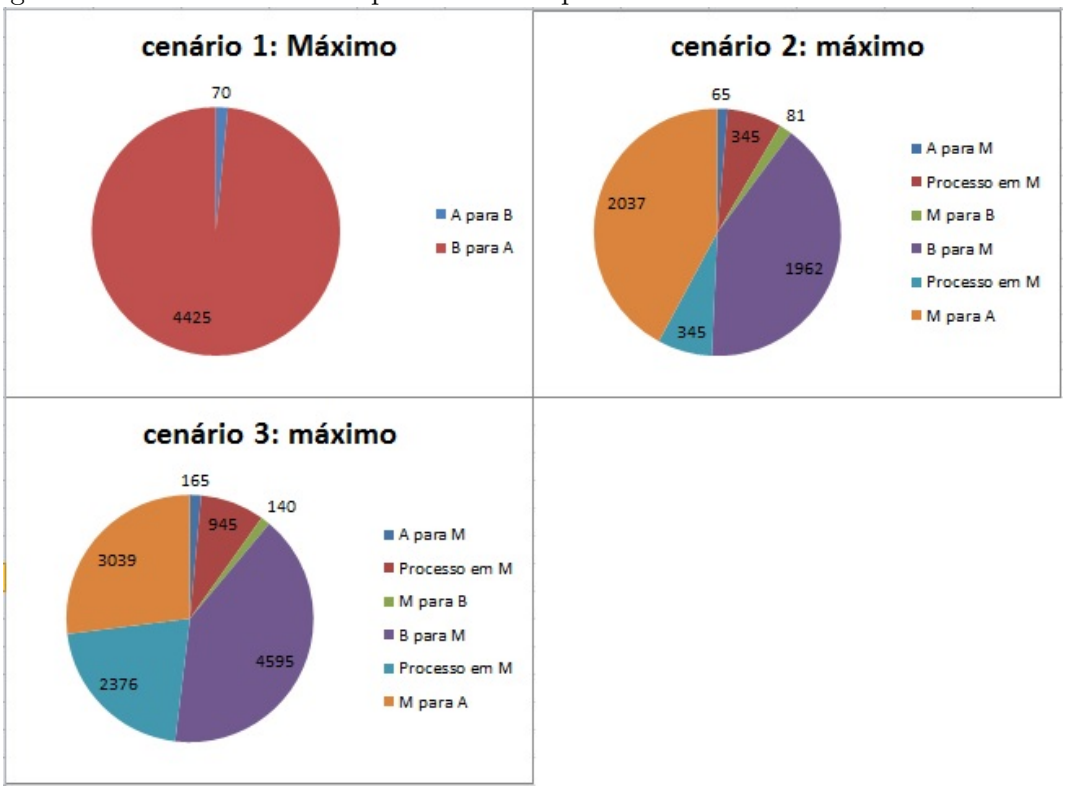
Todos os tempo apresentados nos gráficos 6.9, 6.10 e 6.11 estão em milissegundos.

Figura 6.9 - Média dos tempos (em ms) de envio e processamento em cada cenário avaliado.



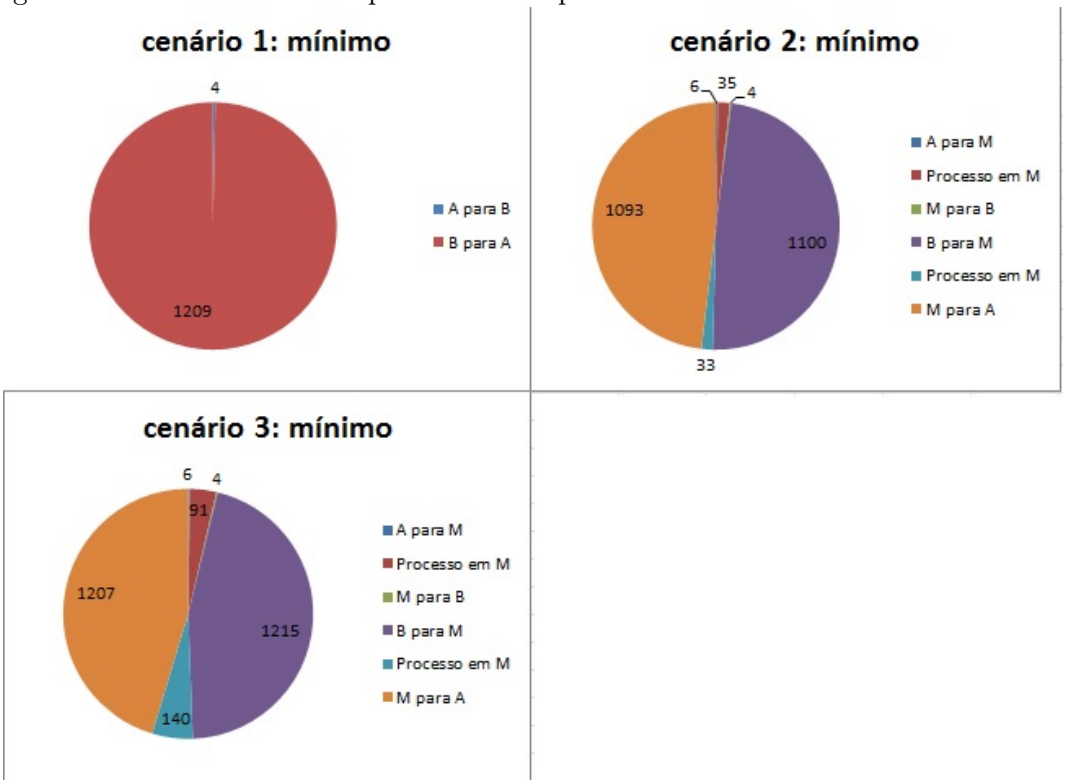
Fonte: Produção do autor

Figura 6.10 - Máximo dos tempos de envio e processamento em cada cenário avaliado.



Fonte: Produção do autor

Figura 6.11 - Mínimo dos tempos de envio e processamento em cada cenário avaliado.



Fonte: Produção do autor

A Tabela 6.1 descreve os tempos obtidos na transmissão da mensagem de B para A nos 3 cenários testados. Direto de B para A, passando pelo *Middleware* e passando pelo *Middleware* fazendo a conversão da mensagem.

Tabela 6.1 - Tempo de comunicação de B para A

Tempos de comunicação de A para B			
	cenário 1	cenário 2	cenário 3
caso mínimo	1209ms	2226ms	2562ms
caso médio	1350ms	2825ms	3681ms
caso máximo	4425ms	4344ms	10010ms

Comparando-se o cenário 1, onde uma mensagem é enviada de A para B, com o cenário 2, onde uma mensagem é enviada de A para B passando por M, pode-se avaliar que no caso de tempo mínimo, até que a mensagem contendo o XML requisitado chegue a A, o cenário 1 consumiu cerca de 99% do tempo total da comunicação, já no cenário 2, até que a mensagem chegue a A este tempo correspondeu a cerca de 98%

do tempo total de comunicação, uma diminuição no tempo total da comunicação de no mínimo 1%.

Já no caso máximo, para o cenário 1, para que a mensagem chegue a A, consumiu-se cerca de 98% do tempo total de comunicação. No cenário 2, consumiu-se cerca de 89% do tempo total. No caso máximo a diminuição da porcentagem total de tempo de toda a comunicação para que a mensagem chegue a A foi de 9%.

Para o caso médio destes mesmos cenários obteve se a porcentagem de consumo do tempo total de comunicação para que a mensagem chegue a A de cerca de 99% e 97%, respetivamente para os cenários 1 e 2.

Agora comparando-se os cenários 2 e 3, pode-se avaliar que no caso mínimo onde a mesma mensagem é enviada de B para A passando por M e no cenário 3 que existe uma conversão para outra versão de XSD. No cenário 2 obteve-se para esse envio um consumo de cerca de 98% do tempo total da comunicação e para o cenário 3 obteve-se cerca de 96% do tempo total de comunicação causando um aumento de 2% na porcentagem de tempo do total. Isso se deve ao fato de que a um aumento no tempo de processamento em M da mensagem quando enviada de A para B.

Para o caso máximo na comparação entre os cenários 2 e 3 a porcentagem de tempo de envio da mensagem até B ficou desta forma: para o cenário 2 a porcentagem foi de 89% do tempo total de comunicação e para o cenário 3 foi de 88% do tempo total de comunicação causando uma queda de 1% na porcentagem de tempo do total.

Para o caso médio, na comparação entre os cenários 2 e 3 a porcentagem de tempo de envio da mensagem até A ficou desta forma: para o cenário 2 a porcentagem foi de 97% do tempo total de comunicação e para o cenário 3 foi de 94% do tempo total de comunicação causando uma queda de 3% na porcentagem de tempo do total.

É possível notar ao comparar-se o cenário 1 que envia uma mensagem diretamente para A e o cenário 2 que passa pelo *Middleware* sem fazer conversões existe uma diminuição de 2% do tempo de total da transmissão em média e para o cenário 3 uma diminuição de 5%. Porém levando-se em conta nessa transmissão os tempos absolutos como mostra a Tabela 6.1 vemos que existe um grande aumento de tempo indicando que o processamento das mensagens pelo *Middleware* é muito custoso, pois necessita verificar se a mensagem precisa de transformações, se os *scripts* para as conversões existem no repositório e os aplicar à mensagem a ser transmitida.

6.5.3 Impacto nas aplicações

A partir do cenário 1, onde existiam duas aplicações reais trocando informações entre si foi possível verificar qual o impacto de alterações no código fonte para que elas se adequassem ao modelo de arquitetura proposto. Como se trata de *softwares* reais com código fechado, não foi possível apresentar neste trabalho o código fonte das aplicações. No entanto, pode-se descrever as alterações efetuadas em termos de número de linhas de código e número de classes inseridas, removidas e alteradas nos projetos.

Como explicitado anteriormente, a aplicação fornecedora do serviço não necessitou de alterações, logo, os dados apresentados a seguir se referem apenas a aplicação que consome o serviço. Na Tabela 6.2 é descrito o conjunto de alterações efetuadas para adequar o código fonte da aplicação consumidora em termos de linhas de código inseridas, alteradas e removidas, bem como a quantidade de classes inseridas, removidas ou alteradas.

Tabela 6.2 - Alterações do código fonte da aplicação consumidora

Alterações do código fonte da aplicação consumidora	
Nº classes adicionadas	5
Nº classes alteradas	1
Nº classes removidas	4
Linhas de código adicionadas	323
Linhas de código alteradas	25
Linhas de código removidas	1

Na Tabela 6.2 pode se ver a necessidade de poucas alterações nas linhas de código existentes, mas uma considerável inclusão de novas classes e linhas de código. As novas classes adicionadas referem-se a criação da estrutura de *proxy* para acessar o novo serviço, neste caso, o *Middleware* que substituiu o *proxy* utilizado para enviar a mensagem ao fornecedor do serviço. A mensagem criada pelo consumidor não é alterada no seu conteúdo, contendo as mesmas *tags* utilizadas para o envio direto ao fornecedor, porém, no momento do envio é utilizado o novo *proxy* que aponta para o *Middleware*.

Quanto a remoção de linhas de código e classes, se deu porque a estrutura do antigo *proxy* (o que apontava diretamente para o fornecedor do serviço) ficou obsoleto, não sendo mais necessário no projeto.

6.6 Análise dos dados

Essa subseção utiliza os resultados obtidos na subseção 6.5 para responder as questões de pesquisa levantadas na subseção 6.1.

(Q1)A arquitetura é capaz de gerar as transformações necessárias nas mensagens e de forma transparente para as aplicações que a utilizam?

Verificou-se que quando as aplicações adotam a arquitetura apresentada nesta pesquisa as mensagens são entregues e devolvidas de modo quase transparente. No caso do envio é apenas necessário incluir no cabeçalho da mensagem qual a versão do XSD utilizado para que seja possível comparar com a versão utilizada no provedor do serviço. Quando uma mensagem precisa ser enviada a um provedor de serviços e ele espera receber esta mensagem em outra versão, o *Middleware* executa as transformações de modo que tanto a aplicação consumidora quanto a aplicação fornecedora não necessitem de mais quaisquer outros processos.

Isto foi visto na avaliação do cenário 3, onde a aplicação fornecedora devolveu uma mensagem em outra versão e a aplicação consumidora do serviço a recebeu de modo transparente, isto é, ela não precisa tratar o fato de que o fornecedor está utilizando outra versão do contrato. Isso mostra que a arquitetura cumpre o requisito sobre a transparência às aplicações (R2) e a execução de transformações nas mensagens (R3).

(Q2) O que é preciso mudar nas aplicações para que possam utilizar a arquitetura apresentada?

Verificou-se na obtenção dos dados durante os experimentos que a aplicação fornecedora dos serviços não necessita de alterações visto que ela recebe a mensagem no formato esperado e deve apenas responder a quem fez a chamada no mesmo formato de seu contrato.

A aplicação consumidora dos serviços necessita de algumas alterações. Nestes experimentos necessitou-se alterar o destinatário da mensagem. Ao invés de solicitar diretamente ao fornecedor do serviço, foi necessário alterar o código fonte para solicitar ao *Middleware* e incluir na mensagem qual a versão do contrato utilizado. Este parâmetro foi necessário para definir se a mensagem deve ser convertida ou não quando comparada ao contrato do fornecedor do serviço.

Como algumas alterações nas entidades dos contratos podem gerar mudanças opci-

onais, como multiplicidade de um para muitos ou uma entidade ser obrigatória ou não, pode ser que isso gere equívocos quanto a versão do contrato utilizado. Optou-se por incluir na mensagem qual a versão do contrato utilizado para prevenir esse possível erro. Seria possível remover esse parâmetro, porém custaria muito já que a mensagem deveria ser comparada com todas as versões do contrato disponíveis no repositório e ainda sim podendo gerar erros quando mais de uma versão satisfizesse o formato.

Com poucas alterações nas aplicações envolvidas: nenhuma no fornecedor do serviço e algumas alterações no consumidor, haja vista que essas mudanças no código das aplicações consumidoras devem ser feitas apenas uma vez, e caso ocorra outras alterações nos contratos de serviços, as aplicações não necessitam serem alteradas novamente. A única ressalva é que se deve primeiramente popular o repositório com todas as versões em que se deseja a conversão bem como seus *scripts*. Isso se torna mais fácil quando é utilizada a ferramenta Chrysalis para alterar os arquivos XSD que foi modificada para enviar para o Gerenciador de Versões as modificações efetuadas pelo usuário em arquivos XSD bem como seus *scripts* de conversões.

A necessidade de alterações no código-fonte da aplicação consumidora não altera em quase nada o código existente. Esse desacoplamento faz com que as classes sofram pouco impacto com a migração para a nova arquitetura. Os trechos de código a seguir exemplificam esse baixo acoplamento mostrando a mensagem sendo enviada diretamente do consumidor para fornecedor do serviço e utilizando a arquitetura proposta. As classes de domínio existentes antes da mudança não sofreram alterações sendo utilizadas da mesma forma, o que mudou foi a adição de um cabeçalho a mensagem e o envio através de outro *proxy*.

Figura 6.12 - Trecho do código do consumidor antes da alteração.

```
1 public void requisitarMundo() throws IOException {
2       MundoProxy proxy = new MundoProxy();
3       MundoExportadoDoCenarioRequest request = new
            MundoExportadoDoCenarioRequest();
4       request.setVersao("2273");
5       proxy.mundoExportadoDoCenario(request);
6   }
7 }
```

Fonte: Produção do autor

Figura 6.13 - Trecho do código do consumidor depois da alteração.

```
1  public void enviar() {
2          MiddlewareProxy proxy = new MiddlewareProxy();
3          MundoExportadoDoCenarioRequest request = new
               MundoExportadoDoCenarioRequest();
4          request.setVersao("2273");
5          byte[] yourBytes = serializarMensagem(request);
6          Middleware_ServiceLocator wsLocator = new
               Middleware_ServiceLocator();
7          Middleware_PortType ws = wsLocator.getmiddlewareSOAP();
8          proxy.enviarMensagem(yourBytes, "Contrato1.xsd");
9      }
10
11     public void adicionarHeader() {
12         SOAPHeaderElement version = new SOAPHeaderElement(
13                 "http://localhost:8080/middleware/", "versao");
14         SOAPHeaderElement numero = new SOAPHeaderElement(
15                 "http://localhost:8080/middleware/", "numero", "1");
16         version.addChild(numero);
17         ((Middleware_BindingStub) ws).setHeader(version);
18     }
19
20     public byte[] serializarMensagem(Object request) throws
           IOException {
21         ByteArrayOutputStream bos = new ByteArrayOutputStream();
22         ObjectOutput out = null;
23         byte[] yourBytes = null;
24         out = new ObjectOutputStream(bos);
25         out.writeObject(request);
26         out.flush();
27         yourBytes = bos.toByteArray();
28         bos.close();
29     }
30 }
```

Fonte: Produção do autor

(Q3) Qual o impacto no desempenho das aplicações?

Constatou-se a diminuição em termos de porcentagem do tempo de transmissão da mensagem, mas em números absolutos um aumento grande de tempo de comunicação. Isso se deve ao fato do *Middleware* necessitar verificar se as aplicações estão na mesma versão, se existem *scripts* de conversão, recuperá-los do repositório de versões e aplicá-los à todo o XML da mensagem. Em sistemas que já utilizam ESB para a distribuição de mensagem esse custo também é notado pelo simples fato de apenas receber a mensagem e processar a quem ele se destina.

No estudo de caso foi utilizado um XML grande com muitas entidades como já foi mencionado anteriormente. Com XML menores e que possuem menos dados a serem verificados e transformados este tempo de processamento pode cair, diminuindo o impacto de desempenho. Uma alternativa testada foi criar um *cache* de *scripts* recuperados do repositório de versões a fim se tentar diminuir o tempo de processamento para recuperação de *scripts* de conversão. Este teste se mostrou pouco eficaz indicando que o aumento de tempo de envio da mensagem está realmente na aplicação dos *scripts* e não na sua recuperação do repositório de versões.

Tendo todos esses indicadores de desempenho, nota-se que existe um aumento significativo de tempo na transmissão da mensagem - em média levou-se pouco mais de 3 segundos para se efetuar uma requisição, não podendo ser ignorado, porém para certos tipos de aplicações que não necessitam de transmissões de mensagens em tempo real ou que não utilizam invocações síncronas onde quem solicitou o serviço tem de esperar receber a resposta para poder fazer uma nova requisição, esta solução pode valer a pena pelos outros benefícios obtidos.

Uma ressalva quanto ao aumento do tempo de transmissão é que ele aumenta esponencialmente a cada nova versão utilizada, isto é, se as aplicações consumidora e fornecedora estão uma versão de diferença e com uma nova mudança elas passam a estar a duas versões de distância em suas implementações o tempo aumenta de forma esponencial. Esse aumento esponencial de tempo se deve ao fato de mais *scripts* terem de ser aplicados a mensagem e de forma sequencial. os *scripts* de transformação da versão da aplicação consumidora para a proxima versão são aplicados, desta versão para a próxima até que chegue na versão do fornecedor do serviço. Este é um preço que os clientes pagam por não se adequarem ao formato do fornecedor do serviço. Não deixam de transmitr as mensagem, mas perdem em desempenho.

6.7 Ameaças à validade

Para o estudo realizado foram utilizados 2 cenários onde a arquitetura proposta foi implantada, porém pode se obter resultados diferentes quando se utiliza massa de dados diferentes ou outros tipos de alterações no contrato de serviço. Essa seção discute possíveis ameaças à validade dos resultados da avaliação do modelo apresentada nesse capítulo.

Uma das ameaças à validade se refere a quantidade de fornecedores e consumidores de serviços utilizados. Nestes cenários foi utilizado apenas um fornecedor e um consumidor de serviços com apenas uma operação e uma alteração na mensagem.

Foram feitas análises sobre esses dados coletados neste contexto, porém a utilização de várias operações contendo mais alterações nas mensagens poderia mostrar resultados diferentes dos obtidos nos casos avaliados neste trabalho. Também pode ser incluído como parte dessa ameaça o fato de que outras técnicas de aplicação de *scripts* sobre as mensagem poderia resultar em medições de desempenho diferentes.

Ainda quanto a ameaça ao desempenho da arquitetura, como dito anteriormente, foram utilizadas apenas uma aplicação fornecedora e uma consumidora acessando sequencialmente o serviço disponibilizado. A utilização de múltiplas aplicações consumidoras requisitando o serviço paralelamente pode ter um impacto diferente no desempenho. A análise de desempenho de apenas um fornecedor e um consumidor pode-se mostrar diferente quando se tem mais aplicações envolvidas.

Outra ameaça está relacionada a linguagem utilizada para desenvolver e alterar as aplicações dos cenários testados. Foi utilizada a linguagem Java bem como as ferramentas e *frameworks* a ela relacionadas. A utilização de outras linguagens e outros *frameworks* para criação dos serviços *web* poderia demandar um quantidade diferente de mudanças na aplicação consumidora.

Outra ameaça à validade é a validação em um ambiente controlado onde as aplicações, servidores de aplicação e as máquinas utilizadas estavam sendo exclusivamente sendo utilizadas para os testes. Em um ambiente real outros fatores devem ser levados em consideração, pois os servidores podem abrigar inúmeras aplicações e a rede pode sofrer oscilações de tráfego, podendo alterar os valores obtidos nos tempos de respostas das aplicações.

Apesar dessas ameaças o estudo conseguiu demonstrar a ordem de grandeza do custo em desempenho da utilização da solução proposta. Como as questões apresentadas afetariam também o desempenho da invocação do serviço sem a arquitetura apresentada, acredita-se que seria possível estimar que a perda de desempenho fosse proporcionalmente equivalente. Também foi demonstrado que é possível implantar a solução nas aplicações clientes com pequenas alterações de código, apesar dessas alterações poderem ser diferentes de acordo com a linguagem e plataforma utilizada.

7 CONCLUSÃO

Neste trabalho, foi abordada a comunicação de aplicações consumidoras e fornecedoras de serviços *web*. Quando uma aplicação fornecedora de um serviço *web* necessita evoluir seu contrato de serviço,todas as aplicações consumidoras tem que se adequar ao novo contrato de serviço o que nem sempre é possível de ser feito imediatamente, inviabilizando a comunicação entre elas.

A arquitetura desenvolvida foi pensada para reduzir o impacto de alterações de contratos *web* sobre seus consumidores, desacoplando todos os componentes existentes. O modelo de arquitetura apresenta um conjunto de componentes que mantém as aplicações se comunicando, gerenciando todas as versões de contratos existentes e aplicando *scripts* de conversões nas mensagens de forma transparente às aplicações.

No modelo, existe o papel do *Middleware* que faz a intermediação de todas as mensagem transmitidas em aplicações consumidoras e fornecedoras de serviços. Este componente é capaz de receber uma mensagem, verificar a quem se destina e se necessita aplicar alguma transformação na mensagem para que chegue ao destinatário no formato por ele esperado. Para isso, o *Middleware* conta com um componente interno, desenvolvido para ser capaz de aplicar *scripts* de conversão sobre a mensagem, caso eles estejam disponíveis e formatar a mensagem em uma outra estrutura.

Ainda sobre o modelo, foi criado um repositório de versões de contratos, aplicações e *scripts* que guarda informações sobre as versões dos contrato das aplicações fornecedoras do serviço, os contratos utilizados e todos os *scripts* de conversão entre versões. Esse repositório é utilizado pelo conversor de versões para que faça as mudanças na mensagem.

Para o modelo, foi alterada a ferramenta Chrysalis que possui funcionalidades para a refatoração de um contrato e gera automaticamente os *scripts* de conversão entre as versões. A alteração permitiu que ele acessasse diretamente o repositório de versões e enviasse os arquivos gerados sem a necessidade de fazer isto manualmente.

Para a avaliação do modelo proposto foi realizado um estudo de caso com três cenários diferentes onde o primeiro é a comunicação direta entre consumidor e fornecedor do serviço, sem a utilização do modelo proposto. Ele serviu de base de comparação com os outros cenários testados. O segundo cenário contou a utilização do modelo proposto, porém sem nenhum tipo de conversão das mensagens. Apenas verificou-se se a mensagem enviada pelo remetente estava na mesma versão do destinatário.

E por fim o terceiro cenário que contou com a utilização do modelo proposto e a mensagem necessitava de conversão para ser entregue ao destinatário para que ele pudesse lê-la corretamente.

Nestes três cenários foram avaliados três requisitos: primeiro, se o modelo de arquitetura proposto era capaz de executar as transformações sobre as mensagens e entregá-las ao destinatário. Viu-se, após a avaliação, que o modelo é realmente capaz de gerar as transformações sobre as mensagens e de forma transparente às aplicações que a utilizam. Todo o processo de recebimento da mensagem pelo *Middleware*, verificação de compatibilidade de versões de contrato utilizados, recuperação de *scripts* e aplicação e a entrega da mensagem ao destinatário foram executados com sucesso. A alteração da ferramenta Chrysalis também funcionou como esperado fazendo as alterações no contrato e enviando os arquivos de *script* diretamente ao repositório de versões.

O segundo requisito avaliado foi o impacto da adoção do modelo sobre as aplicações envolvidas. Na avaliação do modelo criado foi visto que o fornecedor do serviço não sofre nenhum tipo de alteração sendo indiferente à ação do modelo. Nas aplicações consumidoras viu-se que o código-fonte sofreu poucas alterações, onde as classes existentes das aplicações quase não foram alteradas. O que foi necessário foi a inclusão de novas classes e uma mudança no momento de envio da mensagem que passou a enviá-la ao *Middleware* através de outro *proxy* e com um cabeçalho informando a versão do contrato utilizado.

O terceiro requisito avaliado foi o impacto da adoção da arquitetura no desempenho das aplicações envolvidas. Comparando-se o cenário 1, onde a comunicação era direta entre consumidor e fornecedor do serviço e os outros cenários chegou-se a conclusão de que a adoção do modelo aumenta consideravelmente o tempo de comunicação do serviço. Conclui-se que a sua adoção é mais indicada em casos em que a transmissão das mensagens não necessita ser em tempo real ou de forma síncrona. Para os outros casos, a sua adoção garante os outros benefícios do modelo como o gerenciamento das versões de contratos e aplicação de *scripts*.

Este trabalho se mostrou em alguns aspectos original no sentido de introduzir as já existentes intermediação de mensagens e aplicação de *scripts* em um só componente. Além de mostrar que sua utilização pode ser somada ao gerenciamento de contratos de versões sem intervenções humana. Todos estes pontos apresentados e o baixo impacto de sua adoção nas aplicações torna este modelo uma solução original, pois ainda não tinham sido reunidos em um só modelo de arquitetura todos estes pontos

apresentados.

7.1 Contribuições

Este trabalho apresentou as seguintes contribuições:

- criação de um modelo de arquitetura para o gerenciamento de versões de contratos de serviços *web* onde a aplicação das conversões e gerenciamento das suas múltiplas versões é feita de forma transparente às aplicações. Essa transparência é importante para a diminuição do impacto de sua adoção e original no sentido de que concatenou a aplicação de *scripts* e o gerenciamento de múltiplos contratos em um só modelo.
- implementação de prova de conceito do modelo proposto mostrando que não só conceitualmente, mas que é possível implementar o modelo proposto. Tem uma importância significativa já que mostra que é exequível e pode servir de base para novos estudos.
- estudo de caso utilizando *softwares* e mensagens reais considerando cenários distintos e uma avaliação quanto a sua eficiência e eficácia, mostrando os impactos da utilização do modelo proposto. Apresentoou os casos em que o modelo é aplicável e em quais casos esta solução não é recomendada. Esse estudo é importante para mostrar que esta solução é aplicável a *softwares* reais que estão sendo utilizados tanto no meio científico como no meio industrial.
- evolução da ferramenta Chrysalis que sofreu alterações para ser introduzida como um componente ao modelo proposto.

7.2 Trabalhos futuros

Um trabalho futuro será a implantação desde modelo em um ambiente real. Apesar do estudo abranger fornecedor e consumidor reais, todos os testes foram feitos em laboratório. Sua implantação em um ambiente real poderá trazer novas informações e dados mais confiáveis quanto a sua eficiência.

Outro trabalho futuro é a pesquisa por soluções que diminuam o impacto do aumento de tempo nas requisições. A busca por novas soluções que diminuam o tempo de requisições pode contribuir para que este modelo possa ser adotado por aplicações que exijam dados em tempo real.

Outra evolução deste trabalho é se obter uma forma de identificar a versão do contrato de uma requisição de serviço sem a necessidade de adicionar ao cabeçalho da mensagem a versão do contrato utilizada. Essa descoberta da versão sem a necessidade de inclusão no cabeçalho da mensagem contribuirá para a diminuição do impacto sobre as aplicações consumidoras que se adequam ao modelo.

REFERÊNCIAS BIBLIOGRÁFICAS

ALMEIDA, D. B. F. C.; GUERRA, E. M. Evolution of xsd documents and their variability during project life cycle: a preliminary study. In: **Computational Science and Its Applications - ICCSA 2016: 16th International Conference on**. Cham, Beijing, China: Springer International Publishing, 2016. v. 4, p. 392–406. ISBN 978-3-319-42089-9. 3

ALTOVA. **XMLSpy XML editor**. 2010. http://www.altova.com/xmlspy.html. Online; accessed 11 January 2015. 58

AMBLER, S. W.; SADALAGE, P. J. **Refactoring databases**: evolutionary database design. [S.l.]: Pearson Education, 2006. 12

BRAY, T.; PAOLI, J.; SPERBERG-MCQUEEN, C. M.; MALER, E.; YERGEAU, F. Extensible markup language (xml). **World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210**, v. 16, p. 16, 1998. 9

CASTRO, E. H. S. **Suporte automatizado de refatoração de documentos XML**. Monography — Instituto Tecnológico de Aeronáutica, Sâo José dos Campos, São Paulo - Brazil, 2012. Disponível em: <http://www.bdita.bibl.ita.br/TGsDigitais/lista_resumo.php?num_tg=63904>. Acesso em: 2016. 17, 67

CHIPONGA, K.; TARWIREYI, P.; ADIGUN, M. O. A version-based transformation proxy for service evolution. In: **Proceedings of the 6th International Conference on Adaptive Science & Technology (ICAST)**. Covenant University 10 Idioko Road: IEEE, 2014. p. 1–5. ISBN 978-1-4799-4998-4. 30, 33, 34

CHOI, S. W.; HER, J. S.; KIM, S. D. Qos metrics for evaluating services from the perspective of service providers. In: IEEE, 2007, Hong, Kong, China. **Proceedings of the e-Business Engineering (ICEBE)**. [S.l.]: IEEE, 2007. p. 622–625. 69

DAIGNEAU, R. **Service design patterns**: fundamental design solutions for soap/wsdl and restful web services. [S.l.]: Addison-Wesley, 2011. 25

DELFIM, S. M. **Papel do ESB dentro de uma arqutetura SOA**. 2009. Disponível em: <http://www.portalarquiteto.com.br/papel-do-esb-dentro-de-uma-arqutetura-soa/>. 24

DUARTE, A. P. **Ferramenta para refatoração de documentos XML**. Monography — Instituto Tecnológico de Aeronáutica, São José dos Campos, São Paulo - Brazil, 2010. Disponível em: <http://www.bdita.bibl.ita.br/TGsDigitais/lista_resumo.php?num_tg=000563802>. Acesso em: 2016. 4, 15, 67

ECLIPSE. **Eclipse IDE for java developers**. 2016. https://eclipse.org/. Online; accessed 14 December 2015. 58

ERL, T. **SOA design patterns**. [S.l.]: Pearson Education, 2008. 18, 20, 22, 23

EXTOL INTERNATIONAL. **What's the dtory with UDDI?** 2009. http://www.extol.com/blog/?p=374. Online; accessed 14 December 2016. 21

FOWLER, M.; BECK, K. **Refactoring**: improving the design of existing code. [S.l.]: Addison-Wesley Professional, 1999. 12

FRANÇA, D. S.; GUERRA, E. M. Modelo arquitetural para evolução no contrato de serviços no contexto de aplicações de comando e controle. In: SIMPóSIO INTERNACIONAL DE GUERRA ELETRONICA, 15., 2013, São José dos Campos - São Paulo, Brazil. **Proceedings...** [S.l.]: Instituto Tecnológico de Aeronáutica, 2013. p. 33. 1

FRANÇA, D. S.; GUERRA, E. M.; ANICHE, M. F. Como o formato de arquivos xml evolui. um estudo sobre sua relação com o codigo-fonte. In: **Proceedings...** [S.l.]: 3rd Workshop on Software Visualization, Evolution and Maintenance (VEM), 2015. p. 113–120. 2, 38

FRANK, D.; LAM, L.; FONG, L.; FANG, R.; KHANGAONKAR, M. Using an interface proxy to host versioned web services. In: IEEE, 2008, Honolulu, HI,USA. **Proceedings of the Services Computing, 2008. SCC'08. IEEE International Conference on**. [S.l.]: IEEE, 2008. v. 2, p. 325–332. 29, 32, 33

GIT. **Git version control documentation**. 2016. https://git-scm.com/doc. Online; accessed 14 December 2015. 2

HALL, R.; PAULS, K.; MCCULLOCH, S.; SAVAGE, D. **OSGi in action**: creating modular applications in java. [S.l.]: Manning Publications Co., 2011. 18

INFOWORLD. **UDDI is a Dead Parrot**. 2005. https://web.archive.org/web/20060103121926/http://weblog.infoworld.com/realworldsoa/archives/2005/12/uddi_is_a_dead.html. Online; accessed 14 December 2016. 22

INNOQ. **UDDI R.I.P.** 2010. https://www.innoq.com/blog/st/2010/03/uddi-r.i.p./. Online; accessed 14 December 2016. 21, 22

KALIN, M. **Java web services: up and running**: a quick, practical and thorough introduction. [S.l.]: O'Reilly Media, Inc., 2013. 1, 18

KAMINSKI, P.; LITOIU, M.; MÜLLER, H. A design technique for evolving web services. In: IBM CORP., 2006, Toronto, Ontario, Canada. **Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research**. [S.l.], 2006. p. 23. 31

KLUSCH, M.; FRIES, B.; SYCARA, K. Automated semantic web service discovery with owls-mx. In: AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 5., 2006, Hakodate, Japan. **Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems**. [S.l.]: ACM, 2006. p. 915–922. 35

LEITNER, P.; MICHLMAYR, A.; ROSENBERG, F.; DUSTDAR, S. End-to-end versioning support for web services. In: IEEE INTERNATIONAL CONFERENCE, 2008, Honolulu, HI, USA. **Proceedings of the Services Computing, 2008. SCC'08.** [S.l.], 2008. v. 1, p. 59–66. 26, 28, 29

NTPBR. **NTP.br**. 2016. http://ntp.br/estrutura.php. Online; accessed 10 July 2016. 78

OTTE, R.; PATRICK, P.; ROY, M. **Understanding CORBA (common object request broker architecture)**. [S.l.]: Prentice-Hall, Inc., 1995. 18

PAUTASSO, C.; ZIMMERMANN, O.; LEYMANN, F. Restful web services vs. big'web services: making the right architectural decision. In: ASSOCIATION FOR COMPUTING MACHINERY, 17., 2008, Beujing, China. **Proceedings of the 17th international conference on World Wide Web**. [S.l.], 2008. p. 805–814. ISBN 978-1-60558-085-2. 19

POSTGRESSQL. **PostgresSQL documentation**. 2015. http://www.postgresql.org/docs/. Online; accessed 14 January 2015. 62

ROMAN, E.; SRIGANESH, R. P.; BROSE, G. **Mastering enterprise javabeans**. [S.l.]: John Wiley & Sons, 2005. 18

SALERNO, G.; PEREIRA, M.; GUERRA, E.; FERNANDES, C. **A refactoring model for XML documents**. 2010. 3 p. 12, 67

SALERNO, G. R. **Um modelo de refatoração em documentos XML.** Monography — Instituto Tecnológico de Aeronáutica, São José dos Campos, São Paulo - Brazil, 2009. Disponível em: <http://www.bdita.bibl.ita.br/TGsDigitais/lista_resumo.php?num_tg=00553895>. Acesso em: 2016. 67

SAXON. **Saxon**: saxonica limited. 2015. http://www.saxonica.com/welcome/welcome.xml. Online; accessed 14 December 2015. 12

SCHULTE, R. **Predicts 2003**: enterprise service buses emerge. 2002. Report, Gartner, December. 24

SUBVERSION. **Apache subversion documentation**. 2016. https://subversion.apache.org/docs/. Online; accessed 14 December 2015. 2

TIDWELL, D. **Xslt**. [S.l.]: O'Reilly Media, Inc., 2008. 10, 15

VALENTINE, C.; DYKES, L.; TITTEL, E. **XML schemas**. [S.l.]: Sybex, 2002. 10

W3C. **Extensible markup language schema(XML schema)**. 2016. http://www.w3.org/XML/Schema.html. Online; accessed 14 December 2015. 10