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

Implementação de um Protocolo Criptográfico Baseado em REST em Haskell

Alexandre Lucchesi Alencar

Brasília
2014

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

Implementação de um Protocolo Criptográfico Baseado em REST em Haskell

Alexandre Lucchesi Alencar

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Engenharia de Computação

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Coorientador
Prof. MSc. João José Costa Gondim

Brasília
2014

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Engenharia de Computação

Coordenador: Prof. Dr. Ricardo Zelenovsky

Banca examinadora composta por:

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador) — CIC/UnB
Thiago Mael de Castro — CDS/Exército Brasileiro
Prof. Dr. Robson de Oliveira Albuquerque — ENE/UnB

CIP — Catalogação Internacional na Publicação

Alencar, Alexandre Lucchesi.

Implementação de um Protocolo Criptográfico Baseado em REST em Haskell / Alexandre Lucchesi Alencar. Brasília : UnB, 2014.
85 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2014.

1. REST, 2. protocolos criptográficos, 3. Haskell, 4. sistema de tipos de Haskell

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

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

Implementação de um Protocolo Criptográfico Baseado em REST em Haskell

Alexandre Lucchesi Alencar

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Engenharia de Computação

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Thiago Mael de Castro
CDS/Exército Brasileiro

Prof. Dr. Robson de Oliveira Albuquerque
ENE/UnB

Prof. Dr. Ricardo Zelenovsky
Coordenador do Bacharelado em Engenharia de Computação

Brasília, 08 de julho de 2014

Dedicatória

Aos meus pais, que são os grandes responsáveis pelo cumprimento de mais essa etapa.

Não existem palavras ou gestos capazes de retribuir todo o carinho, amor e apoio que eu recebi durante todos os momentos de minha vida. Mesmo assim, eu dedico de todo o coração este trabalho a vocês, como uma forma mínima de reconhecer todo o esforço e a vida que vocês dedicaram a mim e aos meus irmãos.

Mais do que meus pais, vocês são um exemplo de força, determinação e humildade, valores os quais eu levarei comigo para o resto de minha vida. Eu tenho muita sorte de ter nascido na melhor família desse mundo e eu espero ser, um dia, metade das pessoas que vocês são.

Amo vocês!

Agradecimentos

Primeiramente, a Deus, meus pais, irmãos e amigos, por todo o apoio e incentivo dados durante esses cinco anos de UnB. Sem vocês nada disso seria possível ou teria valor.

Ao meu orientador, Prof. Rodrigo Bonifácio, pela paciência e confiança depositada em mim durante toda a minha vida acadêmica. Mais do que um orientador, eu te tenho como um amigo, que esteve presente durante os momentos mais importantes da minha graduação, sempre contribuindo com ótimos conselhos. Não existem palavras para agradecer todas as oportunidades e todas as portas que você abriu para mim, que contribuíram de forma ímpar para o meu crescimento pessoal e profissional.

Ao meu coorientador, Prof. João Gondim, pela dedicação e atenção dadas a este trabalho. Ao Rogério Alves, pelas interações e discussões produtivas. Aos bons professores que tive durante essa jornada, cujas contribuições, mesmo que indiretas, foram fundamentais para a execução deste trabalho.

À Ingrid, pessoa muito especial que Deus colocou em minha vida antes de toda essa jornada começar. Você nunca deixou de me incentivar e acreditar no meu sucesso, mesmo nos momentos mais infortúnios. Sempre esteve presente com palavras de carinho, força e descontração. Você fez parte de todos esses anos de UnB e já é parte da minha vida. Eu sou muito feliz por te ter ao meu lado. Te amo!

Por fim, aos inventores do café e da Neosaldina, compostos sem os quais este trabalho não seria possível.

Obrigado!

Resumo

Protocolos de segurança baseados em REST, como o OAuth e o OpenID, foram propostos para permitir mecanismos de *autenticação* e *autorização* distribuída em *ambientes abertos*, cujos usuários em potencial são geralmente desconhecidos com antecedência. No entanto, as instituições possuem também a necessidade de se comunicar umas com as outras em ambientes fechados, de modo que elas possam integrar automaticamente os seus processos de negócio. Neste trabalho, relata-se o uso de Haskell, uma linguagem de programação puramente funcional e estaticamente tipada, para a implementação de um protocolo criptográfico baseado em REST para ambientes fechados. Apesar do desenvolvimento experimental de Haskell, que dificulta o uso pragmático da linguagem (principalmente devido a inconsistências de bibliotecas e versões do compilador), os resultados sugerem que Haskell é uma opção interessante para o desenvolvimento de aplicações baseadas em REST em geral (mas, em especial, para o desenvolvimento de protocolos criptográficos), levando a uma baixa utilização de recursos e tempo de inicialização, pequeno esforço de desenvolvimento, alto nível de modularidade através de *typeclasses*, e maior segurança, devido ao sistema de tipos da linguagem.

Palavras-chave: REST, protocolos criptográficos, Haskell, sistema de tipos de Haskell

Abstract

REST-based security protocols, such as OAuth and OpenID, have been proposed to allow a form of distributed *authentication* and *authorization* mechanism target to *open environments*, whose potential users are usually unknown in advance. However, there also exists the need for institutions to communicate to each other within closed environments — so that they could automatically integrate their business processes. In this work, we report on the use of Haskell, a statically typed, purely functional programming language, to implement a REST-based cryptographic protocol for closed environments. Apart from the experimental development of Haskell, which hinders the pragmatic use of the language (mostly due to libraries and compiler versions' inconsistencies), our findings suggest that Haskell is a compelling choice for developing REST-based applications in general (but in particular for the development of cryptographic protocols), leading to a small footprint and initialisation time, small development effort, extensive modularity through typeclasses, and improved safety thanks to the language's type system.

Keywords: REST, cryptographic protocols, Haskell, Haskell type system

Sumário

1 Introdução **1**

1.1 Definição do Problema . . . 2

1.2 Objetivos . . . 2

1.3 Organização do Trabalho . . . 3

2 Segurança da Informação **4**

2.1 Criptografia . . . 5

2.1.1 Cifras Simétricas . . . 6

2.1.2 Cifras Assimétricas . . . 8

2.1.3 Envelope Digital . . . 10

2.2 Protocolos Criptográficos . . . 10

2.3 Problema da Distribuição das Chaves . . . 12

2.4 Funções de *Hash* Criptográfico . . . 13

2.4.1 Códigos de Autenticação de Mensagens . . . 14

3 Arquitetura Orientada a Serviços **15**

3.1 Terminologia e Conceitos . . . 15

3.2 Benefícios da Arquitetura Orientada a Serviços . . . 16

3.2.1 Maior Interoperabilidade Intrínseca . . . 16

3.2.2 Maior Federação . . . 16

3.2.3 Mais Opções de Diversificação de Fornecedores . . . 16

3.2.4 Maior Alinhamento do Domínio de Negócio e de Tecnologia . . . 16

3.2.5 Maior Retorno sobre o Investimento . . . 17

3.2.6 Maior Agilidade Organizacional . . . 17

3.2.7 Menor Carga de Trabalho da TI . . . 17

3.3 Estratégias de Implementação . . . 17

3.3.1 Web Services . . . 17

3.3.2 REST . . . 18

4 Programação Funcional em Haskell **21**

4.1 Sistema de Tipos . . . 21

4.2 Conceitos Básicos . . . 22

4.2.1 Tipos Sinônimos . . . 22

4.2.2 Tipos de Dados Algébricos . . . 23

4.2.3 *Typeclasses* . . . 25

4.2.4 Casamento de Padrões . . . 26

4.2.5 Recursividade . . . 28

4.2.6 Funções de Alta Ordem 28
4.3 *Functional Design-Patterns* 31
4.3.1 Functors 32
4.3.2 Applicative Functors 33
4.3.3 Monads 36

5 Solução **38**
5.1 Requisitos do Protocolo 38
5.2 *Design* em Alto Nível 39
5.2.1 Arquitetura do Protocolo 39
5.3 Funcionamento do Protocolo 42
5.3.1 Primeiro Cenário 43
5.3.2 Segundo Cenário 46
5.4 *Design* em Baixo Nível 46
5.4.1 Tecnologias Utilizadas 47
5.4.2 Biblioteca Criptográfica: `jwt-min` 48
5.4.3 Algoritmo de Assinatura 49
5.4.4 Algoritmo de Cifragem 51
5.4.5 Servidores 54
5.4.6 Banco de Dados: Apache CouchDB 66
5.4.7 Módulo Cliente: `rest-client` 69
5.5 Análise de Desempenho 71

6 Conclusão e Trabalhos Futuros **72**

Referências **73**

Lista de Figuras

2.1 Modelo de comunicação utilizando criptografia [39]. 5
2.2 Esquema de criptografia simétrica [39]. 6
2.3 Esquema de criptografia assimétrica [39]. 8
2.4 Comparação entre algoritmo e protocolo [38]. 11
2.5 Funções unidirecionais. Adaptado de [19]. 13

5.1 Arquitetura do sistema. 40
5.2 Fluxo do protocolo de autenticação e autorização proposto. 43
5.3 Estrutura de um *token* JWT. 48
5.4 Arquitetura dos servidores. 54

Lista de Tabelas

3.1 Restrições REST. 19

5.1 Códigos HTTP no protocolo. 55
5.2 Matriz comparativa dos *frameworks* Web Haskell. Adaptado de [41]. 56

Capítulo 1

Introdução

A Computação Orientada a Serviços (SOC) surgiu como uma tecnologia proeminente para a implementação de processos de negócios que envolvem diferentes instituições. No entanto, incorporar o fator *segurança* nesses processos de negócio ainda é visto como um desafio para muitas instituições, como é o caso, por exemplo, da Polícia Civil do Distrito Federal (PCDF) que, no que diz respeito a sua competência de Polícia Judiciária, possui atribuições que tangenciam em vários pontos as atribuições de diferentes órgãos ou instituições conveniadas, tais como: Ministério Público da União (MPU), do Tribunal de Justiça do Distrito Federal e Territórios (TJDFT), Departamento de Trânsito do Distrito Federal (DETRAN-DF), Secretaria de Segurança Pública do Distrito Federal (SSP-DF), Secretárias de Justiça do DF e estados, entre outros.

Devido a criticidade e a sensibilidade das informações que são trocadas entre essas instituições, falhas de segurança nos mecanismos de integração podem ser exploradas para causar diversos danos, como comprometer de forma significativa uma investigação policial ou gerar injustiças. Se existir uma vulnerabilidade, por exemplo, entre a integração dos sistemas da PCDF e do TJDFT, uma pessoa inocente pode ser condenada injustamente à prisão. As necessidades de segurança de uma organização podem ser decompostas em uma série de objetivos específicos. Alguns desses objetivos podem ser alcançados através de processos de segurança e do uso adequado de criptografia, como a obtenção de sigilo, integridade de dados ou autenticidade.

A fim de apoiar o processo de integração entre a PCDF e seus órgãos ou instituições conveniadas, foi *especificado* um protocolo criptográfico baseado em REST para arquitetura orientada a serviços (SOA) [6]. Outros protocolos de segurança baseados em REST, como o OAuth e o OpenID, foram propostos para permitir mecanismos de *autorização e autenticação* distribuída em *ambientes abertos*, cujos usuários em potencial são geralmente desconhecidos com antecedência. Por outro lado, o protocolo proposto tem como alvo *ambientes fechados*, que são contextos controlados de utilização, onde as partes envolvidas devem, primeiro, estabelecer contratos formais de cooperação, antes que os serviços prestados pela entidade servidora possam ser acessados.

Desta forma, este trabalho descreve uma *implementação* em *software*, utilizando a linguagem de programação Haskell, do protocolo especificado por [6]. É importante notar que protocolos criptográficos são, geralmente, implementados utilizando linguagens imperativas, principalmente porque eles executam várias computações impuras (tais como operações de entrada e saída e geração de números aleatórios). Apesar disso, escolheu-se

Haskell, uma linguagem de programação declarativa e puramente funcional [21, 27], para a implementação do protótipo. A motivação para a utilização de Haskell é baseada em experiências anteriores com a linguagem e à convicção de que algumas de suas características (como o sistema de tipos forte, a concisão e o isolamento de efeitos colaterais) podem contribuir para o rápido desenvolvimento e para o cumprimento dos principais requisitos do protocolo.

Este trabalho possui três contribuições principais. Em primeiro lugar, descreve-se a implementação de um protocolo criptográfico RESTful usando Haskell (Seção 5.4). Tanto quanto se sabe, não existem fontes que descrevam o uso de uma linguagem de programação puramente funcional nesse nicho de aplicação. Em segundo lugar, com base nas lições aprendidas durante o projeto e o desenvolvimento do protótipo, descreve-se alguns pontos fortes e limitações de Haskell para esse domínio específico. Uma das principais conclusões é que, mesmo em aplicações que são compostas em sua maioria por computações impuras, é possível se beneficiar do sistema de tipos forte e de recursos de modularidade de Haskell (como as *typeclasses*). No entanto, alguns problemas relacionados com a pragmática da linguagem, como modificações constantes nas bibliotecas e a falta de compatibilidade entre diferentes versões da plataforma Haskell, podem diminuir a produtividade. Em terceiro lugar, tem-se uma implementação em *software* de um sistema em rede que atende aos requisitos de segurança e desempenho da PCDF e que pode ser generalizado e facilmente implantado em contextos similares.

1.1 Definição do Problema

Como forma de mitigar os problemas encontrados na PCDF durante o compartilhamento de serviços com órgãos ou instituições parceiras, foi proposto um protocolo de referência baseado em arquitetura orientada a serviços com foco em segurança [6]. O principal objetivo desse protocolo é incorporar o fator *segurança* como característica transversal às aplicações, centralizando as políticas de segurança em uma interface comum e retirando essa responsabilidade do desenvolvedor. Contudo, atualmente não existe uma implementação em *software* desse protocolo, que valide o seu funcionamento e que possa atuar, de fato, na integração dos processos de negócios da PCDF e seus órgãos ou instituições parceiras.

1.2 Objetivos

Este trabalho tem como objetivo o desenvolvimento de um protótipo, provendo uma implementação eficiente do protocolo proposto por [6], a partir do estudo, avaliação e aplicação de técnicas, ferramentas e procedimentos que garantam os requisitos de segurança e escalabilidade necessários para a integração dos sistemas e automatização dos processos entre a PCDF e seus órgãos ou instituições parceiras, e que possa ser facilmente implantado em contextos similares.

Os objetivos específicos são:

- Validar o protocolo proposto por meio do desenvolvimento de um protótipo funcional em *software*.

- Avaliar técnicas, ferramentas e procedimentos que garantam os requisitos de segurança e escalabilidade necessários para a integração dos sistemas e automatização dos processos entre órgãos ou instituições parceiras.

- Avaliar a utilização de programação funcional, mais especificamente da linguagem Haskell, no nicho de aplicação em questão, que é predominantemente imperativo, comparando *frameworks* e APIs existentes.

- Generalizar a arquitetura proposta, provendo uma implementação eficiente que dê suporte à integração de aplicações em *ambientes fechados*, onde uma entidade deseja prover serviços implementados em diferentes tecnologias através da Internet a parceiros conhecidos de forma segura e escalável.

1.3 Organização do Trabalho

Este trabalho está organizado em seis capítulos. O Capítulo 2 apresenta aspectos gerais referentes à segurança da informação, os principais objetivos de segurança, e como muitos desses objetivos podem ser alcançadas através da utilização adequada de criptografia. O Capítulo 3 apresenta uma fundamentação teórica e os principais conceitos relacionados à arquitetura orientada a serviços (SOA), seus principais benefícios e as tecnologias comumente utilizadas para implementações em SOA. O Capítulo 4 busca introduzir as principais características de Haskell, fundamentais para o entendimento e compreensão dos trechos de código apresentados nos capítulos posteriores. O Capítulo 5 apresenta o protótipo desenvolvido, descrevendo os requisitos do protocolo, uma visão de alto nível acerca de seu funcionamento, os detalhes de implementação e uma análise de desempenho. Por fim, o Capítulo 6 apresenta as conclusões finais, uma síntese das principais contribuições deste trabalho e os trabalhos futuros.

Capítulo 2

Segurança da Informação

A segurança da informação pode ser definida como um conjunto de ações que são executadas com a finalidade de prover segurança às informações de indivíduos e organizações [6]. Atualmente a segurança é um requisito importante para qualquer aplicação distribuída, tais como aplicações governamentais, aplicações de segurança pública e de defesa, dentre outras [4].

A segurança da informação se manifesta de várias formas, de acordo com a situação, os requisitos e os interesses dos agentes envolvidos em uma comunicação, que devem ter confiança de que os objetivos associados com a segurança serão atingidos [35]. As necessidades de segurança de uma organização podem ser decompostas em inúmeros objetivos específicos que, combinados, formam o objetivo geral de segurança do negócio.

A *privacidade* ou *confidencialidade*, por exemplo, é um desses objetivos específicos, que visa manter a informação sigilosa de todos que não estejam autorizados a visualizá-la. Já a *integridade de dados* busca garantir que a informação não foi alterada de forma desconhecida ou não-autorizada. A *autenticação*, *identificação da entidade* ou *integridade de origem* procura comprovar a identidade de uma entidade (uma pessoa, um terminal de computador, um cartão de crédito, entre outros). A *autenticação da mensagem* ou *integridade dos dados* visa comprovar a origem dos dados. A *assinatura* representa uma forma de associar a informação a uma entidade, enquanto a *autorização* visa a concessão, à outra entidade, de permissão para fazer ou ser algo. O *controle de acesso*, por sua vez, atua restringindo o acesso a recursos a apenas entidades privilegiadas, e assim por diante.

No mundo real, onde os dados são representados em um suporte físico material, como o papel, esses objetivos podem ser obtidos a partir de um conjunto de técnicas e protocolos que foram desenvolvidos ao longo dos séculos. No entanto, no mundo digital, os dados são codificados por meio de sequências de *bits* que são transportadas por sinais elétricos, magnéticos, sonoros, dentre outros. Dessa forma, os aspectos de segurança da informação no "mundo dos *bits*" consistem na aplicação de técnicas que garantam os objetivos desejados independentemente do meio físico que armazena ou transporta os dados, fazendo com que eles sejam intrínsecos aos próprios dados.

No caso da PCDF, alguns desses objetivos são particularmente importantes dada a criticidade e a sensibilidade das informações que são tratadas. O sigilo e a autenticidade são importantes porque o vazamento de informações a entidades não autorizadas pode comprometer de forma significativa uma investigação policial. Já a integridade dos dados é importante para se evitar interpretações equivocadas ou injustiças. O controle de acesso

e a validação temporal são úteis para conceder às instituições parceiras acesso aos serviços providos pela PCDF, e assim por diante.

Alcançar a segurança da informação no universo eletrônico requer um vasto conjunto de competências técnicas e legais. Não há, no entanto, garantias de que todos os objetivos de segurança da informação necessários possam ser atendidos. O meio técnico para se obter tais objetivos advém do uso adequado de *criptografia* [35].

2.1 Criptografia

A criptografia pode ser definida como o estudo de técnicas matemáticas relacionadas aos aspectos de segurança da informação, tais como confidencialidade, integridade dos dados, autenticação de entidade e autenticação de origem dos dados [35]. É a arte de se implementar cifras robustas, que inviabilizam um atacante obter a mensagem somente a partir do texto cifrado. Vale frisar que a criptografia é capaz de oferecer mecanismos de proteção às mensagens somente durante a transmissão (no canal), entre as etapas de codificação e decodificação.

A literatura usualmente exemplifica o desafio da comunicação utilizando as entidades Alice, Bob e Oscar. Por padrão, Alice e Bob desejam se comunicar através de um canal inseguro de modo que um adversário, Oscar, não possa compreender o conteúdo da mensagem entre os dois primeiros. Este canal pode ser uma linha telefônica, uma rede de computador ou até mesmo o mecanismo de correspondência postal [42].

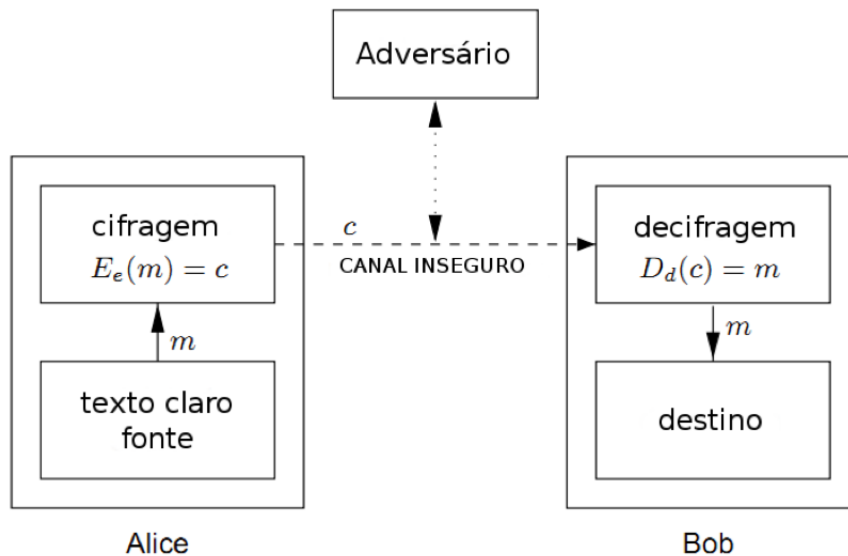

Figura 2.1: Modelo de comunicação utilizando criptografia [39].

A Figura 2.1 exemplifica um modelo simplificado de comunicação entre duas entidades utilizando criptografia com o objetivo de agregar sigilo. O modelo pressupõe uma etapa inicial de troca de chaves, que deve ser realizada por Alice e Bob através de um canal seguro. Essa troca de chaves consiste na escolha de um par de chaves (e, d) que servem, respectivamente, para cifrar e decifrar as mensagens. Dessa forma, quando Alice deseja

enviar uma mensagem m para Bob, ela calcula o criptograma $c = E_e(m)$ e envia para Bob. Ao receber o criptograma, Bob faz o processo inverso, que consiste no cálculo do texto em claro: $m = D_d(c)$ e, assim, ter acesso à mensagem original [35].

Os algoritmos criptográficos são comumente classificados pelo tipo de cifra que implementam. Existem, basicamente, três tipos: algoritmos restritos, algoritmos simétricos e algoritmos assimétricos. Um algoritmo restrito é um algoritmo que implementa cifra secreta, indo de encontro ao Princípio de Kerkhoff [29], ou seja, presume-se ser desconhecido de quem o ataca, fazendo com que a robustez da cifra dependa do sigilo do algoritmo (e das chaves, se existirem) [40]. A Seção 2.1.1 apresenta os algoritmos simétricos, a Seção 2.1.2 apresenta os algoritmos assimétricos e a Seção 2.1.3 apresenta como os dois tipos podem ser combinados para a criação de cifras robustas.

2.1.1 Cifras Simétricas

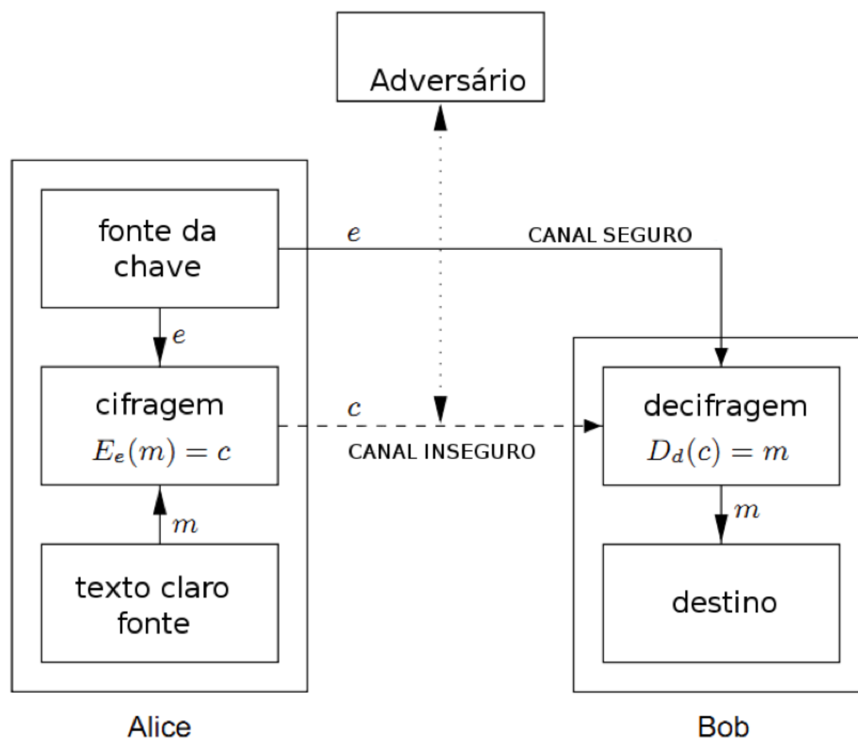

Figura 2.2: Esquema de criptografia simétrica [39].

Um algoritmo simétrico é um algoritmo que implementa cifra de chaves secretas. É projetado para que a robustez da cifra não dependa do sigilo do algoritmo f e para que uma das chaves possa ser facilmente obtida a partir da outra. Nesse caso, um par de funções (e, e^{-1}) é geralmente indexado por uma chave k (dita secreta), sendo e utilizada para cifrar e e^{-1} para decifrar as mensagens. A robustez de f depende do sigilo de k, da equiprobabilidade da escolha de k no espaço de chaves K, e de K ser grande [38, 40].

Entretanto, na maioria desses algoritmos, as chaves de cifragem e decifragem são as mesmas. Nesse caso, são denominados algoritmos de chave secreta ou de chave única e

requerem que os principais compartilhem uma chave em um momento anterior à comunicação [40].

A eficiência de sistemas criptográficos simétricos está associada ao sigilo da chave compartilhada. Se ocorrer o vazamento da chave, um terceiro escutando o canal estará apto a cifrar e decifrar mensagens.

A Figura 2.2 ilustra uma comunicação utilizando criptografia simétrica. Em relação à Figura 2.1, observa-se o aparecimento de um *canal seguro* por meio do qual a chave simétrica deve ser compartilhada entre o emissor e o receptor antes do início das transmissões. Esse canal, denominado *canal de confiança* ou *canal lateral*, deve ser diferente do canal que se quer proteger, sendo necessário para a transferência de material criptográfico sigiloso e para garantir a eficiência do algoritmo [37].

Essa etapa inicial da comunicação que consiste na troca de chaves é um dos maiores desafios desse tipo de sistema, descrito como o *Problema da Distribuição de Chaves* [33], apresentado na Seção 2.3.

Vantagens

As principais vantagens de sistemas criptográficos simétricos podem ser sintetizadas como sendo: alto desempenho, baixo consumo de banda, capacidade composicional e maturidade [35, 39].

No que tange ao desempenho, cifras de chave simétrica podem ser desenvolvidas para atingir altas taxas de rendimento ou processamento. Algumas implementações em *hardware* podem atingir taxas de cifragem que chegam à centenas de *megabytes* por segundo, enquanto a implementação em *software* atinge taxas de processamento de *megabytes* por segundo.

As chaves dos sistemas criptográficos simétricos costumam ser relativamente pequenas em comparação com as chaves assimétricas e esses sistemas podem, ainda, serem combinados para geração de cifras mais fortes. Para isso, utiliza-se produtos de cifra e iterações com transformações simples para construir as cifras resultantes, que continuam simétricas.

Por fim, a criptografia de chave simétrica tem uma extensa história. Assim, acredita-se que seus aspectos de segurança já foram amplamente explorados.

Desvantagens

As principais desvantagens de sistemas criptográficos simétricos podem ser sintetizadas como sendo: aumento das premissas de sigilo das chaves para o uso eficaz dos mecanismos de proteção e baixa escalabilidade em uma grande rede [35, 39].

Em um sistema de comunicação entre duas partes utilizando criptografia simétrica, a chave deve permanecer em sigilo pelas duas entidades, diferentemente da utilização de criptografia assimétrica, onde cada agente é responsável por manter apenas a sua chave privada em sigilo, e um possível comprometimento da chave privada de um dos agentes não gera impactos sobre o outro.

Além disso, a boa prática criptográfica determina que, nesse tipo de comunicação, as chaves sejam trocadas frequentemente e, em alguns casos, trocadas a cada sessão de comunicação, gerando uma certa complexidade.

Em uma grande rede, onde existem vários pares de chaves para serem gerenciados, necessita-se, ainda, de métodos eficientes de gerência de chaves e o uso de terceiros de confiança.

Por fim, mecanismos de assinatura digital que estão surgindo através de criptografia de chave simétrica, usualmente, requerem chaves muito grandes para funções de verificação pública ou o uso de terceiros de confiança.

2.1.2 Cifras Assimétricas

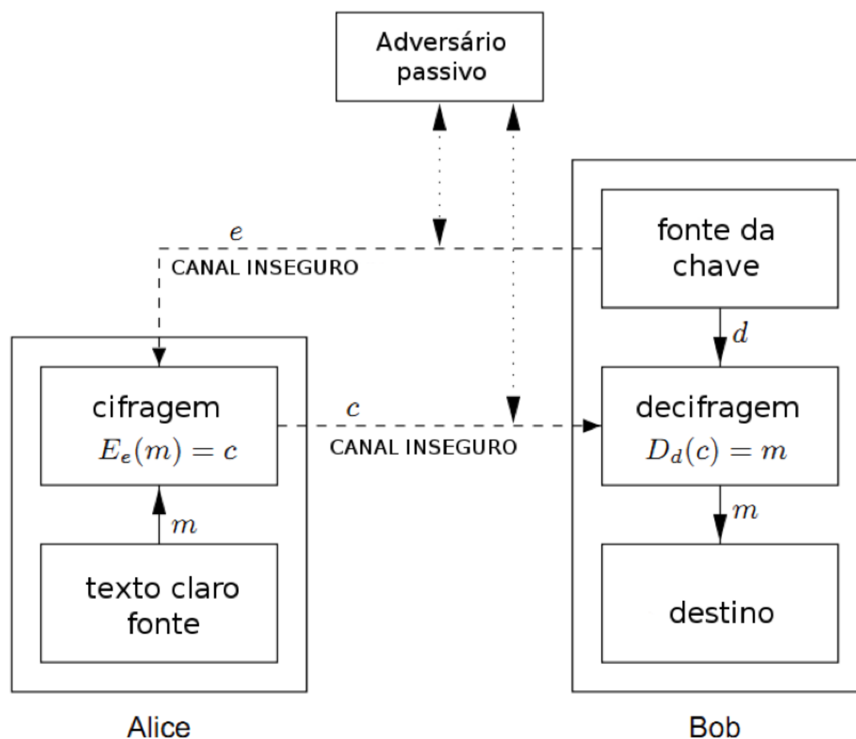

Figura 2.3: Esquema de criptografia assimétrica [39].

Um algoritmo assimétrico é um algoritmo que implementa cifra de chaves públicas. É projetado para que a robustez da cifra não dependa do sigilo nem do algoritmo f e nem da chave que cifra e. Nesse caso, para qualquer par de chaves (e, d), o custo de dedução da chave privada d a partir da chave pública e ou do texto cifrado c tem que ser inviável. A robustez de f depende do sigilo de d, da equiprobabilidade da escolha de (e, d) no espaço de chaves K, e de K ser grande [38].

A Figura 2.3 ilustra um esquema de comunicação baseado em criptografia assimétrica. Bob seleciona um par de chaves (e, d), envia a chave (pública) de cifragem e para Alice e mantém a chave (privada) de decifragem d em segredo. Dessa forma, Alice pode se comunicar de forma segura com Bob, cifrando as mensagens enviadas com a chave pública de Bob, isto é, para enviar uma mensagem m, Alice calcula $c = E_e(m)$. Ao receber o criptograma c enviado por Alice, Bob utiliza sua chave privada d para obter a mensagem

original, realizando a operação inversa D_d. É importante frisar que o bom funcionamento desse esquema pressupõe a autenticidade da chave pública recebida por Alice.

Observa-se na Figura 2.3 que, diferentemente do sistema simétrico apresentado na Figura 2.2, não existe um "canal seguro" para distribuição de material criptográfico. A chave que Alice utiliza para cifrar mensagens para Bob é recebida através de um canal inseguro, podendo ser, inclusive, o mesmo que será utilizado para transmissão dos criptogramas. Por isso a chave e de Bob é denominada *pública*, porque ela não precisa ser mantida em segredo. A implicação direta deste fato é que qualquer entidade portando a chave pública de Bob poderá enviar mensagens cifradas que apenas ele, o portador da chave privada correspondente, poderá decifrar.

A criptografia de chave pública assume o conhecimento e autenticidade da chave pública e considera que a derivação da chave privada a partir da chave pública é um problema intratável, ou seja, assume a existência de funções unidirecionais, e autenticação prévia de chaves públicas [39].

Vantagens

As principais vantagens de sistemas criptográficos assimétricos podem ser sintetizadas como sendo: redução das premissas de sigilo das chaves para o uso eficaz dos mecanismos de proteção, alta escalabilidade (podendo requerer a utilização de terceiros de confiança), baixa rotatividade das chaves e a existência de mecanismos eficientes de assinatura digital [35, 39].

Em um sistema criptográfico assimétrico, apenas as chaves privadas dos agentes deve ser mantida em segredo. Entretanto, a verificação de autenticidade das chaves públicas é necessária. Essa verificação pode ser feita, por exemplo, por meio de mecanismos de *certificação digital*. É válido ressaltar, porém, que a certificação digital não é *necessária* para habilitar o funcionamento de esquemas que utilizam criptografia assimétrica, e sim, *útil* para tornar esses esquemas escaláveis.

A administração de chaves em uma rede escalável requer a presença de um terceiro de confiança funcionalmente confiável ao invés de um incondicionalmente confiável. Dependendo do modo de operação, o terceiro de confiança pode ser exigido apenas de forma *offline*, entrando em cena apenas para decidir quando houver conflitos de interesses.

Dependendo do modo de operação, o par de chaves (pública e privada) pode permanecer sem modificação por um grande período de tempo, isto é, sendo usado durante várias sessões e com validade de anos, desde que seja preservado o sigilo da chave privada.

Muitos esquemas de chave pública fornecem eficientes mecanismos de assinatura digital. A chave usada para descrever a função de verificação pública é geralmente muito menor do que a de criptografia simétrica para fins equivalentes.

Finalmente, em uma grande rede, o número de chaves necessárias é consideravelmente menor do que no cenário da criptografia simétrica.

Desvantagens

As principais desvantagens de sistemas criptográficos assimétricos podem ser sintetizadas como sendo: desempenho geral inferior, maior consumo de banda e baixa maturidade, quando comparados a sistemas simétricos [35, 39].

As taxas de processamento para os métodos criptográficos de chave pública são menores do que o melhor esquema de chave simétrica. Além disso, os tamanhos das chaves são geralmente maiores do que os necessários para a criptografia simétrica e o tamanho da assinatura de chave pública é maior, fornecendo autenticação de origem de dados por técnicas de chave simétrica.

Até o momento, nenhum método de chave pública se provou seguro. O método de criptografia assimétrica mais difundido atualmente tem sua segurança baseada na dificuldade de um pequeno conjunto de problemas em Teoria dos Números, chamado de problema da fatoração de inteiros.

Por fim, criptografia de chave pública tem uma história ainda recente em comparação com a criptografia simétrica. Dessa forma, acredita-se que houve pouca exploração com respeito a sua segurança.

2.1.3 Envelope Digital

São cifras definidas por meio da combinação de cifras simétricas e assimétricas. Têm como objetivo agregar as vantagens desses dois tipos de cifras na criação de esquemas balanceados em termos de desempenho e distribuição de chaves.

Envelopes digitais são comumente utilizados da seguinte maneira: o emissor gera uma chave simétrica k e cifra uma mensagem m utilizando um algoritmo criptográfico simétrico, obtendo o criptograma $c = k(m)$. Em seguida, o emissor cifra a chave simétrica k utilizando um algoritmo assimétrico e a chave pública e do receptor, obtendo a chave cifrada $k' = e(k)$. Então, o emissor transmite para o receptor o criptograma e a chave cifrada (c, k'). Ao receber a mensagem, o emissor utiliza sua chave privada d para decifrar k', obtendo k. De posse de k, o emissor decifra simetricamente o criptograma c, obtendo a mensagem original.

As principais vantagens desse esquema são: criptografia de chave pública é a melhor forma de se distribuir material criptográfico prévio, necessário à comunicação, através de um canal inseguro e constitue, até então, a única forma de se realizar assinaturas digitais. Além disso, a criptografia de chave simétrica é muito eficiente para cifrar, decifrar e garantir a integridade de grandes volumes de dados. Algoritmos de chave pública são lentos. Calcula-se que algoritmos simétricos são, em média, no mínimo 10^3 vezes mais rápidos do que algoritmos assimétricos [40].

Por esses motivos, para o desenvolvimento do protótipo foi utilizado o esquema de envelope digital para proteção das mensagens transmitidas no protocolo. Mais detalhes sobre os algoritmos utilizados e o formato das mensagens são apresentados na Seção 5.4.2.

2.2 Protocolos Criptográficos

Um *protocolo* pode ser definido como uma série de passos, envolvendo dois ou mais participantes, que visa alcançar um objetivo. "Uma série de passos" significa que um protocolo tem uma sequência a ser seguida, do começo ao fim, e que nenhuma etapa pode ser executada antes da etapa anterior. "Envolvendo dois ou mais participantes" significa que pelo menos dois participantes são necessários para completar o protocolo. Por fim, "que visa alcançar um objetivo" significa que o protocolo deve apresentar um resultado [40].

Pode-se pensar em um protocolo como sendo um algoritmo distribuído onde a execução dos passos é *alternada* entre dois ou mais executores, conforme ilustrado na Figura 2.4. Para o funcionamento adequado, um protocolo deve ser não ambíguo e completo, tendo todas as etapas bem definidas e sem espaços para interpretações equivocadas, e devendo existir uma ação especificada para cada situação possível. Além disso, todos os envolvidos no protocolo devem conhecer todas as etapas previamente e concordar em segui-las [40].

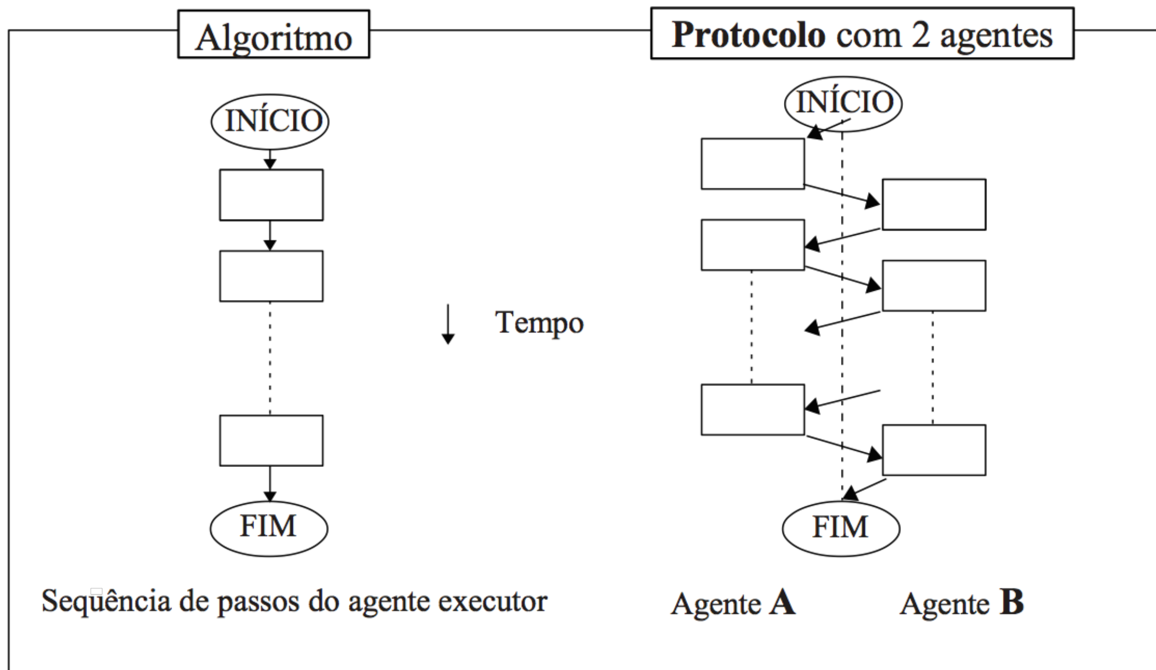

Figura 2.4: Comparação entre algoritmo e protocolo [38].

Além de formalizar um comportamento, protocolos abstraem o processo de se concluir uma tarefa, isto é, um protocolo de comunicação pode ser o mesmo independentemente da plataforma onde está implementado. Assim, é possível avaliar o protocolo sem entrar nos detalhes de implementação.

Um *protocolo criptográfico* tem como propósito proteger interesses dos agentes em processos de comunicação. A criptografia é utilizada para manipular pontos, modos e momentos em que alguma confiança é presumida no processo. Para isso, um protocolo criptográfico pode buscar atingir um ou mais dos seguintes objetivos [38]:

- Transferir sigilo ou detectar adulteração.
- Prevenir vazamentos que o vulnerem.
- Prevenir ou detectar trapaças ou desavenças.
- Prevenir perigosas inferências ou conluios.

Conforme seu propósito e premissas, um protocolo pode especificar a ação de um agente auxiliar, denominado *terceiro de confiança* (TTP), desinteressado, à priori, no

propósito, para resolver impasses entre os agentes principais (interessados no propósito). Os protocolos que utilizam um TTP podem ser classificados em dois tipos: *arbitrado* e *ajuizável*, a depender do tipo de interação com o TTP. Se o TTP precisar interagir com os agentes durante a execução do protocolo, o protocolo é dito arbitrado e o TTP trabalha *online*. Caso contrário, se a participação do TTP se restringir para momentos em que ocorrem conflitos de interesses entre os agentes, sendo necessária uma decisão por parte do terceiro, o protocolo é dito ajuizável e o TTP trabalha *offline*.

O protocolo especificado por [6] para a PCDF não conta com a participação de um TTP e, portanto, é dito *autoverificável*. Esse tipo de protocolo supõe que a execução elimina possíveis vantagens para trapaças ou desavenças entre os agentes, que seriam, nesse caso, a PCDF e os órgãos ou instituições parceiras credenciadas para acessar os serviços providos.

2.3 Problema da Distribuição das Chaves

O Problema da Distribuição das Chaves, mencionado na Seção 2.1.1, consiste na dificuldade que os agentes participando de uma comunicação encontram para trocar material criptográfico que será utilizado para proteger as mensagens durante as transmissões. Esse problema ocorre porque as chaves utilizadas para cifrar as mensagens não devem trafegar pelo mesmo canal onde trafegam os criptogramas, isto é, o canal inseguro que se quer proteger. Elas devem trafegar por meio de um canal alternativo e confiável, antes do início da comunicação. Para isso, os agentes podem se encontrar pessoalmente para realizar a troca de chaves, ou ela pode se dá por meio de uma rede dedicada de banda inferior, entre outros. Esse canal confiável é denominado *canal lateral* ou *canal de confiança* [38].

Em um algoritmo que implementa cifra simétrica, se a chave simétrica for transmitida por um canal inseguro e for interceptada por um terceiro malicioso escutando o canal, ele poderá decifrar toda a comunicação entre os agentes e, pior, poderá se passar por um dos agentes, utilizando a chave para enviar mensagens cifradas.

Se o algoritmo for assimétrico, não se tem o problema de sigilo, uma vez que a chave pública pode trafegar às claras. No entanto, tem-se o problema de autenticidade, pois, o receptor precisará confiar que o detentor da chave privada correspondente à chave pública que ele possui é, realmente, a entidade esperada, ficando sujeito a ataques de *man-in-the-middle*.

Para mitigar esses problemas no caso assimétrico, pode-se utilizar um esquema de certificação digital, que é um esquema de autenticação objetiva recursiva por meio do qual é possível autenticar, através de uma hierarquia de entidades certificadoras (CAs), um certificado digital que, basicamente, serve para associar uma chave pública a uma identidade. É válido ressaltar que um certificado digital não garante que uma pessoa é quem ela diz ser, mas sim, para atestar que uma pessoa é quem ela diz ser no certificado. Essa nuância é muito importante e remete ao fato de que certificados digitais podem ser gerados por meios alheios às CAs e que, mesmo com os certificados emitidos por CAs, podem existir falhas de integridade devido a fraudes ou falta de abrangência na política de certificação da CA.

Dito isso, o protocolo especificado por [6] não sofre do problema da distribuição das chaves, pois, existe um canal de confiança seguro, fora de banda, para a troca de chaves na inicialização do protocolo. O órgão ou instituição parceira interessada em acessar os

serviços da PCDF deve, previamente, se dirigir à sede da PCDF para assinar um termo de uso e um contrato formal de cooperação entre as partes. A troca de chaves é, então, realizada, de forma presencial, entre os representantes das instituições.

Mesmo com a existência desse canal de confiança, optou-se no protocolo pela utilização de criptografia assimétrica [1]. Isso se justifica porque a utilização de uma estrutura de chaves públicas alivia as premissas de sigilo das chaves no servidor de autenticação e autorização, que só precisa manter a *integridade* (e não integridade e sigilo) de uma lista de chaves públicas.

2.4 Funções de *Hash* Criptográfico

Funções de *hash* criptográfico são fundamentais para a criptografia moderna e são utilizadas como primitivas para construção de muitos protocolos [35]. No entanto, para compreendê-las é necessário o entendimento prévio de dois conceitos chave: *funções unidirecionais* e *funções de hash*:

Uma função unidirecional é uma função que é relativamente fácil de se computar, porém extremamente difícil de se inverter, ou seja, a partir de um argumento x, é viável o cálculo da imagem $f(x)$, mas a partir de $f(x)$ é inviável o cálculo de x [40]. A Figura 2.5 ilustra essa definição.

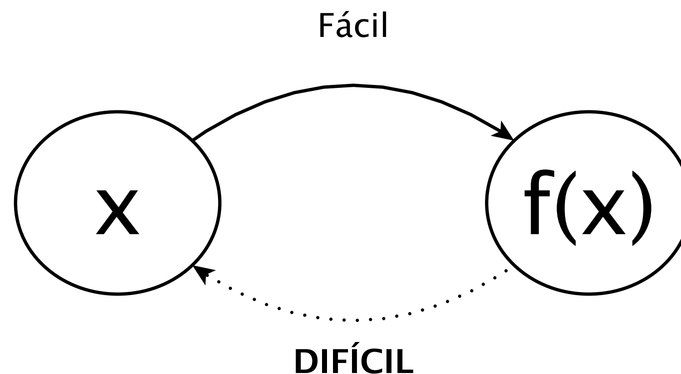

Figura 2.5: Funções unidirecionais. Adaptado de [19].

Matematicamente, não há provas de que funções unidirecionais existam e nem de que seja possível construí-las — afirma Schneier [40] citando [5, 9, 17, 20]. Contudo, existem funções matemáticas que *parecem ser* unidirecionais, isto é, funções que não possuem, até então, técnicas eficientes de se obter suas inversas, e que podem ser computadas de forma rápida.

Por outro lado, uma função de *hash* é uma função que recebe como entrada uma *string* de tamanho variável, denominada *pré-imagem*, e a converte em uma *string* de saída de tamanho fixo, geralmente menor do que a pré-imagem, denominada *valor de hash*. A ideia é criar uma impressão digital da pré-imagem, isto é, produzir um valor que é capaz de

[1] Na verdade, utiliza-se o esquema de envelope digital apresentado na Seção 2.1.3 que, para efeitos práticos, é a aplicação de criptografia assimétrica em uma quantidade menor de dados.

determinar se uma suposta pré-imagem é provável de ser igual à original. Um exemplo simples de função de *hash* consiste em dividir a pré-imagem em uma sequência de *bytes* e aplicar a operação XOR entre eles, obtendo-se como resultado um novo *byte*.

Finalmente, uma função de *hash* criptográfico pode ser definida como uma combinação das duas definições apresentadas: é uma função com a qual é fácil de se calcular um valor de *hash* a partir de uma pré-imagem, mas com a qual é difícil de se gerar uma pré-imagem cujo cálculo do *hash* resultará em algum valor em particular.

Além disso, uma boa função de *hash* criptográfico deve ser resistente a colisões, isto é, deve ser inviável encontrar dois valores cujo cálculo do *hash* produza o mesmo valor. O exemplo de função de *hash* apresentado (cálculo do XOR) não possui essa propriedade, ou seja, dado um *byte* qualquer que represente o valor de *hash*, é trivial encontrar uma *string* de *bytes* cujo XOR seja igual ao daquele *byte*.

2.4.1 Códigos de Autenticação de Mensagens

Um Código de Autenticação de Mensagem (MAC) é uma função de *hash* unidirecional com a adição de uma chave secreta. Assim, o valor de *hash* é uma função parametrizada em termos da pré-imagem e da chave. A teoria é exatamente igual a das funções de *hash*, exceto o fato de que somente alguém que detenha a chave secreta pode verificar o valor de *hash* [40].

O propósito de um MAC é facilitar, sem o uso de mecanismos adicionais, garantias a respeito da origem e da integridade de uma mensagem [35]. Para isso, uma função de *hash* criptográfico $h()$ é usada com uma chave secreta k para validação subjetiva, entre emissor e receptor, da integridade de uma mensagem m. Para isso, o emissor calcula o valor de *hash* $h' = h(m, k)$ e o transmite junto à m, gerando $m' = (m, h')$. Ao receber m', o receptor, que conhece k, pode recalcular o valor de *hash* e compará-lo a h'. Se os valores de *hash* forem iguais, a mensagem é considerada íntegra. É importante ressaltar o sigilo da chave k, ou seja, se um terceiro também conhece k, a integridade não pode ser verificada por MAC.

Capítulo 3

Arquitetura Orientada a Serviços

A arquitetura orientada a serviços (SOA) estabelece um modelo arquitetural que visa a aprimorar a eficiência, a agilidade e a produtividade de uma empresa, posicionando os serviços como os principais meios para que a solução lógica seja representada no suporte à realização dos objetivos estratégicos associados à computação orientada a serviços.

A SOA busca prover interoperabilidade entre sistemas através de *serviços*: unidades de modularização — independentes (com baixo acoplamento), autocontidos, sem estado e de baixa granularidade (com poucas responsabilidades), que são disponibilizados e consumidos por meio de protocolos bem definidos [15, 28]. SOA objetiva o compartilhamento de serviços, a independência de plataforma e linguagem de programação, e a flexibilidade e agilidade no desenvolvimento de aplicações [8]. Além disso, segundo [28], outro objetivo de SOA é estruturar sistemas distribuídos com base nas abstrações de regras e funções de negócio.

Este capítulo busca apresentar os principais conceitos relacionados à SOA, seus principais benefícios e as tecnologias comumente utilizadas atualmente para o desenvolvimento de soluções utilizando esse padrão arquitetural.

3.1 Terminologia e Conceitos

A *orientação a serviços* é um paradigma de *design* que abrange um conjunto específico de princípios de *design*. A aplicação desses princípios ao *design* da lógica da solução resulta em uma *lógica orientada a serviços*. A unidade fundamental da lógica orientada a serviços é o *serviço*.

Os serviços existem como programas de *software* fisicamente independentes, com características de *design* distintos, que dão suporte à obtenção dos objetivos estratégicos associados à computação orientada a serviços. Cada serviço recebe seu próprio contexto funcional distinto e possui um conjunto de capacidades relacionadas a esse contexto. Essas capacidades, adequadas para o acesso por programas externos, são comumente expressas através de *contratos de serviços* públicos (como uma espécie de API).

3.2 Benefícios da Arquitetura Orientada a Serviços

A visão por trás da computação orientada a serviços é extremamente ambiciosa e, por isso, também muito atraente a qualquer organização interessada em verdadeiramente aprimorar a eficácia de sua área de TI. Esta seção busca apresentar um conjunto de objetivos e benefícios comuns que, segundo Erl [8], surgiram para formar essa visão.

3.2.1 Maior Interoperabilidade Intrínseca

Interoperabilidade refere-se a compartilhamento de dados. Quanto mais interoperáveis forem os programas de *software*, mais facilmente trocarão informações. Os programas de *software* que não são interoperáveis precisam ser integrados; portanto, a integração pode ser vista como um processo que permite a interoperabilidade. Um objetivo da orientação a serviços é estabelecer a interoperabilidade nativa dentro dos serviços, a fim de reduzir a necessidade da integração. Na realidade, a integração, como conceito, começa a desaparecer nos ambientes orientados a serviços.

3.2.2 Maior Federação

Um ambiente de TI federado é aquele em que os recursos e os aplicativos permanecem unidos e, ao mesmo tempo, mantêm a autonomia individual e a auto-governança. A SOA visa a aumentar a perspectiva federada de uma empresa, independentemente de sua aplicação. Isso se consegue por meio da implementação em larga escala de serviços padronizados e capazes de se compor, onde cada um dos quais, encapsule um segmento da empresa e o expresse de maneira consistente.

3.2.3 Mais Opções de Diversificação de Fornecedores

A diversificação de fornecedores significa a capacidade que uma organização tem de escolher inovações tecnológicas e produtos do "melhor fornecedor da categoria" e de utilizá-los conjuntamente em uma empresa. Ter e manter essa opção exige que a arquitetura da tecnologia não esteja associada ou atada à plataforma de um fornecedor específico.

Projetando uma arquitetura orientada a serviços alinhada com as principais plataformas SOA dos fornecedores, porém neutra em relação a elas, e posicionando contratos de serviços como pontos de contato padronizados em todas as partes de uma empresa federada, os detalhes da implementação de um serviço proprietário podem ser abstraídos, a fim de se estabelecer um *framework* de comunicações interserviços consistente.

3.2.4 Maior Alinhamento do Domínio de Negócio e de Tecnologia

A computação orientada a serviços traz um paradigma de *design* que promove a abstração em vários níveis. Um dos mais eficazes meios para aplicar a abstração funcional é estabelecer camadas de serviço que encapsulem e representem precisamente os modelos de negócios. Fazendo isso, as representações comuns e preexistentes da lógica do negócio podem existir na forma implementada, como serviços físicos.

Além disso, o fato de os serviços serem projetados para serem intrinsecamente interoperáveis facilita diretamente a modificação dos negócios. Como os processos de negócio

são ampliados em resposta a vários fatores, os serviços podem ser configurados em novas composições, que refletem a lógica modificada do negócio. Isso permite que uma arquitetura de tecnologia orientada a serviços evolua em consonância com o próprio negócio.

3.2.5 Maior Retorno sobre o Investimento

A computação orientada a serviços defende a criação da *lógica de solução agnóstica* — a lógica que é agnóstica a um propósito qualquer e, portanto, útil a diversos propósitos. Essa lógica de vários propósitos ou reusável tira proveito total da natureza intrinsecamente interoperável dos serviços. O potencial reúso dos serviços agnósticos aumentou e pode ser percebido, permitindo que eles sejam repetidamente montados em diferentes composições. Qualquer serviço agnóstico pode, portanto, ser adaptado inúmeras vezes para que seja possível automatizar diferentes processos de negócio como parte de diferentes soluções orientadas a serviços.

3.2.6 Maior Agilidade Organizacional

Boa parte da computação orientada a serviços visa ao estabelecimento da agilidade organizacional comum. Quando a orientação a serviços é aplicada em toda a empresa, ela resulta na criação de serviços altamente padronizados e reusáveis e, por isso, agnósticos a processos de negócios e a ambientes de aplicativos específicos. O resultado final é uma maior capacidade de resposta na entrega de projetos e um tempo menor para o mercado potencial, o que se traduz em maior agilidade organizacional.

3.2.7 Menor Carga de Trabalho da TI

Aplicar consistentemente a orientação a serviços resulta em uma empresa de TI com menor desperdício e redundância, menor tamanho e custo operacional e menos despesas indiretas associadas a governança e evolução. Essa empresa pode se beneficiar por meio da obtenção de aumentos cruciais na eficiência e na rentabilidade econômica.

3.3 Estratégias de Implementação

A SOA representa um modelo arquitetural agnóstico em relação às tecnologias de implementação. Dessa forma, uma empresa tem a liberdade de perseguir continuamente os objetivos estratégicos associados com a computação orientada a serviços, aproveitando os futuros avanços tecnológicos. Contudo, atualmente, as principais tecnologias associadas à implementação de serviços são: Web Services e REpresentational State Transfer (REST) — apresentadas nesta seção.

3.3.1 Web Services

SOA é uma forma de tecnologia arquitetural que adere aos princípios de orientação a serviços. Quando implementada por meio da plataforma tecnológica de Web services, SOA adquire o potencial de suportar e promover esses princípios nos processos de negócio e na automação dos domínios de uma empresa [7].

A plataforma de Web *services* é definida por vários padrões da indústria suportados pela comunidade de fornecedores. A plaforma original da tecnologia de Web *services* é composta das principais especificações e tecnologias a seguir: Web Services Description Language (WSDL), XML Scheme Definition Language (XSD), SOAP (Simple Object Access Protocol), UDDI (Universal Description, Discovery, and Integration) e o WS-I Basic Profile.

No entanto, outras extensões foram, posteriormente, incorporadas com o objetivo de suprir algumas brechas relacionadas às áreas de segurança no nível de mensagens, transações de serviços e troca de mensagens confiável. Essas extensões são, em geral, rotuladas prefixadas por "WS-*", como o padrão WS-Policy.

Um Web *service* é tipicamente composto de três partes: um *contrato de serviço*, a *lógica de programação* e a *lógica de processamento de mensagens*. O contrato de serviço é um conjunto de documentos, fisicamente desacoplados, consistindo de uma definição WSDL, uma definição de esquema XML e, possivelmente, uma definição WS-Policy. Esse contrato de serviço expõe funções públicas (denominadas operações) e, portanto, é similar a uma *Application Programming Interface* (API).

A lógica de programação é a implementação da lógica de negócio do serviço, podendo ser desenvolvida especificamente para o Web *service* ou consistir de uma lógica legada que será englobada por um serviço para o provimento de sua funcionalidade de acordo com os padrões de comunicação de Web Services.

A lógica do processamento de mensagens consiste em uma combinação de *parsers*, processadores e agentes de serviços que pode ser personalizada mas que, em grande parte, é provida pelo ambiente de execução.

3.3.2 REST

REST é um estilo arquitetural híbrido derivado de vários outros estilos arquiteturais baseados em rede, combinando restrições adicionais que definem uma interface de conexão uniforme. Foi desenvolvido com foco em sistemas distribuídos hipermídia para representar um modelo de como a Web moderna deveria funcionar [12].

REST provê um conjunto de restrições arquiteturais que, quando aplicadas como um todo, enfatizam a escabilidade das interações entre componentes, generalização de interfaces, implantação independente dos componentes, e componentes intermediários para reduzir a latência de interação, reforçar a segurança e encapsular sistemas legados [12]. Essas restrições arquiteturais e seus respectivos objetivos são sintetizados na Tabela 3.1.

Devido ao fato de o protocolo *HyperText Transfer Protocol* (HTTP) possuir os elementos necessários para a implementação de sistemas com o padrão arquitetural REST, é comum encontrar na literatura definições que os associam. Contudo, é importante lembrar que Fielding [11, 12] define sistemas RESTful — sistemas que aderem às restrições REST, de forma mais abstrata, desacoplada de quaisquer implementações, sendo independente de protocolo de comunicação, formato de mensagem e forma de identificação dos recursos.

Neste trabalho, REST foi utilizado como padrão arquitetural, utilizando *Uniform Resource Identifiers* (URI) e a semântica do protocolo HTTP para implementação dos serviços Web. URIs são os elementos mais simples e mais importantes da arquitetura Web. Antigamente, eram definidos como identificadores de documentos, instruindo autores a definerem identificadores em termos da localização de um documento na rede, e poderiam

Tabela 3.1: Restrições REST.

Restrição	Objetivo
Cliente-servidor	Separação de interesses, melhorando a portabilidade e escalabilidade através da simplificação dos componentes no servidor, além de permitir que os componentes evoluam de forma independente.
Sem estado (*stateless*)	Visibilidade: o servidor não precisa olhar além da requisição atual; confiabilidade: facilita a recuperação de falhas parciais; e escalabilidade: não ter que armazenar estado entre requisições simplifica e otimiza o servidor.
Cache	Eliminar algumas interações, melhorando a eficiência, escalabilidade e desempenho percebido pelo usuário, reduzindo a latência média de uma série de interações.
Interface Uniforme	Simplificar a arquitetura geral do sistema, melhorar a visibilidade das interações e tornar a evolução dos componentes independente.
Sistema em Camadas	Limitar a complexidade do sistema e promover a independência das camadas.
Código sob demanda (opcional)	Simplificar os clientes, reduzindo o número de recursos requeridos a serem pré-implementados, e melhorar a extensibilidade do sistema.

ser utilizados em conjunto com os protocolos Web para recuperar esse documento. No entanto, essa definição se mostrou insatisfatória por uma série de razões [12]:

1. A definição sugere que o autor está identificando o conteúdo transferido, o que implica que o identificador deve mudar quando o conteúdo mudar.

2. Existem muitos endereços que correspondem a serviços e não documentos.

3. Existem endereços que não correspondem a documentos em certos períodos de tempo, como quando o documento ainda não existe ou quando o endereço está sendo utilizado somente para nomear, ao invés de localizar, informações.

REST redefine URIs para identificadores de *recursos* que, diferentemente de documentos, são definidos como a semântica do que se deseja identificar, ao invés do valor correspondente a essa semântica no momento em que a referência é criada. Além disso, utiliza-se o princípio de que os identificadores devem mudar o mais raramente possível. Dessa forma, REST define os objetos que são manipulados como *representações do recurso* identificado, ao invés do recurso em si.

De forma prática, ao se projetar RESTful APIs utilizando o protocolo HTTP, é comum a criação de mapeamentos entre os principais métodos HTTP: `GET`, `POST`, `PUT` e `DELETE` — e operações de criação, recuperação, atualização e remoção (CRUD) de recursos:

- `GET`: utilizado para se obter uma representação do recurso endereçado. Geralmente, o formato utilizado é Javascript Object Notation [1] (JSON), mas nada impede a utilização de XML ou qualquer outro padrão de formatação de dados.

- `POST`: utilizado para a criação de um novo objeto. No momento da criação do objeto, um identificador (URI) deve ser atribuído, sendo retornado como resultado da operação.

- `PUT`: utilizado para atualizar um objeto. Esse método, diferentemente do `POST`, é utilizado para recursos já existentes que, portanto, já possuem um URI. No momento da atualização, caso não exista um recurso associado ao URI, o recurso deve ser criado.

- `DELETE`: utilizado para remover o recurso endereçado pelo URI.

De acordo com a especificação HTTP 1.1 [10], os métodos `GET`, `PUT` e `DELETE` devem ser *idempotentes*, isto é, os efeitos colaterais produzidos pela execução de múltiplas requisições idênticas deve ser o mesmo produzido por somente uma requisição. Em outras palavras, o resultado final dessas operações deve ser independente do número de vezes em que elas foram efetuadas. O método `POST`, por sua vez, não possui essa propriedade, e a submissão de várias requisições do tipo `POST` em um servidor pode resultar na criação de múltiplos objetos idênticos, mas identificados por URIs diferentes.

[1] `http://json.org/`.

Capítulo 4

Programação Funcional em Haskell

O protótipo desenvolvido durante a execução deste trabalho foi escrito na linguagem de programação puramente funcional Haskell. Dessa forma, este capítulo tem como objetivo introduzir as principais características de Haskell, fundamentais para o completo entendimento e compreensão dos trechos de código apresentados na Seção 5.4. Além disso, durante a apresentação dos conceitos, busca-se justificar a escolha de Haskell e familiarizar o leitor, comparando os conceitos e funcionalidades com equivalentes em linguagens imperativas.

4.1 Sistema de Tipos

Existem diferentes sistemas de tipos, cada um com características particulares e, em uma linguagem de programação, o sistema de tipos afeta significativamente a forma com que se pensa e se escreve código. O sistemas de tipos de Haskell permite que as soluções sejam modeladas em um alto nível de abstração e permite a escrita de programas de forma concisa e poderosa [36].

Todas as expressões e funções em Haskell têm um *tipo*. O valor `True`, por exemplo, tem o tipo `Bool`, enquanto o valor `"foo"` tem o tipo `String`. O tipo de um valor indica que ele compartilha certas propriedades com outros valores do mesmo tipo, ou seja, números podem ser somados e listas podem ser concatenadas. Existem três aspectos principais sobre o sistema de tipos em Haskell: ele é *fortemente tipado*, ele é *estaticamente tipado*, e ele pode *inferir automaticamente* os tipos das expressões [1].

Quando se diz que Haskell tem um sistema de tipos forte, isso significa que o sistema de tipos garante que um programa não pode conter certos tipos de erros, provenientes da escrita de expressões que não fazem sentido, como por exemplo passar uma *string* como argumento para uma função que espera um inteiro.

Um sistema de tipos forte pode tornar mais difícil a escrita de certos programas (ex: por não implementar conversões automáticas de tipos e impedir certos tipos de conversões), porém, traz o grande benefício da captura de *bugs* em tempo de compilação, antes que eles possam causar problemas [36].

Um sistema de tipos *estático* implica que o compilador verifica, em tempo de compilação, os tipos de todos os valores e expressões. Se um programa for escrito contendo

[1] Às vezes, quando mais de um tipo é possível para uma expressão, o compilador não consegue inferir automaticamente o tipo correto e é necessário informá-lo explicitamente.

expressões cujos tipos não casem, um compilador ou interpretador Haskell detectará a inconsistência e rejeitará o programa antes que ele seja executado.

Um sistema de tipos estático pode tornar difícil a escrita de alguns tipos de código úteis. Em linguagens de programação dinâmicas como Python, existe o conceito de *duck typing*, que permite que, se um tipo se comportar de forma suficientemente similar a outro tipo, ele possa se passar por este tipo. Em contrapartida, Haskell utiliza o mecanismo de *typeclasses*, apresentado na Seção 4.2.3, que provê quase todos os benefícios de sistemas de tipos dinâmicos de forma segura e conveniente [36].

O fato de Haskell ser fortemente e estaticamente tipada impede a ocorrência de erros de tipos em tempo de execução. É, inclusive, um truísmo na comunidade Haskell que, uma vez que um código passa pela checagem de tipos, ele provavelmente funcionará de forma mais correta do que em outras linguagens.

Programas escritos em linguagens dinamicamente tipadas requerem abrangentes suítes de teste para se obter alguma garantia de que simples erros de tipos não podem ocorrer. Contudo, suítes de teste não podem proporcionar uma cobertura de código completa: algumas tarefas simples, tais como refatorar um programa para torná-lo mais modular, podem introduzir novos erros de tipos que uma suíte de testes não irá detectar.

Em Haskell, o compilador garante a ausência de erros de tipo: um programa Haskell que compile não sofrerá de erros de tipos durante sua execução. Dessa forma, refatorar o código, geralmente, consiste apenas na movimentação de fragmentos de código e posterior compilação do programa (até passar pelo compilador).

Uma analogia útil para a compreensão do valor da checagem estática de tipos é entender o sistema de tipos como se ele estivesse colocando peças em um quebra-cabeças. Em Haskell, se uma peça tiver o formato errado, ela simplesmente não encaixa. Em uma linguagem dinâmica, todas as peças são iguais e sempre encaixam, fazendo com que o desenvolvedor tenha que examinar constantemente a figura resultante e checar, por meio de testes, sua corretude.

Implementações Haskell podem inferir automaticamente os tipos de quase todas as expressões em um programa. Esse processo é conhecido como *inferência de tipos*. Haskell permite a declaração explícita do tipo de qualquer valor, mas a presença do mecanismo de inferência de tipos torna essa tarefa quase sempre opcional.

4.2 Conceitos Básicos

Esta seção apresenta os conceitos básicos da linguagem Haskell, envolvendo a construção e manipulação de estruturas de dados e alguns mecanismos de abstração e generalização de código. Esses conceitos formam a base de construções mais complexas, como as que são apresentadas na Seção 4.3.

4.2.1 Tipos Sinônimos

Tipos sinônimos permitem a atribuição de um *novo nome* a um tipo já existente, geralmente um nome que facilite o entendimento e aumente a legibilidade do código no contexto em questão.

São definidos utilizando a palavra-chave `type` e seu funcionamento é análogo ao da palavra-chave `typedef` na linguagem de programação C. Suponha uma função `encrypt`,

que recebe dois parâmetros: uma chave simétrica e o texto em claro — e retorna o texto cifrado. A assinatura dessa função em Haskell poderia ser:

```
encrypt :: ByteString -> ByteString -> ByteString
```

Essa definição informa apenas que a função `encrypt` recebe dois *arrays* de *bytes* e retorna um terceiro. Somente a partir da assinatura é difícil dizer o que os parâmetros representam e mesmo que eles possam ser deduzidos a partir do contexto, continua difícil saber qual é o ordem dos parâmetros, isto é, se o primeiro parâmetro representa a chave ou o texto em claro.

Dessa forma, essa definição de `encrypt` não é legível, podendo ser melhorada através da introdução de tipos sinônimos:

```
type SymmetricKey = ByteString
type PlainText    = ByteString

encrypt' :: SymmetricKey -> PlainText -> ByteString
```

Assim, a nova definição `encrypt'` é mais clara em relação ao significado e à ordem dos parâmetros. No entanto, ela pode gerar dúvidas sobre a natureza de `SymmetricKey` e `PlainText`, ou seja, sendo tipos sinônimos, eles são apenas um novo nome para o tipo `ByteString`, mas a assinatura não diz isso, podendo gerar a interpretação equivocada de que eles são, de fato, um novo tipo.

Felizmente, esse impasse pode ser facilmente resolvido consultando-se a documentação, se existir, ou por meio de um interpretador Haskell, que permite, dentre outras, inspecionar os tipos das expressões ou funções.

4.2.2 Tipos de Dados Algébricos

A seção anterior apresentou como criar um sinônimo para um tipo já existente através de tipos sinônimos. Esta seção apresenta a criação de novos tipos, por meio dos chamados *tipos de dados algébricos* (ADTs).

ADTs são assim denominados porque são criados a partir de duas *operações algébricas*: *somas* e *produtos*. A operação "soma" representa *alternância*: `A | B`, que significa `A` ou `B`, mas não ambos. Já a operação "produto" representa *combinação*: `A B`, que significa `A` e `B` juntos [1]. O código abaixo exemplifica essa definição:

```
data Pair  = P Int Double
data Pair' = I Int | D Double
```

A primeira definição, `Pair`, representa um par de números, um inteiro (`Int`) *e* um número de ponto flutuante (`Double`), juntos. O rótulo `P` é usado para combinar os dois valores contidos em uma estrutura única. Já a segunda definição, `Pair'`, representa apenas um número: um `Int` *ou* um `Double`. Nesse caso, os rótulos `I` e `D` são usados para se distinguir entre as duas alternativas.

As operações "soma" e "produto" podem ser repetidamente combinadas para a criação de estruturas cada vez mais complexas. Dessa forma, um ADT pode ser definido como uma estrutura unificada para a descrição de tipos de dados [36].

Precisamente, um ADT é definido utilizando a palavra-chave `data` e é composto de duas partes: um *construtor de tipo* e um ou mais *construtores de valor*. Ambos os construtores podem ser parametrizados, recebendo zero ou múltiplos parâmetros. Um construtor de tipo é parametrizado durante a definição de tipos polimórficos, permitindo, por exemplo, a definição de um ADT `List` que atue como um container de quaisquer outros tipos, sejam eles inteiros, *strings*, outra `List`, e assim por diante. Já os parâmetros de um construtor de valor representam os atributos que aquele valor possui.

Considere o ADT abaixo, que representa uma versão simplificada de uma entidade *serviço*:

```
data Service = Service {
    identifier  :: Int,
    description :: String,
    host        :: String,
    port        :: Int
}
```

A palavra-chave `data` indica a criação de um novo tipo. O identificador `Service` à esquerda do `=` é o tipo (ou construtor de tipo) e não recebe nenhum parâmetro. O identificador `Service` à direita do `=` é o construtor de valor, podendo ter o mesmo nome do construtor de tipo e sendo usado para a criação de valores do tipo `Service`. Esse construtor de valor recebe quatro parâmetros, isto é, a criação de um serviço requer o fornecimento de um identificador inteiro, uma *string* representando a descrição do serviço, uma *string* representando o servidor que provê o serviço e um inteiro representando o número da porta em que o serviço é executado. Nesse caso, tem-se apenas um construtor de valor (`Service`).

A nomeação dos campos — como `identifier`, `description`, `host` e `port`, é opcional e representa uma funcionalidade chamada *record syntax*, que consiste na derivação automática de funções acessoras, que permitem acesso aos campos do ADT definidos à direita do `::`. De posse de um valor do tipo `Service`, é possível acessar, por exemplo, o inteiro que representa seu identificador utilizando-se a função `identifier`, cuja assinatura é: `identifier :: Service -> Int`.

A utilização de mais de um construtor de valor é útil quando se tem mais de uma possível representação para um tipo. Considere um ADT simplificado, que representa as possíveis requisições que um servidor consegue tratar:

```
data Request =
    Request1 {
        -- parâmetros
    } | Request2 {
        -- parâmetros
    } | Request3 {
        -- parâmetros
    } | RequestN {
        -- parâmetros
    } ...
```

Nesse caso, qualquer um dos construtores de valor apresentados: `Request1`, `Request2`, `Request3` ou `RequestN` pode ser utilizado para se criar um valor do tipo `Request`. Os parâmetros recebidos por cada um desses construtores de valor são independentes uns dos outros.

A definição de tipos enumerados utilizando ADTs consiste, simplesmente, na utilização de construtores de valor sem parâmetros. O tipo `Bool`, por exemplo, que representa valores booleanos em Haskell, é definido como:

```
data Bool = True | False
```

Essa definição atribui ao tipo `Bool` apenas dois valores possíveis: `True` ou `False`, representando, respectivamente, verdadeiro ou falso.

Os exemplos apresentados ilustraram apenas a utilização de construtores de tipo sem parâmetros. Entretanto, o grau de generalidade que se obtém da utilização de tipos parametrizados foi fundamental durante o desenvolvimento do protótipo. O código abaixo apresenta o ADT `Maybe`, que é um tipo padrão da linguagem Haskell e que foi amplamente utilizado para reportar situações de erro:

```
data Maybe a = Just a
             | Nothing
```

O construtor de tipo `Maybe` é parametrizado pela variável polimórfica `a`. Um construtor de valor recebe como parâmetro outros valores, enquanto um construtor de tipo recebe como parâmetro outros tipos. Isso significa que `Maybe` é um tipo abstrato que, para se materializar em um tipo concreto, deve receber como parâmetro um outro tipo concreto, como `Int` ou `String`. `Maybe Int`, por exemplo, é um tipo concreto que possui dois valores possíveis: `Nothing` — que representa a ausência de um valor, ou `Just v` — que representa a presença de um valor inteiro `v`.

O tipo `Maybe` pode ser entendido como o equivalente em Haskell da utilização de referências para objetos em linguagens OO, com o construtor de valor `Just` representando uma referência para um objeto que existe, e `Nothing` representando o valor `null`. Vale ressaltar que, em Haskell, a utilização desse ADT para representar a semântica de presença ou ausência de valores impossibilita a ocorrência de exceções comuns em linguagens OO, como tentar acessar uma referência nula (`NullPointerException`), uma vez que para acessar o suposto valor existente no ADT, será necessário extrai-lo antes do construtor `Just`.

4.2.3 *Typeclasses*

Typeclasses definem um conjunto de funções que podem ter diferentes implementações dependendo do tipo de dados [36]. Uma *typeclass* é uma espécie de interface que define comportamentos [32]. Se um tipo de dados é uma instância de uma *typeclass*, isso significa que esse tipo implementa o comportamento que a *typeclass* descreve.

Nesse sentido, *typeclasses* possuem certas semelhanças com *interfaces* ou *classes abstratas* presentes em linguagens orientadas a objetos (OO). Contudo, algumas diferenças são importantes: uma *typeclass* pode definir comportamentos padrão para suas funções, diferentemente das interfaces, conforme definidas, por exemplo, na linguagem Java. Além

disso, um tipo de dados pode ser instância de várias *typeclasses*, diferindo-as das classes abstratas do Java que, apesar de permitirem a definição de implementações padrão, restringem o reúso dos comportamentos por não implementar herança múltipla, fazendo com que as subclasses possam estender, no máximo, uma superclasse.

Mesmo que Java implementasse herança múltipla, como a linguagem C++, ainda haveriam diferenças em relação às *typeclasses* de Haskell. Para adicionar determinado comportamento a uma classe existente em C++ é necessário alterar sua definição, tornando-a uma subclasse da classe que define o novo comportamento. O mesmo ocorre quando se deseja fazer com que uma classe existente implemente uma interface. Isso viola o princípio de *design* que determina que classes devem ser fechadas para modificação [14, 16].

Nessas linguagens, a utilização adequada de padrões de projeto permitem contornar, até certo nível, esse problema de extensibilidade. Já a utilização de *typeclasses* permitem a definição de novas instâncias de forma extensível e modular, conferindo à Haskell extensibilidade nativa no domínio de operações [31].

É possível definir *typeclasses* customizadas mas, para exemplificar, considere a *typeclass* padrão `Eq`, que descreve os operadores (ou funções) `==` e `/=`, representando, respectivamente, as operações de teste de *igualdade* e *desigualdade* [36]:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

O valor `a` é um tipo polimórfico e significa que as funções `(==)` e `(/=)` recebem dois valores quaisquer, do mesmo tipo (sendo esse tipo uma instância de `Eq`) e retorna um booleano. Essa definição é particularmente interessante porque define uma implementação padrão para as operações de igualdade e desigualdade para todos os tipos possíveis que forem instâncias da *typeclass* `Eq`, definindo cada operação em termos da outra. Isso significa que para adicionar esses comportamentos a um determinado ADT, é suficiente definir uma das operações: `(==)` ou `(/=)`.

O código abaixo ilustra como é possível tornar um *serviço*, conforme definido na Seção 4.2.2, uma instância de `Eq`, considerando que dois serviços são iguais se todas as suas propriedades forem iguais:

```
instance Eq Service where
    (Service i1 d1 h1 p1) == (Service i2 d2 h2 p2) =
        i1 == i2 && d1 == d2 && h1 == h2 && p1 == p2
```

Por meio de casamento de padrões, os atributos dos dois serviços são extraídos e comparados. A função `(/=)` não precisa ser definida.

4.2.4 Casamento de Padrões

A seção anterior apresentou a criação de novos tipos de dados através do uso de ADTs. Esta seção apresenta como é possível manipulá-los com o mecanismo de *casamento de padrões*.

A utilização de um construtor de valor permite a criação de um valor de determinado tipo. Para operar com esse valor, é importante saber qual construtor de valor foi utilizado para construir o valor; e, se o construtor de valor utilizado possuir componentes de dados, deve ser possível extrair esses dados do valor.

Um construtor de valor é utilizado para *construir* um valor, enquanto *casamento de padrões* é utilizado para se *desconstruir*, permitindo a atribuição de nomes aos dados que o valor carregar. Com casamento de padrões, as funções são definidas como uma *série de equações*, que definem o comportamento da mesma função para diferentes padrões de entrada [36]. Considere o exemplo abaixo:

```
showNumber 0 = "Zero"
showNumber 1 = "One"
showNumber 2 = "Two"
showNumber n = "Number is not between 0 and 2. It is: " ++ show n ++ "."
```

A função `showNumber` espera um número e retorna uma *string*. Não é necessário adicionar essas informações de tipo explicitamente através de uma assinatura, pois o mecanismo de inferência de tipos, explicado na Seção 4.1, consegue deduzi-las automaticamente. As quatro equações de `showNumber` apresentadas definem a mesma função e consistem apenas em uma forma de se especificar diferentes valores de retorno para a mesma função dependendo dos parâmetros de entrada, isto é, ao se chamar `showNumber` passando-se como argumento um número entre 0 e 2, o resultado será uma *string* que representa esses números. Caso contrário, uma *string* genérica, contendo o número `n`, será exibida. O operador ++ realiza a concatenação de listas [2] e a função `show` transforma o número `n` em uma *string*.

Quando determinado dado não é relevante ou não é usado pela função, pode-se ignorá-lo através do caracter *underscore* (`_`), ou seja, se o corpo da equação `showNumber n` não utilizasse o valor `n` para construir a *string* resultante, esta equação poderia ser definida como:

```
showNumber _ = "Number is not between 0 and 2."
```

É possível utilizar casamento de padrões com qualquer ADT. Considere uma função que receba como parâmetro um *serviço*, de acordo com o ADT definido na Seção 4.2.2, e retorne o caminho completo até esse serviço, definido como a concatenação, separada pelo caracter de dois pontos (`:`), do endereço do servidor e o número da porta em que o serviço está sendo executado:

```
url :: Service -> String
url (Service _ _ h p) = h ++ ":" ++ show p
```

Presume-se na função `url` que o endereço do servidor e o número da porta sejam válidos. A função `url` realiza o casamento de padrões com o construtor de valor `Service` e associa os campos referentes ao endereço do servidor e o número da porta, respectivamente, às "variáveis" `h` e `p`. Os campos referentes ao identificador e a descrição do serviço não são usados e, portanto, são ignorados. Então, o função `url` converte o campo `p`, que é um inteiro, para uma *string*, e concatena as partes gerando o resultado da função.

[2] Existem várias representações para *strings* em Haskell. Nesse caso, uma *string* está sendo representada pelo tipo `String`, que é um tipo sinônimo para uma lista de caracteres (`[Char]`), permitindo a utilização do operador de concatenação (++).

4.2.5 Recursividade

A utilização de *recursão* é bastante importante na linguagem Haskell, porque diferentemente de linguagens imperativas, Haskell é *declarativa*, ou seja, as computações são especificadas em termos do que elas *são* ou *representam*, ao invés de *como*, de qual *maneira* ou após qual *sequência de passos* o resultado pode ser obtido. Além disso, soluções recursivas são, geralmente, mais concisas e elegantes do que suas respectivas soluções imperativas.

Em suma, recursão consiste em uma forma de definir uma função cuja a definição utiliza a própria função. Definições matemáticas são geralmente feitas de forma recusiva, como a definição da sequência de Fibonacci:

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & \text{caso contrário} \end{cases}$$

Primeiro, são definidos os dois primeiros termos da sequência de Fibonacci de forma não recursiva: $f(0)$ e $f(1)$. Em seguida, define-se que para qualquer outro número natural, o resultado da função é a soma dos dois números anteriores. Dessa forma, $f(3)$ é $f(2)+f(1)$ que, por sua vez, é $(f(1) + f(0)) + f(1)$. Tem-se agora apenas definições não-recursivas da função f e, portanto, é possível concluir que $f(3) = 2$.

Ter um elemento ou dois em uma definição recursiva definidos de forma não-recursiva, como $f(0)$ e $f(1)$, é necessário para que a recursão termine. Esses elementos constituem o denominado *caso base* da recursão.

Não existem laços de repetição em Haskell (estruturas `for`, `while` e `do...while` comumente presentes em linguagens imperativas), sendo necessário recorrer a mecanismos recursivos para a implementação de repetição. É importante frisar que o aspecto de desempenho (processos recursivos são naturalmente mais lentos do que processos iterativos), nesse caso, é tratado pelos compiladores Haskell. Dentre as várias otimizações que ele é capaz de fazer, uma é denominada *Tail Call Optimization* (TCO). A TCO consiste na tradução automática de alguns processos recursivos (aqueles em que a chamada recursiva é a última computação que a função faz) em processos iterativos no código compilado, fornecendo ao desenvolvedor um alto nível de expressividade com o mínimo de impacto no desempenho da aplicação.

4.2.6 Funções de Alta Ordem

Em Haskell, funções podem ser passadas como argumento para outras funções ou serem retornadas como valores de retorno. Qualquer função que tenha alguma dessas características, ou seja, que receba como parâmetro outra função ou retorne uma função como resultado, é denominada *função de alta ordem*. Uma das consequências disso é que funções e dados são tratados sem distinção e de forma uniforme em Haskell, sendo, inclusive, encontrado na literatura o termo *cidadãos de primeira classe* (*first-class citizens*) para se referir às funções e fazer alusão a esse fato.

Funções de alta ordem constituem um mecanismo poderoso de resolução de problemas e são imprescindíveis em uma linguagem declarativa, onde as soluções não são baseadas

na definição de uma sequência de passos e alterações em um estado mutável. O restante dessa seção apresenta algumas outras características relacionadas às funções em Haskell.

Currificação

Toda função em Haskell recebe apenas *um* parâmetro. Funções que, aparentemente, parecem receber mais de um parâmetro são, na verdade, *funções currificadas*. Isso significa que as funções podem ser *parcialmente aplicadas*, ou seja, pode-se passar para uma função uma quantidade menor de parâmetros do que ela (aparentemente) recebe e, assim, obter uma nova função que recebe os parâmetros que não foram passados. Essa nova função pode ser utilizada quantas vezes e conforme for necessário.

Para exemplificar, considere a assinatura da função `max` apresentada abaixo:

```
max :: (Ord a) => a -> a -> a
```

A função `max` é uma função padrão que recebe quaisquer dois tipos de dados que sejam instâncias da *typeclass* `Ord`, que define tipos que possam ser ordenados, e retorna o maior deles. Assim, ao chamar `max 4 5`, o resultado é o valor `5`. Contudo, internamente acontece a seguinte sequência de passos:

1. A chamada `max 4 5` primeiro cria uma função que recebe *um* parâmetro e retorna `4` ou aquele parâmetro, dependendo de qual for o maior;

2. O valor `5` é, então, aplicado à função criada no Passo 1, o que produz como resultado o próprio valor `5`.

Dessa forma, as seguintes chamadas são equivalentes: `max 4 5` e `(max 4) 5`. Espaços em branco (" ") são espécies de operadores que denotam *aplicação de função* e possuem o maior nível de precedência. Assim, a assinatura da função `max` poderia ser reescrita de forma equivalente como:

```
max :: (Ord a) => a -> (a -> a)
```

Essa assinatura pode ser lida como: `max` é uma função que recebe um `a` e retorna (significado de `->`) uma outra função que recebe um `a` e retorna um `a`.

Abstração de Padrões

É muito comum na linguagem Haskell a detecção de problemas recorrentes cuja solução é abstraída sob a forma de uma função de alta ordem. Isso estabelece, de certa forma, padrões de desenvolvimento que favorecem o desenvolvedor por dois motivos principais: a legibilidade do código é favorecida e o entendimento é facilitado por meio da utilização de um idioma comum; e a existência, em si, de soluções elegantes para situações que serão frequentemente encontradas. Esse é o caso das funções `map`, `filter` e `fold`.

Conforme explicitado em seções anteriores, em linguagens declarativas é muito comum a utilização de soluções recursivas para implementar estruturas de repetição. Uma outra forma de se implementar essas *estruturas de controle*, é por meio da utilização de estruturas de dados, tais como *listas*. Esta seção busca apresentar a função `map`, bastante utilizada no desenvolvimento do protótipo.

Considere o seguinte problema: deseja-se incrementar cada elemento da lista `[1,2,3,4, 5,6,7]`. Uma solução recursiva para o problema seria da forma:

```
increment :: (Num a) => [a] -> [a]
increment []     = []
increment (x:xs) = (x + 1) : increment xs
```

É uma solução perfeitamente aceitável, isto é, tem-se uma função que recebe uma lista de números e retorna também uma lista de números. A recursão termina quando a travessia da lista for completada, ou seja, o caso base da recursão é a lista vazia (`[]`), e a cada chamada soma-se 1 ao elemento corrente.

Considere, porém, um novo problema: deseja-se agora multiplicar por 2 cada elemento da lista. Uma possível solução para o novo problema seria:

```
double :: (Num a) => [a] -> [a]
double []     = []
double (x:xs) = (x * 2) : double xs
```

Observe que as funções `increment` e `double` são estruturalmente muito parecidas, diferindo apenas na ação a ser executada no elemento corrente da lista, ou seja, na primeira é aplicada a função "soma" (`+`), enquanto na segunda é aplicada a função "multiplicação" (`*`). Em contrapartida, tem-se como semelhanças o fato de que ambas as funções executam a travessia de uma lista, aplicando uma função a cada elemento, mas sem modificar a estrutura da lista (a quantidade de elementos não é alterada). Esse padrão foi generalizado e abstraído no que constitui a função `map`, definida como:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map xs
```

O primeiro parâmetro da função `map` é a função a ser aplicada a cada elemento da lista e o segundo é a lista. Observe que o primeiro parâmetro é uma função `(a -> b)`, o que significa que a lista retornada *pode* (mas não necessariamente, uma vez que `b` pode ser igual a `a`) conter elementos cujo tipo é diferente da lista de entrada. Dessa forma, as funções `increment` e `double` podem ser definidas, utilizando aplicação parcial, em termos de `map` como:

```
increment' :: (Num a) => [a] -> [a]
increment' = map (+ 1)

double' :: (Num a) => [a] -> [a]
double' = map (* 2)
```

Aplicação de Função com o Operador "$"

Aplicar uma função a um valor, em Haskell, é uma operação que possui precedência máxima. Essa operação pode ser realizada colocando-se espaços (" ") entre o nome da função e os seus argumentos e possui associatividade à esquerda, ou seja, aplicar uma função `f` aos valores `a`, `b` e `c` utilizando espaços: `f a b c` — é equivalente à `((f a) b) c`.

O operador *aplicação de função* (`$`) tem a mesma função do espaço — aplicar uma função a um argumento, porém, possui precedência mínima e associatividade à direita, sendo definido como:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Muitas vezes, esse operador é utilizado como uma função utilitária para reduzir a quantidade de parênteses utilizados durante a definição de expressões mais complexas, deixando o código mais limpo e mais claro.

Composição de Funções

No projeto de programas funcionais, é comum a utilização do padrão de desenvolvimento que consiste na decomposição de um problema em funções pequenas, com poucas responsabilidades, que possam ser testadas individualmente e depois combinadas para formar uma solução completa [22].

Em Haskell, uma das formas de se combinar essas funções é através do operador de *composição de funções*: `(.)`. Na matemática, a composição de funções é definida como $(f \circ g)(x) = f(g(x))$, que significa que a composição de duas funções produz uma nova função que, quando aplicada a uma entrada x, é equivalente a aplicação de x à g, seguida da aplicação do resultado à f. Esse comportamento é o mesmo em Haskell, e o operador `(.)` é definido como:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

O primeiro parâmetro é uma função `(b -> c)` e o segundo, uma função `(a -> b)`. Aplicando-se o operador `(.)` parcialmente a esses dois parâmetros, obtém-se uma função `(a -> c)`, que funciona como se a saída da segunda função fosse conectada à entrada da primeira.

4.3 *Functional Design-Patterns*

A combinação de pureza, funções de alta ordem, tipos de dados algébricos parametrizados e *typeclasses*, presentes em Haskell, permitem a implementação de poliformismo em um nível muito mais elevado do que em outras linguagens. Ao invés de modelar as soluções definindo tipos e inserindo-os em hierarquias, como frequentemente é feito em linguagens OO, em Haskell, primeiro define-se o tipo de dados e só depois pensa-se sobre os comportamentos que ele pode ter, por meio da declaração de instâncias das respectivas *typeclasses*. Esse processo é realizado de forma extremamente modular e extensível, tendo em vista o caráter *aberto* das *typeclasses* (*open-typeclasses*).

Typeclasses são abertas, ou seja, é possível definir um tipo de dados algébrico, pensar sobre quais comportamentos ele pode ter e, então, torná-lo instância das *typeclasses* que definem os comportamentos. Isso, associado ao sistema de tipos forte de Haskell, permite a definição de *typeclasses* que são gerais e abstratas, porém muito úteis no desenvolvimento de soluções concisas e elegantes, no estabelecimento de um idioma comum na linguagem e na criação de um catálogo de soluções para problemas recorrentes.

Esta Seção apresenta três padrões definidos em termos de *typeclasses*: `Functor`, `Applicative Functor` e `Monad` — cuja utilização permeia toda a implementação do protótipo.

4.3.1 Functors

A *typeclass* `Functor` é definida como:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Ou seja, `Functor` define apenas uma função: `fmap` — que admite duas interpretações distintas, porém equivalentes, sobre seu comportamento [32, 36]:

- `fmap` é uma função que recebe dois parâmetros, uma função (`a -> b`) e um tipo de dados parametrizado, `f a`, e retorna um `f b`. A partir disso, pode-se concluir que a função `fmap` aplica a função recebida no valor contido dentro do tipo de dados `f a` recebido.

- `fmap` é uma função que recebe uma função pura (`a -> b`), que representa uma função ou mapeamento entre os tipos `a` e `b`, sendo `a` e `b` tipos não-parametrizados, e a transforma em uma função mais abstrata (*lifting*), que possa ser aplicada a tipos parametrizados (que sejam instâncias de `Functor`). Essa definição se torna mais clara ao se reescrever a assinatura de `fmap`, observando-se o fato de que as funções em Haskell, conforme explicado na Seção 4.2.6, são currificadas:

  ```
  fmap :: (a -> b) -> (f a -> f b)
  ```

De forma prática, a *typeclass* `Functor` é utilizada para definir tipos de dados que podem ser mapeados, ou seja, tipicamente, suas instâncias são tipos que *contêm* outros tipos, como listas, árvores, entre outros. A metáfora de *containers de tipos* facilita, em muitos casos, a compreensão acerca do funcionamento dessas estruturas e será utilizada daqui em diante.

A Seção 4.2.6 apresentou a função `map`, utilizada para aplicar uma função em todos os elementos de uma lista sem alterá-la estruturalmente, isto é, o número de elementos foi preservado e todos eles foram retornados dentro de outra lista. É possível observar também certa semelhança entre as assinaturas de `map` e `fmap`. Isso ocorre porque a função `map` é uma especialização da função `fmap`, que é mais genérica, para listas. O tipo de dados lista (`[]`) é uma instância de `Functor`, e `map` é, na verdade, um "sinônimo" de `fmap`, sendo definida como:

```
instance Functor [] where
    fmap = map
```

Vale ressaltar que, em Haskell, tanto o construtor de tipo do "tipo lista" quanto o valor que representa uma lista vazia são representados por `[]`. Assinaturas de funções contém declarações de tipo. Na assinatura de `fmap`, `f` é um tipo abstrato, isto é, ele precisa receber um outro tipo como parâmetro para que se obtenha um tipo concreto (`f a`). O mesmo ocorre com as listas, uma vez que, toda lista precisa de um tipo para se concretizar em: uma lista de inteiros, uma lista de *strings*, uma lista de listas de *booleanos*, entre outros. Para listas, `[]` é um tipo abstrato (ou construtor de tipo), enquanto `[String]` é um tipo concreto. Por isso, `map` é `fmap` aplicada à listas, quando `f` é `[]`.

O tipo `Maybe`, apresentado na Seção 4.2.2, também pode ser interpretado como um container que, diferentemente das listas, carrega apenas um valor. Dessa forma, `Maybe` também é uma instância de `Functor`:

```
instance Functor Maybe where
    fmap f (Just v) = Just (f v)
    fmap _ Nothing  = Nothing
```

Essa definição pode ser entendida da seguinte maneira: se existir um valor (`Just v`), então a função `f` é aplicada no valor e retornada, caso contrário, independentemente de qual for a função, não há nada para ser retornado.

Uma das instâncias de `Functor` que é essencial para o entendimento de *applicative functors*, explicados na Seção 4.3.2, é o tipo `(->) r`, que representa *funções* em Haskell. Na assinatura das funções, os parâmetros são separados por `->`, e uma sequência como `r -> a` é denominada o tipo da função. Ocorre que esse tipo pode ser escrito de uma outra forma, como se `->` fosse um operador aplicado de forma pré-fixada: `(->) r a`.

Como a *typeclass* `Functor` exige tipos parametrizados com apenas um parâmetro, o tipo `(->) r a` deve ser parcialmente aplicada para ser uma instância de `Functor`, sendo definida como:

```
instance Functor ((->) r) where
    fmap f g = f . g
```

Assim, mapear uma função `f` sobre uma função `g` é o mesmo que realizar a composição dessas duas funções.

Além de ser uma instância de `Functor`, duas propriedades têm que ser verificadas para que um tipo de dados possa ser considerado um *functor*. A primeira propriedade determina que *functors* devem preservar a *identidade*, ou seja, mapear a função identidade `id` sobre um *functor* deve retornar o *functor* original: `fmap id = id`. Já a segunda propriedade determina que *functors* devem ser *composicionais*, ou seja, compor duas funções e mapear a função resultante sobre um *functor* deve ser equivalente a mapear a primeira função sobre o *functor* e, em seguida, mapear a segunda: `fmap (f . g) = fmap f . fmap g`. A linguagem Haskell não provê garantias acerca do cumprimento dessas propriedades, que devem ser validadas pelo desenvolvedor.

4.3.2 Applicative Functors

Applicative functors são *functors* aprimorados. Também são definidos na linguagem Haskell por meio de uma *typeclass*: `Applicative` — encontrada no módulo `Control.Applicative`.

Conforme explicado na Seção 4.2.6, as funções em Haskell são currificadas por padrão. Isso significa que uma função cuja assinatura é: `a -> b -> c`, na verdade, recebe apenas um argumento (`a`) e retorna uma nova função de um parâmetro (`b -> c`). Dessa forma, é possível chamar uma função como `f x y` ou `(f x) y`, permitindo que as funções sejam parcialmente aplicadas, resultando em funções que podem, por exemplo, serem passadas como argumento para outras funções.

Ao se mapear uma função de múltiplos parâmetros em um *functor*, como por exemplo utilizar `fmap` para mapear a função soma `(+)` no valor `Just 3`, obtém-se como resultado o valor `Just ((+) 3)`, isto é, uma função dentro de um construtor `Just`. Apenas, com a funcionalidade provida pela *typeclass* `Functor` (`fmap`), não é possível extrair a função `((+) 3)` e mapeá-la em outro *functor*, como `Just 5`, de forma concisa.

Applicative functors são úteis para esses casos, em que se tem um *functor* contendo uma função e outro *functor* no qual se deseja mapear a função, provendo uma forma mais geral e abstrata de se obter esse comportamento e que funcione entre diferentes *functors*. *Applicative functors* são definidos por meio da *typeclass* `Applicative`:

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Observa-se que `Applicative` restringe suas instâncias para apenas tipos que sejam instâncias de `Functor` através da declaração `Functor f` à esquerda de `=>`. Dessa forma, garante-se que todo *applicative functor* é, antes, um *functor* e, como tal, define a função `fmap` e deve obedecer as regras de preservação de identidade e composição.

A partir da assinatura da função `pure`, conclui-se que se trata de uma função que recebe um valor de qualquer tipo e retorna um *applicative functor* com o valor dentro, seguindo a analogia de containers de tipos. Outra forma de se raciocionar sobre `pure` é como uma função que recebe um valor qualquer e o coloca em algum tipo de contexto padrão (ou puro), isto é, um contexto minimalista que, caso seja requisitado, retorna aquele valor [32].

Já a função `(<*>)` possui uma assinatura similar à função `fmap`, exceto pelo primeiro parâmetro, que é `f (a -> b)`, ao invés de `(a -> b)`. Isso significa que, diferentemente de `fmap` que recebe uma função e um *functor* e aplica a função dentro do *functor*, `(<*>)` recebe um *functor* contendo uma função e outro *functor*, extrai a função do primeiro e a mapeia no segundo.

O código abaixo ilustra como o tipo `Maybe` é uma instância de `Applicative`:

```
instance Applicative Maybe where
    pure = Just
    Nothing  <*> _ = Nothing
    (Just f) <*> x = fmap f x
```

A função `pure` é definida aplicando-se parcialmente o construtor `Just`. Já a função `(<*>)` é definida da seguinte forma: se o primeiro parâmetro for `Nothing`, não existe uma função (dentro de um *functor*) para ser aplicada e `Nothing` é retornado. Por outro lado, se o primeiro parâmetro for um `Just`, então `f` contém a função que deve ser mapeada no *functor* `x` contido no segundo parâmetro. Então, mapear `f` em `x` consiste simplesmente em se chamar `fmap`.

De forma prática, *applicative functors* provêem um nível de abstração maior que *functors* e menor que *monads* [34], e foram utilizados no desenvolvimento de diversas funcionalidades do protótipo, simplificando o código e tornando-o mais idiomático. Contudo, sua principal utilização foi na escrita de *parsers* para transformar *strings* representando objetos JSON em ADTs Haskell e na serialização de ADTs Haskell para *strings* JSON.

Assim como *functors*, *applicative functors* devem respeitar algumas leis [2, 34]:

1. Identidade: colocar a função identidade em um contexto minimalista utilizando a função `pure` e aplicá-lo a um *functor* utilizando `(<*>)` deve retornar o próprio *functor*:

   ```
   pure id <*> v = v
   ```

2. Homomorfismo: Aplicar uma função sem efeito em um argumento sem efeito em um contexto com efeito deve ser o mesmo que aplicar a função ao argumento e, então, injetar o resultado no contexto com `pure`:

   ```
   pure f <*> pure x = pure (f x)
   ```

3. Alternância: A avaliação da aplicação de uma função com efeito em um argumento puro não deve depender da ordem em que o argumento ou a função são avaliados:

   ```
   u <*> pure y = pure ($ y) <*> u
   ```

4. Composição: Essa lei expressa uma espécie de propriedade associativa da função `(<*>)`:

   ```
   u <*> (v <*> w) = pure (.) <*> u <*> v <*> w
   ```

Por fim, o módulo `Control.Applicative` exporta ainda o operador `(<$>)`, que é um sinônimo de `fmap`, e a *typeclass* `Alternative`, definida como:

```
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a
```

A função `empty` pode ser entendida como um constante representando erro. Para o tipo `Maybe`, por exemplo, `empty` é definida como `Nothing`, fazendo com que uma sequência de computações no contexto de `Maybe`, falhe, sempre que for chamada. Já a função `(<|>)` recebe dois *alternatives* e funciona da seguinte maneira: tenta executar o primeiro e, em caso de sucesso, o resultado da computação é retornado e o segundo é ignorado. Caso o primeiro *alternative* falhe, executa o segundo e retorna o resultado. Para o tipo `Maybe`, o segundo parâmetro será retornado se o primeiro for `Nothing` ou `empty`. O operador de alternativa `(<|>)` foi fundamental na escrita de *parsers* e na definição do fluxo de execução dos *handlers* no protótipo, conforme apresentado na Seção 5.4.5.

4.3.3 Monads

Monads e *functors* estão intimamente relacionados. Esses termos são derivados de um ramo da matemática chamado *Teoria das Categorias*, mas sua transição para Haskell não ocorreu de forma totalmente transparente [36].

Em Teoria das Categorias, um *monad* é construído a partir de um *functor*. Dessa forma, seria natural imaginar que a *typeclass* `Monad` fosse uma subclasse de `Functor`. Entretanto, por motivos históricos, `Monad` foi introduzida na linguagem Haskell antes de `Functor`, e não é dessa forma que `Monad` está definida no `Prelude` (módulo padrão Haskell).

Contudo, para contornar essa decisão questionável, a comunidade de desenvolvedores segue a convenção de definir uma instância de `Functor` sempre que uma instância de `Monad` for definida para determinado tipo de dados. Dessa forma, é razoável supor que a função `fmap` funcionará para qualquer `Monad`. Existe também no módulo `Control.Monad` uma função denominada `liftM`, cuja assinatura é:

```
liftM :: Monad m => (a1 -> r) -> m a1 -> m r
```

Observa-se que a assinatura de `liftM` é bastante similar à `fmap`, exceto pela restrição polimórfica em `m`, que é feita em `Monad`, ao invés de `Functor`. De fato, a função `fmap` faz o mesmo que a função `liftM` em *monads*: abstrai uma função pura no *monad*.

Em Haskell, um *monad* é um tipo parametrizado que é uma instância da *typeclass* `Monad` e que obedece algumas leis. O elemento fundamental de um *monad* é o operador `(>>=)`, que permite o encadeamento de funções. *Monads* são um conceito abstrato e certo tempo é necessário para que se adquira intuição acerca de seu funcionamento e utilidade. São fundamentais na linguagem Haskell, possuindo diversas aplicações, constituindo, inclusive, a base para a introdução de efeitos colaterais na linguagem, como a execução de operações de entrada e saída (I/O), preservando-se o aspecto de pureza funcional.

A *typeclass* `Monad` é definida como:

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

A função `return` é idêntica à função `pure` em *applicative functors*, sendo utilizada para injetar um valor puro em um contexto computacional mais abstrato. O operador `(>>=)`, denominado *bind*, é uma função que recebe uma computação com efeito (`m a`) e uma função que mapeia um valor puro (`a`) em uma nova computação com efeito (`m b`). Assim, o *bind* recebe um valor e o combina com uma função para produzir um novo valor. Esse novo valor, em seguida, pode ser passado para um outro `(>>=)` que repetirá o processo, e assim por diante. Dessa forma, o operador `(>>=)` pode ser entendido como um mini-avaliador de expressões.

O *bind* serve para combinar funções, criando uma cadeia computacional. Como sua implementação é diferente para cada tipo que é uma instância de `Monad`, essa *combinação* pode ocorrer de diferentes maneiras: por meio do sequenciamento das computações, pela execução de uma função múltiplas vezes, pela execução em ordem reversa da cadeia computacional, por meio do descarte de algumas computações, pela execução de certas computações da cadeia em *threads* independentes, entre outros.

A linguagem Haskell provê também a notação `do`, que é um "açúcar sintático" para a escrita de código monádico (que utiliza *monads*). Essa sintaxe alternativa aumenta a legibilidade e agrega um estilo mais imperativo ao código [2]. O código abaixo apresenta a utilização dessa notação, apresentando duas formas equivalentes de escrita de código no *monad* `IO`. A função `f` utiliza, explicitamente, o operador *bind* e expressões *lambda*. Já a função `f'`, utiliza a notação `do`:

```
f :: IO ()
f = putStrLn "What is your name?" >>= \_ ->
    getLine >>= \name ->
    putStrLn $ "Your name is: " ++ name

f' :: IO ()
f' = do
    putStrLn "What is your name?"
    name <- getLine
    putStrLn $ "Your name is: " ++ name
```

De forma similar aos *functors* e aos *applicative functors*, *monads* devem obedecer algumas leis que permitam que suposições razoáveis sejam feitas sobre o tipo e seus comportamentos [2, 32]:

1. Identidade à esquerda: Injetar um valor puro em um contexto padrão, utilizando a função `return`, e aplicá-lo à uma função utilizando o operador `(>>=)`, deve produzir o mesmo resultado de aplicar a função diretamente ao valor:

   ```
   return x >>= f = f x
   ```

2. Identidade à direita: Utilizar o operador `(>>=)` para aplicar um valor monádico à função `return` deve resultar no valor monádico original:

   ```
   m >>= return = m
   ```

3. Associatividade: O resultado final da avaliação de uma cadeia de funções encadeadas com o operador `(>>=)` deve ser independente da maneira com que as computações são agrupadas:

   ```
   (m >>= f) >>= g = m >>= (\x -> f x >>= g)
   ```

Capítulo 5

Solução

Este capítulo descreve a implementação do protocolo proposto. Inicialmente, são descritos os requisitos do protocolo. Em seguida, apresenta-se o *design* em alto nível e o funcionamento do protocolo em dois cenários possíveis de utilização. Por fim, os detalhes de implementação e os resultados de uma análise de desempenho são apresentados.

5.1 Requisitos do Protocolo

O protocolo de autenticação e autorização proposto deverá ser aderente à arquitetura REST, de forma que possa permitir que os serviços ofertados pela Divisão de Tecnologia da PCDF possam ser acessados por um número relativamente grande de clientes. Os requisitos inerentes ao protocolo, conforme especificados por [6], são descritos nesta seção.

RQ1. Para promover a segurança de sessão, toda a comunicação entre cliente e servidor será realizada utilizando o protocolo HTTPS (*HyperText Transfer Protocol over Secure Sockets Layer*), usando o SSL/TLS para garantir a confidencialidade e integridade das mensagens. Para isso, serão utilizados certificados digitais X.509, emitidos por uma entidade certificadora, necessários para que o cliente possa autenticar o servidor e enviar mensagens cifradas assimetricamente por meio da chave pública contida no certificado. Antes de qualquer interação com os servidores, o cliente deverá realizar a validação do certificado.

RQ2. Deverá ser utilizada criptografia assimétrica para proteger as mensagens trocadas entre o cliente e os servidores da PCDF. Todas as mensagens deverão ser assinadas digitalmente por meio de algoritmos de *hash* criptográfico.

RQ3. O protocolo deverá permitir acesso aos serviços apenas ao pessoal ou instituições autorizadas, de forma que a autenticação e a autorização siga padrões definidos na política de segurança, ou seja, para ser autenticado e autorizado, o usuário deverá apresentar credenciais válidas. Essas credenciais deverão ser cifradas, assinadas e enviadas no cabeçalho das requisições HTTPS.

RQ4. O protocolo deverá ser escalável, em termos de sobrecarga, tamanho do domínio de proteção e de manutenção e permitir a preservação de privacidade, uma vez que, para proteger de entidades maliciosas os clientes e fornecedores de recursos, as interações deverão revelar o mínimo de informações possível.

RQ5. A autenticação e autorização deverá ser baseada no modelo de *desafios e respostas*, que serão elaborados a partir da apresentação de declarações de identidade (*claims*). Tal requisito torna mais flexível o gerenciamento da identidade do usuário, uma vez que possibilita ao administrador desabilitar credenciais que tenham sido comprometidas de forma transparente ao usuário.

RQ6. A política de autenticação e autorização proposta no protocolo será estabelecida por meio de contrato, onde serão definidas todas as regras que deverão ser atendidas pelos usuários e pelo fornecedor do serviço.

Dessa forma, para que um usuário ou instituição possa ter acesso aos serviços ofertados pela Divisão de Tecnologia da PCDF, ele deverá concordar com um contrato prévio de acesso, devendo, primeiramente, ser cadastrado e ter definido quais são seus privilégios de acesso/autorização. Uma vez cadastrado, o usuário ou instituição deverá informar os dados que possam comprovar sua identidade no momento da autenticação, de forma que ele possa ser autorizado de acordo com seus privilégios ou permissões.

No momento do credenciamento serão geradas para o cliente múltiplas credenciais, que serão utilizadas no processo de autenticação e autorização. Essas informações serão compartilhadas entre o cliente e os servidores. Além disso, um *contrato* (cliente) poderá ter acesso a múltiplos serviços.

5.2 *Design* em Alto Nível

Esta seção tem como objetivo apresentar uma visão mais abstrata do protocolo, sem abordar detalhes de implementação, apresentando as decisões arquiteturais e as interações entre os componentes que o constituem.

5.2.1 Arquitetura do Protocolo

O protocolo proposto por [6] pode ser definido como um sistema em rede cujas mensagens possuem duas camadas de proteção independentes. A primeira consiste na utilização do protocolo HTTPS e certificados digitais X.509. A segunda consiste na assinatura digital e cifragem das mensagens com chaves diferentes das que são utilizadas na primeira etapa. Nessa camada, os agentes participantes do processo de comunicação são responsáveis pela manutenção da integridade de origem das chaves públicas (de forma similar ao protocolo SSH), que devem ser trocadas previamente por meio de um canal seguro. No caso da PCDF, essa troca é realizada fora de banda em uma *cerimônia de troca de chaves*, logo após a assinatura de um contrato formal de cooperação entre as partes. A utilização de uma camada adicional de proteção visa aumentar a robustez do protocolo tendo em vista a criticidade dos dados que são trocados.

As quatro entidades principais que constituem o protocolo são: um Servidor de Autenticação e Autorização para realizar os procedimentos de autenticação e autorização dos usuários; um Servidor de Fachada, atuando como interface única de acesso aos serviços providos e concentrando as políticas de segurança; um banco de dados, que atua como interface entre os dois servidores; e um módulo Cliente, para automatização dos passos do protocolo, podendo ser executado localmente ou em um *proxy* dentro de uma instituição.

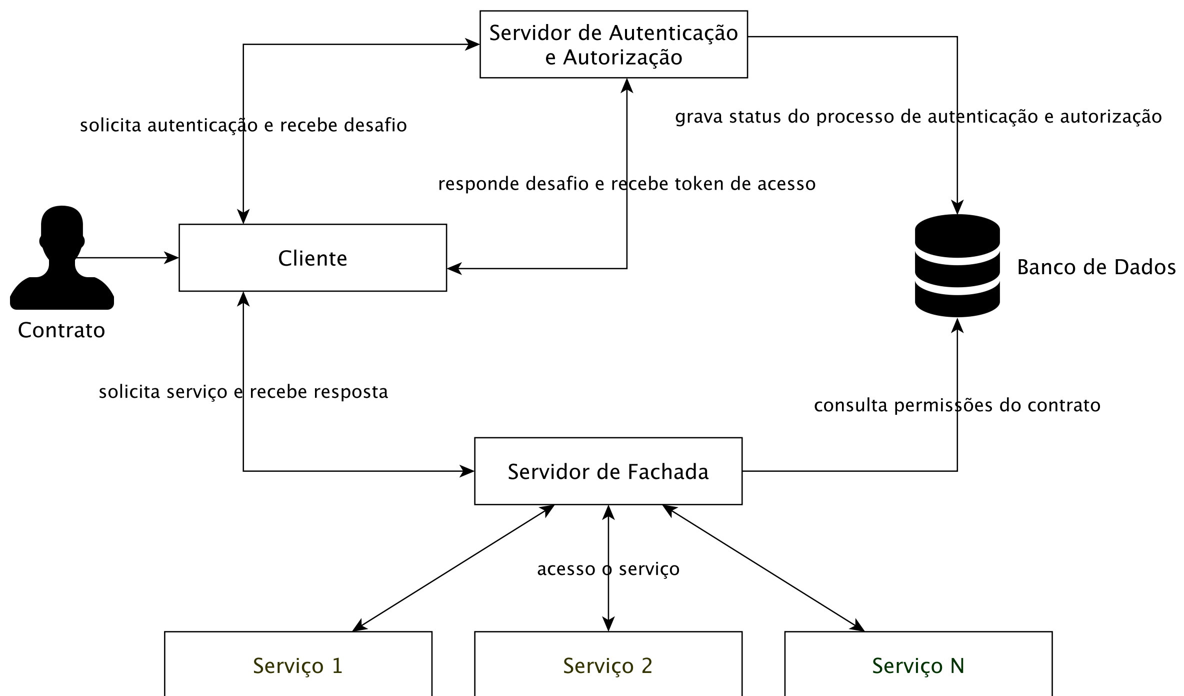

Figura 5.1: Arquitetura do sistema.

A Figura 5.1 ilustra esses componentes e suas interações. É importante notar que não há comunicação direta entre os servidores, que utilizam um banco de dados para a persistência e troca de informações referentes *exclusivamente* ao processo de autenticação e autorização de usuários.

Servidor de Autenticação e Autorização

O Servidor de Autenticação e Autorização é responsável por realizar os procedimentos de autenticação e autorização, validando a identidade dos usuários e implementando as políticas de acesso, emitindo *tokens* de autorização para os usuários de acordo com as permissões cadastradas. Os usuários ou clientes são aqui denominados *contratos* e representam um órgão parceiro da PCDF. Exemplos de *contratos* podem ser: Ministério Público da União (MPU), Tribunal de Contas do Distrito Federal (TCDF), Departamento de Trânsito do Distrito Federal (DETRAN/DF), entre outros.

O processo de autenticação se dá, basicamente, pela geração de *desafios* por parte do servidor, os quais devem ser respondidos pelo cliente. Os desafios e suas respectivas respostas são geradas a partir de um segredo compartilhado entre o cliente e o Servidor de Autenticação e Autorização. Esse segredo compartilhado é, na verdade, uma *lista de credenciais*. Dessa forma, tanto o cliente quanto o servidor devem manter essa lista em sigilo e garantir sua integridade, controlando as políticas de atualização para que não sejam introduzidas inconsistências. O Servidor de Autenticação e Autorização "conhece" as credenciais de todos os clientes.

A autenticação é baseada no modelo de *claims*, ou seja, para se autenticar e obter um *token* de autorização de acesso a um serviço, um cliente deve fazer uma requisição ao

Servidor de Autenticação e Autorização apresentando uma identidade (a ser verificada) e solicitando autenticação. A ideia é que, ao fazer isso, o cliente está *atestando* (*claiming*) ser alguém (um *contrato*). O Servidor de Autenticação e Autorização, a fim de validar a identidade do cliente, seleciona aleatoriamente uma de suas credenciais e gera um desafio solicitando que o cliente a apresente. O cliente, ao receber essa resposta, obtém a credencial correspondente ao identificador recebido e a envia de volta para o Servidor de Autenticação e Autorização. Finalmente, o Servidor de Autenticação e Autorização verifica se a credencial apresentada está correta e equivale à que foi solicitada. Em caso positivo, a autenticação é bem sucedida e a etapa de autorização é iniciada.

A autorização consiste em verificar se o *contrato* tem *permissão* para acessar o serviço solicitado. As permissões que um determinado contrato possui são atribuídas pelo administrador do sistema na PCDF após a assinatura de um termo de uso entre a PCDF e o Órgão parceiro. Caso o *contrato* tenha permissão de acesso, o Servidor de Autenticação e Autorização emite um *token* temporário de autorização de acesso ao serviço que deve ser apresentado em todas as requisições realizadas para o Servidor de Fachada.

A autorização é, então, realizada *por usuário* e *por serviço*, e o *token* de autorização associado possui uma janela de tempo de validade restrita, mitigando os riscos em potencial que uma possível interceptação poderia gerar. Esse esquema é mais restritivo que esquemas utilizados em grandes provedores de serviços na Internet, como o *Single Sign-On* (SSO), amplamente utilizado por empresas como o Google e o Facebook. O SSO é uma propriedade de controle de acesso que possibilita que um usuário possa acessar sistemas diferentes e independentes (apesar de muitas vezes relacionados) realizando *login* apenas uma vez. O mecanismo utilizado é também mais restritivo do que o utilizado em protocolos como o OpenID ou o OAuth, pois o tempo de expiração do *token* de acesso aos serviços é consideravelmente menor.

Servidor de Fachada

O Servidor de Fachada atua como um *proxy* de acesso aos diferentes serviços providos pela PCDF. Os serviços oferecidos podem ter sido desenvolvidos em diferentes linguagens de programação, estarem sendo executados em diferentes sistemas operacionais e em diferentes máquinas. O Servidor de Fachada fornece uma interface Web única de acesso aos serviços, sendo agnóstico em relação às tecnologias de implementação ou execução dos serviços. O objetivo é esconder do cliente informações que possam ser úteis durante um ataque, como o tipo e versão do servidor utilizado, o sistema operacional em que o serviço está sendo executado, entre outras.

Além disso, há benefícios com respeito à *modularidade*, ou seja, essa arquitetura possibilita a concentração de todas as políticas de segurança em uma interface única de acesso que, além de retirar certas responsabilidades relativas à segurança dos desenvolvedores de serviços, concentrando-as no Servidor de Fachada, aumenta a manutenibilidade da aplicação, tornando-a menos vulnerável e menos suscetível a erros. O protocolo busca integrar o fator *segurança*, de acordo com os requisitos elicitados na Seção 5.1, como característica transversal às aplicações. Dessa forma, do ponto de vista do cliente, os serviços são *opacos*, pois tudo que o cliente conhece sobre um serviço é sua interface, que consiste basicamente em: um identificador universal único (UUID — *Universally Unique IDentifier*), métodos

HTTP implementados e os parâmetros (mais detalhes na Seção 5.4.5). Em contrapartida, o Servidor de Fachada se torna um ponto único de falha.

De forma prática, o Servidor de Fachada recebe uma requisição HTTPS contendo informações sobre o serviço que o cliente deseja consumir, uma declaração de identidade do cliente e um *token* de autorização de acesso ao serviço. O Servidor de Fachada, então, verifica se o serviço solicitado existe e se o *token* fornecido é válido, isto é:

1. O *token* não está expirado.

2. O *token* dá acesso ao serviço solicitado.

3. O *token* foi emitido para o *contrato* apresentado.

Caso todas as condições sejam satisfeitas, o Servidor de Fachada intermedeia a comunicação entre o cliente e o serviço, fazendo a requisição correspondente ao serviço e retornando a resposta para o cliente. Caso a primeira restrição não seja satisfeita, o Servidor de Fachada retorna para o cliente uma mensagem de redirecionamento para o Servidor de Autenticação e Autorização, para que o cliente possa obter um novo *token*. Caso uma das duas últimas restrições não seja satisfeita, o Servidor de Fachada retorna uma mensagem de erro, informando que o *token* apresentado não dá acesso ao serviço solicitado.

Módulo Cliente

O módulo Cliente é um programa que deve ser executado localmente na máquina do usuário que deseja acessar os serviços providos pela PCDF ou em uma máquina *proxy* dentro de um órgão ou instituição parceira. O objetivo é a automatização de todo o processo de autenticação, autorização e acesso aos serviços dentro do protocolo. Dessa forma, dentro de uma organização, como o MPU ou o TCDF, pode-se executá-lo em um *proxy* e vários usuários podem ter acesso aos serviços através da mesma instância do Cliente.

De forma prática, o Cliente, assim como o Servidor de Fachada, também atua como um *proxy*, porém, intermediando a comunicação entre o *usuário final* e o *protocolo*. Ele captura e interpreta as intenções do usuário, mapeando-as em mensagens no protocolo. Além disso, o Cliente possui uma certa "inteligência", ou seja, ele é capaz de seguir possíveis redirecionamentos automaticamente, como por exemplo quando o *token* de autorização chega no Servidor de Fachada expirado e a resposta recebida é um redirecionamento para o Servidor de Autenticação e Autorização.

O Cliente mantém uma lista de *tokens* de autorização e, quando chega uma requisição para um determinado serviço, se existir algum *token* que ainda esteja válido (não expirado), o Cliente acessa diretamente o Servidor de Fachada. Evitar o *overhead* de se dirigir ao Servidor de Autenticação e Autorização traz ganhos de desempenho, uma vez que dá maior vazão às requisições.

5.3 Funcionamento do Protocolo

Toda a comunicação no protocolo é realizada sob o protocolo HTTPS, com as mensagens do protocolo sendo enviadas no cabeçalho das requisições. Essas mensagens carregam

dados codificados no formato JSON e contam ainda com uma camada adicional de proteção, ou seja, o conteúdo presente no cabeçalho é um *token* JSON assinado digitalmente e cifrado simetricamente, onde a chave simétrica é cifrada com a chave pública do remetente e enviada junto com a mensagem, de acordo com o esquema de *envelope digital* apresentado na Seção 2.1.3. A escolha desse esquema se deve ao fato de que ele busca balancear os requisitos de segurança e os requisitos de desempenho. Mais detalhes sobre os algoritmos utilizados e o formato do *token* são apresentados na Seção 5.4.2.

Esta seção busca ilustrar os cenários de uso do protocolo, mostrando os passos a serem seguidos para que um usuário possa se autenticar, receber um *token* de acesso a um serviço e, por fim, acessar o serviço. Apresenta-se também, com um maior nível de detalhamento, os formatos de mensagens e as ferramentas criptográficas utilizadas para garantir que os requisitos de segurança e desempenho elicitados na Seção 5.1 sejam atendidos.

No primeiro cenário, representado na Figura 5.2, o cliente ainda não foi autenticado. Já no segundo, ele está autenticado e possui um *token* de autorização válido.

5.3.1 Primeiro Cenário

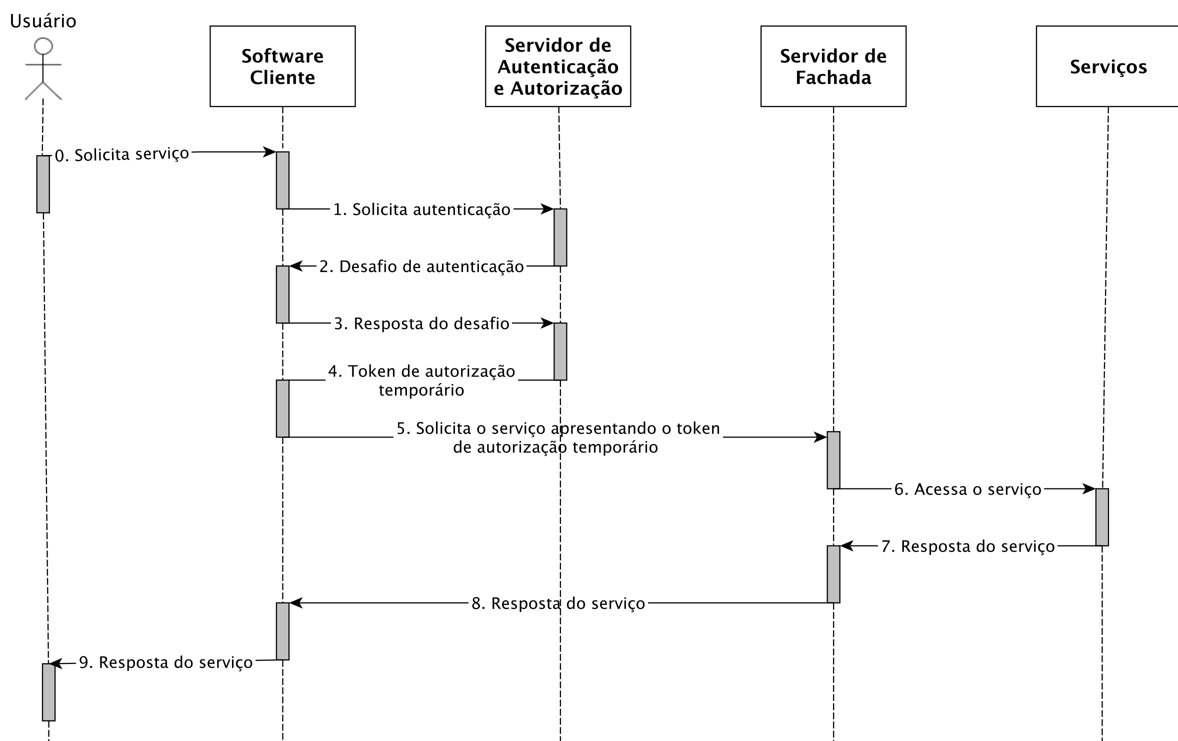

Figura 5.2: Fluxo do protocolo de autenticação e autorização proposto.

O protocolo tem início quando o usuário envia uma solicitação de autenticação ao Servidor de Autenticação e Autorização. Esse pedido é realizado por meio de uma requisição HTTPS (mensagem 1 na Figura 5.2) que contém em seu cabeçalho um *token* assinado digitalmente e cifrado de acordo com o esquema de envelope digital. Esse *token* contém

o UUID do contrato, ou seja, seguindo o modelo de *claims* apresentado na Seção 5.2.1, o cliente está afirmando sua identidade. O cliente autentica o servidor de duas formas:

1. Durante o estabelecimento da conexão SSL, por meio do certificado digital, que é um documento que associa uma identidade a uma chave pública e que é validado por um mecanismo de autenticação objetiva recursiva;

2. Por meio do *token* enviado no cabeçalho, que é cifrado com a chave pública da PCDF que é recebida fora de banda, durante a assinatura do contrato. Essa chave é diferente da que está contida no certificado digital e consiste em uma forma de autenticar o servidor porque, supondo-se as premissas de autenticidade da chave pública e sigilo da chave privada correspondente, apenas a PCDF (detentora da chave privada) conseguirá decifrar os criptogramas.

Na segunda mensagem, ao receber uma solicitação de autenticação, o Servidor de Autenticação e Autorização extrai os dados do *token*, decifrando-o com uma chave simétrica decifrada a partir de sua chave privada, e verifica a integridade da mensagem por meio da verificação da assinatura digital. Se houver qualquer problema, uma mensagem de erro HTTP *Bad Request* (código 400) é retornada para o usuário.

Caso não haja problemas, procede-se com o processo de autenticação do usuário, que consiste em consultar em uma base de dados se o UUID apresentado está associado a um contrato válido. Em caso afirmativo, o Servidor de Autenticação e Autorização gera para o usuário um desafio de autenticação, selecionando aleatoriamente o código de uma das credenciais do contrato. Uma *credencial* é uma tupla `<chave,valor>`. Esse desafio é baseado no modelo de *segredo compartilhado*, onde o servidor pede ao cliente que apresente a resposta para uma pergunta que, supostamente, apenas os dois conhecem (o valor de uma credencial).

Antes de enviar para o usuário uma reposta solicitando a apresentação dessa credencial, o Servidor de Autenticação e Autorização grava na base de dados o desafio gerado, associado a um selo temporal (*timestamp*) e uma *flag* (opcional) indicando se o desafio já foi respondido ou não. O selo temporal indica o tempo limite em que o desafio permanece válido e pode ser respondido pelo cliente. A *flag* é um número inteiro, indicando o número de vezes que o desafio foi respondido (correta ou incorretamente), sendo um mecanismo de proteção adicional que permite a configuração de um número máximo de tentativas. Para garantir a eficiência e segurança do protocolo, o campo *flag* deve ser configurado, idealmente, com o valor um, que é o valor padrão.

Por fim, envia-se para o cliente uma mensagem contendo o UUID do desafio (o identificador da pergunta), o UUID da credencial (a pergunta a ser respondida), um selo temporal (tempo limite para resposta) e uma URL (para quem responder). Além de impor uma janela de tempo para resposta ao usuário, o selo temporal funciona como um *nonce*, impedindo que um terceiro escutando o canal possa realizar ataques de repetição (*replay*) por meio da análise dos criptogramas. Já a URL foi inserida pensando-se na questão da extensibilidade, ou seja, o protótipo desenvolvido permite a utilização de mais de um Servidor de Autenticação e Autorização, possibilitando um balanceamento da carga e ganho de desempenho.

A mensagem é, então, cifrada com a chave pública do cliente, que fica associada ao contrato, no banco de dados do servidor. Para completar a autenticação, o cliente precisa

responder o desafio, porém, novamente, presumindo-se a autenticidade da chave pública do cliente e o sigilo da chave privada correspondente, esse passo, por si só, já consiste em uma forma de autenticar o cliente, ou seja, se um terceiro tentar se passar por um contrato enviando uma solicitação para o servidor contendo um UUID forjado, ele receberá como resposta um criptograma que não será capaz de decifrar.

Na terceira mensagem, após receber o desafio do Servidor de Autenticação e Autorização, o Cliente extrai os dados do *token* de forma similar à explicada anteriormente (decifrando e verificando a assinatura). Se tudo estiver correto, o Cliente busca em sua base de dados local a credencial solicitada. De posse da credencial, o Cliente envia uma mensagem para a URL recebida contendo: o UUID do desafio (o identificador da pergunta), o valor da credencial (resposta da pergunta), o UUID do contrato (quem está respondendo a pergunta), o UUID do serviço a ser acessado e um selo temporal. De forma similar à mensagem anterior, esse selo temporal também atua como um *nonce*, fazendo com que respostas iguais gerem criptogramas diferentes [1].

Já na quarta mensagem, o Servidor de Autenticação e Autorização recebe a resposta do desafio de autenticação e, após decodificar e verificar o *token*, inicializa a checagem da resposta. Para isso, primeiro verifica:

1. O desafio existe.
2. O desafio não está expirado.
3. O desafio foi emitido para o contrato especificado.
4. O número de vezes em que o desafio foi respondido não excede o número máximo de vezes pré-estabelecido.
5. O resposta do desafio está correta.

Se algum dos predicados elicitados for falso, o Servidor de Autenticação e Autorização responde com uma mensagem HTTP *Forbidden* (código 403), indicando que o acesso ao serviço não pôde ser concluído. Caso contrário, verifica-se as permissões do contrato, isto é, se o serviço existir e o contrato puder acessá-lo, emite um *token* temporário de autorização de acesso ao serviço. É importante ressaltar que o *token* emitido é gerado, caso não exista na base de dados um *token* ainda válido, emitido para aquele contrato e para aquele serviço; ou reemitido, caso contrário. O *token* de autorização gerado é, então, persistido na base de dados, associado com uma data de expiração, o contrato e o serviço para os quais foi emitido. Isso é feito para que, futuramente, o Servidor de Fachada possa verificar quais privilégios a entidade requisitante do serviço possui. Finalmente, o *token* de autorização é enviado para cliente.

Na quinta mensagem, de posse do *token* de autorização temporário, o Cliente envia uma mensagem para o Servidor de Fachada com os dados do serviço que deseja acessar (identificador do serviço, método HTTP e parâmetros) e o *token* recebido.

Finalmente, após receber a requisição e decodificar os dados, o Servidor de Fachada checa o *token* de autorização recebido:

[1] Se o valor de *flag* for configurado com o valor um, restringindo os desafios para que possam ser respondidos apenas uma vez, o UUID do desafio pode ser utilizado como *nonce*, inutilizando o selo temporal.

1. O *token* de autorização existe.
2. O *token* é válido (não expirou).
3. O *token* foi emitido para o contrato especificado.
4. O *token* foi emitido para o serviço especificado.

Se 1 ou 2 forem falsas, o Servidor de Fachada responde com um redirecionamento (código HTTP 401 — *Unauthorized*) para o Servidor de Autenticação e Autorização para que o usuário possa obter um *token* válido. Se 3 ou 4 forem falsas, o Servidor de Fachada responde com uma mensagem HTTP *Forbidden* (código 403), indicando que o usuário não tem permissão e que o acesso ao serviço não pôde ser concluído. Caso não haja problemas, o Servidor de Fachada acessará o serviço solicitado e retornará a resposta para o cliente, intermediando a comunicação. Para isso, obtém-se da base de dados a URL real do serviço (utilizando-se o UUID recebido) e realiza-se uma nova requisição HTTP ou HTTPS (recurso configurável), passando-se os parâmetros e o método HTTP correspondentes.

A sexta mensagem é a resposta do serviço.

5.3.2 Segundo Cenário

Neste cenário, o usuário deseja consumir um serviço e possui um *token* de autorização (para acessar aquele serviço) que ainda está válido, isto é, o limite de tempo de expiração do *token* não foi excedido.

Dessa forma, para consumir o serviço, não é necessário passar antes pelo Servidor de Autenticação e Autorização, uma vez que as etapas de autenticação e autorização já foram realizadas previamente, sendo suficiente o envio do *token* que o usuário já possui, em conjunto com a requisição do serviço, diretamente para o Servidor de Fachada.

Quando a requisição chega no Servidor de Fachada, o procedimento executado é o mesmo que ocorre no processamento da mensagem 5 no primeiro cenário, ou seja, o *token* de autorização recebido é checado e, caso esteja válido, o serviço exista e o usuário tenha permissões para acessá-lo, o Servidor de Fachada acessa o serviço e retorna a resposta para o usuário.

Se durante a checagem o *token* for considerado inválido (ex: ter expirado durante a transmissão), o Servidor de Fachada responde com um redirecionamento para o Servidor de Autenticação e Autorização para que o cliente possa obter um novo *token* de autorização.

5.4 *Design* em Baixo Nível

Esta seção apresenta detalhes de implementação do procolo, mostrando a estrutura a nível de sistema e os detalhes arquiteturais a nível de subsistema. São apresentados os algoritmos, ferramentas e tecnologias escolhidas e o funcionamento geral da aplicação.

A implementação foi dividida em subprojetos independentes, desenvolvidos de modo a otimizar a modularidade, o reúso, a testabilidade e a auditabilidade do sistema. Mais especificamente, foram desenvolvidos: uma biblioteca criptográfica, denominada `jwt-min`,

para utilização em todos os subprojetos; um servidor para realizar os processos de autenticação e autorização dos usuários; um servidor *proxy* para centralizar a aplicação das políticas de segurança e esconder detalhes de implementação dos serviços; um modelo de persistência utilizando um banco de dados REST de alto desempenho; um módulo cliente para automatização dos passos do protocolo no lado dos usuários.

Esses sistemas constituem uma solução puramente funcional para o problema exposto. É importante frisar o fator *independência* na composição desses sistemas, ou seja, eles não são independentes apenas do ponto de vista de módulos reusáveis, mas também do ponto de vista de *sistemas em rede* que, de acordo com Tanenbaum e van Renesse [43], são diferentes de sistemas distribuídos. Um sistema distribuído é aquele que enxerga seus usuários como um sistema centralizado comum, mas que é executado em várias CPUs independentes. Por outro lado, sistemas baseados em rede são aqueles capazes de operar através de uma rede, mas não necessariamente de uma forma que é transparente para o usuário.

Todos os sistemas desenvolvidos (servidores, banco de dados e o módulo cliente) podem ser executados de forma totalmente independente, estando cada um em uma máquina diferente. Além disso, todos os subsistemas são parametrizados pelo endereço IP e número de porta dos demais subsistemas com os quais interage. Como o protocolo funciona pela Internet, sob o protocolo HTTPS, esses dois dados são suficientes para o mapeamento das aplicações, identificando, respectivamente, o endereço da máquina (*host*) de destino e a aplicação executada nessa máquina. Esses parâmetros são configurados de forma não intrusiva, por meio de um arquivo de texto constituído de tuplas no formato: `<chave> = <valor>`.

5.4.1 Tecnologias Utilizadas

O protótipo foi desenvolvido na linguagem de programação puramente funcional Haskell, com o compilador Glasgow Haskell Compiler (GHC) versão 7.6.3. Para gerenciamento de dependências e controle de *build*, foi utilizada a ferramenta `cabal`, com o novo recurso de *cabal sandboxes* disponível a partir da versão 1.18.0. Com isso, é possível minimizar a ocorrência de conflitos de dependências (*Cabal Hell*) [18]. Essas ferramentas podem ser obtidas através da instalação da Plataforma Haskell [2].

Foi utilizado o Snap Framework, que é um *framework* Web Haskell, para manipulação das requisições e respostas HTTP e o banco de dados não-relacional Apache CouchDB para persistência dos dados.

A implementação do SSL utilizada foi o OpenSSL versão 0.9.8y, que não é vulnerável ao ataque recente conhecido como HeartBleed [3]. Para utilizar o SSL a partir de código Haskell, foi utilizada a API `HsOpenSSL`. Os testes foram realizados com chaves privadas e certificados digitais X.509 gerados com o utilitário de linha de comando `openssl`.

Por fim, o desenvolvimento foi realizado sob o sistema de controle de versão distribuído e gerenciamento de código-fonte (SCM) Git, e todo o código-fonte está publicamente disponível no GitHub [4]

[2] `http://www.haskell.org/platform/`.
[3] `http://heartbleed.com/`.
[4] `https://github.com/alexandrelucchesi/pfec/`.

5.4.2 Biblioteca Criptográfica: `jwt-min`

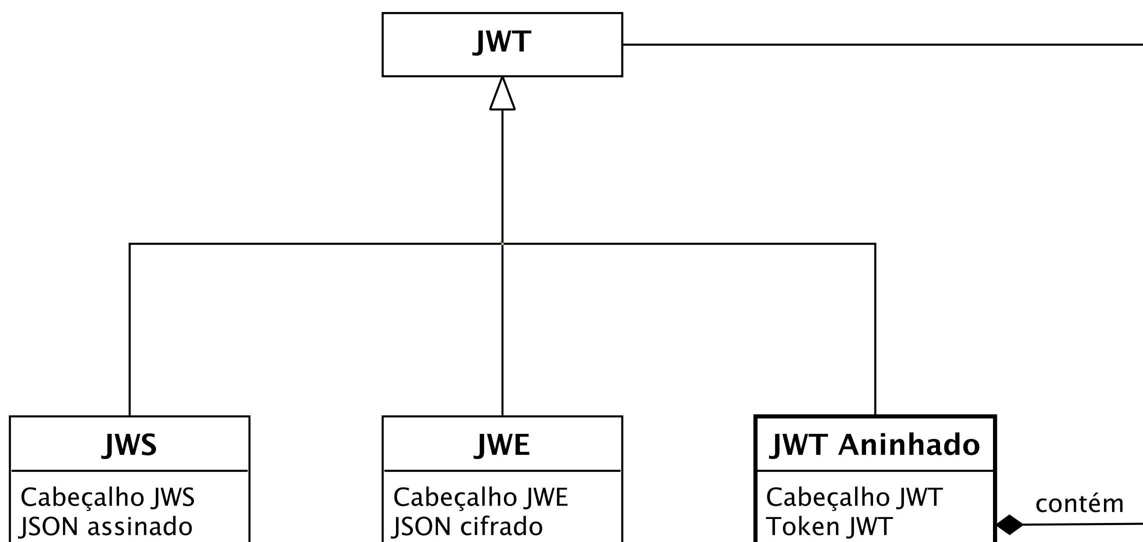

Figura 5.3: Estrutura de um *token* JWT.

A `jwt-min` é uma biblioteca criptográfica que foi desenvolvida para ser utilizada por todos os projetos, contendo uma implementação parcial da especificação JSON Web Tokens [25] (JWTs), permitindo a construção e desconstrução de *tokens* assinados digitalmente e cifrados de acordo com as especificações JSON Web Signatures [24] (JWS) e JSON Web Encryption [26] (JWE) na forma compacta e um subconjunto de algoritmos da especificação JSON Web Algorithms [23] (JWA). A `jwt-min` foi desenvolvida para viabilizar o desenvolvimento do protótipo, pois, até o momento, não foram encontradas implementações alternativas estáveis ou minimamente funcionais da especificação JWT em Haskell.

O JWT é um formato de representação compacta de *claims* destinado a ambientes com restrição de recursos (espaço de armazenamento ou banda), tais como cabeçalhos de autorização HTTP e parâmetros de consulta (*query parameters*) em URIs. Essas *claims* são codificadas e transmitidas como um objeto no formato Javascript Object Notation (JSON) codificado em Base64, que é usado como o *payload* de uma estrutura JWS ou o texto em claro de uma estrutura JWE, permitindo que as *claims* sejam assinadas digitalmente e/ou cifradas. JWTs são sempre representados utilizando a *serialização compacta* do JWS ou do JWE.

As mensagens no formato JWT podem ser assinadas digitalmente, cifradas, ou assinadas e cifradas, podendo conter vários níveis de assinatura e cifragem. Nesse caso, o JWT é denominado *aninhado* (*nested* JWT), e este foi o padrão escolhido para a implementação no protocolo: um JWT aninhado de duas camadas, onde a mensagem é primeiramente assinada e, depois, cifrada utilizando uma combinação de algoritmos simétricos e assimétricos, de acordo com o esquema de *envelope digital* descrito na Seção 2.1.3. Como a assinatura da mensagem é realizada antes da cifragem, somente o detentor da chave pri-

vada que "abre o envelope" pode verificar sua validade. A Figura 5.3 ilustra a estrutura de um *token* JWT.

A escolha do padrão JWT para codificação das mensagens do protocolo se deve aos seguintes fatores:

- Complacência com o formato JSON, que foi o formato escolhido para codificação das mensagens do protocolo e que é amplamente utilizado em aplicações Web baseadas em REST.
- Destinado a ambientes com restrição de recursos, codificando os dados em mensagens pequenas e maximizando o desempenho pela otimização da utilização de banda e espaço de armazenamento.

5.4.3 Algoritmo de Assinatura

Um *token* JWS representa conteúdo protegido com assinaturas digitais ou Códigos de Autenticação de Mensagens (MACs) usando a notação JSON e codificado em Base64. O algoritmo `RSASSA-PKCS-v1_5 SHA-256` foi o escolhido para implementação de assinatura digital nas mensagens do protocolo.

Esta seção apresenta as etapas de *codificação* e *validação* de *tokens* JWS. A codificação consiste na seguinte sequência de passos: criação do cabeçalho JWS codificado em Base64; codificação em Base64 da mensagem; assinatura da *string* resultante da concatenação do cabeçalho JWS com a mensagem, ambos codificados em Base64; e construção do *token* JWS. A validação consiste na seguinte sequência de passos: descoberta do algoritmo de assinatura; obtenção da assinatura e do conteúdo a ser verificado; e a verificação da assinatura.

Codificação

A etapa de codificação de uma mensagem utilizando o algoritmo `RSASSA-PKCS-v1_5 SHA-256` consiste na seguinte sequência de passos:

1. Criação do cabeçalho JWS: Na `jwt-min`, utiliza-se um cabeçalho contendo o menor número de informações possível que viabilize a execução do protocolo, consideradas as regras de campos obrigatórios e opcionais da especificação. O único campo obrigatório em um *token* JWS é o que informa sobre o algoritmo utilizado para a assinatura digital. Como apenas um algoritmo foi implementado na `jwt-min`, o conteúdo do cabeçalho é sempre o seguinte (caracteres codificados em UTF-8):

   ```
   {"alg":"RS256"}
   ```

   Codificando esse JSON em Base64, tem-se a *string* que representa o cabeçalho do *token* JWS.

2. Codificação em Base64 da mensagem: Todas as mensagens utilizadas na aplicação estão no formato JSON. Considere a seguinte mensagem de exemplo, que é a primeira mensagem enviada pelo Cliente para o Servidor de Autenticação e Autorização para iniciar o processo de autenticação (quebras de linhas apenas para efeitos ilustrativos):

```
{
    "contractUUID":"1d15162a-7120-4210-9f47-edfd699e1e54",
    "sentAt":"2014-06-19T19:42:39.110Z"
}
```

Codificando esse JSON em Base64, tem-se a *string* que representa o *payload* do *token* JWS.

3. Concatenação do cabeçalho com o *payload*: O próximo passo consiste na concatenação, separada por um caracter de ponto final ("."), das duas *strings* calculadas nos passos 1 e 2.

4. Assinatura: A *string* resultante do passo 3 é a representação em ASCII de uma sequência de octetos que é utilizada como entrada para uma função de assinatura digital. Essa função recebe como parâmetro a chave privada RSA e a função de *hash* a ser utilizada. Nesse algoritmo e em todas as mensagens da `jwt-min`, é utilizada a `SHA-256`.

5. Construção do *token* JWS: O *token* JWS na representação compacta a ser transmitido é formado pela concatenação dos valores em ordem do cabeçalho, *payload* e assinatura, separados por um caracter de ponto final (".").

Validação

Ao receber uma mensagem assinada conforme o esquema apresentado, o receptor pode validar a assinatura seguindo a seguinte sequência de passos:

1. Descoberta do algoritmo: Para validar a assinatura da mensagem, o receptor precisa saber qual algoritmo de assinatura digital foi utilizado na codificação. Para isso, o receptor deve extrair a primeira parte do *token* JWS recebido, referente ao cabeçalho, e realizar a decodificação em Base64. O resultado da decodificação deve ser um objeto JSON que contenha, dentre outros, o campo obrigatório `alg`, contendo um identificador do algoritmo utilizado. No algoritmo em questão e para todas as mensagens do protocolo, o valor desse campo é sempre a *string* "RS256", indicando que a validação deve ser feita com o algoritmo `RSASSA-PKCS-v1_5 SHA-256`.

2. Obtenção da assinatura e do conteúdo a ser verificado: A assinatura é obtida a partir da decodificação em Base64 da última *substring* do *token* JWS. Já a *string* a ser verificada, isto é, que será utilizada como entrada para a função de verificação, começa no primeiro caracter do *token* JWS e termina, não incluindo, o segundo ponto final ("."). Esta, por sua vez, não precisa ser decodificada em Base64, pois, durante a assinatura, é assinada nesta codificação.

3. Verificação da assinatura: Finalmente, passa-se como argumento para uma função de verificação de assinatura `RSASSA-PKCS-v1_5` configurada para usar a função de *hash* `SHA-256`, a chave pública RSA e a assinatura e a *string* a ser verificada, obtidas no passo 2.

5.4.4 Algoritmo de Cifragem

Um *token* JWE representa conteúdo cifrado usando a notação JSON, codificado em Base64. Os algoritmos criptográficos utilizados em um *token* JWE estão descritos a parte na especificação JWA. Os algoritmos `RSAES-PKCS1-V1_5` e `AES_128_CBC_HMAC_SHA_256` foram os escolhidos para a cifragem das mensagens do protocolo com os objetivos de sigilo e integridade dos dados. O primeiro é um algoritmo que implementa uma cifra assimétrica, enquanto o segundo, é um algoritmo que implementa uma cifra simétrica e utiliza uma função de *hash* para geração de rótulos de autenticação para as mensagens (MACs).

É importante frisar que a escolha desses algoritmos vai de encontro ao especificado por [6] para o protocolo, que previu a utilização de apenas algoritmos assimétricos para a proteção das mensagens. A mudança proposta neste trabalho busca refinar a utilização da criptografia, trazendo uma utilização mais racional dos mecanismos criptográficos a partir do esquema de *envelope digital* apresentado na Seção 2.1.3. Esse esquema tem como objetivo balancear os requisitos de segurança e os requisitos de desempenho, utilizando criptografia assimétrica para cifrar uma chave simétrica que, por sua vez, é utilizada para cifrar simetricamente uma mensagem. De acordo com [40], cifras simétricas são, em média, de 10^3 a 10^4 vezes mais rápidas que cifras assimétricas, e esse custo cresce com o tamanho das mensagens: quanto maior a mensagem, mais lento serão os processos de cifragem e decifragem.

Mesmo com as mensagens do protocolo sendo pequenas, o que justificaria a utilização apenas de criptografia assimétrica, uma vez que a diferença de desempenho seria irrisória, julgou-se relevante a utilização do esquema de envelope digital por dois motivos: extensibilidade, ou seja, para acomodar mudanças futuras que possam vir a ocorrer durante a evolução do protocolo; e para não limitar o tamanho das mensagens provenientes das respostas dos serviços.

Esta seção apresenta as etapas de codificação e validação de *tokens* JWE. A codificação consiste na seguinte sequência de passos: criação do cabeçalho JWE codificado em Base64; geração da chave de cifragem de conteúdo; cifragem assimétrica dessa chave; geração de um vetor de inicialização; cifragem da mensagem e criação de um código verificador de integridade; e construção do *token* JWE. A validação consiste na seguinte sequência de passos: descoberta do algoritmo de cifragem; decodificação dos campos que constituem o *token* JWT; obtenção das chaves de decifragem e de verificação de integridade; decifragem do texto cifrado; e verificação de integridade da mensagem.

Codificação

A etapa de codificação de uma mensagem utilizando o algoritmo `RSAES-PKCS1-V1_5` e `AES_128_CBC_HMAC_SHA_256` consiste na seguinte sequência de passos:

1. Criação do cabeçalho JWE:

   ```
   {"alg":"RSA1_5","enc":"A128CBC-HS256","cty":"JWT"}
   ```

   O cabeçalho JWE apresentado declara, por meio do campo `alg` (*algorithm*), que a chave de cifragem do conteúdo foi cifrada no *token* utilizando o algoritmo `RSAES-PKCS`

1-V1_5. O campo enc (*encryption algorithm*) declara que o conteúdo (texto em claro) foi cifrado com o algoritmo AES_128_CBC_HMAC_SHA_256 para produzir o texto cifrado recebido. Por fim, o campo cty (*content type*), declara que o conteúdo do *token* é, ainda, um JWT, isto é, indica que o receptor, após o processo de decifragem, obterá um novo *token* JWT, que deverá ser decodificado. O campo cty é obrigatório ao se utilizar *tokens* JWT aninhados e, no caso específico do protocolo, tem-se dois níveis de codificação, onde o nível mais externo do *token* JWT recebido codifica um JWE que, após decifrado, retorna um *token* JWS que, após verificada a assinatura, contém a mensagem original. Codificando esse JSON em Base64, tem-se a *string* que representa o cabeçalho do *token* JWE.

2. Geração da chave de cifragem de conteúdo: O texto em claro ou conteúdo do *token* é cifrado utilizando-se um algoritmo de criptografia simétrico e a integridade da mensagem é verificada a partir de uma função de *hash* criptográfico. Dessa forma, é necessária a geração de uma chave simétrica que, posteriormente, será cifrada assimetricamente para ser utilizada pelo algoritmo AES_128_CBC_HMAC_SHA_256, e de uma outra chave para ser utilizada pela função de *hash*. Essas duas chaves são derivadas de uma terceira chave, com tamanho de 256 *bits* e gerada aleatoriamente, conforme ilustrado no Passo 5a. A seguir tem-se a representação em ASCII de uma possível chave:

   [254,159,204,155,64,235,105,28,72,143,214,255,98,1,103,209,244,200,
   157,59,108,209,149,236,2,212,148,111,57,97,234,162]

3. Cifragem assimétrica da chave: Essa etapa consiste em cifrar a chave simétrica de cifragem de conteúdo assimetricamente, utilizando o algoritmo RSAES-PKCS1-V1_5 e a chave pública RSA do receptor. O resultado desse processo produz a *chave JWE cifrada*, que deve ser codificada em Base64 para posterior transmissão.

4. Geração do vetor de inicialização: Essa etapa consiste na geração aleatória de um vetor de inicialização para ser utilizado no algoritmo AES_128_CBC_HMAC_SHA_256. Esse vetor deve ter 128 *bits* de tamanho e ser codificado em Base64 para transmissão.

5. Cifragem da mensagem: Essa etapa consiste na cifragem do texto em claro. Conforme descrito na especificação [23], os algoritmos da família AES_CBC_HMAC_SHA2 são implementados utilizando *Advanced Encryption Standard* (AES) no modo *Cipher Block Chaining* (CBC) com *padding* PKCS #5 para realizar a cifragem e uma função HMAC SHA-2 para realizar a verificação de integridade. No caso do protocolo, os algoritmos escolhidos foram, respectivamente, o AES 128 CBC e o HMAC SHA-256, que juntos formam o algoritmo denominado AES_128_CBC_HMAC_SHA_256.

   (a) Extração da MAC_KEY e ENC_KEY: Conforme explicado no Passo 2, essa etapa consiste na extração das chaves a serem utilizadas para cifragem e verificação de integridade da mensagem. O processo é simples, os primeiros 128 *bits* da chave gerada naquele passo constituem a chave MAC (MAC_KEY), que é:

   [145,87,175,217,31,115,208,191,49,150,243,102,91,47,173,44]

A chave de cifragem (`ENC_KEY`) é, então, constituída pelos últimos 128 *bits*:

`[183,208,169,45,216,26,165,120,180,100,67,9,139,159,2,38]`

(b) Cifragem do texto em claro: Consiste na geração do texto cifrado utilizando o algoritmo `AES 128` em modo CBC com *padding* PKCS #5 e a `ENC_KEY` gerada. O texto em claro, nesse caso, é a representação compacta do *token* JWS apresentado no Passo 5.

(c) Criação do rótulo de autenticação: Consiste na geração de uma sequência de *bytes* para verificação de integridade utilizando-se o algoritmo `HMAC SHA-256`. A mensagem utilizada como parâmetro de entrada para a função é o vetor de inicialização, obtido no Passo 4, concatenado com o texto cifrado, obtido na etapa anterior. O resultado é, então, truncado nos primeiros 128 *bits*.

6. Construção do *token* JWE: A única diferença do *token* JWT para o *token* JWE, nesse caso, é o campo `cty` no cabeçalho. O *token* a ser enviado é constituído pela codificação em Base64 e posterior concatenação, em ordem, dos campos: cabeçalho JWE, chave gerada de 256 *bits*, vetor de inicialização, o texto cifrado e o rótulo de autenticação.

Validação

Ao receber uma mensagem cifrada conforme o esquema apresentado, o receptor pode decifrá-la seguindo a seguinte sequência de passos:

1. Descoberta do algoritmo: Para decifrar a mensagem, o receptor precisa saber qual algoritmo foi utilizado na codificação. Para isso, o receptor deve extrair a primeira parte do *token* JWT recebido, referente ao cabeçalho, e realizar a decodificação em Base64. O resultado da decodificação deve ser um objeto JSON que contenha, dentre outros, os campos obrigatórios `alg` e `enc`, contendo um identificador do algoritmo utilizado para cifrar assimetricamente a chave simétrica e os algoritmos utilizados para cifrar e autenticar a mensagem, respectivamente. No algoritmo em questão e para todas as mensagens do protocolo, o valor desses campos será sempre "RSA1_5" para o campo `alg`, indicando que a chave deve ser decifrada com o algoritmo `RSAES-PKCS1-V1_5`, e "A128CBC-HS256" para o campo `enc`, indicando que a mensagem deve ser decifrada e autenticada com o algoritmo `AES_128_CBC_HMAC_SHA_256`.

2. Decodificação do conteúdo do *token* JWT: O *token* JWT recebido é composto de 5 partes, separadas pelo caracter de ponto final (".") que, em ordem, representam: o cabeçalho do *token*, a chave de cifragem de conteúdo cifrada assimetricamente com a chave pública do receptor, o vetor de inicialização, o texto cifrado (nesse caso, um *token* JWS) e o rótulo de autenticação. Todos esses campos estão codificados em Base64 e devem, nesse momento, serem decodificados.

3. Obtenção da `MAC_KEY` e da `ENC_KEY`: Essa etapa consiste na decodificação da chave de cifragem de conteúdo em duas chaves, uma para verificar a integridade da mensagem (`MAC_KEY`) e outra para decifrar o texto cifrado (`ENC_KEY`). Elas podem ser

obtidas a partir do processo inverso do realizado durante a codificação, ou seja, utilizando sua chave privada e o algoritmo `RSAES-PKCS1-V1_5`, o receptor pode decifrar a chave, obtendo uma sequência de 256 *bits*, onde os primeiros 128 *bits* constituem a `MAC_KEY` e os últimos 128 *bits*, a `ENC_KEY`.

4. Decifragem do texto cifrado: A mensagem original é obtida quando, de posse da `ENC_KEY`, o receptor decifra o texto cifrado utilizando o algoritmo `AES 128` em modo CBC com *padding* PKCS #5.

5. Verificação de integridade: A fim de verificar a integridade da mensagem, o receptor deve concatenar o vetor de inicialização com o texto cifrado e recalcular, utilizando o algoritmo `HMAC SHA-256` e a `MAC_KEY` obtida, o *hash*, que deve ser truncado após os primeiros 128 *bits*. A mensagem é considerada íntegra se o resultado for igual ao rótulo de autenticação recebido.

5.4.5 Servidores

Programas funcionais podem ser visualizados como uma série de transformações em dados [32]. Esse raciocínio facilita o entendimento acerca da arquitetura dos dois servidores desenvolvidos, que foram construídos em cima da mesma arquitetura, ilustrada na Figura 5.4. O diagrama sintetiza um sistema que recebe como entrada um texto cifrado, no qual são aplicadas algumas transformações, para que no final se tenha uma resposta. Essas transformações são passíveis de falhas, que podem afetar a resposta final.

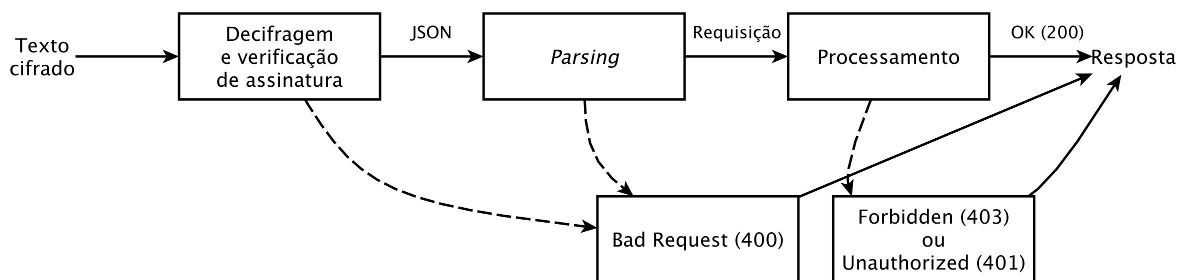

Figura 5.4: Arquitetura dos servidores.

Conforme mencionado nas seções anteriores, todas as mensagens do protocolo são cifradas e assinadas. Dessa forma, a requisição recebida pelos servidores consiste em uma sequência de *bytes* que, inicialmente, não contêm significado. Assim, o servidor precisa decifrar o texto cifrado, obtendo a mensagem original, e verificar sua assinatura. A mensagem original é uma *string* codificada no formato JSON, representando a solicitação do cliente. Deve-se, então, realizar o *parsing* dessa *string* para um tipo de dados algébrico, que represente a requisição feita e que pode ser manipulado a partir de código Haskell. Nesse momento, a requisição é passada para um *handler* para ser processada. Após o processamento, a resposta correspondente é gerada e, finalmente, enviada para o cliente.

As respostas possuem como metadado um código HTTP, que sintetiza o resultado do tratamento dado à requisição. A Tabela 5.1 apresenta os códigos HTTP com suas respectivas situações de uso no protocolo.

Tabela 5.1: Códigos HTTP no protocolo.

Código HTTP	Significado	Descrição	Utilização no Protocolo
400	*Bad Request*	A requisição não pode ser processada devido a problemas de sintaxe.	Utilizado pelo Servidor de Autenticação e Autorização e pelo Servidor de Fachada quando o *parsing* da mensagem para uma requisição do protocolo falha.
403	*Forbidden*	O acesso ao recurso é proibido.	Utilizado pelo Servidor de Autenticação e Autorização quando o cliente tenta responder um desafio que não existe, ou que já está expirado, ou que já foi respondido ou cuja resposta esteja errada e quando o cliente não possui permissões para acessar o serviço solicitado. Utilizado pelo Servidor de Fachada quando o usuário não tem permissão para acessar o serviço solicitado.
401	*Unauthorized*	Autenticação é possível, mas falhou.	Utilizado pelo Servidor de Fachada para redirecionar o cliente para o Servidor de Autenticação e Autorização quando o *token* de autorização apresentado está expirado.
200	OK	Resposta padrão para requisições HTTP processadas com sucesso.	Utilizado pelo Servidor de Autenticação e Autorização e pelo Servidor de Fachada quando o término do processamento de uma mensagem ocorre corretamente.

O fluxo de dados é o mesmo em ambos os servidores. O que difere de um para o outro são os tipos de requisição que são tratadas e, consequentemente, o processamento que é realizado, afetando, respectivamente, as etapas de *parsing* e processamento.

Os ADTs que representam as requisições e respostas em cada servidor foram definidos em módulos separados. No Servidor de Autenticação e Autorização, esses módulos são, respectivamente, `Messages.RqAuth` e `Messages.RespAuth`. Já no Servidor de Fachada, os módulos são `Messages.RqFacade` e `Messages.RespFacade`.

O processo de compilação em cada servidor deve resultar na geração de dois artefatos: o código executável do servidor e uma biblioteca que exponha uma interface de comunicação (os ADTs que representam as mensagens). Essa biblioteca é útil para facilitar a interação de outros projetos desenvolvidos em Haskell, como o módulo Cliente, com os servidores. Esse processo pode ser automatizado por meio da ferramenta `cabal` que, dentre outras, foi utilizada para gerência de *build*.

Snap Framework

A fim de facilitar o tratamento de requisições e respostas HTTP, foi necessário escolher um *framework* Web para o desenvolvimento do protótipo.

Os três principais *frameworks* Web Haskell são: o Snap Framework, o Yesod e o Happstack, cada um deles tendo particularidades quanto aos idiomas usados para o desenvol-

vimento de aplicações e *tradeoffs*. Existe o *tradeoff* de código estático, com segurança de tipos e determinado em tempo de compilação ou com menos segurança de tipos, mais flexível e determinado em tempo de execução. Há também o *tradeoff* entre a concisão de linguagens específicas de domínio (DSLs) e a expressividade e o poder de bibliotecas de *combinators*. A Tabela 5.2 sintetiza essas características.

Tabela 5.2: Matriz comparativa dos *frameworks* Web Haskell. Adaptado de [41].

	DSLs	*Combinators*
Algumas partes dinâmicas		Snap Framework
Tudo com segurança de tipos	Yesod	Happstack

Por sua simplicidade, alto desempenho, robustez e documentação abrangente, foi feita a opção pelo Snap Framework para o desenvolvimento do protótipo. Ele é completamente escrito em Haskell, possui um rápido servidor HTTP embarcado e provê uma API para manipulação de requisições e respostas HTTP no mesmo nível de abstração dos Java *servlets* [13]. O desenvolvimento com o Snap se dá em torno de um *monad* chamado `Snap`, que possibilita:

- Acesso com estado (*stateful*) a objetos que representam a requisição e a resposta HTTP corrente;
- Possibilidade de interromper o processamento de uma requisição retornando, imediatamente, uma resposta para o cliente;
- Encadeamento de *handlers*.

Além disso, o Snap possui um baixo custo de instalação e configuração e contempla um modo de desenvolvimento que possibilita a modificação do código em tempo de execução, sem que seja necessário reiniciar o servidor quando alterações forem efetuadas. Esse modo de desenvolvimento é similar ao mecanismo de *hot-deploy*, popular em plataformas de desenvolvimento imperativo, como servidores de aplicação Java EE, e agiliza significativamente o desenvolvimento.

A instalação do Snap Framework, que pode ser feita através da ferramenta `cabal` com o comando `cabal install snap` (e possível atualização da variável de ambiente `PATH`), contempla um executável utilitário chamado `snap`, utilizado para criação e configuração de um novo projeto. O comando `snap init` cria no diretório corrente uma estrutura de diretórios padrão, alguns arquivos de configuração e *build*, e três módulos Haskell: `Main.hs`, `Application.hs` e `Site.hs`.

O módulo `Main.hs` contém código para possibilitar o recarregamento dinâmico do sistema em modo de desenvolvimento e, geralmente, não deve ser alterado. O módulo `Application.hs` é primariamente utilizado para se definir um tipo de dados algébrico que contém o estado inicial de cada *thread*, ou seja, cada requisição recebida pelo servidor Snap é processada em uma *thread* independente que recebe uma cópia desse ADT, fazendo com que eventuais modificações de estado realizadas sobre ele só afete a *thread* corrente.

Caso seja necessário a utilização de um estado global de aplicação mutável, deve-se usar uma construção *thread-safe*, como um `IORef` ou `MVar`. Por fim, o módulo `Site.hs` é onde a lógica da aplicação deve ser definida, contendo uma função de inicialização que é executada, à princípio, somente durante o carregamento da aplicação; os *handlers*, que efetuarão o processamento das requisições; e a definição das rotas, que mapeiam URIs e métodos HTTP em *handlers*.

Habilitar o SSL é simples, sendo suficiente adicionar a *flag* de compilação `-fopenssl` durante a instalação do pacote `snap-server` (servidor HTTP embarcado do Snap Framework) e passar, no momento da execução do servidor, os diretórios do certificado digital e da chave privada RSA, ambos devendo estar no formato PEM. É possível configurar de forma dinâmica os números de porta HTTP e HTTPS, podendo ambos os protocolos estarem habilitados no mesmo servidor no mesmo instante de tempo. O protótipo pode ser configurado, por exemplo, para aceitar requisições HTTP locais (provenientes de máquinas na rede local) ou apenas requisições provenientes da interface de *loopback*, dentre outros.

Servidor de Autenticação e Autorização: `auth-server`

Esta seção apresenta o funcionamento interno do Servidor de Autenticação e Autorização. Alguns trechos de código Haskell são apresentados para exemplificar o processamento que é realizado em cada uma das etapas apresentadas na Figura 5.4.

O estado inicial das *threads*, definido no módulo `Application.hs`, é representado pelo seguinte ADT:

```
data App = App {
    _facadeServerURL :: URI,
    _maybeContract   :: Maybe Contract
}
```

O construtor de valor `App` possui dois campos. O primeiro, `_facadeServerURL`, contém uma URI para o Servidor de Fachada, utilizada após as etapas de autenticação e autorização, na última mensagem do protocolo, para redirecionar o cliente. Já o segundo, `_maybeContract`, é um campo utilitário, usado como forma de compartilhar o suposto contrato, que está realizando a requisição corrente, com os diferentes *handlers* que processam a requisição de forma fácil. Ocorre que acessar o contrato é uma tarefa recorrente durante o processamento das requisições realizadas ao Servidor de Autenticação e Autorização. Diante disso, tem-se duas outras alternativas: acessar o banco de dados todas as vezes que o contrato for necessário, o que é custoso; passar o contrato corrente como argumento para todas as funções que necessitarem (ou não) utilizá-lo, o que aumenta a complexidade e dificulta a legibilidade.

O ADT `App` é construído uma vez durante o carregamento da aplicação pela função de inicialização definida no módulo `Site.hs`. Essa função lê um arquivo de configuração, `devel.cfg`, organizado como uma lista de tuplas `<chave> = <valor>`, e preenche o campo `_facadeServerURL` com a URI especificada para o Servidor de Fachada. Esse valor se mantém constante entre as *threads* durante todo o ciclo de vida da aplicação, a menos que seja feita uma reinicialização forçada. É possível forçar uma reinicialização na aplicação, sem a necessidade de se reiniciar o servidor, acessando da máquina local

o caminho `/admin/reload`. Essa chamada reexecuta a função de inicialização e permite que a URI para o Servidor de Fachada seja alterada modificando-a no arquivo `devel.cfg`.

O campo `_maybeContract` é inicializado com o valor `Nothing`, uma vez que, inicialmente, não se sabe qual é o contrato que está realizando a requisição e nem se ele existe. Assim que o *handler* obtém o contrato `c`, durante o tratamento da requisição, consultando o banco de dados com o UUID recebido, o valor de `_maybeContract` é reconfigurado para `Just c`. É importante frisar que cada *thread* receberá uma cópia de `App` tal como ele foi inicializado e que trocar o valor de qualquer um dos campos durante o processamento não terá reflexos em outras *threads* ou requisições futuras.

Os nomes dos campos de `App` são precedidos com um *underscore* (`_`) por convenção, porque o Snap utiliza *lenses* [30] para derivar automaticamente acessores para `App`. *Lenses* são espécies de "*getters and setters* composicionais" e permitem o acesso e manipulação do estado de forma extremamente modular.

No que se refere aos *handlers*, procurou-se simular uma arquitetura conhecida na literatura, sobretudo de *frameworks Model-View-Controller* (MVC), como *front-controller*. A ideia básica é que se tenha um componente responsável por receber todas as requisições, realizar algum tipo de processamento prévio comum e, então, encaminhá-la para o componente adequado para que seja tratada. Como não existe a noção de *controllers* no Snap, sugere-se que essa arquitetura seja caracterizada como *front-handler* ou *dispatcher-handler*. Diante disso, a configuração das rotas é trivial, isto é, todas as requisições são mapeadas para o mesmo *handler*, sendo definida no Snap como:

```
routes :: [(ByteString, AppHandler ())]
routes = [ ("/", auth) ]
```

No caso do Servidor de Autenticação e Autorização, esse *front-handler* foi denominado `auth`, e é definido como:

```
auth :: AppHandler ()
auth = do
    rq <- getRequest
    case getHeader "JWT" rq of
        Just jwtCompact -> do
            rqAuth <- fromJWT jwtCompact
            maybe badRequest (sendResponse <=< handlerRqAuth) rqAuth
        _ -> badRequest
```

O *handler* `auth` é uma função que explicita a expressividade de Haskell, isto é, como é possível codificar uma grande quantidade de operações a partir de um código sucinto. O código está no *monad* Snap e, por isso, é possível utilizar a notação `do`, evidenciando o estilo imperativo do código.

A função `getRequest` retorna em `rq` o objeto que representa a requisição corrente. Em seguida, através da função `getHeader`, procura-se no cabeçalho HTTP pelo campo nomeado "JWT". Caso esse campo exista, seu conteúdo deve ser um *token* JWT na forma compacta, e a função `getHeader` retornará uma *string* [5] dentro do construtor `Just`. Caso

[5] Na verdade, será retornado um tipo `ByteString`, que é mais eficiente.

contrário, o processamento é encerrado e uma mensagem de erro *Bad Request* é retornada para o usuário.

Nesse momento, passa-se o texto cifrado, contido em `jwtCompact`, para a função `fromJWT`, onde ocorrem as etapas de decifragem, verificação da assinatura e *parsing* da requisição. Essa etapa corresponde aos dois primeiros blocos apresentados na Figura 5.4. Assim, a função `fromJWT` tem a seguinte assinatura:

```
fromJWT :: ByteString -> AppHandler (Maybe RqAuth)
```

Em Haskell, a assinatura de uma função diz muito sobre seu comportamento. Apenas olhando para a assinatura de `fromJWT`, sabe-se que ela recebe um *array* de *bytes* e, possivelmente, retorna um `RqAuth`. `RqAuth` é um ADT utores de valor representam todos os tipos de requisição tratadas pelo Servidor de Autenticação e Autorização. A Figura 5.2 mostra que existem duas mensagens que o cliente pode mandar para o Servidor de Autenticação e Autorização: uma solicitação de autenticação (mensagem 1) e a resposta do desafio de autenticação gerado (mensagem 3). Dessa forma, `RqAuth` é definido como:

```
data RqAuth =
    RqAuth01 {
        contractUUID :: UUID,
        sentAt       :: UTCTime
    } | RqAuth02 {
        challengeUUID :: UUID,
        contractUUID  :: UUID,
        serviceUUID   :: UUID,
        credential    :: ByteString,
        sentAt        :: UTCTime
    }
```

Para solicitar autenticação (requisição `RqAuth01`), o cliente só precisa informar o seu identificador do contrato (`contractUUID`). Ao responder um desafio (requisição `RqAuth02`), o cliente precisa informar o identificador do desafio que está respondendo (`challengeUUID`), seu código de contrato (`contractUUID`), o código do serviço que deseja consumir (`serviceUUID`) e a resposta do desafio (`credential`). O campo `sentAt` em ambas as requisições é um selo temporal utilizado como um `nonce`.

Como as mensagens são codificadas no formato JSON, é necessário especificar funções para realizar o *parsing* de uma *string* representando um objeto JSON para o tipo `RqAuth`. Isso pode ser feito de forma trivial com a biblioteca `aeson` [6], que é otimizada para facilidade de uso e alto desempenho. Ela provê duas *typeclasses*: `FromJSON` e `ToJSON`, além de uma série de *combinators* para auxiliar na definição dos *parsers*. O código a seguir ilustra como podemos tornar `RqAuth` uma instância de `FromJSON`:

```
instance FromJSON RqAuth where
    parseJSON (Object v) =
        RqAuth02 <$> v .: "challengeUUID"
                 <*> v .: "contractUUID"
```

[6] http://hackage.haskell.org/package/aeson-0.7.0.6

```
                 <*> v .: "serviceUUID"
                 <*> v .: "credential"
                 <*> v .: "sentAt"
        <|> RqAuth01 <$> v .: "contractUUID"
                     <*> v .: "sentAt"
    parseJSON _ = mzero
```

A *typeclass* `FromJSON` define a função `parseJSON`. A *string* recebida codificará um objeto JSON e, portanto, para outras estruturas de dados, ocorrerá o casamento de padrões com a segunda definição de `parseJSON` e o *parser* falhará (`mzero`). Se for um objeto JSON, a primeira definição será utilizada. Essa definição utiliza o *combinator* (`.:`) para extrair os valores da *string* JSON e utiliza a notação de *applicative functors* para tentar instanciar um dos construtores de `RqAuth`. É importante notar o operador de alternativa (`<|>`), ou seja, primeiro tenta-se construir um `RqAuth02` e, se o *parsing* falhar, tenta-se o `RqAuth01`.

Em muitos casos, a ordem dos construtores não é relevante para o resultado do *parsing*, porém, nesse caso específico, em que ambas as requisições possuem campos com o mesmo nome — “contractUUID” e “sentAt”, é necessário colocar a mais específica (que contenha maior quantidade de campos) antes. Isso ocorre porque, por padrão, a `aeson` está configurada para não resultar em erros caso haja campos sobrando, fazendo com que mensagens que deveriam ser transformadas em `RqAuth02` virem `RqAuth01`, quando a ordem é invertida.

Voltando para a função `fromJWT`, ela internamente decifra a mensagem utilizando a chave privada do servidor e, para verificar a assinatura, recupera a chave pública do contrato consultando o banco de dados utilizando o identificador do contrato recebido. As funções utilizadas para se realizar essas operações foram descritas na Seção 5.4.2. Detalhes de como os dados são armazenados no banco de dados serão apresentados na Seção 5.4.6. Se tudo estiver certo, um ADT `RqAuth` é retornado para `auth` dentro de um `Just`. Caso contrário, `Nothing` é retornado.

O resultado de `fromJWT` é então checado por meio do *combinator* `maybe`, que recebe três parâmetros: a ação a ser tomada caso o valor do terceiro parâmetro seja `Nothing`; a ação a ser tomada caso o valor do terceiro parâmetro seja `Just <valor>`; e o valor do tipo `Maybe`. Conclui-se assim que, se `fromJWT` falhar (retornar `Nothing`), o processamento é interrompido e retorna-se *Bad Request* para o cliente. Caso contrário, a requisição é passada para o *handler* `handlerRqAuth` para processamento e, em seguida, para o *handler* `sendResponse` para enviar a resposta para o cliente.

Observe que independentemente do tipo de requisição retornada por `fromJWT`, o mesmo *handler* (`handlerRqAuth`) é chamado. É de se esperar, no entanto, que requisições diferentes tenham tratamentos diferentes. É nesse ponto que é utilizado *casamento de padrões*. A função `handlerRqAuth` é definida como:

```
handlerRqAuth :: RqAuth -> AppHandler RespAuth
handlerRqAuth (RqAuth01 contractUUID _) = ...
handlerRqAuth (RqAuth02 challengeUUID contractUUID
    serviceUUID credential _) = ...
```

Com casamento de padrões, é possível determinar de forma elegante diferentes implementações para a mesma função, de acordo com os argumentos recebidos. Além disso,

através de composição de funções é possível expressar o comportamento das funções de forma transparente, concentrando-se em definir o problema de forma declarativa e utilizando o sistema de tipos para validar a solução. Abaixo é apresentada uma definição mais completa de `handlerRqAuth`:

```
handlerRqAuth :: RqAuth -> AppHandler RespAuth
handlerRqAuth (RqAuth01 contractUUID _) =
    generateChallenge >>= makeResponse

handlerRqAuth (RqAuth02 challengeUUID contractUUID
    serviceUUID credential _) =
    verifyChallenge >>= verifyPermissions >> emitAuthToken >>= makeResponse
```

Esse padrão de desenvolvimento, que consiste na decomposição de um problema em funções pequenas, com poucas responsabilidades, que possam ser testadas individualmente e depois combinadas para formar uma solução completa é o que define programação funcional [22]. Esse nível de modularidade permite que o desenvolvedor se concentre nas pequenas partes que compõem o programa. Isso, associado ao sistema de tipos forte de Haskell, facilita o raciocínio sobre os problemas computacionais e, sem dúvida, acarreta no desenvolvimento de programas com menos *bugs*.

No primeiro `handlerRqAuth`, a função `generateChallenge` é responsável por gerar um desafio de autenticação, sorteando aleatoriamente uma das credenciais do contrato identificado por `contractUUID`, adicionando um tempo de expiração, e gravando-a no banco de dados.

Já no segundo, a função `verifyChallenge` verifica o desafio de autenticação identificado por `challengeUUID`, considerando o tempo de expiração, corretude da resposta (`credential`), entre outros, e retorna um contrato. Verifica-se, então, com a função `verifyPermissions`, se esse contrato tem permissão para acessar o serviço solicitado, identificado por `serviceUUID`. Caso não tenha permissões, o processamento é encerrado internamente na função `verifyPermissions` que retorna *Forbidden* para o cliente. Caso contrário, a função `emitAuthToken` é executada. Ela consulta o banco de dados por um *token* de autorização para aquele serviço que ainda esteja válido e, se não existir, gera um novo.

Em ambos os casos, a função `makeResponse` constrói um ADT `RespAuth` que representa a resposta a ser enviada para o cliente: um desafio, no primeiro caso; ou um token de autorização de acesso a serviço, no segundo caso. Essa resposta é, então, passada para a função `sendResponse`, que a serializa para uma mensagem JSON, assinada com a chave privada do servidor e cifrada com a chave pública do contrato atual.

O código abaixo apresenta o ADT `RespAuth`, definido no módulo `Messages.RespAuth`, que representa as mensagens 2 e 4 da Figura 5.2:

```
data RespAuth =
    RespAuth01 {
        replyTo        :: URI,
        credentialCode :: Int64,
        challengeUUID  :: UUID,
        expiresAt      :: UTCTime
```

```
    } | RespAuth02 {
        replyTo            :: URI,
        authorizationToken :: ByteString,
        expiresAt          :: UTCTime
    }
```

O campo `replyTo` em ambas as respostas contém a URI para a qual o cliente deve enviar a próxima mensagem. No caso do protótipo, em que foram utilizados apenas dois servidores — um de autenticação e autorização e outro de fachada, esse campo não é muito relevante. No entanto, ele foi adicionado com fins de extensibilidade, para cobrir o caso em que se tenha mais servidores, sendo útil para implementar, por exemplo, políticas de balanceamento de carga.

Os demais campos de `RespAuth01` — `credentialCode`, `challengeUUID` e `expiresAt`, contém, respectivamente, um código indexador de credenciais, o identificador do desafio corrente e um selo temporal de expiração, representando o limite de tempo máximo que o cliente tem para responder o desafio. Ao receber essa mensagem, o cliente deve consultar a credencial indexada por `credentialCode` e enviá-la para o Servidor de Autenticação e Autorização.

Já `RespAuth02` contém uma sequência de *bytes* que consiste no *token* de autorização temporário (campo `authorizationToken`) e o tempo de validade desse *token*, representado por `expiresAt`. Futuras requisições de acesso a serviços realizadas ao Servidor de Fachada deverão conter o valor de `authorizationToken`.

Servidor de Fachada: `facade-server`

O funcionamento do Servidor de Fachada é bastante similar ao funcionamento do Servidor de Autenticação e Autorização apresentado na Seção 5.4.5.

O estado inicial das *threads*, definido no módulo `Application.hs`, é representado pelo seguinte ADT:

```
data App = App {
    _authServerURL :: URI,
    _maybeContract :: Maybe Contract
}
```

O campo `_authServerURL` contém a URI para o Servidor de Autenticação e Autorização e é utilizado para redirecionar o cliente quando o *token* de autorização temporário está expirado. O campo `_maybeContract` possui o mesmo comportamento já explicado.

O *front-handler* utilizado foi denominado `facade`, e é definido como:

```
facade :: AppHandler ()
facade = do
    rq <- getRequest
    case getHeader "JWT" rq of
        Just jwtCompact -> do
            rqFacade <- fromJWT jwtCompact
            maybe badRequest handlerRqFacade rqFacade
        _ -> badRequest
```

Observe que o *handler* `facade` é muito parecido com o *handler* `auth`, exceto por não ter a função `sendResponse` encadeada ao *handler* que processará a requisição. Isso ocorre porque o Servidor de Fachada é mais simples que o Servidor de Autenticação e Autorização, possuindo apenas uma requisição possível — a de solicitação de acesso a serviço, representada pela mensagem 6 na Figura 5.2, e uma resposta possível — redirecionamento para o Servidor de Autenticação e Autorização. Dessa forma, simplificou-se a implementação para que o próprio *handler* `handlerRqFacade` emitisse a resposta.

O código abaixo apresenta o ADT `RqFacade`:

```
data RqFacade = RqFacade01 {
    contractUUID       :: UUID,
    authorizationToken :: ByteString
}
```

A requisição `RqFacade01` contém dois campos — `contractUUID` e `authorizationToken`, representando, respectivamente, o identificador do contrato que está solicitando o serviço e o *token* de autorização temporário de acesso ao serviço. Nota-se que não existe nenhum campo que represente explicitamente qual o serviço que se deseja consumir. Diante disso, existem duas possibilidades:

1. Considerar que o serviço está implícito no *token* de autorização, uma vez que, a lógica atual do protocolo associa por meio do *token*, um e somente um contrato a um e somente um serviço. Dessa forma, é possível consultar o banco de dados utilizando o valor do *token* de autorização temporário e obter as informações do serviço correspondente.

   Essa técnica, por sua vez, não é extensível, uma vez que mudanças futuras, como a alteração da lógica para possibilitar que o mesmo *token* de autorização dê acesso a mais de um serviço, podem ocasionar em mudanças no código-fonte da aplicação e alterações nas mensagens do protocolo, gerando incompatibilidades. Nesse caso, por exemplo, a requisição `RqFacade01` teria que ser alterada para que fosse possível determinar o serviço solicitado.

2. Especificar o serviço por um canal alternativo, utilizando de forma mais rica a requisição HTTP que é enviada ao Servidor de Fachada. Seguindo a filosofia REST, poderia-se utilizar a própria URI e o método HTTP associado. Considere uma requisição da forma:

   ```
   GET https://<caminho até o Servidor de Fachada>/
       <identificador do serviço>?<param1>&<param2>&<paramN>.
   ```

   O Servidor de Fachada pode, então, interpretar essa requisição, recuperando do banco de dados o endereço IP e número de porta da máquina que provê o serviço identificado por `<identificador do serviço>` e realizar uma requisição replicando o método e os parâmetros recebidos, ou seja, `GET` e os parâmetros *query*: `<param1>`, `<param2>` e `<paramN>`.

Certa inteligência é, no entanto, necessária no lado do servidor, ou seja, se o método HTTP for um `POST`, é possível que argumentos sejam passados tanto como parâmetros *query* na URL quanto no corpo da requisição, sendo necessário tratamentos especiais dependendo do método utilizado.

Apesar da flexibilidade dessa técnica, existe um prejuízo no que se refere à segurança, ou seja, as informações referentes ao serviço que se está acessando estarão protegidas somente com o SSL, enquanto os dados que trafegam encapsulados no JWT contam com uma camada adicional de proteção.

Diante do exposto, a segunda opção foi escolhida e implementada na versão atual do protótipo. No entanto, a opção de se alterar a mensagem `RqFacade01` para conter as informações referentes ao serviço deve ser avaliada. Sugere-se o seguinte formato:

```
type Params = Map ByteString [ByteString]

data RqFacade = RqFacade01 {
    contractUUID       :: UUID,
    authorizationToken :: ByteString,
    serviceUUID        :: UUID,
    httpMethod         :: Method,
    serviceParams      :: Params
}
```

Uma alternativa, que manteria inclusive a compatibilidade com a versão atual, seria manter as duas opções de especificação de serviço, tornando opcionais os novos campos:

```
data RqService = RqService {
    serviceUUID   :: UUID,
    httpMethod    :: Method,
    serviceParams :: Params
}

data RqFacade = RqFacade01 {
    contractUUID       :: UUID,
    authorizationToken :: ByteString,
    service            :: Maybe RqService
}
```

O *handler* `handlerRqFacade` é definido como:

```
handlerRqFacade :: RqFacade -> AppHandler ()
handlerRqFacade (RqFacade01 contractUUID authToken) =
    (allow >>= proxify) <|> redirectToAuthServer
```

O entendimento dessa função pode ser dividido em duas partes, dado o operador de alternativa (`<|>`), ou seja, tem-se duas expressões: `allow >>= proxify` e `redirectToAuthServer`.

A primeira executa a função `allow`, que verifica a validade do *token* de autorização recebido, extrai o identificador do serviço solicitado da URL e verifica se o contrato possui

permissões para acessá-lo. Se algum erro ocorrer, a função `allow` falha chamando a função `pass`, que interrompe imediamente o processamento atual e passa o controle para a função `redirectToAuthServer`. Caso contrário, a função `allow` retorna um serviço para a função `proxify`, que será responsável por intermediar a comunicação entre o cliente e o serviço.

Nesse momento, o Servidor de Fachada atua, de fato, como um *proxy*, fazendo uma requisição à máquina que provê o serviço e retornando a resposta recebida para o cliente. É importante frisar que esse processo é opaco para o cliente, isto é, para todos os efeitos, é como se o serviço estivesse implementado diretamente no Servidor de Fachada. Metadados expostos no cabeçalho HTTP que possam revelar informações acerca da plataforma ou do serviço provido são filtrados, ou seja, ao acessar um serviço sendo executado, por exemplo, sob o Servidor Apache, é comum receber na resposta um cabeçalho "Server" com a descrição do servidor, versão, entre outros.

A função `redirectToAuthServer` simplesmente redireciona o cliente para o Servidor de Autenticação e Autorização. Para isso, é utilizado o ADT `RespFacade` definido no módulo `Messages.RespFacade`:

```
data RespFacade = RespFacade01 {
    replyTo :: URI
}
```

A resposta `RespFacade01` representa um redirecionamento e o campo `replyTo` contém a URI que deve ser seguida pelo cliente. Dada a simplicidade dessa operação e a existência de um padrão dentro do protocolo HTTP para se fazer redirecionamentos — poderia-se adicionar a URI no cabeçalho `Location` associado ao código adequado na resposta HTTP, utilizar um JWT para realizá-la pode parecer um *overhead*. No entanto, optou-se por esse tipo de implementação para manter a regularidade no protocolo, isto é, todas as negociações (requisições e respostas) devem estar presentes no campo "JWT".

API utilitária: `server-common`

Pode-se perceber que o Servidor de Autenticação e o Servidor de Fachada possuem muito em comum. Com o objetivo de minimizar a repetição de código entre os projetos `facade-server` e `auth-server`, criou-se a API `server-common`, contendo as definições das entidades utilizadas no sistema, funções utilitárias (manipulação de Base64, JSON, respostas HTTP) e código de acesso ao banco de dados.

O código abaixo apresenta a entidade `Token`, definida no módulo `Model.Token`, que modela um *token* de autorização temporário:

```
data Token = Token {
    uuid           :: Maybe UUID,
    revision       :: Maybe ByteString,
    value          :: ByteString,
    contractUUID   :: UUID,
    serviceUUID    :: UUID,
    allowedMethods :: [Method],
    expiresAt      :: UTCTime
}
```

Um *token* de autorização temporário é uma entidade associativa que associa um contrato a um serviço por meio de um relacionamento de permissão de acesso a serviço. Essa relacionamento é representando pelos campos `contractUUID` e `serviceUUID`. Além disso, um *token* tem um valor, representado como uma sequência de *bytes* pelo campo `value`, e um selo temporal de expiração, representado pelo campo `expiresAt`. O campo `allowedMethods` contém uma lista de métodos HTTP, que permite especificar em um nível ainda mais granular as permissões de um contrato em um serviço, ou seja, se determinado contrato tiver somente permissão de leitura, é natural que o valor de `allowedMethods` seja `[GET]`. Os campos `uuid` e `revision` são opcionais e são utilizados pelo CouchDB para identificação e controle de versão dos documentos.

Um detalhe importante sobre o módulo `Model.Token` é que ele não exporta o construtor de valor `Token`. O efeito obtido é o de encapsulamento, similar ao da utilização do padrão de projeto *Factory* [16] em linguagens orientadas a objetos. Para se instanciar um `token` é necessário chamar uma função construtora — `new`, definida como:

```
new :: ByteString -> UUID -> UUID -> [Method] -> UTCTime -> Token
new = Token Nothing Nothing
```

A função `new` aplica parcialmente o construtor de valor `Token` atribuindo aos campos `uuid` e `revision` o valor `Nothing`. Isso é feito porque os valores desses campos são atribuídos durante a gravação do objeto no banco de dados, não sendo relevantes para o desenvolvedor.

Por fim, outro detalhe importante é como a entidade `Token` é uma instância de `ToJSON`:

```
instance ToJSON Token where
    toJSON (Token _ _ v cu su am ea) =
        object [ "type"           .= ("token" :: ByteString)
               , "value"          .= v
               , "contractUUID"   .= cu
               , "serviceUUID"    .= su
               , "allowedMethods" .= am
               , "expiresAt"      .= ea ]
```

Observa-se que os dois primeiros campos de `Token`: `uuid` e `revision` — são ignorados. Isso ocorre porque pelo mesmo motivo explicado acima, ou seja, esses valores são passados de forma alternativa durante a gravação da entidade no banco de dados. Além disso, é adicionado um novo campo: `type` — com o valor constante “token”. Mais detalhes sobre esse campo serão apresentados na próxima seção, mas essa *string* é importante para que se possa diferenciar os tipos dos documentos armazenados no banco de dados, que serão todos representados como *strings* JSON.

5.4.6 Banco de Dados: Apache CouchDB

Conforme apresentado na Figura 5.1, no protótipo desenvolvido o banco de dados atua como interface entre o Servidor de Autenticação e Autorização e o Servidor de Fachada, ou seja, durante as etapas de autenticação e autorização, informações sobre o *status* do processo são gravadas no banco de dados. Essas informações são, posteriormente,

consultadas pelo Servidor de Fachada para conceder ou recusar o acesso a determinado serviço.

Escolheu-se o banco de dados Apache CouchDB [7] para a implementação do protótipo. O CouchDB é um banco de dados não-relacional, implementado em uma linguagem funcional (Erlang [8]), baseado em REST e que utiliza o formato JSON para os documentos. Isso significa que, diferentemente de bancos de dados relacionais, o CouchDB é orientado a documentos e *schema-free*, além de possuir um motor de busca extremamente eficiente para o processamento dos dados e um *design* voltado para a modularização e escalabilidade [3].

O CouchDB opera, primariamente, com dados autocontidos que tem poucos relacionamentos, através de uma interface Web RESTful. Foi desenvolvido para lidar com tráfego variante, ou seja, a ocorrência de picos na quantidade de requisições fará com que o CouchDB absorva as várias requisições concorrentes sem cair. É possível que mais tempo seja demandado para responder cada requisição, porém, todas serão eventualmente respondidas. Quando o pico acabar, o CouchDB voltará ao seu funcionamento usual.

O modelo de persistência utilizado pelo CouchDB se refletiu em ganhos de produtividade durante o desenvolvimento do protótipo, pois, mudanças nas entidades a serem persistidas não se traduziam em mudanças no *schema* do banco de dados, facilitando a experimentação e tornando o desenvolvimento mais ágil. Além disso, a arquitetura do CouchDB foi ao encontro da arquitetura do protocolo, isto é, a utilização de um modelo baseado em REST e a utilização do formato JSON.

O *design* do CouchDB é fortemente baseado na arquitetura da Web e nos conceitos de recursos, métodos e representações, aprimorando-a com técnicas poderosas de se consultar, mapear, combinar e filtrar os dados [3]. Essas técnicas são sintetizadas no mecanismo conhecido como *Map-Reduce*.

Views Criadas

A principal ferramenta de consulta no CouchDB são as chamadas *views*. As *views* são úteis para diferentes propósitos [3]:

- Filtrar os documentos no banco de dados para encontrar aqueles relevantes para um processo em particular.
- Extrair dados dos documentos e apresentá-los em uma ordem específica.
- Construir índices eficientes para encontrar documentos por qualquer valor ou estrutura de dados que eles contenham.
- Usar esses índices para representar relacionamentos entre os documentos.
- Executar qualquer tipo de computação sobre os dados nos documentos.

Views são usualmente especificadas através de código Javascript e armazenadas dentro de documentos especiais denominados *design documents*. As seguintes entidades são armazenadas no banco de dados: contratos, serviços, desafios e *tokens* de autorização. Dessa forma, foi necessária a criação de algumas *views* para acessar essas entidades.

[7] http://couchdb.apache.org/.

[8] http://www.erlang.org/.

O código abaixo apresenta a *view* `listAvailableTokens`, utilizada para se recuperar *tokens* de autorização que ainda estejam válidos para um contrato:

```
function(doc) {
    if (doc.type && doc.type == 'token') {
        emit([doc.contractUUID, doc.serviceUUID, doc.expiresAt], doc);
    }
}
```

O código representa uma função `map`, ou seja, uma função que recebe apenas um parâmetro — um documento, e que é aplicada uma vez em cada documento do banco de dados. Como os documentos no banco de dados são representados apenas como uma coleção de objetos JSON, quando se tem várias entidades, é necessário adicionar um campo adicional para identificar o tipo daquele documento. Essa é a função do campo `type`, que foi adicionado em cada entidade persistida. A função `emit` recebe dois parâmetros: uma *chave* e um *valor*. Cada vez que `emit` é chamada, uma nova entrada é adicionada em uma lista de objetos resultante, que representa a *view*. Nesse caso, tem-se uma chave composta (representada pelo vetor `[doc.contractUUID, doc.serviceUUID, doc.expiresAt]`) e o valor é o próprio documento.

Dessa forma, a função `listAvailableTokens` filtra os documentos do tipo `token` e provê uma estrutura de dados de consulta indexada pelo identificador do contrato, o identificador do serviço e a data de expiração do *token*. Ainda, por padrão, essa tabela é ordenada de forma crescente, fazendo com que os *tokens* emitidos para um contrato acessar serviços fiquem agrupados e ordenados pela data de expiração. Assim, para se obter o *token* de autorização mais recente emitido para um contrato acessar um serviço, basta consultar essa tabela de forma decrescente passando-se os identificadores do contrato e do serviço e limitando o resultado para retornar apenas um objeto.

É exatamente assim que a função `findAvailableToken` é definida no módulo `CouchDB.DBToken`. O *token* mais recente é recuperado e verifica-se se ele está expirado ou não, retornando `Nothing` ou `Just <token>` para a função chamadora. Utiliza-se a API Haskell `couchdb-conduit` para interfaceamento com o CouchDB:

```
findAvailableToken :: UUID -> UUID -> IO (Maybe Token)
findAvailableToken contractUUID serviceUUID = do
    tokens <- runCouch conn $
        let uuids = fmap toByteString [contractUUID, serviceUUID]
        in couchView_ dbName "token" "listAvailableTokens"
            [ ("endkey", Just $ encodeKeys uuids)
            , ("descending", Just "true")
            , ("limit", Just "1")
            ]
            $ rowValue =$= toType =$ CL.consume
    now <- getCurrentTime
    return $ listToMaybe $ filter ((> now) . expiresAt) tokens
```

Por fim, as últimas *views* criadas foram `listTokensWithContractUUID` e `listChallengesWithContractUUID`. A primeira indexa o *token* de autorização pelo identificador do contrato e o valor do *token*, sendo utilizada para verificar se o *token* apresentado foi emitido para aquele contrato. Já a segunda, indexa um desafio por seu

identificador e pelo identificador do contrato, sendo utilizada para recuperar os desafios que foram realizados para um contrato. Elas são apresentadas abaixo:

```
// listTokensWithContractUUID
function(doc) {
  if (doc.type && doc.type == 'token') {
      emit([doc.contractUUID, doc.value], doc);
  }
}

// listChallengesWithContractUUID
function(doc) {
  if (doc.type && doc.type == 'challenge') {
      emit([doc._id, doc.contractUUID], doc);
  }
}
```

5.4.7 Módulo Cliente: `rest-client`

O módulo Cliente, ou simplesmente Cliente, é um programa de computador a ser executado no cliente para automatização dos passos do protocolo. Uma instância do `rest-client`, ou seja, um processo `rest-client`, deve ser executado por contrato. Dessa forma, dentro de uma organização, como o MPU ou o TCDF, pode-se executá-lo em uma máquina *proxy* e vários usuários podem ter acesso aos serviços providos através do mesmo cliente.

Do ponto de vista técnico, o cliente é um servidor e também foi desenvolvido em cima do Snap Framework. Para um melhor entendimento de seu comportamento, apresenta-se o ADT definido no módulo `Application.hs`, que define o estado inicial recebido por cada *thread*:

```
data App = App {
    _contract     :: Contract,
    _activeTokens :: MVar [Token],
    _authURL      :: String,
    _facadeURL    :: String,
    _httpMngr     :: Manager
}
```

Todos esses campos são inicializados pela função de inicialização. O campo `_contract` é configurado com os dados do contrato que a instância do cliente representa. Esses dados são carregados a partir do arquivo `contract.json` na pasta `resources`, cujas permissões de acesso devem ser controladas por meio do sistema operacional e serem, preferencialmente, configuradas do modo mais restritivo possível que dê acesso de leitura para o processo `rest-client`. Esse arquivo está codificado no formato JSON e contém dados sensíveis, dentre outros, a lista de credenciais do contrato. Para aprimorar a segurança, esse arquivo deveria ser, idealmente, cifrado. No entanto, para efeitos de protótipo, isso não foi implementado.

O campo `_activeTokens` mantém em memória uma lista de *tokens* de autorização e, diferentemente dos demais campos, funciona como uma espécie de variável global mutável. Isso significa que `_activeTokens` é acessada de forma concorrente por todas as *threads* e que eventuais modificações afetam toda a aplicação. A ideia dessa variável é criar um *cache* de *tokens* de autorização de acesso a serviço válidos para minimizar o número de requisições ao Servidor de Autenticação e Autorização, aumentando a vazão e o desempenho da aplicação.

Os campos `_authURL` e `_facadeURL` contém, respectivamente, os caminhos para o Servidor de Autenticação e Autorização e para o Servidor de Fachada. Por fim, o campo `_httpMngr` é um ADT utilizado para se realizar requisições HTTP a partir de código Haskell. Ele foi colocado em `App` também por questões de desempenho, uma vez que instanciá-lo é uma operação custosa. Dessa forma, é necessário instanciá-lo apenas uma vez, na inicialização da aplicação, e reusá-lo entre as *threads*, simulando mais uma vez um dos padrões de projeto definido por [16]: o *Singleton*.

Algo interessante sobre o cliente é que a interface que ele provê é também REST, ou seja, quando um usuário deseja consumir um serviço ele deve acessar o navegador ou algum cliente HTTP e submeter um requisição da forma:

```
GET https://<caminho até o Cliente>/client/
    <método HTTP>/<identificador do serviço>?<parâmetros>
```

A partir desses dados, o cliente deverá executar os procedimentos do protocolo para, ao final, retornar a resposta solicitada para o usuário. Para isso, o cliente começa extraindo da URL o identificador do serviço e verificando localmente, na lista de *tokens* ativos, se ele já possui um *token* de autorização ainda válido para acessar o serviço. Em caso afirmativo, a etapa de autenticação e autorização pode ser pulada e o cliente solicita o serviço diretamente ao Servidor de Fachada. Caso contrário, o fluxo apresentado na Figura 5.2 é executado.

O cliente precisa interagir tanto com o Servidor de Autenticação e Autorização quanto com o Servidor de Fachada. Para isso, ele importa as bibliotecas `facade-server` e `auth-server`, que contêm os ADTs Haskell que representam as mensagens do protocolo. Conforme apresentado em seções anteriores, as requisições para o Servidor de Autenticação e Autorização são definidas no ADT `RqAuth`, enquanto as requisições para o Servidor de Fachada são definidas no ADT `RqFacade`. O mesmo ocorre com as respostas, que são definidas, respectivamente, nos módulos `RespAuth` e `RespFacade`. No entanto, seria interessante se houvesse uma maneira de agrupar `RqAuth` e `RqFacade` como *requisições do protocolo*; e `RespAuth` e `RespFacade` como *respostas do protocolo* — tornando o tratamento dado às diferentes mensagens mais uniforme.

Como isso pode ser feito, considerando que os ADTs estão definidos em projetos separados? Em uma linguagem orientada a objetos, não existe uma maneira modular de se obter tal comportamento. Seria necessário, por exemplo:

1. Definir interfaces para especificar os comportamentos de objetos *requisição* e de objetos *resposta*.
2. Compartilhar essas interfaces entre os diferentes projetos.
3. Alterar as classes que definem as mensagens para implementar essas interfaces.

Felizmente, em Haskell, tem-se o conceito de *typeclasses* abertas [31]. O código abaixo ilustra como é possível agrupar as diferentes respostas utilizando a *typeclass* `Resp`:

```
class FromJSON a => Resp a
instance Resp RespFacade
instance Resp RespAuth
```

Observa-se que a *typeclass* `Resp` não define nenhuma operação. Ela é utilizada apenas para uniformizar os ADTs `RespFacade` e `RespAuth` perante o sistema de tipos. Além disso, é possível notar a restrição `FromJSON` na definição de `Resp`, que limita instâncias da *typeclass* `Resp` para apenas ADTs que já forem instâncias de `FromJSON`. Isso é útil para a definição da função `parseResponse`:

```
parseResponse :: (Resp a) => Response b -> IO a
```

A função `parseResponse` recebe uma resposta HTTP, proveniente da utilização do `_httpMngr`, e retorna dentro do *monad* `IO`, uma instância da *typeclass* `Resp` que pode ser, nesse caso, ou um `RespFacade` ou um `RespAuth`.

A utilização de *typeclasses* provê flexibilidade para adicionar comportamentos a ADTs de forma extremamente modular, sem que haja a necessidade de se alterar sua definição original. É possível agregar comportamentos, tornando um ADT uma instância de uma *typeclass*, em módulos diferentes daquele onde o ADT foi definido. Por essa característica, diz-se que Haskell possui extensibilidade nativa no *domínio de operações* [31].

5.5 Análise de Desempenho

Uma análise de desempenho do protótipo implementado foi realizada por [6], que dividiu a execução dos testes em dois estudos de caso. O primeiro utilizou como métrica de avaliação o tempo médio de resposta de um serviço que é amplamente utilizado pela PCDF e suas instituições parceiras. O tempo médio de resposta consiste no tempo despendido desde o momento da solicitação do serviço até a chegada da resposta na máquina do usuário. Já o segundo, visou a obtenção de resultados mais próximos da realidade vivenciada na PCDF, analisando o protocolo em um cenário extremo de utilização e considerando como métricas, o tempo médio de resposta e a vazão das requisições.

Os estudos de caso realizados evidenciaram que a utilização do protocolo proposto — com SSL/TLS e várias etapas de negociação até a concretização do acesso ao serviço, impactam de forma significativa o desempenho geral da aplicação. Contudo, mesmo com desempenho inferior, os tempos médios de resposta obtidos nos experimentos atendem aos requisitos de negócio da PCDF e estão dentro do esperado.

Por fim, é importante frisar que não foram realizadas otimizações de desempenho no código-fonte da aplicação que, a nível de protótipo, teve como objetivo a implementação correta da arquitetura e dos mecanismos de segurança especificados, a fim de validar o protocolo proposto.

Capítulo 6

Conclusão e Trabalhos Futuros

Este trabalho apresentou a utilização de programação funcional, mais especificamente, da linguagem Haskell, na implementação de um protocolo criptográfico para o provimento de serviços de forma segura em uma arquitetura orientada a serviços baseada em REST.

As características de Haskell, sobretudo, o sistema de tipos forte, a expressividade e o alto grau de modularidade, a colocam como uma alternativa interessante para o desenvolvimento de soluções, inclusive em nichos predominantemente imperativos, permitindo a rápida prototipação e a escrita de programas com menos *bugs*. No entanto, Haskell carece de um ambiente de desenvolvimento e ferramentas que facilitem o desenvolvimento de aplicações de médio ou grande porte. Além disso, enfrentou-se problemas referentes ao gerenciamento de dependências e portabilidade da aplicação, que se mostrou incompatível com versões diferentes da plataforma Haskell disponíveis para diferentes sistemas operacionais.

Como contribuições deste trabalho, tem-se uma implementação em *software* de um sistema em rede que atende aos requisitos de segurança e desempenho da PCDF e que pode ser generalizado e facilmente implantado em outros contextos, onde uma entidade deseja prover serviços de forma segura e escalável a parceiros conhecidos pela Internet. Mais especificamente, foram desenvolvidos: dois servidores — um de autenticação e autorização e outro de fachada, um módulo cliente baseado em REST para automatização dos passos do protocolo e uma biblioteca criptográfica minimalista, totalmente escrita em Haskell, implementando parcialmente a especificação JWT. Essa biblioteca, denominada `jwt-min`, é pioneira, uma vez que até o momento do desenvolvimento deste trabalho, não foram encontradas implementações estáveis ou minimamente funcionais em Haskell dessa especificação.

Como trabalhos futuros, propõe-se a realização de um processo de auditoria no código-fonte e nas APIs de terceiros utilizadas, tendo-se em vista aspectos de programação segura; melhorias na documentação e exportação para formatos mais acessíveis; otimizações de desempenho através da detecção de gargalos e *profiling* de código; e melhoria no empacotamento e distribuição dos artefatos de *software* desenvolvidos.

Por fim, todos os artefatos de *software* produzidos durante o desenvolvimento deste trabalho estão publicamente disponíveis no GitHub [1].

[1] `https://github.com/alexandrelucchesi/pfec/`.

Referências

[1] Algebraic data type. http://www.haskell.org/haskellwiki/Algebraic_data_type, July 2014. 23

[2] Typeclassopedia. http://www.haskell.org/haskellwiki/Typeclassopedia, July 2014. 34, 37

[3] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, Inc., 2010. 67

[4] Elisa Bertino, Lorenzo D. Martino, Federica Maria Francesca Paci, and Anna Squicciarini. *Security for Web Services and Service-Oriented Architectures*. Springer, 2010. 4

[5] G. Brassard. A note on the complexity of cryptography (corresp.). *IEEE Trans. Inf. Theor.*, 25(2):232–233, September 2006. 13

[6] R. A. da Conceição, R. B. Almeida, and E. D. Canedo. Arquitetura de referência orientada a serviços com foco em segurança. 2014. 1, 2, 4, 12, 38, 39, 51, 71

[7] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. 17

[8] Thomas Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007. 15, 16

[9] S. Even and Y. Yacobi. Cryptography and np-completeness. volume 293 of *Proceedings of the 7th International Colloquium on Automata, Languages, and Programming*, pages 195–207. Springer-Verlag, 1980. 13

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999. 20

[11] Roy Fielding. Rest apis must be hypertext-driven. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, October 2008. 18

[12] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887. 18, 19

[13] Snap Framework. Snap framework documentation. http://snapframework.com, March 2014. 56

[14] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004. 26

[15] Camille Furtado, Vinícios Pereira, Leonardo Azevedo, Fernanda Baião, and Flávia Santoro. Arquitetura orientada a serviço — conceituação. 2009. 15

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, 1994. 26, 66, 70

[17] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. 13

[18] Mikhail Glushenkov. An introduction to cabal sandboxes. `http://coldwa.st/e/blog/2013-08-20-Cabal-sandbox.html`, July 2014. 47

[19] Oded Goldreich. *Foundations of Cryptography: Basic Techniques*. Cambridge University Press, 2001. x, 13

[20] Joachim Grollmann and Alan L. Selman. Complexity measures for public-key cryptosystems. *SIAM J. Comput.*, 17(2):309–335, April 1988. 13

[21] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. 2

[22] John Hughes. Why functional programming matters. *The Computer Journal*, 32:98–107, 1984. 31, 61

[23] M. B. Jones. Json web algorithms (jwa) — draft-ietf-jose-json-web-algorithms-23. `http://tools.ietf.org/html/draft-ietf-jose-json-web-algorithms-23`, March 2014. 48, 52

[24] M. B. Jones, J. Bradley, and N. Sakimura. Json web signature (jws) — draft-ietf-jose-json-web-signature-23. `http://tools.ietf.org/html/draft-ietf-jose-json-web-signature-23`, March 2014. 48

[25] M. B. Jones, J. Bradley, and N. Sakimura. Json web token (jwt) — draft-ietf-oauth-json-web-token-18. `http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-18`, March 2014. 48

[26] M. B. Jones, E. Rescorla, and J. Hildebrand. Son web encryption (jwe) — draft-ietf-jose-json-web-encryption-23. `http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-23`, March 2014. 48

[27] Simon L Peyton Jones et al. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. 2

[28] Nicolai Josuttis. *Soa in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007. 15

[29] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, pages 161–191, 1883. 6

[30] Edward A. Kmett. Functional lenses. http://lens.github.io/, March 2014. 58

[31] Ralf Lämmel and Klaus Ostermann. Software extension and integration with type classes. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 161–170, New York, NY, USA, 2006. ACM. 26, 71

[32] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011. 25, 32, 34, 37, 54

[33] Tsutomu Matsumoto and Hideki Imai. On the key predistribution system: A practical solution to the key distribution problem. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 185–193. Springer, 1987. 7

[34] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008. 34

[35] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Criptography*. CRC Press, 1996. 4, 5, 6, 7, 9, 13, 14

[36] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008. 21, 22, 23, 25, 26, 27, 32, 36

[37] Pedro A. D. Rezende. Modelos de confiança para segurança em informática. 2012. 7

[38] Pedro Antonio Dourado Rezende. Segurança de dados – material e Áudio/vídeo-aulas. http://www.cic.unb.br/~rezende/segdados_files/audio-video/index.html, March 2014. x, 6, 8, 11, 12

[39] Bruno Cesar Dias Ribeiro. Implementação eficiente de algoritmos para teste de primalidade. 2013. x, 5, 6, 7, 8, 9

[40] Bruce Schneier. *Cryptography: Theory and Practice*. John Wiley & Sons, 2nd edition, 1996. 6, 7, 10, 11, 13, 14, 51

[41] Software Simply. Haskell web framework matrix. http://softwaresimply.blogspot.com.br/2012/12/haskell-web-framework-matrix_20.html, March 2014. xi, 56

[42] Douglas R. Stinson. *Cryptography: Theory and Practice*. CRC Press, 3rd edition, 2005. 5

[43] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed operating systems. *ACM Comput. Surv.*, 17(4):419–470, December 1985. 47