UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

MIDDLEWARE DE COMUNICAÇÃO DE WEB SERVICE PARA THRIFT

ROBSON DA SILVA

BLUMENAU
2015

2015/2-21

ROBSON DA SILVA

MIDDLEWARE DE COMUNICAÇÃO DE WEB SERVICE PARA THRIFT

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Francisco Adell Péricas, Mestre - Orientador

BLUMENAU
2015

2015/2-21

INTERFACE DE CONVERSÃO THRIFT PARA WEB SERVICE

Por

ROBSON DA SILVA

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof. Francisco Adell Péricas, Mestre – Orientador, FURB
Membro:	Prof. Marcel Hugo, Mestre, FURB
Membro:	Prof. Paulo Fernando da Silva, Doutor, FURB

Blumenau, 9 de dezembro de 2015

AGRADECIMENTOS

A minha família, principalmente aos meus pais Donizete da Silva e Rosené da Silva, pelo apoio incondicional em todos os momentos.

A minha namorada, Camilla Viecelli, pela paciência e compreensão ao longo do desenvolvimento deste trabalho.

Aos meus amigos e colegas de classe, que direta ou indiretamente tiveram participação na minha vida acadêmica e no resultado dela.

Ao professor Mauro Marcelo Mattos, que na condição de orientador me auxiliou na ideia e formulação da proposta, mas que não pôde me acompanhar ao longo do desenvolvimento da mesma.

Ao meu orientador Francisco Adell Péricas, por ter abraçado a proposta deste trabalho e por todo apoio técnico oferecido ao longo do desenvolvimento dele.

I got this.
Winston, Opie

RESUMO

Este trabalho apresenta o desenvolvimento de um *middleware* que possibilita a comunicação com servidores *thrift* através de requisições HTTP. O *middleware* permite que os usuários façam o cadastro e manutenção dos dados de acesso dos servidores *thrift* através de uma aplicação *web* desenvolvida e, com estes dados, efetua o processo de geração dos *web services* REST responsáveis pelo tratamento das requisições HTTP recebidas. Todas as funcionalidades do *middleware* foram implementadas com a linguagem C#, utilizando recursos disponibilizados pelo .NET. A aplicação *web* foi desenvolvida em HTML5, CSS3 e AngularJS. Com os resultados obtidos, comprovou-se que é possível a geração dinâmica de interfaces *web service* que possibilitam o acesso às funcionalidades de um servidor *thrift*, isentando os clientes deste servidor da necessidade de implementações específicas do *thrift* em suas aplicações.

Palavras-chave: Integração de sistemas. Thrift. Web service. REST. HTTP.

ABSTRACT

This document presents the development of a middleware that allows the communication with thrift servers through HTTP requests. The middleware allows the users to register and manage the access information of thrift servers through an web application and then execute the process that generate REST web services responsible for treating the HTTP requests. All the middleware features were implemented using C# language and some other .NET resources. The web application was implemented using HTML5, CSS3 and AngularJS. With the results obtained, was proved possible the dynamic generation of web service interfaces that can access functions of a thrift server, eliminating any requirement for specific thrift implementation on the cliente side.

Key-words: Systems integration. Thrift. Web service. REST. HTTP.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API – *Application Programming Interface*

CORBA – Common Object Request Broker Architecture

CSS – *Cascading Style Sheets*

DLL – Dynamic Link Library

HTML – HyperText Markup Language

HTTP – HyperText Transfer Protocol

IDL – Interface Description Language

JSON – *JavaScript Object Notation*

LAMP – Linux, Apache, MySQL e PHP

MVC – *Model View Control*

OMG – *Object Management Group*

REST – *Representational State Transfer*

RPC – *Remote Procedure Call*

SOAP – *Simple Object Access Protocol*

SQL – Structured Query Language

URL – *Uniform Resource Locator*

WSDL – Web Services Description Language

XML – eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Um dos maiores problemas no desenvolvimento de sistemas distribuídos é a comunicação entre os elementos de software (JONES, 2014). Grande parte deste problema está relacionado à utilização de diferentes tecnologias e linguagens de programação por parte dos sistemas que necessitam de integração.

Em uma época em que a utilização de *web services* tornou-se solução cotidiana para equipes de desenvolvimento de software, pouco se fala sobre outras tecnologias que possibilitam a interoperabilidade de sistemas com eficiência e escalabilidade. O *thrift* é uma tecnologia baseada em uma biblioteca de software e um conjunto de ferramentas de geração de código desenvolvida pelo Facebook para viabilizar o desenvolvimento de seus serviços *back-end*, levando em consideração que no Facebook cada serviço é implementado na linguagem mais apropriada, de acordo com sua finalidade. A ferramenta tem como objetivo principal possibilitar a comunicação eficiente e confiável entre linguagens de programação, abstraindo detalhes de cada linguagem (SLEE; AGARWAL; KWIATKOWSKI, 2007).

O *thrift* surgiu em função de que o *framework* LAMP, que era utilizado pelo Facebook anteriormente, não conseguiu suportar o aumento de demanda de serviços experimentada pela companhia (SLEE; AGARWAL; KWIATKOWSKI, 2007). Mesmo sendo o *thrift* uma tecnologia robusta e bem estruturada, ele demanda uma certa curva de aprendizagem para sua utilização. Apesar deste custo, alguns serviços já disponibilizam servidores utilizando esta tecnologia e exigem que seus clientes se comuniquem com seu sistema através deste meio, como é o caso do sistema e-SUS, do Ministério da Saúde (2015).

O *web service*, por sua vez, é uma tecnologia que se popularizou nos últimos anos e que já é utilizada em larga escala na resolução de problemas de comunicação entre aplicações distribuídas (LIMA, 2012). Os estudos sobre essa tecnologia já estão bem avançados e existe bastante suporte e documentação para sua implementação, permitindo uma certa comodidade com relação ao seu uso no desenvolvimento de aplicações.

Dado este cenário, a ideia deste trabalho é desenvolver um *middleware* que facilite a integração de sistemas que utilizam *thrift*. O *middleware* fica responsável por abstrair as funcionalidades de servidores *thrift*, disponibilizando as mesmas através de uma interface *web service* que é gerada em tempo de execução. Dessa forma, é possível que os clientes destes servidores possam acessar e utilizar suas funcionalidades através de requisições HTTP, sem a necessidade de realizar qualquer implementação do *thrift*.

Além disso, para os clientes que querem implementar diretamente a estrutura do *thrift* em seus sistemas, o *middleware* possibilita o *download* dos códigos fonte na linguagem de programação mais conveniente. As linguagens de programação oferecidas são as mesmas que já são suportadas pelo *kit* de geração de código do próprio *thrift*.

1.1 OBJETIVOS

O objetivo deste trabalho é disponibilizar uma interface que possibilite o acesso a recursos de um servidor *thrift* através da tecnologia *web service*.

Os objetivos específicos são:

a) disponibilizar bibliotecas C# para abstrair as funcionalidades de geração de código do *thrift* e de *web services*;
b) disponibilizar um *middleware* capaz de gerar interfaces de *web service* em tempo de execução para acesso a servidores *thrift*;
c) disponibilizar uma aplicação *web* para que o usuário possa interagir com os processos implementados no *middleware*;
d) possibilitar a execução de funcionalidades de um servidor *thrift* através de requisições HTTP realizadas para o *middleware* desenvolvido.

1.2 ESTRUTURA

O trabalho está organizado em 4 capítulos principais. O primeiro capítulo apresenta a introdução ao tema, bem como os objetivos. O segundo capítulo trata da fundamentação teórica com base na pesquisa realizada. O terceiro capítulo contempla a especificação e detalhes da implementação do *middleware* proposto. Por fim, o quarto e último capítulo aborda as conclusões e resultados obtidos na pesquisa e desenvolvimento deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta uma visão geral sobre integração de sistemas, as principais tecnologias e padrões utilizados para realização desta integração e uma visão mais aprofundada do *thrift.* Por fim são apresentados alguns trabalhos correlatos que também fazem uso do *thrift*.

2.1 INTEGRAÇÃO DE SISTEMAS

A integração de aplicações e sistemas permite o compartilhamento de informações de uma organização internamente ou com parceiros (CHAVES, 2004). Nesta sentença o conceito de integração é bem amplo, podendo referir-se desde sistemas que simplesmente possuem uma base de dados compartilhada até sistemas distribuídos de alta complexidade.

Grande parte das organizações já veem a questão da integração de sistemas como um item crítico no desempenho dos negócios (SORDI; MARINHO, 2007). Basta analisar como o desenvolvimento de aplicativos para dispositivos móveis ganhou e segue ganhando mercado. De forma geral, estes aplicativos necessitam de integração principalmente pelo fato das diferentes plataformas que eles buscam atender.

A demanda inicial das empresas consiste em compartilhar os dados e processos sem a necessidade de grandes mudanças nas aplicações e estruturas de dados (CHAVES, 2004). De forma geral, efetuar alterações na estrutura de uma aplicação para fazer a integração dela implica em maiores custos de desenvolvimento, além de que, tendo-se uma regra de negócios extensa, problemas ainda maiores podem surgir devido à complexidade no entendimento da solução. Ainda para Chaves (2014), a solução mais viável nestes casos é acoplar os sistemas, disponibilizando uma interface que viabilize a comunicação das aplicações através de um modelo ou protocolo comum.

Hoje, em se tratando de integração, a principal tecnologia considerada para a realização de integração de sistemas são os *web services*. Segundo Lima (2012), a implementação deles se tornou comum em uma época onde constantemente um sistema necessita disponibilizar informações ou funcionalidades para que outro, interno ou de terceiros, possa consumir.

2.2 RPC

Chamada Remota de Procedimento (RPC), tradução de *Remote Procedure Call*, é um protocolo onde o software faz a chamada de um serviço que se encontra em execução em

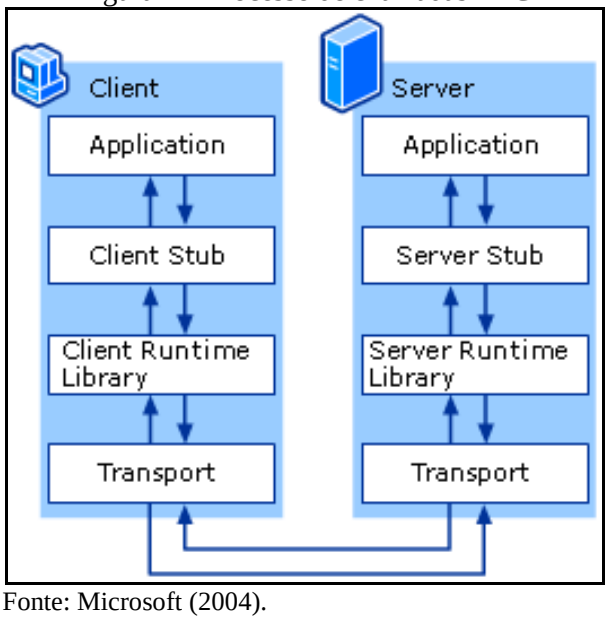
outro computador, sem necessitar de maiores detalhes (ROUSE, 2009). A estrutura segue o modelo cliente/servidor e, de forma geral, as chamadas funcionam de forma síncrona.

Basicamente, o cliente e o servidor compartilham uma interface pré-definida para que o cliente saiba os recursos que pode consumir e como deve se comunicar com o servidor. Conhecendo esta interface, e sabendo o endereço para acesso do servidor, o cliente pode fazer sua chamada passando os parâmetros necessários. O servidor, por sua vez, recebe a requisição, a processa e devolve o resultado do processamento ao cliente, que então segue sua execução normalmente. Segundo Rouse (2009), para este tipo de comunicação, sistemas RPC implementam uma camada de transporte própria.

Conforme demonstrado na Figura 1, o processo de uma chamada RPC se inicia na aplicação cliente no momento em que ela faz uma chamada para um *client stub*. Um *stub* é um trecho de código gerado e incluído na aplicação no momento do desenvolvimento e que contém os parâmetros que o cliente necessita para realizar a transmissão. O *client stub* encaminha as informações para a *client runtime library*, que é responsável por especificar um protocolo de transporte, preparar a mensagem de acordo com esse protocolo e então lançá-la na camada de transporte, que faz a transmissão através da rede (MICROSOFT, 2004).

Do outro lado, quando o servidor recebe uma requisição em sua camada de transporte, ele a repassa para a *server runtime library* que então faz a chamada para um *server stub*. O *server stub* faz a conversão da mensagem recebida, interpreta os parâmetros e solicita a execução da função requisitada no servidor. Finalizada a execução, o caminho inverso é feito para informar a aplicação cliente do resultado, sendo possível também o retorno de parâmetros conforme a necessidade (MICROSOFT, 2004).

Figura 1 – Processo de chamadas RPC



Fonte: Microsoft (2004).

Todo este processo remete a um modelo onde o processamento dos dados se torna distribuído: o cliente faz uma chamada remota a um serviço e o processamento desta chamada é realizado por um outro computador, sendo que o cliente não precisa se preocupar com os detalhes deste processamento, apenas tratar os dados que receber ao final. Algumas das implementações mais conhecidas são o CORBA e os *web services* (GOKHALE; KUMAR; SAHUGUET, 2002).

O *Common Object Request Broker Architecture* (CORBA) é um modelo de integração para sistemas distribuídos, baseado em objetos, definido pelo *Object Management Group* (OMG) (GOKHALE; KUMAR; SAHUGUET, 2002). Trata-se de uma tecnologia onde o cliente e o servidor devem implementar uma interface em comum, para que dessa forma possam trafegar objetos com estrutura pré-definida em sua comunicação.

2.3 WEB SERVICES

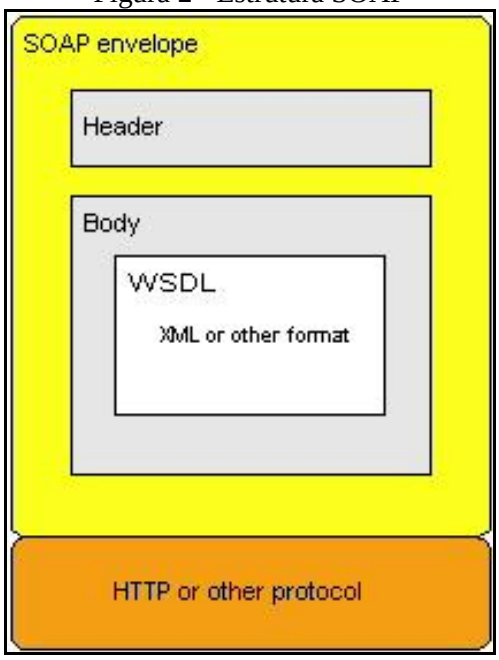
Um *web service* é um sistema de software desenvolvido para suportar interoperabilidade entre máquinas sobre uma rede, o qual pode apresentar uma interface que o descreve (MORO; DORNELES; REBONATTO, 2011). De forma geral, *web services* são implementações específicas de RPC que trabalham em cima de diversos protocolos, porém o mais comum deles é o *Hypertext Transfer Protocol* (HTTP). Além disso, para Lima (2012), existem dois tipos principais de estrutura de comunicação: SOAP e REST.

2.3.1 SOAP

O *Simple Object Access Protocol* (SOAP) utiliza o conceito de troca de mensagens através de envelopes, que são documentos *Extensible Markup Language* (XML). Estes documentos são padronizados, de forma que possam servir como meio de comunicação entre o cliente e o servidor. Ao disponibilizar um *web service*, um documento *Web Service Description Language* (WSDL) também deve ser disponibilizado. Um WSDL nada mais é do que um outro arquivo XML que informa ao cliente o formato e estrutura do *web service* que ele está consumindo (GOKHALE; KUMAR; SAHUGUET, 2002).

A Figura 2 representa a estrutura de mensagens SOAP. O envelope é composto por 2 itens: *header* e *body*. O *header*, ou cabeçalho, é um item opcional onde são colocadas informações de autenticação, *encoding*, entre outros detalhes que possam ser úteis para que o servidor realize o processamento do envelope. O *body*, ou corpo, contém a mensagem propriamente dita, definida por um WSDL, que como citado anteriormente, é normalmente representada no formato XML (BARRY, 2003).

Figura 2 - Estrutura SOAP



Fonte: Barry (2003).

O Quadro 1 descreve um exemplo de envelope SOAP no formato XML. O envelope monta uma requisição para o endereço http://www.myservice.com/users, solicitando a execução do método `GetUser` e passando um parâmetro `UserID` com o valor 123.

Quadro 1 - Exemplo de requisição SOAP

```
1  <?xml version="1.0"?>
2  <soap:Envelope
3     xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
4     soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
5     <soap:Body pb="http://www.myservice.com/users">
6        <pb:GetUser>
7           <pb:UserID>123</pb:UserID>
8        </pb:GetUser>
9     </soap:Body>
10 </soap:Envelope>
```

Assim como as requisições, as respostas enviadas por servidores SOAP são envelopadas para serem transmitidas ao cliente. Todo este processo de geração, transmissão e interpretação dos envelopes resulta em custos de processamento e de tráfego que foram tornando esta arquitetura inviável para alguns modelos de serviço que necessitam de mais performance. Para estes casos, arquiteturas alternativas de implementação de *web services*, como o REST, surgiram para prover uma melhor experiência na realização das requisições entre cliente e servidor.

Apesar do modelo REST ser muito utilizado no desenvolvimento de APIs públicas, ainda existem alguns argumentos para a utilização do SOAP. Segundo Hunsaker (2015), este modelo é bastante viável para aplicações que precisam manter contratos formais entre o cliente e o servidor.

2.3.2 REST

O *Representational State Transfer* (REST) é um estilo de arquitetura baseado nos princípios que descrevem como os recursos de rede são definidos e endereçados. Estes princípios surgiram inicialmente como uma alternativa ao uso do SOAP na comunicação de *web services* (BARRY, 2007).

A utilização em maior escala do modelo REST para desenvolvimento de *web services* se deu mais recentemente, se comparado ao modelo SOAP. O REST define uma interface de transmissão de dados, normalmente baseada em protocolo HTTP, sem ser necessária uma camada adicional de mensagem, como acontece com os envelopes SOAP (MORO; DORNELES; REBONATTO, 2011).

No Quadro 1 pode-se observar um exemplo de envelope SOAP que é montado para realizar uma requisição básica ao servidor, buscando as informações de um usuário pelo seu identificador. Com o modelo REST, e utilizando o protocolo HTTP, a mesma funcionalidade pode ser realizada executando uma simples requisição do tipo GET para o endereço http://www.myservice.com/users/123. O servidor recebe a requisição, interpreta os parâmetros e retorna uma resposta contendo as informações do usuário cujo identificador é o valor 123.

Dado o seu forte vínculo com HTTP, serviços REST trabalham com 4 operações principais para gerenciar e acessar os recursos do servidor:

a) GET (requisita um ou vários recursos do servidor);
b) POST (envia um recurso novo para o servidor, em geral, para inclusão);
c) PUT (envia um recurso pré-existente para o servidor, em geral, para alteração);
d) DELETE (exclui um recurso existente do servidor).

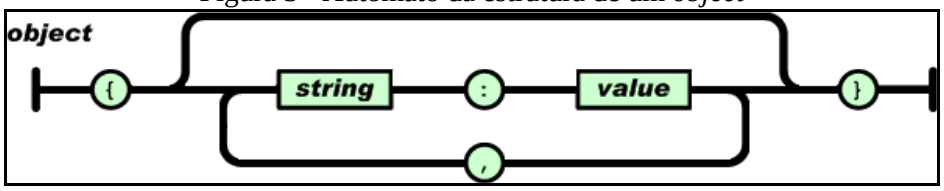
Devido ao seu conceito de dispensar armazenamento de estado e envelopes adicionais de comunicação e por possuir bom suporte para *cache* de informações, serviços REST apresentam melhor performance quando comparados a serviços SOAP (MORO; DORNELES; REBONATTO, 2011).

O modelo REST oferece suporte a vários formatos de dados, porém apresenta um predominante uso de JSON devido ao suporte oferecido, principalmente pelos navegadores (HUNSAKER, 2015). A combinação de REST e JSON, transmitindo os dados com base no protocolo HTTP, resulta em serviços com um forte potencial de performance e escalabilidade, atributos que estão no topo dos requisitos de aplicativos e softwares da atualidade.

O *JavaScript Object Notation* (JSON) é um formato leve para transmissão de dados, além de ser de fácil leitura e escrita para humanos e com baixo custo de geração e análise para as máquinas (JSON.ORG, 2003). O JSON consiste em duas estruturas de dados principais, que são os *objects* e os *arrays.*

A Figura 3 representa o autômato da estrutura de um objeto JSON. Basicamente, um objeto possui um conjunto de pares de propriedade e valor. A propriedade e seu respectivo valor são separados por dois pontos, enquanto os pares são separados entre si por vírgula. Todo o conjunto de pares deve ser colocado entre chaves. O Quadro 2 ilustra um código simples de um objeto JSON.

Figura 3 - Autômato da estrutura de um *object*



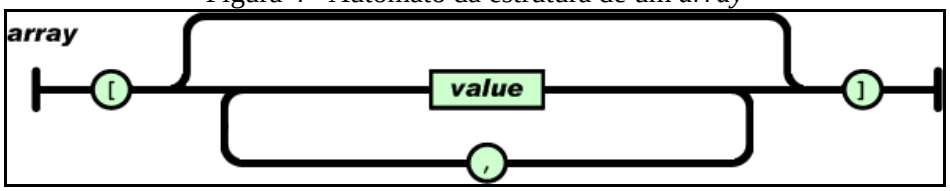
Fonte: Json.org (2003).

Quadro 2 - Exemplo de código JSON de um *object*

```
1  {
2     “nome”: “João”,
3     “idade”: 18,
4     “cidade”: “Blumenau”
5  }
```

A Figura 4 representa o autômato da estrutura de um *array* JSON. Um *array* é um conjunto de valores e/ou objetos, que são separados entre si por vírgula. Todo o conjunto deve ser colocado entre colchetes. O Quadro 3 ilustra um código simples de um *array* de objetos JSON.

Figura 4 - Autômato da estrutura de um *array*



Fonte: Json.org (2003).

Quadro 3 - Exemplo de código JSON de um *array*

```
1  [
2     {“nome”: “João”, “idade”: 18, “cidade”: “Blumenau”},
3     {“nome”: “Maria”, “idade”: 20, “cidade”: “São Paulo”}
4  ]
```

2.4 THRIFT

O thrift é uma biblioteca desenvolvida pelo Facebook para auxiliar na implementação de serviços *back-end* eficientes e escaláveis (SLEE; AGARWAL; KWIATKOWSKI, 2007). Junto com a biblioteca é fornecido um conjunto de ferramentas de geração de código, que é utilizado para gerar as estruturas RPC para cliente e servidor na linguagem de programação que for necessária.

A ideia inicial do *thrift* veio junto com a necessidade do Facebook de expandir os seus serviços (SLEE; AGARWAL; KWIATKOWSKI, 2007). Na fase de planejamento foram selecionadas várias linguagens de programação que seriam mais eficientes em diferentes procedimentos dentro da estrutura da companhia. Porém, era necessário que fosse desenvolvida uma forma performática e transparente para que essas funcionalidades se mantivessem integradas.

Algumas estruturas importantes foram desenvolvidas para tornar o *thrift* uma ferramenta robusta e que funcionasse com as variadas linguagens de programação adotadas no desenvolvimento de aplicações do Facebook (SLEE; AGARWAL; KWIATKOWSKI, 2007). Um destes itens foi a definição de uma *Interface Definition Language* (IDL), que é responsável pela criação da interface em que os arquivos devem ser gerados. Os objetos e serviços devem ser descritos nesta IDL para que posteriormente o *thrift* os utilize como base para geração das estruturas nas linguagens de programação. As seguintes linguagens são algumas das suportadas pela ferramenta de geração de código do *thrift*:

a) C++;
b) Java;
c) Python;
d) PHP;
e) C#;
f) Delphi.

Apesar de ter sido desenvolvido pelo Facebook, o *thrift* foi disponibilizado como *open source* em abril de 2007 e incubado como um projeto Apache em 2008, recebendo a nomenclatura de Apache Thrift (APACHE SOFTWARE FOUNDATION, 2014).

2.4.1 Tipos

Ao se definir uma IDL para geração de código entre diversas linguagens, é também necessário que seja definido um sistema de tipos. O *thrift* aceita os seguintes tipos primitivos:

a) *bool*;

b) *byte*;

c) i16, i32, i64 (*integers*);

d) *double*;

e) *string*.

Além dos tipos primitivos também é aceita a declaração de *structs*, que são equivalentes à declaração de classes/objetos em linguagens de alto nível (SLEE; AGARWAL; KWIATKOWSKI, 2007). Assim como nestas linguagens, uma *struct* é declarada com suas variáveis e propriedades e, após sua declaração, também pode ser utilizada como um tipo. A IDL ainda aceita a utilização de três tipos de conjuntos de dados. São eles:

a) *list* (lista ordenada de elementos);

b) *set* (lista desordenada de elementos únicos);

c) *map* (lista de registros com chave e valor, onde a chave é única).

Também são aceitas *exceptions* e *services*. As *exceptions* servem basicamente para tratar erros de forma comum entre as linguagens (SLEE; AGARWAL; KWIATKOWSKI, 2007). Os *services*, de forma geral, são as declarações equivalentes às interfaces que são utilizadas em linguagens de alto nível, ou seja, neles são declarados os métodos que devem ser implementados, informando seu tipo de retorno e parâmetros.

O Quadro 4 mostra um trecho de código escrito na IDL do *thrift* onde são declarados uma *struct* e um *service*, que no código gerado pelo *thrift* se tornam, respectivamente, uma classe para representação de um objeto e uma interface de declaração dos métodos. Os *namespaces* podem ser definidos de acordo com a linguagem necessária. Neste caso foi definido um *namespace* para o Java e outro para o restante das linguagens que vierem a ser utilizadas.

Quadro 4 - Código no formato *thrift* IDL

```
1   namespace * Object.Generated
2   namespace java Generated
3
4   struct MessageObject
5   {
6      1: i32 id,
7      2: string content
8   }
9
10  service MessageService
11  {
12     list<MessageObject> getMessages();
13     MessageObject getMessageById(i32 id);
14  }
```

2.4.2 Transporte e Protocolos

As camadas de transporte e protocolo fazem parte da biblioteca de execução do *thrift* e são independentes da camada de código gerada. Sendo assim, a aplicação pode alterar a forma de comunicação sem a necessidade de recompilar o código gerado (DIMOPOULOS, 2013).

A camada de protocolo implementa serviços de serialização e deserialização e serve basicamente para definir o formato em que as mensagens são transmitidas entre os componentes da comunicação (SLEE; AGARWAL; KWIATKOWSKI, 2007). Os seguintes protocolos são suportados pelo *thrift*:

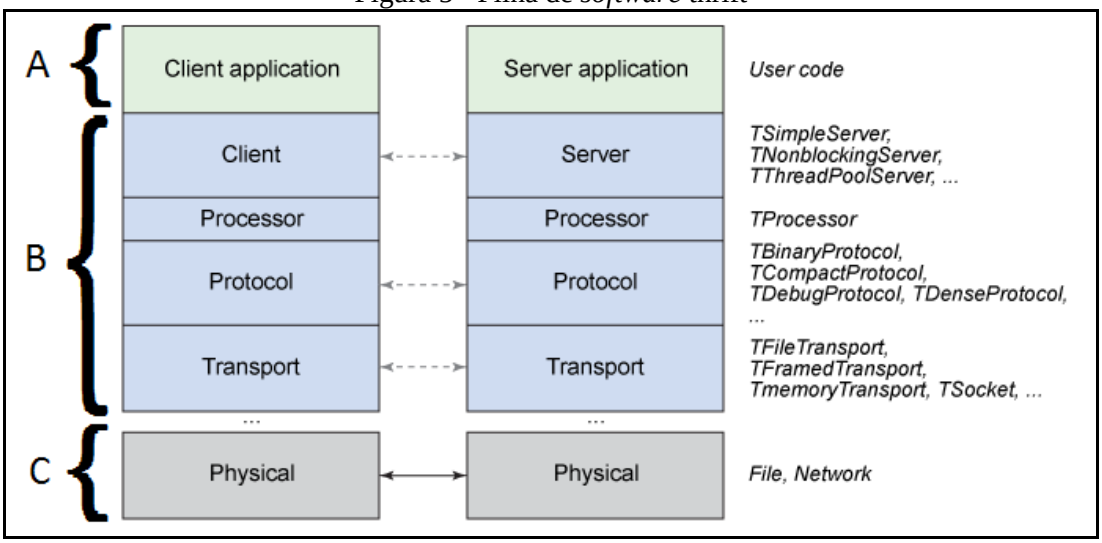
a) TBinaryProtocol (codificação binária);
b) TCompactProtocol (codificação binária ainda mais compacta);
c) TDenseProtocol (similar ao TCompactProtocol, com detalhes à parte);
d) TJsonProtocol (utiliza JSON para codificação);
e) TSimpleJsonProtocol (somente escrita, utilizado para *script languages*);
f) TDebugProtocol (utiliza linguagem humana, útil para *debug*).

A camada de transporte é responsável pela leitura e escrita dos dados baseada na camada de protocolos utilizada (SLEE; AGARWAL; KWIATKOWSKI, 2007). Esta camada gerencia os detalhes de entrada e saída de dados para que eles possam ser trafegados através da rede. Os seguintes modelos de transporte são suportados:

a) TSocket (transmissão através de *sockets* de entrada e saída; síncrono);
b) TFrameTransport (transmissão de informação em quadros; assíncrono);
c) TFileTransport (transmite as informações para arquivos);
d) TMemoryTransport (usa recursos de memória para entrada e saída);
e) TZlibTransport (utiliza compactação zlib, necessita de outro transporte).

A Figura 5 ilustra as camadas de uma aplicação *thrift*, evidenciando alguns dos componentes citados no texto. As camadas A se referem aos sistemas de cliente e servidor que implementam o *thrift*. As camadas B são da estrutura interna do *thrift*, que fazem o gerenciamento de transporte, protocolos, processamento e interpretação no cliente e servidor. As camadas C se referem ao *hardware* e outros componentes que fazem a comunicação entre os sistemas.

Figura 5 - Pilha de *software* thrift



Fonte: Jones (2014).

2.4.3 Processador e Servidores

A camada de processador (TProcessor) recebe como argumento um protocolo de entrada e um de saída (SLEE; AGARWAL; KWIATKOWSKI, 2007). Desta forma, quando ela faz a leitura de dados de uma entrada, sabe qual protocolo utilizar, efetua o processamento e utiliza o protocolo de saída para retornar a informação para o outro lado.

A camada de servidores (TSimpleServer) é a camada utilizada no software que irá disponibilizar o serviço (SLEE; AGARWAL; KWIATKOWSKI, 2007). Esta camada é responsável por monitorar a porta definida como entrada e ao receber alguma informação, repassá-la ao processador para que ele possa deserializar e realizar o processamento. Os seguintes modelos de servidor podem ser utilizados:

a) TSimpleServer (*thread* única; síncrono);
b) TThreadPoolServer (várias *threads*; síncrono);
c) TNonblockingServer (várias *threads*; assíncrono).

2.5 TRABALHOS CORRELATOS

A seguir são descritos 3 trabalhos correlatos ao proposto, sendo que ambos utilizam o *thrift* como interface de comunicação. O item 2.5.1 descreve o BeeSUS, uma ferramenta nacional para realizar a integração de cadastros do sistema de saúde e-SUS. O item 2.5.2 descreve o Remote TestKit Thrift API, que é uma ferramenta que emula dispositivos móveis para realização de testes de aplicativos. O item 2.5.3 descreve o Evernote, que se refere a uma ferramenta para organização de dados pessoais.

2.5.1 BeeSUS

Recentemente o sistema e-SUS do Ministério da Saúde (2015) definiu o *thrift* como interface de comunicação com os sistemas próprios dos clientes que utilizam o serviço. A BeeIT (2015), uma empresa brasileira, desenvolveu o BeeSUS, uma aplicação que facilita a integração dos sistemas próprios com o e-SUS abstraindo a camada *thrift* da comunicação. Abstrair, neste caso, significa que a implementação do cliente *thrift* foi encapsulada, de forma que os clientes não precisem fazer uso direto dele em seus sistemas.

Os clientes da BeeIT disponibilizam acesso a camada de dados de seus sistemas, escrevendo as *queries* ou *views* em SQL de acordo com o *layout* previamente definido pelo e-SUS. A partir daí o BeeSUS faz a leitura destes dados, converte eles para um formato compatível com o *thrift* e realiza o envio das informações para o servidor.

A BeeIT também disponibiliza o uso de *web services* para realizar o envio das informações ao e-SUS. Neste processo os clientes submetem as informações via *web service* e internamente o serviço da BeeIT faz a conversão dos dados para envio ao e-SUS através do *thrift*.

O produto deve atender à necessidade dos clientes do serviço e-SUS, pois ao fazer essa implementação, o BeeSUS isenta seus clientes dos possíveis problemas técnicos que eles possam ter com o *thrift*. Além disso, os dois serviços oferecidos pela BeeIT facilitam a comunicação com o servidor, uma vez que os clientes não necessitam sequer conhecer o *thrift*, somente disponibilizar consultas SQL ou consumir uma camada de *web service*.

2.5.2 Remote TestKit Thrift API

Remote TestKit é uma ferramenta baseada em nuvem para testes de aplicativos *mobile*. Trata-se de um software desenvolvido pela empresa japonesa NTT Resonant (2014), que funciona como um emulador de dispositivos móveis onde o desenvolvedor publica os fontes e consegue executá-los através de uma interface, que pode ser *desktop* ou *web*.

Inicialmente, para fazer uso dos recursos desta ferramenta, os desenvolvedores precisavam implementar um software cliente intermediário, pois a empresa fornecia somente o servidor com as funcionalidades. Dessa forma, cada cliente possuía uma implementação específica para usar a ferramenta, descentralizando o modelo de consumo e gerando retrabalho para cada cliente que resolvesse implementar o módulo.

Para viabilizar a utilização da ferramenta a empresa desenvolveu uma extensão, o Remote TestKit Thrift API. A tarefa desta API é basicamente disponibilizar as funcionalidades do servidor na linguagem mais apropriada para uso no sistema que está implementando. Esta versatilidade se deve ao fato de que, com o *thrift*, basta que os serviços sejam descritos uma vez na sua IDL e, após isso, ele se encarrega de gerar todo o código fonte na linguagem solicitada.

Para implementação desta API a NTT apenas precisou definir o arquivo *thrift*, implementar um servidor baseado nos códigos fonte gerados e disponibilizar este serviço aos seus clientes. A partir daí, resta aos clientes solicitar uma cópia do arquivo *thrift*, compilar ele na linguagem necessária e adicionar aos seus projetos. Dessa forma, através do código fonte, o desenvolvedor passa a ter acesso a toda estrutura de serviços disponibilizadas pelo servidor do Remote TestKit, sem ser necessária uma curva maior de aprendizado.

2.5.3 Evernote

O Evernote é um software para gerenciamento e organização de dados pessoais, desenvolvido pela Evernote Corporation (2008). Existem versões do software para as mais diversas plataformas *desktop, mobile* e *web.*

Desde o início do projeto, em 2007, a equipe de desenvolvimento já tinha em mente os requisitos do software a ser desenvolvido. A aplicação deveria possuir um esquema de sincronização eficiente, uma vez que a ideia principal é a de manter os dados do usuário de forma centralizada, refletindo todas as alterações em qualquer dispositivo que ele esteja usando. Além disso, os conceitos de multiplataforma, versionamento, *open source* e gerenciamento eficiente do formato de dados transmitidos estavam no topo da lista de requisitos.

Na época, a equipe fez o estudo de caso de algumas tecnologias candidatas à atender estes requisitos. Entre SOAP, XML-RPC e algumas outras, o *framework* que se destacou, atendendo todos os requisitos, foi o *thrift.* Com o *thrift* eles teriam a performance necessária para a comunicação entre servidores e clientes através de protocolo binário e poderiam

compilar os serviços na linguagem que mais lhes fosse conveniente. Além disso, o *thrift* havia recentemente sido disponibilizado como *open source* e adotado como um projeto Apache.

Utilizando *thrift* o Evernote construiu uma API robusta e escalável, que roda código nativo em cada um dos dispositivos para o qual é disponibilizado. A empresa também mantém sua API *thrift* documentada e com disponibilidade para aceitar a integração de implementações externas.

2.5.4 Relação do trabalho proposto com os correlatos

A relação do trabalho proposto com o BeeSUS (BEEIT, 2015) está no conceito de evitar que o cliente necessite fazer qualquer implementação ligada ao *framework thrift*. A vantagem do trabalho proposto é que o BeeSUS foi implementado visando a integração de um sistema específico, o e-SUS, enquanto o trabalho proposto sugere uma implementação que seja capaz de abstrair qualquer servidor *thrift*, desde que lhe sejam informados os dados de acesso e o arquivo com a IDL que descreve os serviços. Outro ponto em que as implementações são semelhantes é a disponibilização de *web services* para comunicação com o servidor *thrift*.

A relação do trabalho proposto com os outros dois correlatos, o Remote TestKit (NTT RESONANT, 2014) e o Evernote (EVERNOTE CORPORATION, 2008), está basicamente no estudo e uso do *thrift* nas implementações. Estes correlatos não trabalham com o conceito de abstrair o *thrift* para o cliente, mas sim o de o cliente obter a IDL dos serviços, gerar as estruturas na linguagem do seu sistema próprio e usar as funcionalidades a partir destas estruturas. Para estes casos em que o cliente optar por realizar a implementação do *thrift* ao invés de utilizar a interface REST disponibilizada, o trabalho proposto também será útil.

Para gerar os arquivos nas linguagens de programação com base na IDL é necessário que seja feito o download do executável do *thrift* e que, através de linha de comando, ele seja executado com todos os parâmetros necessários. O trabalho proposto também pretende abstrair essa funcionalidade, possibilitando ao cliente que, a partir do momento que um serviço esteja devidamente cadastrado com a sua IDL, seja possível fazer o download dos arquivos automaticamente na linguagem solicitada.

3 DESENVOLVIMENTO

Para melhor entendimento do desenvolvimento do *middleware* proposto neste trabalho, alguns conceitos utilizados devem ser previamente apresentados. Tratam-se de alguns termos escolhidos para representar as entidades dentro da implementação, que são devidamente contextualizados a seguir:

a) organização: é uma entidade utilizada na implementação para representar uma empresa ou organização;
b) serviço: é uma entidade utilizada para abstrair um servidor *thrift* dentro da implementação, contendo suas informações de acesso e o código em IDL *thrift* que foi compilado para gerar sua interface;
c) *build*: é um processo que será detalhadamente explicado nas seções a seguir, mas resumidamente, é o responsável por interpretar os dados informados no serviço e, a partir deles, gerar a interface *web service* que se comunica com o servidor *thrift*;
d) *controller:* dentro da plataforma .NET, mais especificamente tratando-se de ASP.NET Web API, é uma classe que representa um *endpoint* do *web service* que está em execução. Em outras palavras, é a interface final para realizar o acesso de recursos do *web service*;
e) *assembly*: representa um arquivo do tipo *Dynamic Link Library* (DLL), que é gerado ao final do processo de *build*. É um arquivo que contém as *controllers* já compiladas e que é carregado na aplicação para possibilitar o acesso ao servidor *thrift* através do *web service* REST implementado.

3.1 REQUISITOS

O *middleware* deverá:

a) permitir o cadastro e manutenção de organizações (Requisito Funcional - RF);
b) permitir o cadastro e manutenção de serviços (RF);
c) permitir a associação de organizações com serviços de terceiros (RF);
d) permitir o download de arquivos nas linguagens de programação suportadas pelo *thrift* com base no código *thrift* associado ao serviço (RF);
e) permitir executar *builds* que disponibilizem as interfaces *web service* para comunicação com os servidores *thrift* (RF);
f) permitir acesso via HTTP aos *web services* gerados pela funcionalidade de *build* dos serviços (RF);
g) utilizar a linguagem C# para implementação do *middleware* e das bibliotecas

necessárias (Requisito Não Funcional - RNF);

h) utilizar o *thrift* para geração dos códigos fonte que serão utilizados no processo de *download* e *build* (RNF);

i) utilizar o *framework* ASP.NET Web API para implementação dos *web services* REST gerados no processo de *build* (RNF);

j) utilizar MongoDB para persistência dos dados (RNF).

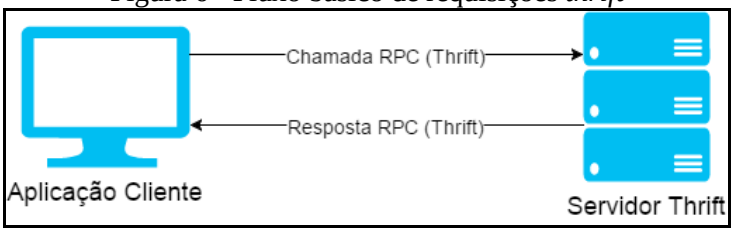
3.2 ESPECIFICAÇÃO

Esta seção apresenta a especificação do *middleware* desenvolvido. Foram elaborados esquemas e diagramas de casos de uso, pacotes, classes e sequência através da ferramenta on-line *draw.io*.

3.2.1 Proposta do *middleware*

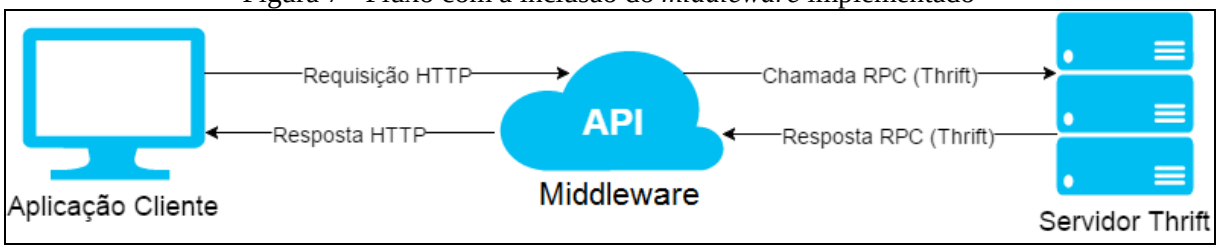
A proposta do *middleware* implementado é a conversão de interface de servidores *thrift* para *web services* REST. Esta conversão possibilita que as aplicações clientes de um determinado servidor não necessitem fazer qualquer implementação relacionada ao *thrift*, uma vez que as mesmas podem ser feitas através do protocolo HTTP. A Figura 6 ilustra o fluxo básico de requisições entre uma aplicação cliente e um servidor *thrift*.

Figura 6 - Fluxo básico de requisições *thrift*



No fluxo básico, o cliente chama uma funcionalidade do servidor através de uma requisição RPC utilizando protocolos específicos do *thrift*. O *middleware* proposto se localiza no meio deste fluxo e assume a responsabilidade de converter e encaminhar estas requisições para o servidor. A partir deste momento, o cliente precisa apenas se preocupar em fazer requisições e tratar as respostas com base no protocolo HTTP, conforme fluxo demonstrado na Figura 7.

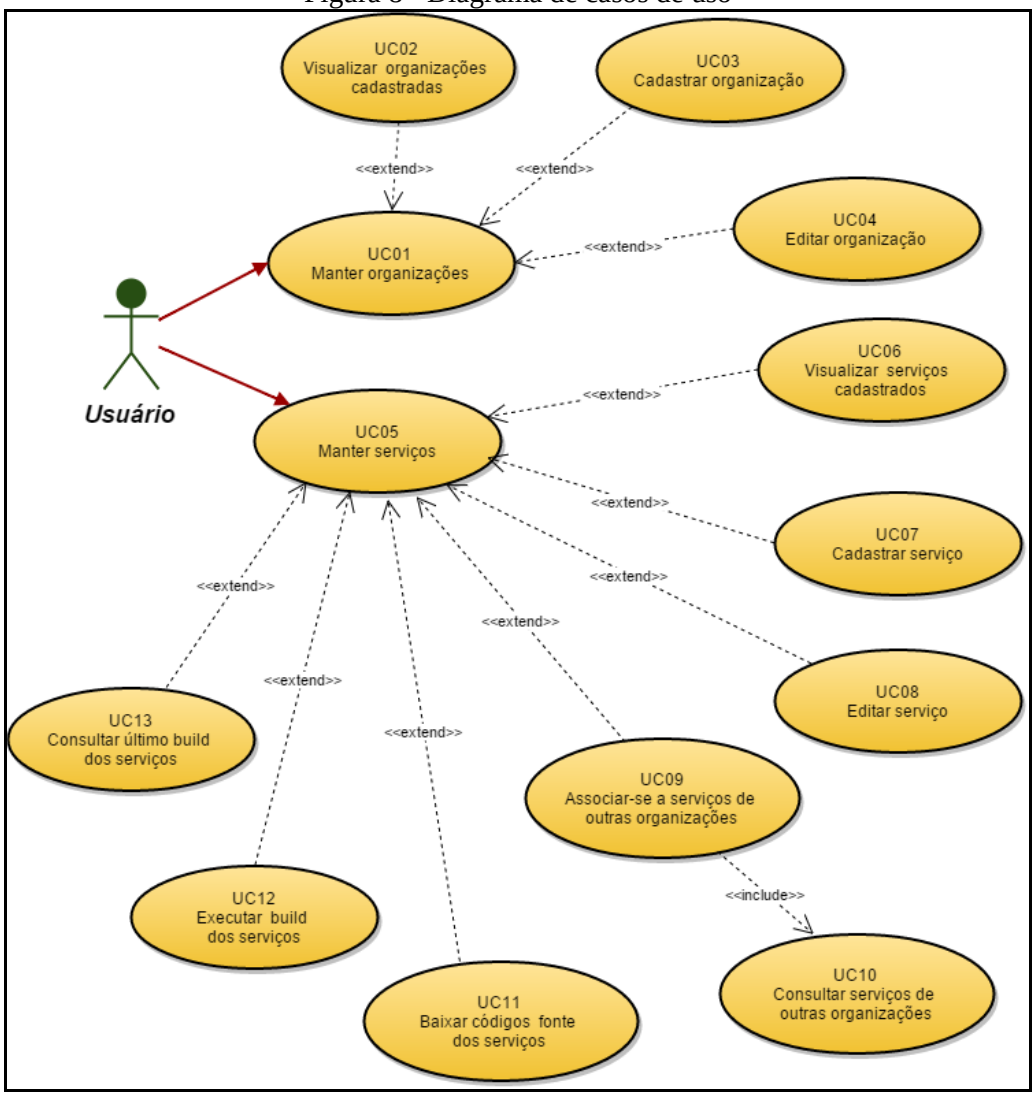
Figura 7 - Fluxo com a inclusão do *middleware* implementado



3.2.2 Diagrama de casos de uso

A Figura 8 mostra o diagrama de casos de uso com as ações disponibilizadas ao usuário final da ferramenta e que implementa os requisitos funcionais propostos para este trabalho.

Figura 8 - Diagrama de casos de uso



A seguir são descritos os casos de uso da Figura 8.

a) `UC01 – Manter organizações:` permite realizar a manutenção de organizações;

b) `UC02 – Visualizar organizações cadastradas:` permite visualizar uma lista das organizações cadastradas;

c) `UC03 – Cadastrar organização:` permite cadastrar novas organizações;

d) `UC04 – Editar organização:` permite editar as informações de organizações já cadastradas;

e) `UC05 – Manter serviços:` permite realizar a manutenção dos serviços;

f) `UC06 – Visualizar serviços cadastrados`: permite visualizar uma lista dos serviços cadastrados;

g) `UC07 – Cadastrar serviço`: permite cadastrar novos serviços;

h) `UC08 – Editar serviço`: permite editar as informações de serviços já cadastrados;

i) `UC09 – Associar-se a serviços de outras organizações`: permite a associação de uma organização com serviços de outra organização com o objetivo de se tornar cliente daquele serviço;

j) `UC10 – Consultar serviços de outras organizações`: permite a consulta de serviços de outras organizações para realizar o caso `UC09`;

k) `UC11 – Baixar códigos fonte dos serviços`: permite o *download* de códigos fontes nas linguagens suportadas baseado no código em IDL *thrift* escrito para o serviço;

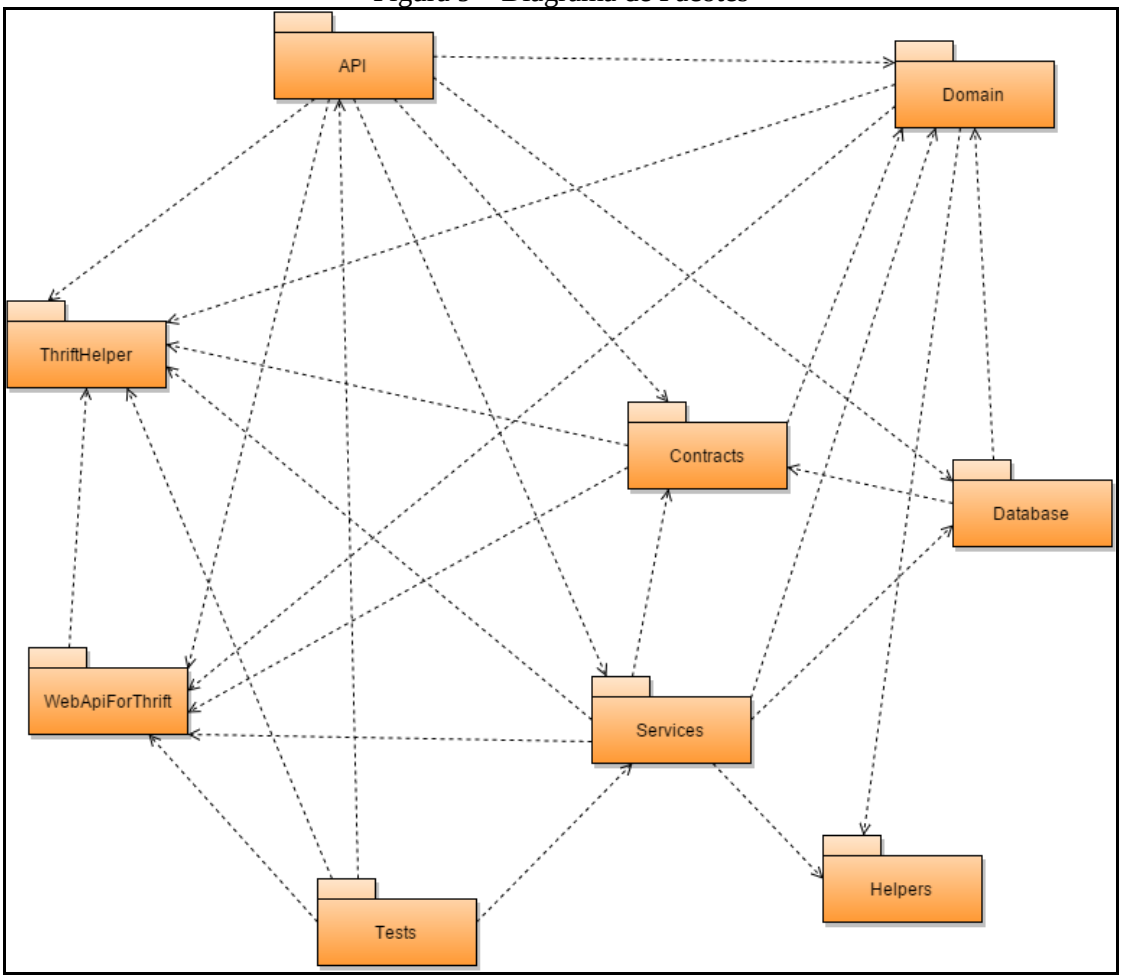
l) `UC12 – Executar build dos serviços`: permite a execução de *build* para geração dos *web services* com base nas informações fornecidas sobre o servidor *thrift*. Este processo só pode ser executado por um usuário responsável pelo serviço e não por associados;

m) `UC13 – Consultar último build dos serviços`: permite a realização da consulta do resultado e informações do último *build* executado para um serviço. Este processo pode ser executado tanto pelo responsável quanto por um associado do serviço.

3.2.3 Diagramas de pacotes e de classes

Nesta seção são apresentados os diagramas de pacotes e de classes relacionados à estrutura do projeto implementado. A Figura 9 apresenta o diagrama de pacotes.

Figura 9 – Diagrama de Pacotes



A seguir são feitas breves descrições dos pacotes demonstrados na Figura 9:

a) `API`: consiste na interface final da implementação, com um *web service* REST e uma aplicação *web* que o implementa;

b) `Services`: é o pacote principal para realização das implementações utilizadas pela `API`. Ele centraliza todas as chamadas que são necessárias a outros pacotes do projeto;

c) `Contracts`: contém apenas as declarações de interfaces que são implementadas em outros pacotes;

d) `ThriftHelper`: é um pacote que foi desenvolvido para abstrair a geração de código do *thrift*;

e) `WebApiForThrift`: é um pacote que foi desenvolvido para gerar os códigos fonte das *controllers* do ASP.NET Web API;

f) `Helpers`: contém implementações genéricas que são utilizadas ao longo do projeto;

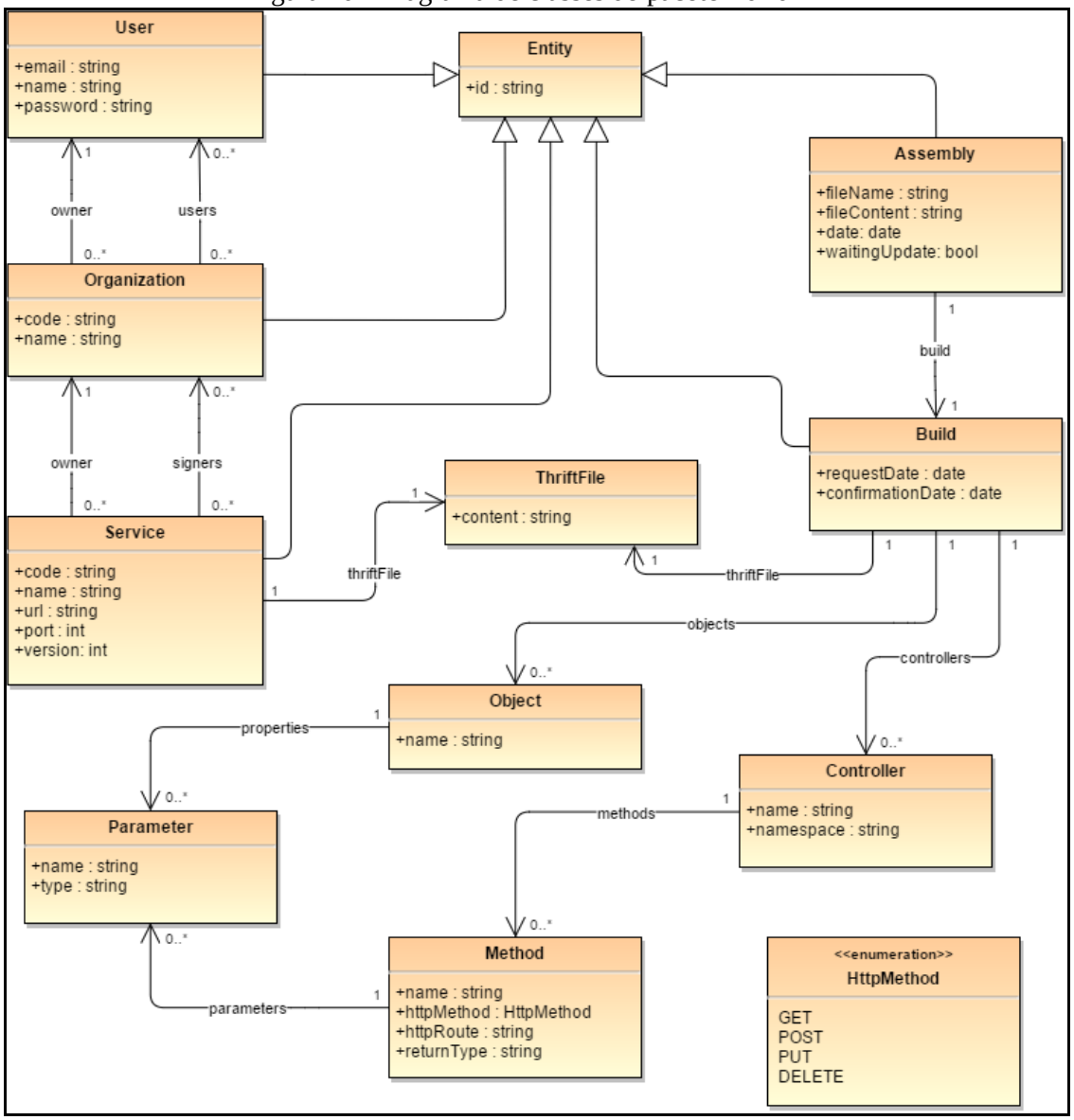
g) `Domain`: contém as declarações das classes que são utilizadas na aplicação, desde a camada de persistência até a camada final da `API`;
h) `Database`: implementa as funcionalidades necessárias para a configuração e acesso ao banco de dados MongoDB utilizado para persistência dos dados;
i) `Tests`: implementa os testes unitários dos pacotes principais, garantindo que novas implementações não interfiram na execução das pré-existentes.

Todos os pacotes descritos são essenciais para o resultado final do trabalho, porém os pacotes `Domain`, `ThriftHelper` e `WebApiForThrift` são detalhadamente descritos nas próximas seções. Isto se deve ao fato da importância deles para o entendimento da implementação.

3.2.3.1 Pacote `Domain`

O pacote `Domain` contém as classes que representam o domínio da aplicação implementada. O diagrama de classes apresentado na Figura 10 demonstra as classes principais pertencentes a este pacote, bem como a descrição de seus atributos e relacionamentos.

Figura 10 - Diagrama de classes do pacote `Domain`



A seguir é feita uma breve descrição de cada uma das classes apresentadas no diagrama da Figura 10:

a) `Entity`: utilizada para herança de objetos que necessitam de um identificador único, em geral, para persistência na base de dados;

b) `User`: representa os usuários que interagem com a aplicação. As operações que um usuário pode efetuar estão brevemente descritas nos casos de uso da Figura 8;

c) `Organization`: representa as organizações criadas por usuários para gerenciamento dos serviços. Uma organização pode ter seus próprios serviços ou associar-se a serviços de outras organizações, tornando-se cliente destes serviços;

d) `Service`: representa os serviços cadastrados por usuários. As informações desta classe são utilizadas para geração das estruturas no processo de *build*, possibilitando o acesso ao servidor *thrift* referenciado. Um serviço possui uma

organização responsável, mas pode ter diversas outras a ele associadas no papel de cliente;

e) `ThriftFile`: é uma classe simples que contém basicamente um código em IDL *thrift* que descreve a interface de um servidor;
f) `Build`: é a classe utilizada para tratar as informações de um *build* executado pela aplicação. Após a execução, este objeto irá conter tanto as estruturas geradas pela compilação do *thrift* quanto as *controllers* geradas para o *web service*;
g) `Assembly`: é utilizada para tratar os dados de um arquivo *assembly* gerado pelo processo de *build*;
h) `Controller`: utilizada para representar as *controllers* geradas e compiladas para o *web service*;
i) `Object`: utilizada para representar os objetos gerados pelo *thrift* e que precisam ser referenciados para que possam ser utilizados nas *controllers*;
j) `Method`: representa os métodos das *controllers*. Os métodos são gerados dinamicamente para possam fazer as devidas requisições ao servidor *thrift* para o qual foram designados;
k) `Parameter`: por conter apenas os atributos `type` e `name`, esta classe é utilizada para representar tanto os parâmetros de um `Method` quanto as propriedades de um `Object`;
l) `HttpMethod`: trata-se de um enumerador criado para definir os tipos de *request* HTTP que podem ser efetuados para acessar o *web service*. Cada `Method` precisa de um `HttpMethod` para que saiba como deve ser realizado o acesso.

3.2.3.2 Pacote `ThriftHelper`

O `ThriftHelper` é um pacote simples e independente que foi desenvolvido para atender algumas necessidades imediatas relacionadas ao *thrift*. A principal dessas necessidades se refere à geração de código fonte, que em primeira instância, só é possível através do executável fornecido pela Apache.

Para uma equipe de desenvolvimento que deseja utilizar o *thrift*, a geração de código é um dos primeiros processos a ser realizado. Com o executável do *thrift* é possível processar o comando que compila o código da IDL e cria as estruturas na linguagem necessária. Os parâmetros necessários para execução deste comando são:

a) a linguagem de programação;
b) um diretório destino onde serão gerados os arquivos;

c) o caminho do arquivo escrito na IDL *thrift*.

Dentro da implementação proposta neste trabalho, a geração de código a partir da estrutura do *thrift* é um dos procedimentos iniciais executado no processo de *build*. Em pesquisas realizadas, não foi encontrada nenhuma biblioteca C# capaz de efetuar a geração de código do *thrift* em tempo de execução e com tal finalidade o pacote `ThriftHelper` foi desenvolvido.

3.2.3.3 Pacote `WebApiForThrift`

Outra funcionalidade importante dentro da implementação deste trabalho é o da disponibilização de *web services* REST para os clientes. Estes *web services* não possuem uma estrutura pré-definida uma vez que suas funcionalidades devem ser dinamicamente baseadas no servidor *thrift* informado no cadastro do serviço. O pacote `WebApiForThrift` foi implementado justamente para realizar a construção destes serviços de forma dinâmica.

Este pacote utiliza o `ThriftHelper` para gerar o código do servidor *thrift* em C#. O código é compilado em tempo de execução pelo .NET e com o resultado dessa compilação é possível identificar as estruturas geradas pelo *thrift*, as classes e métodos que precisam ser utilizados. Tendo o conhecimento da estrutura do servidor é possível construir o código fonte das *controllers* do Web API, que também são compilados pelo .NET.

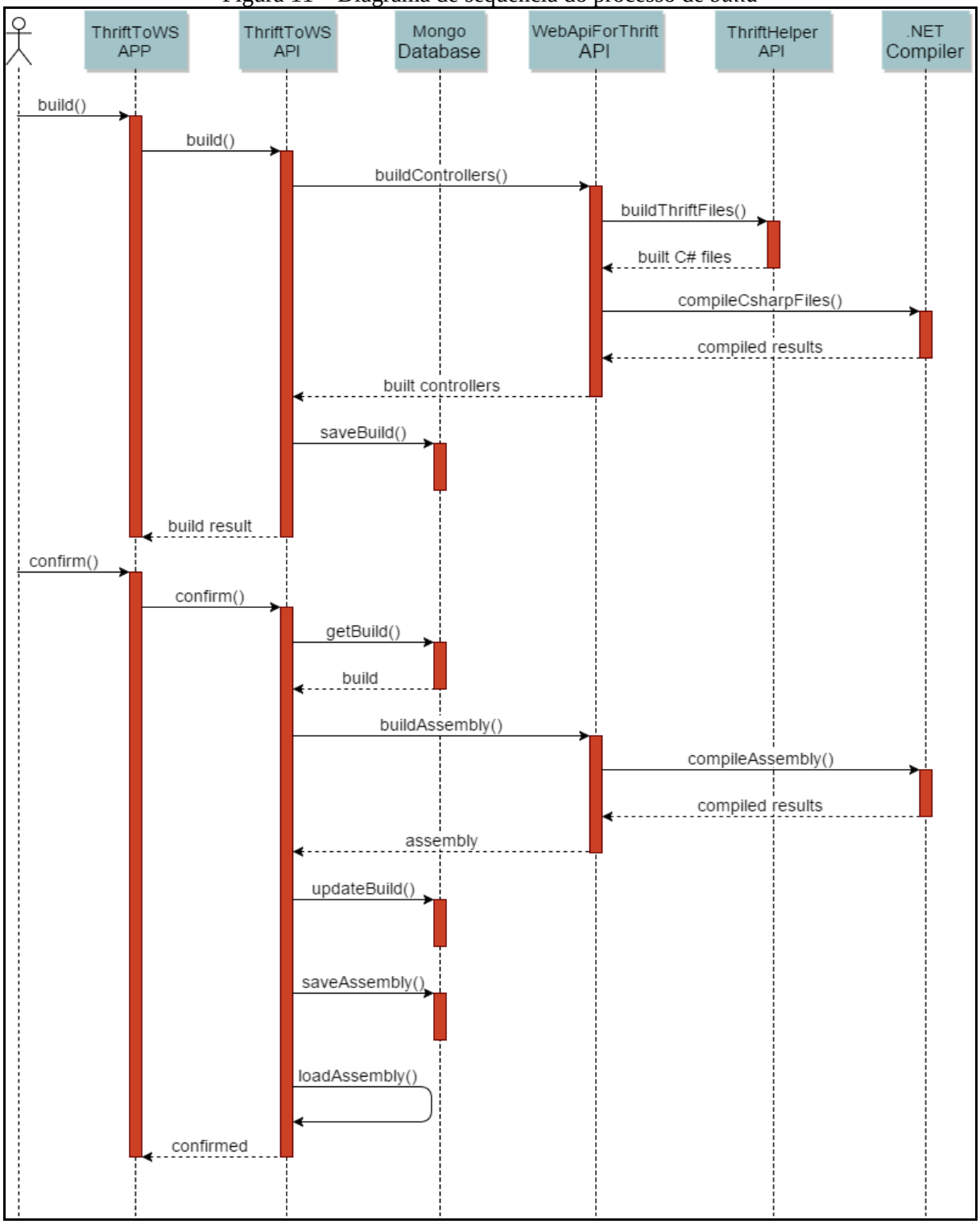
A compilação das *controllers* resulta em um *assembly*, que se trata de um arquivo DLL. Este *assembly* é o retorno do processo de *build* e, para que os serviços implementados dentro dele possam ser utilizados a partir de um *web service*, ele deve ser referenciado em tal aplicação. A responsabilidade deste pacote se dá até o retorno do *assembly*. O processo posterior de referência ou quaisquer outros que sejam efetuados é de responsabilidade da aplicação que está utilizando as funcionalidades do `WebApiForThrift`.

Os pacotes `ThriftHelper` e `WebApiForThrift` foram desenvolvidos de forma desacoplada do restante da aplicação. Isso quer dizer que eles não possuem outras dependências, a não ser entre si, e dessa forma é possível que eles sejam publicados em formato *open source* para que possam ser utilizados em projetos de terceiros.

3.2.4 O processo de *build*

O processo de *build* é detalhado nesta seção, uma vez que nele é concentrada a inteligência principal da implementação. O diagrama de sequência da Figura 11 apresenta as atividades executadas pelos atores envolvidos ao longo do processo.

Figura 11 – Diagrama de sequência do processo de *build*



O processo é iniciado quando o usuário da aplicação solicita a execução de um *build* para o seu serviço através da aplicação web, o `ThriftToWS APP`. A aplicação repassa a solicitação para o `ThriftToWS API`, que é um *web service* REST que representa o *middleware* propriamente dito.

Ao receber uma requisição de *build*, em um primeiro momento, o `ThriftToWS API` solicita a construção das *controllers*, que é feita através do método `buildControllers` do pacote `WebApiForThrift`. O `WebApiForThrift` solicita a geração dos arquivos *thrift* para o `ThriftHelper`, que retorna os arquivos gerados na linguagem C#. Estes arquivos contêm as definições das classes, interfaces e demais estruturas que o servidor utiliza e, a partir delas, é gerado o código das *controllers* do ASP.NET Web API.

O código gerado é então compilado pelo `.NET Compiler`, uma estrutura nativa da *framework* do .NET. Este código consiste basicamente em chamadas RPC ao servidor *thrift* cadastrado pelo usuário no serviço. Se a compilação acontecer de forma correta, o `ThriftToWS API` recebe uma coleção de objetos com as informações das *controllers* geradas, solicita que o `Mongo Database` salve os dados desse *build* através do método `saveBuild` e então retorna as informações para o `ThriftToWS APP` para que haja uma nova interação do usuário.

Neste momento, na interface web, o usuário visualiza uma prévia das informações do *web service* que está sendo gerado para o seu serviço. As informações consistem em dados como os endereços HTTP para cada um dos métodos do seu serviço, os parâmetros que devem ser passados e o formato de retorno de cada um destes endereços. Ao usuário é possibilitada a adequação de algumas informações, como por exemplo, o tipo de *request* que deve ser feito para aquele endereço (GET, POST, PUT ou DELETE).

Finalizada a interação do usuário, ele deve confirmar a execução daquele *build* e, consequentemente, a criação daqueles endereços dentro do *web service*. Da mesma forma como é feito na requisição do *build*, a confirmação também é iniciada pelo `ThriftToWS APP` e repassada para o `ThriftToWS API`. O *middleware* consulta no `Mongo Database`, através do método `getBuild`, as informações salvas anteriormente para aquele *build*, verifica a sua validade e faz a atualização dos dados em memória com base nas novas informações passadas pelo usuário.

Com as informações do *build* atualizadas, o *middleware* solicita ao `WebApiForThrift` a construção do *assembly* chamando o método `buildAssembly`. Nesta parte do *build* o `WebApiForThrift` reconstrói o código das *controllers*, devido a atualização das informações, e solicita a compilação dos fontes ao `.NET Compiler`. Com o resultado desta compilação o `ThriftToWS API` recebe um *assembly*, que é um arquivo DLL com toda a estrutura do *web service* já preparada para as chamadas remotas ao servidor *thrift*.

Ao receber o *assembly* o `ThriftToWS API` chama novamente o `Mongo Database` para salvar as alterações do *build*, através do método `updateBuild`, e também para salvar a estrutura do *assembly*, através do método `saveAssembly`. O *assembly* é então carregado pelo `ThriftToWS API` através de um mecanismo do próprio *framework* do .NET que permite a inclusão de novas estruturas em tempo de execução. A partir deste momento os endereços HTTP que foram gerados nas *controllers* ao longo do *build* ficam disponíveis para que as requisições para o *web service* possam ser realizadas.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as ferramentas e técnicas utilizadas para a implementação do *middleware*, com a descrição das principais partes do seu desenvolvimento e usabilidade.

3.3.1 Técnicas e ferramentas utilizadas

Para o desenvolvimento do *middleware* de conversão da interface dos servidores *thrift* para *web services* foi utilizada a tecnologia ASP.NET Web API em conjunto com a linguagem C#, ambas da plataforma .NET. A escolha destas tecnologias se deve à possibilidade do desenvolvimento de serviços REST estruturados e escaláveis.

As bibliotecas `ThriftHelper` e `WebApiForThrift` que foram desenvolvidas para auxiliar no desenvolvimento do *middleware* também são escritas na linguagem C#. Estas bibliotecas foram desenvolvidas de forma desacoplada e independentemente do restante da implementação, podendo ser utilizadas em outros projetos.

Também foi desenvolvida uma aplicação *web* que se comunica diretamente com o *middleware*. Esta aplicação é responsável pela interação com os usuários e é através dela que são executadas as funcionalidades de cadastro e demais processos. Ela foi desenvolvida utilizando HTML5, CSS3 e AngularJS, que é um *framework* Javascript que facilita o desenvolvimento e manutenção de aplicações através de modelos como o MVC.

Para armazenamento de dados, em um primeiro momento, foi utilizado o banco de dados MySQL como o modelo relacional. Para alguns processos executados ao longo do ciclo de vida do *middleware*, este modelo demonstrou-se inadequado e, dessa forma, a camada de persistência de dados foi substituída para utilizar MongoDB, um banco de dados não-relacional.

A ferramenta utilizada para codificação foi o Visual Studio 2015, que oferece total suporte às tecnologias .NET e também para o desenvolvimento com tecnologias *web*. Para facilitar a manutenção do banco de dados foi utilizado o MongoLab, um serviço que oferece o

armazenamento e outras funcionalidades do MongoDB na nuvem, sem a necessidade de maiores configurações.

3.3.2 Etapas do desenvolvimento

Nesta seção são apresentadas detalhadamente as etapas do desenvolvimento deste trabalho, desde os primeiros testes com as tecnologias utilizadas até a implementação do *middleware* final.

3.3.2.1 Primeiros passos com o *thrift*

A proposta deste trabalho consiste na implementação de um *middleware* para conversão de interfaces e, para tal, é necessário que se tenha conhecimento da estrutura de ambos os lados envolvidos. Dessa forma, o primeiro passo após a definição dos objetivos foi estudar de forma mais técnica o funcionamento do *thrift*.

Conforme já apresentado na fundamentação teórica, o *thrift* possui uma IDL e a partir dos códigos descritos nela é possível gerar os fontes nas linguagens de programação suportadas. Os códigos gerados devem ser incluídos nas implementações do servidor e dos clientes, para que eles conheçam a estrutura de objetos e serviços com a qual a comunicação será realizada. Para entender de forma clara todos estes processos, as primeiras implementações realizadas foram as de um servidor e um cliente *thrift*. O Quadro 5 contém o código da IDL *thrift* utilizado para implementação.

Quadro 5 - Código da IDL *thrift* para implementação do cliente e servidor

```
1  namespace * Object.Generated
2
3  struct Message
4  {
5      1: i32 id,
6      2: string content
7  }
8
9  service MessageService
10 {
11     list<Message> getMessages();
12 }
```

A linha 1 declara o *namespace* para o qual as classes e demais estruturas são geradas. O trecho de código entre as linhas 3 e 7 contém a declaração de uma *struct* que se chama `Message` e contém os atributos `id` e `content`. Na geração do código através do *thrift* essa *struct* resulta, no caso do C#, em uma classe que também possui o nome `Message` e contém as propriedades `Id` (do tipo *int*) e `Content` (do tipo *string*). O trecho de código entre as linhas 9 e 12 declara um *service* com o nome `MessageService` e uma função `getMessages` cujo o

retorno é uma lista de objetos do tipo `Message`, declarado anteriormente. Na geração de código C# este *service* resulta em uma classe de mesmo nome. Esta classe contém a definição de uma interface que possui o método `getMessages`. Na implementação do servidor essa interface deve ser utilizada, ou seja, o método `getMessages` deve ser estendido e implementado. Na implementação do cliente a interface também será utilizada para que seja possível realizar a chamada deste método no servidor.

A geração das classes em C# é efetuada a partir do executável do *thrift* através de uma linha de comando cujo a estrutura é demonstrada no Quadro 6.

Quadro 6 – Linha de comando do *thrift* para geração de código

```
1  --gen "csharp" -out "c:\temp" "c:\idl_code.thrift"
```

Desmontando a linha de comando temos:

a) `--gen`: comando reservado do *thrift* para a geração de código;

b) `"csharp"`: a linguagem para qual o código será gerado (C#);

c) `-out "c:\temp`: o parâmetro que define o destino dos arquivos gerados;

d) `"c:\idl_code.thrift"`: o caminho do arquivo que contém a IDL *thrift*.

Iniciando pelo desenvolvimento do servidor, os arquivos gerados pela linha de comando executada devem ser adicionados ao projeto. Com os arquivos associados, o procedimento seguinte é realizar a implementação da interface de `MessageService`, que foi declarado na IDL, conforme apresentado no Quadro 7.

Quadro 7 – Implementação em C# da interface gerada pelo *thrift*

```
1  public class MessageServiceHandler : MessageService.Iface
2  {
3      List<Message> _messages;
4
5      public MessageServiceHandler()
6      {
7          _messages = new List<Message>();
8          _messages.Add(new Message() { Id = 1, Content = "Hello!" });
9          _messages.Add(new Message() { Id = 2, Content = "Bye bye!" });
10     }
11
12     public List<Message> getMessages()
13     {
14         Console.WriteLine("getMessages() called!");
15         return _messages;
16     }
17 }
```

Na linha 1, a classe `MessageServiceHandler` estende `MessageService.Iface`, que é a interface que contém a definição do método `getMessages` e foi gerada pelo *thrift*. Na linha 3 é declarada uma lista de objetos do tipo `Message`, que também se refere à *struct* que foi compilada pelo *thrift*. O trecho entre as linhas 5 e 10 se refere ao construtor da classe, onde a

lista é inicializada e são incluídos dois itens. Entre as linhas 12 e 16 é feita a implementação do método `getMessages`, estendido da interface `MessageService.Iface` e que retorna a lista de objetos do tipo `Message`.

A classe `MessageServiceHandler` é utilizada no código que inicializa o servidor e é ela quem executa as requisições feitas pela aplicação cliente. O Quadro 8 apresenta o código para rodar o servidor *thrift* disponibilizando as funcionalidades do *handler*.

Quadro 8 – Implementação em C# do servidor *thrift*

```
1  public static void Main(string[] args)
2  {
3      var handler = new MessageServiceHandler();
4      var processor = new MessageService.Processor(handler);
5      var transport = new TServerSocket(9090);
6      var server = new TThreadPoolServer(processor, transport);
7
8      Console.WriteLine("Server started!");
9
10     server.Serve();
11 }
```

Na linha 3 é criada uma instância de `MessageServiceHandler`, que contém o método `getMessages` implementado. Na linha 4 é instanciado um objeto nativo do *thrift* que faz o processamento do *handler* para que seja possível executar suas funcionalidades. Na linha 5 é criado um objeto de transporte, também nativo do *thrift*, que associa a porta 9090 ao *socket* do servidor. É através desta porta e do endereço no qual o servidor está rodando que o cliente é capaz de se comunicar posteriormente. Na linha 6 é instanciado o objeto que representa o servidor *thrift* e este recebe como parâmetro as configurações instanciadas nas linhas anteriores. Na linha 10, quando o método `Serve` é executado, o servidor *thrift* fica disponível e a partir deste momento pode ser acessado por uma aplicação cliente.

Para implementação do cliente *thrift*, assim como para o servidor, os arquivos C# gerados a partir da IDL devem ser associados ao projeto. O código para rodar um cliente *thrift* é apresentado no Quadro 9.

Quadro 9 – Implementação em C# do cliente *thrift*

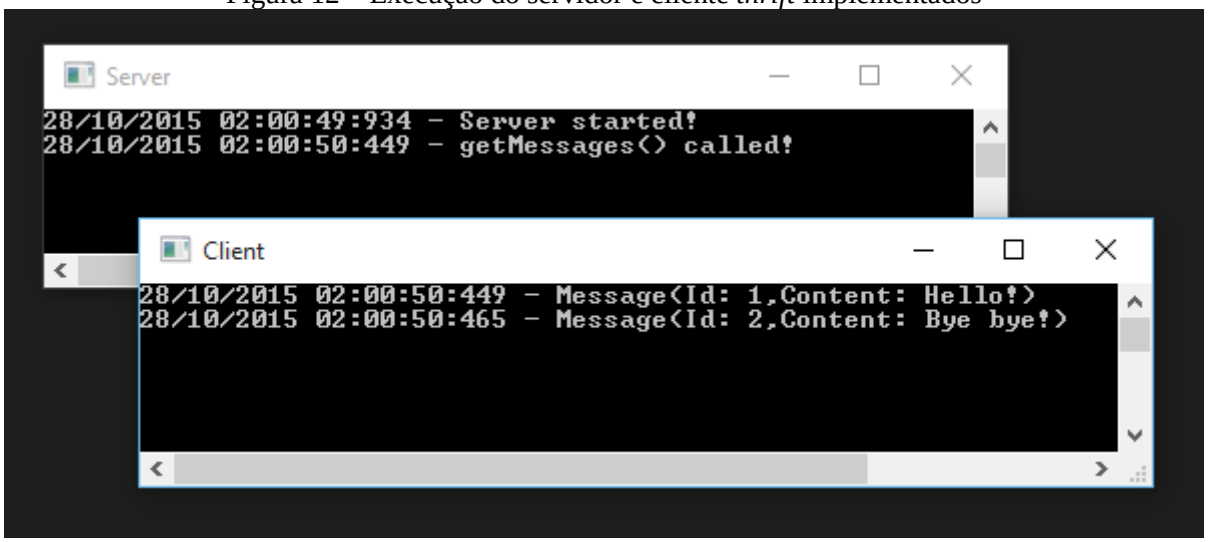
```
1  public static void Main(string[] args)
2  {
3      var transport = new TSocket("localhost", 9090);
4      var protocol = new TBinaryProtocol(transport);
5      var client = new MessageService.Client(protocol);
6      transport.Open();
7      var messages = client.getMessages();
8      messages.ForEach(x => Console.WriteLine(x.ToString());
9  }
```

Na linha 3 é feita a criação de um objeto de transporte, que recebe como parâmetros o endereço e a porta nos quais o servidor *thrift* está sendo executado. Na linha 4 é criado o

protocolo sobre o qual a comunicação com o servidor é feita, neste exemplo, um protocolo binário. Na linha 5 é criado o objeto que representa o cliente e este é criado a partir da classe `MessageService`, gerada anteriormente na compilação da IDL. Na linha 6 é executado o método `Open` do objeto de transporte, que neste momento, possui referência ao objeto cliente e, consequentemente, aos métodos implementados pela classe `MessageService`. O cliente consegue então executar os métodos da forma como é feito na linha 7, onde o método `getMessages` é invocado e os objetos retornados por ele são armazenados na variável `messages`.

A Figura 12 demonstra a execução das implementações do servidor e do cliente *thrift*. Na tela do servidor (`Server`) é possível visualizar as mensagens identificando o momento em que ele foi iniciado e o momento em que o método `getMessages` foi invocado pelo cliente. No cliente (`Client`) é possível visualizar o momento e as informações recebidas do servidor.

Figura 12 – Execução do servidor e cliente *thrift* implementados



3.3.2.2 ASP.NET Web API e carregamento dinâmico de *controllers*

O ASP.NET Web API é um *framework* da plataforma .NET que permite e facilita a implementação de serviços HTTP e é ideal para o desenvolvimento de aplicações REST. Para desenvolver um serviço Web API são necessárias algumas poucas configurações e a implementação das *controllers*. As *controlllers*, já apresentados anteriormente, contém as implementações de métodos e cada método é responsável por realizar uma ação quando uma requisição HTTP é feita para o endereço associado. O Quadro 10 mostra a implementação de uma *controller* simples.

Quadro 10 – Exemplo de implementação de uma *controller* Web API

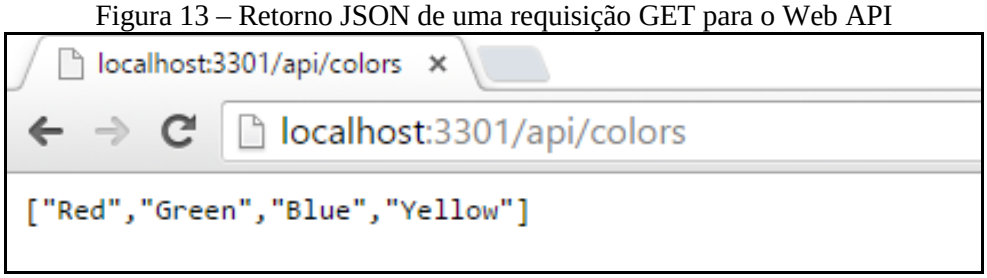
```
1  public class ColorController : ApiController
2  {
3      private IList<string> _colors;
4
5      public ColorController()
6      {
7          _colors = new List<string>() { "Red", "Green", "Blue", "Yellow" };
8      }
9
10     [HttpGet]
11     [Route("api/colors")]
12     public IHttpActionResult GetColors()
13     {
14         return Ok(_colors);
15     }
16 }
```

Na linha 1 é declarada a classe `ColorController`, que estende a classe `ApiController`. A classe `ApiController` é uma estrutura nativa do *framework* que contém as implementações base para uma *controller*. Na linha 3 é declarada a lista `_colors`, do tipo *string*, que é responsável por armazenar nomes de cores. No trecho entre as linhas 5 e 8 a lista `_colors` é inicializada e recebe alguns valores dentro do construtor da classe.

No trecho entre as linhas 10 e 15 é implementado o método `GetColors`. O atributo `HttpGet` da linha 10 define que o método será invocado quando for feita uma requisição HTTP do tipo GET para o endereço definido no atributo `Route` da linha 11. A linha 12 declara o método com o retorno IHttpActionResult, que também é nativo do *framework*. Por fim, na linha 14, o método retorna uma resposta HTTP com o status `Ok` e a lista de valores contidos em `_colors`.

Ao ser executado um serviço Web API que contém a *controller* apresentada no Quadro 10, a rota definida para o método `GetColors` torna-se disponível para acesso. Se o serviço foi configurado para retornar as respostas em formato JSON e for feita uma requisição HTTP do tipo GET para a rota `api/colors`, um conteúdo igual ao da Figura 13 será recebido como resposta.

Figura 13 – Retorno JSON de uma requisição GET para o Web API



As *controllers* normalmente são implementadas dentro do próprio projeto Web API e, nestes casos, tudo é resolvido em tempo de compilação, resultando em apenas um *assembly*

ao final do processo. Porém existem alguns casos em que é necessário o carregamento de *controllers* adicionais em tempo de execução sem que haja a interrupção do serviço.

Dentro da plataforma .NET existe uma série de estruturas que permitem o carregamento de *assemblies* em tempo de execução através do código fonte. Com estas estruturas é possível injetar *controllers* de um *assembly* externo no *assembly* que está executando o serviço Web API. Neste trabalho esta técnica foi implementada na biblioteca `WebApiForThrift` para os *assemblies* gerados pelo processo de *build*.

3.3.2.3 Desenvolvimento das bibliotecas `ThriftHelper` e `WebApiForThrift`

Uma breve descrição da utilidade das bibliotecas `ThriftHelper` e `WebApiForThrift` já foi feita na especificação delas. Nesta seção são abordados os aspectos técnicos e a implementação destas bibliotecas.

A biblioteca `ThriftHelper` basicamente encapsula o uso do executável do *thrift* para que suas funcionalidades possam ser utilizadas através de código C#. Na biblioteca foi implementada a classe `Factory` que contém os métodos de execução do *thrift*. O principal destes métodos é o `Execute`, que é utilizado para geração de código no processo de *build* do *middleware* e é apresentado no Quadro 11.

Quadro 11 – Método `Execute` da biblioteca `ThriftHelper`

```
1  public DirectoryModel Execute()
2  {
3      string formattedLanguage = FormatLanguage(_language);
4      string guid = Guid.NewGuid().ToString();
5      string tempPath = Path.GetTempPath();
6      string tempPathOut = Path.Combine(tempPath, guid, "Out");
7      string tempPathMap = Path.Combine(tempPath, guid, "Map");
8      string mapFilePath = Path.Combine(tempPathMap, "Code.thrift");
9      string arguments = $"--gen \"{formattedLanguage}\" -out \"{tempPathOut}\" \"{mapFilePath}\"";
10
11     Directory.CreateDirectory(tempPathMap);
12     File.WriteAllText(mapFilePath, _thriftCode);
13
14     Directory.CreateDirectory(tempPathOut);
15
16     ExecuteThrift(arguments);
17
18     var directoryModel = new DirectoryModel(guid);
19     directoryModel.FillFromFileSystem(tempPathOut);
20
21     return directoryModel;
22 }
```

Na linha 1 é feita a declaração do método `Execute`, que não aceita nenhum parâmetro pois as informações necessárias são passadas no construtor da classe. O método tem como retorno um objeto da classe `DirectoryModel`, que contém propriedades que representam

arquivos e subdiretórios e foi implementada dentro da biblioteca para abstrair, em memória, uma estrutura de diretórios do sistema de arquivos. Esta representação em memória foi implementada porque o executável do *thrift* é baseado em sistema de arquivos e, em tempo de execução, é inviável a leitura e passagem por parâmetro neste formato.

No trecho de código entre as linhas 3 e 9 são declaradas as variáveis responsáveis por auxiliar na construção dos parâmetros que devem ser passados para execução do *thrift*. Estes parâmetros ficam armazenados na variável `arguments`. A variável `_language` utilizada na linha 3 é recebida como parâmetro no construtor da classe e informa a linguagem para qual deve ser gerado o código. Na linha 11 é criado o diretório onde o executável busca o arquivo com o código escrito na IDL *thrift*, enquanto na linha 12 este arquivo é criado com o conteúdo da variável `_thriftCode` que também foi recebida como parâmetro no construtor da classe.

Na linha 14 é criado o diretório de destino dos arquivos gerados pelo *thrift*. Na linha 16 o método `ExecuteThrift` é invocado recebendo a variável `arguments` por parâmetro. O método `ExecuteThrift` é responsável por utilizar o executável do *thrift* para gerar os arquivos com base nos parâmetros recebidos. Nas linhas 18 e 19 um objeto `DirectoryModel` é criado com base no diretório onde os arquivos foram gerados e na linha 21 este objeto é retornado pelo método.

A biblioteca `WebApiForThrift` é responsável por gerar as *controllers* do serviço Web API que fazem a comunicação com um servidor *thrift*. Para realizar este processo a biblioteca precisa montar todos os códigos C# de um cliente *thrift* em tempo de execução, colocá-los dentro de métodos das *controllers* e efetuar a compilação que resulta em um *assembly*.

O processo de geração do *assembly* nesta biblioteca acontece em dois estágios. O primeiro estágio é a construção das *controllers*, que é efetuado quando o método `BuildControllers` da classe `Factory` é invocado. O Quadro 12 apresenta a implementação deste método.

Quadro 12 – Método `BuildControllers` da biblioteca `WebApiForThrift`

```
1  public IList<ControllerModel> BuildControllers()
2  {
3      var thriftFilesContent = BuildThriftFilesContent(_thriftCode);
4      var referencedAssemblies = new List<string>() { _thriftDllFile };
5
6      var compilerResults = Compile(thriftFilesContent, referencedAssemblies, true);
7      var types = compilerResults.CompiledAssembly.GetTypes().ToList();
8      var objectClasses = types.Where(x => x.IsClass && x.GetInterface("TBase") != null);
9      var parentClasses = types.Where(x => x.IsClass && x.GetNestedType("Iface") != null);
10
11     Objects = new List<ObjectModel>();
12     Controllers = new List<ControllerModel>();
13
14     foreach (var objectClass in objectClasses)
15     {
16         if (!objectClass.Name.EndsWith("_args") && !objectClass.Name.EndsWith("_result"))
17         {
18             var obj = new ObjectModel(objectClass);
19             Objects.Add(obj);
20         }
21     }
22
23     foreach (var parentClass in parentClasses)
24     {
25         var controller = new ControllerModel(parentClass, parentClass.GetNestedType("Iface"));
26         Controllers.Add(controller);
27     }
28
29     return Controllers;
30 }
```

Na linha 1 é declarado o método `BuildControllers`, cujo o retorno é uma lista de objetos da classe `ControllerModel`. As classes `ControllerModel` e `ObjectModel` foram implementadas para abstrair, respectivamente, as informações das *controllers* e dos objetos utilizados ao longo do processo de geração de código.

Na linha 3, o método `BuildThriftFilesContent` é executado enviando o parâmetro `_thriftCode`, que é uma variável recebida no construtor da classe `Factory` e contém o código em IDL *thrift*. Este método utiliza a biblioteca `ThriftHelper` para gerar os códigos em C# e os armazena na variável `thriftFilesContents`. Na linha 6 os códigos gerados pelo *thrift* são executados pelo compilador do .NET e o resultado é armazenado na variável `compilerResults`. Com o resultado da compilação é possível, através de código C#, acessar as classes e demais estruturas que foram geradas. Utilizando esta funcionalidade, entre as linhas 7 e 9 é feita a interpretação do código gerado e as classes que possuem relevância para o processo são armazenadas em variáveis.

Nos trechos entre as linhas 14 e 21 e entre as linhas 23 e 27 as classes que foram armazenadas anteriormente são filtradas para identificar se elas se referem a *structs* ou *services* dentro na IDL do *thrift*. As que forem *structs* são abstraídas para objetos da classe

`ObjectModel` e armazenadas na propriedade `Objects`, da instância de `Factory` em questão, para que se tenha referência dos objetos que são utilizados pelas *controllers*. As que forem *services* são abstraídas para objetos da classe `ControllerModel` pois estas contêm as definições dos métodos do servidor *thrift* e é para estas que as *controllers* do Web API são geradas.

O processo de construção do *assembly* é feito em dois estágios para que possa haver modificações externas na estrutura das *controllers*. A aplicação que está utilizando a biblioteca `WebApiForThrift` inicialmente solicita a construção das *controllers* e recebe como retorno uma lista de objetos da classe `ControllerModel`. A aplicação neste momento pode alterar informações como a rota e o tipo de requisição HTTP dos métodos das *controllers* contidas na lista. Após as alterações é iniciado o segundo estágio do processo, quando o método `BuildAssembly`, também da classe `Factory`, é invocado recebendo a lista de *controllers* alteradas como parâmetro. O Quadro 13 apresenta a implementação do método `BuildAssembly`.

Quadro 13 – Método `BuildAssembly` da biblioteca `WebApiForThrift`

```
1  public AssemblyModel BuildAssembly(ConfigurationModel config, IList<ControllerModel> controllers)
2  {
3      var filesContent = BuildThriftFilesContent(_thriftCode);
4
5      foreach (var controller in controllers)
6      {
7          filesContent.Add(BuildControllerContent(config, controller));
8      }
9
10     var referencedAssemblies = new List<string>()
11     {
12         "Newtonsoft.Json.dll",
13         "System.dll",
14         "System.Web.Http.Cors.dll",
15         "System.Web.Http.dll",
16         _thriftDllFile
17     };
18
19     var compilerResults = Compile(filesContent, referencedAssemblies, false);
20     var bytes = File.ReadAllBytes(compilerResults.PathToAssembly);
21     var base64 = Convert.ToBase64String(bytes);
22
23     return new AssemblyModel(base64);
24 }
```

Na linha 1 é declarado o método `BuildAssembly`, que recebe como parâmetros um objeto do tipo `ConfigurationModel` e a lista de *controllers* já ajustada pela aplicação. A classe `ConfigurationModel` contém os dados necessários para realizar o acesso ao servidor *thrift*, uma vez que as *controllers* devem implementar clientes *thrift* que se comuniquem com esse servidor para fazer o processamento das solicitações.

Na linha 3, a biblioteca `ThriftHelper` é utilizada novamente, através do método `BuildThriftFilesContent`, para obter as classes geradas pelo *thrift* e as armazena na variável `filesContent`. Em seguida, no trecho entre as linhas 5 e 8, a lista de *controllers* recebida por parâmetro é iterada e para cada item o método `BuildControllerContent` é invocado e o seu resultado é adicionado na variável `filesContent`. O método `BuildControllerContent` é responsável por montar o código C# das *controllers* em um formato semelhante ao do Quadro 10, sendo que cada método da *controller* implementa um cliente *thrift* semelhante ao do Quadro 9 para executar uma função do servidor *thrift*.

Na linha 19 o compilador do .NET novamente é utilizado para compilar os códigos contidos na variável `filesContent`. Neste momento a variável contém todos os códigos gerados pelo *thrift* e os códigos das *controllers* Web API que foram gerados pelo método `BuildControllerContent`. A compilação resulta em um *assembly* que é salvo no sistema de arquivos como um arquivo DLL e em seguida, nas linhas 20 e 21, é lido, convertido para o formato base64 e retornado pelo método para ser gerenciado em memória sem manter vínculo com o sistema de arquivos.

Ao final deste processo, o *assembly* obtido contém toda a estrutura necessária para que o serviço Web API que for utilizá-lo possa se comunicar com o servidor *thrift*. Quando o *assembly* for carregado, as *controllers* podem ser acessadas através de requisições HTTP e internamente fazer as chamadas *thrift*, tratar o resultado do processamento e retornar uma resposta HTTP para a requisição.

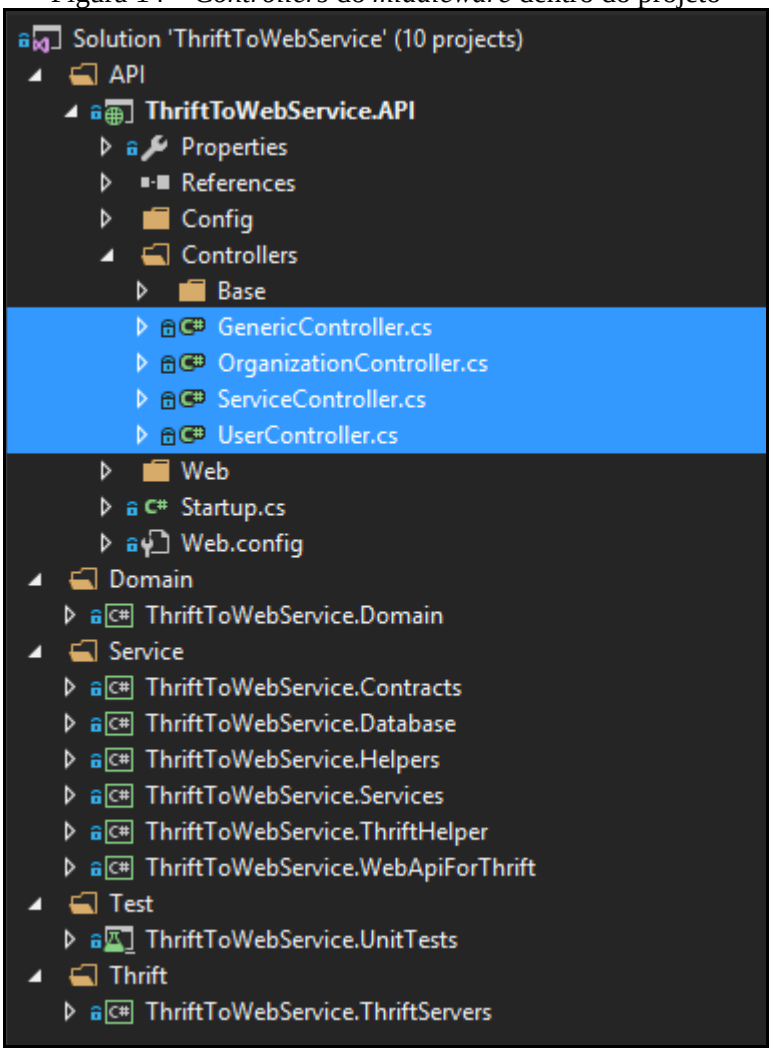
3.3.2.4 Desenvolvimento do *middleware*

O *middleware* foi desenvolvido como um serviço ASP.NET Web API, com o conceito de *web service* REST e utilizando o protocolo HTTP para comunicação. Dentro da implementação esta é a interface final de todos os outros pacotes e funcionalidades que foram desenvolvidos. Para interagir com o *middleware* foi desenvolvida uma aplicação *web* que viabiliza uma interface amigável para o usuário.

Além de ser utilizada como *back-end* da aplicação *web*, o serviço Web API também pode ser acessado através de requisições HTTP de outros sistemas que possam vir a implementar suas funcionalidades. Também é neste serviço que os *assemblies* gerados pelo processo de *build* são carregados e passam a ficar disponíveis para consumo externo, novamente, através de requisições HTTP.

Dentro da estrutura do projeto foram desenvolvidas 4 *controllers* que são responsáveis por tratar as requisições do *middleware*. Estas *controllers* são evidenciadas na Figura 14.

Figura 14 – *Controllers* do *middleware* dentro do projeto



A seguir são descritas brevemente as responsabilidades de cada uma das *controllers* implementada:

a) `GenericController`: utilizado para algumas consultas genéricas;

b) `OrganizationController`: utilizado para as funcionalidades de busca, cadastro e edição de organizações;

c) `ServiceController`: utilizado para as funcionalidades de busca, cadastro e edição de serviços. Nesta *controller* também são implementadas as funcionalidades de *build*, uma vez que estas são realizadas a partir dos serviços;

d) `UserController`: utilizado para as funcionalidades de controle dos usuários que utilizam o *middleware*.

As funcionalidades que envolvem busca, cadastro e edição de entidades do *middleware* realizam estas operações persistindo em um banco de dados MongoDB. Todas as operações utilizadas pelo *middleware* estão encapsuladas no pacote `Services`, que é responsável por

toda a regra de negócios da implementação e, inclusive, pela persistência com o banco de dados.

Além das funcionalidades básicas de cadastro e manutenção das entidades, o pacote `Services` também encapsula as operações de *build*. Para realizar estas operações ele utiliza os recursos das bibliotecas `ThriftHelper` e `WebApiForThrift`. Conforme apresentado no diagrama de sequência da Figura 11, ao longo das etapas do processo de *build* o estado de algumas entidades é gravado em banco para manter o histórico. O Quadro 14 apresenta o método `RunAsyncDTO` que implementa a primeira etapa do processo de *build*.

Quadro 14 – Método `RunAsyncDTO` do *middleware*

```
1  public async Task<BuildDTO> RunAsyncDTO(string idService)
2  {
3      var serviceDTO = _serviceBusiness.GetByIdAsyncDTO(idService);
4      var organizationDTO = _organizationBusiness.GetByIdAsyncDTO(serviceDTO.Owner.ToString());
5      var factory = new WebApiForThriftFactory(serviceDTO.ThriftFile.Content);
6      var controllers = factory.BuildControllers();
7
8      var buildDTO = new BuildDTO(serviceDTO, string.Empty, factory.Objects, controllers);
9
10     foreach (var controller in buildDTO.Controllers)
11     {
12         controller.Namespace = FormatNamespace(serviceDTO, organizationDTO);
13
14         foreach (var method in controller.Methods)
15         {
16             method.HttpRoute = FormatRoute(serviceDTO, organizationDTO, controller, method);
17         }
18     }
19
20     _builds.InsertOneAsync(buildDTO);
21
22     return buildDTO;
23 }
```

Na linha 1, o método `RunAsyncDTO` é declarado recebendo como parâmetro o `idService`, que se refere ao identificador do serviço para o qual o *build* deve ser executado. Na linha 3 o método `_serviceBusiness.GetByIdAsyncDTO` é utilizado para consultar no banco de dados o serviço referente ao `idService` recebido, enquanto na linha 4 é executado o método `_organizationBusiness.GetByIdAsyncDTO` para obter a organização responsável por aquele serviço.

Dentro da entidade de serviço que foi trazida do banco de dados, além dos dados de acesso ao servidor *thrift*, está contido também o código em IDL *thrift*. Na linha 5 é criada uma instância da classe `WebApiForThriftFactory` que se refere à biblioteca `WebApiForThrift`. Esta instância recebe como parâmetro a propriedade `serviceDTO.ThriftFile.Content`, que contém o código em IDL *thrift* do serviço. Na linha 6, o método `BuildControllers` é

executado. O método `BuildControllers` é o mesmo já apresentado anteriormente no Quadro 12.

Na linha 8 é criada uma instância da classe `BuildDTO`, que é a entidade utilizada para persistir no banco de dados as informações de um *build*. Esta entidade recebe como parâmetro o `serviceDTO`, que contém as informações do serviço consultadas no banco de dados e as propriedades `Objects` e `Controllers` da variável `factory`, que após a execução do método `BuildControllers` contém as estruturas geradas pela biblioteca `WebApiForThrift`.

No trecho entre as linhas 10 e 18 as *controllers* e seus respectivos métodos são iterados para alterar o *namespace* da classe e as rotas HTTP dos métodos. Para identificar as *controllers* e endereços de forma única foi definido um padrão em que se usa o código da organização, o código do serviço e um número de versão para nomear as estruturas. O número de versão é um valor numérico que fica armazenado no serviço e é incrementado sempre que um *build* é executado com sucesso para aquele serviço. Na linha 20 o objeto `buildDTO` é inserido no banco de dados e na linha 22 ele é retornado pelo método.

Após a execução do método `RunAsyncDTO` o objeto gerado com as informações do *build* é retornado para o *middleware*. Neste momento o *middleware* possibilita a alteração de algumas informações do *build*, como o tipo de requisição HTTP que deve ser feito para cada método gerado no *build*. Finalizando as alterações necessárias, o *middleware* deve executar a segunda etapa do *build*, que consiste na execução do método `ConfirmAsyncDTO` apresentado no Quadro 15 .

Quadro 15 – Método `ConfirmAsyncDTO` do *middleware*

```
1  public async Task<BuildDTO> ConfirmAsyncDTO(Build build)
2  {
3      var objectId = new ObjectId(build.Id);
4      var buildDTO = await _builds.Find(x => x.Id.Equals(objectId)).FirstOrDefaultAsync();
5
6      if (buildDTO == null)
7          throw new NotFoundException("Build not found.");
8
9      var serviceDTO = await _serviceBusiness.GetByIdAsyncDTO(buildDTO.Service.ToString());
10     var organizationDTO = _organizationBusiness.GetByIdAsyncDTO(serviceDTO.Owner.ToString());
11     var factory = new WebApiForThriftFactory(buildDTO.ThriftFile?.Content);
12     var configuration = new ConfigurationModel(serviceDTO.ServerUrl,
13         serviceDTO.ServerPort,
14         ThriftProtocol.Binary);
15
16     var controllersDTO = buildDTO.Controllers ?? new List<BuildControllerDTO>();
17     var controllersModel = factory.BuildControllers();
18
19     UpdateBuildInfo(build, serviceDTO, organizationDTO, controllersDTO, controllersModel);
20
21     string assemblyFileName = $@"{organizationDTO.Code}
22         .{serviceDTO.Code}
23         .v{serviceDTO.CurrentVersion + 1}
24         .dll".ToLowerInvariant();
25
26     var assembly = factory.BuildAssembly(configuration, controllersModel);
27     var assemblyDTO = new AssemblyDTO(assembly, assemblyFileName, buildDTO);
28     var updateBuildBuilder = Builders<BuildDTO>
29           .Update
30           .Set(x => x.Controllers, controllersDTO)
31           .Set(x => x.ConfirmationDate, DateTime.Now);
32
33     buildDTO = await _builds
34         .FindOneAndUpdateAsync(x => x.Id.Equals(buildDTO.Id), updateBuildBuilder);
35
36     assemblyDTO = await _assemblyBusiness.InsertAsyncDTO(assemblyDTO);
37
38     serviceDTO = await _serviceBusiness
39         .UpdateVersionAsyncDTO(serviceDTO.Id.ToString(), serviceDTO.CurrentVersion + 1);
40
41     _assemblyBusiness.Load(assemblyDTO);
42
43     return buildDTO;
44 }
45
```

Na linha 1, o método `ConfirmAsyncDTO` é declarado recebendo um objeto da classe `Build` como parâmetro. Este objeto contém as informações geradas na execução do método `RunAsyncDTO` com algumas possíveis alterações que possam ter sido efetuadas. Na linha 12 um objeto do tipo `ConfigurationModel` é declarado recebendo como parâmetro as propriedades `ServerUrl` e `ServerPort` do serviço referente ao *build*. Este objeto é utilizado no momento da geração do *assembly* e contém as informações necessárias para comunicação com o servidor *thrift*.

Na linha 17, o método `BuildControllers` da biblioteca `WebApiForThrift` é executado para novamente gerar as *controllers* a partir do código da IDL. Esta geração de código é refeita pois é inviável guardar todos os arquivos gerados na primeira etapa do *build*. Na linha 19 o método `UpdateBuildInfo` atualiza as *controllers* geradas com as informações alteradas que foram recebidas por parâmetro.

Na linha 26 o método `BuildAssembly`, também da biblioteca `WebApiForThrift`, é invocado passando por parâmetro o objeto `configuration`, que contém as informações de acesso do servidor *thrift*, e o objeto controllersModel, que contém as informações das *controllers*. Este método, já apresentado no Quadro 13, é responsável por gerar o código das *controllers* e efetuar a compilação, retornando um *assembly* para o *middleware*.

O retorno do *assembly* para o *middleware* significa que o processo de *build* foi completado com sucesso. No trecho entre as linhas 28 e 34 a entidade que representa o *build* tem suas informações atualizadas no banco de dados. Na linha 36 um objeto que representa o *assembly* gerado também é salvo no banco de dados e na linha 38 e 39 a entidade que representa o serviço é atualizada no banco de dados, incrementando sua versão.

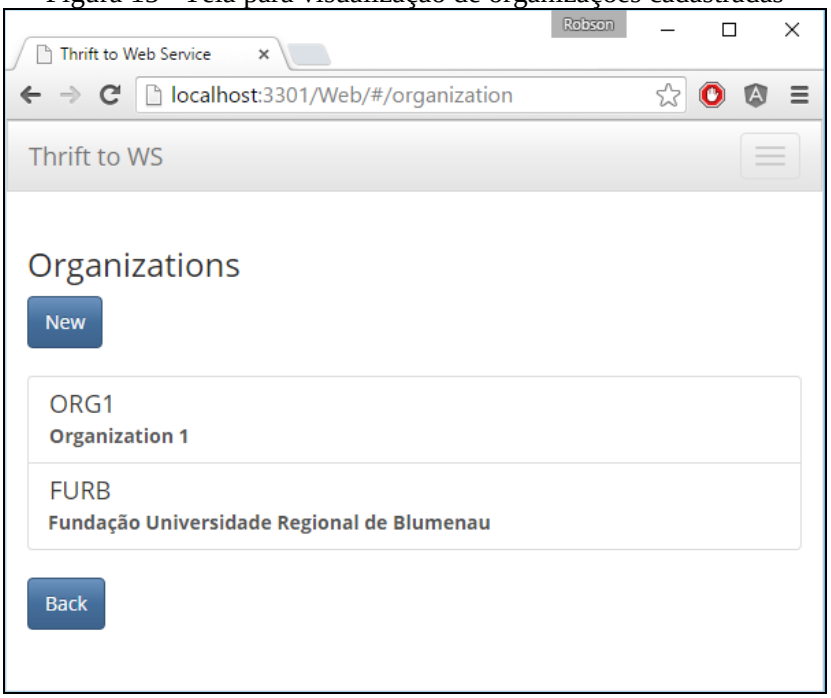
Na linha 41 acontece a execução do método `_assemblyBusiness.Load`. Este método recebe como parâmetro o *assembly* gerado e, utilizando os recursos do *framework* do .NET, carrega a nova DLL na aplicação que está executando, neste caso, o *middleware*. A partir deste momento o *middleware* possui referência para todas as classes de objetos e *controllers* que foram geradas pelo processo de *build.* Com estas referências passa a ser possível acessar, através de requisições HTTP para as rotas definidas, os recursos do servidor *thrift* para qual o *build* foi executado.

3.3.3 Operacionalidade da implementação

Esta seção apresenta a operacionalidade do *middleware* implementada através da aplicação *web* que foi desenvolvida para interação do usuário. A aplicação foi implementada utilizando HTML5 e AngularJS e se comunica com as *controllers* do serviço Web API do *middleware* através de requisições HTTP.

A Figura 15 apresenta a tela onde são listadas as organizações cadastradas pelo usuário logado e cada item da lista é um *link* para uma tela com os detalhes da respectiva organização. A tela também possui um botão `New`, que redireciona o usuário para um formulário onde ele pode efetuar o cadastro de novas organizações.

Figura 15 - Tela para visualização de organizações cadastradas

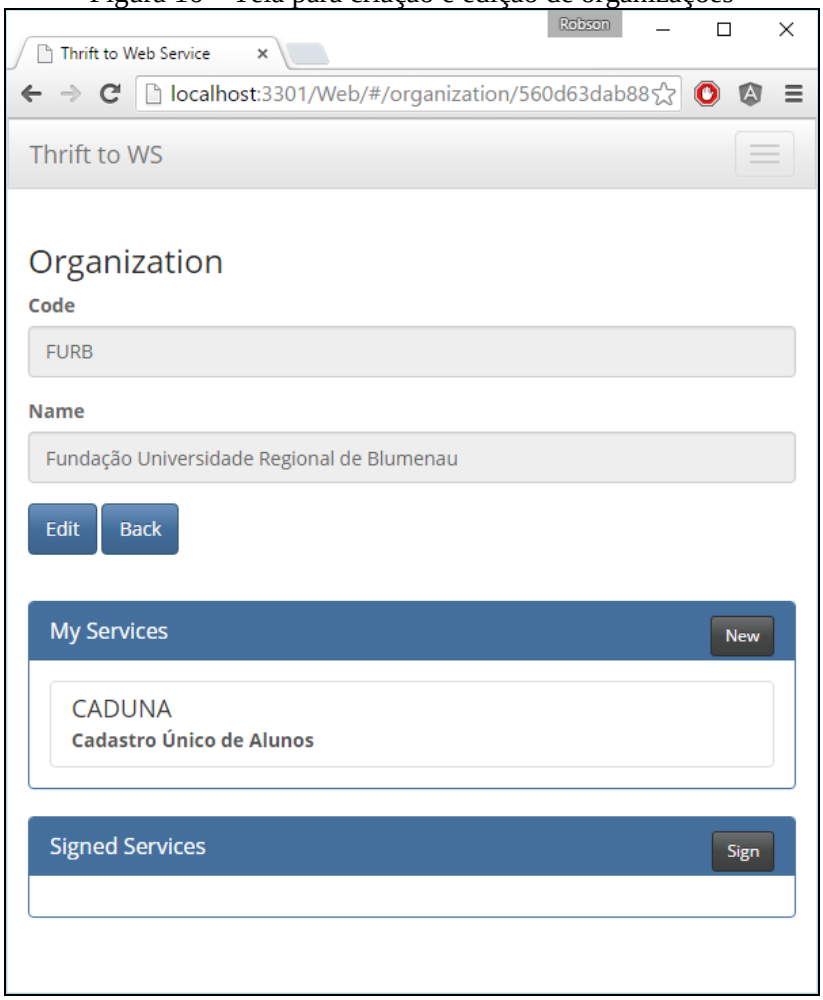


A Figura 16 apresenta a tela que contém o formulário para cadastro e edição de organizações. Os campos `Code` e `Name` são utilizados para identificação da organização e representam, respectivamente, um código e um nome atribuídos pelo usuário responsável pelo cadastro. O botão `Edit` habilita o formulário para edição e habilita um botão `Save`, que possibilita a gravação das alterações. O botão `Back` retorna o usuário para a tela de listagem de organizações.

No quadro `My Services` é apresentada uma lista com os serviços cadastrados pelo usuário para a organização. Estes serviços representam servidores *thrift* que são de responsabilidade da organização em questão. Cada item da lista é um *link* que redireciona o usuário para uma tela com os detalhes do respectivo serviço, enquanto o botão `New` redireciona o usuário para realização do cadastro de um novo serviço.

No quadro `Signed Services` é apresentada uma lista com os serviços nos quais o usuário associou a organização. Estes serviços representam servidores *thrift* de outras organizações e, ao serem associados à organização em questão, permitem que ela visualize as informações para utilização das funcionalidades daquele servidor. Cada item da lista é um *link* que redireciona o usuário para uma tela com os detalhes do respectivo serviço, enquanto o botão `Sign` permite que o usuário realize uma pesquisa pelo código de um serviço e se associe a ele.

Figura 16 – Tela para criação e edição de organizações

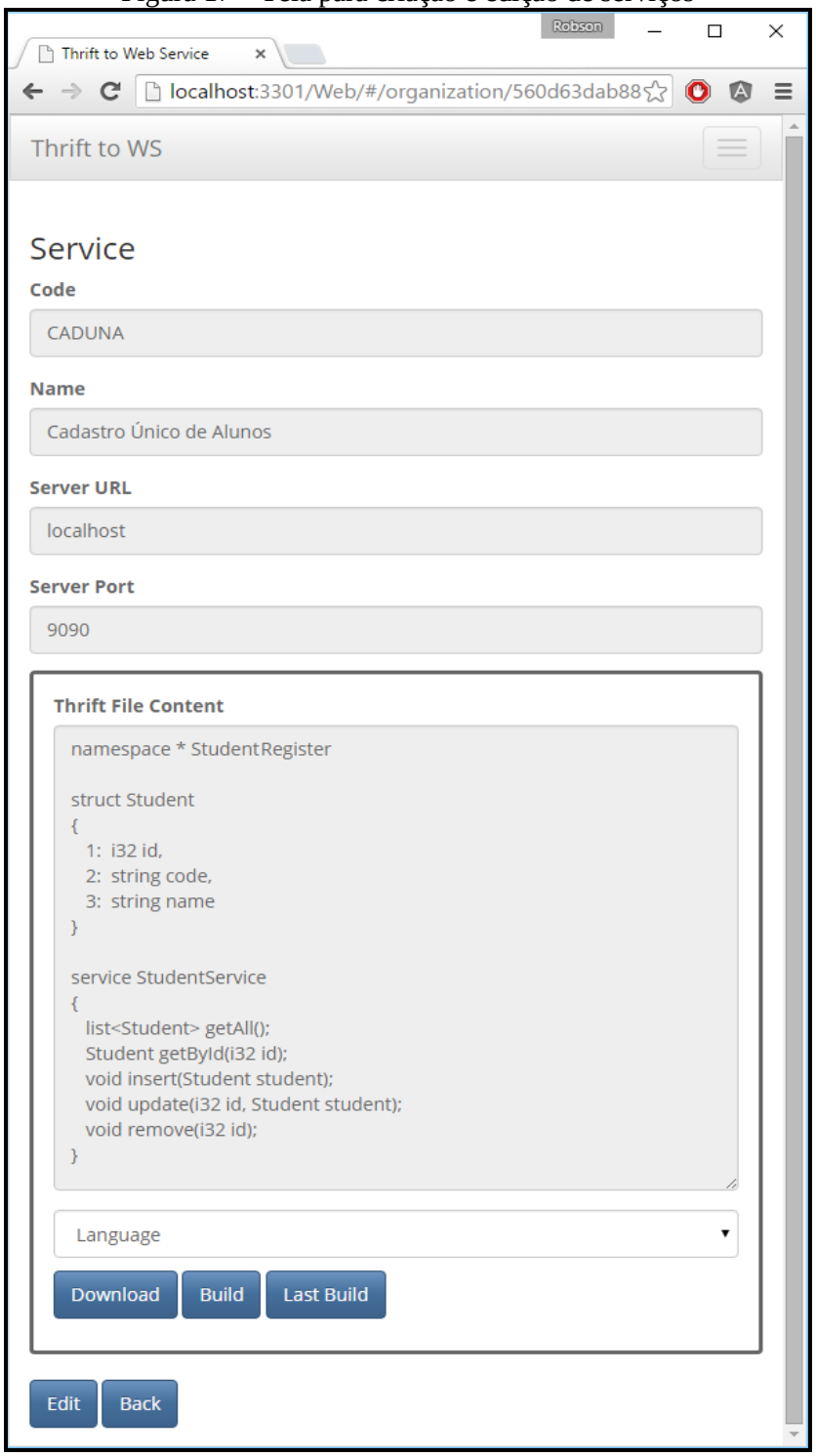


A Figura 17 apresenta a tela que contém o formulário para criação e edição de serviços. Os campos `Code` e `Name` são utilizados para identificação do serviço, assim como na organização. Os campos `Server URL` e `Server Port` se referem, respectivamente, ao endereço e porta para estabelecer comunicação com o servidor *thrift*. O campo `Thrift File Content` contém o código em IDL *thrift* que foi utilizado para implementar as funcionalidades do servidor *thrift*.

O botão `Download` permite que o usuário baixe os códigos fontes gerados a partir do código em IDL cadastrado. O botão `Build` inicia um processo de *build* para o serviço em questão, enquanto o botão `Last Build` permite ao usuário visualizar os resultados do último *build* executado com sucesso.

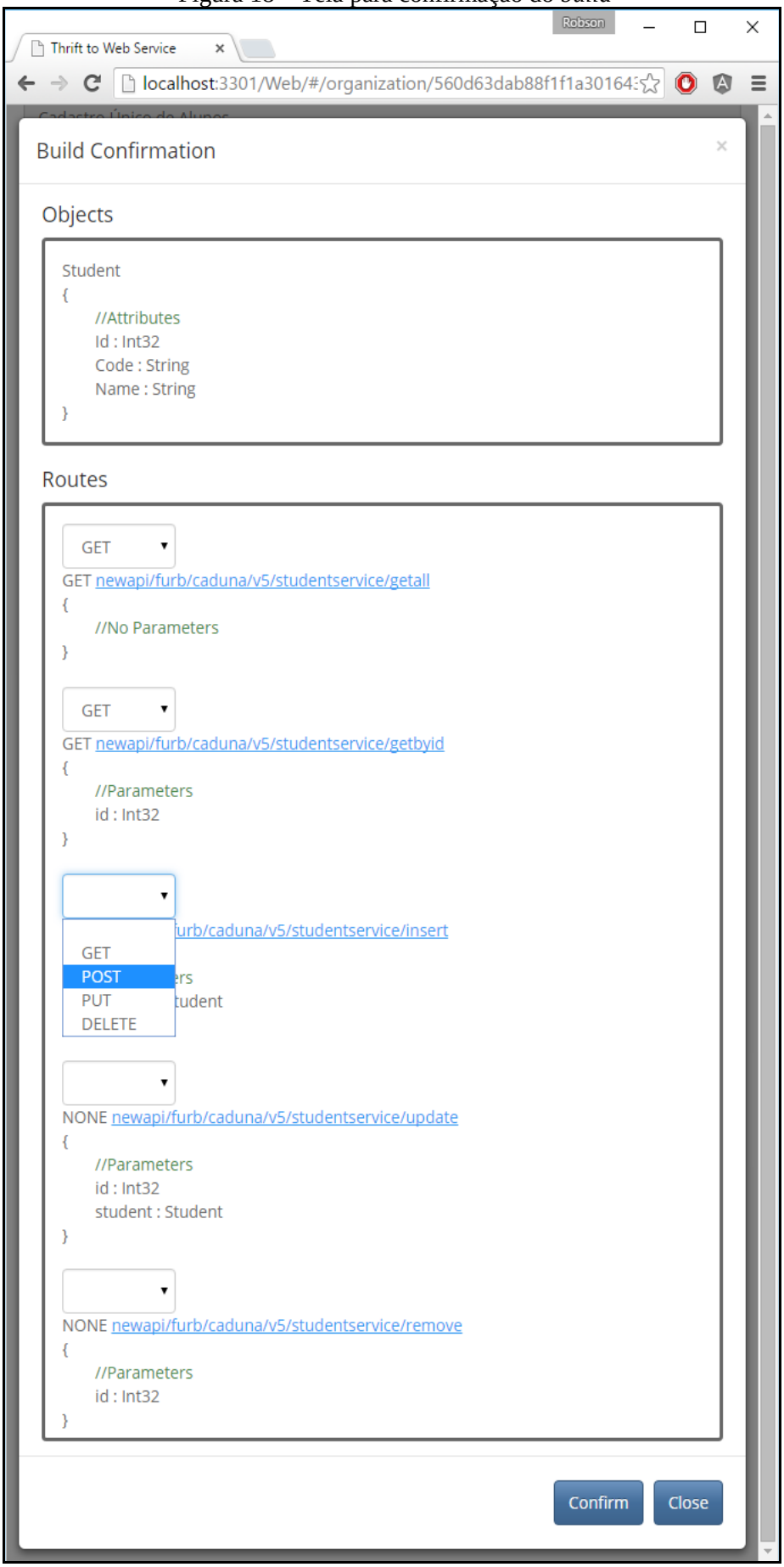
O botão `Edit` habilita o formulário para edição e adiciona um botão `Save`, que possibilita a gravação das alterações. O botão `Back` retorna o usuário para a tela da organização responsável pelo serviço.

Figura 17 – Tela para criação e edição de serviços



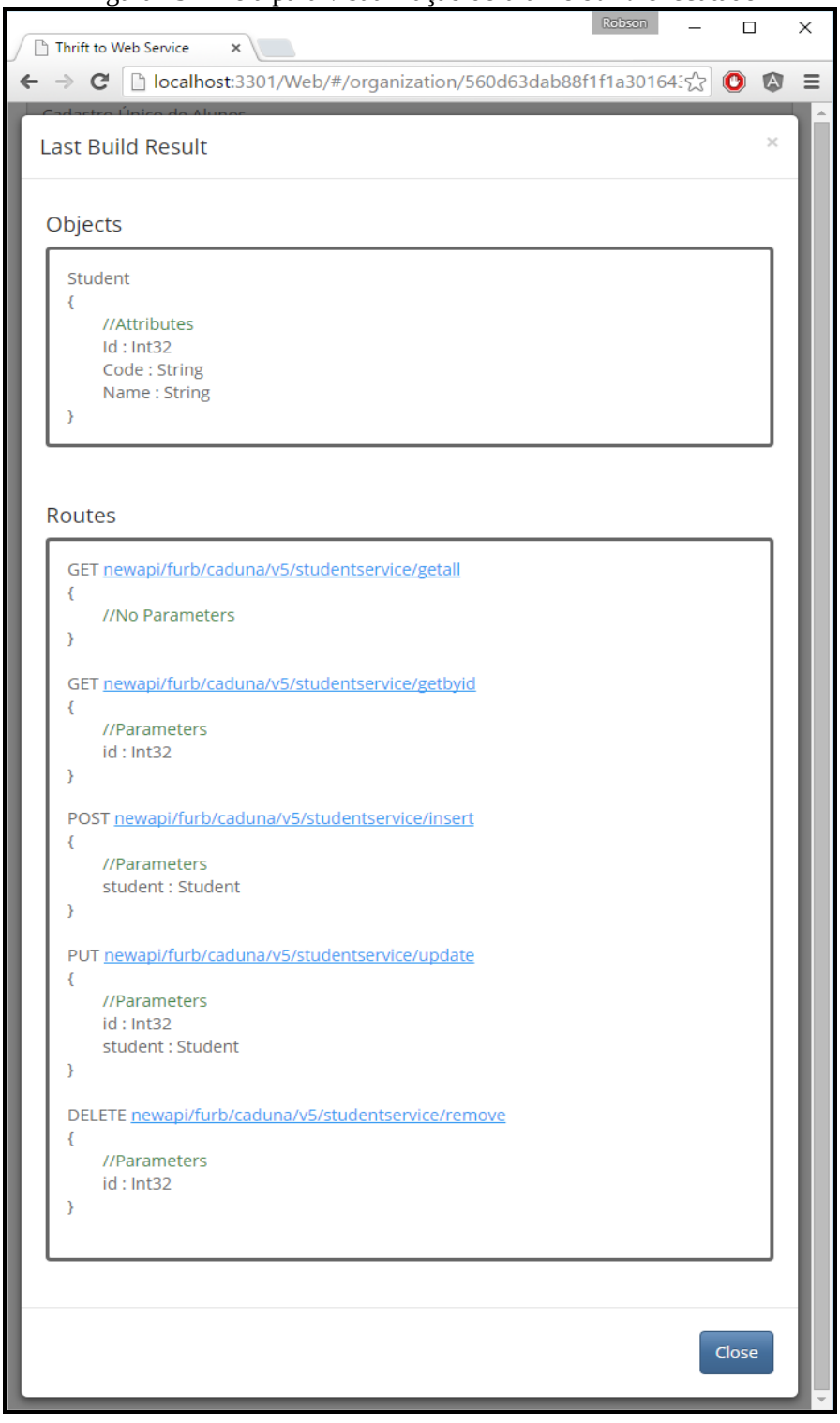
A Figura 18 apresenta a tela onde o usuário pode efetuar a confirmação de um *build* que foi executado para o serviço. São listados os objetos e as rotas que serão criados no *middleware* para possibilitar a chamada das funções do servidor *thrift* através de requisições HTTP. Em cada rota é disponibilizada uma seleção de métodos HTTP para que o usuário determine qual o tipo de requisição ela deve aceitar. O botão `Confirm` efetua a confirmação do *build* e a criação das estruturas, enquanto o botão `Close` cancela a solicitação.

Figura 18 – Tela para confirmação do *build*



A Figura 19 apresenta a tela para visualização do resultado do último *build* executado para o serviço. A tela é semelhante à de confirmação de *build*, porém não permite nenhum tipo de edição das informações. Para cada rota criada dentro do *middleware* são apresentados o endereço, o tipo de requisição HTTP e parâmetros que devem ser passados para que seja possível realizar a execução dos métodos do servidor *thrift*.

Figura 19 – Tela para visualização do último *build* executado



3.4 RESULTADOS E DISCUSSÕES

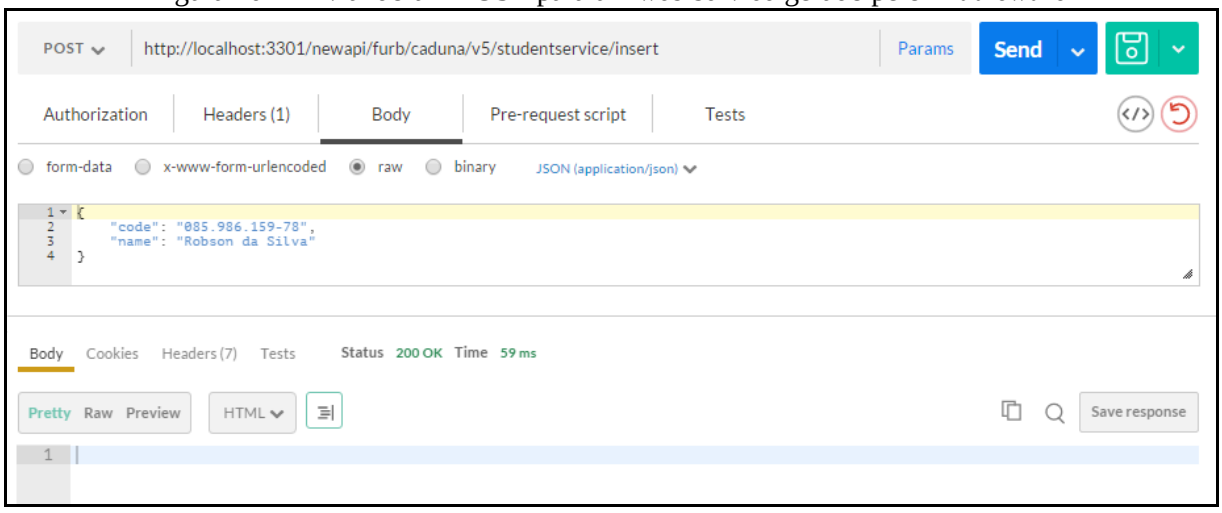
Esta seção é dedicada a apresentar os resultados obtidos com o desenvolvimento do *middleware* através da demonstração da sua usabilidade. Além disso são apresentadas as principais dificuldades encontradas ao longo do seu desenvolvimento, juntamente com as suas respectivas soluções.

Com base no serviço cujo cadastro foi apresentado na Figura 17 e no resultado do seu *build*, que foi apresentado na Figura 19, foram realizados alguns testes para comprovar a usabilidade do *middleware*. Para este teste foi desenvolvido um servidor *thrift* com base na IDL cadastrada no serviço, que gerencia os dados em memória e roda no endereço *localhost* e porta 9090, também configurados no serviço.

O teste consiste na realização de requisições HTTP para as rotas geradas pelo processo de *build*, que são apresentadas na Figura 19 juntamente com seus respectivos tipos e parâmetros. As requisições foram feitas através da ferramenta Postman (Google), que permite a configuração da requisição, inclusão dos parâmetros e visualização dos resultados.

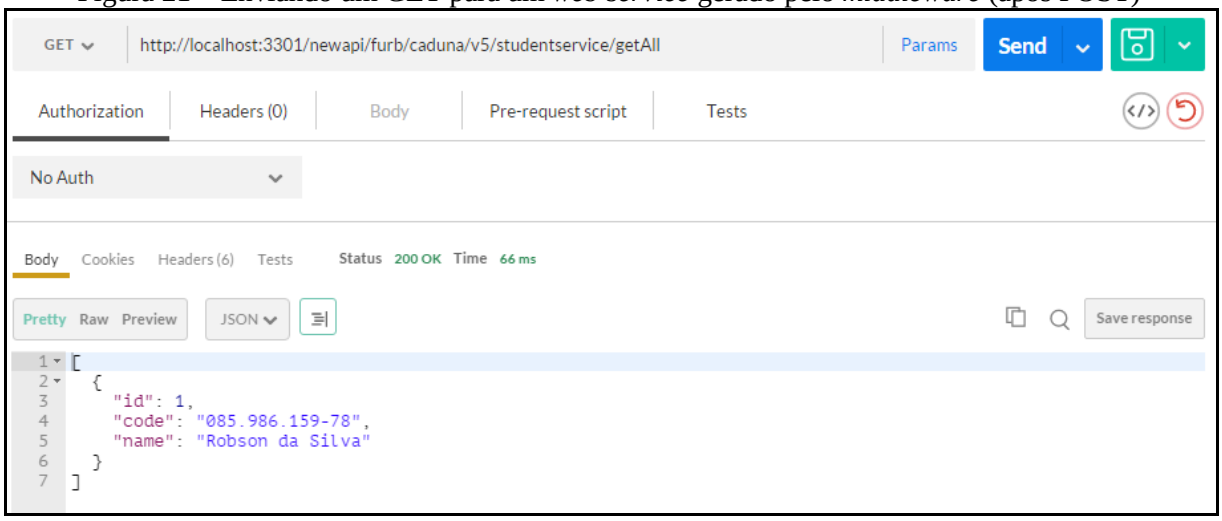
A primeira requisição, apresentada na Figura 20, consiste em um POST para o endereço HTTP do método `insert`. Para esta requisição é necessário o envio de um parâmetro, com as propriedades do objeto `Student`, no formato JSON. No servidor *thrift*, a execução deste método insere um novo item em uma lista armazenada em memória.

Figura 20 – Enviando um POST para um *web service* gerado pelo *middleware*



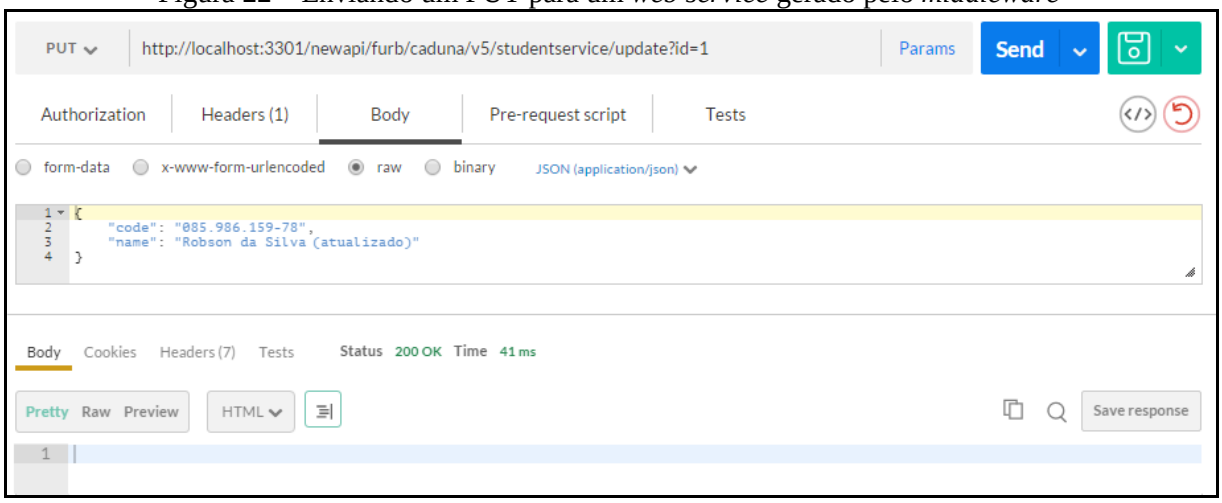
A segunda requisição consiste em um GET para o endereço HTTP do método `getAll`. Esta requisição retorna todos os objetos `Student` armazenados em memória no servidor *thrift*. O retorno é recebido em formato JSON, conforme apresentado na Figura 21.

Figura 21 – Enviando um GET para um *web service* gerado pelo *middleware* (após POST)



A terceira requisição consiste em um PUT para o endereço HTTP do método `update`, que é apresentada na Figura 22. Esta requisição solicita dois parâmetros para execução: o identificador do objeto e o objeto com as alterações necessárias. O identificador é informado na URL, através do parâmetro `id`, enquanto o objeto `Student` com as propriedades alteradas é enviado através de um JSON. No servidor *thrift*, este método busca o objeto na lista através do identificador e, em seguida, atualiza suas propriedades.

Figura 22 – Enviando um PUT para um *web service* gerado pelo *middleware*

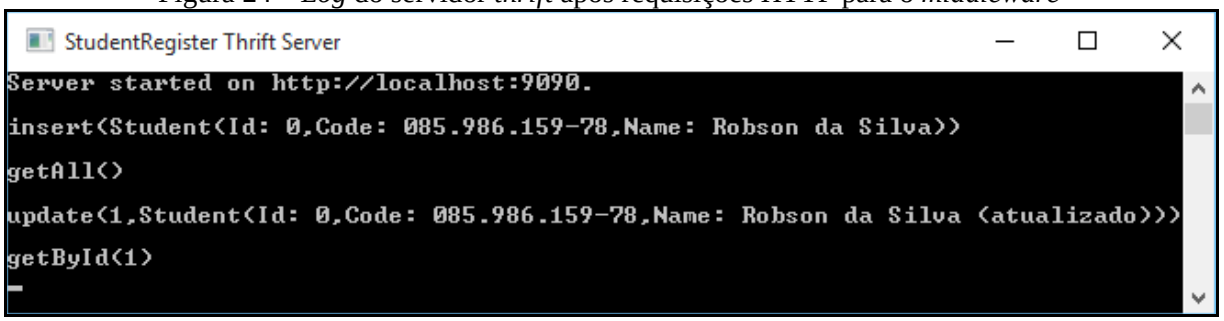


A quarta e última requisição efetuada consiste em um GET para o endereço HTTP do método `getById`. Para esta requisição deve ser informado um parâmetro contendo o identificador do objeto que deve ser consultado. O servidor *thrift* retorna o objeto `Student` referente ao identificador, que no cliente HTTP é recebido no formato JSON, conforme apresentado na Figura 23.

Figura 23 – Enviando um GET para um *web service* gerado pelo *middleware* (após PUT)

Todas as requisições efetuadas no fluxo de testes são recebidas pelo *middleware* e ele, através das estruturas geradas no processo de *build*, efetua as chamadas RPC para execução dos métodos no servidor *thrift*. A Figura 24 apresenta o *log* do servidor *thrift* que atendeu as chamadas.

Figura 24 – *Log* do servidor *thrift* após requisições HTTP para o *middleware*



O custo de comunicação com o servidor através do *middleware* implementado é, logicamente, maior do que em uma troca direta de mensagens entre um servidor e um cliente *thrift*. Isto se deve ao fato de que é inserida uma camada de acesso adicional na comunicação, resultando em um percurso maior entre o servidor e o cliente final. Este já era um resultado esperado para o trabalho proposto, mas para alguns serviços, a utilização do *middleware* é válida. Em muitos casos, a isenção de que cada cliente de um servidor *thrift* necessite conhecer e implementar a tecnologia pode ser mais interessante do que o breve ganho de performance que se tem com a comunicação direta.

Ao longo do desenvolvimento do trabalho algumas dificuldades foram apresentadas, sendo a primeira delas a interpretação dos servidores *thrift*. Era necessária uma maneira de reconhecer as funcionalidades disponibilizadas por cada servidor cadastrado no *middleware*. Uma das maneiras de resolver esta dificuldade seria interpretar a IDL do servidor para entender suas funcionalidades. Esta solução implicaria na necessidade de se construir um interpretador.

A possibilidade de se implementar um interpretador da IDL do *thrift* foi desconsiderada, pois além do custo adicional de desenvolvimento, a documentação para entendimento da IDL também não era abrangente com todas as suas funcionalidades. Considerando que a ferramenta de geração de código do *thrift* em algum momento já realiza a interpretação da IDL para sua execução, ela foi escolhida como caminho para entendimento das funcionalidades do servidor.

Dessa forma, surgiu a ideia de implementar a primeira biblioteca auxiliar, a `ThriftHelper`, responsável por utilizar as ferramentas do próprio *thrift* para geração de código. O *middleware* foi desenvolvido em C# e esta linguagem possibilita a utilização da técnica de *reflection*, que permite que em tempo de execução o programa consiga obter informações sobre as estruturas do código compilado. Assim, utilizando a `ThriftHelper` para gerar código C# e, em seguida, compilando este código e interpretando o resultado através de *reflection*, foi resolvido o problema de como obter as funcionalidades dos servidores *thrift*.

Outra dificuldade apresentada ao longo do desenvolvimento foi a de gerenciamento dos *web services* gerados pelo processo de *build*. A geração do código C# para estes *web services* e a compilação deles em *assemblies* foram facilmente alcançadas através dos recursos oferecidos pelo .NET e pela segunda biblioteca auxiliar desenvolvida, a `WebApiForThrift`. Porém, o carregamento e atualização destes *assemblies* dentro do *middleware*, em tempo de execução, sem interferir no restante das funcionalidades dela, trouxe alguns problemas.

O .NET permite o fácil carregamento de *assemblies* externos em uma aplicação que já está em execução. Assim que carregado, as funcionalidades contidas no *assembly* passam a ficar disponíveis para que a aplicação as utilize. Porém, após carregado, não existe suporte do *framework* para que um *assembly* seja completamente removido da aplicação sem que ela seja reiniciada.

Este era um problema para o *middleware* implementado, pois os servidores *thrift* podem alterar sua estrutura e, com isso, os *web services* também devem ser atualizados para refletir as alterações. Apenas carregar o novo *assembly* no *middleware* não foi uma solução funcional devido a conflitos com o *assembly* que já estava carregado para aquele servidor. Além disso, todas as técnicas testadas para remover o *assembly* antigo não foram bem-sucedidas, pois causavam erros no *middleware*, interrompendo sua execução.

Por fim, a solução encontrada foi realizar o versionamento dos *builds* efetuados para cada serviço. Cada *assembly* gerado é carregado no *middleware* sendo identificado como único através da sua versão. Esta solução, além de resolver o problema do carregamento de

assemblies atualizados, adicionou uma funcionalidade interessante para o *middleware*, que é a de conter a versão atualizada dos serviços e, ainda assim, guardar referência para as versões dos *assemblies* antigos.

A última dificuldade a ser comentada nesta seção é a da alteração da tecnologia utilizada para persistência dos dados. Inicialmente, o *middleware* foi desenvolvido utilizando um banco de dados MySQL. Em um determinado momento, este formato de armazenamento passou a ser ineficaz, uma vez que as informações consistiam em arquivos e objetos com conteúdo extenso e cuja manutenção das relações com outras entidades não era um item crítico. Neste cenário, a camada de persistência de dados foi migrada para um banco de dados MongoDB, onde o *middleware* ganhou performance pelo fato de dispensar as regras do modelo entidade-relacionamento implementadas por bancos relacionais.

Após a avaliação dos resultados obtidos com a implementação do *middleware*, foi realizada uma breve comparação com as funcionalidades dos trabalhos correlatos. O Quadro 16 apresenta as funcionalidades comparadas.

Quadro 16 – Comparação entre os trabalhos correlatos e o *middleware* desenvolvido

Funcionalidades	**Middleware desenvolvido**	**BeeSUS**	**Remote TestKit**	**Evernote**
Utiliza o *thrift* como base para o processo de comunicação com o servidor	Sim	Sim	Sim	Sim
Possui *web services* como uma alternativa para isenção do *thrift* no cliente	Sim	Sim	Não	Não
Oferece bibliotecas específicas como uma alternativa para isenção do *thrift* no cliente	Não	Não	Sim	Sim
A camada de *web services* disponibilizada utiliza uma camada *thrift* encapsulada para comunicação com o serviço	Sim	Sim	Não	Não
A camada de *web services* é gerada dinamicamente por um processo baseado na estrutura do servidor	Sim	Não	Não	Não

Apesar da finalidade do *middleware* desenvolvido não ser a mesma dos trabalhos correlatos, algumas semelhanças podem ser observadas nas funcionalidades. O *thrift* é um elemento em comum entre todas as implementações, sendo ele o responsável pela comunicação com os respectivos servidores. As outras comparações se referem às alternativas utilizadas por cada serviço para facilitar, ou até mesmo isentar, o uso do *thrift* por parte dos clientes. A grande vantagem do *middleware* que foi desenvolvido é a capacidade de gerar os *web services* para qualquer servidor *thrift* que for necessário, enquanto as alternativas oferecidas pelos outros serviços são específicas para seus próprios servidores.

4 CONCLUSÕES

O *middleware* desenvolvido neste trabalho atingiu os objetivos propostos, pois ela possibilita que funcionalidades de servidores *thrift* possam ser executadas utilizando apenas requisições HTTP. Isto permite que organizações disponibilizem servidores *thrift* para integração com seus clientes sem que haja a necessidade de cada um deles estudar e implementar tal tecnologia.

Conforme apresentado nos resultados, a utilização do *middleware* para realizar a comunicação com os servidores *thrift* tem um custo maior se comparado com o fluxo direto entre um cliente e um servidor *thrift*. Este é um item importante de ser apontado nos resultados da implementação, apesar de não ter grande influência na proposta do *middleware*.

O desenvolvimento das bibliotecas auxiliares para gerenciar algumas das funcionalidades do *thrift* e do ASP.NET Web API também foi bem-sucedido. Elas foram utilizadas pelo próprio *middleware* para desacoplar algumas responsabilidades da implementação principal e, além disso, podem ser utilizadas em outros projetos relacionados ao *thrift*.

A escolha das tecnologias utilizadas para implementação deste trabalho também foi acertada. O .NET, juntamente com a linguagem C#, atendeu perfeitamente todas as necessidades encontradas para o funcionamento ideal do *middleware*, necessidades como a compilação do código gerado em tempo de execução e até o carregamento dos *assemblies* compilados sem a necessidade de qualquer interrupção na aplicação. O MongoDB também atendeu as necessidades relacionadas à persistência de dados, apesar de este não ser um fator determinante para a regra de negócios do *middleware*.

O *thrift* demonstrou-se, nos estudos e implementações realizados, um *framework* bastante interessante para a integração de sistemas. Esta é uma afirmação baseada no suporte oferecido pela ferramenta de geração de código para diversas linguagens, juntamente com os protocolos e modelos de transporte implementados por ele. Apesar do *middleware* implementado neste trabalho funcionar como uma abstração para utilização do *thrift* nos clientes, ela não invalida de forma alguma a utilização direta dele. A ideia é simplesmente oferecer uma opção adicional de comunicação com servidores *thrift*, onde o cliente pode utilizar uma tecnologia mais comum no seu dia-a-dia, no caso, o *web service*.

4.1 EXTENSÕES

Algumas extensões possíveis para este trabalho são:

a) realizar a análise de desempenho, comparando a comunicação padrão do *thrift* com a utilização do middleware desenvolvido em diversos cenários;
b) implementar a geração de *web services* SOAP como uma alternativa ao modelo REST utilizado no *middleware*;
c) aprimorar o processo de *build* para que ele reconheça a estrutura do servidor *thrift* apenas interpretando a IDL, sem ser necessária a geração e compilação dos códigos;
d) estudar a possibilidade de realizar a conversão das requisições de *web service* para *thrift* utilizando somente *reflection*, dispensando o processo de *build*.

REFERÊNCIAS

APACHE SOFTWARE FOUNDATION. **Apache Thrift.** 2014. Disponível em: <http://thrift.apache.org/about>. Acesso em: 25 maio 2015.

BARRY, Douglas K. **Service Architecture:** Representational State Transfer (REST). 2007. Disponível em: <http://www.service-architecture.com/articles/web-services/representational_state_transfer_rest.html>. Acesso em: 07 set. 2015.

BARRY, Douglas K. **Service Architecture:** SOAP. 2003. Disponível em: <http://www.service-architecture.com/articles/web-services/soap.html>. Acesso em: 07 set. 2015.

BARRY, Douglas K. **Service Architecture:** Web Services Explained. 2013. Disponível em: <http://www.service-architecture.com/articles/web-services/web_services_explained.html>. Acesso em: 07 set. 2015.

BEEIT. **BeeSUS:** Integre o sistema próprio ao E-SUS. 2015. Disponível em: <http://www.beeit.com.br/site/integracaoesus.php>. Acesso em: 20 mar. 2015.

CHAVES, Leonardo Grandinetti. **O EAI como abordagem de integração de sistemas corporativos para a obtenção de vantagem competitiva.** 2004. Disponível em: <http://www.linhadecodigo.com.br/artigo/429/o-eai-como-abordagem-de-integracao-de-sistemas-corporativos-para-a-obtencao-de-vantagem-competitiva.aspx>. Acesso em: 07 set. 2015.

DIMOPOULOS, Stratos. **Thrift protocol stack.** 2013. Disponível em: <http://thrift-tutorial.readthedocs.org/en/latest/thrift-stack.html>. Acesso em: 20 mar. 2015.

EVERNOTE. **Sobre o Evernote.** 2008. Disponível em: <https://evernote.com/intl/pt-br/corp/>. Acesso em: 03 out. 2015.

GOKHALE, Aniruddha; KUMAR, Bharat; SAHUGUET, Arnaud. **Reinventing the Wheel?:** CORBA vs. Web Services. 2002. Disponível em: <http://www2002.org/CDROM/alternate/395/>. Acesso em: 17 maio 2015.

HUNSAKER, Claire. **REST vs SOAP:** When Is REST Better?. 2015. Disponível em: <https://stormpath.com/blog/rest-vs-soap/>. Acesso em: 07 set. 2015.

JONES, M. Tim. **Simplificando o desenvolvimento de software em nuvem escalável com Apache Thrift.** 2014. Disponível em: <http://www.ibm.com/developerworks/br/library/os-cloud-apache-thrift/>. Acesso em: 20 mar. 2015.

JSON.ORG. **Introducing JSON.** 2003. Disponível em: <http://json.org/>. Acesso em: 07 set. 2015.

LIMA, Jean Carlos Rosário. **Web Services (SOAP X REST).** 2012. 41 f. TCC (Graduação) - Curso de Tecnólogo em Processamento de Dados, Faculdade de Tecnologia de São Paulo, São Paulo, 2012. Disponível em: <http://www.fatecsp.br/dti/tcc/tcc00056.pdf>. Acesso em: 23 maio 2015.

MICROSOFT. **How RPC Works.** 2004. Disponível em: <https://technet.microsoft.com/pt-br/library/Cc738291(v=WS.10).aspx>. Acesso em: 07 set. 2015.

MINISTÉRIO DA SAÚDE. **Sistema e-SUS Atenção Básica:** Manual de Exportação - API Thrift. 2015. Disponível em: <http://dab.saude.gov.br/portaldab/esus/manual_exportacao_1.3/docs/manualExportacao_e-SUSABv1_3.pdf>. Acesso em: 23 maio 2015.

MORO, Tharcis dal; DORNELES, Carina F.; REBONATTO, Marcelo Trindade. Web services WS-* versus Web Services REST. **Reic - Revista de Iniciação Científica,** Passo Fundo, v. 11, n. 1, p.36-51, 2011.

NTT RESONANT. **About Remote TestKit Thrift API.** 2014. Disponível em: <https://appkitbox.com/en/testkit/doc/remote-testkit-thrift-api>. Acesso em: 20 mar. 2015.

ROUSE, Margaret. **Remote Procedure Call (RPC) definition.** 2009. Disponível em: <http://searchsoa.techtarget.com/definition/Remote-Procedure-Call>. Acesso em: 20 mar. 2015.

SILVA, Leonardo Goncalves da. **Integrando Apache Camel e Apache Thrift.** 2013. Disponível em: <http://www.devmedia.com.br/integrando-apache-camel-e-apache-thrift/26994#ixzz3VynDftC9>. Acesso em: 20 mar. 2015.

SLEE, Mark; AGARWAL, Aditya; KWIATKOWSKI, Marc. **Thrift:** Scalable Cross-Language Services Implementation. 2007. Disponível em: <http://thrift.apache.org/static/files/thrift-20070401.pdf>. Acesso em: 20 mar. 2015.

SORDI, José Osvaldo de; MARINHO, Bernadete de Lourdes. Integração entre Sistemas: Análise das Abordagens Praticadas pelas Corporações Brasileiras. **Revista Brasileira de Gestão de Negócios,** São Paulo, v. 9, n. 23, p.78-93, jan. 2007.

APÊNDICE A – Arquivos gerados pelo thrift

Neste apêndice constam alguns trechos de código obtidos através da ferramenta de geração de código do *thrift*. Os códigos são apresentados nos quadros Quadro 17 e Quadro 18 e foram gerados para a linguagem C#.

Quadro 17 – Trecho de código C# gerado para um *service* do *thirft*

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Text;
5  using System.IO;
6  using Thrift;
7  using Thrift.Collections;
8  using System.Runtime.Serialization;
9  using Thrift.Protocol;
10 using Thrift.Transport;
11
12 namespace StudentRegister
13 {
14   public partial class StudentService {
15     public interface Iface {
16       List<Student> getAll();
17       Student getById(int id);
18       void insert(Student student);
19       void update(int id, Student student);
20       void remove(int id);
21     }
22
23     public class Client : IDisposable, Iface {
24       public Client(TProtocol prot) : this(prot, prot) { }
25
26       public Client(TProtocol iprot, TProtocol oprot)
27       {
28         iprot_ = iprot;
29         oprot_ = oprot;
30       }
31
32       protected TProtocol iprot_;
33       protected TProtocol oprot_;
34       protected int seqid_;
35
36       public TProtocol InputProtocol { get { return iprot_; } }
37       public TProtocol OutputProtocol { get { return oprot_; } }
38
39       #region " IDisposable Support "
40       private bool _IsDisposed;
41
42       public void Dispose()
43       {
44         Dispose(true);
45       }
46
```

```
47        protected virtual void Dispose(bool disposing)
48        {
49          if (!_IsDisposed)
50          {
51            if (disposing)
52            {
53              if (iprot_ != null)
54              {
55                ((IDisposable)iprot_).Dispose();
56              }
57              if (oprot_ != null)
58              {
59                ((IDisposable)oprot_).Dispose();
60              }
61            }
62          }
63          _IsDisposed = true;
64        }
65        #endregion
66
67        public List<Student> getAll()
68        {
69          send_getAll();
70          return recv_getAll();
71        }
72
73        public void send_getAll()
74        {
75          oprot_.WriteMessageBegin(new TMessage("getAll", TMessageType.Call, seqid_));
76          getAll_args args = new getAll_args();
77          args.Write(oprot_);
78          oprot_.WriteMessageEnd();
79          oprot_.Transport.Flush();
80        }
81
82        public List<Student> recv_getAll()
83        {
84          TMessage msg = iprot_.ReadMessageBegin();
85          if (msg.Type == TMessageType.Exception) {
86            TApplicationException x = TApplicationException.Read(iprot_);
87            iprot_.ReadMessageEnd();
88            throw x;
89          }
90          getAll_result result = new getAll_result();
91          result.Read(iprot_);
92          iprot_.ReadMessageEnd();
93          if (result.__isset.success) {
94            return result.Success;
95          }
96          throw new TApplicationException(
97            TApplicationException.ExceptionType.MissingResult, "getAll failed: unknown result");
98        }
99      }
100   }
101 }
```

Quadro 18 – Trecho de código C# gerado para uma *struct* do *thirft*

```
1   using System;
2   using System.Collections;
3   using System.Collections.Generic;
4   using System.Text;
5   using System.IO;
6   using Thrift;
7   using Thrift.Collections;
8   using System.Runtime.Serialization;
9   using Thrift.Protocol;
10  using Thrift.Transport;
11
12  namespace StudentRegister
13  {
14    [Serializable]
15    public partial class Student : TBase
16    {
17      private int _id;
18      private string _code;
19      private string _name;
20
21      public int Id {
22        get { return _id; }
23        set { __isset.id = true; this._id = value; }}
24
25      public string Code {
26        get { return _code; }
27        set { __isset.code = true; this._code = value; }}
28
29      public string Name {
30        get { return _name; }
31        set { __isset.name = true; this._name = value; }}
32
33      public Isset __isset;
34
35      public struct Isset {
36        public bool id;
37        public bool code;
38        public bool name;
39      }
40
41      public Student() { }
42
43      public void Read (TProtocol iprot)
44      {
45        TField field;
46        iprot.ReadStructBegin();
47        while (true)
48        {
49          field = iprot.ReadFieldBegin();
50          if (field.Type == TType.Stop) {
51            break;
52          }
53          switch (field.ID)
54          {
55            case 1:
56              if (field.Type == TType.I32) {
57                Id = iprot.ReadI32();
58              } else {
59                TProtocolUtil.Skip(iprot, field.Type);
```

```
60            }
61            break;
62          case 2:
63            if (field.Type == TType.String) {
64              Code = iprot.ReadString();
65            } else {
66              TProtocolUtil.Skip(iprot, field.Type);
67            }
68            break;
69          case 3:
70            if (field.Type == TType.String) {
71              Name = iprot.ReadString();
72            } else {
73              TProtocolUtil.Skip(iprot, field.Type);
74            }
75            break;
76          default:
77            TProtocolUtil.Skip(iprot, field.Type);
78            break;
79        }
80        iprot.ReadFieldEnd();
81      }
82      iprot.ReadStructEnd();
83    }
84
85    public void Write(TProtocol oprot) {
86      TStruct struc = new TStruct("Student");
87      oprot.WriteStructBegin(struc);
88      TField field = new TField();
89      if (__isset.id) {
90        field.Name = "id";
91        field.Type = TType.I32;
92        field.ID = 1;
93        oprot.WriteFieldBegin(field);
94        oprot.WriteI32(Id);
95        oprot.WriteFieldEnd();
96      }
97      if (Code != null && __isset.code) {
98        field.Name = "code";
99        field.Type = TType.String;
100       field.ID = 2;
101       oprot.WriteFieldBegin(field);
102       oprot.WriteString(Code);
103       oprot.WriteFieldEnd();
104     }
105     if (Name != null && __isset.name) {
106       field.Name = "name";
107       field.Type = TType.String;
108       field.ID = 3;
109       oprot.WriteFieldBegin(field);
110       oprot.WriteString(Name);
111       oprot.WriteFieldEnd();
112     }
113     oprot.WriteFieldStop();
114     oprot.WriteStructEnd();
115   }
116 }
117 }
```