UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

Relatórios Técnicos
do Departamento de Informática Aplicada
da UNIRIO
n° 0002/2012

Propagação de Identidade em Acesso a Dados Via Serviços Web

Felipe Leão
Leonardo Guerreiro Azevedo
Talles Santana
Fernanda Baião
Claudia Cappelli

Departamento de Informática Aplicada

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
Av. Pasteur, 458, Urca - CEP 22290-240
RIO DE JANEIRO – BRASIL

Projeto de Pesquisa

Grupo de Pesquisa Participante

Patrocínio

PETROBRAS

Relatórios Técnicos do DIA/UNIRIO, No. 0002/2012
Editor: Prof. Gleison Santos Agosto, 2012

Propagação de Identidade em Acesso a Dados Via Serviços Web

Felipe Leão, Leonardo Guerreiro Azevedo, Talles Santana, Fernanda Baião, Claudia Cappelli

Núcleo de Pesquisa e Prática em Tecnologia (NP2Tec)
Departamento de Informática Aplicada (DIA) – Universidade Federal do Estado do Rio de Janeiro (UNIRIO)

{ felipe.leao, azevedo, talles.santana, fernanda.baiao, claudia.cappelli}@uniriotec.br

Abstract. Information Security is one of the main issues in current organizations. In this context, role-based data access control and user identity propagation in n-tier architectures are foremost concerns. This work proposes a pragmatic architecture and its implementation that handles these concerns on web services systems. The implementation is based on the SOAP protocol and interceptor handlers. Experimental tests were conducted and positive results were obtained regarding the proposal effectiveness, efficiency and robustness in concurrent scenarios.

Keywords: Database security, Identity Propagation, Dependency Injection, Data Access Control, Web Services, SOAP Protocol, Authorization Rules, TPC-H Benchmark.

Resumo. Segurança de informação é uma das principais questões em organizações atuais. Neste contexto, destacam-se especialmente a necessidade de controle de acesso aos dados por perfil de usuário e a propagação da identidade do usuário em arquiteturas de múltiplas camadas. Este artigo propõe uma arquitetura para sistemas que fazem o uso de serviços web envolvendo mecanismos para propagação de identidade e aplicação de regras de autorização. A implementação desta arquitetura está baseada no uso do protocolo SOAP e de handlers. Testes experimentais foram realizados e demonstraram bons resultados em relação à eficácia, eficiência e robustez da solução proposta em cenários com concorrência no acesso aos dados.

Palavras-chave: Segurança em Banco de Dados, Propagação de Identidade, Injeção de Dependência, Controle de Acesso a Dados, Serviços Web, Protocolo SOAP, Regras de Autorização, Benchmark TPC-H.

Sumário

Figuras

1 Introdução

A segurança de informação é uma das principais questões de organizações governamentais e privadas, as maiores questões se encontram nos mecanismos de controle (ou autorização) de acesso para garantia da integridade [Sandhu *et al.*, 1996; Yang, 2009; Cali e Martinenghi, 2008; Murthy e Sedlar, 2007]. Estes mecanismos fazem com que regras de negócio do tipo assertiva de ação de autorização sejam garantidas. Segundo [BRG 2009], regra de negócio é uma declaração que define ou restringe algum aspecto de uma organização. Regras de negócio têm como objetivo definir a estrutura de um negócio ou controlar ou influenciar o seu comportamento. Em particular, uma categoria de regras de negócio é a assertiva de ação de autorização, ou **regra de autorização**, a qual restringe **quem** é permitido realizar uma **ação** na organização sobre quais **informações**.

Para que uma regra de autorização possa ser aplicada, é necessário ter as informações de quem deseja executar a operação restrita no momento em que a operação será executada. Este trabalho trata de autorização de acesso a dados. Logo, as informações do usuário que serão utilizadas para aplicar a autorização de acesso devem estar presentes no momento da execução da operação no banco de dados. Para tal, é necessário que, inicialmente, o usuário seja autenticado recebendo uma credencial com informações de segurança. Em seguida, estas informações devem ser propagadas pelas diferentes camadas existentes na aplicação até chegar ao banco de dados onde o controle de acesso aos dados será aplicado sobre a operação que o usuário deseja tratar.

O objetivo deste trabalho é propor uma abordagem para propagação de identidade que possa ser utilizada em acesso a dados via serviços, o qual corresponde ao quarto cenário de arquiteturas de aplicação descritas por Azevedo *et al.* [2009a] e Leao *et al.* [2011]. Neste cenário, a aplicação cliente invoca serviços web, executando em servidor de aplicação. Os serviços acessam a base de dados e retornam para o usuário apenas as informações que o usuário tem acesso. Para retornar que apenas as informações que o usuário tenha acesso sejam retornadas, foi implementada a execução de regras de autorização utilizando o *framework* FARBAC proposto por [Azevedo *et al.* 2010] cujos testes de implementação apresentados por [Puntar *et al.* 2011] demonstraram seu desempenho e flexibilidade em relação à solução tradicional.

A implementação da arquitetura está baseada no uso de serviços web, protocolo SOAP [Gudgin *et al.* 2007] e de *handlers* [Oracle 2008] para propagar a identidade do usuário. A principal tecnologia para implementação de serviços web é a de web services [Erl, 2005]. Um web service é uma aplicação de software identificado por uma URI (Uniform Resource Identifier), cujas interfaces e ligações (*bindings*) são capazes de serem definidas, descritas, e descobertas como artefatos XML. Um web service suporta interação direta com outros agentes de software usando trocas de mensagens baseadas em XML via protocolos baseado na internet (García *et al.* [2006] apud W3C [2004]). Para implementar os protótipos que demonstram as soluções propostas, foram consideradas duas especificações Java para desenvolvimento de web services, a JAX-RPC e a JAX-WS. A especificação JAX-RPC é um antigo padrão Java para serviços web cujo nome reflete sua ênfase em um modelo de programação de chamada remota de procedimento (*Remote Procedure Call*) para desenvolvimento de serviços web. A especificação foi renomeada para JAX-WS durante a mudança da versão 1.1 para 2.0 para refletir o fato de que a especificação agora apoia tanto modelos de programação de chamadas remotas de procedimentos quanto centralizados por mensagens (*message-centric*).

A Figura 1 ilustra o cenário de acesso a dados via serviços. Duas situações são apresentadas: (i) Aplicação cliente acessa serviço diretamente; (ii) Aplicação cliente acessa servidor de aplicação que acessa serviço. No primeiro caso, o usuário ABC, utilizando uma aplicação cliente, acessa o serviço, e este acessa o banco de dados. O usuário de conexão com o banco de dados utilizado pelo serviço é SRV1. No segundo caso, o usuário XYZ utiliza uma aplicação cliente que acessa o servidor de aplicação que invoca o serviço que acessa o dado. O serviço executando no servidor de aplicação é executado pelo usuário SRV2. O usuário de conexão com o banco de dados utilizado pelo serviço é SRV1. O dado é retornado do serviço para o servidor de aplicação e deste para a aplicação cliente.

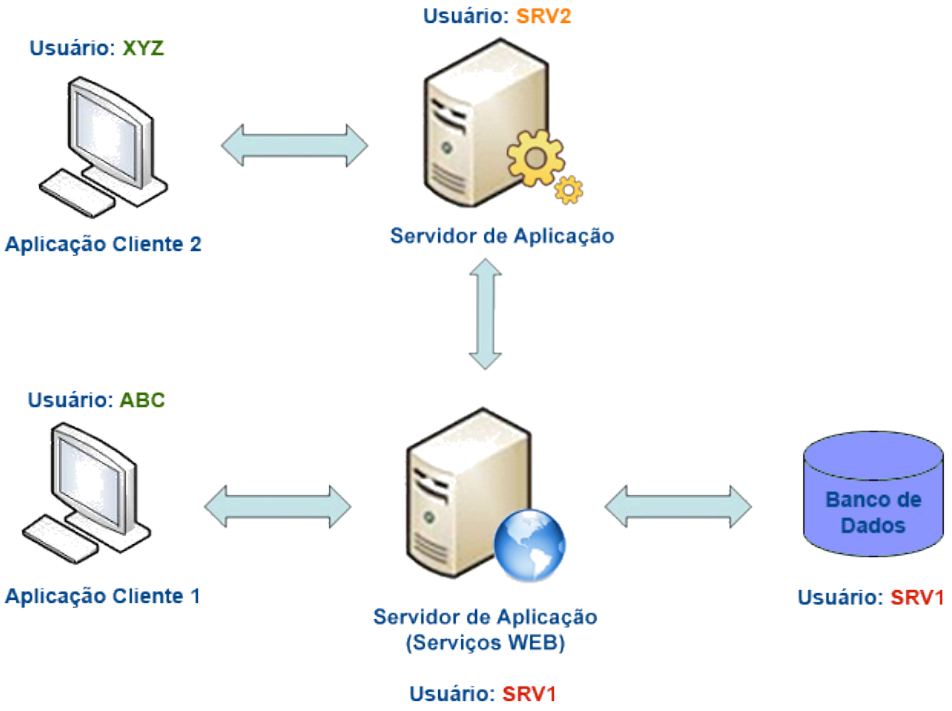

Figura 1 – Acesso a dados via serviços

As seguintes etapas foram consideradas para atender o objetivo do cenário tratado neste relatório:

- Analisar os cenários existentes e ilustrar com desenhos de arquiteturas que apresentem de forma resumida o que será tratado;
- Buscar por exemplos reais de arquiteturas para avaliação de propostas;
- Pesquisar por soluções existentes que representem o estado-da-arte e possam ser aplicadas na prática. Logo, será dada prioridade para soluções de mercado e com amplo uso;
- Avaliar em laboratório soluções estudadas;
- Analisar resultados obtidos a fim de definir a melhor abordagem para adoção.

Este relatório foi produzido pelo Projeto de Pesquisa em Autorização de Informação como parte das iniciativas dentro do contexto do Projeto de Pesquisa do Termo de Cooperação entre UNIRIO/NP2Tec e a PETROBRAS/TIC-E&P/GIDSEP.

Esse relatório está organizado em 5 capítulos, sendo o capítulo 1 a presente introdução. No capítulo 2, são apresentados os principais conceitos referentes a este trabalho. O capítulo 3 apresenta a proposta de solução. O capítulo 4 apresenta os testes experimentais realizados. O capítulo 5 discorre a respeito da transmissão das informações do usuário utilizando o protocolo HTTP ao invés do protocolo SOAP.

Finalmente, os capítulos 5 e 6 apresentam conclusões e as referências bibliográficas, respectivamente.

2 Conceituação

Esta seção apresenta os principais conceitos necessário ao entendimento da solução proposta para propagação de identidade para arquitetura que acessa dados via serviços.

2.1 Java Beans

Um Java Bean [Voss, 1996] é um componente de software reutilizável que pode ser manipulado visualmente por ferramentas de desenvolvimento de software.

Componentes de software reutilizáveis adicionam interfaces padronizadas e mecanismos de introspecção de objetos para *widgets* permitindo que ferramentas de desenvolvimento de software consultem os componentes sobre suas propriedades e comportamento. Componentes de software não precisam estar visíveis em uma aplicação executando; eles apenas precisam estar visíveis quando a aplicação é construída.

O que diferencia *beans* de uma classe Java típica é a instrospecção. Para que a introspecção funcione, as assinaturas dos métodos dentro de um *Bean* devem seguir um determinado padrão a fim de que as ferramentas de instrospecção reconheçam como *Beans* podem ser manipulados.

Na prática *beans*[1] são classes escritas em linguagem de programação Java que estão de acordo com uma convenção em particular. Eles são usados para encapsular muitos objetos em um único objeto (o *bean*), a fim de que eles possam ser passados como um único objeto ao invés de serem passados vários objetos individualmente. Um JavaBean é um objeto Java que é serializável, tem um construtor nulo, e permite acessar propriedades usando métodos *getter* e *setter*.

As convenções requeridas para *beans* são:

- A classe deve ter um construtor default (sem argumentos) público. Isto facilita instanciação dentro de *frameworks* para edição e ativação.
- As propriedades da classe devem ser acessíveis usando métodos *get*, *set*, *is* (usado para propriedades booleanas ao invés do método *get*) e outros métodos (chamados de métodos de acesso e métodos de mudança), seguindo uma convenção de nomes padrão. Isto permite facilidade para inspeção automatizada e atualização do estado do *bean* dentro de *frameworks*, muitos dos quais incluem editores customizados para vários tipos de propriedades.
- A classe deve ser serializável. Ela permite aplicações e *frameworks* salvar, armazenar e restaurar confiavelmente o estado do *bean* de uma forma independente da Máquina Virtual Java e da plataforma.

O Apêndice 1 apresenta um exemplo de Java Bean.

[1] http://en.wikipedia.org/wiki/JavaBean

2.2 JAX-RPC e JAX-WS

A API Java para serviços baseados em XML, JAX-WS, é a especificação que define um modelo de programação Java para construção de serviços web e seus clientes. JAX-WS 2.0 é a parte da especificação Java EE 5 e provê facilidades para serviços web mais fracamente acoplados e *handlers*. *Handlers* (ou manipuladores) fornecem lógica para processamento de mensagem reutilizável que pode ser injetado no caminho de invocação dos provedores do serviço e consumidores.

JAX-RPC 1.1 é um antigo padrão Java para serviços web cujo nome reflete sua ênfase em um modelo de programação de chamada remota de procedimento (*Remote Procedure Call*) para desenvolvimento de serviços web. A especificação foi renomeada para JAX-WS durante a mudança da versão 1.1 para 2.0 para refletir o fato de que a especificação agora apóia tanto modelos de programação de chamadas remotas de procedimentos quanto centralizados por mensagens (*message-centric*). JAX-WS 2.0, a tecnologia sucessora à JAX-RPC 1.1, evoluiu seu método de mapeamento usando JAXB, uma especificação técnica padrão definida pela comunidade Java.

Enquanto alguns aspectos da JAX-WS são meramente evoluções de JAX-RPC, outros são revolucionários. Por exemplo, JAX-WS não fornece mapeamento entre esquema XML e JAVA, uma característica importante da JAX-RPC. Em vez disso, JAX-WS usa outra tecnologia definida pela comunidade Java, JAXB (a arquitetura Java para XML *binding*) 2.0, para fazer seu mapeamento de dados. JAX-WS representa o modelo de invocação de Serviços Web. Ela não mais se preocupa com Java *Beans* que representam dados da aplicação; apenas foca na entrega deles para o serviço web.

O mapeamento de nomes XML para nomes Java em JAX-RPC e JAX-WS/JAXB é essencialmente o mesmo, enquanto mapear tipos simples tem leves diferenças. Butek e Gallardo [2006] apresentam uma tabela com as diferenças de mapeamento entre JAX RPC 1.1 e JAXB 2.0 para tipos simples XML.

Em um Java *Bean* gerado por JAX-RPC 1.1, não é possível saber a diferença entre:

- Um campo de elemento e um campo de atributo
- Um campo mapeado com *minOccurs*="*0*" *type*="*xsd:int*" e um campo mapeado *nillable*="*true*" *type*="*xsd:int*". Ou seja, dois campos do mesmo tipo, mas um que permite nulo e outro que tem no mínimo 0 ocorrências.
- Um campo mapeado com *type=”xsd:string”* e outro campo mapeado com *type=”xsd:string” minOccurs=”0”*

Atualmente, isso não é um problema graças ao uso de novas anotações Java pela especificação JAXB. Butek e Gallardo [2006] apresentam exemplos de mapeamento de vetores e tipos complexos feito pelas duas tecnologias, ressaltando suas diferenças, bem como uma tabela da equivalência de tipos primitivos e objetos no mapeamento de Java para esquema XML.

Embora JAX-WS não seja uma mudança radical das versões anteriores da especificação, JAX-WS não é compatível com JAX-RPC. Por conseguinte, o WebLogic Server ainda fornece suporte para o antigo modelo de programação JAX-RPC para apoiar os usuários que têm aplicações existentes usando o modelo antigo que querem atualizar para a versão mais recente do WebLogic Server. Mais a frente neste relatório será explicado as diferenças entre os métodos de criação de um projeto JAX-RPC e de um projeto JAX-WS

2.3 Arquitetura do Oracle WebLogic Server

O Oracle WebLogic Server fornece um contêiner de serviços web para abrigar código Java que processa requisições SOAP [Patrick *et al.*, 2010] e gera respostas SOAP. O contêiner implementa a especificação JAX-WS 2.1 e fornece como apoio um WS-Basic Profile 1.1 embutido para interoperabilidade entre JAX-WS e JAX-RPC. Também é integrado com o WebLogic Server's Security Service para fornecer autenticação e autorização e apoio aos padrões de segurança de serviço web como WS-Security e SAML para qualquer serviço web implantado. A Figura 2 mostra a estrutura do contêiner de web services do WebLogic Server e sua relação com alguns outros subsistemas do WebLogic Server.

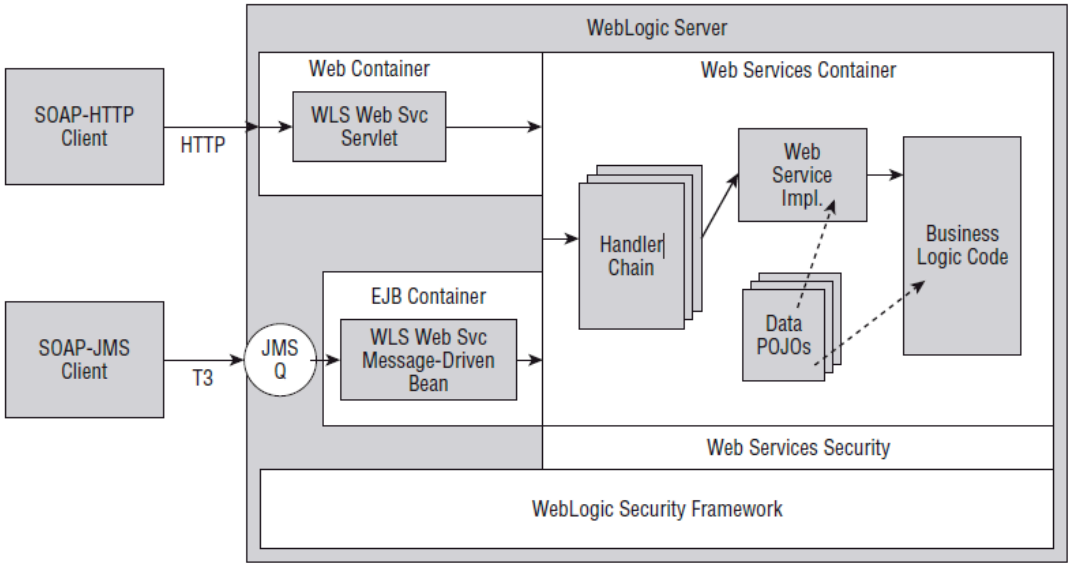

Figura 2 – Arquitetura Oracle WebLogic Server

O contêiner de serviços web fornece dois mecanismos de transporte para invocação de serviços: SOAP sobre HTTP e SOAP sobre JMS. JMS[2] é a API Java para *middleware* orientado a mensagens e através dela componentes baseados em J2EE podem criar, enviar, receber e ler mensagens. Ambos os mecanismos suportam segurança na camada de transporte usando SSL. Requisições SOAP sobre HTTP são, inicialmente, processadas por um contêiner web que as despacham para um *servlet* interno (*built-in*) conhecido como WebServiceServlet. Este *servlet* simplesmente encaminha as requisições para o contêiner de serviços web para processamento. SOAP sobre JMS usa fila JMS do WebLogic como intermediária e os mecanismos JMS do Weblogic para entregar as mensagens para a fila JMS. Uma vez na fila, o WebLogic Server usa um *bean* controlado por mensagens que desempilha as requisições e as envia para o contêiner do serviço web.

Quando uma requisição SOAP chega ao contêiner do serviço, o primeiro passo é identificar a classe que implementa o serviço a ser invocado. A classe de implementação é uma classe Java que foi desenvolvida utilizando modelo de programação JAX-WS ou JAX-RPC e implantada no WebLogic Server. Antes de invocar a classe, o contêiner invoca qualquer *handler* que esteja registrado. Considerando que os *handlers* não causem nenhum problema no processamento, o contêiner usa a especificação JAXB para desserializar (*unmarshal*) o corpo da mensagem SOAP nos objetos Java apropriados que são então passados como argumentos para o método Java na classe que implementa o serviço [Patric *et al.*,

[2] http://en.wikipedia.org/wiki/Java_Message_Service

2010]. Uma vez terminado o método, o contêiner serializa (*marshal*) o valor de retorno do método Java para o corpo da mensagem SOAP apropriada para a resposta, invoca os *handlers* registrados, na ordem inversa, e retorna a resposta para o cliente SOAP.

O contêiner do WebLogic Server do lado cliente fornece uma arquitetura similar. Quando um cliente Java invoca um serviço, o contêiner intercepta a invocação, serializa (*marshal*) os argumentos Java, invoca os *handlers* do lado do cliente definidos e, finalmente, invoca o serviço web remoto. A resposta se desenrola através dos mesmos passos, até que a invocação do método do serviço pelo cliente Java retorna a resposta como objeto Java.

3 Proposta de solução

A proposta de solução deste trabalho segue a arquitetura apresentada na Figura 3. A arquitetura tem como premissa o usuário ter sido devidamente autenticado pela aplicação cliente e ter suas informações armazenadas em um objeto que pode ser acessado de forma isolada. Quando a aplicação cliente faz uma requisição a um serviço web (web service – WS) executando em um servidor de aplicação, um *handler* intercepta a invocação do serviço e inclui a chave do usuário na mensagem de requisição (Figura 3.a). Quando a mensagem de requisição chega no servidor de aplicação, antes do serviço ser executado, a chave do usuário é extraída da mensagem e armazenada em um objeto cujo escopo é o do tratamento da requisição feita pela aplicação cliente (Figura 3.b). Quando o serviço acessa o banco de dados, a chave do usuário é obtida deste objeto e é injetada no contexto do banco de dados para que a operação sendo invocada pelo serviço no banco de dados acesse apenas os dados que o usuário tem acesso. Dessa forma, o banco de dados retorna apenas os dados autorizados para o serviço que os retorna para a aplicação cliente. A Figura 3 exemplifica um cenário em que a aplicação cliente invoca diretamente o serviço web. No entanto, cenários semelhantes podem incluir a chamada de mais de um serviço neste fluxo (i.e., aplicação cliente invoca serviço web que invoca outro serviço web que acessa o banco de dados) ou mesmo a invocação pela aplicação cliente de um objeto remoto (EJB ou Spring) que invoca um serviço web. No entanto, ressaltamos que a arquitetura proposta contempla estes cenários.

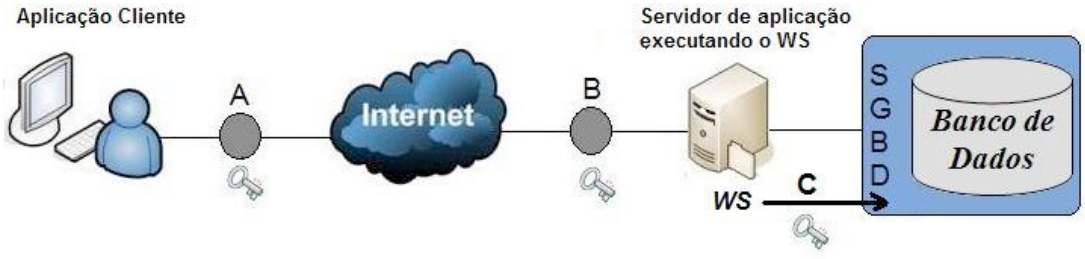

Figura 3 – Arquitetura de solução

A implementação da solução empregada neste trabalho para propagação consiste no armazenamento da chave do usuário no cabeçalho da mensagem SOAP [Gudgin *et al.*, 2007] enviada para o serviço web. A Figura 4 ilustra a arquitetura empregada para realização dos testes neste cenário. Neste caso, a aplicação cliente invoca um serviço web de lógica que invoca um serviço de dados que utiliza o framework Hibernate. O último é o framework responsável por injetar o usuário no contexto do Sistema de Gerência Banco de Dados (SGBD) responsável por executar a consulta.

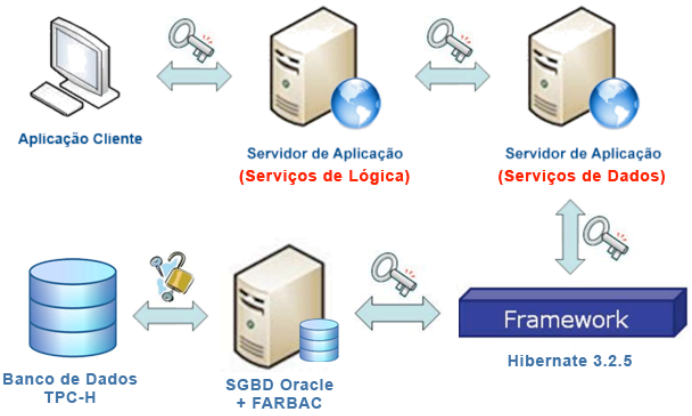

Figura 4 – Arquitetura empregada para os testes

Como dito anteriormente, mensagens SOAP podem ser transmitidas via HTTP ou JMS, isto é, o protocolo SOAP pode ser utilizado para trafegar dados (carga útil) dentro de mensagens HTTP ou JMS. A solução apresentada a seguir contempla tanto a propagação da identidade de um usuário através da manipulação do cabeçalho de uma mensagem HTTP quanto do cabeçalho de uma mensagem SOAP. Esta última alternativa é a mais flexível, pois independe do protocolo de transporte sobre o qual a mensagem será enviada.

As subseções a seguir demonstram a implementação de um serviço de dados, que será utilizado por um serviço de lógica, que por sua vez será acessado por uma aplicação cliente. Após as explicações dos métodos utilizados para criar tais projetos será demonstrado o que deve ser executado para que a propagação de identidade se faça possível. Para avaliar o protótipo foi utilizado o *benchmark* TPC-H (TPC Council 2008), que simula uma aplicação de apoio à decisão no contexto de uma empresa que realiza vendas mundialmente.

Os serviços desenvolvidos para avaliação da proposta fazem uso da tecnologia JAX-WS, tendo sido implantados em servidor de aplicação Weblogic 10. Posteriormente será demonstrado que a mesma proposta de arquitetura pode ser aplicada à serviços desenvolvidos com JAX-RPC e servidor Weblogic 9.

3.1 Implementação do Serviço de Dados

Serviços de dados encapsulam aspectos específicos de plataforma e implementam detalhes de modo que o consumidor não precisa saber como o serviço é implementado [Josuttis, 2007]. Neste protótipo foi implementado um serviço de dados que tem como funcionalidade acessar a base do TPC-H [TPC Council, 2008] e realizar a consulta (Figura 5) "*Order Priority Check*" deste *benchmark*, cujo objetivo é determinar quão bem o sistema de ordem de prioridades está funcionando para avaliar a satisfação dos clientes. Na prática a consulta retorna uma lista que relaciona os níveis de prioridade considerados pelo sistema e a quantidade de pedidos para cada um destes níveis. Um exemplo de retorno da consulta pode ser visto na Figura 6.

```
1  select o_orderpriority, count(*) as order_count from orders
2  where
3      o_orderdate >= date '[DATE]'
4      and o_orderdate < date '[DATE]' + interval '3' month
5      and exists (
6          select * from lineitem
7          where
8              l_orderkey = o_orderkey
9              and l_commitdate < l_receiptdate
10     )
11 group by o_orderpriority
12 order by o_orderpriority;
```

Figura 5 – Consulta do benchmark TPC-H para checagem de prioridades de pedido [TPC Council, 2008]

O_ORDERPRIORITY	ORDER_COUNT
1-URGENT	10594
2-HIGH	10476
3-MEDIUM	10410
4-NOT SPECIFIED	10556
5-LOW	10487

Figura 6 - Resultado da consulta *Order Priority Check*

Este serviço foi implementado através da IDE de programação *Oracle Workshop for Weblogic*, sendo posteriormente implantado em um servidor Weblogic. O serviço foi desenvolvido com JAX-WS (também para possibilitar as modificações que proverão a propagação de identidade).

O serviço expõe um único método, o *orderPriorityCheck()* (Figura 7), que recebe como parâmetros duas *Strings*, a primeira informando a data que deve ser considerada como data de início da consulta e a segunda como o intervalo de tempo entre a data de início e a data de fim. O serviço retorna um objeto "PriorityList", que comporta uma lista de objetos "OrderPriorityItem" (montada a partir do código apresentado na Figura 7, linhas 11 a 16), cada um destes objetos contém um nível de prioridade e o total de pedidos relacionados à este nível no intervalo de datas definido pelos parâmetros recebidos. O método faz uso de um objeto DAO (TpchDAO - Figura 8) para acessar o banco de dados. Este objeto DAO, por sua vez, faz uso do *framework* de mapeamento objeto-relacional Hibernate (Figura 9) para efetuar o acesso ao banco de dados propriamente dito.

```java
@WebService
public class DataService {

    public DataService(){}

    @WebMethod
    public PriorityList orderPriorityCheck(String dataInicio, String intervalo) {
        TpchDAO dao = new TpchDAO();
        PriorityList priorityList = new PriorityList();
        try{
            Map<String,Integer> priorityMap = dao.orderPriorityCheck(dataInicio,intervalo);
            List<OrderPriorityItem> lista = new ArrayList<OrderPriorityItem>();
            for(String key : priorityMap.keySet()){
                lista.add(new OrderPriorityItem(key, priorityMap.get(key)));
            }
            priorityList.setLista(lista);
        }catch(Exception e){
            System.out.println("DataService Exception: " + e.getMessage());
        }
        return priorityList;
    }
}
```

Figura 7 - Implementação do Serviço de Dados

```java
public class TpchDAO {
    private SessionFactory sessionFactory = HibernateUtil.getSesionFactory();

    public TpchDAO(){}

    public Map<String, Integer> orderPriorityCheck(String dataInicio, String intervalo){
        Map<String, Integer> priorityCount = new HashMap<String, Integer>();
        String sql = "select o_orderpriority, count(*) as order_count from tpch.orders"+
            " where o_orderdate >= date '"+dataInicio+"'"+
            " and o_orderdate < date '"+dataInicio+"' + interval '"+intervalo+"' month"+
            " and exists ("+
                " select * from tpch.lineitem "+
                " where l_orderkey = o_orderkey" +
                " and l_commitdate < l_receiptdate "+
            " ) "+
            " group by "+
            " o_orderpriority "+
            " order by "+
            " o_orderpriority";

        Session s = sessionFactory.openSession();
        Connection con = s.connection();
        PreparedStatement ps;
        try {
            ps = con.prepareStatement(sql);
            ResultSet rs = ps.executeQuery();
            s.close();
            while(rs.next()){
                priorityCount.put(rs.getString("o_orderpriority"),rs.getInt("order_count"));
            }
        } catch (SQLException e) {
            System.out.println("SQL EXCEPTION: "+e.getMessage());
        }

        return priorityCount;
    }
}
```

Figura 8 - Implementação da classe DAO

O objeto DAO precisa recuperar sessões com o banco de dados, para tal é feito o uso de um objeto Session Factory provido pela classe HibernateUtil (Figura 8 - linha 2), um padrão proposto pelos desenvolvedores do *framework* Hibernate para centralizar em um único ponto da aplicação sua inicialização e configuração. Mais a frente, quando forem tratadas as modificações propostas para viabilizar a propagação de identidade, será explicada a importância do uso de uma fábrica de sessões.

A implementação da classe HibernateUtil e dos arquivos de configuração necessários ao *framework* seguem as explicações detalhadas pelos desenvolvedores do *framework* em Kinkg *et al.* [2010a, 2010b] e são exemplificadas abaixo, respectivamente, pela Figura 9 e pela Figura 10.

```java
public class HibernateUtil {

    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            return new Configuration().configure().buildSessionFactory();
        }
        catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

}
```

Figura 9 - Implementação da classe HibernateUtil

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <property name="connection.url">jdbc:oracle:thin:@10.0.0.121:1521:ORCL</property>
        <property name="connection.username">UserDB</property>
        <property name="connection.password">********************</property>

        <property name="connection.pool_size">1</property>
        <property name="dialect">org.hibernate.dialect.Oracle10gDialect</property>
        <property name="current_session_context_class">thread</property>
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
        <property name="show_sql">true</property>
    </session-factory>
</hibernate-configuration>
```

Figura 10 - Arquivo de configuração hibernate.cfg.xml

3.2 Implementação do Serviço de Lógica

O serviço de lógica implementado tem como objetivo informar, dentre todos os pedidos realizados em um determinado espaço de tempo, qual o tipo de pedido mais comum em uma determinada escala de prioridades composta por cinco níveis: Urgente, Alta, Média, Não Especificada e Baixa. O retorno do serviço é uma *String* informando o nome do nível de prioridade com mais pedidos realizados. Este serviço é classificado como serviço "de lógica" de acordo com a classificação de Josuttis [2007] que apresenta que um serviço básico de lógica executa uma operação lógica em cima de dados que lhe foram passados por outras fontes, não cabendo a ele fazer o acesso à bases de dados ou sequer saber como tais acessos são feitos. O serviço executa uma checagem em cima de uma lista de objetos que armazenam o total de pedidos para um determinado nível de prioridade. Ao fim do processamento é retornado o nível de prioridade que possui maior quantidade de pedidos. Para recuperar a listagem com todos os níveis de prioridade e as respectivas quantidades de pedidos, o serviço de lógica invoca o serviço de dados apresentado na seção 3.1 . Neste teste, este serviço foi implantado no mesmo servidor de aplicação Weblogic do serviço de lógica, mas poderia estar disponível em um outro servidor de aplicação. Por fazer uso de um outro serviço o serviço de lógica se torna um consumidor de web service. Com o objetivo de gerar o conjunto de *stubs* necessários para realizar o acesso ao serviço de dados o utilitário

WSIMPORT [Oracle, 2011] foi utilizado. Maiores detalhes da implementação são apresentados a seguir.

Para criar o web service foi utilizada a IDE *Oracle Workshop for Weblogic*. Criou-se um novo projeto do tipo "WebService Project" (Figura 11).

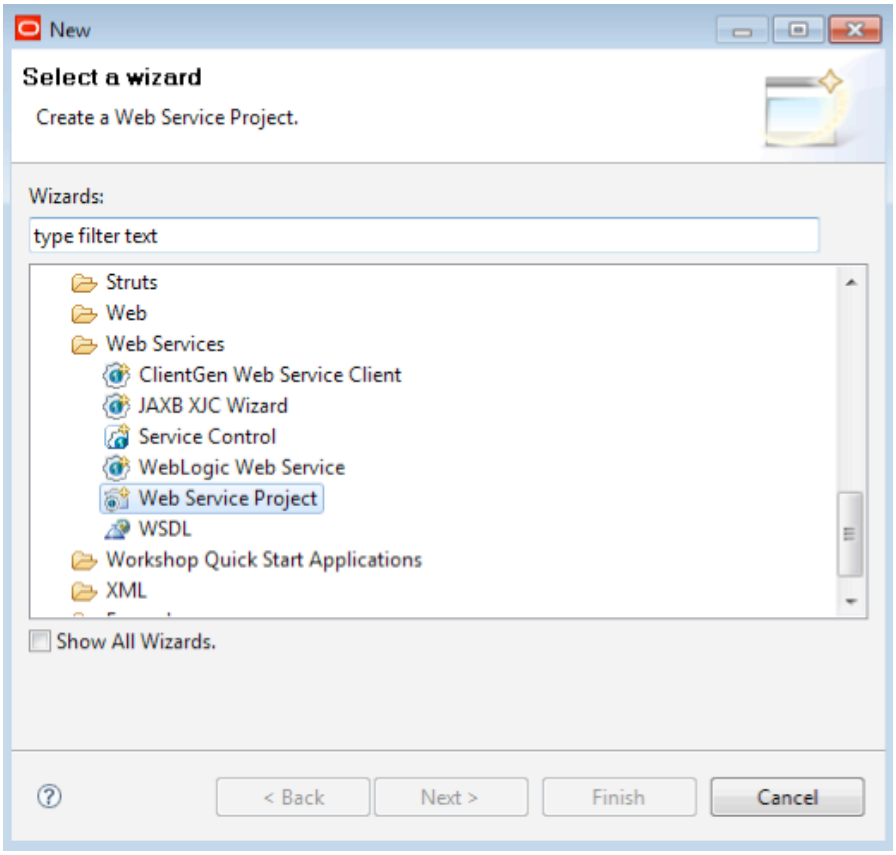

Figura 11 - Criação de um WebService Project

A seguir foi definido que seria utilizado JAX-WS, como descrito pela Figura 12 (necessário para as modificações que serão feitas posteriormente para possibilitar a propagação de identidade). A tela apresentada na Figura 12 propõe bibliotecas a serem adicionadas ao projeto. Para este exemplo não foram feitas quaisquer modificações nas bibliotecas propostas.

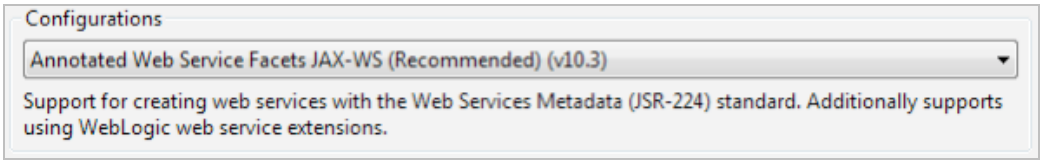

Figura 12 - Configuração do projeto com JAX-WS

Inicialmente é necessário informar ao novo projeto qual o caminho para o WSDL [Christensen *et. al.*, 2001] do serviço de dados implantado no servidor Weblogic. Tendo conhecimento do WSDL é possível criar o conjunto de *stubs* necessários para que seja feito o acesso ao serviço. Para gerar o pacote de *stubs* foi utilizado o WSIMPORT, um utilitário que é parte integrante do *core* da versão 6 do Java. Este utilitário se encarrega de gerar os *stubs*, tornando muito mais fácil o trabalho de escrever clientes de serviços. Para fazer uso do WSIMPORT é necessário que algumas variáveis de sistema estejam definidas. A variável JAVA_HOME deve ter sido criada e seu valor apontando para o caminho da JDK mais atual presente no computador. A variável "Path" deve ser atualizada para apontar também para o diretório "/bin"da JDK, para isto basta concatenar ao fim do valor presente o texto ";%JAVA_HOME%/bin".

O utilitário WSIMPORT é utilizado através do prompt de comando (também conhecido como Terminal). Abra o Prompt de comando e navegue até o diretório SRC do projeto cliente, então faça uso do comando "WsImport" definindo qual o pacote aonde deverão ser armazenados os *stubs* do serviço e qual o caminho para o WSDL deste serviço. O serviço deve estar implantado e executando no servidor. Uma vez localizado o diretório SRC do projeto do serviço de lógica, executa-se o utilitário de acordo com o ilustrado pela Figura 13. Os parâmetros inseridos indicam ao utilitário que os *stubs* deverão ser guardados ("-keep") em um pacote de nome "`br.uniriotec.stubs`" ("-p") e que o serviço a ser utilizado se encontra descrito pelo WSDL disponível através do endereço "`localhost:7001/C4ServicoDados/DataServiceService?wsdl`".

```
wsimport -keep -p br.uniriotec.stubs
http://localhost:7001/C4ServicoDados/DataServiceService?wsdl
```

Figura 13 - Uso do WsImport para criação do consumidor do serviço de dados

Deste modo o conjunto de *stubs* é gerado, sendo então possível utilizar o serviço de dados.

A classe LogicService apresentada na Figura 14 comporta a lógica do serviço em si, sendo composta pelo único método exposto (*checkMostCommonPriority()*). O método acessa o serviço de dados repassando os parâmetros que recebeu de entrada e obtem de retorno um objeto "PriorityList" (Figura 15), um POJO que possui apenas um atributo privado, uma lista de objetos "OrderPriorityItem". O objeto complexo foi criado para demonstrar a capacidade de retorno de objetos complexos por parte do serviço web. Uma simples implementação da classe java.util.List poderia ser utilizada. O objeto "OrderPriorityItem" (Figura 16) é, por sua vez, um POJO que possui apenas dois atributos, uma String indicando o nível de prioridade e um inteiro indicando o total de pedidos para aquele nível de prioridade. Com a lista em mãos o serviço identifica qual o nível de prioridade com maior quantidade de pedidos (Figura 14 – linhas 13 a 19), e retorna o nome deste nível.

```
1  @WebService
2  public class LogicService {
3
4      @WebMethod
5      public String checkMostCommonPriority(String dataInicio, String intervalo){
6          String result = "";
7
8          DataServiceService service = new DataServiceService();
9          DataService port = service.getDataServicePort();
10
11         PriorityList lista = port.orderPriorityCheck(dataInicio, intervalo);
12
13         int priorityCount = -1;
14         for(OrderPriorityItem item : lista.getLista()){
15             if(priorityCount < item.getQuantidade()){
16                 result = item.getPrioridade();
17                 priorityCount = item.getQuantidade();
18             }
19         }
20         return result;
21     }
22 }
```

Figura 14 - Implementação do Serviço de Lógica

```
6  public class PriorityList implements Serializable{
7      private static final long serialVersionUID = 1L;
8      private List<OrderPriorityItem> lista;
9
10     public PriorityList(){}
11
12     public List<OrderPriorityItem> getLista() {
13         return lista;
14     }
15
16     public void setLista(List<OrderPriorityItem> lista) {
17         this.lista = lista;
18     }
19 }
```

Figura 15 - Implementação da classe PriorityList

```
5  public class OrderPriorityItem implements Serializable {
6      private static final long serialVersionUID = 1L;
7      private String prioridade;
8      private int quantidade;
9
10     public OrderPriorityItem(){}
11
12     public OrderPriorityItem(String prioridade, int quantidade){
13         this.prioridade = prioridade;
14         this.quantidade = quantidade;
15     }
16
17     public String getPrioridade() {
18         return prioridade;
19     }
20
21     public void setPrioridade(String prioridade) {
22         this.prioridade = prioridade;
23     }
24
25     public int getQuantidade() {
26         return quantidade;
27     }
28
29     public void setQuantidade(int quantidade) {
30         this.quantidade = quantidade;
31     }
32
33 }
```

Figura 16 - Implementação da classe OrderPriorityItem

3.3 Aplicação Cliente

Um cliente JavaSE foi desenvolvido para executar a invocação de um Web Service disponibilizado em um servidor WebLogic. Neste caso, o serviço de lógica apresentado na seção 3.2 . Para criar a aplicação cliente foi utilizada a IDE Eclipse em sua versão *Indigo* (a IDE *Oracle Worshop for Weblogic* também pode ser utilizada).

O utilitário WSIMPORT foi utilizado novamente para gerar os *Stubs* de acesso ao serviço, como exemplificado na Figura 17. Observe que é necessário que o serviço de lógica esteja implantado no servidor de aplicação e que este se encontre ativo e acessível para que o WSIMPORT gere os *Stubs*.

```
wsimport -keep -p br.uniriotec.stubs
http://localhost:7001/C4ServicoLogica/LogicServiceService?wsdl
```

Figura 17 - Uso do WsImport para criação do consumidor do serviço de lógica

Os parâmetros inseridos indicam ao utilitário que os *stubs* deverão ser guardados ("-keep") em um pacote de nome "`br.uniriotec.stubs`" ("-p") e que o serviço a ser utilizado se encontra descrito pelo WSDL disponível através do endereço "`localhost:7001/C4ServicoLogica/LogicServiceService?wsdl`".

Na aplicação cliente um novo pacote foi criado e dentro dele implementou-se a classe "ConsumidorServico", que faz o papel de invocador do serviço de lógica. A Figura 18 ilustra o procedimento de invocação do serviço e o código utilizado para exibir o resultado no console da IDE.

```
1  public class ConsumidorServico {
2
3      public static void main(String[] args) throws IOException{
4          LogicServiceService service = new LogicServiceService();
5          LogicService port = service.getLogicServicePort();
6
7          String result = port.checkMostCommonPriority("1992-01-01","3");
8          System.out.println("Tipo mais comum de Pedido: "+result);
9      }
10
11 }
```

Figura 18 - Implementação inicial da classe ConsumidorServico

As linhas 4 e 5 da classe são responsáveis por recuperar a porta de acesso do serviço no servidor Weblogic, enquanto a linha 7 realiza a invocação do método do serviço que recupera o nível de prioridade mais comum entre os pedidos realizados (método *checkMostCommonPriority()*). Tanto a classe `LogicServiceService` quanto a classe `LogicService` foram criadas automaticamente com o uso do utilitário WSIMPORT Como visto o acesso ao serviço é feito com grande simplicidade.

3.4 Viabilizando Propagação de Identidade

As seções 3.1 , 3.2 e 3.3 explicaram a criação do cenário utilizado para testar a arquitetura proposta neste trabalho. O objetivo desta seção é explicar a proposta e o seu modo de implementação. Para tal será utilizado um SOAP *handler* [Oracle, 2008], classe que provê funcionalidades de interceptação para o envio e recepção de mensagens SOAP. O objetivo final é possibilitar que a identidade do usuário seja propagada até o serviço de dados utilizando *handlers* para interceptar mensagens SOAP, introduzindo em seu cabeçalho os dados a serem propagados. Uma vez que a informação tenha sido propagada até o serviço de dados será feita uma breve modificação na fábrica de sessões com o banco de dados para que o usuário seja injetado na conexão.

3.4.1 Arquitetura de Mensagens SOAP

Antes de abordarmos a solução menos intrusiva e menos impactante no cenário citado, é preciso entender a arquitetura da mensagem SOAP. Segundo Kalin [2009], uma mensagem SOAP é uma transmissão de sentido único do remetente para o destinatário; portanto, o princípio do padrão de troca de mensagem (Message Exchante Pattern – MEP) para SOAP é unidirecional. Aplicações baseadas em SOAP como serviços web são livres para criar padrões de conversação que combinam

mensagens unidirecionais de modo mais rico. Em um serviço web, o padrão de troca de mensagens requisição/resposta é um breve diálogo que uma requisição inicia a conversa e uma resposta a conclui. Padrões de troca de mensagem como requisição/resposta e solicitação/resposta podem ser colocados juntos de modo a suportar padrões de conversação mais amplos conforme necessário.

SOAP provê um modelo de processamento distribuído que assume que uma mensagem SOAP é originada em um remetente e tem como objetivo um destinatário final, através de zero ou mais intermediários [Gudgin *et. al.*, 2007], que são os destinatários não terminais ou nós ao longo da rota do remetente para o destinatário final. Um intermediário pode inspecionar e até mesmo manipular uma mensagem SOAP de entrada antes de enviar a mensagem em seu caminho em direção ao destinatário final. A Figura 19 apresenta um exemplo de um remetente SOAP, dois intermediários e um destinatário final.

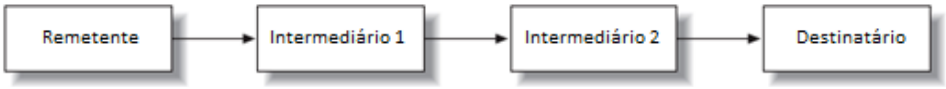

Figura 19 - Remetente, intermediários e destinatário

De acordo com Gudgin *et. al.* [2007], uma mensagem SOAP tem um corpo obrigatório, que pode ser vazio, e um cabeçalho opcional. Um intermediário deve fiscalizar e processar apenas os elementos do cabeçalho SOAP, em vez de qualquer informação no corpo da mensagem SOAP, independentemente do conteúdo que transporta o remetente visa o destinatário final para recebê-la. O cabeçalho, pelo contrário, destina-se a carregar metainformações, sendo adequado tanto à inserção de informações extras que devem ser vistas tanto pelo destinatário final como por intermediários. Por exemplo, o cabeçalho pode conter assinatura digital do remetente como um *voucher* ou incluir um *timestamp* que indica quando a informação no corpo da mensagem se torna obsoleta. Uma opção para quando se faz necessário o envio de múltiplos parâmetros através do cabeçalho SOAP (como por exemplo um objeto complexo) é a anexação de cada valor a uma lista ordenada, que será lida pelo destinatário quando a mensagem for recebida e que então poderá associar a posição do parâmetro na lista à um determinado atributo de um objeto complexo. Elementos XML dentro do cabeçalho opcional são blocos de cabeçalho na conversa SOAP.

3.4.2 Criação do projeto compartilhado

Na proposta de solução apresentada neste trabalho para propagação de identidade, um projeto compartilhado foi criado para comportar todos os artefatos que podem ser utilizados por mais de um serviço ou mais de um cliente. Neste caso, o projeto compartilhado comporta a classe *handler*, que prove a propagação, sendo utilizado tanto no emissor das requisições quanto nos receptores destas requisições. Este *handler* desempenha dois papéis:

1. Inserir os dados do usuário a serem propagados no cabeçalho das mensagens SOAP sempre que um cliente invocar um serviço, e

2. Procurar por dados do usuário no cabeçalho da mensagem SOAP sempre que uma requisição for recebida.

A especificação JWS provê um *framework* para criação de *handlers*. Kalin [2009] afirma que um *handler* pode ser injetado no *framework* em dois passos:

1. Um passo é criar uma classe *handler* que implemente a interface *Handler* do pacote `javax.xml.ws.handler`. JWS provê duas subinterfaces de *Handler*, *LogicalHandler* e *SOAPHandler*. O *LogicalHandler* tem acesso apenas ao *payload*

do corpo da mensagem SOAP, enquanto a interface *SOAPHandler* precisa definir três métodos, incluindo *handleMessage*, o qual dá acesso à mensagem ao programador. Os outros dois métodos são *handleFault* e *close*.

2. O outro passo é colocar o *handler* dentro de uma *handler chain*. Isto é feito tipicamente através de um arquivo de configuração, embora *handlers* também possam ser gerenciados através de código. A *handler chain* especifica quais *handlers* devem ser utilizados pela aplicação e em qual ordem.

Uma vez injetado no *handler framework*, o *handler* escrito pelo programador age como um interceptador de mensagem que tem acesso a toda mensagem de chegada e saída. Ou seja, considerando a visão de uma aplicação cliente, uma mensagem sendo enviada é interceptada, e manipulada, logo após ser disparada, e uma mensagem de resposta é interceptada antes de ser entregue ao objeto de destino.

Como pode existir mais de um intermediário, é necessário determinar quais *handlers* serão utilizados e em que ordem deverão ser invocados. Neste trabalho, foi utilizada a configuração de *handler* por arquivo de configuração, devido à sua flexibilidade e facilidade de manutenção, além de exigir menos intrusão no código legado, ocasionando apenas a inclusão de algumas anotações em determinadas classes. O *handler* criado nesta implementação de proposta de solução recebeu o nome de "UsernameHandler" e implementa a interface SOAPHandler, tendo como função incluir a identidade do usuário em toda mensagem SOAP de saída (quando esta estiver disponível) e procurar por esta mesma informação em toda mensagem SOAP de entrada. Tais funcionalidades foram implementadas no método *handleMessage()* da classe. Inicialmente são definidas três variáveis estáticas no escopo da classe, sendo elas: NAMESPACE, HEADER_ELEMENT_NAME e _username, esta última sendo declarada como pública, para que a informação do usuário nela armazenada possa ser utilizada pela aplicação. A Figura 20 exemplifica a definição das variáveis no início da classe *handler*.

```
private final static String NAMESPACE = "http://br.unirio.webservice";
private final static String HEADER_ELEMENT_NAME = "username";
public static final ThreadLocal<String> _username = new ThreadLocal<String>();
```

Figura 20 - Definição das variáveis estáticas do *handler*

A variável "_username" é do tipo *ThreadLocal*, sendo esta classe utilizada para garantir o isolamento do dado em situações de concorrência. Quando um serviço recebe requisições, uma *thread* é criada para atender cada uma destas requisições feitas ao servidor. Desta forma, cada requisição irá possuir sua própria variável _username. O método *handleMessage()* armazena nesta variável o valor extraído do cabeçalho da mensagem SOAP, garantindo que em situações onde múltiplas requisições são atendidas simultaneamente, cada requisição tenha conhecimento do usuário injetado no cabeçalho da mensagem SOAP que originou a requisição em si.

O método *handleMessage()* verifica se a mensagem SOAP interceptada é *outbound* ou *inbound*, isto é, se está sendo enviada, ou recebida pela aplicação. Caso a mensagem seja *outbound* (sendo enviada), o *handler* insere as informações do usuário como um novo elemento no cabeçalho da mensagem SOAP através do comando *addTextNode()*, como pode ser visto na Figura 21.

```
SOAPMessage msg = context.getMessage();
SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
SOAPHeader soapHeader = env.getHeader();

// Caso a mensagem não tenha cabeçalho, o handler o cria
if (soapHeader == null){
    soapHeader = env.addHeader();
}

// cria o elemento(DOM) que representa o usuário.
QName qname = new QName(NAMESPACE, HEADER_ELEMENT_NAME);


//adiciona o elemento criado no header da mensagem
SOAPHeaderElement headerElement = soapHeader.addHeaderElement(qname);

// adiciona um nó texto cujo conteúdo É o identificador do usuário.
headerElement.addTextNode(_username.get().toString());
msg.saveChanges();
```

Figura 21 - Código para inserir dados no cabeçalho SOAP

Caso a mensagem seja *inbound* (ou seja, está sendo recebida), o *handler* verifica se existe no cabeçalho da mensagem SOAP um elemento com o mesmo *namespace* definido na variável NAMESPACE e com o mesmo nome definido pela variável HEADER_ELEMENT_NAME, ou seja, verifica se no cabeçalho desta mensagem SOAP foi injetado um usuário. Caso exista um elemento com este *namespace* e com o nome especificado, o valor armazenado é extraído e armazenado no objeto "_username". A Figura 22 mostra o código responsável por executar estas operações.

```

SOAPHeader soapHeader = context.getMessage().getSOAPHeader();
NodeList headerElements = soapHeader.getElementsByTagNameNS(NAMESPACE, HEADER_ELEMENT_NAME);
String value = null;

if ((headerElements != null) && (headerElements.getLength() > 0)) {
    Node node = headerElements.item(0);
    if (node != null) {
        value = node.getFirstChild().getNodeValue();
        _username.set(value);
    }
}
```

Figura 22 - Código para extrair dados do cabeçalho SOAP

Com o *handler* criado, o projeto compartilhado deve então ser construído, a fim de gerar um arquivo ".JAR" que posa ser utilizado pelo cliente e pelos dois serviços.

3.4.3 Inclusão do *handler* no projeto cliente e nos serviço de lógica e dados

Como dito anteriormente a proposta contida neste documento tem como objetivo ser o mais não-intrusiva possível. Para que os projetos possam injetar o usuário no cabeçalho da mensagem SOAP e para que este mesmo dado possa ser lido por quem receber a mensagem, emissor e receptor devem fazer uso do *handler* disponibilizado pelo projeto compartilhado, portanto, é necessário que o JAR do projeto compartilhado seja adicionado ao *classpath* destes projetos. Começaremos explicando as alterações necessárias à aplicação cliente para que esta injete os dados do usuário na mensagem SOAP.

Neste protótipo, não nos propusemos a abordar técnicas para a autenticação do usuário que está acessando a aplicação cliente, mas sim considerando que a aplicação se encontra sendo executada por um usuário previamente autenticado, desta forma podemos também considerar que uma vez que o módulo autenticador tenha retornado uma resposta positiva em relação às credencias do usuário e lhe tenha permitido acesso, seja possível logo na sequência fazer a chamada ao método

que armazena no atributo "_username" o identificador deste usuário. Como o protótipo implementado é de extrema simplicidade arquitetural, usaremos o próprio método "*main()*" da aplicação cliente para armazenar no atributo "_username" da classe UsernameHandler o dado a ser propagado até o banco de dados. A Figura 23 mostra o comando que foi inserido (linha 4) logo ao início do método "*main()*" para que uma determinada chave de usuário fosse armazenada.

```
3  public static void main(String[] args) throws IOException{
4      UsernameHandler._username.set("Y2R7");
5
6      LogicServiceService service = new LogicServiceService();
7      LogicService port = service.getLogicServicePort();
8
9      String result = port.checkMostCommonPriority("1992-01-01","3");
10     System.out.println("Tipo mais comum de Pedido: "+result);
11 }
```

Figura 23 - Alteração no método *main()* do projeto cliente

Apesar de o dado já ter sido injetado na classe UsernameHandler, ainda não foi definido que esta deve ser utilizada pela aplicação como um *handler*. Para que o projeto reconheça a classe como um *handler* devemos ir até o pacote de *stubs* gerados através do comando WsImport e criar o arquivo "handlerchain.xml" (Figura 24), na raiz do projeto, onde especificaremos quais classes devem ser utilizadas como *handlers* e em qual ordem.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
3     <handler-chain>
4         <handler>
5             <handler-class>br.uniriotec.propid.shared.UsernameHandler</handler-class>
6         </handler>
7     </handler-chain>
8 </handler-chains>
```

Figura 24 - Arquivo "handlerchain.xml"

O arquivo handlerchain deve então ser posto em uso, para isto deve-se alterar a classe que representa o serviço sendo invocado, no caso deste protótipo, a classe que representa o serviço é a LogicServiceService (Figura 25). Nesta classe, logo antes do seu nome é inserida a anotação @HandlerChain (Figura 25 – linha 4), onde é informado qual arquivo de configuração deve ser utilizado como *handler chain*.

```
1  @WebServiceClient(name = "LogicServiceService",
2      targetNamespace = "http://services.logica.c4.propid.uniriotec.br/",
3      wsdlLocation = "http://localhost:7001/C4ServicoLogica/LogicServiceService?wsdl")
4  @HandlerChain(file="handlerchain.xml")
5  public class LogicServiceService extends Service{
6
7          /* conteúdo da classe omitido */
8
9  }
```

Figura 25 - Uso da anotação @HandlerChain

Uma vez que a aplicação cliente tenha sofrido as devidas alterações, devemos alterar o serviço de lógica para que este possa extrair os dados injetados no cabeçalho da mensagem SOAP. O serviço de lógica deste protótipo não chega a usar as informações injetadas durante seu processamento, entretanto, como o serviço de dados deve injetar esta informação no contexto da sessão com o banco de dados, é necessário que todo e qualquer serviço intermediário (como o serviço de lógica aqui descrito) repassem a informação adiante.

A classe UsernameHandler, tanto injeta o usuário em mensagens SOAP de saída quanto procura pelo dado em mensagens SOAP de entrada, no caso a aplicação cliente, apesar de ambas as operações estarem sendo executadas, não havia interesse em utilizar um possível usuário sendo recebido pelo cabeçalho da mensagem SOAP.

Como o serviço de lógica deve desempenhar simultaneamente o papel de um serviço a ser consumido (pela aplicação cliente) e o papel de um cliente que consome um determinado serviço (o serviço de dados), a existência das duas funcionalidades na classe UsernameHandler passa a ser crucial. O artefato UsernameHandler irá então recuperar o usuário sendo enviado pela aplicação cliente ao serviço de lógica e injetar o dado na mensagem enviada por este serviço ao serviço de dados.

A primeira alteração que deve ser realizada no projeto é a criação do arquivo de configuração "handlerchain.xml", que recebe a mesma configuração demonstrada na Figura 24. O arquivo atuará como *handler chain* para as mensagens de saída e de entrada, definindo que o *handler* UsernameHandler deverá ser utilizado para interceptar os dois tipos de mensagem. Com o arquivo criado deve-se então inserir a anotação @HandlerChain na classe que representa o serviço, como apresentado pela linha 2 da Figura 26.

```
2   @WebService
3   @HandlerChain(file="handlerchain.xml")
4   public class LogicService {
5
6       @WebMethod
7       public String checkMostCommonPriority(String dataInicio, String intervalo){
8           String result = "";
9
10          DataServiceService service = new DataServiceService();
11          DataService port = service.getDataServicePort();
12
13          PriorityList lista = port.orderPriorityCheck(dataInicio, intervalo);
14
15          int priorityCount = -1;
16          for(OrderPriorityItem item : lista.getLista()){
17              if(priorityCount < item.getQuantidade()){
18                  result = item.getPrioridade();
19                  priorityCount = item.getQuantidade();
20              }
21          }
22
23          return result;
24      }
25  }
```

Figura 26 - Inserção da anotação @HandlerChain no Serviço de Lógica

Assim como na aplicação cliente, a seção 3.1 demonstrou como o utilitário WsImport foi utilizado para criar o conjunto de *stubs* necessários para o acesso ao serviço de dados. Para determinar que ocorra a injeção do dado do usuário nas mensagens de saída com destino ao serviço de dados deve ser feita a inclusão da anotação @HandlerChain (linha 5 da Figura 27) na classe *stub* que especifica o acesso ao serviço de dados (no caso deste protótipo a classe DataServiceService).

```
3   @WebServiceClient(name = "DataServiceService",
4           targetNamespace = "http://services.c4.propid.uniriotec.br/",
5           wsdlLocation = "http://localhost:7001/C4ServicoDados/DataServiceService?wsdl")
6   @HandlerChain(file="handlerchain.xml")
7   public class DataServiceService  extends Service{
8       /** Código interno da classe omitido */
9   }
```

Figura 27 - Inserção da anotação @HandlerChain no acesso ao serviço de dados

O serviço de dados sofre algumas alterações a mais, não devido à recepção do dado injetado (que ocorre da mesma forma que no serviço de lógica), mas à necessidade de injetar o dado recebido em uma conexão com o banco de dados. Para tal serão utilizadas técnicas similares às propostas em Leão *et al.* (2011), onde uma fábrica de sessões alternativa é utilizada, em detrimento da fábrica fornecida pelo *framework* Hibernate.

Inicialmente será apresentada a alteração que permite, assim como no serviço de lógica, que o serviço de dados recupere o usuário injetado no cabeçalho da mensagem SOAP. Um artefato XML deve ser criado para comportar a configuração de *handlers* a serem utilizados pelo serviço, neste caso, um único *handler* será utilizado, o UsernameHandler, provido pelo projeto compartilhado. O arquivo "handlerchain.xml" carrega as configurações e é constituído dos mesmos parâmetros apresentados pela Figura 24 e utilizados tanto para a aplicação cliente quanto para o serviço de lógica. Para que o arquivo seja reconhecido como a *handler chain* a ser utilizada nas mensagens recebidas pelo serviço de dados a anotação @HandlerChain deve ser inserida na classe que implementa o serviço propriamente dito, como apresentado pela linha 2 da Figura 28.

```
1  @WebService
2  @HandlerChain(file="handlerchain.xml")
3  public class DataService {
4
5      public DataService(){}
6
7      @WebMethod
8      public PriorityList orderPriorityCheck(String dataInicio, String intervalo) {
9          TpchDAO dao = new TpchDAO();
10         PriorityList priorityList = new PriorityList();
11         try{
12             Map<String, Integer> priorityMap = dao.orderPriorityCheck(dataInicio, intervalo);
13             List<OrderPriorityItem> lista = new ArrayList<OrderPriorityItem>();
14             for(String key : priorityMap.keySet()){
15                 lista.add(new OrderPriorityItem(key, priorityMap.get(key)));
16             }
17             priorityList.setLista(lista);
18             return priorityList;
19         }catch(Exception e){
20             return priorityList;
21         }
22
23     }
24 }
```

Figura 28 - Inserção da anotação @HandlerChain no Serviço de Dados

A nova fábrica de sessões a ser utilizada difere da fornecida pelo *framework* Hibernate por realizar operações adicionais antes de retornar a sessão ao ponto da aplicação que a requisitou, de fato a nova fábrica faz uso da fábrica de sessões do Hibernate para gerar uma nova sessão e realiza as operações adicionais sobre esta sessão. A nova SessionFactory é apresentada pela Figura 29.

```
1  public class SessionFactory{
2      private static SessionFactory _instance = null;
3
4      private SessionFactory(){}
5
6      public static synchronized SessionFactory getInstance(){
7              if(_instance == null){
8                  _instance = new SessionFactory();
9              }
10             return _instance;
11     }
12
13     public Session openSession(){
14         Session s = HibernateUtil.getSessionFactory().openSession();
15
16         if(UsernameHandler._username.get() != null){
17             Connection con = s.connection();
18             try{
19                 CallableStatement st = con.prepareCall(
20                         "{call dbms_application_info.set_client_info(?)}");
21                 st.setString(1, UsernameHandler._username.get().toString());
22                 st.executeUpdate();
23             }catch(SQLException e){
24                 return null;
25             }
26         }
27         return s;
28     }
29 }
```

Figura 29 - Fábrica de sessões para possibilitar a propagação da identidade

A fábrica de sessões do Hibernate é acessada através do método *getSessionFactory()* da classe HibernateUtil, que configura o *framework*. Com a referência para a fábrica em mãos, é utilizado o método *openSession()*, que devolve uma sessão com o banco de dados. A partir deste ponto entra o código responsável por realizar a injeção do usuário na sessão. Verifica-se se o artefato UsernameHandler possui o atributo "_username" ajustado (o acesso direto a este atributo é possível pois o mesmo é declarado como "*static*" na classe UsernameHandler). O fato de o atributo armazenar algum valor significa que o serviço recebeu, através do cabeçalho da mensagem SOAP que carregou a requisição, a informação de qual usuário está utilizando a aplicação cliente. No caso de o atributo "_username" possuir valor diferente de NULL, é realizada a chamada à procedure DBMS_APPLICATION_INFO.SET_CLIENT_INFO(), informando como parâmetro o usuário armazenado pelo atributo. A procedure citada faz parte do *namespace* USERENV do banco de dados Oracle [Jeloka *et. al.*, 2008], sendo utilizada para ajustar no contexto da sessão o usuário que está de fato fazendo uso daquela sessão. Sendo o usuário reconhecido é possível utilizar o *framework* FARBAC para aplicar regras de autorização de acesso aos dados armazenados.

Para realizar a troca de fábricas basta alterar no objeto DAO responsável por realizar as operações com o banco de dados a linha que define qual classe deve ser utilizada como fábrica de sessões. A Figura 30 apresenta a alteração realizada à linha 2 da classe, que define a fábrica que será utilizada pela classe. O código omitido no método *orderPriorityCheck()* é o mesmo exposto pela Figura 8.

```
public class TpchDAO {
    private SessionFactory sessionFactory = SessionFactory.getInstance();

    public TpchDAO(){}

    @SuppressWarnings("deprecation")
    public Map<String, Integer> orderPriorityCheck(String dataInicio, String intervalo){

        /** Código omitido */

    }
}
```

Figura 30 - Alteração ao objeto DAO

3.5 Aplicação da Proposta de Solução à JAX-RPC

Como dito anteriormente, o padrão para desenvolvimento de webservice JAX-RPC foi sucedido pela tecnologia JAX-WS. Entretanto, muitos sistemas legados, ou construídos sobre tecnologias mais antigas, continuam a fazer uso do primeiro padrão. Caso seja necessário implementar propagação de identidade em sistemas que façam uso destas tecnologias, a mesma arquitetura proposta ao cenário com JAX-WS pode ser aplicada, realizando-se apenas algumas alterações na implementação, como demonstrado a seguir.

Para avaliar a aplicação da proposta ao cenário que contempla o uso das tecnologias JAX-RPC, Java 4 e Weblogic 9, um simples serviço "Hello World" foi desenvolvido. O serviço foi então invocado através da aplicação SoapUI[3], que disponibiliza, a partir do descritor WSDL do serviço, as mensagens SOAP de invocação à cada método exposto. A aplicação permite ainda a alteração da mensagem que será utilizada para realizar a invocação, característica importante, que possibilitou de forma simples a inserção da chave do usuário no cabeçalho da mensagem, de forma a simular a manipulação que poderá ser feita por diferentes linguagens de desenvolvimento, como Java, .NET, Python, etc. A interceptação dos dados se tornou possível devido ao uso de *Handlers*, utilizados tanto com o objetivo de injetar os dados na mensagem quanto de extraí-los no lado servidor. O *Handler* implementado foi armazenado em um projeto compartilhado para facilitar a sua utilização por múltiplas aplicações.

Nesta implementação da proposta de solução com JAX-RPC, foi desconsiderado o acesso ao banco de dados, que poderá ser realizado como descrito para os serviços JAX-WS (Seção 3.1 e Seção 3.2). Neste caso, desejando-se utilizar o controle de acesso a dados diretamente no banco considerando o usuário da aplicação cliente, o usuário deverá ser propagado entre as camadas da aplicação e o acesso aos dados deverá ser controlado pelo *framework* FARBAC. O ajuste do usuário que realiza o acesso no contexto da seção deverá ser feito pela chamada da *procedure* DBMS_APPLICATION_INFO.SET_CLIENT_INFO().

3.5.1 Implementação do Serviço JAX-RPC

O serviço desenvolvido tem como objetivo expor um único método, que não recebe parâmetros e é responsável por retornar uma *String* com o texto "Olá Mundo!". A princípio o serviço não faz uso de Propagação de Identidade, tal funcionalidade será incluída adiante. A Figura 31 mostra o código referente ao serviço, que foi então implantado em um servidor de aplicação Weblogic versão 9.2.

[3] http://www.soapui.org/

```java
package br.uniriotec.propid;

import javax.jws.*;

@WebService
public class ServicoHello {

    @WebMethod
    public String hello() {
        return "Olá Mundo!";
    }
}
```

Figura 31 - Serviço desenvolvido com JAX-RPC

3.5.2 Chamada do Serviço pelo aplicativo SoapUI

A invocação do Serviço foi realizada através do aplicativo SoapUI. A localização do descritor WSDL do serviço é fornecida ao aplicativo que cria modelos de mensagens SOAP para invocar cada método exposto por este serviço. Uma vez que o serviço "ServicoHello" foi implantado no servidor, o endereço de seu WSDL foi então fornecido ao SoapUI, que criou o projeto apresentado pela Figura 32.

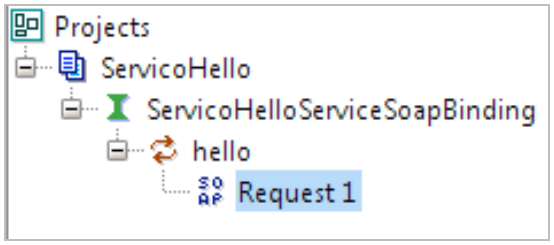

Figura 32 - Projeto SoapUI

A invocação foi então realizada e o serviço retornou, de forma bem sucedida, a mensagem "Olá Mundo!" demonstrando que o serviço estava ativo e acessível (Figura 33).

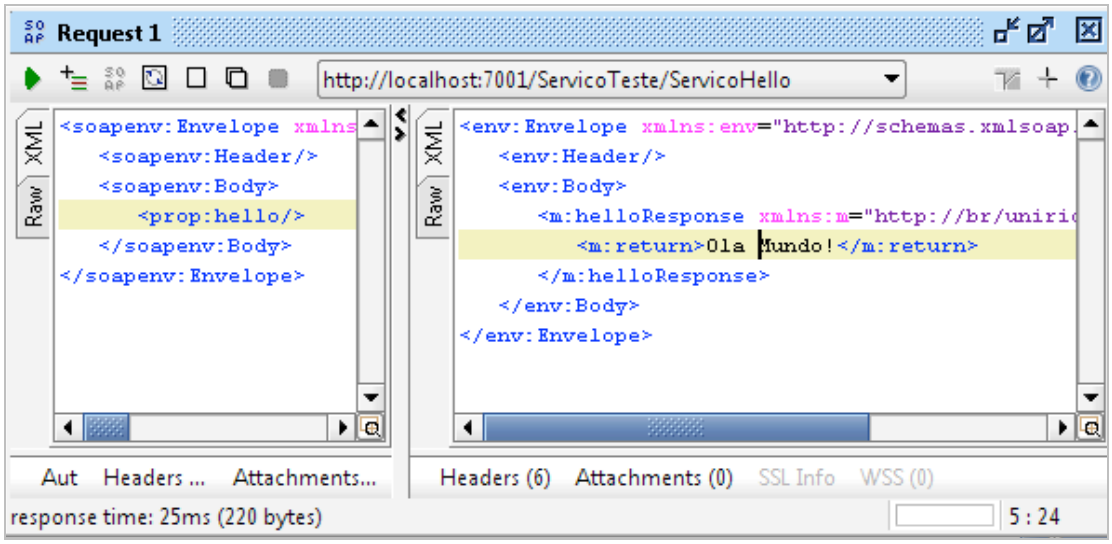

Figura 33 - Invocação e Resposta do método "hello()" no serviço

3.5.3 Viabilizando Propagação de Identidade em serviços JAX-RPC

Para viabilizar a propagação de identidade por serviços desenvolvidos através de JAX-RPC é proposto então utilizar *Handlers*, assim como nos serviços em JAX-WS.

Entretanto, a especificação do JAX-RPC conta com um conjunto diferente de classes e funcionalidades. Logo, para tal tarefa, será utilizado um *Handler* levemente diferente daquele implementado para a solução com JAX-WS.

O *Handler* criado faz apenas o papel de verificar se a mensagem SOAP de entrada, ou seja, aquela sendo utilizada para invocar o método do serviço, carrega um cabeçalho específico, onde se encontra o identificador do usuário na aplicação cliente. Para fins de teste o *Handler* foi inserido no mesmo projeto onde se encontra o código do serviço e segue abaixo, descrito pela Figura 34.

```
1  package br.uniriotec.propid;
2
3  import javax.xml.namespace.QName;
4  import javax.xml.rpc.handler.GenericHandler;
5  import javax.xml.rpc.handler.MessageContext;
6  import javax.xml.rpc.handler.soap.SOAPMessageContext;
7  import javax.xml.soap.Name;
8  import javax.xml.soap.SOAPEnvelope;
9  import javax.xml.soap.SOAPException;
10 import javax.xml.soap.SOAPHeader;
11 import javax.xml.soap.SOAPHeaderElement;
12 import javax.xml.soap.SOAPMessage;
13
14 import org.w3c.dom.Node;
15 import org.w3c.dom.NodeList;
16
17 public class UsernameHandlerRPC extends GenericHandler{
18     private final static String NAMESPACE = "http://br.unirio.webservice";
19     private final static String HEADER_ELEMENT_NAME = "username";
20     public static final ThreadLocal<String> _username = new ThreadLocal<String>();
21
22     @Override
23     public boolean handleRequest(MessageContext messageContext) {
24         SOAPMessageContext context = (SOAPMessageContext) messageContext;
25         try {
26             SOAPHeader soapHeader = context.getMessage().getSOAPHeader();
27
28             NodeList headerElements =
29                 soapHeader.getElementsByTagNameNS(NAMESPACE, HEADER_ELEMENT_NAME);
30             String value = null;
31
32             if ((headerElements != null) && (headerElements.getLength() > 0)) {
33
34                 Node node = headerElements.item(0);
35
36                 if (node != null) {
37                     value = node.getFirstChild().getNodeValue();
38                     _username.set(value);
39                 }
40             }
41         }catch(SOAPException e){
42             System.out.println(e);
43         }
44
45         return true;
46     }
47
48     @Override
49     public QName[] getHeaders() {
50         return null;
51     }
52 }
```

Figura 34 - Handler para serviços JAX-RPC

Como pode ser observado no código demonstrado pela Figura 34, o *handler* difere em funcionalidade daquele descrito pela seção 3.4.2 por realizar somente a extração de dados do cabeçalho SOAP. Deve-se observar ainda que este faz uso da interface GenericHandler em detrimento da interface SOAPHandler, e devido a isto, implementa um método de nome "handleRequest()" ao invés do método "handleMessage()". No processamento realizado pelo handler, é verificada a existência de um atributo "username" com *namespace* "http://br.unirio.webservice". Caso exista o atributo, este é inserido em uma variável do tipo *ThreadLocal*, de escopo *static*, que pode então ser acessada pelo método do serviço.

Com o *Handler* criado, sua utilização deve ser indicada através de uma combinação de anotações que precede a declaração da classe. São utilizadas as anotações @SOAPMessageHandlers (que denota um conjunto de handlers) e @SOAPMessageHandler (que especifica um handler a ser utilizado).

Para verificar a utilização do *Handler* o serviço foi alterado para retornar, ao invés de uma simples mensagem "Usuário não identificado!", uma saudação ao usuário que realizou a requisição. Caso nenhum usuário seja injetado, o serviço retornará a saudação padrão. A Figura 35 apresenta o serviço reescrito para considerar a utilização do *handler*.

```
1  package br.uniriotec.propid;
2
3  import javax.jws.WebMethod;
4  import javax.jws.WebService;
5  import javax.jws.soap.SOAPMessageHandler;
6  import javax.jws.soap.SOAPMessageHandlers;
7
8  @WebService
9  @SOAPMessageHandlers(
10     {@SOAPMessageHandler(className="br.uniriotec.propid.UsernameHandlerRPC")})
11 public class ServicoHello {
12
13     @WebMethod
14     public String hello() {
15         if(UsernameHandlerRPC._username.get() != null){
16             return "Ola "+ UsernameHandlerRPC._username.get().toString() + "!";
17         }else{
18             return "Usuario nao identificado.";
19         }
20     }
21 }
```

Figura 35 - Serviço JAX-RPC Reescrito

A chamada ao método exposto foi realizada através do SoapUI. Primeiramente foi realizada uma invocação onde não se informou o usuário, a seguir a mensagem SOAP foi editada para incluir a chave do usuário. O retorno sem a identificação do usuário é demonstrado pela Figura 36, enquanto a mensagem SOAP editada com a inserção de um novo cabeçalho é demonstrada pela Figura 37 e a resposta da invocação pela Figura 38.

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
   <env:Header/>
   <env:Body>
      <m:helloResponse xmlns:m="http://br/uniriotec/propid">
         <m:return>Usuario nao identificado.</m:return>
      </m:helloResponse>
   </env:Body>
</env:Envelope>
```

Figura 36 - Resposta do Serviço sem uso de propagação

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:prop="http://br/uniriotec/propid"
   xmlns:teste="http://br.unirio.webservice">
   <soapenv:Header>
      <teste:username>Y5QL</teste:username>
   </soapenv:Header>
   <soapenv:Body>
      <prop:hello/>
   </soapenv:Body>
</soapenv:Envelope>
```

Figura 37 - Mensagem SOAP de requisição informando usuário no Header

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
   <env:Header/>
   <env:Body>
      <m:helloResponse xmlns:m="http://br/uniriotec/propid">
         <m:return>Ola Y5QL!</m:return>
      </m:helloResponse>
   </env:Body>
</env:Envelope>
```

Figura 38 - Resposta do Serviço com uso de propagação

3.5.4 Invocação do serviço por aplicação cliente

Uma aplicação cliente foi criada para realizar o acesso ao serviço disponibilizado. A criação do cliente contou com o uso da ferramenta WSimport a fim de que fossem criadas as classes responsáveis por localizar e realizar o acesso ao serviço. Nesta aplicação utilizou-se o *framework* para testes unitários TestNG, para realizar a invocação do método que fazia uso do serviço remoto.

A propagação da identidade foi realizada com base no handler implementado para o serviço JAX-WS. Isto se fez possível pois, apesar de o serviço invocado ter sido desenvolvido com a tecnologia JAX-RPC, a sua invocação se dá simplesmente através de uma mensagem SOAP, sendo portanto transparente ao cliente a linguagem de programação e a especificação tecnológica do serviço invocado. O método injeta no handler o usuário que faz uso da aplicação, e quando o serviço é de fato invocado, o usuário é inserido no *header* da mensagem SOAP. O serviço, então, procura no *header* pela informação e, ao encontrá-la, a extrai-la e armazena para uso futuro.

O uso do *framework* TestNG possibilitou a execução concorrente de múltiplas invocações do serviço (injetando diferentes usuários) para que fosse validado inclusive o isolamento entre cada execuçao no servidor. A criação do método "BeforeClass", anotado pela anotação "@BeforeClass", realiza a criação de uma lista com os usuários a serem utilizados para o teste e os valores esperados como retorno da invocação quando cada um destes usuários é propagado. O *framework* foi configurado para executar o método 100 vezes, fazendo uso de um *pool* de 5 *threads*. A Figura 39 mostra a implementação do método para invocação do serviço na aplicação cliente.

```
public class ClientProp {
    private Map<String, String> lista;
    private int flag = 0;

    @BeforeClass
    public void beforeClass() {
        lista = new HashMap<String, String>();
            lista.put("Y2R7", "Ola Y2R7!");
            lista.put("Y5QL", "Ola Y5QL!");
            lista.put("Y2T0", "Ola Y2T0!");
    }

    @Test(threadPoolSize = 5, invocationCount = 100)
    public void invokeInject() {
      String[] usuarios = lista.keySet().toArray(new String[0]);
      String usuario = usuarios[flag];
      String msgEsperada = lista.get(usuario);

      UsernameHandler._username.set(usuario);

        if(flag == lista.size()-1){
            flag = 0;
        }else{
            flag++;
        }


      ServicoHelloService service = new ServicoHelloService();
      ServicoHello port = service.getServicoHelloSoapPort();

      Assert.assertEquals(port.hello(), msgEsperada,
            "Valor retornado pelo servico difere do esperado");

    }
}
```

Figura 39 - Código da aplicação cliente para invocação do serviço JAX-RPC

O cliente foi capaz de demonstrar a eficácia da utilização da proposta na tarefa de propagar a identidade de um usuário quando utilizando serviços desenvolvidos através da linguagem de programação Java com a especificação para criação de serviços web JAX-RPC.

4 Testes Experimentais

A fim de avaliar a proposta de solução implementada utilizando as especificações JAX-WS e JAX-RPC em um ambiente de alta concorrência, foram realizados teste experimentais analisar ass abordagens.

Na implementação da solução empregando JAX-WS, a aplicação cliente e os serviços de dados e de lógica foram implementados utilizando tecnologia Java (Java SE para aplicação cliente e JAX-WS para os serviços). Um banco de dados foi carregado seguindo o esquema do *benchmark* TPC-H [TPC Council 2008] sendo configurado com o *framework* FARBAC. Para disponibilizar os serviços, foi utilizado o Oracle Weblogic Server (10g R3) como servidor de aplicação e o Oracle 10g como SGBD. Para realizar os testes concorrentes utilizou-se o *framework* para testes unitários TestNG [Beust e Suleiman 2007], uma vez que este fornece por natureza as ferramentas necessárias para execução de testes de paralelismo de forma simples e eficaz. Outra possibilidade de execução de testes bastante utilizada na indústria é o JUnit, mas este não foi utilizado pela simplicidade do TestNG.

A implementação da solução considerando a especificação JAX-RPC foi testada através da criação de um handler para o serviço existente na empresa. O Handler possui funcionalidade identica à implementada no handler para a especificação JAX-WS, entretanto, como explicado pela seção 3.5.3 foi necessário estender outra interface (GenericHandler). O Handler foi então inserido no projeto e a classe que

especifica o serviço foi alterada para considerá-la, além do handler que já se encontrava em uso. O serviço foi então invocado através da aplicação SoapUI, levando-se em conta uma invocação onde nenhum dado adicional é inserido na mensagem SOAP e outra onde os dados de um usuário são informados através do *header* da mensagem SOAP de invocação.

4.1 Testes para a especificação JAX-WS

Para os testes relativos à especificação JAX-WS dois diferentes computadores foram utilizados. O primeiro (Windows 7 Professional - Intel Core i3, 4GB RAM, 320 GB de HD) executou simultaneamente o servidor de aplicação Weblogic e o banco de dados Oracle (com o *framework* FARBAC). O segundo, que foi configurado em uma máquina virtual Windows 7 sobre arquitetura Mac OS X Lion (Intel Core i5, 2 GB RAM, 40GB HD) fez o papel de aplicação cliente, neste caso executando o caso de teste através da IDE Eclipse (Indigo) e do *framework* TestNG. Os testes foram executados com ambas as máquinas na mesma rede em momento de baixa demanda desta.

O *benchmark* TPC-H simula o cenário de um sistema de apoio a decisão, provendo um contexto realista que representa a atividade de vendas de uma indústria que administra suas vendas para todo o mundo. Para este teste foi utilizada a consulta *OrderPriorityCheck*, proposta pelos desenvolvedores do *benchmark*, que visa obter a lista com todos os níveis de prioridade dos pedidos e a quantidade de pedidos para cada nível realizados em um determinado espaço de tempo. Esta consulta é executada pelo serviço de dados, cuja implementação é apresentada na Figura 28. Já o serviço de lógica (cuja implementação é apresentada na Figura 26) tem como objetivo retornar o nível de prioridade mais comum entre os pedidos realizados, para isto recupera-se a lista com todos os níveis de prioridade fornecidos pelo serviço de dados e filtra-se aquele que possui a maior quantidade de pedidos registrada.

A seguinte regra de autorização foi implementada no *framework* FARBAC: "Um gerente da América do Norte e Ásia deve somente acessar pedidos de clientes localizados no hemisfério norte e que se encontram nas regiões da Ásia e América". Considerando a consulta, as informações a serem protegidas se encontram nas tabelas *ORDERS*, que armazena os pedidos realizados, e *LINEITEM*, que armazena os itens de cada pedido. As tabelas possuem aproximadamente 3 milhões e 12 milhões de registros, respectivamente. A Figura 40 apresenta o predicado retornado pelo FARBAC para a tabela *ORDERS*; enquanto que a Figura 41 apresenta o predicado referente à tabela *LINEITEM*.

```
(O_CUSTKEY in (
select C_CUSTKEY
from TPCH.CUSTOMER
inner join TPCH.NATION on N_NATIONKEY = C_NATIONKEY
inner join TPCH.REGION on R_REGIONKEY = N_REGIONKEY
where R_NAME IN ('AMERICA' , 'ASIA') AND N_HEMISPHERE IN ('NORTH')))
```

Figura 40 - Predicados FARBAC para a tabela *ORDERS*

```
(L_ORDERKEY in (select O_ORDERKEY from TPCH.ORDERS))
```

Figura 41 - Predicados FARBAC para a tabela *LINEITEM*

Dois usuários foram criados: *Bob*, como gerente da América do Norte e Ásia; e *Alice*, a presidente da empresa, que recebeu o privilégio *EXEMPT ACCESS POLICY*, fazendo com que todas as políticas sejam ignoradas.

A aplicação cliente foi simulada através de um caso de teste implementado com o auxílio do *framework* TestNG (Figura 42). Como a proposta de solução não considera a autenticação de usuários, a passagem destes ao *handler* (para que este realize a injeção na mensagem SOAP) é feita dentro do próprio caso de teste. A utilização do serviço de lógica pode ser observada nas linhas 21, 22 e 24, onde é invocado o método *checkMostCommonPriority()*, que retorna um objeto *String* informando o nível mais comum de prioridade entre os pedidos realizados no intervalo de 3 meses a partir da data 02/07/1992. Espera-se que o usuário *Alice* receba como resposta o nível "1-URGENT", enquanto o usuário *Bob* deve receber o resultado "5-LOW".

O caso de teste foi configurado para ser executado 100 vezes, permitindo o paralelismo de até 5 *threads* por vez. De forma intercalada, foram realizadas invocações com os usuários Bob e Alice. Foi possível observar que a regra de autorização implementada foi de fato aplicada de forma isolada, dado que os valores retornados foram iguais aos declarados como esperados para cada usuário e houve intercalação nas respostas recebidas. Portanto, foi possível observar que mesmo com diferentes usuários realizando requisições simultâneas a proposta de solução é aplicável em ambientes de alta concorrência utilizados por múltiplos usuários, garantindo que cada usuário só visualize os dados aos quais tem permissão de acesso.

```
1  public class TesteConcorrenciaTPCH {
2      private Map<String, String> mapUsuarios;
3      private int flag = 0;
4
5      @BeforeTest
6      public void beforeTest() {
7          mapUsuarios = new HashMap<String, String>();
8              mapUsuarios.put("ALICE", "1-URGENT       ");
9              mapUsuarios.put("BOB", "5-LOW          ");
10     }
11
12     @Test(threadPoolSize = 5, invocationCount = 100)
13     public void testConcorrenciaConsulta() {
14         String[] usuarios = mapUsuarios.keySet().toArray(new String[0]);
15         String usuario = usuarios[flag];
16
17         UsernameHandler._username.set(usuario);
18
19         if(flag == 1){flag = 0;}else{flag = 1;}
20
21         LogicServiceService service = new LogicServiceService();
22         LogicService port = service.getLogicServicePort();
23
24         String result = port.checkMostCommonPriority("1992-07-02","3");
25         Assert.assertEquals(result, mapUsuarios.get(usuario));
26     }
27 }
```

Figura 42 - Caso de Teste para simular a aplicação Cliente

A seguir foi realizado teste de desempenho com o objetivo de observar o custo de processamento adicionado pelo uso de *handlers* para realizar a propagação de identidade. Testes unitários foram executados, primeiramente com serviços que consideravam a propagação através de *handlers* e posteriormente com serviços que não faziam uso dos interceptadores. Em ambos os casos utilizou-se um *Mock Object* para realizar o papel de objeto DAO, objetivando a retirada do banco de dados do cenário de testes para que o processamento deste não interferisse na medição. Assim como no teste de concorrência, o cenário contou com um cliente e dois serviços web, sendo um de lógica e um de dados. Os testes unitários foram executados pela mesma máquina onde se encontravam disponíveis serviços, de modo que o desempenho da rede também não interferisse nos resultados, sendo configuradas 5000 requisições permitindo o paralelismo de até 5 *threads* por vez. Como resultado, observou-se que no cenário sem propagação de identidade as 5000 requisições

foram realizadas em 3 minutos e 30 segundos (média de 0,042 segundos por requisição), enquanto no cenário considerando a propagação as mesmas 5000 requisições foram executadas em 3 minutos e 59 segundos (média de 0,0478 segundos por requisição), ou seja, 14% a mais de tempo de processamento, em média. Deve-se observar que o tempo para realizar a comunicação com o serviço é extremamente pequeno e, considerando um cenário onde é requisito acesso seguro a informações, o *overhead* produzido pela proposta é mínimo, demonstrando que a solução proposta não gera acréscimo de processamento que impacte de forma considerável o ambiente ou o tempo de resposta da arquitetura.

4.2 Testes para a especificação JAX-RPC

Os testes que considerando a especifiação JAX-RPC foram realizados em um cenário real. O serviço utilizado é um serviço para disponibilização, em tempo real, de medições de propriedades físicas aquisitadas por instrumentos durante a perfuração de poços.

Na implementação empregando este serviço, foi adicionado o handler UsernameHandlerRPC, que extrai a chave do usuário armazenadas no atributo ("username") no cabeçalho da mensagem SOAP. O método "getCurveNatureList()" do serviço real foi modificado para escrever no console do servidor a mensagem "Nenhum usuário Injetado" quando o cabeçalho não comportar informações adicionais. Quando houver um atributo "username" associado ao *namespace* "http://br.unirio.webservice" presente no cabeçalho da mensagem SOAP, o método escreve no console do servidor qual usuário foi injetado. A Figura 43 mostra a anotação que define o uso do handler pelo serviço e as alterações ao método "getCurveNatureList()" para que o usuário injetado seja exibido no console.

```
@WebService
@SOAPMessageHandlers({
    @SOAPMessageHandler(className="br.uniriotec.np2tec.propid.handler.UsernameHandlerRPC"),
    @SOAPMessageHandler(className="br.com.petrobras.gdiep.handlers.SoapFaultHandler")})
public class DrillingLogService extends ServiceBase {

    @WebMethod
    @RolesAllowed(value={@SecurityRole(role="users")})
    public CurveNatureList getCurveNatureList() throws SOAPException, SQLException {
        if(UsernameHandlerRPC._username.get() != null){
            System.out.println("Usuario Injetado: " +
                UsernameHandlerRPC._username.get().toString());
        }else{
            System.out.println("Nenhum Usu·rio Injetado.");
        }
        return control.getCurveNatureList();
    }

    //Outros métodos ocultados//

}
```

Figura 43 - Alterações realizadas no serviço wue representa cenário real

O método foi invocado duas vezes. Na primeira invocação utilizou-se a mensagem SOAP criada como padrão pelo SoapUI, sem qualquer inclusão de dado extra que devesse ser propagado. A Figura 44 demonstra a mensagem SOAP utilizada para a invocação do método.

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Header/>
    <soapenv:Body>
        <ser:getCurveNatureList/>
    </soapenv:Body>
</soapenv:Envelope>
```

Figura 44 - Mensagem SOAP criada pelo SoapUI

Como resultado, o método exibiu no console do servidor a mensagem "Nenhum Usuário Injetado", como esperado Na segunda vez, a mensagem SOAP gerada pelo SoapUI foi alterada, com o objetivo de adicionar um novo namespace ("teste") e um novo parâmetro ao cabeçalho da mensagem (Figura 45).

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:teste="http://br.unirio.webservice">
    <soapenv:Header>
        <teste:username>Y5QL</teste:username>
    </soapenv:Header>
    <soapenv:Body>
        <ser:getCurveNatureList/>
    </soapenv:Body>
</soapenv:Envelope>
```

Figura 45 - Mensagem SOAP alterada para considerar a injeção de um usuário

Como esperado o método exibiu no console do servidor a mensagem "Usuário Injetado: Y5QL", demonstrando que o serviço foi capaz de acessar com sucesso o valor injetado através do cabeçalho da mensagem SOAP de requisição.

5 Propagação de identidade no header HTTP

A proposta de solução contempla a interceptação da mensagem SOAP que é enviada com a requisição ao Web service e posteriormente a interceptação da resposta. Outra possível forma de propagar uma informação consiste em injetar os dados a serem propagados no cabeçalho da mensagem HTTP, ao invés da mensagem SOAP. Esta seção esclarece um pouco sobre esta possibilidade e explica alterações que deveriam ser realizadas em clientes e serviços para que seja possível, respectivamente, injetar um dado no cabeçalho da mensagem HTTP enviada e ler este dado quando a mensagem é recebida.

Kalin [2009], explica o conceito de *context* ("contexto") para a programação, dizendo que este é familiar nos sistemas de programação modernos, incluindo Java. Servlet tem um *ServletContext*, EJB tem um *EJBContext* (com os subtipos apropriados como *SessionContext*), e web services tem um *WebServiceContext*. Do ponto de vista da arquitetura, o contexto é o que dá ao objeto acesso ao seu container (*servlet container*, EJB *container*, *web service container*). Contêineres, por sua vez, fornecem o suporte para o objeto. Do ponto de vista da programação, um contexto é um Map<String, Object>, ou seja, uma coleção chave-valor cujas chaves são textos e os valores são objetos arbitrários.

Faz sentido que o nível de aplicação de um *web service* assuma normalmente o contexto da mensagem, tratando-o como infra-estrutura invisível, ou seja, a aplicação invoca o serviço da mesma forma que invocaria um método de qualquer outra classe Java, e a biblioteca SOAP subjacente cuida de gerar e enviar uma mensagem SOAP para o servidor onde o serviço está publicado. Ao nível do *handler*, o contexto da mensagem está devidamente exposto como os tipo de parâmetros de

retorno, para que um intermediário SOAP ou lógico possa acessar a mensagem SOAP e seu conteúdo (*payload*), respectivamente. Agora vamos ver uma situação mais incomum em que o contexto da mensagem é acessado fora dos manipuladores, ou seja, nos componentes principais da aplicação: o serviço e seus clientes.

Mensagens SOAP são entregues predominantemente sobre HTTP. A questão é, então, quanto da infra-estrutura HTTP é exposta através da *MessageContext* em serviços web baseados em Java. O que vale para HTTP também é válido para os transportes alternativos, tais como SMTP ou mesmo JMS.

Em um *handler* ou serviço implementado com Java é permitido o acesso à mensagens HTTP em um *MessageContext*. Em um cliente baseado em Java, a linguagem também dá acesso ao nível HTTP, mas neste caso, através do *BindingProvider* e os contextos de requisição/resposta, que são expostos como propriedades do *BindingProvider*.

A seguir, mostramos a utilização das classes *MessageContext* e *BindingProvider* para inserção da identidade do usuário no cabeçalho da mensagem HTTP. No serviço, será necessário realizar uma injeção de dependência para utilizar o contexto do web service (*WebServiceContext*). No entanto, esta é uma tarefa executada automanticamente pelo contêiner do serviço. Este modelo de propagação de identidade assemelha-se ao da proposta de solução contida neste documento e explicitada pela seção 3 onde uma informação também é injetada por interceptadores entre os dados que transitam entre cliente e serviço.

A Figura 46 apresenta a codificação da classe que implementa o serviço *ConsultaPoco*, cujo objetivo é expor um método que possibilite a recuperação de um objeto "Poço" dado um determinado atributo identificador. Diferenças aparecem em relação à propagação através da interceptação da mensagem SOAP nos seguintes aspectos:

- o código relacionado ao uso de handlers foi removido e foi adicionado o código referente à utilização do contexto do *web service* e acesso ao cabeçalho HTTP.
- Nas linhas 20 e 21 a classe declara um atributo chamado *webServiceContext* do tipo *WebServiceContext*, que é anotado com @Resource. Essa anotação é usada para solicitar injeção de dependência, fazendo com que o container injete o objeto que contém os dados do contexto neste objeto.
- A classe *WebServiceContext* é usada para acessar a *Message Context* (linha 25) que é transformada em *Map* (linha 26) para obter o cabeçalho de transporte (geralmente HTTP). O cabeçalho HTTP é um *Map*<*String, List*> em que a chave representa um texto e o valor representa um conjunto de valores para aquela chave. Neste caso, é obtida a lista de valores para a chave "username" (linha 27). Se essa lista é não nula, é impresso na saída padrão o valor da primeira posição. Neste exemplo, é esperado que a lista tenha apenas um elemento, visto que cada usuário gera uma requisição diferente para o serviço.

ConsultaPoco.java

```
1  package br.propid.service;
2
3  import java.util.List;
14
15 @WebService(serviceName = "ConsultaPoco",
16         targetNamespace = "http://localhost:7001/ConsultaPocoService")
17 @SOAPBinding(style = SOAPBinding.Style.DOCUMENT, use = SOAPBinding.Use.LITERAL)
18 public class ConsultaPoco {
19
20     @Resource
21     WebServiceContext webServiceContext;
22
23     @WebMethod
24     public Poco getPoco(int id) {
25         MessageContext mctx = webServiceContext.getMessageContext();
26         Map http_headers = (Map) mctx.get(MessageContext.HTTP_REQUEST_HEADERS);
27         List ulist = (List) http_headers.get("Username");
28
29         if (ulist != null) {
30             System.out.println("Usuário no cabeçalho http: " +  ulist.get(0));
31         }
32         Poco poco = new Poco();
33         poco.setId(1);
34         poco.setNome("Poco 1");
35         poco.setEstado("Rio de Janeiro");
36
37         return poco;
38     }
39 }
```

Figura 46 - Classe ConsultaPoco que recupera o usuário do cabeçalho HTTP

A Figura 47 apresenta o código da classe consumidora do serviço com as alterações necessárias. A linha 18 declara um atributo chamado *endpoint* do tipo String para armazenar o endereço do serviço. Da linha 25 até a 29, temos o código adicionado que recupera o contexto de requisição, cria um objeto *hdr* do tipo *Map<String, List<String>>*. No objeto hdr é adicionado um par, "Username" como chave e uma lista com um único objeto ("H3WT") representando a identidade do usuário. Por fim, adiciona-se o objeto *hdr* no contexto de requisição previamente recuperado.

ConsumidorPoco.java

```
1  package br.propid.consumidor;
2
3  import java.util.Collections;
14
15 public class ConsumidorPoco {
16
17
18     private static final String endpoint = "http://localhost:7001/PocoInfo/ConsultaPocoService";
19
20     public static void main(String[] args) {
21
22     ConsultaPocoService service = new ConsultaPocoService();
23     ConsultaPoco port = service.getConsultaPocoPort();
24
25     Map<String, Object> req_ctx = ((BindingProvider) port).getRequestContext();
26     req_ctx.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpoint);
27     Map<String, List<String>> hdr = new HashMap<String, List<String>>();
28     hdr.put("Username", Collections.singletonList("H3WT"));
29     req_ctx.put(MessageContext.HTTP_REQUEST_HEADERS, hdr);
30
31     Poco p = port.getPoco(1);
32
33         if(p != null){
34             System.out.println(p.getId());
35             System.out.println(p.getNome());
36             System.out.println(p.getEstado());
37         }
38     }
39 }
```

Figura 47– Classe ConsumidorPoco que adiciona usuário no cabeçalho HTTP

Esta solução, assim como a solução com a inclusão da identidade do usuário no *header* da mensagem SOAP, introduz poucas linhas de código nas classes cliente e provedora do serviço.

A solução apresentada neste trabalho baseou-se no uso do protocolo SOAP porque este é independe do protocolo de transporte sobre o qual a mensagem será enviada.

6 Conclusões

Segurança da Informação é um importante desafio para a maioria das organizações. De forma geral, problemas nesta área são solucionados através da implementação de mecanismos para controle de acesso e de regras de autorização nas próprias aplicações. Entretanto, quando tais regras se alteram, é necessário atualizar todos os sistemas que possuem tais regras implementadas, tarefa de grande complexidade, especialmente em cenários compostos por aplicações legadas e um grande número de regras de autorização. Logo, centralizar as regras no banco de dados garantindo uniformidade no acesso a dados é uma solução importante e vem sendo estudada por vários trabalhos da literatura [DoD, 1983; Ferraiolo e Khun, 1992; Sandhu *et al.*, 1996; Jeloka *et al.*, 2008; Azevedo *et al.*, 2010]. No entanto, para que o banco de dados possa aplicar as regras adequadamente, é necessário que as informações do usuário da aplicação sejam propagadas até o banco de dados de forma adequada.

Este trabalho apresentou uma solução para controle de acesso em uma arquitetura que faz uso de serviços web, com foco em propagação de identidade e aplicação de regras de autorização. Duas versões de web services foram utilizadas JAX-RPC e JAX-WS para implementar a solução. Em ambos os casos, foram implementados SOAP *handlers* responsáveis por interceptar requisições a serviços e injetar a identidade real do usuário acessando a aplicação cliente na mensagem SOAP, de modo a propagar tal identidade até o SGBD, permitindo assim que mecanismos de controle de acesso no SGBD possam ser aplicados efetivamente.

Testes experimentais foram executados para testar o isolamento da solução e desempenho da proposta. Em relação ao isolamento, foi avaliado se o usuário executando a aplicação cliente é propagado até o banco de dados, e se apenas os dados que este usuário tem acesso são disponibilizados, sem interferência de outras conexões concorrentes. Os resultados dos testes confirmaram isolamento. Além disso, observou-se baixíssimo impacto a sistemas que utilizam web services, facilitando a tarefa de mantê-los atualizados. Para realizar a propagação da identidade entre as camadas de aplicação, quem consome o serviço JAX-WS (envia a identidade do usuário) precisa-se configurar um arquivo com a cadeia de *handlers* (p.e., "*handlerchain.xml*" - Figura 24) e incluir no *stub* que representa o serviço invocado uma anotação para considerar a cadeia (anotação @HandlerChain - Figura 25). No serviço que atende à requisição (recebe a identidade do usuário) também é necessário configurar o arquivo da cadeia de *handlers* (p.e., "*handlerchain.xml*" - Figura 24) e indicar que o *handler* deve ser utilizado quando a mensagem de entrada for recebida (inserção da anotação @HandlerChain - Figura 26). Já para propagar a identidade para o contexto do SGBD, propomos o uso de uma nova fábrica de sessões para injeção do usuário (uma proposta de método para criação da conexão é apresentado na Figura 29). Em relação ao teste de desempenho, os resultados demonstraram que o overhead é muito pequeno (média de 14%), especialmente quando considera-se o tempo médio por rquisição (aumento de 0,042 segundo por requisição para 0,0478 segundo por requisição). No caso do acesso a serviços JAX-RPC, a cadeia de handlers é descrita diretamente na anotação "@SOAPMessageHandlers"

A principal premissa deste trabalho é o uso de serviços web implementados utilizando tecnologia Java. Como trabalho futuro temos a implementação da solução empregando outras tecnologias como, por exemplo, .Net e o próprio uso da solução

em ambientes em que ambas as tecnologias aparecem. Além disso, propomos expandir a solução para tratar outras questões de segurança como, por exemplo, autenticação e confidencialidade.

Referências Bibliográficas

AASATO, L. *et al.* **Oracle Web Services Manager, Administrator's Guide, 10g (10.1.3.1.0).** Oracle Corporation, 2007. Disponível em <http://download.oracle.com/docs/cd/B31017_01/integrate.1013/b31008.pdf>. Acessado em Outubro de 2011.

AZEVEDO, L.; DUARTE, D.; PUNTAR, S.; ROMEIRO, C.; BAIAO, F.; CAPPELLI, C. **Avaliação de Ferramentas para Gestão e Execução de Regras de Autorização.** Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), RT-0027/2009, 2009a. Disponível em <http://www.seer.unirio.br/index.php/monografiasppgi>. Acessado em Outubro de 2011.

AZEVEDO, L.; SOUSA, H. P.; BAIAO, F.; SANTORO, F. **Desenvolvendo web services no BEA Workshop for WebLogic Platform**. Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), RT-0014/2009, 2009b. Disponível em <http://www.seer.unirio.br/index.php/monografiasppgi>. Acessado em Outubro de 2011.

AZEVEDO, L.G. *et al.* **A Flexible Framework for Applying Data Access Authorization Business Rules**. In Proceedings of the 12th International Conference on Enterprise Information Systems (ICEIS 2010). Funchal, Madeira, Portugal , pp. 275-280, 2010.

BEUST, C., SULEIMAN, H. **Next Generation Java Testing: TestNG and Advanced Concepts**. Publicado por Addison-Wesley Professional , ISBN-10: 0321503104, 2007.

BRG . **The Business Rules Group**. http://www.businessrulesgroup.org, 2009.

BUTEK R.; GALLARDO; N. **Web services hints and tips: JAX-RPC versus JAX-WS, Part 2**. IBM, 2006. Disponível em <http://www.ibm.com/developerworks/webservices/library/ws-tip-jaxwsrpc2.html>. Acessado em Outubro de 2011.

CANTOR, J., KEMP, J., PHILPOTT, R., MALER, E. **Assertions and Protocols for the OASIS, Security Assertion Markup Language (SAML) V2.0.** OASIS (Organization for the Advancement of Structured Information Standards) Standard, 2005**.**

CHRISTENSEN, E., CURBERA, F., MEREDITH, G., WEERAWARANA, S. **Web Services Description Language (WSDL) 1.1**, Disponível em: < http://www.w3.org/TR/wsdl>. Acessado em Novembro de 2011.

Erl, T. **Service-Oriented Architecture: concepts, technology, and Design**. Prentice Hall, Crawfordsville: Indiana, 792 p, 2005.

GARCÍA, C. A. G., PATON, E. F.-M.,VELTHIUS, M. P. **Web Services Security**. In *Web and Information Security,* ed. Elena Ferrari and Bhavani Thuraisingham, 32-51 (2006).

GUDGIN, M., HADLEY, M., MENDELSOHN, N. *et al.* **SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)**. World Wide Web Consortium, April 2007. Disponível em <http://www.w3.org/TR/2007/REC-soap12-part1-20070427/>. Acessado em Outubro de 2011.

HUGHES, J., EVE, M., PHILPOTT, R., **Technical Overview of the OASIS Security Assertion Markup Language (SAML).** OASIS (Organization for the Advancement of Structured Information Standards) Standard, 2004**.**

JELOKA, S., MULAGUND, G., LEWIS N. et al. (2008). ―Oracle Database Security Guide, Oracle RDBMS 10gR2‖. Oracle Corporation. http://download.oracle.com/docs/cd/B19306_01/network.102/b14266.pdf.

JOHNSON, R., HOELLER, J., DONALD, K. et al., **Spring Framework Reference Documentation (3.0)**, 2010. Disponível em <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/pdf/spring-framework-reference.pdf>. Acessado em Maio de 2011.

Josuttis, N. M. **SOA in Practice: The Art of Distributed System Design**. O'Reilly, 2007.

KALIN, M. **Java Web Services: Up and Running.** O'Reilly, 2009.

KING, G., BAUER, C., BERNARD, E., EBERSOLE, S. **Hibernate Getting Started Guide**. Red Hat, Inc., 2010a.

KING, G., BAUER, C., ANDERSEN, M. R. *et al* **Hibernate Core Reference Manual**. Red Hat, Inc., 2010b.

LEÃO, F., PUNTAR, S., AZEVEDO, L. G., CAPPELLI, C., BAIÃO, F. **Controle de Acesso a Dados em Sistemas de Informação através de Mecanismos de Propagação de Identidade e Execução de Regras de Autorização**. In: VII Simpósio Brasileiro de Sistemas de Informação (SBSI 2011), Salvador, Brazil, 2011.

MURTHY, R., SEDLAR, E. **Flexible and efficient access control in oracle**. In *ACM SIGMOD 2007*, pp. 973-980, Beijing, 2007.

ORACLE, 2008. Programming Advanced Features of WebLogic Web Services Using JAX-WS, 10g Release 3. Disponível em: < http://download.oracle.com/docs/cd/E12840_01/wls/docs103/pdf/webserv_adv.pdf>, Acessado em Novembro de 2008.

ORACLE, 2011. **WSimport – Java TM API for XML Web Services (JAX-WS) 2.0.** Disponível em: < http://download.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html>. Acessado em Novembro de 2011.

PATRICK, R., NYBERG, G., ASTON, P. **Professional Oracle WebLogic Server.** Wiley Publishing, 2010.

PRESSMAN, R. S. **Software engineering: a practitioner's approach**. McGraw-Hill Higher Education, 7. ed., 2010.

SANDHU, R.S., COYNE, E.J., FEINSTEIN, H.L., YOUMAN, C.E. **Role-based access control models**. IEEE Computer, vol. 29, no. 2, pp 38-47, 1996.

VOSS, G. **Introducing Java Beans**. Tutorials & Code Camps, Oracle Sun Developer Network (SDN), 1996. Disponível em <http://java.sun.com/developer/onlineTraining/Beans/Beans1/>. Acessado em Abril de 2011.

YANG, L. **Teaching database security and auditing**. ACM SIGCSE'09, v.1, issue 1, pp. 241—245, 2009.

TPC Council, 2008, *TPC Benchmark H Standard Specification Revision 2.8.0*. Transaction Processing Performance Council. Disponível em <http://www.tpc.org/tpch/spec/tpch2.8.0.pdf>. Acessado em Outubro de 2011.

W3C, **Web Services Architecture**. World Wide Web Consortium (W3C). 2004. Disponível em < http://www.w3.org/TR/ws-arch>. Acessado em Outubro 2011.

Apêndice 1 – Exemplo de JavaBean

A seguir é apresentado um exemplo de JavaBean.

```
package beans;

/**
 * Class <code>PersonBean</code>.
 */
public class PersonBean implements java.io.Serializable {

    private String name;

    private boolean deceased;

    /** No-arg constructor (takes no arguments). */
    public PersonBean() {
    }

    /**
     * Property <code>name</code> (note capitalization)
readable/writable.
     */
    public String getName() {
        return this.name;
    }

    /**
     * Setter for property <code>name</code>.
     * @param name
     */
    public void setName(final String name) {
        this.name = name;
    }

    /**
     * Getter for property "deceased"
     * Different syntax for a boolean field (is vs. get)
     */
    public boolean isDeceased() {
        return this.deceased;
    }

    /**
     * Setter for property <code>deceased</code>.
     * @param deceased
     */
    public void setDeceased(final boolean deceased) {
        this.deceased = deceased;
    }
}
```

Apêndice 2 – Criação de projetos JAX-RPC e JAX-WS

Na seção 3.1 foi mostrado, brevemente, o método para a criação de um serviço em JAX-WS através da IDE de programação *Oracle Workshop for Weblogic* (baseada na IDE Eclipse). Nesta seção serão dadas explicações mais detalhadas sobre o procedimento de criação e no que este deve diferir para que um projeto JAX-RPC seja criado.

Criando um projeto JAX-WS

Para criar um serviço que faça uso da tecnologia JAX-WS (evolução do JAX-RPC), deve-se estabelecer um projeto JAX-WS. Para tal, ao utilizar-se a IDE de programação Oracle Workshop for Weblogic, deve-se selecionar dentro do menu "File > New" o subdiretório "Web Services" a opção Web Service Project, como ilustrado pela Figura 48.

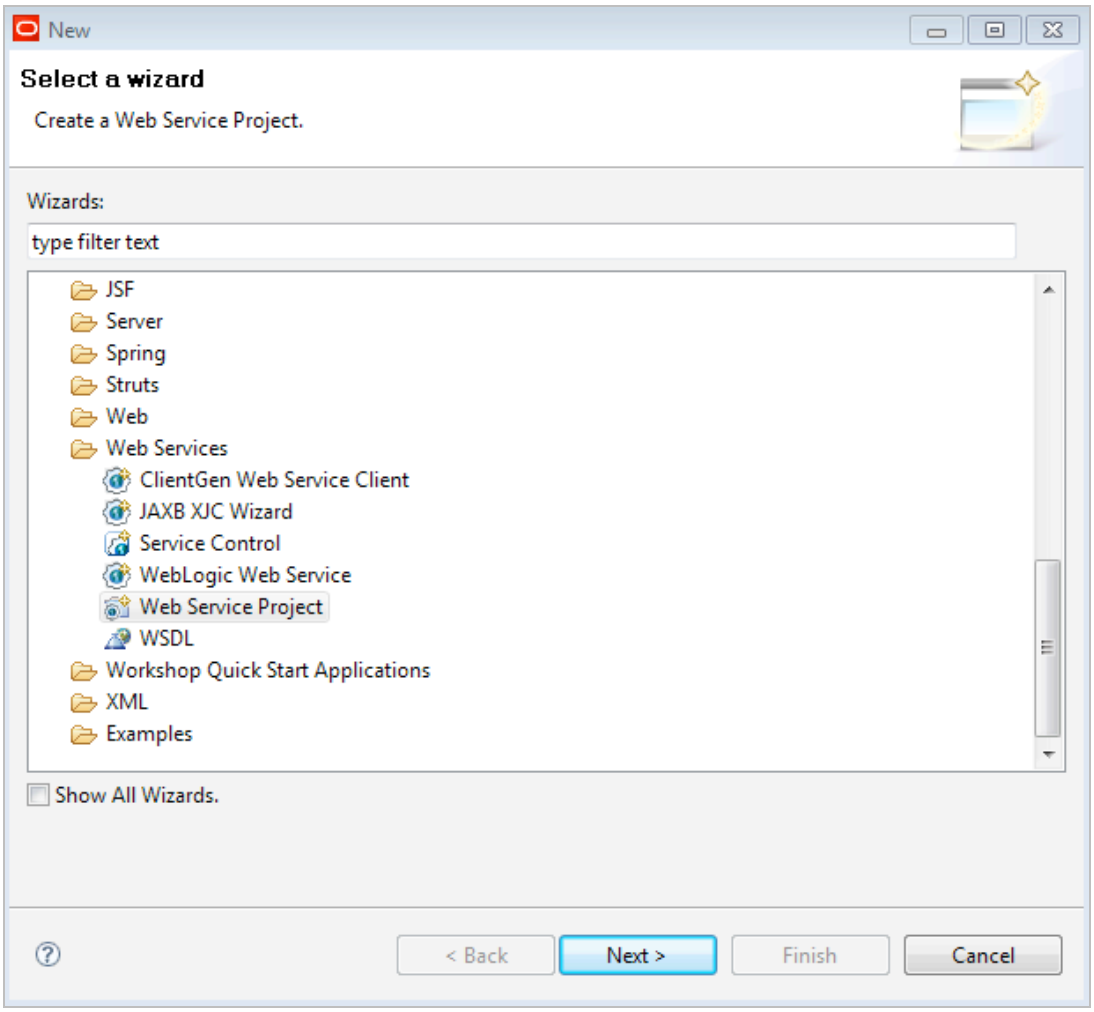

Figura 48 - Criando um novo projeto para web service

Na tela seguinte deve-se preencher o nome do projeto, o local aonde ele deverá ser criado no computador (por padrão a IDE irá criá-lo no diretório "Workspace/default" dentro do diretório "user_projects") e a configuração do tipo de projeto a ser utilizado, neste caso o "*Annotated Web Services Facets JAX-WS (Recommended)*" (Figura 49).

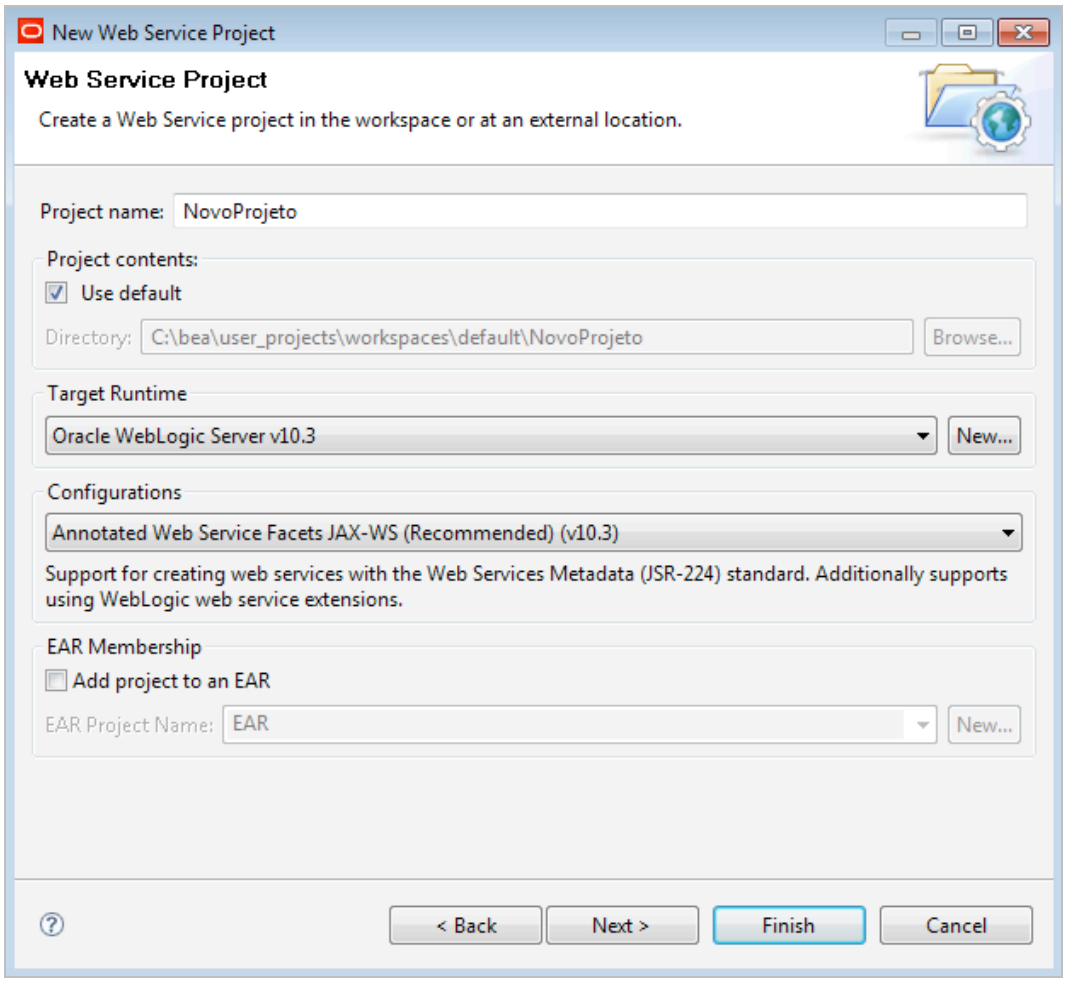

Figura 49 - Configuração do projeto JAX-WS

Na terceira tela de criação do projeto devem ser selecionadas as bibliotecas e *frameworks* a serem utilizados pelo projeto. Para Um projeto básico bastaria manter as opções já selecionadas pelo próprio *wizard* da IDE, enumerados na Figura 50.

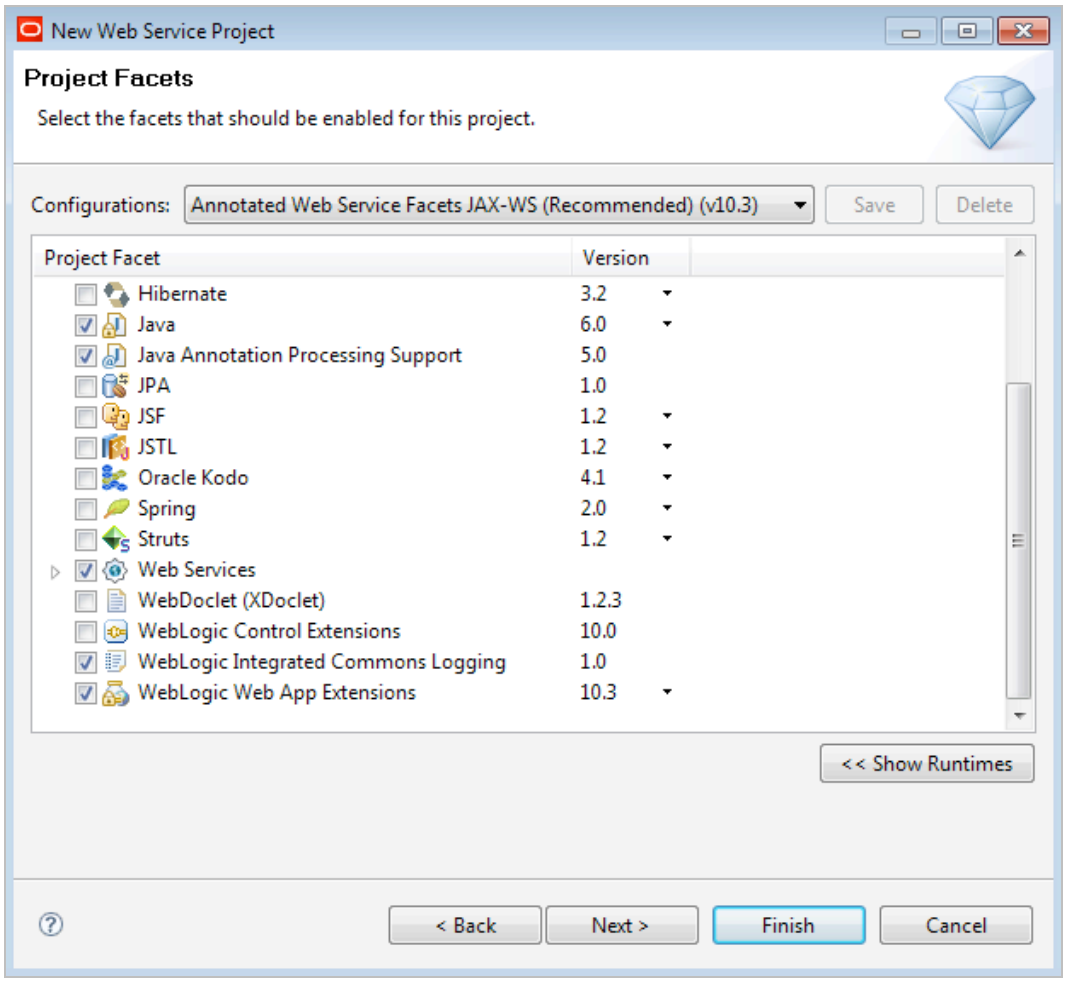

Figura 50 - Bibliotecas e *frameworks* necessários ao projeto JAX-WS

Após estas configurações basta ao usuário finalizar o *wizard* clicando no botão "*finish*".

Quanto à criação de web services dentro do projeto, deve-se utilizar o *wizard* novamente, porém desta vez, ao aparecer a tela apresentada pela Figura 48, deve-se selecionar dentro do subdiretório "Web Services" a opção "Weblogic Webservice".

Uma vez que a classe representante do serviço tenha sido criada basta implementar a lógica desejada. Em casos simples, não se faz necessário criar métodos para serialização em XML ou criar manualmente arquivos XSD e WSDL. A simples implantação do serviço no servidor (através do comando "run as") irá gerar os arquivos necessários para que o serviço seja acessado.

Criando um projeto JAX-RPC

A criação de um serviço utilizando a tecnologia JAX-RPC é muito similar à criação de serviço utilizando a tecnologia JAX-WS. Uma diferença é no momento da criação do projeto onde o serviço será codificado. É preciso informar a configuração do projeto que neste caso é "*Annotated Web Service Facets JAX-RPC*", como mostrado na Figura 51.

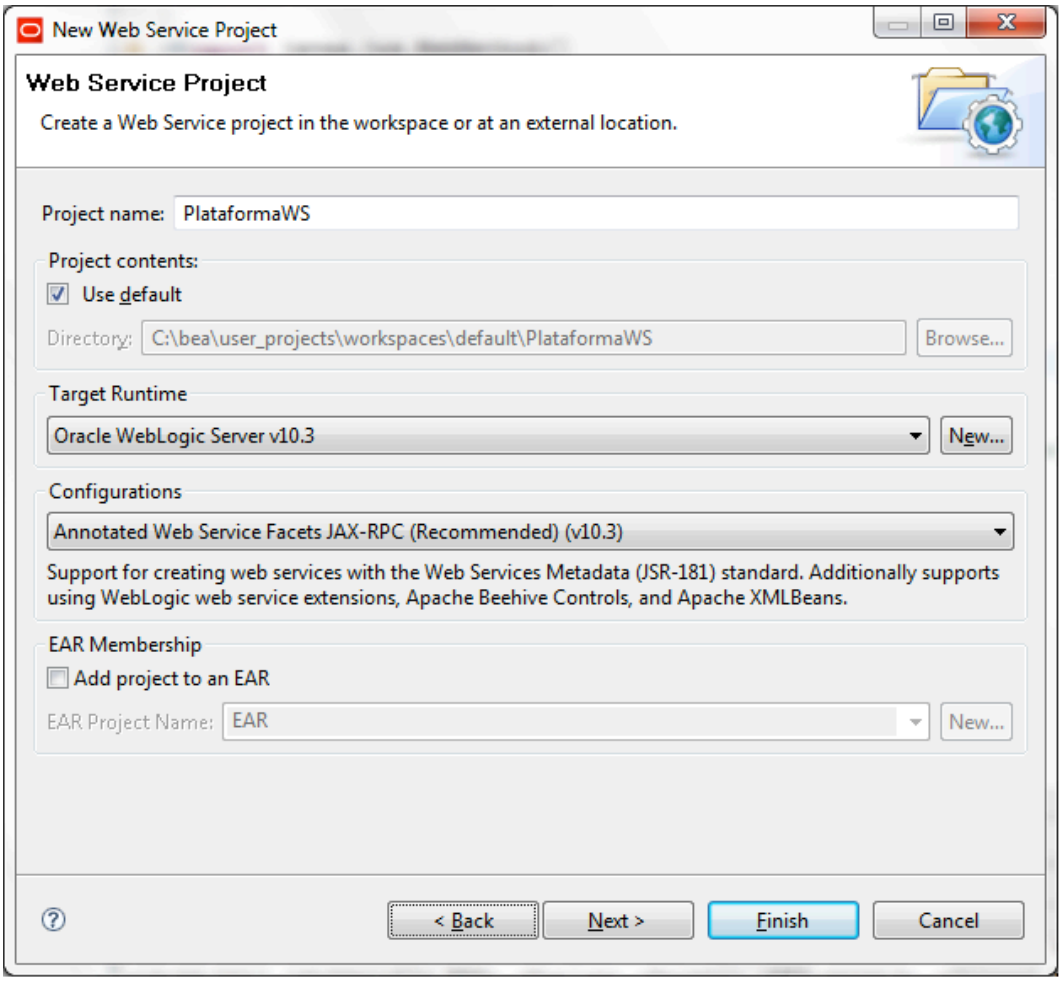

Figura 51 – Criação de Serviço JAX-RPC

Após a criação, a codificação do serviço é feita da mesma forma. No entanto, é necessário criar métodos para serialização em XML e criar manualmente arquivos XSD. Azevedo *et al.* [2009b] apresentam a criação de serviços JAX-RPC weblogic.

Apêndice 3 – Handlers e JAX-RPC

A utilização de Handlers na especificação JAX-RPC é diferente da mostrada anteriormente que segue a especificação JAX-WS. Uma característica importante é que essa configuração dá suporte às tecnologias Apache Beehive Controls e Apache XML Beans o que não acontece ao utilizar JAX-WS.

Para um serviço web ou consumidor de serviço web, podem existir zero ou mais *handlers*. Uma coleção de *handlers* constitui uma cadeia de *handlers* (*handler chain)* que é mantida pela implementação em tempo de execução JAX-RPC. O seu comportamento padrão é chamar cada *handler* da cadeia. Entretanto, um *handler* pode mudar esse comportamento através da sua implementação da interface javax.xml.rpc.handler.Handler. Por exemplo, retornar *false* no método *handleMessage* vai impedir que a *runtime* prossiga para o próximo *handler* da cadeia. Lançar uma execeção causa um efeito similar.

Em última instância, um *handler* JAX-RPC é configurado e registrado no descritor de implantação do Web Service (webservices.xml). Se a implantação do serviço web é feita por tarefas *Ant* não há necessidade de adicionar os *handlers* no arquivo webservices.xml. Se a implantação é feita por linha de comando ou manualmente, os *handlers* são adicionados pela insersão de elementos filho do elemento <*port-component*>.

No lado cliente, *handlers* JAX-RPC podem interceptar e processar mensagens enviadas da aplicação cliente e a mensagem de retorno correspondente do serviço web, e por exemplo, processar a mensagem SOAP. Os *handlers* podem ser registrados usando a tarefa *genProxy* do *Ant*. Na Figura 52 o *handler* br.unirio.ClientPocoHandler é disponibilizado para o cliente.

```
1<oracle:genProxy
2            wsdl="http://localhost:7001/PocoInfo/ConsultaPoco?wsdl"
3            output="build/src/client"
4            packageName="br.unirio">
5            <oracle:handler
6                  name="ClientPocoHandler"
7                  handlerClass="br.unirio.ClientPocoHandler" />
8</oracle:genProxy>
```

Figura 52 - Tarefa genProxy do Ant

Apêndice 4 – *Frameworks* para Testes Unitários

Pressman [2010] define teste unitário como o teste que foca na menor unidade compilável de um programa - o subprograma (por exemplo, componentes, subrotinas e procedimentos).

Frameworks específicos podem ser utilizados na tarefa de automatizar tais testes, possibilitando repeti-los sempre que desejado como, por exemplo, a cada nova versão do sistema, garantindo que funcionalidades legadas não deixaram de se comportar como esperado devido às novas adições. Estes *frameworks* permitem também ooutros tipos de testes além dos unitários como os de caixa preta, caixa branca, caixa cinza, regressão, etc [Pressman, 2010].

Dois *frameworks* para automatização de testes muito utilizados no mercado são o JUnit[4] e o TestNG[5]. Ambos possuem versões atuais e se assemelham em muitos casos, dispondo muitas vezes dos mesmos mecanismos, como testes de exceções, métodos de configurações, anotações e timeouts [Beust e Suleiman, 2007]. Entretanto, o TestNG fornece nativamente uma solução para testes de paralelismos, essencial para a execução dos testes de concorrência realizados neste trabalho. Tal funcionalidade só pode ser utilizada no *framework* JUnit quando este é combinado ao plugin "Parallel-Junit", que requer configuração adicional.

Criando um Caso de Teste com JUnit

A seguir é demonstrada a criação de um caso de teste. O exemplo foi desenvolvido utilizando-se o *framework* JUnit, entretanto todas as técnicas descritas (uso de anotações, métodos de configurações e funções assertivas, por exemplo) estão também presentes no *framework* TestNG, na verdade, até seus nomes são iguais em alguns casos e semelhantes em outros.

Algumas IDEs de programação ja possuem como *framework* padrão para o desenvolvimento de testes unitários o JUnit (como por exemplo o Eclipse e o Netbeans, em suas versões mais recentes). Alternativamente é possível realizar a instalação dos plugins através dos próprios gerenciadores de plugins das IDEs. Para o exemplo de caso de teste descrito a seguir foi utilizada a IDE Eclipse (versão Indigo) junto ao JUnit versão 4.1.0. Para utilizar o *framework* foi necessário simplesmente adicionar ao *classpath* do projeto a biblioteca referente a versão 4 do JUnit.

Para exemplificar a utilização de um caso de testes criou-se inicialmente um novo projeto (Figura 53), que define uma classe "TesteUtils" com dois métodos: *olaMundo()* e *somar()* (Figura 54).

[4] http://www.junit.org/

[5] http://testng.org/

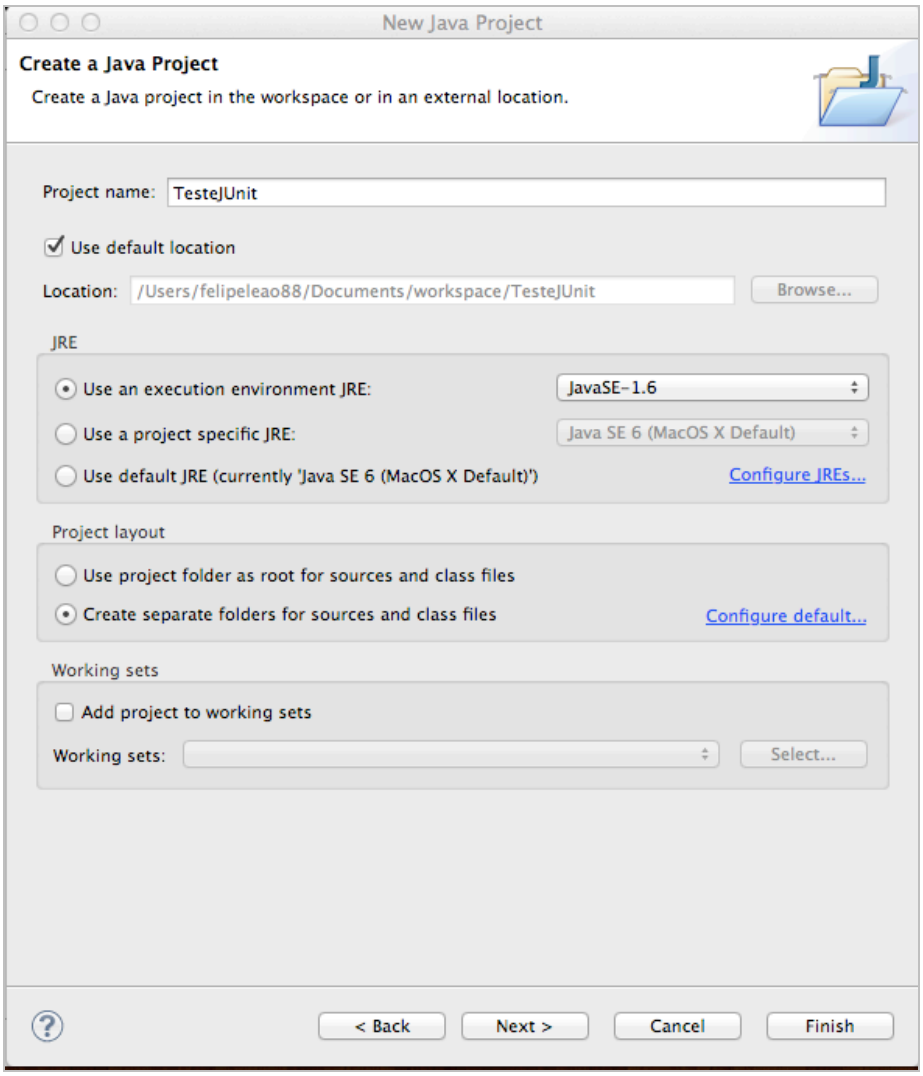

Figura 53 - Criação do projeto para testes com JUnit

```
package br.uniriotec.np2tec.propid.c4.classes;

public class TesteUtils {

    public String olaMundo(){
        return "Ola Mundo!";
    }

    public double somar(double x, double y){
        return x+y;
    }

}
```

Figura 54 - Implementação da classe TesteUtils

Uma vez criado o projeto e a classe que será testada, foi criado um novo diretório no projeto, para comportar todas as classes de teste (passo opcional), este diretório recebeu o nome de "Test".

Para criar o caso de teste basta clicar com o botão direito sobre a classe que se deseja testar. Feito isso deve-se selecionar no menu "Novo → Outro" sob a seção "Java/JUnit" a opção JUnitTestCase. Uma tela de configuração similar a apresentada pela Figura 55 será exibida.

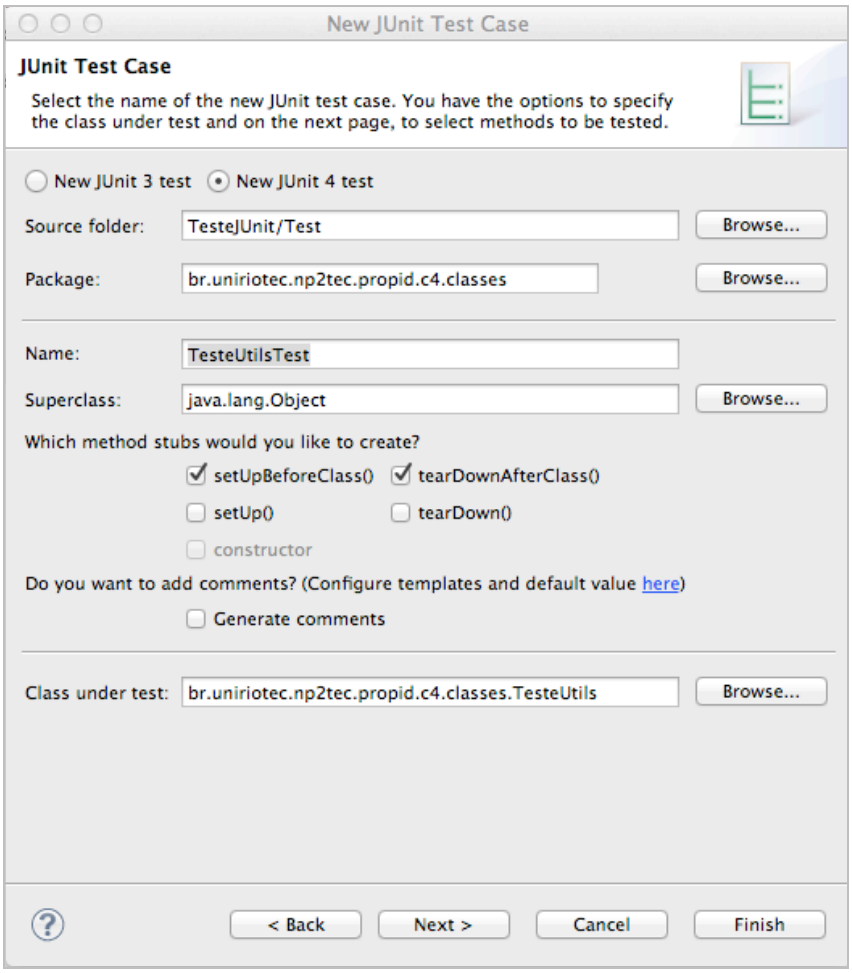

Figura 55 - Configuração do caso de teste Unitário

Foi definido que o teste deverá ser implementado utilizando-se a versão 4 do JUnit, que o teste deverá ser armazenado no diretório "Test" dentro de um pacote com o mesmo nome do pacote onde se encontra a classe testada. Comumente a classe onde se implementam os testes recebe o nome da classe testada concatenando-se ao fim do nome o sufixo "*test*". É possível ainda indicar durante a configuração quais métodos da classe original serão testados, neste caso serão gerados autmaticamente métodos de teste vazios.

A anotação BeforeClass `@BeforeClass` (criada a partir da marcação *setUpBeforeClass*) permite indicar um método `public static void` sem argumentos para ser executado uma vez antes de qualquer método de teste. Já a anotação *tearDownAfterClass* AfterClass `@AfterClass` (criada a partir da marcação *tearDownAfterClass*) permite definir um método `public static void` para ser executada depois de todos os testes na classe tenham sido executados. Todos os métodos anotados com `@AfterClass` garantidamente executam mesmo que um método `BeforeClass` levante uma exceção.

A Figura 56 mostra a classe de testes gerada. Apesar de presente, o método *tearDownAfterClass()* (anotado por *@AfterClass*) não foi utilizado. O método *setUpBeforeClass()* (anotado por *@BeforeClass*) é executado pelo *framework* logo que a classe é inicializada.

```java
package br.uniriotec.np2tec.propid.c4.classes;

import static org.junit.Assert.*;

public class TesteUtilsTest {

    private static TesteUtils utils;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        utils = new TesteUtils();
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    @Test
    public void testOlaMundo() {
        assertEquals("Retorno diferente do esperado", "Ola Mundo!", utils.olaMundo());
    }

    @Test
    public void testSomar() {
        assertEquals(8.32, utils.somar(3.30, 5.02), 0.0001);
    }

}
```

Figura 56 - Implementação do Caso de Teste

Os métodos precedidos pela anotação *@Test* são aqueles que implementam a lógica de teste. Como visto, são métodos simples, públicos, sem retorno (void) que recebem o mesmo nome que os métodos a serem testados presentes na classe original, porém precedidos pelo prefixo "*test*" (em versões anteriores do Junit, antes do *framewrok* fazer uso de anotações, a presença do prefixo "*test*" indicava que o método era um teste). Para estes testes a lógica consistiu simplesmente em invocar o método original e verificar se o valor de retorno coincidia com o valor esperado. Tal verificação é feita pela função *assertEquals*. Esta função é automaticamente localizada pelo *framework* (neste caso JUnit) e seu resultado adicionado ao relatório de testes gerado. Quando utilizado para testar métodos que retornam *String*, o *assertEquals* recebe três parâmetros, onde o primeiro é a mensagem a ser lançada para o caso do teste falhar, o segundo é o valor que se espera como retorno do método e o terceiro é o retorno do método em si (neste caso foi feit diretamente a invocação do método). Quando utilizado para verificar métodos que retornam *Double*, o primeiro parâmetro corresponde ao valor esperado como retorno, o segundo ao valor retornado pelo método e o terceiro à margem de erro considerada aceitável.

A Figura 57 demonstra o relatório de testes fornecido pelo *framework* JUnit, indicando quais testes foram bem sucedidos e quais foram mal sucedidos, apontando ainda o motivo da falha de cada testes.

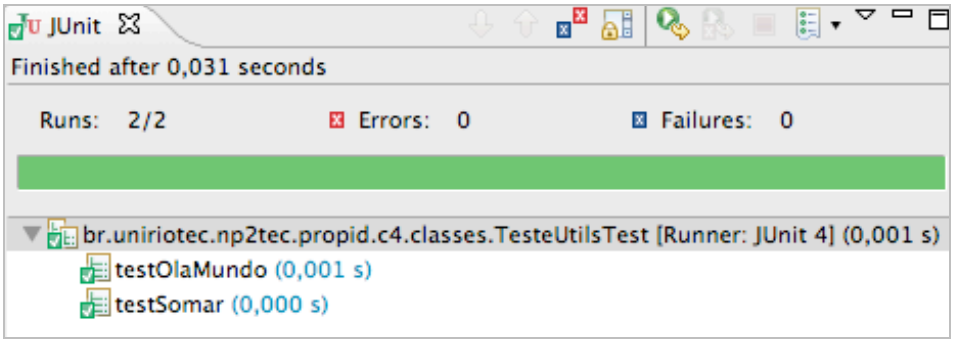

Figura 57 - Relatório de Testes JUnit

Criando um Teste Paralelos com TestNG

Por fornecer uma solução nativa que provê funcionalidades para testes paralelos (característica essencial ao teste de concorrência), optou-se pelo *framework* TestNG para a implementação dos testes de concorrência neste trabalho.

Ao contrário do JUnit, não é comum que IDEs venham com o plugin disponível "de fábrica", portanto se faz necesário que o usuário realize o download da versão mais atual diretamente do repositório do projeto. Recomenda-se que o passo-a-passo para a instalação do plugin seja visto no site do próprio projeto (http://testng.org/doc/download.html).

A criação de testes paralelos ocorre inicialmente de maneira muito similar a criação de testes unitários no JUnit. Para a classe que possui o método que se pretende testar, uma classe de teste deve ser criada. É importante observar que para o caso de o método estar exposto por outro modo, como um EJB ou um webservice, uma classe comum pode ser criada e métodos de testes podem ser manualmente definidos importando-se pacotes e classes como as anotações @Test, @BeforeClass, etc.

Para exemplificar a execução de testes paralelos foi utilizada a classe implementada para definir os métodos *somar()* e *olaMundo()* no teste com o *framework* JUnit. Ao clicar com o botão direito sobre a classe TesteUtils, selecionou-se o menu "Novo → Outro" e sob a seção "TestNG" a opção TestNG Class. A seguir selecionou-se o método "*somar()*". Outros parâmetros da classe de teste são exibidos pela Figura 58.

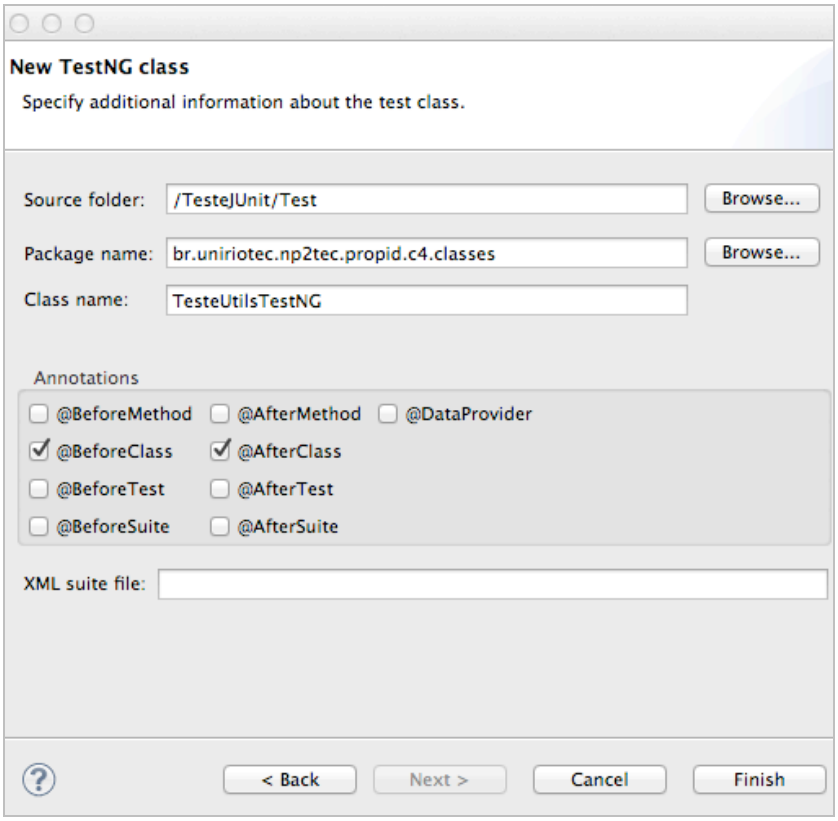

Figura 58 - Configurações para a classe de teste do TestNG

O teste executado consiste em realizar 1000 vezes a invocação do método *somar()*, passando valores diferentes a cada invocação (obtidos através do método *Math.random()*). Foi permitido ao *framework* utilizar até 5 diferentes *threads* para executar os testes, desta forma, com *threads* sendo executadas simultaneamente, tem-se uma situação de invocações paralelas do método *somar()*.

Para especificar a quantidade de invocações e quantas *threads* devem ser consideradas, o TestNG permite a adaptação da anotação *@Test* para comportar a especificação de parâmetros, neste caso o *threadPoolSize* e o *invocationCount*. A Figura 59 mostra a implementação do teste de paralelismo.

```
package br.uniriotec.np2tec.propid.c4.classes;

import org.testng.Assert;
import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

public class TesteUtilsTestNG {
    private static TesteUtils utils;

    @BeforeClass
    public void beforeClass() {
        utils = new TesteUtils();
    }

    @AfterClass
    public void afterClass() {

    }


    @Test(threadPoolSize = 5, invocationCount = 1000)
    public void somar() {
        double x = Math.random();
        double y = Math.random();

        Assert.assertEquals(x+y, utils.somar(x, y), 0.0001);

    }
}
```

Figura 59 - Implementação da classe de testes com TestNG

Assim como o JUnit, o TestNG exibe um relatório com a quantidade de testes realizados (considerando inclusive as múltiplas invocações de um mesmo método), de sucessos e de falhas. O *framework* fornece ainda um relatório mais detalhado em XML e possibilita sua visualização em HTML. A Figura 60 mostra a versão mais simples deste relatório, exibida através da própria interface da IDE Eclipse. O relatório em questão indica que as 1000 invocações do método *somar()* foram realizadas com sucesso e nenhuma falha foi detectada.

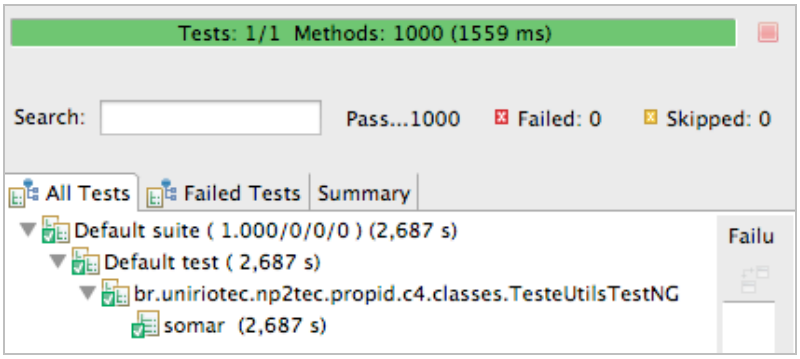

Figura 60 - Relatório de Testes do TestNG

É importante observar que o teste de paralelismo aqui apresentado consiste em meramente invocar múltiplas vezes o mesmo método e garantir que em todas as invocações ele se comporta de acordo com o esperado; portanto, o fato de tais invocações ocorrerem de forma paralela não ocasiona real interferência no resultado. Testes mais complexos podem ser realizados fazendo-se uso desta funcionalidade, como por exemplo testes de concorrência (utilizando-se atributos, artefatos ou dados concorrentemente) e testes de desempenho (onde poderia-se

avaliar, por exemplo, o desempenho de um determinado servidor ao responder a um alto número de requisições em paralelo).

Considerações Finais sobre os *Frameworks*

Ambos os *frameworks* possuem funcionalidades essenciais a iniciantes e àqueles que fazem uso de técnicas mais avançadas de testes. O TestNG, em especial, procura oferecer soluções para determinados aspectos de testes não abordados pelo JUnit, ou então propor uma visão alternativa sobre como determinados testes devem ocorrer. Neste trabalho em especial (mais especificamente na seção 3.5) optou-se pelo TestNG em detrimento do JUnit devido à facilidade de uso do TestNG para realizar os testes de concorrência necessários para avaliar a proposta.