Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma Implementação do Protocolo OAuth 2 em Erlang para uma Arquitetura Orientada a Serviço

Alysson de Sousa Ribeiro

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Orientadora
Prof.a Dr.a Edna Dias Canedo

Brasília
2017

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

de Sousa Ribeiro, Alysson
dD467i Uma Implementação do Protocolo OAuth 2.0 em Erlang para uma Arquitetura Orientada a Serviço / Alysson de Sousa Ribeiro; orientador Edna Dias Canedo. -- Brasília, 2017.
93 p.

Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) -- Universidade de Brasília, 2017.

1. Segurança. 2. OAuth 2.0. 3. Autorização. 4. Arquitetura Orientada a Serviço (SOA). 5. Enterprise Service Bus (ESB). I. Dias Canedo, Edna, orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma Implementação do Protocolo OAuth 2 em Erlang para uma Arquitetura Orientada a Serviço

Alysson de Sousa Ribeiro

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof.a Dr.a Edna Dias Canedo (Orientadora)
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
FGA/UnB/PPCA

Prof. Dr. Eduardo Luzeiro Feitosa
IC/UFAM

Prof. Dr. Marcelo Ladeira
Coordenador do Programa de Pós-graduação em Computação Aplicada

Brasília, 17 de julho de 2017

Dedicatória

Dedico esse trabalho aos meus pais Maria Fátima e Adonato, ao meu irmão Andersson, à minha esposa Aline e ao meu filho Artur.

Agradecimentos

Agradeço à minha orientadora, a Prof.ª Dr.ª Edna Dias Canedo pelo suporte essencial durante este trabalho.

Aos professores e colegas do Mestrado Profissional em Computação Aplicada por todo o conhecimento compartilhado e em especial aos amigos Reinaldo Baldoino, Renan Costa e Sigfredo Rocha, que estiveram comigo desde o início do curso.

Aos colegas Everton Vargas Aguilar e Ranato Carauta pela valiosa contribuição técnica durante a implementação da solução desenvolvida neste trabalho.

Aos amigos Marcelo Karam, Domingos Pereira e Juvenal Barreto pelo apoio e incentivo.

Aos colegas do CPD pelo constante auxílio e compreensão durante o mestrado.

Resumo

A utilização da Arquitetura Orientada a Serviço (SOA) oferece alguns benefícios, tais como: baixo acoplamento e interoperabilidade, sendo bastante utilizada para a integração de aplicações dentro de uma organização. Essa característica faz com que a arquitetura orientada a serviço seja utilizada na modernização de sistemas legados. No entanto, a sua implantação ainda merece alguns cuidados relacionados aos problemas de segurança. Este trabalho apresenta um mapeamento sistemático a cerca dos mecanismos de autenticação e autorização em SOA e levanta algumas questões de pesquisa, bem como alguns protocolos utilizados em SOA. Como resultado deste mapeamento foi identificado uma solução de autorização considerada adequada para a arquitetura utilizada pelo CPD para modernizar os seus sistemas legados. O protocolo OAuth 2.0 foi implementado no *Enterprise Service Bus* (ESB) que será utilizado para a modernização dos sistemas legados da UnB. Foram realizados testes de desempenho na solução permitindo verificar o aumento da latência introduzida pelo protocolo e a vazão média suportada. Foram realizadas ainda simulações de segurança com o objetivo de verificar o comportamento do protocolo implementado quando exposto a uma ataque de repetição.

Palavras-chave: Segurança, OAuth 2.0, Autorização, *SOA*, ESB, Erlang, ErlangMS, Modernização dos Sistemas Legados.

Abstract

The utilization of Service-Oriented Architecture (SOA) offers certain benefits, such as low coupling and interoperability. It widely used for the integration of applications within an organization. This characteristic makes it so service-oriented architecture is used in the modernization of legacy systems, being thoroughly discussed and used as an architecture solution for the modernization of the legacy systems of the IT Center (CPD) of University of Brasília (UnB). Nevertheless, its implementation still requires some care related to the security problems. This study presents a systematic mapping regarding the authentication and authorization mechanisms in SOA, and raises some research questions, as well as some of the protocols used in SOA. As a result of the mapping, an authorization solution considered adequate for the architecture used by the CPD to modernize its legacy systems was identified. The OAuth 2.0 protocol was implemented in the Enterprise Service Bus (ESB) that will be used for modernization of legacy systems of UnB. Performance tests were carried out in the solution allowing to check the increase in the latency introduced by the Protocol and the average flow supported. Simulations were carried out with the objective to verify the behavior of the Protocol implemented when exposed to a replay attack.

Keywords: *Security, OAuth 2.0, Authorization, SOA, ESB, ErlangMS, Modernization of Legacy System.*

Sumário

1 Introdução **1**
1.1 Problema de Pesquisa . . . 2
1.2 Justificativa . . . 2
1.3 Objetivos . . . 3
1.3.1 Objetivo Geral . . . 3
1.3.2 Objetivos Específicos . . . 3
1.4 Resultados Esperados . . . 3
1.5 Estrutura do trabalho . . . 3

2 Mapeamento Sistemático **5**
2.1 Questões de Pesquisa . . . 5
2.2 Estratégia de Busca . . . 5
2.3 Critérios de Inclusão/Exclusão e Procedimentos de Seleção . . . 6
2.4 Processo de Extração dos Dados . . . 6
2.5 Análise inicial . . . 9
2.6 Resultados . . . 11
2.6.1 QP1) Quais os principais protocolos utilizados para autenticação e autorização em SOA? . . . 11
2.6.2 QP2) Quais problemas de autenticação e autorização encontrados em sistemas que utilizam a arquitetura SOA? . . . 12
2.6.3 QP3) Quais os principais estilos arquiteturais abordados nos estudos? 13
2.6.4 QP4) Quais protocolos de autenticação e autorização são mais utilizados em REST? . . . 13
2.7 Discussão dos resultados . . . 14
2.8 Síntese do Capítulo . . . 15

3 Fundamentação Teórica **16**
3.1 Arquitetura Orientada a Serviço - SOA . . . 16
3.1.1 *Representational State Transfer* - REST . . . 17

3.1.2 *Simple Object Access Protocol* - SOAP . . . 17
3.1.3 Escolha do estilo de comunicação para a UnB . . . 18
3.1.4 Barramento Orientado a Serviço . . . 18
3.1.5 ErlangMS . . . 19
3.2 Segurança em SOA . . . 20
3.2.1 Criptografia . . . 20
3.2.2 Função de *Hash* . . . 22
3.2.3 *Message Authentication Code (MAC)* . . . 23
3.2.4 Assinatura Digital . . . 24
3.3 Soluções de Autenticação e Autorização . . . 24
3.3.1 OpenID Connect . . . 24
3.3.2 SAML . . . 25
3.3.3 XACML . . . 26
3.3.4 OAuth 1.0 . . . 26
3.3.5 OAuth 2.0 . . . 27
3.4 Trabalhos Relacionados . . . 33
3.5 Síntese do Capítulo . . . 34

4 Protocolo de Autenticação e Autorização 35
4.1 Requisitos . . . 35
4.2 Vulnerabilidades do OAuth 2.0 . . . 36
4.2.1 Ataques em rede . . . 37
4.2.2 *Cross-Site Request Forgery (CSRF)* . . . 37
4.3 OAuth 2.0 e ErlangMS . . . 38
4.3.1 Catálogo de serviços . . . 38
4.3.2 Arquitetura do protocolo . . . 40
4.4 Considerações de Segurança . . . 41
4.4.1 Ataque *Replay* com o Código de Autorização . . . 41
4.4.2 Ataque de representação com o Código de Autorização . . . 41
4.4.3 Captura do *Token* de Acesso . . . 41
4.5 Síntese do Capítulo . . . 45

5 Análise do Protocolo 46
5.1 Teste de Performance . . . 46
5.1.1 Objetivos . . . 46
5.1.2 Questões . . . 47
5.1.3 Métricas . . . 47
5.1.4 Ambiente de teste . . . 47

5.1.5 Resultados 50
5.2 Teste de Segurança 52
5.2.1 Ataque *Replay* com o Código de Autorização 52
5.2.2 Ataque Replay com o *MAC Token* 54
5.3 Discussão dos Resultados 55
5.4 Síntese do Capítulo 56

6 Conclusão **57**
6.1 Contribuições 58
6.2 Trabalhos Futuros 58

Referências **60**

Apêndice **63**

A Catálogo de serviço do OAuth 2.0 **64**

B Códigos do OAuth 2.0 no ErlangMS **70**

Lista de Figuras

2.1 Fonte de Pesquisa : Bibliotecas Digitais. 10
2.2 Distribuição dos artigos por ano da publicação. 10
2.3 Protocolos encontrados. 11
2.4 Estilos arquiteturais encontrados. 13
2.5 Soluções encontradas para REST. 14

3.1 Roteamento de mensagens [1]. 20
3.2 Arquitetura do ErlangMS [1]. 21
3.3 Processo de cifragem de um texto [2]. 22
3.4 Processo de assinatura digital [3]. 24
3.5 Fluxo abstrato do *OpenID Connect* [4]. 25
3.6 Fluxo Abstrato do OAuth 2.0 [5]. 28
3.7 Concessão por Código de Autorização [5]. 30
3.8 Concessão Implícita [5]. 31
3.9 Concessão por Credenciais do Proprietário do Recurso [5]. 32
3.10 Concessão por credencial do cliente [5]. 33

4.1 Exemplo de catálogo de serviço. 39
4.2 Arquitetura do OAuth 2.0 no ErlangMS 40
4.3 Solicitação de *token* de acesso do OAuth 2.0. 42
4.4 *Token* de acesso do OAuth 2.0. 42
4.5 Solicitação utilizando um *token* padrão do OAuth 2.0. 42
4.6 Solicitação do *token* de acesso no OAuth 1.0a. 43
4.7 *Token* de acesso no OAuth 1.0a. 43
4.8 Solicitação utilizando um MAC *token*. 44
4.9 Solicitação de MAC *token* de acesso no ErlangMS OAuth 2.0. 44
4.10 *Token* de acesso do ErlangMS OAuth 2.0. 44
4.11 Solicitação utilizando um MAC *token*. 45

5.1 Cenário de teste com o uso do OAuth 1.0. 48

5.2 Cenário de teste com o uso do OAuth 2.0. 49
5.3 Ambiente de teste. 50
5.4 Latência. 51
5.5 Vazão. 52
5.6 Cenário de ataque de repetição no código de autorização. 53
5.7 Cenário de ataque de repetição com o MAC *token*. 55

Lista de Tabelas

2.1 Resultados do processo de seleção dos artigos. 7
2.2 Estudos selecionados. 8

4.1 Comparação entre as soluções encontradas. 36

5.1 *Infraestrutura de Hardware* utilizada nos testes. 50
5.2 Latência nos 04 cenários. 51
5.3 Vazão (req/s). 52
5.4 Ataque *replay* utilizando o código de autorização. 54
5.5 Ataque *replay* utilizando o código de autorização. 54

Lista de Abreviaturas e Siglas

ABE Attribute-based encryption.

ACM Association for Computing Machinery.

API Application Programing Interface.

CA Certification Authority.

CAPES Coordenação de Aperfeiçoamento de Pessoal de Nível Superior.

CPD Centro de Informática.

CSRF Cross-Site Request Forgery.

DBLP Digital Bibliography and Library Project.

DNS Domain Name System.

ESB Enterprise Service Bus.

GQM Goals, Questions and Metrics.

HMAC Hash-based Message Authentication Code.

HTTP Hyper Text Transfer Protocol.

HTTPS Hyper Text Transfer Protocol Secure.

ICPEdu Infraestrutura de Chaves Públicas para Ensino e Pesquisa.

IEEE Institute of Electrical and Electronics Engineers.

IsP Internet Service Provider.

JSON JavaScript Object Notation.

LDAP Lightweight Directory Access Protocol.

MAC Message Authentication Code.

MD5 Message Digest Algorithm 5.

OASIS Advancement of Structured Information Standards.

OAuth Open Authorization Protocol.

ODBC Open DataBase Connectivity.

REST Representational State Transfer.

RFC Request for Comments.

SAML Security Assertion Markup Language.

SCA Sistema de Controle de Acesso.

SHA1 Secure Hash Algorithm 1.

SOA Service-Oriented Architecture.

SOAP Simple Object Access Protocol.

SP Service Provider.

TLS Transport Layer Security.

UACM Unified Access Control Model.

UnB Universidade de Brasília.

URI Uniform Resource Identifier.

WSDL Web Services Description Language.

XACML eXtensible Access Control Markup Language.

XML eXtensible Markup Language.

Capítulo 1

Introdução

A modernização de sistemas legados é um tema que vem sendo cada vez mais discutido na Universidade de Brasília (UnB). Um sistema legado é um *software* que permanece ativo, mas implementado com critérios e tecnologia ultrapassados. Com o processo de modernização espera-se reduzir os custos com a manutenção dos sistemas legados e aumentar a integração dos fluxos de negócios entre os sistemas [1, 6]. O processo de modernização dos sistemas legados apresenta alguns desafios, dentre eles a integração entre os sistemas que estão sendo modernizados.

A necessidade de modernização dos sistemas legados da UnB iniciou as discussões para a adoção de uma solução baseada em uma Arquitetura Orientada a Serviços - Service-Oriented Architecture (SOA).

Em uma Arquitetura SOA, uma organização pode criar serviços e compartilhá-los de modo a interagir em tempo de execução com outros serviços da Instituição ou até de outras Organizações [7]. SOA tem sido utilizada pelas organizações para apoiar a modernização dos sistemas legados, oferecendo integração entre os sistemas legados e os novos sistemas [8].

Foi proposto um *Enterprise Service Bus (ESB)* em [1] para a modernização dos sistemas legados na UnB. A solução foi denominada Erlangms e utiliza a arquitetura *Representational State Transfer (REST)* e foi desenvolvida na linguagem funcional Erlang.

No entanto, essa integração trazida pela arquitetura SOA pode trazer problemas de segurança, como por exemplo, o acesso às informações por agentes não autorizados. Dada a criticidade desses sistemas e informações compartilhadas, preocupações relacionadas a segurança devem ser tratadas quando da implementação de serviços baseados em SOA.

Este trabalho apresenta a implementação do protocolo *Open Authorization Protocol (OAuth)* como um serviço do barramento ErlangMS proposto em [1], com o objetivo de garantir o requisito de autorização no processo de modernização dos sistemas legados da UnB. Para isso foi realizado um mapeamento sistemático para identificar as principais

soluções de autenticação e autorização em SOA e selecionar a mais aderente aos requisitos necessários para o ErlangMS.

O protocolo implementado foi testado e avaliado, com o objetivo de verificar se houve um aumento no tempo de processamento das requisições geradas pelo processo de autorização. Foi realizada ainda uma verificação para avaliar a existência de vulnerabilidades conhecidas do OAuth 2.0 na implementação realizada neste trabalho.

1.1 Problema de Pesquisa

Com a modernização de sistemas utilizando SOA, torna-se importante controlar o acesso e a autorização das chamadas de serviços. O desafio encontrado neste estudo é disponibilizar o acesso aos serviços via REST de maneira segura.

O ErlangMS já trata de alguns requisitos de segurança como: integridade e privacidade através da utilização do protocolo *Transport Layer Security (TLS)* implementado em [1]. Quanto a autenticação a UnB possui algumas soluções como o *Lightweight Directory Access Protocol (LDAP)* e o Sistema de Controle de Acesso (SCA). Sendo assim, é importante que o protocolo implementado neste trabalho utilize as bases de usuário já existentes na UnB, porém de maneira autônoma da tecnologia de autenticação, pois o *Enterprise Service Bus* (ESB) deve ser utilizado por qualquer instituição independentemente da solução de autenticação utilizada.

Em um contexto universitário, dados valiosos são utilizados, como resultados de pesquisas, dados de alunos, professores e funcionários, dados administrativos da Universidade, entre outros. Com isso é importante garantir que essas informações sejam acessadas apenas às entidades autorizadas. A publicação indevida de informações confidenciais ou o acesso por pessoas não autorizadas devem ser tratados pela solução de segurança proposta para estas arquiteturas. Com isso o principal problema tratado neste estudo é tratar da autorização aos serviços disponibilizados no barramento Erlangms com o menor impacto de tempo de resposta possível às requisições via REST.

1.2 Justificativa

A necessidade de se implementar uma abordagem para a modernização dos sistemas legados no Centro de Informática (CPD) da UnB e a necessidade pelo uso dos dados da Universidade em outros projetos, impulsionaram a proposição de uma arquitetura, conforme apresentado em [1].

É necessário, no entanto, garantir a segurança dos dados utilizados nessas transações, dada a criticidade das informações envolvidas. A arquitetura proposta carece de

mecanismos de segurança que possam garantir a autorização para acessar os serviços disponibilizados via REST.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo deste trabalho é identificar as soluções de autenticação e autorização para SOA utilizadas na literatura e implementar a solução de autorização mais adequada à arquitetura proposta para a modernização dos sistemas legados no Centro de Informática (CPD) da Universidade de Brasília, verificando a viabilidade de sua utilização.

1.3.2 Objetivos Específicos

Para atingir o objetivo principal, os seguintes objetivos específicos são definidos:

- Realizar um mapeamento sistemático para identificar os protocolos de autenticação e autorização utilizados em SOA e *Representational State Transfer (REST)*;
- Avaliar os protocolos existentes e verificar se atendem as exigências de segurança para os mecanismos de autenticação e autorização do CPD.
- Avaliar a necessidade de customizações dos protocolos identificados;

1.4 Resultados Esperados

O resultado esperado com o desenvolvimento deste trabalho é a implementação de uma solução de autorização em SOA que apoie a modernização dos sistemas legados da Universidade de Brasília de forma a garantir a segurança dos recursos envolvidos.

1.5 Estrutura do trabalho

Este trabalho está organizado em cinco capítulos além deste. O Capítulo 2 apresenta o protocolo de mapeamento sistemático utilizado para identificar as principais soluções de autenticação e autorização em SOA e os resultados obtidos. No Capítulo 3 são apresentados os principais conceitos utilizados e os trabalhos correlatos. A solução de autorização proposta é apresentada no Capítulo 4, bem como os requisitos para a sua implementação. No Capítulo 5 são apresentados os testes executados. No Capítulo 6 são apresentadas as

conclusões, quais as impressões obtidas no decorrer do estudo e os objetivos a serem traçados em trabalhos futuros. O trabalho também é composto pelo Apêndice A - Catálogo de serviço do OAuth 2.0 e pelo Apêndice B - Códigos do OAuth 2.0 no ErlangMS.

Capítulo 2

Mapeamento Sistemático

Um mapeamento sistemático consiste em uma pesquisa da literatura para determinar quais estudos abordam uma questão definida, onde foram publicados, em que bibliotecas tenham sido indexados, que tipo de resultados estes estudos tenham retornado [9]. O objetivo deste mapeamento sistemático e responder as questões de pesquisa que foram definidas para de identificar os principais desafios e soluções sobre autenticação e autorização em SOA.

2.1 Questões de Pesquisa

Com o objetivo de encontrar respostas para a solução de segurança a ser adotada pelo CPD para a modernização de seus sistemas legados, foram elaboradas quatro questões de pesquisa **(QP)** que serão respondidas no desenvolvimento do mapeamento sistemático.

As questões de pesquisa definidas para este trabalho foram:

QP1) Quais são os principais protocolos utilizados para autenticação e autorização em SOA?

QP2) Quais são problemas de autenticação e autorização encontrados em sistemas que utilizam a arquitetura SOA?

QP3) Quais são os principais estilos arquiteturais abordados nos estudos?

QP4) Quais são protocolos de autenticação e autorização são mais utilizados em REST?

2.2 Estratégia de Busca

Para responder as questões de pesquisas foram realizadas buscas em algumas bibliotecas digitais. A construção da *string* de busca seguiu a estratégia definida por [10], que consiste

em identificar palavras-chaves a partir das questões de pesquisa e utiliza os conectores *AND* para combinar as palavras-chaves e *OR* para combinar os termos alternativos ou sinônimos.

Como resultado da estratégia de busca adotada foi obtida a seguinte *string* de busca: *((Authentication AND authorization) OR "Access Control"OR Security) AND (SOA OR ESB OR REST)*

As bibliotecas digitais utilizadas para realizar as buscas foram:

- IEEE xplore – ieeexplore.ieee.org/;
- *Association for Computing Machinery (ACM)* – http://www.acm.org/;
- Springer – https://link.springer.com/;
- ScienceDirect – www.sciencedirect.com/.

2.3 Critérios de Inclusão/Exclusão e Procedimentos de Seleção

Os critérios de inclusão e exclusão foram definidos para selecionar os trabalhos mais relevantes em relação as questões investigadas, para isso foram analisados inicialmente o título, o resumo, a introdução e a conclusão dos estudos.

Os critérios de inclusão adotados **(CI)** foram:

CI1) Estudos publicados no período compreendido do ano de 2007 até 2017 e que satisfizessem a *string* de busca.

CI2) Estudos que estão disponíveis nos periódicos da CAPES em inglês ou português.

Os critérios de exclusão **(CE)** adotados foram:

CE1) Estudos que não tratam de autenticação ou autorização em SOA ou tratam de maneira superficial;

CE2) Estudos incompletos, como apenas resumos e resumos expandidos.

CE3) Estudos publicados como *Short Paper*.

2.4 Processo de Extração dos Dados

Na etapa de busca descrita na Seção 2.2 foram obtidos uma grande quantidade de artigos. Para refinar a busca foram excluídos inicialmente, estudos que claramente não teriam

contribuição para o trabalho, por exemplo, estudos de outras áreas do conhecimento (medicina, matemática), estes estudos puderam ser facilmente eliminados através dos filtros presentes nas páginas das bibliotecas digitais. Nesta etapa, foram analisados ainda os títulos, os resumos e as palavras-chave.

Na segunda etapa foram aplicados os critérios de inclusão e exclusão definidos na seção 2.3. A Tabela 2.1 apresenta um resumo das etapas do processo de extração de dados.

Tabela 2.1: Resultados do processo de seleção dos artigos.

		Etapa 1	**Etapa 2**			
Fontes	**Estudos retornados**	**Estudos Relevantes**	**Excluídos**			**Incluídos**
			QE1	**QE2**	**QE3**	
IEEE	45	38	11	2	3	**22**
ACM	89	24	8	0	0	**16**
ScienceDirect	2238	10	5	0	0	**5**
Springer Link	1893	11	7	0	2	**2**
Total	4265	82	31	2	5	**45**

Obteve-se com o processo de exclusão, 45 estudos (conforme apresentado na Tabela 2.2) que foram classificados de acordo com os campos:

1. Veículo de publicação: fontes de pesquisa e conferências ou jornais da publicação;
2. Biblioteca digital;
3. Ano da publicação;
4. Solução:
   (a) Oauth;
   (b) OpenID;
   (c) SAML;
   (d) XACML;
   (e) Solução Própria.
5. Estilo arquitetural.

Tabela 2.2: Estudos selecionados.

E1,Tassanaviboon, Anuchart, and Guang Gong. Oauth and abe based authorization in semi-trusted cloud computing: aauth.	E12,Alipour, Hadiseh Seyyed, Mehdi Sabbari, and Eslam Nazemi. A policy based access control model for web services.	E23,Nordbotten, Nils Agne. XML and web services security standards.	E34,Yao, Danfeng, et al. Decentralized authorization and data security in web content delivery.
E2,Torroglosa-Garc¡a, Elena, et al. Integration of the OAuth and Web Service family security standards.	E13,Alam, Masoom, et al. Behavioral Attestation for Web Services using access policies.	E24,Durbeck, Stefan, et al. A Semantic Security Architecture for Web Services The Access-eGov Solution.	E35,Han, Song, et al. Secure web services using two-way authentication and three-party key establishment for service delivery.
E3,Sliman, Layth, et al. Single sign-on integration in a distributed enterprise service bus.	E14,Al-kofahi, Majd, Su Chang, and Thomas E. Daniels. SCWIM an Integrity Model for SOA Networks.	E25,Lopez, Gabriel, et al. A network access control approach based on the AAA architecture and authorization attributes.	E36,Thomas, Ivonne, and Christoph Meinel. An identity provider to manage reliable digital identities for SOA and the web.
E4,do Prado Filho, et. al. MultiAuth-WoT: a Multimodal Service for Web of Things Athentication and Identification.	E15,Qi-rui, Peng, et al. An authentication and authorization solution supporting SOA-based distributed systems.	E26,Shang, Chaowang, et al. SAML Based Unified Access Control Model for Inter-platform Educational Resources.	E37,Inoue, Takeru, et al. Key roles of session state: Not against rest architectural style.
E5,Memeti, Agon, et al. A framework for flexible REST services: Decoupling authorization for reduced service dependency.	E16,Nagarajan, Aarthi, Vijay Varadharajan, and Nathan Tarr. Trust enhanced distributed authorisation for web services.	E27,Wolf, Martin, et al. A message meta model for federated authentication in service-oriented architectures.	E38,Alam, Masoom, et al. xDAuth: a scalable and lightweight framework for cross domain access control and delegation.
E6,Cirani, Simone, et al. Iot-oas: An oauth-based authorization service architecture for secure services in iot scenarios.	E17,Boehm, Oliver, et al. Federated Authentication and Authorization: A Case Study.	E28,Li, Jun, and Alan H. Karp. Access control for the services oriented architecture.	E39,Hüffmeyer, Marc, and Ulf Schreier. RestACL: An Access Control Language for RESTful Services.
E7,Field, John P., Stephen G. Graham, and Tom Maguire. A framework for obligation fulfillment in REST services.	E18,Sabbari, Mehdi, and Hadiseh Seyyed Alipour. Improving attribute based access control model for web services.	E29,El Yamany, Hany F., and Miriam AM Capretz. An authorization model for Web Services within SOA.	E40,Wu, Meng-Yu, and Tsern-Huei Lee. Design and implementation of cloud API access control based on OAuth.

E8,Tokunaga, Kazuhiro, et al. IMS presence authorization applied to Web applications using REST.	E19,Jarmakiewicz, Jacek, et al. Design and implementation of multilevel security subsystem based on XACML and WEB services.	E30,Mazzoleni, Pietro, et al. XACML policy integration algorithms.	E41,Hsieh, George, and Moeti M. Masiane. Towards an Integrated Embedded Fine-Grained Information Protection Framework.
E9,Richardson, Leonard. Developers enjoy hypermedia, but may resist browser-based OAuth authorization.	E20,Noureddine, Moustafa, and Rabih Bashroush. An authentication model towards cloud federation in the enterprise.	E31,Turkmen, Fatih, and Bruno Crispo. Performance evaluation of XACML PDP implementations.	E42,Foping et.al. Design and Implementation of a Private RESTful API to Leverage the Power of an eCommerce Platform.
E10,Hollrigl, T., et al. Towards systematic engineering of Service-Oriented access control in federated environments.	E21,Noureddine, M., and R. Bashroush. A provisioning model towards OAuth 2.0 performance optimization.	E32,Xu, Jie, et al. Dynamic authentication for cross-realm SOA-based business processes.	E43,da Silva Maciel et al. An optimistic technique for transactions control using REST architectural style.
E11,Hummer, Waldemar, et al. An integrated approach for identity and access management in a SOA context.	E22,Gonz lez, et al. Reverse OAuth: A solution to achieve delegated authorizations in single sign-on e-learning systems.	E33,Thomas, Ivonne, et al. Using quantified trust levels to describe authentication requirements in federated identity management.	E44,Dalsgaard, Esben, Kåre Kjelstrøm, and Jan Riis. A federation of web services for Danish health care.
E45,Oktian, Yustus Eko, Sang-Gon Lee, and JunHuy Lam. OAuthkeeper: An Authorization Framework for Software Defined Network.			

2.5 Análise inicial

A biblioteca digital que mais retornou estudos foi o IEEE xplorer com 48,9%, seguido pela ACM (35,6%), ScienceDirect (11,1%) e Springer Link (4,4%), conforme apresenta a Figura 2.1.

Os estudos selecionados pertencem a 31 conferências ou revistas distintas, os veículos que mais retornaram artigos foram a SWS - Secure Web Services (3) e a WS-REST - Workshop on Web APIs and RESTful Design (2). O restante dos veículos tiveram uma publicação cada.

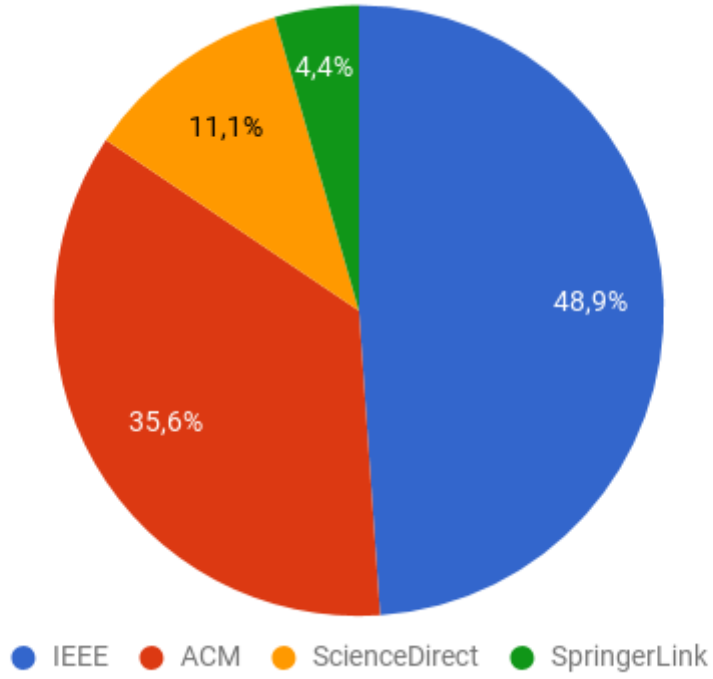

Figura 2.1: Fonte de Pesquisa : Bibliotecas Digitais.

A Figura 2.2 apresenta a distribuição dos artigos selecionados pelo ano de sua publicação. Os trabalhos estão distribuídos entre os anos de 2007 e 2017, lembrando que artigos anteriores a 2006 foram excluídos pelo critério de exclusão 1 definido na Seção 2.3.

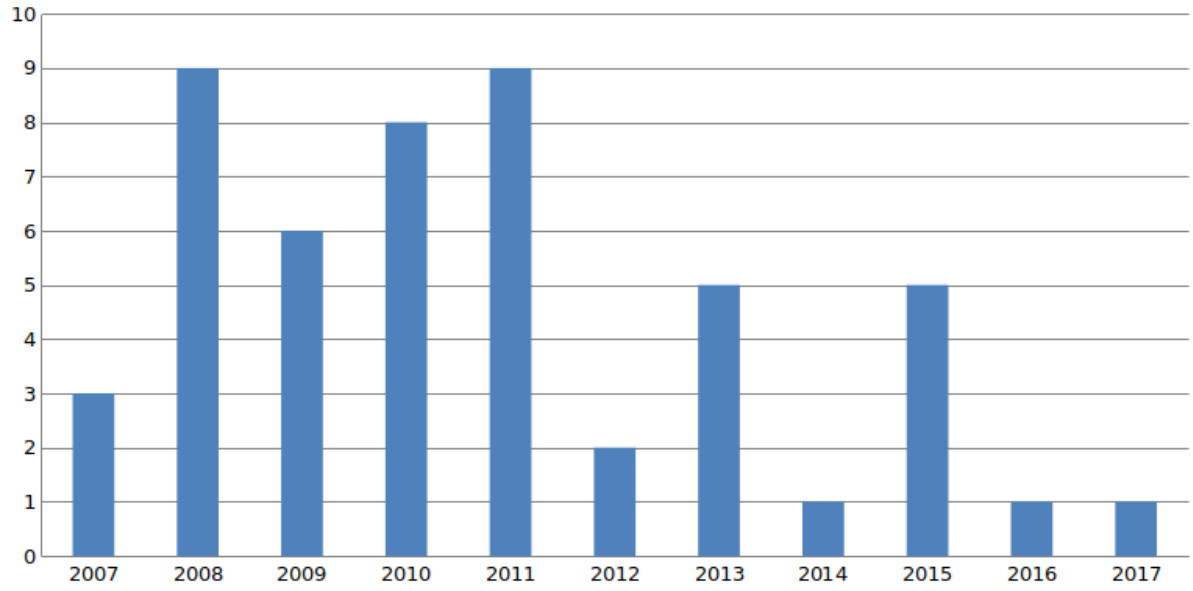

Figura 2.2: Distribuição dos artigos por ano da publicação.

2.6 Resultados

Nesta Seção é apresentado um resumo dos resultados encontrados através do mapeamento sistemático, organizado de acordo com as questões de pesquisa definidas na Seção 2.1.

2.6.1 QP1) Quais os principais protocolos utilizados para autenticação e autorização em SOA?

O objetivo desta questão é descobrir quais são as soluções de autenticação e autorização em SOA mais abordadas nos trabalhos atuais. A Figura 2.3 apresenta os protocolos abordados no mapeamento sistemático.

Foram abordados 6 tipos de soluções no mapeamento sistemático, o padrão XACML foi o mais abordado, sendo citado em 13 estudos (28,9%) assim como proposições de novas soluções . O padrão SAML e protocolo o OAuth foram citados em 12 estudos (26,7%), o conjunto de normas WS-* foi citado em 7 estudos - 15,6% e o protocolo OpenID em 3 estudos (6,7%).

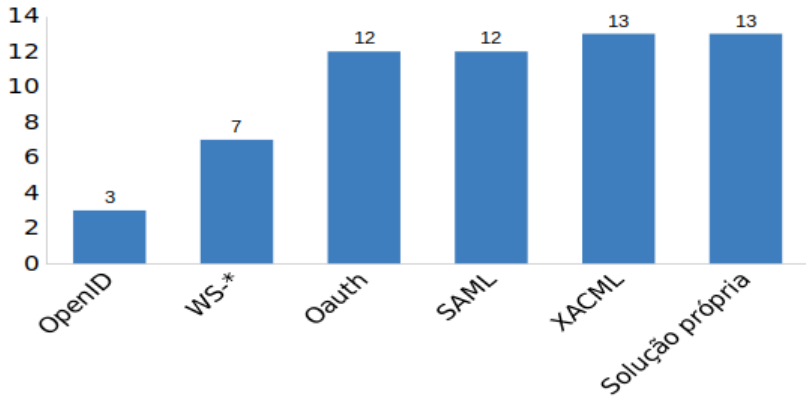

Figura 2.3: Protocolos encontrados.

O padrão XACML pode ser utilizado em conjunto com o SAML para transmitir informações de controle de acesso [11]. A principal vantagem dessa abordagem é a utilização dos controles de acesso mais ricos do XACML em cima do padrão SAML para troca de mensagens. Por esse motivo os padrões SAML e XACML são citados juntos em 5 dos trabalhos selecionados.

A WS-* é usado para referenciar um conjunto de normas de segurança para XML mantidos pela OASIS [12]. Os padrões WS-Security, WS-Trust, WS-SecureConversation e WS-Policy são exemplos de normas que fazem parte da família WS-*.

2.6.2 QP2) Quais problemas de autenticação e autorização encontrados em sistemas que utilizam a arquitetura SOA?

O objetivo desta questão é selecionar os principais desafios para autenticação e autorização em SOA.

Heterogeneidade de Soluções e integração de serviços

A heterogeneidade de soluções para autenticação em SOA pode prejudicar a interoperabilidade de serviços que utilizam a arquitetura.

O problema abordado pelo trabalho proposto por [13] consiste na heterogeneidade relacionada aos tipos distintos de concessão da autorização e de *tokens* de acesso relacionados às diferentes implementações do OAuth. Esta heterogeneidade pode ser combatida através da integração com o WS-Trust, que é especialmente destinado a oferecer mecanismos de integração entre os serviços que implementam especificações WS-*.

Os trabalhos [14, 15, 16, 17, 18] também abordam o problema da integração de serviços em domínios inter organizacionais. Estes estudos propõem soluções próprias ou customização de soluções existentes de federação de identidade para solucionar o problema. Já o trabalho [19] aborda uma solução seguindo o padrão SAML.

Ponto único de falha

Uma solução adotada para autorização em SOA é denominada de *Single Sign-On* (*SSO*). A idéia principal desse tipo de solução é centralizar o gerenciamento de usuários, liberando os usuários de lembrar uma quantidade grande de credenciais. No entanto, esta abordagem tem algumas limitações. Em primeiro lugar, centralizando os resultados da gestão de usuário em um ponto único de falha. Em alguns casos, nenhum dos sistemas é acessível quando a solução de *SSO* falhar. Além disso, uma solução *SSO* requer armazenamento central de informações do usuário. Isto só é possível, se todos os sistemas estão em um domínio localizado na mesma administração, por exemplo, sob o controle de uma organização [20].

Vulnerabilidades relacionadas a ataques DoS

O aumento no número de processos e de troca de mensagens introduzidos por um protocolo de autenticação pode potencializar o impacto de um ataque de negação de serviço (DOS).

Os trabalhos [21] e [22] apresentam uma customização do protocolo Oauth 2.0 com o objetivo de reduzir as chances de ataques *DoS (Denial-of-Service)* e *DDoS (Distributed Denial-of-Service)*.

Performance

Os trabalhos [13], [7] e [23] apresentaram avaliações sobre impacto da implementação de soluções próprias relativas a performance. Nestes trabalhos foram realizados testes empíricos para mensurar o impacto da implementação do novo protocolo. São comparados os tempos gastos em transações com e sem autenticação. Essas soluções se mostraram adequadas para o trabalho proposto pelos autores.

2.6.3 QP3) Quais os principais estilos arquiteturais abordados nos estudos?

O objetivo desta questão é mostrar quais padrões arquiteturais são os mais abordados nos estudos selecionados durante o mapeamento sistemático.

A Figura 2.3 apresenta os resultados encontrados. O padrão SOAP (*Simple Object Access Protocol*) foi o mais encontrado nos trabalhos(46,7%), seguido pelo REST(31,1%). Em 20% dos trabalhos encontrados, os autores trataram da arquitetura SOA de maneira geral e por fim, 2 trabalhos (4,4%) falavam de *Cloud Computing*.

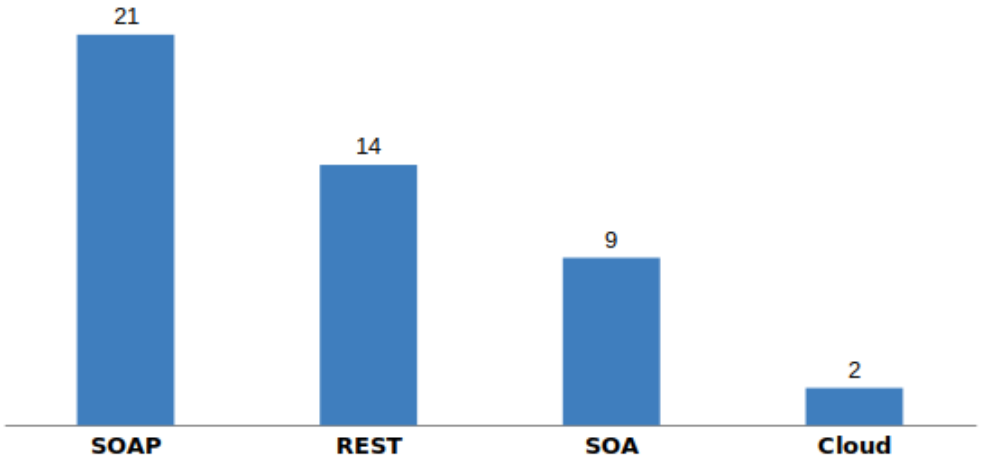

Figura 2.4: Estilos arquiteturais encontrados.

2.6.4 QP4) Quais protocolos de autenticação e autorização são mais utilizados em REST?

Esta questão visa identificar as soluções mais utilizadas para o padrão arquitetural REST, que é o padrão utilizado para o projeto de migração dos sistemas legados na UnB [1]. As

soluções encontradas na literatura foram as mesmas utilizadas para a arquitetura SOA conforme apresenta a Figura 2.5.

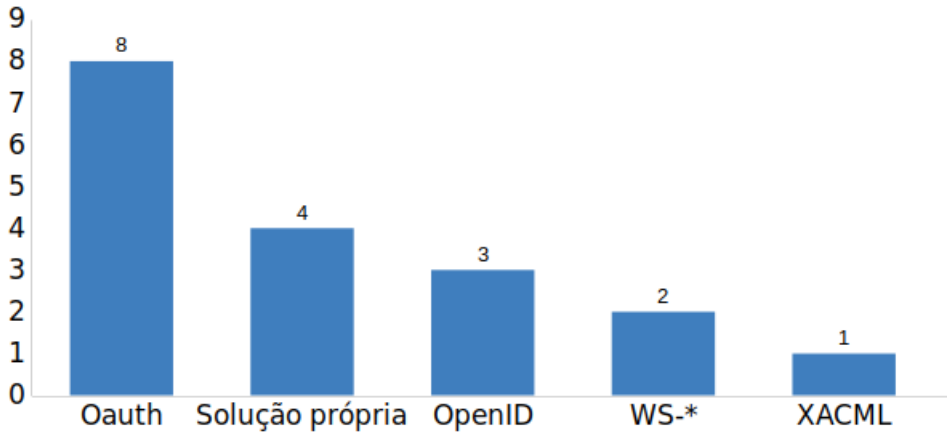

Figura 2.5: Soluções encontradas para REST.

A solução mais abordada para o estilo foi o Oauth (8 estudos), seguida por novas proposições (4 estudos), pelo protocolo OpenID Connect (3 estudos) e por fim o padrão XACML foi encontrado em um único estudo.

2.7 Discussão dos resultados

O problema mais abordado nos artigos selecionados está relacionado à variedade de soluções relacionada a transações inter organizacionais. A criação de soluções próprias foi a abordagem mais utilizada para a solução deste problema. Uma explicação para isso é a diversidade de soluções de autenticação em SOA implementada pelas organizações e a inexistência de um padrão global para autenticações inter organizacionais.

Os protocolos mais utilizados foram listados para responder a questão de pesquisa 1 (QP1). Com isso é possível ter uma visão atualizada sobre os padrões utilizados atualmente sobre autentização e autorização em SOA. Dentre os protocolos listados destacam-se o protocolo OAuth e o padrão SAML (podendo ser utilizado em conjunto com XACML). Ambos os protocolos podem ser implementados utilizando normas da família WS-*, conforme mostram os trabalhos[12] e [13], garantido uma maior interoperabilidade entre domínios de segurança.

Na questão de pesquisa QP3, foi observado que o estilo mais abordado na literatura foi o SOAP. O estilo arquitetural REST foi abordado em 14 trabalhos. Como este estilo

será o adotado no projeto de modernização dos sistemas legados da UnB [1] utilizamos os 14 trabalhos para responder a QP4.

Para o estilo arquitetural REST a solução mais adotada foi o protocolo Oauth, o que pode ser explicado pela utilização do próprio estilo REST para a troca de mensagens de autenticação e autorização, tornando o protocolo Oauth mais aderente ao padrão arquitetural. O SAML, por sua vez usa mensagens SOAP no processo de autenticação, não foi encontrado em nenhum estudo sobre REST.

2.8 Síntese do Capítulo

Este Capítulo apresentou o protocolo de mapeamento sistemático utilizado para identificar as soluções de autenticação e autorização em SOA mais aderentes ao ErlangMS. O protocolo OAuth foi identificado como o mais aderente aos padrões utilizados na UnB. Foi possível ainda identificar os principais desafios em autenticação e autorização em SOA.

Capítulo 3

Fundamentação Teórica

Este Capítulo apresenta uma revisão dos principais conceitos relacionados ao tema deste trabalho. A Seção 3.1 apresenta uma visão sobre arquiteturas orientadas a serviços. A Seção 3.2 apresenta alguns conceitos de segurança importantes para a execução do trabalho. Na Seção 3.3 são apresentadas soluções de autenticação e autorização em SOA. A Seção 3.4 apresenta os principais trabalhos relacionados a este estudo.

3.1 Arquitetura Orientada a Serviço - SOA

Uma Arquitetura Orientada a Serviço é um conjunto de componentes distribuídos que fornecem e consomem serviços. Os componentes em SOA podem usar plataformas e linguagens de implementação diferentes. Os serviços são em grande parte independentes, podendo pertencer a diferentes organizações [24].

Alguns princípios de uma arquitetura SOA são [25, 26]:

- **Baixo acoplamento**: Prestadores e consumidores de serviço são independentes e capazes de tomar decisões sobre a tecnologia adotada. Um consumidor de serviço não deve ser vinculado a uma determinada tecnologia (por exemplo, linguagem de programação, base de dados, plataforma), a fim de usar um serviço.
- **Reutilização**: Um serviço reutilizável apresenta funcionalidade auto contida que pode ser utilizada em múltiplos processos de negócio. Serviços agnósticos tem maior possibilidade de reutilização.
- **Modularidade**: O objetivo final de modularidade é prover a capacidade de mudar os componentes da arquitetura sem impactar os consumidores do serviço.
- **Capacidade de Descoberta**: Se refere a capacidade de um serviço ser encontrado e que suas interfaces sejam compreendidas pelo consumidor. Todo serviço possui

um contrato informando o que o serviço faz e como se comunica, quais entradas são esperadas e quais saídas podem ser geradas.

Essas características fazem com que a arquitetura SOA seja amplamente utilizada pelas organizações em projetos envolvendo a modernização dos sistemas legados [8].

Além de provedores e consumidores de serviço, uma arquitetura SOA pode ter componentes auxiliares como por exemplo um barramento orientado a serviço (ESB). Existem ainda os padrões utilizados para a comunicação entre serviços, dentre eles destacam-se o REST (*Representational State Transfer*) e o SOAP(*Simple Object Access Protocol*) [27].

3.1.1 *Representational State Transfer* - REST

O estilo arquitetural *Representational State Transfer* (REST) foi proposto por Roy Fielding em 2000 em sua tese de doutorado. Pode ser descrita como um conjunto de princípios arquiteturais que podem ser utilizados para o desenvolvimento de serviços web. Utilizam o protocolo *Hyper Text Transfer Protocol (HTTP)* para realizar as trocas de mensagens. Um recurso em REST é a informação que pode ser acessada por identificador único (URI - *Uniform Resource Identifier*) [28]. Serviços REST podem usar as notações JSON (*JavaScript Object Notation*) ou XML (*eXtensible Markup Language*).

Uma URI pode ser estruturada de maneira hierárquica ou através de uma estrutura <chave> = <valor>. Os exemplos a seguir obtém dados de um aluno com identificação 100 [28, 1]:

- http://unb.sistemas.br/alunos/100
- http://unb.sistemas.br/alunos?id=100

A troca de informações ocorre por meio do protocolo HTTP, com uma semântica específica para cada operação:

- **POST**: Cria um novo recurso.
- **GET**: Utilizado para recuperar um informação.
- **PUT**: Atualiza os dados de um recurso.
- **DELETE**: Apaga os dados de determinado recurso.

3.1.2 *Simple Object Access Protocol* - SOAP

Outra forma de se implementar uma comunicação em SOA é através de *Web Services* - SOAP. O SOAP é o protocolo de transporte responsável pela troca de mensagens, o qual

segue as normas da W3C, sendo baseado em XML, o que aumenta a compatibilidade com outras plataformas e linguagens que tenham suporte para a manipulação para este tipo de arquivo [29].

Nesta tecnologia, as interfaces de serviço são definidas na linguagem WSDL (*Web Services Description Language*) [27]. O WSDL é um documento em formato XML que descreve o serviço oferecido, como acessá-lo, e quais as operações e os métodos disponíveis [30].

3.1.3 Escolha do estilo de comunicação para a UnB

O estudo [1] foi definido que o estilo arquitetural REST será utilizado no processo de modernização dos sistemas legados da UnB. A decisão é apoiada pelas seguintes vantagens do REST:

- Normalmente serviços SOAP apresentam um maior *overhead*, devido ao tamanho das mensagens no estilo, já que as mensagens são descritas na linguagem WSDL e envelopadas pelo protocolo SOAP, em formato XML [30] .
- REST permite o uso de JSON em vez do formato XML para a especificação das mensagens [24]. Uma das suas principais vantagens de se usar JSON consiste na facilidade no desenvolvimento, no aproveitamento da infraestrutura web existente e um esforço de aprendizado menor. O que é mais condizente com a modernização de sistemas legados, evitando que as aplicações tenham que lidar com vários protocolos, caso fosse escolhido o estilo SOAP [31, 1].

3.1.4 Barramento Orientado a Serviço

A invocação de um serviço pode ser mediada por um barramento orientado a serviço. Um ESB roteia as mensagens entre os consumidores e prestadores de serviços. Além disso, um ESB pode converter as mensagens de um protocolo para outro, executar transformações de dados (por exemplo, formato, conteúdo), executar verificações de segurança e gerenciar transações [24]. A seguir são listadas algumas vantagens e desafios de um ESB[27]:

Vantagens

- **Manutenibilidade:** mudanças no contrato do serviço podem ser tratadas no ESB;
- **Segurança:** controle de acesso e outras políticas de segurança podem ser tratadas no ESB;

- **Interoperabilidade:** conversão de protocolo e transformação de mensagens permitem integrar componentes que não são interoperáveis;
- **Confiabilidade:** mensagens podem ser reenviadas em caso de falha.

Desafios

- **Performance:** O desempenho é afetado pelo tempo adicional associado a transformações de mensagens e protocolos e roteamento de mensagens;
- **Segurança:**O ESB é mais um componente a proteger e se mal configurado pode se tornar um ponto único de falha;
- **Manutenibilidade/portabilidade:** A lógica de transformação e roteamento é codificada no ESB e pode não ser portável para outros produtos;
- **Alto Custo:** Custo alto de aprendizado.

3.1.5 ErlangMS

O trabalho iniciado por Aguilar em [1] propõe o uso da arquitetura SOA utilizando um barramento aderente ao estilo (REST) na modernização de sistemas legados da Universidade de Brasília. A solução denominada ErlangMS é um barramento orientado a serviço (*Enterprise Service Bus*, ESB) sendo desenvolvido na liguagem Erlang, utiliza o formato JSON para troca de mensagens com o cliente.

O barramento de serviços é baseado no conceito de catálogo de serviços, permitindo a reusabilidade dos componentes de software, uma vez que, estando o serviço publicado no barramento de serviços, ele poderá ser acessado por outras aplicações. O estudo apresentou o mapeamento sistemático realizado e o estudo de caso já iniciado.

O esquema de comunicação do barramento ocorre em duas vias. Na primeira via, existe a comunicação do cliente via REST (independente da sua linguagem de programação ou plataforma) para consumir algum serviço no barramento. Na segunda via ocorre a comunicação do barramento com o serviço, que está implementado em alguma linguagem de programação. Essa comunicação dá-se via sistema de mensageria disponível em Erlang que possibilita uma comunicação assíncrona com várias linguagens de programação de forma muito rápida por trafegar os dados no formato binário e com baixa latência na rede. Assim, a função do barramento é rotear as requisições REST para algum serviço processar e devolver o resultado de volta para o cliente, conforme apresentado na Figura 3.1.

A Figura 3.2 apresenta a arquitetura do ErlangMS. O processo inicia-se com uma requisição REST ao barramento. O módulo *server* HTTP/REST é o responsável pelo

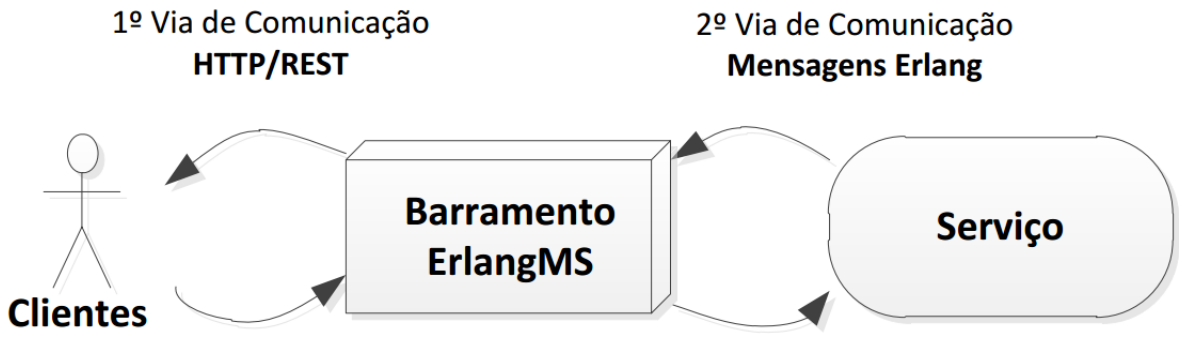

Figura 3.1: Roteamento de mensagens [1].

recebimento e envio das requisições junto ao cliente. O Processo *Dispatcher* que é o responsável por consultar o catálogo de serviços (módulo Catalogo) e rotear a requisição ao serviço correspondente.

O módulo *Auth* será o responsável pela autenticação, verificar a identidade do cliente antes do acesso ao serviço, e autorização, verificar se o cliente tem os privilégios necessários para acessar o serviço solicitado.

3.2 Segurança em SOA

Esta Seção apresenta alguns conceitos de segurança importantes para SOA:

- **Autenticação:** o processo que verifica a identidade de um agente externo antes de iniciar uma iteração. Consiste em provar, por exemplo, que o usuário é na verdade quem afirma ser [32, 33].
- **Autorização:** refere-se a verificação dos privilégios solicitados por agentes externos autenticados. Visa estabelecer se um determinado usuário tem a permissão para prosseguir com a tarefa que está solicitando.
- **Privacidade:** é a capacidade em que os dados restritos são mantidos em sigilo de pessoas e serviços não autorizados;
- **Integridade:** a capacidade de ter confiança de que as mensagens não são modificadas intencionalmente.

3.2.1 Criptografia

Um sistema criptográfico define duas transformações: cifragem e decifragem. A cifragem é aplicada sobre os dados no formato original (em claro) para transformá-los em dados cifrados e a decifragem é aplicada sobre os dados cifrados para torná-los em dados em

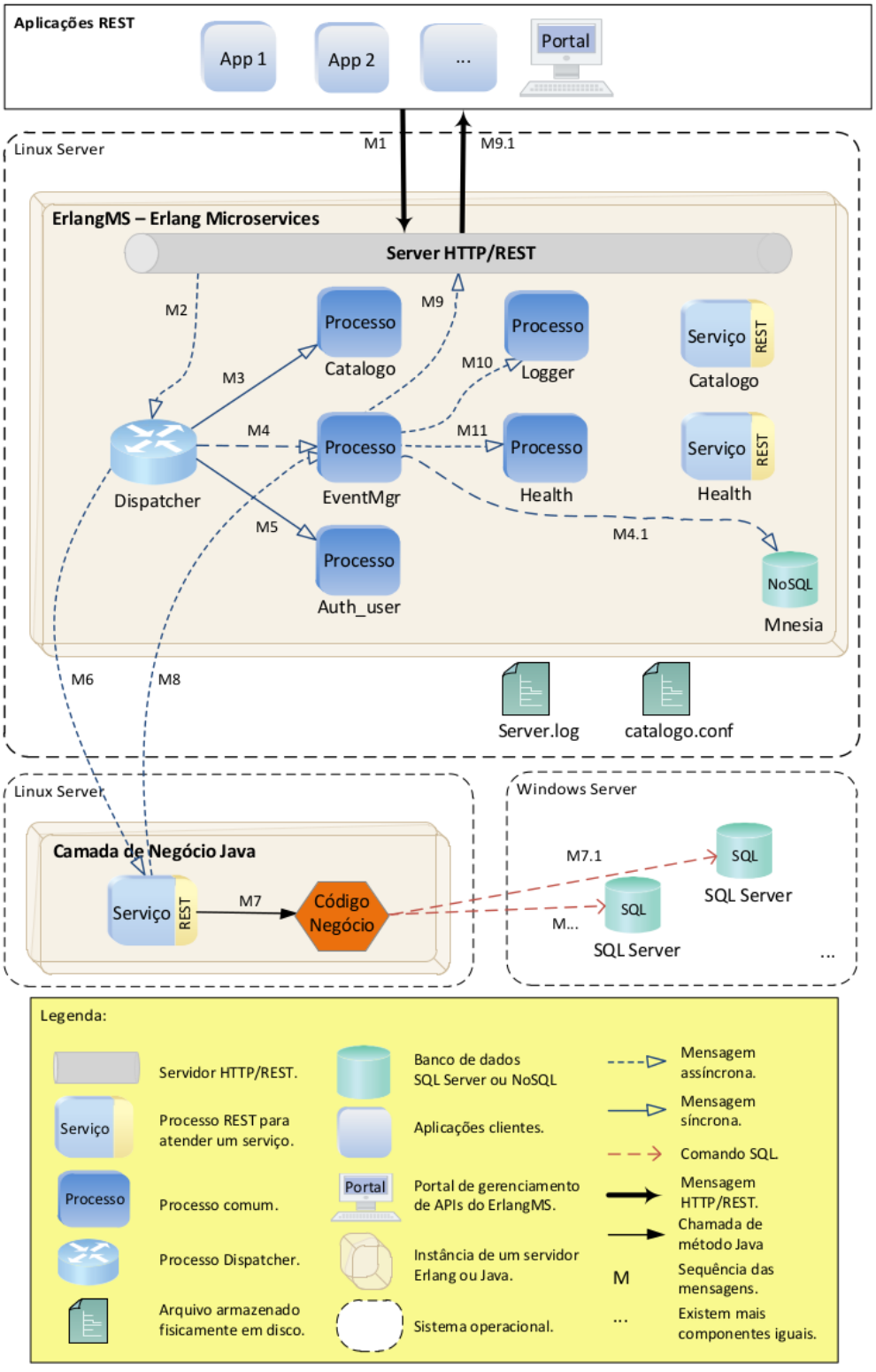

Figura 3.2: Arquitetura do ErlangMS [1].

claro novamente. Além dos dados, um valor denominado chave é usado em ambas as transformações [2].

A Figura 3.3 apresenta o processo de cifragem e decifragem de um texto em claro.

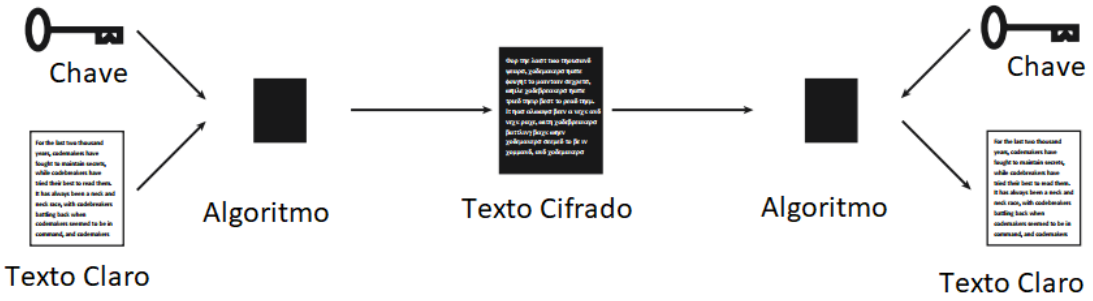

Figura 3.3: Processo de cifragem de um texto [2].

Existem dois tipos de sistemas criptográficos classificados de acordo com a chave utilizada: o de chave simétrica e o de chave assimétrica [2].

Sistemas Criptográficos de Chave Simétrica

Um sistema criptográfico de chave simétrica ou secreta é aquele em que a mesma chave é usada na cifragem e na decifragem dos dados.

Sistemas Criptográficos de Chave Assimétrica

Um sistema criptográfico de chave assimétrica ou pública faz uso de um par de chaves complementares, denominadas chave privada e chave pública. Assim, se a chave privada for utilizada para a cifragem, só a chave pública do par poderá ser utilizada para a decifragem. Assim como se a chave pública for utilizada para a cifragem, só a chave privada do par poderá ser utilizada para a decifragem.

Neste sistema, todas as entidades têm acesso a chave pública porém a chave privada deve ser de conhecimento apenas do seu proprietário.

3.2.2 Função de *Hash*

Uma função de *hash* tem como entrada uma mensagem de tamanho variável como entrada e produz um valor de *hash* de tamanho fixo. O tipo de função de *hash* utilizada para aplicações de segurança é conhecido como função de *hash* criptográfica. [34]. As propriedades desejáveis dessa função são:

1. **Não-inversão:** dado um *hash* é computacionamente inviável recuperar a mensagem de entrada;
2. **Resistência à colisão:** baixa probabilidade de que dois objetos sejam mapeados para o mesmo *hash*.

Alguns algoritmos de função de *hash* comuns são o *Secure Hash Algorithm 1 (SHA1)* e o *Message Digest Algorithm 5 (MD5)*.

A principal aplicação de segurança de uma função de *hash* está relacionada a integridade de dados. Uma mudança em qualquer bit em uma mensagem de entrada deve gerar uma mudança no *hash* gerado [34]. Funções de hash são utilizadas para autenticação de mensagem, através de código de autenticação de mensagem (MAC) e para assinaturas digitais.

3.2.3 *Message Authentication Code (MAC)*

A autenticação de mensagem pode ser alcançada usando códigos MAC. Códigos MACs são usados entre duas partes que compartilham uma chave secreta para autenticar as mensagens trocadas entre elas.

Um MAC é construído aplicando uma função a uma mensagem e a chave simétrica compartilhada. A mensagem pode então ser autenticada pelo receptor gerando um novo MAC da mensagem recebida e da chave compartilhada e comparando com o valor do MAC recebido.

A autenticação da mensagem utilizando códigos permite garantir a integridade da mensagem e a identidade do emissor mesmo em um canal inseguro. Um atacante pode alterar a mensagem mas não conseguirá alterar o valor MAC associado sem conhecer a chave secreta [34].

Dentre as funções de MAC conhecidas destaca-se o HMAC, utilizadas para gerar MACs baseados em funções de *hash*.

Hash-based Message Authentication Code (HMAC)

O HMAC é um algoritmo para geração de código MAC baseado em funções de *hash* criptográficas, é especifica no RFC 2104 [35], podem ser utilizadas função de *hash* conhecidas como o MD5 e o SHA1. O uso de funções de *hash* traz algumas vantagens:

- Funções de *hash* são mais rápidas do que cifras de bloco para o desenvolvimento *software*;
- Grande quantidade de códigos de bibliotecas para funções de *hash*;
- Fácil substituição de uma função de *hash* por outra.

3.2.4 Assinatura Digital

A forma mais comum de implementação de assinatura digital é a geração de um sumário que é cifrado com a chave privada do remetente, gerando um apêndice que é enviado anexado a mensagem assinada [3].

A assinatura é gerada aplicando uma função de *hash* à mensagem que será enviada e à chave privada do remetente.

O funcionamento é semelhante ao do código MAC. Porém o uso de chave privada garante que a assinatura não possa ser alterada nem mesmo pelo receptor. A Figura 3.4 apresenta o funcionamento do processo de assinatura digital.

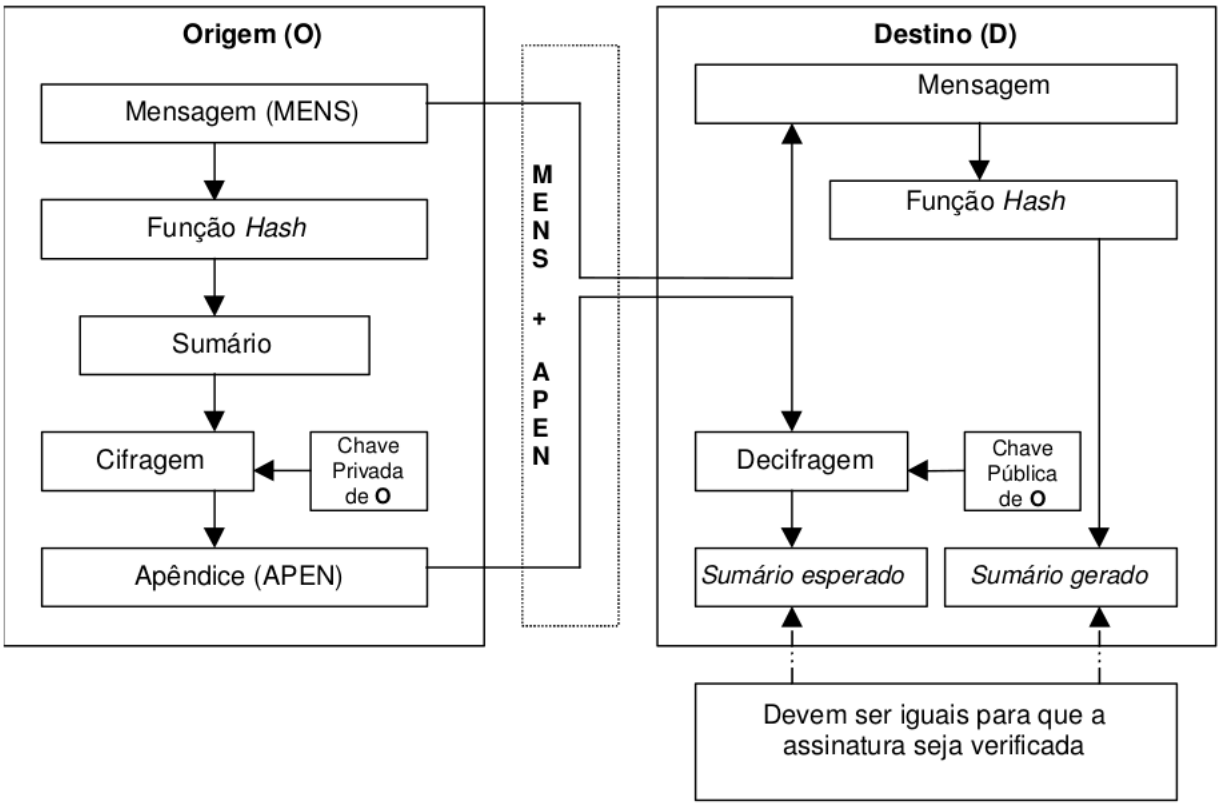

Figura 3.4: Processo de assinatura digital [3].

3.3 Soluções de Autenticação e Autorização

3.3.1 OpenID Connect

O *OpenID Connect* é um protocolo de autenticação *open source* lançado em 2012, que funciona sobre o OAuth 2.0 [36] implementando autenticação como uma extensão para o

processo de autorização do OAuth 2.0. O protocolo utiliza o padrão REST e mensagens JSON no processo de autenticação [4].

A Figura 3.5 apresenta o processo de autenticação utilizando o *OpenID Connect*.

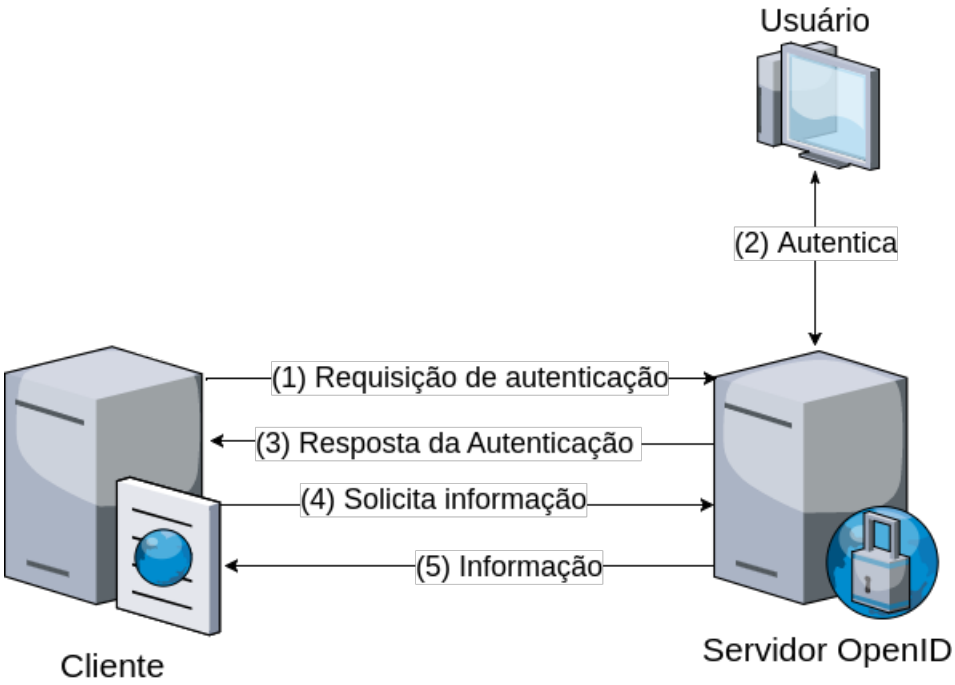

Figura 3.5: Fluxo abstrato do *OpenID Connect* [4].

Os passos para autenticação são:

1. O Cliente envia uma solicitação para o Servidor OpenID.
2. O Provedor OpenID autentica o usuário final e obtém autorização.
3. O Provedor OpenID responde com um *token* de ID e geralmente um *token* de acesso.
4. A Provedor do Serviço pode enviar uma solicitação para acessar a informação do usuário com o *token* de acesso para o servidor OpenID.
5. O Provedor OpenID retorna as informações sobre o usuário final.

3.3.2 SAML

Security Assertion Markup Language (SAML) é um padrão aberto desenvolvido pela OASIS - *Advancement of Structured Information Standards* . Ele faz uso da linguagem XML para trocar dados de autenticação e autorização entre domínios de segurança, geralmente entre um provedor de identidade(IsP) e um prestador de serviços (SP) [13, 37].

O SAML tem quatro componentes: *Assertions*, *Protocols*, *bindings* e *profiles* [37].

Assertions

É um pacote de informações que fornece uma ou mais declarações feitas por uma autoridade SAML [37].

Protocols

São elementos de pedidos/respostas que empacotam as *assertions*. O principal protocolo do SAML é o *Authentication Request Protocol*. No SAML 2.0, o fluxo começa no prestador de serviços que emite uma solicitação de autenticação explícita ao provedor de identidade. O provedor de identidade autentica o usuário e emite uma resposta de autenticação, que é transmitida de volta para o provedor de serviços (através do *browser*) [37].

Bindings

Definem como as mensagens do protocolo SAML são usadas dentro de protocolos de transporte. Para um *browser*, o HTTP *Redirect* e o HTTP POST são comumente usados. O prestador de serviços pode usar "HTTP Redirect" para enviar um pedido, enquanto o provedor usa "HTTP POST"para transmitir a resposta. As solicitações podem ainda ser enviadas diretamente na URL de uma solicitação via HTTP GET [37].

Profiles

Profiles determinam como *assertions*, *protocols* e *bindings* são combinados. Definem restrições e/ou extensões para apoiar o uso de SAML para uma aplicação particular. Por exemplo, o perfil de *SSO Web Browser* especifica como são as *assertions* entre um provedor de identidade e o provedor de serviço para habilitar o *logon* único para um usuário do navegador.

3.3.3 XACML

XACML (*eXtensible Access Control Markup Language*) é um padrão OASIS que compreende basicamente em uma linguagem baseada em XML para políticas de controle de acesso e um modelo para avaliar solicitações de autorização [38].

3.3.4 OAuth 1.0

O OAuth (*Open Authorization Protocol*) é um protocolo de autorização que permite que um aplicativo de terceiros possa obter acesso limitado a um serviço através de trocas de mensagens e *tokens* de acesso. A versão 1.0a é definida pela especificação RFC 5849 [39].

O OAuth 1.0 tem três entidades:

- **Dono do Recurso:** é o usuário que autoriza que uma aplicação acesse seus dados;
- **Consumidor ou Cliente:** aplicação que deseja acessar os dados do usuário;
- **Servidor:** Servidor que concede ao *token* de acesso após verificar a identidade do usuário e a concessão da autorização ao cliente.

O protocolo usa três pares diferentes de credenciais:

1. As credenciais do cliente obtidas durante o registro, consistindo de uma chave do consumidor e um segredo;
2. As credenciais temporárias formada de um *token* de requisição e um segredo do *token*;
3. Credenciais de acesso consistindo de um *token* de acesso e um segredo de *token* de acesso.

No OAuth 1.0a existe a opção da utilização de assinaturas das mensagens através do protocolo *Hash-based Message Authentication Code (HMAC)*. O cliente assina as requisições para o servidor usando as chaves simétricas pré compartilhadas. O segredo do consumidor é usado para assinar a solicitação de credenciais temporárias. Para solicitar as credenciais de acesso, a assinatura é feita utilizando a combinação do segredo do consumidor e o segredo de *token* de solicitação. Por fim, o segredo do consumidor e o segredo de *token* de acesso é usado para acessar o recurso protegido.

Existem ainda outras duas possibilidades para a segurança das requisições: utilizando o algoritmo RSA e sem o uso de assinaturas (*plaintext*). O uso de criptografia assimétrica com o protocolo RSA-SHA1 exige que o consumidor compartilhe previamente sua chave pública com o servidor. O método em texto claro (*plaintext*) não fornece nenhuma proteção de segurança e recomenda-se que apenas seja usado em um canal seguro, como HTTPS.

3.3.5 OAuth 2.0

A versão 2.0 do OAuth traz um protocolo completamente novo em relação a versão 1.0, publicado em dezembro de 2007, não sendo compatível com o mesmo [40]. O protocolo utiliza a arquitetura REST e mensagens JSON.

O OAuth 2.0 define quatro entidades:

- **Dono do Recurso:** é o usuário que autoriza que uma aplicação acesse seus dados;
- **Cliente:** aplicação que deseja acessar os dados do usuário;

- **Servidor do Recurso:** Servidor que hospeda os recursos protegidos, recebe e responde às solicitações de acesso aos recursos com os *tokens* do protocolo;
- **Servidor de Autorização:** Servidor que concede ao *token* de acesso após verificar a identidade do usuário e a concessão da autorização ao cliente.

A Figura 3.6 apresenta o fluxo abstrato do OAuth 2.0.

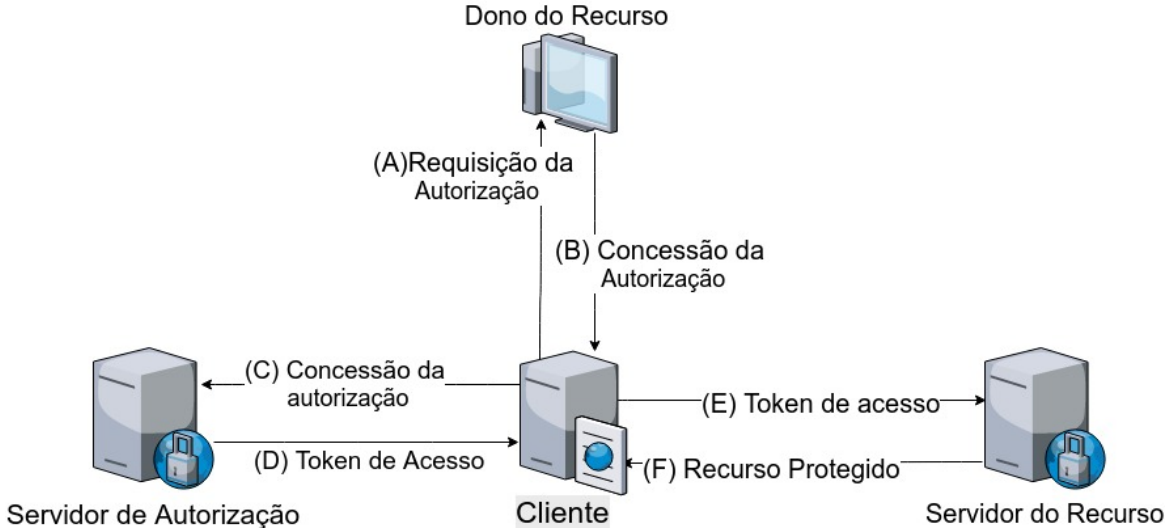

Figura 3.6: Fluxo Abstrato do OAuth 2.0 [5].

As etapas do diagrama incluem:

(A) O cliente solicita uma autorização para acessar os recursos do usuário;

(B) O usuário autoriza o pedido e o cliente recebe a concessão da autorização;

(C) O cliente requisita o *token* de acesso apresentando a concessão da autorização e suas credenciais ao servidor de autorização;

(D) O servidor de autorização autentica o cliente, valida a concessão da autorização e emite o *token* de acesso;

(E) O cliente requisita o recurso protegido e apresenta o *token* de acesso;

(F) O servidor do recurso valida o token de acesso e concede o acesso aos recursos solicitados.

O Oauth 2.0 define quatro tipos de concessão de autorização [5]:

1. por Código de Autorização;

2. Implícita;
3. *Resource Owner Password Credentials*;
4. *Client Credentials*.

Concessão por Código de Autorização

O *token* de acesso é obtido por meio do servidor de autorização. Em vez de solicitar autorização diretamente com o proprietário do recurso, o cliente direciona o proprietário do recurso para um servidor de autorização (através do *user-agent*).O cliente recebe o código de autorização após o servidor de autorização autenticar o proprietário do recurso. Nesse tipo de concessão o proprietário do recurso somente autentica no servidor de autorização, suas credenciais nunca são compartilhadas com o cliente.

A concessão por código de autorização fornece alguns benefícios de segurança, como: a capacidade de autenticar o cliente, a transmissão do *token* de acesso diretamente ao cliente sem passá-la através do *user-agent* do usuário não expondo a outras entidades. A Figura 3.7 apresenta as etapas da concessão por código de autorização.

(A) O cliente inicia o fluxo direcionando o *user-agent* do proprietário do recurso ao servidor de autorização. O cliente inclui seu identificador, o escopo, o estado e uma URI para o qual o servidor de autorização envia o *user-agent* de volta depois que o acesso é concedido (ou negado);

(B) O servidor de autorização autentica o proprietário do recurso (através do *user-agent*) e o proprietário do recurso concede ou nega o pedido de acesso do cliente;

(C) Supondo que o proprietário do recurso conceda o acesso, o servidor de autorização redireciona o *user-agent* para o cliente usando o redirecionamento de Uniform Resource Identifier (URI) fornecido anteriormente (no pedido ou durante o registro do cliente). O redirecionamento de URI inclui um código de autorização e de qualquer estado local fornecido pelo cliente no início.

(D) O cliente solicita um *token* de acesso de ponto de extremidade do servidor autorização do *token*, incluindo o código de autorização recebido na etapa anterior. Ao fazer a solicitação, o cliente autentica-se com o servidor de autorização. O cliente inclui o redirecionamento que URI usado para obter o código de autorização para verificação.

(E) O servidor de autorização autentica o cliente, o código de autorização e garante que o redirecionamento URI recebido corresponde a URI usada para redirecionar o

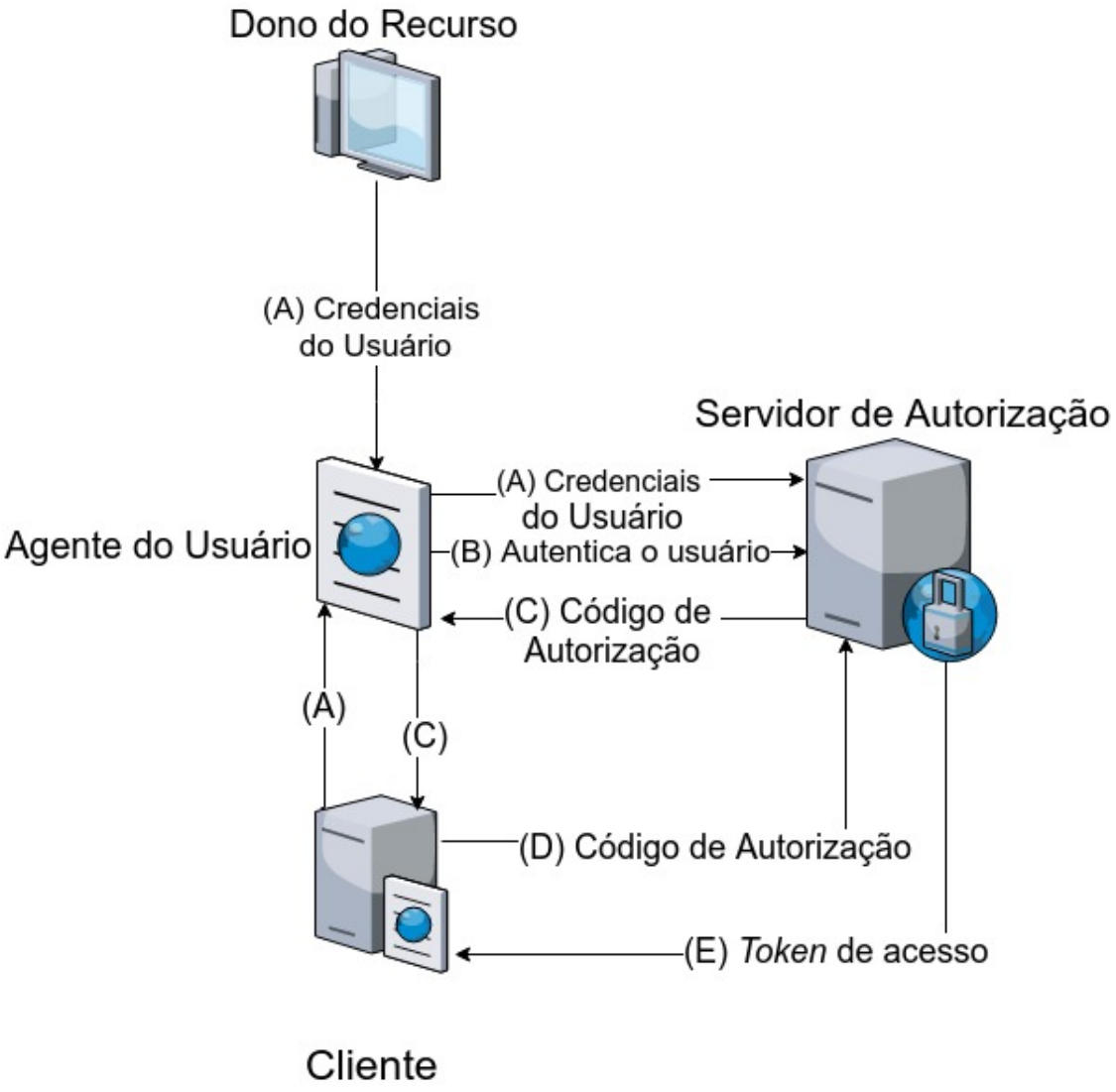

Figura 3.7: Concessão por Código de Autorização [5].

cliente na etapa **(C)**. Se válido, o servidor de autorização responde com um *token* de acesso e, opcionalmente, um *token* de atualização.

Concessão Implícita

É usado por aplicativos de cliente que podem executar *JavaScript* no agente do usuário. Na concessão implícita, em vez de emitir um código de autorização ao cliente, é emitido um *token* após a autorização do proprietário do recurso. Ao emitir um *token* de acesso durante o fluxo de concessão implícita, o servidor de autorização não autentica o cliente. Em alguns casos, a identidade do cliente pode ser verificada através do redirecionamento de URI usado para entregar o *token* de acesso ao cliente. O *token* de acesso pode ser exposto para o proprietário do recurso ou outros aplicativos com acesso para *user-agent*.

Subsídios implícitos melhoram a capacidade de resposta e a eficiência de alguns clientes (por exemplo, um cliente implementado como um aplicativo no navegador), uma vez que reduz os passos necessários para obter um *token* de acesso [5].

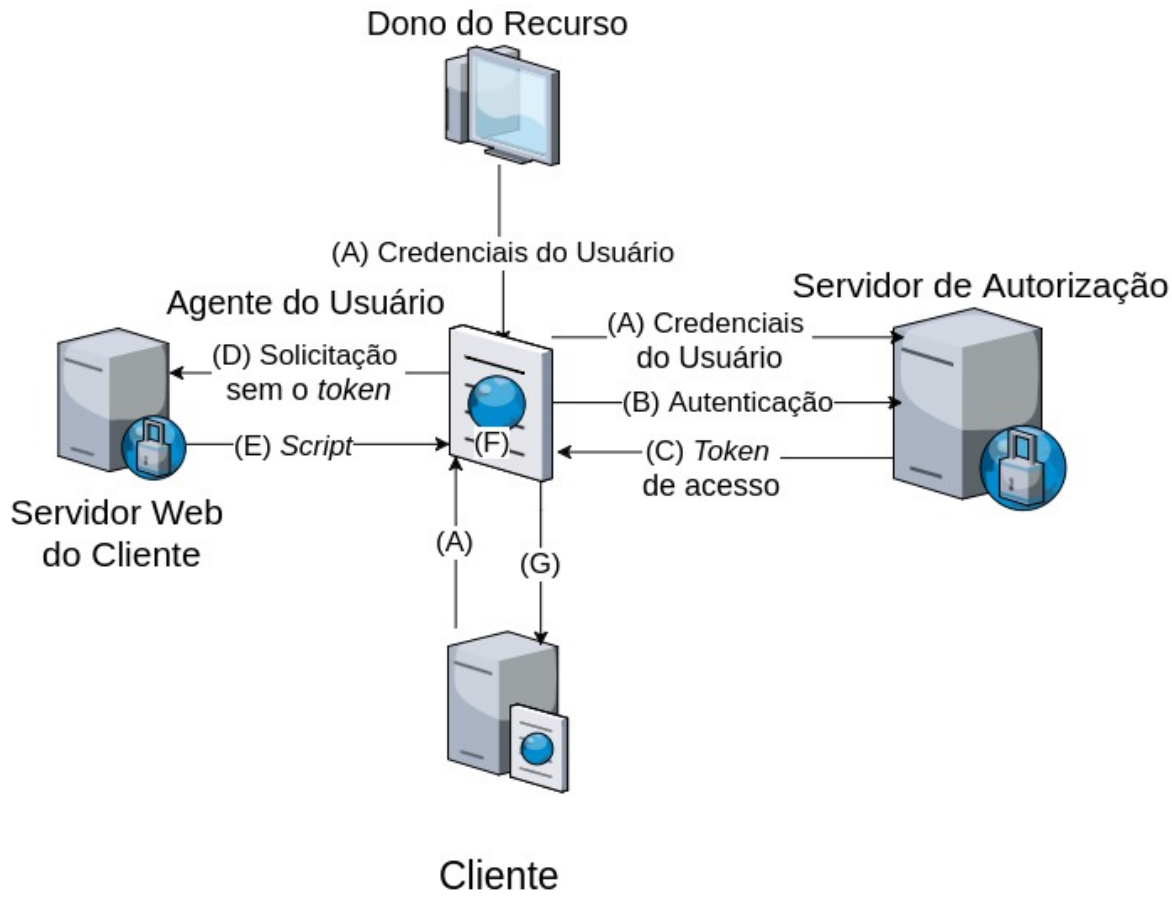

Figura 3.8: Concessão Implícita [5].

(A) O cliente inicia o fluxo direcionando o *user-agent* do proprietário do recurso ao servidor de autorização. O cliente inclui seu identificador, o escopo, o estado e uma URI para o qual o servidor de autorização envia o *user-agent* de volta depois que o acesso é concedido (ou negado);

(B) O servidor de autorização autentica o proprietário do recurso (através do *user-agent*) e o proprietário do recurso concede ou nega o pedido de acesso do cliente;

(C) O servidor de autorização redireciona o *user-agent* para a URI do cliente. O *token* de acesso é enviado na URI;

(D) O *user-agent* faz a requisição do recurso sem enviar o *token* de acesso, a informação do fragmento da URI é retida localmente;

(E) O hospedeiro do recurso retorna um *script* capaz de acessar o *token* de acesso retido pelo *user-agent*.

(F) O *user-agent* executa o *script* e extrai o *token* de acesso;

(G) O *user-agent* envia o *token* de acesso para o cliente.

Resource Owner Password Credentials

Nesse tipo de concessão as credenciais do proprietário de recurso são usadas para obter o *token* de acesso diretamente. A concessão por credenciais do proprietário do recurso pode ser usada quando há um alto grau de confiança entre o cliente e o proprietário do recurso e quando outros tipos de concessão de autorização não estão disponíveis.

As credenciais do usuário são utilizadas para uma única solicitação e são trocadas por um *token* de acesso. Este tipo de concessão elimina a necessidade do cliente para armazenar as credenciais do usuário para uso futuro, utilizando um *token* de acesso de longa duração ou *token* de atualização [5].

A Figura 3.9 apresenta os passos da concessão por credenciais do proprietário do recurso.

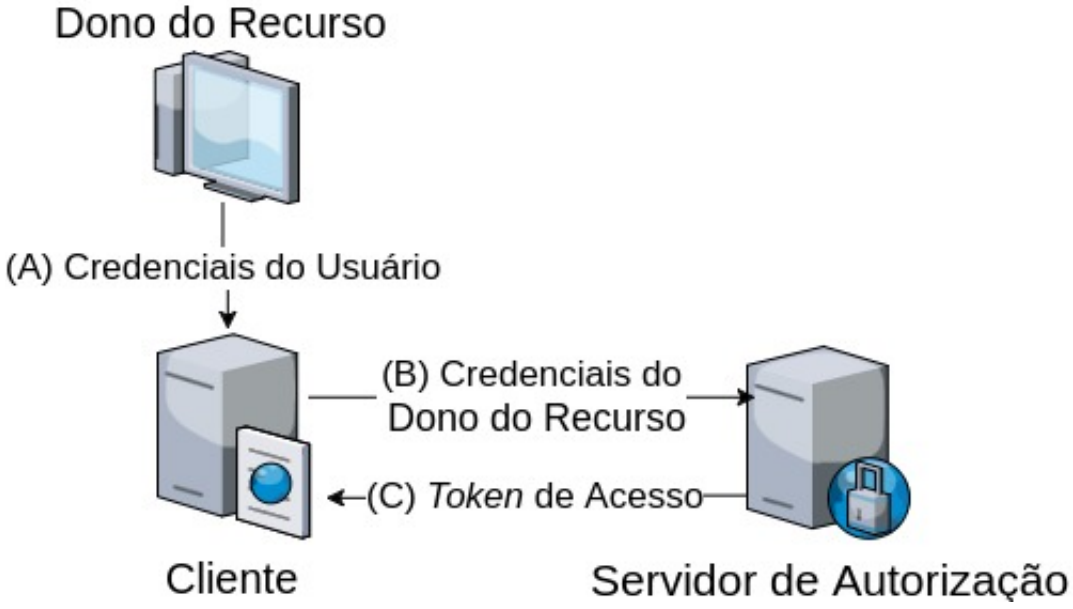

Figura 3.9: Concessão por Credenciais do Proprietário do Recurso [5].

(A) O proprietário do recurso fornece ao cliente seu nome de usuário e senha;

(B) O cliente requisita o *token* de acesso ao servidor de autorização e envia as credenciais do usuário;

(C) O cliente requisita o *token* de acesso apresentando a concessão da autorização e suas credenciais ao servidor de autorização.

Client Credentials

Nesse tipo de concessão são utilizadas as credenciais do cliente. É utilizado quando os recursos protegidos são do controle do próprio cliente ou quando há uma prévia combinação de autorização entre o cliente e o servidor de autorização [5].

As etapas da concessão por credenciais do cliente são apresentadas na Figura 3.10.

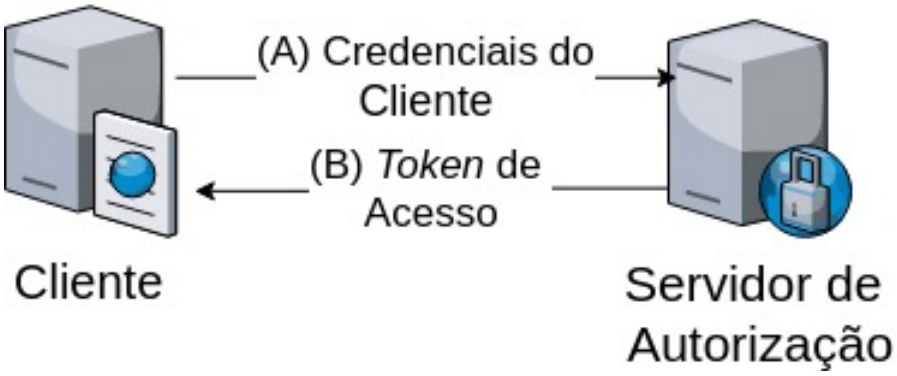

Figura 3.10: Concessão por credencial do cliente [5].

(A) O cliente se autentica junto ao servidor de autorização e faz a requisição do *token* de acesso;

(B) Se as credenciais do cliente forem validadas o servidor de autorização concede o *token* de acesso.

3.4 Trabalhos Relacionados

O estudo [23] apresenta uma avaliação sobre as técnicas, ferramentas e procedimentos utilizados para garantir a segurança em SOA. Para isso, foi feito um mapeamento sistemático utilizando a *string*: *SOA and SECURITY* nas bibliotecas IEEE Xplorer, ACM e *Digital Bibliography and Library Project (DBLP)*, após todas as etapas do mapeamento foram selecionados 25 artigos no total. O estudo propõe um protocolo de autenticação e autorização seguro. O protocolo é aderente ao estilo (REST), tem como finalidade possibilitar que uma arquitetura orientada a serviços possa ser adotada como alternativa única de integração e incorpora mecanismos de segurança tais como: criptografia, assinatura digital e o uso de certificados digitais. O protocolo é analisado de acordo com a lógica BAN.

O trabalho [41] propõe um modelo de controle de acesso unificado denominado *Unified Access Control Model (UACM)*, baseado na tecnologia de controle de acesso de segurança SAML (*Security Assertion Markup Language*). Esse modelo é utilizado para prover controle de acesso para compartilhamentos de recursos educacionais baseados na arquitetura SOA. O UACM (*Unified Access Control Model for Inter-platform Educational Resources*) é dividido em duas partes para garantir a segurança em SOA. O agente de segurança SOAP (SMSA) utiliza chaves assimétricas para garantir a autenticidade de usuários. Um outro módulo é que garante a validade da identidade e autoriza o acesso aos serviços baseado no *openSAML*. O modelo UACM foi testado em um ataque *man-in-the-middle*, que não obteve sucesso, o que demonstra a eficácia da criptografia das mensagens. Outro teste realizado foi em relação ao acesso de recursos por usuários autorizados ou não. O teste também foi concluído com sucesso.

Tassanaviboon propõe em [17] um modelo de autorização (AAuth) para acesso de arquivos em nuvem utilizando o protocolo OAuth e *tokens* com criptografia baseada em atributo (ABE - *Attribute-based encryption*). A confidencialidade e a integridade dos dados são asseguradas pelo esquema de criptografia utilizado e pelo uso de certificados registrados pelas entidades participantes. O teste de performance mostrou um aumento na latência considerado pequeno em relação ao uso do OAuth. O trabalho [42] propõe um modelo para integração de serviços utilizando REST e o protocolo OAuth 2.0 para realizar a autorização de acessos.

O artigo [43] apresenta alguns experimentos com o objetivo de estudar as vulnerabilidades do protocolo OAuth 2.0. Os testes foram executados tanto em clientes como em servidores de autorização. Os testes realizados foram: teste de *replay* em cima do código de autorização (os códigos devem ser usados uma única vez de acordo com a especificação [5]), ataques de *phishing* (onde um cliente malicioso captura os dados do usuário) e *Impersonation Attack* (ocorre a captura do código de acesso por agente malicioso). Por fim, o estudo descobriu algumas vulnerabilidades em implementações existentes do OAuth 2.0.

3.5 Síntese do Capítulo

Este Capítulo apresentou os principais conceitos referentes à SOA e à segurança utilizados neste trabalho. Os estudos correlatos permitiram identificar os principais desafios abordados, os principais ataques utilizados e as vulnerabilidades exploradas, na literatura relacionados com autenticação e autorização em SOA.

Capítulo 4

Protocolo de Autenticação e Autorização

Neste Capítulo será apresentado o protocolo de autorização implementado neste trabalho. Na Seção 4.1 são apresentados os requisitos da solução desejada pelo CPD. Na Seção 4.2 são discutidas as vulnerabilidades do protocolo OAuth 2.0. Na Seção 4.3 são apresentados alguns detalhes da implementação do protocolo OAuth 2.0 no *framework* ErlangMS. Na Seção 4.4 são apresentadas algumas considerações de segurança.

4.1 Requisitos

Os requisitos da solução de autorização propostos neste trabalho, são:

Requisito 1 : **Autorização**. O protocolo deverá conceder autorizações apenas aos recursos que o cliente autenticado tenha direito. As autorizações serão válidas por um período de tempo finito.

Requisito 2 : **Administrador**. O administrador do recurso deve ser capaz de revogar autorizações concedidas a qualquer momento.

Requisito 3 : **Integridade**. Deve ser possível detectar alterações ilícitas nas mensagens durante o processo de autenticação e autorização.

Requisito 4 : **Privacidade**. Dados restritos trocados durante o processo de autorização devem ser mantidos em sigilo de pessoas e serviços não autorizados;

Requisito 5 : **Compatibilidade com o barramento ErlangMS**. O protocolo deve usar a arquitetura REST e o formato JSON para troca de mensagens. Isto é necessário pois a solução proposta funcionará como um serviço no barramento proposto em [1].

Requisito 6 : **Baixo Acoplamento**. O protocolo não deve acoplar regras de negócio da UnB mantendo a característica do ErlangMS de baixo acoplamento.

A Tabela 4.1 apresenta uma comparação entre as soluções encontradas no mapeamento sistemático de acordo com os requisitos estabelecidos.

Tabela 4.1: Comparação entre as soluções encontradas.

Soluções	Req1	Req2	Req3	Req4	Req5	Req6
Oauth 2.0	X	X	X	X	X	X
SAML			X	X		X
OpenID Connect			X	X	X	X
XACML	X	X	X	X		X

O *OpenID Connect* e o SAML são soluções voltadas para autenticação. Normalmente são utilizadas para federação de identidade, processo em que domínios diferentes concordam que um conjunto designado de usuários pode ser autenticado por um determinado conjunto de critérios [32]. Portanto essas soluções não são aderentes ao requisito 1 e nem ao requisito 2, por tratarem de autenticação e não de autorização em SOA .

Quanto aos requisitos 3 e 4, todas as soluções tem mecanismos para garantir a integridade e a privacidade das informações utilizadas no processo de autenticação ou autorização. Porém, a especificação do Oauth 2.0 se restringe ao uso do protocolo TLS (*Transport Layer Security*).

O Oauth 2.0 e o *OpenID Connect* utilizam o mesmo estilo arquitetural (REST) e o mesmo padrão para troca de mensagens (JSON) que o barramento, sendo compatíveis com o requisito 5. Os padrões SAML e XACML utilizam o padrão SOAP e a linguagem XML para troca de mensagens, desta forma, ambos os padrões não são compatíveis com o ErlangMS. Logo, não sendo compatíveis com o requisito 5.

O protocolo OAuth 2.0 é compatível com todos os requisitos definidos na Seção 4.1 e será implementado e testado neste trabalho. A solução XACML é aderente aos requisitos de segurança porém não é compatível com o ErlangMS (requisito 5), esse padrão pode ser implementado em conjunto com o SAML formando uma solução de autenticação e autorização mais completa.

4.2 Vulnerabilidades do OAuth 2.0

Nesta Seção serão apresentados alguns ataques e vulnerabilidades conhecidos de implementações do protocolo OAuth 2.0. Na Seção 4.4 serão abordadas as soluções para as considerações de segurança levantadas.

4.2.1 Ataques em rede

Esta Seção descreverá vulnerabilidades relacionadas a captura e utilização das informações trocadas durante o processo de autorização, supondo que o atacante tenha acesso total à rede utilizada no processo. Para esta análise foi considerado o fluxo de concessão por código de autorização do OAuth 2.0.

Ataque *Replay* com o Código de Autorização

Uma comunicação crítica ocorre durante a solicitação do código de acesso, pois nesse caso os dados são expostos ao agente do usuário (por exemplo o navegador). O código de autorização em conjunto com o segredo do cliente pode ser utilizado para a obtenção do *token*.

Caso algum agente malicioso capture o código de autorização na comunicação entre agente do usuário e o cliente, pode reenviar o código ao cliente e dessa forma iniciar uma sessão com a conta associada ao dono do recurso que iniciou o processo de autorização [43].

Ataque de representação com o Código de Autorização

O atacante pode ainda se comportar como um intermediário entre o agente do usuário e o cliente. Neste caso o atacante consegue capturar o código de autorização e interromper a solicitação verídica do agente do usuário para manter o código funcional (que deve ser de utilização única conforme indicado na especificação [5].

Captura do *Token* de Acesso

O OAuth 2.0 utiliza por padrão *tokens* que dão acesso ao recurso protegido independentemente de outro tipo de verificação (*bearer tokens*). Desta forma, caso o atacante obtenha o *token* de acesso, poderá solicitar o recurso protegido a qualquer momento dentro da validade do *token*.

4.2.2 *Cross-Site Request Forgery (CSRF)*

Quando um usuário faz logon em um site, comumente é gerado um *cookie* como identificador de sessão. Todas as solicitações subsequentes do navegador para o site incluem este identificador e assim o site associa a solicitação à sessão do usuário. A vulnerabilidade CSRF (Falsificação de solicitação inter site) ocorre quando o site se baseia apenas no *cookie* para autorizar operações relacionadas à segurança.

Um site malicioso pode enganar o agente do usuário e enviar uma solicitação de *cross site* para o site vulnerável (usando *JavaScript*, HTTP *redirect* ou através de um link). O navegador então encaminhará automaticamente o *cookie* de sessão do usuário com este pedido forjado. Desta forma, o site vulnerável irá autorizar a solicitação maliciosa sem o conhecimento do usuário [44].

No OAuth 2.0 o invasor faz com que o agente de usuário siga um URI malicioso para um servidor confiável. Permitindo com que um cliente malicioso injete seu código de autorização ou *token* de acesso, o que pode resultar no cliente utilizando o *token* associado ao agente malicioso e não à vítima. Neste caso o atacante pode, por exemplo, obter informações pessoais da conta da vítima como um recurso protegido do invasor [45, 46].

4.3 OAuth 2.0 e ErlangMS

As soluções encontradas no Mapeamento Sistemático (Capítulo 2) e os requisitos definidos para a solução de segurança que será utilizada no barramento ErlangMS serviram de apoio para a escolha do protocolo de autorização implementado neste trabalho.

Desta forma, foi implementado no protocolo de autorização OAuth 2.0 como um serviço do barramento ErlangMS [1], dentro da arquitetura apresentada na Figura 3.2. A implementação do OAuth 2.0 foi facilitada pelo uso das bibliotecas disponibilizadas no barramento e na linguagem Erlang para tratar requisições HTTP/REST. O ErlangMS já possui suporte ao uso de TLS, desta forma, o OAuth 2.0 foi implementado respeitando os requisitos de integridade e privacidade.

4.3.1 Catálogo de serviços

Um elemento importante dentro do barramento ErlangMS é o catálogo de serviços. No catálogo são descritas, no formato JSON, as APIs dos serviços por meio da definição dos contratos de serviços. Para isso, deve-se especificar os serviços através de metadados, como por exemplo: a URL, o tipo de requisição REST (GET, POST, PUT, DELETE), os parâmetros, o dono do serviço, o tipo de autorização, entre outras informações. A publicação de um serviço no catálogo permite que o módulo *Dispatcher* do barramento possa rotear as requisições REST para os serviços solicitados pelos usuários, conforme mostra a figura 3.2 [1].

O catálogo de serviço do OAuth 2.0 descreve as requisições suportadas pela solução de acordo com a especificação RFC 6749 [5]. A definição na Figura 4.1, por exemplo, especifica uma solicitação de autorização, a requisição do tipo HTTP GET deve ser enviada utilizando a URL */authorize* para que o serviço do OAuth 2.0 seja iniciado.

```
{
 "name" : "/authorize",
 "comment": "OAuth 2.0",
 "owner": "oauth2ems",
 "version": "1",
 "service" : "oauth2ems_authorize:execute",
 "url": "/authorize",
 "type": "GET",
 "authorization" : "public",
 "querystring":
  [
        {
                "name": "response_type",
                "type": "string",
                "comment": "The expected response"
        },
        {
                "name": "client_id",
                "type": "string",
                "comment": "The client identifier"
        },
        {
                "name": "redirect_uri",
                "type": "string",
                "comment": "The redirect URI"
        },
        {
                "name": "scope",
                "type": "string",
                "comment": "The scope of the access request"
        }
 ],
 "lang" : "erlang"
},
```

Figura 4.1: Exemplo de catálogo de serviço.

Na Figura 4.1 é possível ver ainda a linguagem ao qual o serviço foi desenvolvido (Erlang), a função que será chamada para tratar a requisição (*oauth2ems_authorize:execute*) e os parâmetros aceitos.

O Apêndice A apresenta o catálogo de serviço completo do OAuth 2.0 no barramento ErlangMS.

4.3.2 Arquitetura do protocolo

Para o desenvolvimento do protocolo dentro do barramento ErlangMS foi utilizada a biblioteca do OAuth 2.0 em erlang disponível no sítio: `https://github.com/kivra/oauth2`. A Figura 4.2 mostra a troca de mensagens entre os módulos do protocolo implementado.

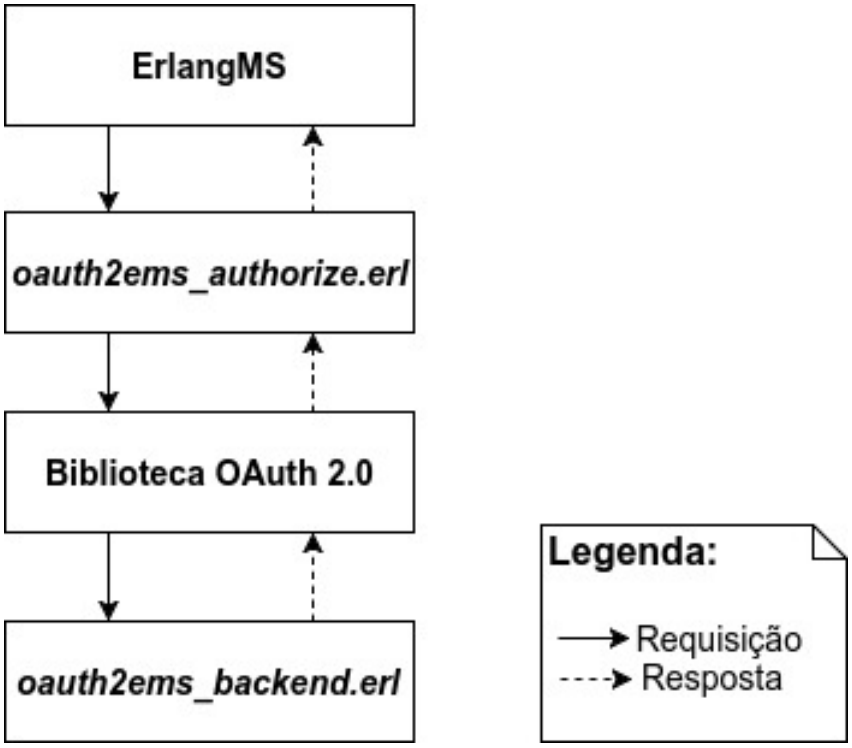

Figura 4.2: Arquitetura do OAuth 2.0 no ErlangMS

Desta forma a implementação do OAuth 2.0 no ErlangMS está organizada em três módulos:

- ***oauth2ems_authorize.erl*** - Possui funções que recebem a solicitação do barramento, extraem os dados da requisição, invocam a função correspondente da biblioteca do OAuth 2.0 e por fim retornam a resposta ao barramento.
- **Biblioteca do OAuth 2.0** - Possui funções que realizam a lógica do protocolo. Recebe os dados extraidos da solicitação e chama a função correspondente localizada no módulo *oauth2ems_backend*, trata sua resposta e passa ao módulo *oauth2ems_authorize*.
- ***oauth2ems_backend.erl*** - Neste arquivo estão as funções que fazem a consulta, atualização e inserção na base de dados utilizada no ErlangMS. Para os testes realizados nesta dissertação a base dados utilizada foi o sistema SCA utilizando o *driver Open DataBase Connectivity (ODBC)*.

O Apêndice B mostra os códigos gerados para a implementação do OAuth 2.0.

4.4 Considerações de Segurança

Nesta Seção é apresentado as soluções adotadas na implementação realizada para tratar as vulnerabilidades apresentadas na Seção 4.2.

4.4.1 Ataque *Replay* com o Código de Autorização

Para diminuir o risco dessa vulnerabilidade, o código de autorização que no fluxo de concessão por código de autorização representa a autorização por parte do usuário para o acesso aos recursos protegidos, deve ser descartado após a solicitação do *token* de acesso [5].

Neste caso a vulnerabilidade só seria explorada caso o atacante consiga capturar o código de autorização e realizar a solicitação do *token* de acesso ao servidor de autorização antes do cliente legítimo.

4.4.2 Ataque de representação com o Código de Autorização

Para diminuir a probabilidade desse ataque é recomendável utilizar o protocolo TLS para proteger a confidencialidade da comunicação e evitar intercepção das mensagens trocadas no processo de autorização. Para isso o barramento devera utiliza um certificado digital emitido por uma autoridade certificadora (CA) que esteja subordinada à hierarquia da Infraestrutura de Chaves Públicas para Ensino e Pesquisa (ICPEdu).

4.4.3 Captura do *Token* de Acesso

O *token* utilizado por padrão no OAuth 2.0 (*Bearer Token*) não é vinculado a nenhum outro parâmetro de segurança, podendo ser utilizado por qualquer um que o obtiver. Na solução adotada no ErlangMS foi adicionado o suporte a utilização de *tokens* do tipo MAC, este tipo de *token* é utilizado na versão 1.0a do protocolo OAuth.

A seguir serão mostrados os processos utilizados para acesso a um recurso utilizando um *bearer token* e um MAC *token*.

Bearer Token

A Figura 4.4 apresenta uma solicitação de *token* de acesso na concessão por código de autorização do OAuth 2.0. O código de acesso é enviado em conjunto com as credenciais do cliente (no campo "*Authorization*"do cabeçalho HTTP) e a URI de redirecionamento.

```
POST /token HTTP/1.1
Host: server.unb.br
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

Figura 4.3: Solicitação de *token* de acesso do OAuth 2.0.

Após a checagem dos parâmetros o servidor OAuth 2.0 concede *bearer token* conforme apresentado na Figura 4.5. A resposta também contém o tempo de duração do *token* (*"expires_in"*) e o *token* de atualização (*"refresh_token"*), que é uma credencial utilizada para obter um novo *token* de acesso após o atual expirar.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
   "access_token":"2YotnFZFEjr1zCsicMWpAA",
   "token_type":"bearer",
   "expires_in":3600,
   "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
}
```

Figura 4.4: *Token* de acesso do OAuth 2.0.

Para acessar o recurso protegido basta enviar os caracteres do *token* de acesso à requisição do recurso protegido, conforme apresentado na Figura 4.5.

```
GET /recurso HTTP/1.1
Host: unb.br
access_token=2YotnFZFEjr1zCsicMWpAA
```

Figura 4.5: Solicitação utilizando um *token* padrão do OAuth 2.0.

MAC *Token* do OAuth 1.0

A solicitação de um *token* de acesso no OAuth 1.0 é mais complexa do que na versão 2.0 do protocolo conforme apresentado na Figura 4.6.

Na versão 1.0 existem um par de credenciais de *token* temporárias para a solicitação do *token* de acesso. O *token* temporário (*"oauth_token"*) é enviado como parâmetro na requisição, o segredo do *token* temporário(*"oauth_token_secret"*) é utilizada como chave,

```
POST /request_token HTTP/1.1
Host: server.example.com
Authorization: OAuth
    oauth_consumer_key="jd83jd92dhsh93js",
    oauth_token="hdk48Djdsa",
    oauth_signature_method="HMAC-SHA1"
    oauth_verifier="473f82d3",
    oauth_nonce="asadaggfr",
    oauth_timestamp="134657964",
    oauth_signature="ja893SD9%26xyz4992k83j47x0b"
```

Figura 4.6: Solicitação do *token* de acesso no OAuth 1.0a.

em conjunto com o segredo do cliente, para gerar a assinatura ("*oauth_signature*"). A assinatura, que é o código MAC para verificação da mensagem, é gerada com a seguinte fórmula:

```
oauth_signature = HMAC - SHA1(chave, string)
```

Onde a chave é formada pelo segredo do token temporário e pelo segredo do consumidor separados pelo caractere &, a *string* é concatenação dos elementos de solicitação em uma única sequência de caracteres [39].

Os parâmetros *nonce* e *timestamp* são utilizados para evitar a execução de um ataque *replay* na solicitação. O "*oauth_nonce*"é uma sequência aleatória de caracteres, gerada pelo cliente para permitir que o servidor verifique se uma solicitação nunca foi feita antes. O valor do *nonce* deve ser exclusivo em todas as solicitações com o mesmo *timestamp*, credenciais do cliente e de *token* [39].

A Figura 4.7 apresenta a resposta da solicitação do *token* de acesso.

```
HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded
oauth_token=j49ddk933skd9dks
        &oauth_token_secret=ll399dj47dskfjdk
```

Figura 4.7: *Token* de acesso no OAuth 1.0a.

Junto com o *token* do tipo MAC é emitido o segredo utilizado para gerar o código de autenticação de mensagem (MAC) usado para assinar determinados componentes das solicitações HTTP/HTTPS. Esta assinatura é enviada junto com o *token* na solicitação de acesso ao recurso protegido. Este processo é igual ao realizado na solicitação do *token* de acesso.

A Figura 4.8 apresenta uma solicitação de acesso à um recurso protegido utilizando um MAC *token*.

```
GET /recurso HTTP/1.1
Host: unb.br
Authorization:
oauth_consumer_key="dpf43f3p2l4k3l03",
oauth_token="nnch734d00sl2jdk",
oauth_signature_method="HMAC-SHA1",
oauth_timestamp="137131202",
oauth_nonce="chapoH",
oauth_signature="MdpQcU8iPSUjWoN%2FUDMsK2sui9I%3D"
```

Figura 4.8: Solicitação utilizando um MAC *token*.

MAC *Token* do ErlangMS OAuth 2.0

Na solicitação do MAC *token* no OAuth 2.0 o código de acesso é enviado de forma isolada, ao contrário do que ocorre com o *bearer token*. Isto se deve ao fato do segredo do cliente ser a chave para gerar a assinatura da requisição do recurso, por isso não deve ser enviada durante o processo de autenticação. A Figura 4.9 apresenta uma solicitação de MAC *token*.

```
POST /mac_token HTTP/1.1
Host: server.unb.br
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
```

Figura 4.9: Solicitação de MAC *token* de acesso no ErlangMS OAuth 2.0.

A resposta da solicitação do *token* de acesso é apresentada na Figura 4.10 . Os segredos dos *tokens* são adicionados à resposta, eles serão utilizados para geração da assinatura. O processo é idêntico ao usado na versão 1.0 do OAuth.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
    "access_token":"2YotnFZFEjr1zCsicMWpAA",
    "token_secret":"axRGVZFEjr1zCsiWqSXDFF",
    "token_type":"mac",
    "expires_in":3600,
    "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
    "refresh_token_secret":"axRGVZFEjr1zCsiWqSXDFF",
}
```

Figura 4.10: *Token* de acesso do ErlangMS OAuth 2.0.

A Figura 4.11 apresenta a solicitação de acesso à um recurso protegido utilizando um MAC *token*. A solicitação é semelhante a do OAuth 1.0a, existem apenas adequações nas nomenclatura dos parâmetros para respeitar o RFC 6749 [5].

```
GET /recurso HTTP/1.1
Host: unb.br
Authorization:
        client_id="dpf43f3p2l4k3l03",
        access_token="nnch734d00sl2jdk",
        signature_method="HMAC-SHA1",
        timestamp="137131202",
        nonce="chapoH",
        signature="MdpQcU8iPSUjWoN%2FUDMsK2sui9I%3D"
```

Figura 4.11: Solicitação utilizando um MAC *token*.

Cross-Site Request Forgery (CSRF)

Para evitar este ataque, a especificação do OAuth 2.0 recomenda que os clientes gerem um valor que está vinculado à sessão do proprietário do recurso e que seja difícil de ser adivinhado (um *hash* de *cookie*, por exemplo) e passe como a variável *state* na requisição do código de autorização ou do *token* de acesso. O servidor de autorização simplesmente retorna este parâmetro em sua resposta, assim o cliente pode verificar se o *token* retornado ou código destina-se para a sessão atual [5].

4.5 Síntese do Capítulo

Este Capítulo apresentou detalhes da escolha e implementação do protocolo OAuth 2.0 na plataforma ErlangMS. Na sequência foram feitas algumas considerações de segurança, a principal delas foi a adição do suporte a *tokens* do tipo MAC utilizado no protocolo. No Capítulo 5 será apresentado os testes realizados no protocolo implementado.

Capítulo 5

Análise do Protocolo

Com a definição da solução que será implementada na UnB é necessário avaliar o impacto do protocolo no ambiente do Centro de Informática. Este Capítulo apresenta a avaliação de desempenho do protocolo OAuth 2.0 implementado no ErlangMS.

5.1 Teste de Performance

A realização de testes de performance tem por objetivo determinar se o desempenho do sistema é adequado com os requisitos do sistema [47].

Para a realização do teste de performance foi utilizada a abordagem *Goals, Questions and Metrics (GQM)*, proposta por van Solingen em [48]. Essa abordagem propõe que a definição das métricas deve ser de cima para baixo, com base nos objetivos e nas questões. O primeiro passo é definir o objetivo que irá especificar a finalidade da métrica. No segundo passo são elaboradas as questões que caracterizam o objeto da medição em relação a um problema de qualidade. No terceiro passo são definidas as métricas necessárias para responder as questões levantadas. Uma mesma métrica pode ainda ser usada para responder a questões diferentes com o mesmo objetivo.

5.1.1 Objetivos

Foram definidos os seguintes objetivos para o teste de desempenho:

Objetivo 1 : Verificar o aumento no tempo de resposta adicionado pelo protocolo de autorização implementado.

Objetivo 2 : Verificar o comportamento do protocolo para a demanda de requisições esperadas para a UnB.

5.1.2 Questões

Foram elaboradas questões que caracterizam os objetivos, as repostas informam se os objetivos definidos podem ser atendidos. As questões definidas foram:

Questão 1 : Qual o aumento no tempo de resposta adicionado pelas mensagens do protocolo de autorização?

Questão 2 : O protocolo suporta a demanda de requisições esperadas para a UnB?

5.1.3 Métricas

As seguintes métricas foram definidas para responder as questões elaboradas para o teste de desempenho do protocolo:

Métrica 1 : **Latência**: tempo entre a chegada da requisição e a produção da resposta do sistema.

Métrica 2 : ***Throughput***: número de requisições ou transações atendidas por unidade de tempo.

Com isso iremos realizar dois tipos de teste de desempenho: o teste de carga e o teste de estresse. Teste de carga avaliam o desempenho do sistema utilizando uma carga de usuários simultâneos reais e o teste de estresse avaliam o desempenho considerando o número máximo de usuários simultâneos que o sistema pode suportar [49]. Estes tipos de testes são importantes para os sistemas desenvolvidos na UnB devido a grande quantidade de usuários ativos da Universidade.

5.1.4 Ambiente de teste

Com o objetivo de avaliar o impacto do protocolo foi implementado um serviço no barramento que retorna informações de rede do ErlangMS. O serviço foi implementado em três cenários de teste de acordo com o modo de acesso:

Cenário 1 : No primeiro cenário o serviço retorna as informações de maneira pública (sem a necessidade de autorização) e sem o uso do protocolo TLS;

Cenário 2 : No segundo cenário as informações são obtidas após a apresentação de um *token* do OAuth 1.0. O *token* é concedido após o processo de autorização do protocolo, todas as mensagens trocadas entre cliente e servidor de autorização são assinadas e verificadas utilizando códigos MAC. A Figura 5.1 apresenta as mensagens trocadas durante o processo de autorização.

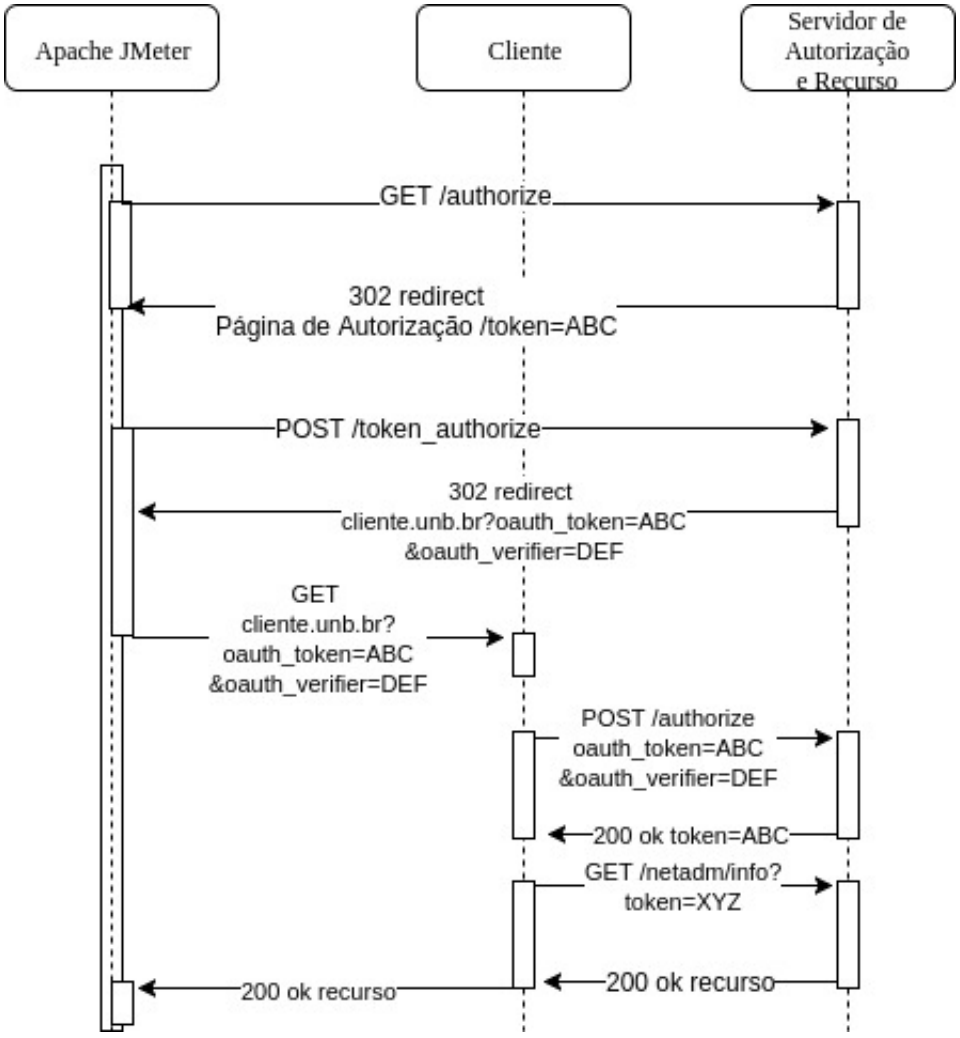

Figura 5.1: Cenário de teste com o uso do OAuth 1.0.

Cenário 3 : No terceiro cenário as informações são retornadas após a apresentação de um *token* do OAuth 2.0 concedido através de um processo de concessão por código de autorização. As mensagens trocadas durante o processo estão detalhadas no diagrama da Figura 5.2. Neste caso, foi usada a mesma máquina para os servidores de autorização e de recurso.

Cenário 4 : No quarto cenário as informações são retornadas após a apresentação de um MAC *token* do OAuth 2.0 concedido através de um processo de concessão por código de autorização. As mensagens trocadas durante o processo são as mesmas do cenário 3 representadas pelo diagrama da Figura 5.2, a única diferença está na solicitação de acesso ao serviço que agora passa pela verificação de integridade através de código MAC.

Para a realização do teste foram utilizados dois servidores de rede com as mesmas

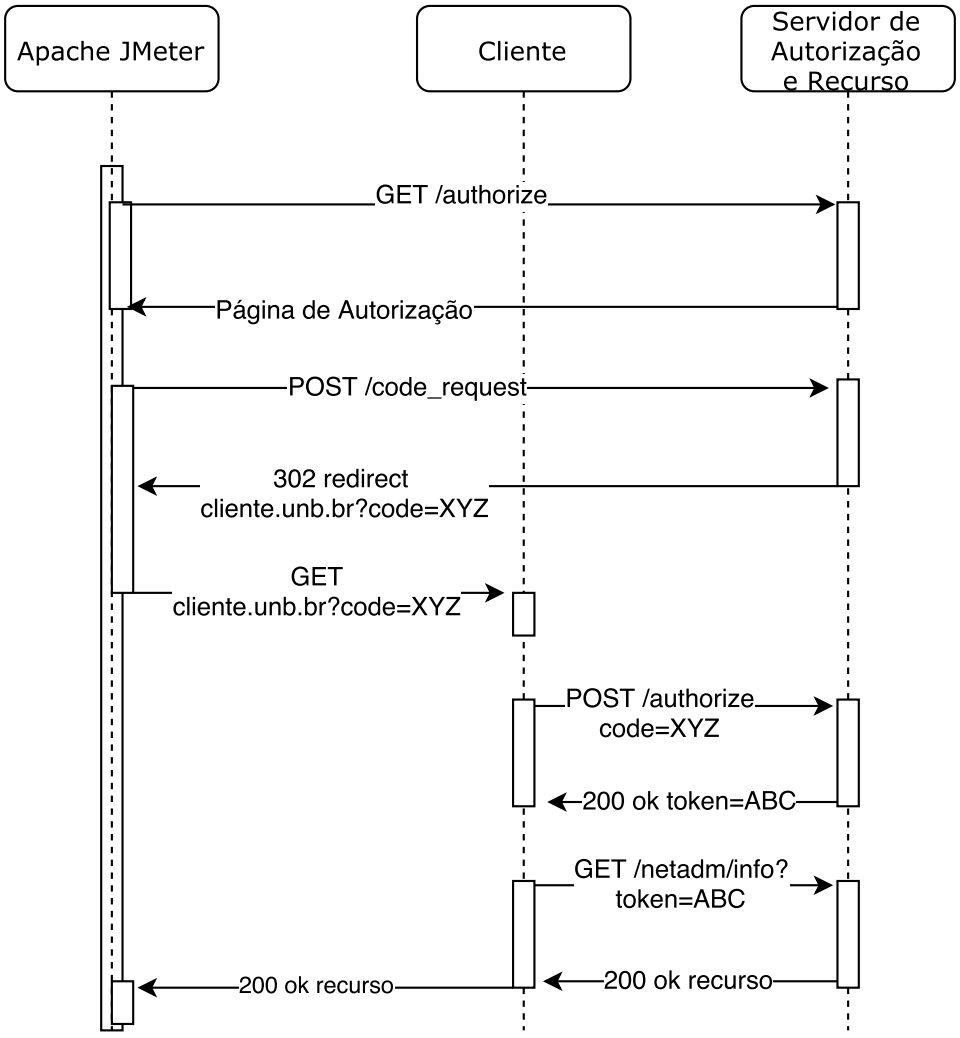

Figura 5.2: Cenário de teste com o uso do OAuth 2.0.

configurações de *hardware* onde foram instalados o cliente e o servidor de autorização/-recurso. Os serviços foram desenvolvidos em linguagem de programação Erlang. Todas as máquinas estão conectadas ao mesmo *switch* e estão na mesma rede lógica conforme apresentado na Figura 5.3. Todas as máquinas utilizavam o sistema operacional Ubuntu *Server* 16.04.2 LTS. As requisições foram disparadas do *desktop*.

Na Tabela 5.1 estão especificados a infraestrutura computacional utilizada nos testes.

Foram geradas diferentes quantidades de requisições para a verificação da latência média e do *throughput* nos três cenários propostos de autorização, através da ferramenta Apache JMeter [50].

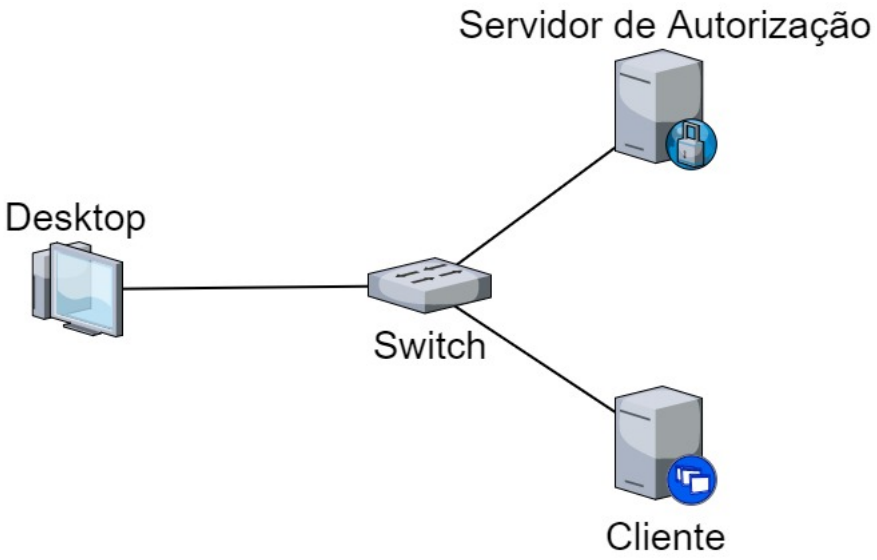

Figura 5.3: Ambiente de teste.

Tabela 5.1: *Infraestrutura de Hardware* utilizada nos testes.

	Tipo	**Características**
Cliente	Servidor	Processador Intel Xeon 3,1 GHz, 4 GB de memória RAM
Servidor de Autorização e Recurso	Servidor	Processador Intel Xeon 3,1 GHz, 4 GB de memória RAM
Desktop	Computador de mesa	Processador Intel i5 2,4 GHz, 4 GB de memória RAM

5.1.5 Resultados

Para responder a questão 1 foram analisados os valores das latências nos 4 cenários definidos anteriormente (acesso direto aos dados, acesso via OAuth 1.0a, acesso via OAuth 2.0 e acesso via OAuth 2.0 com MAC *tokens*). Os cenários foram submetidos a diferentes quantidades de requisições, conforme apresentado na Tabela 5.2.

O resultado apresentado evidencia o aumento na latência ocasionado pelo uso dos protocolos de autorização, conforme mostra a Figura 5.4, o que já era esperado pelo aumento no número de mensagens introduzidas pelo OAuth 1.0 e pelo tipo de concessão por código de autorização do OAuth 2.0. Conforme pode ser observado nas Figuras 5.1 e 5.2, os cenários 2, 3 e 4 utilizam cinco solicitações diferentes, além de utilizar uma terceira entidade para a autorização. Além disso o uso do protocolo TLS e a autenticação de mensagens utilizando HMAC contribuem para o aumento da latência. É possível

Tabela 5.2: Latência nos 04 cenários.

Requisições	Canário 1		Canário 2		Canário 3		Canário 4	
	Média	Mediana	Média	Mediana	Média	Mediana	Média	Mediana
100	3	2	106	100	105	103	101	99
1000	3	2	413	160	128	109	230	116
5000	11	6	608	372	471	194	510	258
10000	28	2	1231	922	634	350	996	892

observar o aumento da latência ocasionada pela autenticação das mensagens do protocolo OAuth 1.0 (cenário 2) e no OAuth 2.0 utilizando MAC (cenário 4) *tokens*.

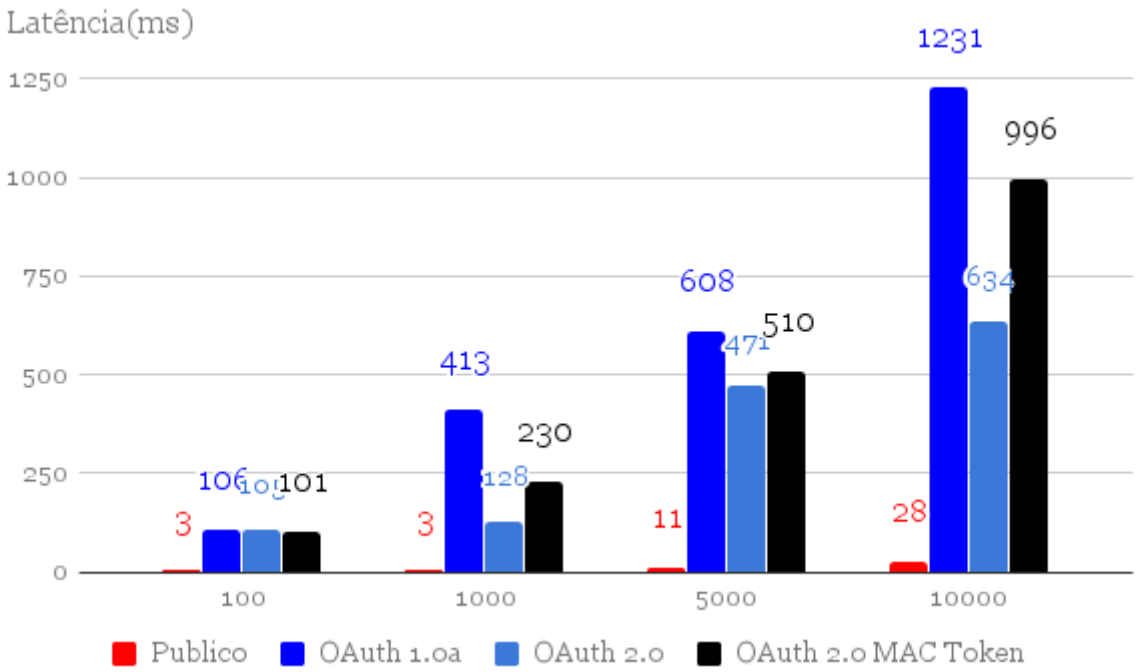

Figura 5.4: Latência.

Para responder a questão 2 foi utilizada a métrica *throughput*. Foi definido o valor médio dos acessos aos principais sistemas da UnB. Os acessos foram gerados e foi verificado o comportamento do sistema. A Tabela 5.3 apresenta a variação da vazão de acordo com o número de requisições.

O valor da vazão foi considerado aceitável para a realidade atual da UnB. Podemos observar que a vazão não variou muito com o aumento das requisições para os cenários que utilizavam o protocolo OAuth (Figura 5.5). Na comparação entre os cenários é possível observar a maior vazão no acesso público (Cenário 1). O OAuth 2.0 obteve uma vazão um pouco maior em relação ao OAuth 1.0.

Tabela 5.3: Vazão (req/s).

Requisições	Cenário 1	Cenário 2	Cenário 3	Cenário 4
100	49,6	45,5	45,9	45,9
1000	447,8	231,6	231,2	252,3
5000	2032,5	285,9	269,1	226,6
10000	3940,1	252,7	302,5	247,0

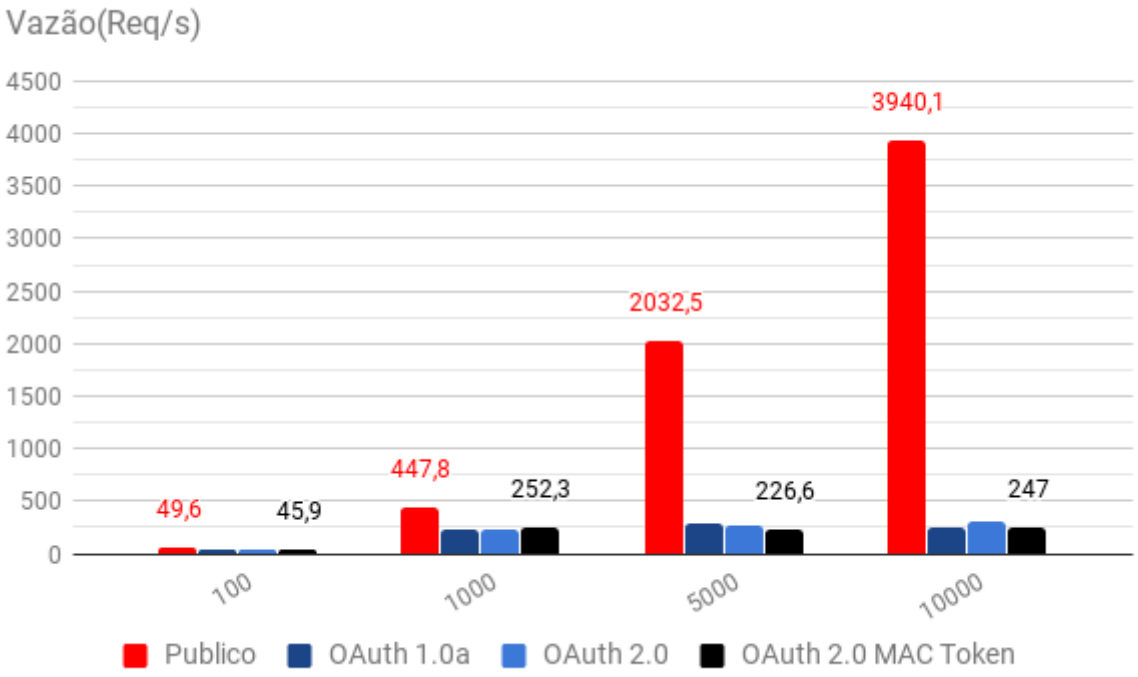

Figura 5.5: Vazão.

5.2 Teste de Segurança

Com o objetivo de testar a segurança do protocolo implementado neste trabalho foram realizadas duas simulações de ataque (ataque *replay* com o código de autorização e com o *token* de acesso). A estrutura computacional utilizada foi a mesma do teste desempenho detalhada na Figura 5.3 e na Tabela 5.1.

5.2.1 Ataque *Replay* com o Código de Autorização

O objetivo desta simulação é verificar a eficiência protocolo OAuth 2.0 no barramento ErlangMS frente a ocorrência de um ataque de repetição na solicitação do *token* de acesso.

As informações são acessadas após a apresentação de um MAC *token* do OAuth 2.0 concedido através de um processo de concessão por código de autorização. Porém durante o teste a solicitação do *token* é repetida simulando um ataque *replay*, conforme mostra a Figura 5.6.

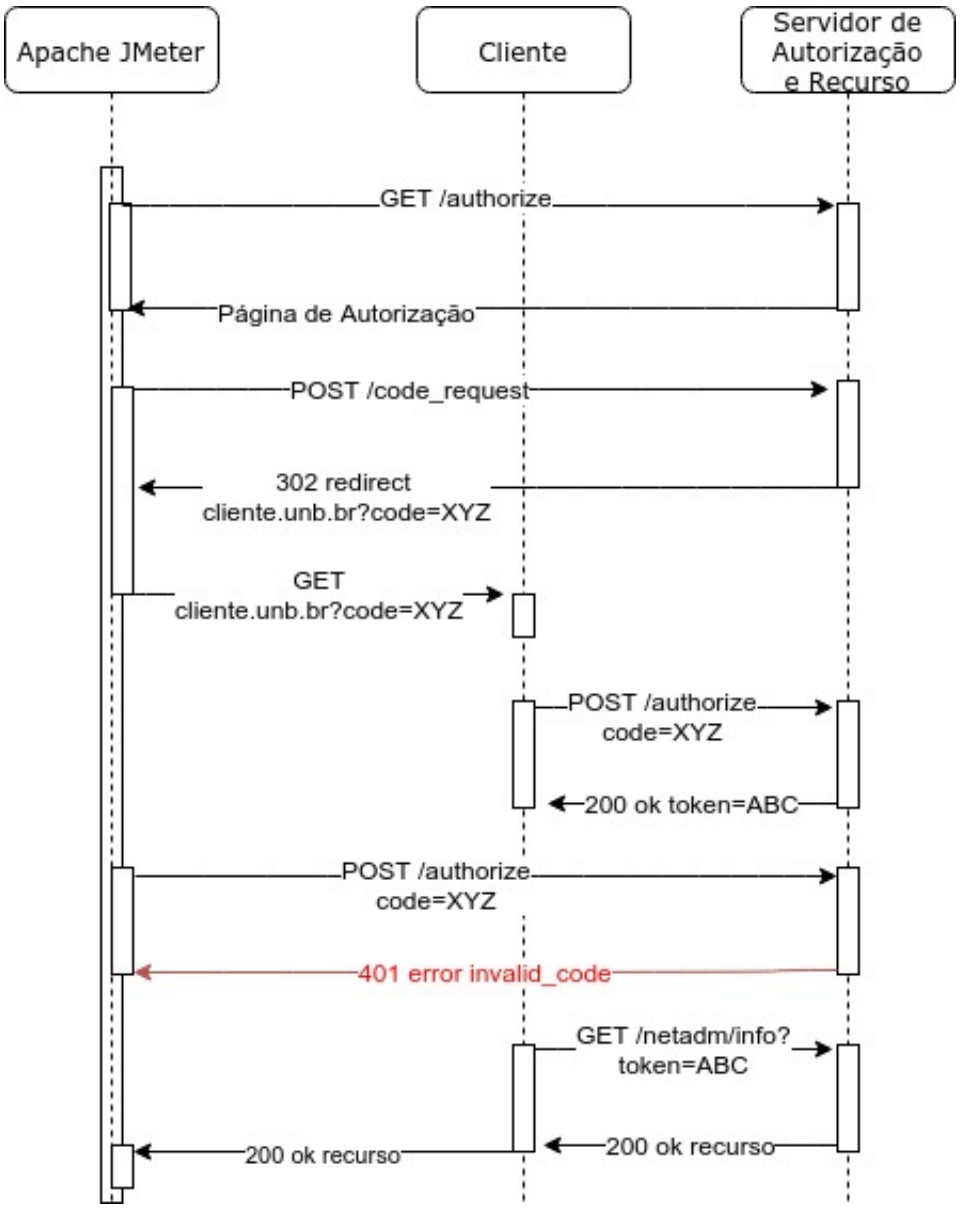

Figura 5.6: Cenário de ataque de repetição no código de autorização.

O código de autorização deve ser de uso único, com isso é esperado que o protocolo negue as solicitações contendo um código de autorização já utilizado.

A tabela 5.4 mostra os resultados.

É possível observar que a medida que as solicitações aumentaram, as requisições repetidas obtiveram mais sucessos. Isso pode ter ocorrido pelo aumento latência do protocolo

Tabela 5.4: Ataque *replay* utilizando o código de autorização.

Requisições	**Repetições**	**Repetições bem sucedidas**
100	100	0
1000	1000	0
5000	5000	3
10000	10000	12

em altas taxas de requisições, o que aumenta a probabilidade de que uma solicitação forjada enviada após a solicitação original seja processada antes pelo barramento. É possível ,também, que erros de rede possam ter ocorrido fazendo com que as requisições originais fossem perdidas.

5.2.2 Ataque Replay com o *MAC Token*

O objetivo desta simulação é verificar a eficiência protocolo OAuth 2.0 no barramento ErlangMS na ocorrência de um ataque de repetição na solicitação de acesso ao recurso protegido.

As informações são acessadas após a apresentação de um MAC token do OAuth 2.0 concedido através de um processo de concessão por código de autorização. Porém durante o teste a solicitação do recurso protegido é repetida simulando um ataque *replay*. A Figura 5.7 mostra as mensagens esperadas para esta simulação.

Os MAC *tokens* contém números aleatórios (parâmetro *nonce*) que identificam a solicitação de acesso ao recurso protegido. Desta forma o protocolo não aceita duas solicitações com o mesmo valor no parâmetro *nonce*, diminuindo de forma considerável a probabilidade de sucesso de um ataque de *replay* utilizando um MAC *token*.

A tabela 5.5 mostra os resultados da simulação. É possível observar que a medida que as solicitações aumentaram, as requisições repetidas obtiveram mais sucessos, assim como ocorreu na seção anterior.

Tabela 5.5: Ataque *replay* utilizando o código de autorização.

Requisições	**Repetições**	**Repetições bem sucedidas**
100	100	0
1000	1000	0
5000	5000	4
10000	10000	10

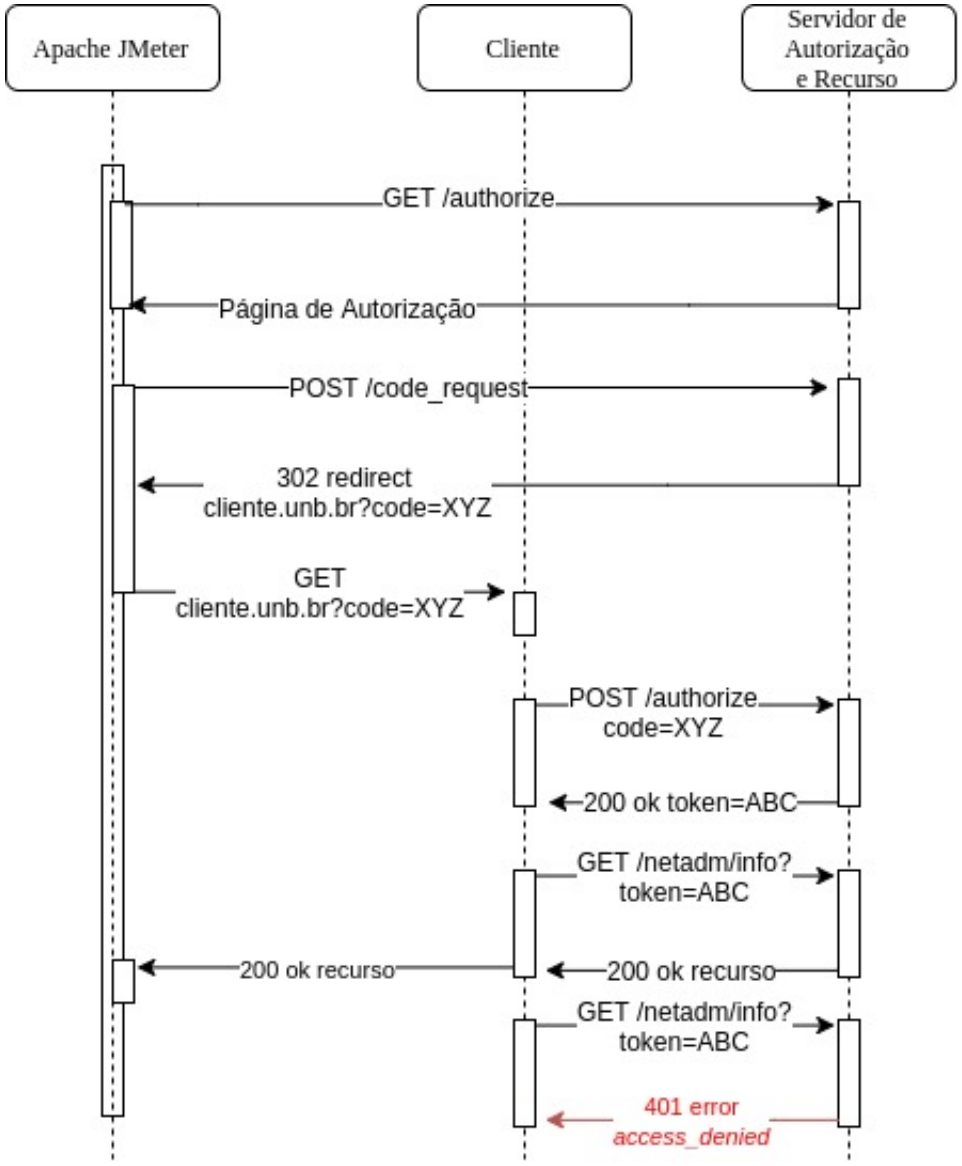

Figura 5.7: Cenário de ataque de repetição com o MAC *token*.

5.3 Discussão dos Resultados

Com o teste de desempenho foi possível verificar o impacto da utilização do OAuth 2.0 no ambiente da UnB. Quando comparado com o acesso ao serviço sem a utilização de um protocolo de autorização ocorreu um aumento significativo na latência.

Este aumento significativo era algo já esperado pelo aumento no número de mensagens trocadas, uma vez que o OAuth 2.0 faz o uso do protocolo TLS. Outro fator que aumenta a latência é o uso de funções de *hash* para assinatura das mensagens, o que pode ser observado na diferença entre os resultados das versões 1.0a e 2.0 do OAuth.

O OAuth 2.0 utilizando MAC *tokens* teve um desempenho melhor que o OAuth 1.0a,

isto acontece pois a versão 1.0a realiza autenticação em todas as mensagens trocadas entre cliente e servidor de autorização. Já a versão 2.0 utilizada no trabalho realiza o processo apenas na requisição do recurso protegido.

Os resultados da vazão foram considerados satisfatórios para todos os cenários com protocolo de autorização. Para 10000 requisições todos os protocolos obtiveram resultados acima de 247 requisições por segundo. O serviço que mais responde à requisições na Universidade de Brasília é o *Domain Name System (DNS)*, que atende em média a 235000 solicitações a cada 15 minutos (dado retirado do equipamento de *firewall* do CPD), dando uma vazão média 261,11 requisições por segundo. O valor é próximo ao alcançado pelo protocolo OAuth 2.0 no ErlangMS, porém o ErlangMS foi testado em um ambiente sem customizações e com hardware limitado quando comparado ao servidor de DNS.

Nos testes de segurança foram simuladas duas situações de ataque de repetição. Em ambos os casos os ataques foram eficientes em uma pequena porcentagem das tentativas (0,12% no pior dos cenários). As simulações porém só testaram os mecanismos de segurança do protocolo OAuth 2.0 implementado no ErlangMS, em cenário real a ocorrência destes ataques será ainda dificultada pelo uso do protocolo TLS (no cenário de testes os ataques vieram de entidades confiáveis). Com isso o desempenho do OAuth 2.0 nos testes de segurança foi considerado satisfatório.

5.4 Síntese do Capítulo

Este Capítulo apresentou os testes de desempenho, de acordo com os 03 cenários definidos, e segurança, simulação de duas situações de ataque de repetição. Os resultados mostram que a solução é adequada ao ambiente da UnB, podendo ser implantada pelo Centro de Informática (CPD/UnB), quando do uso do Barramento ErlangMS.

Capítulo 6

Conclusão

A arquitetura SOA possui características que apoiam o processo de modernização dos sistemas legados. O trabalho [1] deu início a utilização de SOA para a modernização dos sistemas legados no CPD/UnB. Esta abordagem porém carece de alguns cuidados de segurança, dentre eles a autorização aos serviços publicados no barramento.

Neste estudo foi apresentado o estado da arte atual sobre os protocolos de autenticação e autorização em SOA, levando em consideração os 45 artigos selecionados a partir de um mapeamento sistemático. Estes estudos foram utilizados para responder às questões de pesquisas definidas.

Com o mapeamento sistemático realizado neste trabalho foi possível identificar as principais soluções apresentadas para autenticação e autorização em SOA e em REST. É possível concluir que o protocolo OAuth 2.0 é a solução mais aderente ao barramento proposto para a modernização dos sistemas legados da UnB.

A implementação do OAuth 2.0 visa garantir o acesso aos recursos e serviços publicados no barramento apenas às aplicações (clientes) com o privilégio para tal. Esse requisito é essencial para o ErlangMS, contribuindo para a utilização da arquitetura SOA no processo de modernização dos sistemas legados da UnB de forma segura. O ErlangMS está disponível para a comunidade acadêmica em `https://github.com/erlangMS`.

Após análise de segurança verificou-se a necessidade de adicionar suporte a *tokens* de acesso com autenticação de mensagem via funções HMAC (MAC *tokens*). A principal vantagem que os *tokens* do tipo MAC oferecem é a verificação da integridade da mensagem através do uso de códigos MAC.

Foi realizada uma avaliação de desempenho na solução proposta, sendo possível observar o aumento da latência causado pelo maior número de mensagens trocadas no processo de autorização. A solução foi comparada com a versão 1.0a do OAuth (é um protocolo diferente e não compatível com a versão do 2.0). Foi possível observar que a versão 2.0 teve um menor aumento na latência e maior vazão em relação a versão 1.0a. Isto se deve

ao fato da versão 1.0a utilizar autenticação de mensagens via códigos MAC em todas as trocas de mensagens entre cliente e servidor de autorização. O protocolo suportou quantidades altas de requisições mostrando um resultado satisfatório tendo em vista o alto número de usuários da UnB.

Nos testes de segurança foi possível observar que os mecanismos implementados no protocolo foram eficientes na prevenção de ataques de repetição, no pior cenário ocorreram apenas 0,12% de ataques bem sucedidos. Em um cenário real a ocorrência destes ataques será ainda mais difícil pelo uso do protocolo TLS (nos testes de segurança não foi quebrada a segurança do TLS). Com isso o desempenho do OAuth 2.0 nos testes de segurança foi considerável satisfatório.

6.1 Contribuições

A principal contribuição deste trabalho é a implementação de uma solução de autorização na arquitetura adotada para apoiar a modernização dos sistemas legados na UnB. O protocolo OAuth 2.0 foi escolhido após a análise das soluções encontradas no mapeamento sistemático apresentado no Capítulo 2 e os requisitos necessários para a arquitetura ErlangMS definidos na Seção 4.1.

A versão 1.0a do protocolo OAuth foi implementada no barramento, sendo também uma contribuição para o projeto ErlangMS, visto que esta versão do protocolo poderá servir para outras Instituições que poderão utilizar a plataforma.

Outra contribuição importante foi o mapeamento sistemático (Capítulo 2) onde foi possível identificar as principais soluções e os principais desafios de autenticação e autorização em SOA.

Por fim, a fundamentação teórica (Capítulo 3) possibilitou o aprofundamento dos conhecimentos referentes à SOA e à segurança, sendo também uma contribuição importante. Foi possível ainda levantar, através dos estudos relacionados, os principais desafios enfrentados pelo protocolo OAuth 2.0.

6.2 Trabalhos Futuros

Para a continuidade deste trabalho pretende-se:

1. Avaliar a facilidade do uso do protocolo junto aos desenvolvedores de *software* da Universidade de Brasília;
2. Realizar uma análise formal do protocolo implementado;
3. Avaliar o comportamento do protocolo em um ambiente real;

4. Fazer uma análise de segurança em cima vulnerabilidades presentes nos clientes do OAuth 2.0.

Referências

[1] Vargas Agilar, Everton de e Rodrigo Bonifacio de Almeida: *Uma abordagem orientada a serviços para a modernização dos sistemas legados da unb*. 2016. xi, 1, 2, 13, 15, 17, 18, 19, 20, 21, 35, 38, 57

[2] Singh, Simon: *The Code Book: How to Make It, Break It, Hack It, Or Crack it*. Delacorte Press, 2002. xi, 22

[3] Santin, Altair Olivo *et al.*: *Teias de federações: uma abordagem baseada em cadeias de confiança para autenticação, autorização e navegação em sistemas de larga escala*. 2004. xi, 24

[4] OpenID: *Openid connect core 1.0*. 2012. `http://openid.net/specs/openid-connect-core-1_0.html`. xi, 25

[5] Hardt, Dick: *The oauth 2.0 authorization framework*. 2012. `https://tools.ietf.org/html/rfc6749`. xi, 28, 30, 31, 32, 33, 34, 37, 38, 41, 45

[6] Salvatierra, G., C. Mateos, M. Crasso, A. Zunino e M. Campo: *Legacy system migration approaches*. IEEE Latin America Transactions, 11(2):840–851, March 2013, ISSN 1548-0992. 1

[7] Xu, Jie, Dacheng Zhang, Lu Liu e Xianxian Li: *Dynamic authentication for cross-realm soa-based business processes*. Services Computing, IEEE Transactions on, 5(1):20–32, Jan 2012, ISSN 1939-1374. 1, 13

[8] Juric, Matjaz B: *SOA Approach to Integration: XML, Web services, ESB, and BPEL in real-world SOA projects*. Packt Publishing Ltd, 2007. 1, 17

[9] Petticrew, Mark e Helen Roberts: *Systematic reviews in the social sciences*. 2006. 5

[10] Keele, Staffs: *Guidelines for performing systematic literature reviews in software engineering*. Em *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*. 2007. 5

[11] Durbeck, S., C. Fritsch, G. Pernul e R. Schillinger: *A semantic security architecture for web services the access-egov solution*. Em *Availability, Reliability, and Security, 2010. ARES '10 International Conference on*, páginas 222–227, Feb 2010. 11

[12] Nordbotten, N.A.: *Xml and web services security standards*. Communications Surveys Tutorials, IEEE, 11(3):4–21, 2009, ISSN 1553-877X. 11, 14

[13] Torroglosa-García, Elena, Antonio D Pérez-Morales, Pedro Martinez-Julia e Diego R Lopez: *Integration of the oauth and web service family security standards*. Computer networks, 57(10):2233–2249, 2013. 12, 13, 14, 25

[14] Hummer, Waldemar, Patrick Gaubatz, Mark Strembeck, Uwe Zdun e Schahram Dustdar: *An integrated approach for identity and access management in a soa context*. Em *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, páginas 21–30, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0688-1. `http://doi.acm.org.ez54.periodicos.capes.gov.br/10.1145/1998441.1998446`. 12

[15] Sliman, L., Y. Badr, F. Biennier, N. Salatge e Zensho Nakao: *Single sign-on integration in a distributed enterprise service bus*. Em *Network and Service Security, 2009. N2S '09. International Conference on*, páginas 1–5, June 2009. 12

[16] Qi-rui, Peng, Wang Cheng, Wu Jing, Li Jun, Li Qing e Shao Bei-en: *An authentication and authorization solution supporting soa-based distributed systems*. Em *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*, páginas 535–538, July 2010. 12

[17] Tassanaviboon, Anuchart e Guang Gong: *Oauth and abe based authorization in semi-trusted cloud computing: Aauth*. Em *Proceedings of the Second International Workshop on Data Intensive Computing in the Clouds*, DataCloud-SC '11, páginas 41–50, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-1144-1. `http://doi.acm.org.ez54.periodicos.capes.gov.br/10.1145/2087522.2087531`. 12, 34

[18] Thomas, Ivonne, Michael Menzel e Christoph Meinel: *Using quantified trust levels to describe authentication requirements in federated identity management*. Em *Proceedings of the 2008 ACM Workshop on Secure Web Services*, SWS '08, páginas 71–80, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-292-4. `http://doi.acm.org.ez54.periodicos.capes.gov.br/10.1145/1456492.1456504`. 12

[19] Li, Jun e Alan H. Karp: *Access control for the services oriented architecture*. Em *Proceedings of the 2007 ACM Workshop on Secure Web Services*, SWS '07, páginas 9–17, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-892-3. `http://doi.acm.org.ez54.periodicos.capes.gov.br/10.1145/1314418.1314421`. 12

[20] Wolf, M., I. Thomas, M. Menzel e C. Meinel: *A message meta model for federated authentication in service-oriented architectures*. Em *Service-Oriented Computing and Applications (SOCA), 2009 IEEE International Conference on*, páginas 1–8, Jan 2009. 12

[21] Noureddine, Moustafa e Rabih Bashroush: *An authentication model towards cloud federation in the enterprise*. Journal of Systems and Software, 86(9):2269–2275, 2013. 12

[22] Noureddine, M. e R. Bashroush: *A provisioning model towards oauth 2.0 performance optimization*. Em *Cybernetic Intelligent Systems (CIS), 2011 IEEE 10th International Conference on*, páginas 76–80, Sept 2011. 12

[23] Conceição, Rogério Alves da: *Um Protocolo de Autenticação e Autorização Seguro para Arquiteturas Orientadas a Serviços*. Tese de Doutoramento, Universidade de Brasília, 2014. 13, 33

[24] Clements, Paul, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord e Judith Stafford: *Documenting Software Architectures: Views and Beyond (2nd Edition)*. Addison-Wesley Professional, 2ª edição, outubro 2010. `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321552687`. 16, 18

[25] Bianco, Philip, Grace Lewis, Paulo Merson e Soumya Simanta: *Architecting service-oriented systems*. Relatório Técnico CMU/SEI-2011-TN-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2011. `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9829`. 16

[26] Erl, Thomas: *Service-oriented architecture (soa): concepts, technology, and design*, 2005. 16

[27] Bianco, Philip, Rick Kotermanski e Paulo F Merson: *Evaluating a service-oriented architecture*. Relatório Técnico CMU/SEI-2007-TR-015, Software Engineering Institute, setembro 2007. 17, 18

[28] Fielding, Roy Thomas: *Architectural styles and the design of network-based software architectures*. Tese de Doutoramento, University of California, Irvine, 2000. 17

[29] Coyle, Frank P: *XML, Web services, and the data revolution*. Addison-Wesley Longman Publishing Co., Inc., 2002. 18

[30] Schmutz, Guido, Daniel Liebhart e Peter Welkenbach: *Service-oriented Architecture: An Integration Blueprint: a Real-world SOA Strategy for the Integration of Heterogeneous Enterprise Systems: Successfully Implement Your Own Enterprise Integration Architecture Using the Trivadis Integration Architecture Blueprint*. Packt Publishing Ltd, 2010. 18

[31] Haupt, Florian, Dimka Karastoyanova, Frank Leymann e Benjamin Schroth: *A model-driven approach for rest compliant services*. Em *Web Services (ICWS), 2014 IEEE International Conference on*, páginas 129–136. IEEE, 2014. 18

[32] Pulier, Eric e Hugh Taylor: *Understanding enterprise SOA*. Manning Greenwich, Conn, 2006. 20, 36

[33] ISO/IEC: *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001. 20

[34] Stallings, William: *Cryptography and network security: principles and practices*. Pearson Education India, 2006. 22, 23

[35] Krawczyk, Hugo, Ran Canetti e Mihir Bellare: *Hmac: Keyed-hashing for message authentication*. 1997. 23

[36] Prado Filho, Tito Gardel do e Cassio Vinicius Serafim Prazeres: *Multiauth-wot: A multimodal service for web of things athentication and identification*. Em *Proceedings of the 21st Brazilian Symposium on Multimedia and the Web*, WebMedia '15, páginas 17–24, New York, NY, USA, 2015. ACM, ISBN 978-1-4503-3959-9. `http://doi.acm.org.ez54.periodicos.capes.gov.br/10.1145/2820426.2820438`. 24

[37] OASIS: *Security assertion markup language*. OASIS Standard, 2008. `http://saml.xml.org/`. 25, 26

[38] Rissanen, Erik *et al.*: *extensible access control markup language (xacml) version 3.0*, 2013. 26

[39] IETF: *The oauth 1.0 protocol*. RFC, 2010. `https://tools.ietf.org/html/rfc5849`. 26, 43

[40] Hammer-Lahav, E: *Introducing oauth 2.0*. Hueniverse, May, 2010. `https://hueniverse.com/2010/05/15/introducing-oauth-2-0/`. 27

[41] Shang, C., Z. Yang, Q. Liu e C. Zhao: *Saml based unified access control model for inter-platform educational resources*. Em *Computer Science and Software Engineering, 2008 International Conference on*, volume 5, páginas 909–912, Dec 2008. 34

[42] Memeti, A., B. Selimi, A. Besimi e B. Çiço: *A framework for flexible rest services: Decoupling authorization for reduced service dependency*. Em *2015 4th Mediterranean Conference on Embedded Computing (MECO)*, páginas 51–55, June 2015. 34

[43] Yang, F. e S. Manoharan: *A security analysis of the oauth protocol*. Em *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, páginas 271–276, Aug 2013. 34, 37

[44] Bansal, C., K. Bhargavan e S. Maffeis: *Discovering concrete attacks on website authorization by formal analysis*. Em *2012 IEEE 25th Computer Security Foundations Symposium*, páginas 247–262, June 2012. 38

[45] Shernan, Ethan, Henry Carter, Dave Tian, Patrick Traynor e Kevin Butler: *More guidelines than rules: Csrf vulnerabilities from noncompliant oauth 2.0 implementations*. Em *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, páginas 239–260. Springer, 2015. 38

[46] Kaur, G e D Aggarwal: *A survey paper on social sign-on protocol oauth 2.0*. J. Eng. Comput. Appl. Sci, 2(6):93–96, 2013. 38

[47] Zenteno, Torres AH, E Martins, MJE Cuaresma *et al.*: *Teste de desempenho em aplicações sig web*. 2006. 46

[48] Van Solingen, Rini, Vic Basili, Gianluigi Caldiera e H Dieter Rombach: *Goal question metric (gqm) approach*. Encyclopedia of software engineering, 2002. 46

[49] Pressman, Roger S: *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005. 47

[50] JMeter, Apache: *Apache software foundation*, 2017. 49

Apêndice A

Catálogo de serviço do OAuth 2.0

Código A.1: Catálogo de Serviço para o OAuth 2.0.

```
[
 {
        "name" : "/authorize",
        "comment": "OAuth 2.0",
        "owner": "oauth2ems",
        "version": "1",
        "service" : "oauth2ems_authorize:execute",
        "url": "/authorize",
        "type": "GET",
        "result_cache" : 0,
        "authorization" : "public",
        "querystring": [
                {
                        "name": "response_type",
                        "type": "string",
                        "default" : "0",
                        "comment": "The expected response"
                },
                {
                        "name": "client_id",
                        "type": "string",
                        "comment": "The client identifier"
                },
                {
                        "name": "username",
```

```
                        "type": "string",
                        "comment": "The resource owner username"
                },
                {
                        "name": "password",
                        "type": "string",
                        "comment": "The resource owner password"
                },
                {
                        "name": "redirect_uri",
                        "type": "string",
                        "comment": "Redirect URI"
                },
                {
                        "name": "scope",
                        "type": "string",
                        "comment": "The scope of the access req."
                },
                {
                        "name": "secret",
                        "type": "string",
                        "comment": "The client secret"
                },
                {
                        "name": "code",
                        "type": "string",
                        "comment": " The authorization code "
                },
                {
                        "name": "state",
                        "type": "string",
                        "comment": "The state"
                }
        ],
        "lang" : "erlang"
},
```

```
{
        "name" : "/authorize",
        "comment": "Oauth 2.0",
        "owner": "oauth2ems",
        "version": "1",
        "service" : "oauth2ems_authorize:execute",
        "url": "/authorize",
        "type": "POST",
        "authorization" : "public",
        "querystring": [
                {
                        "name": "grant_type",
                        "type": "string",
                        "default" : "0",
                        "comment": "The grant type"
                },
                {
                        "name": "client_id",
                        "type": "string",
                        "comment": "The client identifier"
                },
        {
                        "name": "client_secret",
                        "type": "string",
                        "comment": "The client secret"
                },
                {
                        "name": "secret",
                        "type": "string",
                        "comment": "The client secret"
                },
                {
                        "name": "username",
                        "type": "string",
                        "comment": "The resource owner username"
                },
                {
```

```
                        "name": "password",
                        "type": "string",
                        "comment": "The resource owner password"
                },
                {
                        "name": "redirect_uri",
                        "type": "string",
                        "comment": "The redirect URI"
                },
                {
                        "name": "scope",
                        "type": "string",
                        "comment": "The scope of the access req."
                },
                {
                        "name": "code",
                        "type": "string",
                        "comment": "The authorization code"
                },
                {
                        "name": "state",
                        "type": "string",
                        "comment": "The state"
                        },
                {
                        "name": "refresh_token",
                        "type": "string",
                        "comment": "The refresh token"
                }
        ],
        "lang" : "erlang"
},

{
        "name" : "/code_request",
        "comment": "Oauth 2.0",
        "owner": "oauth2ems",
```

```
"version": "1",
"service" : "oauth2ems_authorize:code_request",
"url": "/code_request",
"type": "GET",
"result_cache" : 0,
"authorization" : "public",
"querystring": [
        {
                "name": "client_id",
                "type": "string",
                "comment": "The client identifier"
        },
        {
                "name": "username",
                "type": "string",
                "comment": "The resource owner username"
        },
        {
                "name": "password",
                "type": "string",
                "comment": "The resource owner password"
        },
        {
                "name": "redirect_uri",
                "type": "string",
                "comment": "The redirect URI"
        },
        {
                "name": "scope",
                "type": "string",
                "comment": "The scope of the access req."
        },
        {
                "name": "state",
                "type": "string",
                "comment": "The state"
        }
```

```
        ],
        "lang" : "erlang"
  }
]
```

Apêndice B

Códigos do OAuth 2.0 no ErlangMS

```
-module(oauth2ems_authorize).

-export([execute/1]).
-export([code_request/1]).
-export([implicit_token_request/1]).

-include("include/ems_config.hrl").
-include("include/ems_schema.hrl").



execute(Request = #request{type = Type, protocol_bin = Protocol, port = Port, host = ↩
    Host}) ->
  TypeAuth = case Type of
    "GET" ->  ems_request:get_querystring(<<"response_type">>, <<>>, Request);
    "POST" -> ems_request:get_querystring(<<"grant_type">>, <<>>, Request)
  end,
    Result = case TypeAuth of
      <<"password">> -> password_grant(Request);
      <<"client_credentials">> -> client_credentials_grant(Request);
      <<"token">> -> authorization_request(Request);
      <<"code">> -> authorization_request(Request);
      <<"authorization_code">> ->   access_token_request(Request,TypeAuth);
      <<"mac">> ->     access_token_request(Request,TypeAuth);
      <<"refresh_token">> ->  refresh_token_request(Request);
       _ -> {error, invalid_oauth2_grant}
  end,
  case Result of
    {ok, ResponseData} ->   ok(Request, ResponseData);
    {redirect, ClientId, RedirectUri} ->
      LocationPath = iolist_to_binary([Protocol,<<"://"/utf8>>, Host, <<":"/utf8>>,↩
          list_to_binary(integer_to_list(Port)),<<"/login/index.html?response_type=code↩
          &client_id=">>, ClientId, <<"&redirect_uri=">>, RedirectUri]),
      redirect(Request, LocationPath);
    Error ->  bad(Request, Error)

  end.
```

```
%% Requisita o codigo de autorizacao - secoes 4.1.1 e 4.1.2 do RFC 6749.
%% URL de teste: GET http://127.0.0.1:2301/authorize?response_type=code2&client_id=↩
    s6BhdRkqt3&state=xyz%20&redirect_uri=http%3A%2F%2Flocalhost%3A2301%2Fportal%2Findex↩
    .html&username=johndoe&password=A3ddj3w
code_request(Request = #request{authorization = Authorization}) ->
    ClientId     = ems_request:get_querystring(<<"client_id">>, [],Request),
    RedirectUri = ems_request:get_querystring(<<"redirect_uri">>, [],Request),
    State       = ems_request:get_querystring(<<"state">>, [],Request),
    Scope       = ems_request:get_querystring(<<"scope">>, [],Request),
    case ems_http_util:parse_basic_authorization_header(Authorization) of
    {ok, User, Passwd} ->
        Authz = oauth2:authorize_code_request({User,list_to_binary(Passwd)}, ClientId, ↩
            RedirectUri, Scope, []),
      case issue_code(Authz) of
        {ok, Response} ->
          Code = element(2,lists:nth(1,Response)),
          LocationPath = <<RedirectUri/binary,"?code=", Code/binary,"&state=",State/↩
              binary>>,
          redirect(Request, LocationPath);
        _ ->
          LocationPath = <<RedirectUri/binary,"?error=access_denied&state=",State/↩
              binary>>,
          redirect(Request, LocationPath)
        end;

    _ ->
      Username = ems_request:get_querystring(<<"username">>, <<>>, Request),
      Password = ems_request:get_querystring(<<"password">>, <<>>, Request),
        Authz = oauth2:authorize_code_request({Username,Password}, ClientId, ↩
            RedirectUri, Scope, []),
      case issue_code(Authz) of
        {ok, Response} ->
          Code = element(2,lists:nth(1,Response)),
          Location = <<RedirectUri/binary,"?code=", Code/binary,"&state=",State/binary↩
              >>,
          redirect(Request, Location);
        _ ->
          LocationPath = <<RedirectUri/binary,"?error=access_denied&state=",State/↩
              binary>>,
          redirect(Request, LocationPath)
        end
    end.
implicit_token_request(Request = #request{authorization = Authorization}) ->
    ClientId     = ems_request:get_querystring(<<"client_id">>, [],Request),
    RedirectUri = ems_request:get_querystring(<<"redirect_uri">>, [],Request),
    State       = ems_request:get_querystring(<<"state">>, [],Request),
    Scope       = ems_request:get_querystring(<<"scope">>, [],Request),
    case ems_http_util:parse_basic_authorization_header(Authorization) of
    {ok, Username, Password} ->
        Authz = oauth2:authorize_code_request({Username,list_to_binary(Password)}, ↩
            ClientId, RedirectUri, Scope, []),
      case issue_token(Authz) of
        {ok, Response} ->
          Token = element(2,lists:nth(1,Response)),
          Ttl = element(4,lists:nth(1,Response)),
          Type = element(10,lists:nth(1,Response)),
```

```
            LocationPath = <<RedirectUri/binary,"?token=", Token/binary,"&state=",State/↩
                binary,"&token_type=",Type/binary,"&expires_in=",Ttl/binary>>,
            redirect(Request, LocationPath);
          _ ->
            LocationPath = <<RedirectUri/binary,"?error=access_denied&state=",State/↩
                binary>>,
            redirect(Request, LocationPath)
          end;

      _ ->
        LocationPath = <<RedirectUri/binary,"?error=access_denied&state=",State/binary>>,
        redirect(Request, LocationPath)
      end.



%%%==========================================================================
%%% Funcoes internas
%%%==========================================================================


%% Cliente Credencial Grant- secao 4.4.1 do RFC 6749.
%% URL de teste: POST http://127.0.0.1:2301/authorize?grant_type=client_credentials&↩
    client_id=s6BhdRkqt3&secret=qwer
client_credentials_grant(Request = #request{authorization = Authorization}) ->
  ClientId = ems_request:get_querystring(<<"client_id">>, <<>>, Request),
  Scope = ems_request:get_querystring(<<"scope">>, <<>>, Request),
  % O ClientId tambem pode ser passado via header Authorization
  case ClientId == <<>> of
    true ->
      case Authorization =/= undefined of
        true ->
          case ems_http_util:parse_basic_authorization_header(Authorization) of
            {ok, Login, Password} ->
              ClientId2 = list_to_binary(Login),
              Secret = list_to_binary(Password),
              Auth = oauth2:authorize_client_credentials({ClientId2, Secret}, Scope, []↩
                  ),
              issue_token(Auth);
            Error -> Error
          end;
        false -> {error, einvalid_client_credentials}
      end;
    false ->
      Secret = ems_request:get_querystring(<<"client_secret">>, <<>>, Request),
      Auth = oauth2:authorize_client_credentials({ClientId, Secret}, Scope, []),
      issue_token(Auth)
  end.

%% Resource Owner Password Credentials Grant - secao 4.3.1 do RFC 6749.
%% URL de teste: POST http://127.0.0.1:2301/authorize?grant_type=password&username=↩
    johndoe&password=A3ddj3w
password_grant(Request) ->
  Username = ems_request:get_querystring(<<"username">>, <<>>, Request),
  Password = ems_request:get_querystring(<<"password">>, <<>>, Request),
  Scope = ems_request:get_querystring(<<"scope">>, <<>>, Request),
  Authorization = oauth2:authorize_password({Username,Password}, Scope, []),
```

```
  issue_token(Authorization).

%% Verifica a URI do Cliente e redireciona para a pagina de autorizacao - Implicit ↩
    Grant e Authorization Code Grant
%% URL de teste: GET http://127.0.0.1:2301/authorize?response_type=code&client_id=↩
    s6BhdRkqt3&state=xyz%20&redirect_uri=http%3A%2F%2Flocalhost%3A2301%2Fportal%2Findex↩
    .html


authorization_request(Request) ->
    ClientId    = ems_request:get_querystring(<<"client_id">>, <<>>, Request),
    RedirectUri = ems_request:get_querystring(<<"redirect_uri">>, <<>>, Request),
    Resposta = case oauth2ems_backend:verify_redirection_uri(ClientId, RedirectUri, [])↩
        of
    {ok,_} ->  {redirect, ClientId, RedirectUri};
    Error ->  Error
  end,
    Resposta.

%% Requisita o codigo de autorizacao - secoes 4.1.1 e 4.1.2 do RFC 6749.
%% URL de teste: GET http://127.0.0.1:2301/authorize?response_type=code2&client_id=↩
    s6BhdRkqt3&state=xyz%20&redirect_uri=http%3A%2F%2Flocalhost%3A2301%2Fportal%2Findex↩
    .html&username=johndoe&password=A3ddj3w


refresh_token_request(Request) ->
    ClientId    = ems_request:get_querystring(<<"client_id">>, [],Request),
    ClientSecret = ems_request:get_querystring(<<"client_secret">>, [],Request),
  Reflesh_token = ems_request:get_querystring(<<"refresh_token">>, [],Request),
  Scope    = ems_request:get_querystring(<<"scope">>, [],Request),
  Authorization = oauth2ems_backend:authorize_refresh_token({ClientId, ClientSecret},↩
      Reflesh_token,Scope),
    issue_token(Authorization).

%% Requisita o token de acesso com o codigo de autorizacao - secoes 4.1.3. e 4.1.4 do↩
    RFC 6749.
%% URL de teste: POST http://127.0.0.1:2301/authorize?grant_type=authorization_code&↩
    client_id=s6BhdRkqt3&state=xyz%20&redirect_uri=http%3A%2F%2Flocalhost%3A2301%2↩
    Fportal%2Findex.html&username=johndoe&password=A3ddj3w&secret=qwer&code=↩
    dxUlCWj2JYxnGp59nthGfXFFtn3hJTqx
access_token_request(Request = #request{authorization = Authorization},TypeAuth) ->
  Code = ems_request:get_querystring(<<"code">>, [],Request),
  ClientId    = ems_request:get_querystring(<<"client_id">>, [],Request),
    RedirectUri = ems_request:get_querystring(<<"redirect_uri">>, [],Request),
    ClientSecret = ems_request:get_querystring(<<"client_secret">>, [],Request),
    case ClientSecret == <<>> of
    true ->
      case Authorization =/= undefined of
        true ->
          case ems_http_util:parse_basic_authorization_header(Authorization) of
            {ok, Login, Password} ->
              ClientId2 = list_to_binary(Login),
              Secret = list_to_binary(Password),
              Auth = oauth2:authorize_code_grant({ClientId2, Secret}, Code, RedirectUri↩
                  , []),
              case TypeAuth of
                <<"mac">> -> issue_mac_token(ClientId2,Secret);
```

```
                <<"authorization_code">> -> issue_token_and_refresh(Auth)
              end;
            _Error -> {error, invalid_request}
          end;
        false -> {error, einvalid_request}
      end;
    false ->
    Authz = oauth2:authorize_code_grant({ClientId, ClientSecret}, Code, RedirectUri, []↩
        ),
    case TypeAuth of
      <<"mac">> -> issue_mac_token(ClientId,ClientSecret);
      <<"authorization_code">> -> issue_token_and_refresh(Authz)
    end
  end.


issue_token({ok, {_, Auth}}) ->
  {ok, {_, Response}} = oauth2:issue_token(Auth, []),
  {ok, oauth2_response:to_proplist(Response)};
issue_token(Error) ->
    Error.


issue_token_and_refresh({ok, {_, Auth}}) ->
  {ok, {_, Response}} = oauth2:issue_token_and_refresh(Auth, []),
  {ok, oauth2_response:to_proplist(Response)};
issue_token_and_refresh(Error) ->
    Error.

issue_code({ok, {_, Auth}}) ->
  {ok, {_, Response}} = oauth2:issue_code(Auth, []),
  {ok, oauth2_response:to_proplist(Response)};
issue_code(Error) ->
    Error.

issue_mac_token(ClientID,Secret) ->
  Consumer = {ClientID,Secret,<<"HMAC-SHA1">>},
  Token  = oauth2_token:generate(<<>>),
  TokenSecret  = oauth2_token:generate(<<>>),
  ems_oauth1:issue_token(Token,TokenSecret,Consumer),
  {ok,<<"oauth_token=",Token/binary,"&oauth_token_secret=",TokenSecret/binary>>}.
issue_mac_token(Error) ->
    Error.

ok(Request, Body) when is_list(Body) ->
  {ok, Request#request{code = 200,
    response_data = ems_schema:prop_list_to_json(Body),
    content_type = <<"application/json;charset=UTF-8">>}
  };
ok(Request, Body) ->
  {ok, Request#request{code = 200,
    response_data = Body,
    content_type = <<"application/json;charset=UTF-8">>}
  }.

bad(Request, Reason) ->
  ResponseData = ems_schema:to_json(Reason),
```

```
  {ok, Request#request{code = 401,
    response_data = ResponseData}
  }.
redirect(Request, LocationPath) ->
  % mudar code para 302
  {ok, Request#request{code = 302,
     response_data = <<"{}">>,
     response_header = #{
          <<"location">> => LocationPath
          }
    }
  }.
```

Código B.1: Arquivo *oauth2ems_authorize.erl*.

```
-module(oauth2ems_backend).
-behavior(oauth2_backend).

-include("../../include/ems_config.hrl").
-include("../../include/ems_schema.hrl").

-export([start/0, stop/0]).
-export([authenticate_user/2]).
-export([authenticate_client/2]).
-export([authorize_refresh_token/3]).
-export([get_client_identity/2]).
-export([associate_access_code/3]).
-export([associate_refresh_token/3]).
-export([associate_access_token/3]).
-export([resolve_access_code/2]).
-export([resolve_refresh_token/2]).
-export([resolve_access_token/2]).
-export([revoke_access_code/2]).
-export([revoke_access_token/2]).
-export([revoke_refresh_token/2]).
-export([get_redirection_uri/2]).
-export([verify_redirection_uri/3]).
-export([verify_client_scope/3]).
-export([verify_resowner_scope/3]).
-export([verify_scope/3]).

-define(ACCESS_TOKEN_TABLE, access_tokens).
-define(ACCESS_CODE_TABLE, access_codes).
-define(REFRESH_TOKEN_TABLE, refresh_tokens).
-define(SCOPE_TABLE, scopes).


-define(TABLES, [?ACCESS_TOKEN_TABLE,
          ?ACCESS_CODE_TABLE,
                    ?REFRESH_TOKEN_TABLE]).

-record(a, { client    = undefined     :: undefined | term()
           , resowner = undefined     :: undefined | term()
           , scope                    :: oauth2:scope()
           , ttl       = 0            :: non_neg_integer()
```

```
            }).
start() ->
    application:set_env(oauth2, backend, oauth2ems_backend),
    lists:foreach(fun(Table) ->
                          ets:new(Table, [named_table, public])
                  end,
                  ?TABLES).

stop() ->
    lists:foreach(fun ets:delete/1, ?TABLES).



%%%===================================================================
%%% OAuth2 backend functions
%%%===================================================================

authenticate_user({Login, Password}, _) ->
    case ems_user:authenticate_login_password(Login, Password) of
    ok -> {ok, {<<>>,Login}};
    %% Padronizar o erro conforme o RFC 6749
    _ -> {error, unauthorized_user}
  end.
authenticate_client({ClientId, Secret},_) ->
    case ems_client:find_by_codigo_and_secret(ClientId,Secret) of
      {ok, Client} -> {ok, {<<>>,Client}};
      _ -> {error, unauthorized_client}
    end.


get_client_identity(ClientId, _) ->
    case ems_client:find_by_codigo(ClientId) of
        {ok, Client} -> {ok, {[],Client}};
        _ -> {error, unauthorized_client}
    end.


associate_access_code(AccessCode, Context, _AppContext) ->
    {put(?ACCESS_CODE_TABLE, AccessCode, Context), Context}.

associate_refresh_token(RefreshToken, Context, _) ->
    {put(?REFRESH_TOKEN_TABLE, RefreshToken, Context), Context}.

associate_access_token(AccessToken, Context, _) ->
    {put(?ACCESS_TOKEN_TABLE, AccessToken, Context), Context}.

resolve_access_code(AccessCode, _) ->
  case get(?ACCESS_CODE_TABLE, AccessCode) of
        {ok,Value} ->    {ok,{[],Value}};
        _Error -> {error, invalid_code}
    end.

resolve_refresh_token(RefreshToken, _AppContext) ->
    case get(?REFRESH_TOKEN_TABLE, RefreshToken) of
       {ok,Value} -> {ok,{[],Value}};
        _Error -> {error, invalid_token}
    end.
```

```
resolve_access_token(AccessToken, _) ->
    case get(?ACCESS_TOKEN_TABLE, AccessToken) of
       {ok,Value} -> {ok,{[],Value}};
        _Error -> {error, invalid_token}
    end.

revoke_access_code(AccessCode, _AppContext) ->
    delete(?ACCESS_CODE_TABLE, AccessCode),
    {ok, []}.

revoke_access_token(AccessToken, _) ->
    delete(?ACCESS_TOKEN_TABLE, AccessToken),
    {ok, []}.

revoke_refresh_token(_RefreshToken, _) ->
    {ok, []}.

get_redirection_uri(ClientId, _) ->
    case get_client_identity(ClientId,[]) of
        {ok, #client{redirect_uri = RedirectUri}} ->
            {ok, RedirectUri};
        _ -> {error, einvalid_uri}
    end.

verify_redirection_uri(ClientId, ClientUri, _) when is_binary(ClientId) ->
    case get_client_identity(ClientId,[]) of
        {ok,{_, #client{redirect_uri = RedirUri}}} ->
      case ClientUri =:= RedirUri of
        true -> {ok,[]};
        _ -> {error, unauthorized_client}
      end;
        Error -> Error
    end;

verify_redirection_uri(#client{redirect_uri = RedirUri}, ClientUri, _) ->
    case ClientUri =:= RedirUri of
    true -> {ok,[]};
    _Error -> {error, error_uri}
    end.


verify_client_scope(#client{codigo = ClientID},Scope, _) ->
  case ems_client:find_by_codigo(ClientID) of
        {ok, #client{scope = Scope0}} ->
      case Scope =:= Scope0 of
        true -> {ok, {[],Scope0}};
        _ -> {error, unauthorized_client}
      end;
        _ -> {error, invalid_scope}
    end.
verify_resowner_scope(_ResOwner, Scope, _) ->
    {ok, {[],Scope}}.

verify_scope(_RegScope, Scope , _) ->
    {ok, {[],Scope}}.
% funcao criada pois a biblioteca OAuth2 nao trata refresh_tokens
```

```
authorize_refresh_token(Client, RefreshToken, Scope) ->
    case authenticate_client(Client, []) of
        {error, _}      -> {error, invalid_client};
        {ok, {_, C}} ->
      case resolve_refresh_token(RefreshToken, []) of
        {error, _} = E          -> E;
        {ok, {_, GrantCtx}} ->
          case verify_client_scope(C, Scope, []) of
            {error, _}          -> {error, invalid_scope};
            {ok, {Ctx3, _}} ->
              {ok, {Ctx3, #a{ client   =C
                , resowner= get_(GrantCtx,<<"resource_owner">>)
                , scope    = get_(GrantCtx, <<"scope">>)
                , ttl      = oauth2_config:expiry_time(password_credentials)
              }}}
          end
            end
    end.



%%%===================================================================
%%% Funcoes internas
%%%===================================================================

get(Table, Key) ->
    case ets:lookup(Table, Key) of
        [] ->
            {error, notfound};
        [{_Key, Value}] ->
            {ok, Value}
    end.
get(O, K, _)  ->
    case lists:keyfind(K, 1, O) of
        {K, V} -> {ok, V};
        false  -> {error, notfound}
    end.

get_(O, K) ->
    {ok, V} = get(O, K, []),
    V.


put(Table, Key, Value) ->
    ets:insert(Table, {Key, Value}),
    ok.

delete(Table, Key) ->
    ets:delete(Table, Key).
```

Código B.2: Arquivo *oauth2ems_backend.erl*.