INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

ÁREA DEPARTAMENTAL DE ENGENHARIA DE ELECTRÓNICA E TELECOMUNICAÇÕES E DE COMPUTADORES

ANSWER - Plataforma em C++ para o desenvolvimento de serviços web

---

RODRIGO MIGUEL CRUZ FERNANDES

(Licenciado em Engenharia Informática e de Computadores)

Mestrado em Engenharia Informática
e de Computadores e Projecto em Engenharia Informática

Orientador:

Prof. Pedro FÉLIX

Júri:

Presidente:

Prof. Coordenador Walter VIEIRA

Vogais:

Prof. Pedro PEREIRA

Prof. Pedro FÉLIX

Setembro, 2012

Resumo

Analisando as tendências actuais de mercado, observa-se a necessidade da capacidade de integração de sistemas recorrendo à utilização de serviços web. Actualmente o C++ é ainda uma das mais populares linguagens de programação, facto justificado pelo abundante portefólio de aplicações, desde alto-nível a sistemas embebidos, com inúmeras bibliotecas que tiram partido dos diversos paradigmas de programação que a linguagem suporta.

Porém, e ao contrário de outras linguagens, como C# ou Java que oferecem suporte à criação de serviços web integrado com a plataforma, existe uma lacuna no suporte ao desenvolvimento de serviços web em C++.

Para dar resposta à lacuna existente é criada a plataforma *ANSWER*, focada em C++, com ênfase num modelo de programação simples, permitindo o rápido desenvolvimento de serviços web RPC/SOAP e REST.

Palavras chave: C++; web services; REST; SOAP; WSDL, SOA;

Abstract

Looking at the current market trend, industry standards for interoperability driven by customer demand are pushing for system integration via web-services. Today C++ is still one of the most popular programming languages, reason which stems from an abundant portfolio of applications, from high level to embedded systems, with innumerable libraries making full use of all the different paradigms it supports.

However, and unlike some languages/platforms, like C# or Java that offer support for web services from the start, there is a real lack of a standards compliant modern solution for C++.

The *ANSWER* platform is thus created, allowing the C++ development of RPC/SOAP and REST webservices, with a programing model focused on development speed, ease of use, performance and standards compliance.

Keywords: C++; web services; REST; SOAP; WSDL, SOA;

Agradecimentos

Ao Prof. Pedro Félix, pelo seu ensino inspirador e constante orientação.

Aos meus colegas da *AnubisNetworks* pelo apoio constante, nas várias iterações da plataforma. Em especial ao Luís Bilo, Paulo Pacheco, Rogério Rocha e Henrique Aparício.

Aos meus pais, pelo imenso esforço e dedicação ao longo de todos estes anos.

Dedico este trabalho à minha esposa, Grazieli,
por todo o seu apoio, amor e dedicação.

Conteúdo

1 Introdução **1**

1.1 Contexto . . . 1

1.2 Objectivos . . . 2

1.3 Organização do documento . . . 3

2 Estado da Arte - Estilos e tecnologias para serviços web **5**

2.1 Perspectiva histórica dos estilos e paradigmas de serviços . . . 5

2.1.1 Unix RPC . . . 5

2.1.2 CORBA . . . 6

2.1.3 DCOM . . . 7

2.1.4 SOAP . . . 8

2.1.5 REST . . . 8

2.2 Tecnologias C++ . . . 8

2.2.1 Axis2/C . . . 8

2.2.2 WSO2/C++ . . . 11

2.2.3 gSOAP . . . 12

2.2.4 PocoProject - Remote . . . 14

2.2.5 Staff . . . 15

2.2.6 AnubisNetworks Webservices Alpha . . . 16

2.3 Outras linguagens e plataformas . . . 20

2.3.1 OpenRasta . . . 20

2.3.2 Jax-RS . . . 21

2.4 Notas finais . . . 22

3 Arquitectura **25**

3.1 Processo de construção . . . 26

3.1.1 Módulo CMake CPP_WSDL . . . 26

3.1.2 Modelo interno do Doxygen . . . 27

3.1.3 XSLTs de geração . . . 27

3.2 Adaptadores e sua configuração 31
3.2.1 Adaptador Axis2/C 32
3.2.2 Adaptador FastCGI 32
3.2.3 Adaptador módulo nativo de Apache 33
3.2.4 Módulos *ANSWER* 34

4 Modelo de Programação 35
4.1 Modelo de Programação RPC/SOAP 35
4.1.1 Definição de Tipos de Mensagens 36
4.1.2 Definição de Serviço 37
4.1.3 Comparação com outros modelos de programação 38
4.2 Modelo de Programação REST sobre HTTP 38
4.2.1 Definição de Recurso 38
4.2.2 Definição de Controlador 39
4.3 Controlo da estratégia instanciação 40
4.4 Formatação 41
4.5 Modelo dos programação de módulos 43
4.6 Excepções e tratamento 45

5 Aspectos de implementação 47
5.1 Utilização de Macros 47
5.2 Registo automático 48
5.3 Integração com Boost::Serialization 49
5.4 Meta programação baseada em *templates* 50
5.4.1 SFINAE 53
5.5 Detecção de tipos complexos 54
5.6 Considerações de *Threading* 56

6 Conclusões e Notas finais 57

Listagens de modelos de programação 59
.1 Exemplo do modelo de programação do Axis2/C 59
.2 Modelo de programação do gSOAP 64

Ficheiros de processo de construção e de configurações 67
.3 Macro de CMakel CPP_WSDL 67
.4 Configuração do build de serviço em PocoProject::Remote 68

Lista de Figuras

2.1 Cronologia dos eventos que antecedem as actuais plataformas de serviços web . . . 6
2.2 Arquitectura RPC . . . 6
2.3 Utilização do modelo interno do Axis2/C . . . 17
2.4 Vista geral da arquitectura do modelo base . . . 18

3.1 Arquitectura Geral da *ANSWER* . . . 25
3.2 Arquitectura do doxygen (adaptado de [dox12b]) . . . 28
3.3 Exemplo de documentação da uma operação . . . 30
3.4 Diagrama de correspondência de adaptadores e servidores HTTP . . . 31

4.1 Ciclo de vida de um pedido . . . 44

Listagens

2.1 Exemplo do conteúdo de um ficheiro services.xml 10
2.2 Conteúdo de uma mensagem de pedido à operação *greet* 10
2.3 Conteúdo da mensagem de resposta da operação *greet* 10
2.4 Definição da função `greet` 12
2.5 Definição de tipos e funções RPC para parsing pelo gerador de código 13
2.6 Código de exemplo PocoProject::Remote 14
2.7 Modelo de programação do *Staff* 15
2.8 Definição da classe de pedido 18
2.9 Definição da classe de resposta e função de seriação manual 19
2.10 Definição da classe de serviço e registo da operação 19
2.11 Classe que representa o recurso Customer 20
2.12 Código de registo da classe recurso e configuração do *endpoint* 21
2.13 Código de registo da classe recurso e configuração do *endpoint* 21
3.1 Utilização da macro CMake CPP_WSDL 27
3.2 Função XSLT `dataElementIsComplex` 28
3.3 Configuração do Axis2/C 32
3.4 Configuração do adaptador FastCGI em Apache 33
3.5 Configuração do adaptador módulo nativo de Apache 33
3.6 Exemplificação do uso de symlinks para ordenação do carregamento de módulos 34
4.1 Definição da classe de pedido 36
4.2 Definição da classe de resposta 37
4.3 Definição da classe de serviço e registo da operação 37
4.4 Definição da classe de serviço e registo da operação 38
4.5 Definição da classe Recurso VCard 38
4.6 Declaração do controlador VCardController 39
4.7 Registo do controlador 40
4.8 Alteração do tipo de instanciação 41
4.9 Exemplo da seriação de um VCard em XML 42

4.10 Definição de um Codec para VCard 42
4.11 Definição de um Decoder para VCard 43
4.12 Declaração de um módulo de fluxo 43
4.13 Definição de um módulo de fluxo 44
4.14 Classe de tratamento de excepção 45
4.15 Classe de tratamento de excepção 45
5.1 Definição da macro `REGISTER_OPERATION` 47
5.2 Programa exemplo de *scope* de instâncias globais 48
5.3 Classe de registo automático `RegisterWebModule` 48
5.4 Utilização da classe `RegisterWebModule` 49
5.5 Código de registo de operação 50
5.6 Classe operation 52
5.7 Classe operation 53
5.8 Exemplo de SFINAE 53
5.9 Utilização de SFINAE para determinar estratégia de instânciação . . 54
5.10 Elementos simples e seu mapeamento em tipos xml 54
5.11 Tipos de colecções reconhecidas 55
1 Código de exemplo Axis2/C (retirado de [axi12d]) 59
2 Código de exemplo para a criação de um serviço em gSOAP (adaptado de [gso12b]) 64
3 Macro de CMake CPP_WSDL 67
4 Ficheiro de configuração para gerador PocoProject::Remote 68
5 Configuração do Doxygen 69

Lista de Tabelas

2.1 Características e principais problemas das plataformas de serviços web C++ 23

3.1 Capacidades dos adaptadores 31

Glossário

AST Abstract Syntax Tree

CORBA Common Object Request Broker Architecture

CRTP Curiously recurring template pattern

DSL Domain Specific Language

DTO Data transfer object

POCO Plain Old CLR Object

POD Plain Old Data

REST Representational State Transfer

RMI Remote method invocation

RPC Remote Procedure Call

SFINAE Substitution failure is not an error

SOAP Simple Object Access Protocol

SOA Service Oriented Architectures

UUID Universally unique identifier

WSDL Web Services Description Language

XML eXtensible Markup Language

STL Standard Template Library (C++)

XSLT eXtensible Stylesheet Language

Capítulo 1

Introdução

No presente relatório tratam-se aspectos de desenho, implementação e optimização de uma plataforma para a criação de serviços web em C++. Este trabalho foi parcialmente desenvolvido no âmbito profissional da empresa *AnubisNetworks*, tendo como objectivo aliar investigação académica à melhoria de produtos concretos no mercado empresarial.

1.1 Contexto

A capacidade de desenvolver serviços web é actualmente um requisito importante, facilitando a tarefa de integração de sistemas e promovendo a interoperabilidade e modularidade de um projecto.

A necessidade crescente do desenvolvimento de serviços web depara-se com uma grave lacuna: A inexistência de uma plataforma adequada para a realização de serviços web em C++. As plataformas disponíveis possuem vários problemas, tais como modelos de programação desadequados, fluxos de trabalho complicados, utilização de meta-linguagens em comentários, entre outros. Em muitos casos esses problemas são resultado directo da plataforma ter sido desenvolvida para C, não tirando partido de qualquer característica do C++. Sendo necessário a criação de serviços web, uma solução possível é evitar por completo recorrer a estas plataformas, optando por integrar o código C++ existente com serviços web realizados em outra linguagem. Porém, esta aproximação resulta invariavelmente no aumento da complexidade do sistema, uma vez que é necessário suportar as várias linguagens utilizadas.

Para dar resposta à lacuna existente é criada uma nova plataforma, focada em C++, com ênfase num modelo de programação simples, denominada por *ANSWER*, capitalização de AnubisNetworkS WebsERvices.

Da multitude de estilos, mecanismos e protocolos disponíveis actualmente para a criação de serviços web, os dois principais são o SOAP (*Simple Object Access Protocol*) e o REST (*Representational State Transfer*). Ambas as aproximações possuem os seus méritos e desvantagens. Cabe ao programador a decisão de qual a melhor a utilizar, em função do problema a resolver. Seguidamente descreve-se sucintamente o SOAP e o REST, no âmbito do projecto.

O SOAP como o próprio nome indica é um protocolo. A especificação do protocolo SOAP, presente no documento [GHM+06] define a estrutura das mensagens XML trocadas entre os participantes a chamadas remotas. Uma das principais características é a extensibilidade, através da qual estão definidas especificações adicionais para segurança, *routing*, etc. Destaca-se também a sua neutralidade face ao protocolo de transporte. Desta forma as mensagens são enviadas através de protocolos como HTTP, SMTP ou até TCP. Também não são impostas restrições quanto ao estilo arquitectural, uma vez que se enquadra de forma igual a um estilo RPC ou SOA.

Embora usualmente apontado como a alternativa/oposto do SOAP, o REST descreve não um protocolo mas um estilo arquitectural. Este estilo resulta da análise das melhores características do desenho da World-Wide-Web, proposto por [Fie00]. Central a este estilo é a noção de recurso, elemento de informação, ao qual pode ser associado um identificador único (URI). Clientes e servidores comunicam mediante uma interface padronizada (ex.: HTTP) trocando representações do estado destes recursos.

1.2 Objectivos

Uma vez que a base para o projecto foi criada em contexto profissional, os objectivos aqui expressos reflectem o desejo de melhoria sobre uma plataforma em constante desenvolvimento.

É princípio fundamental de desenho a simplicidade da utilização da plataforma *ANSWER*. Desta forma o modelo de programação deverá ser o mais simples e sucinto possível por forma a facilitar a adopção por parte do programador, enquanto utilizador da plataforma.

A plataforma deve suportar a criação de serviços web nos variantes RPC, SOAP e REST, definindo o modelo de programação em C++, evitando a utilização de meta-comentários, ou ferramentas de geração de ficheiros intermédios (*stubs*).

Os processo de construção e subsequente publicação de serviços web devem ser

integrados, reduzindo ou eliminado por completo a utilização de processos intermédios que necessitem intervenção manual por parte do utilizador.

A plataforma deve permitir a disponibilização dos serviços em vários servidores web.Por exemplo, devem ser suportados o Apache Webserver, disponível em [apa12], o NGinx, disponível em [ngi12], o Lighty, disponível em [lig12], etc.

Deverá ser desenvolvida com atenção à portabilidade, segurança e desempenho.

1.3 Organização do documento

Para além do presente capítulo de introdução compõem o documento outros cinco capítulos, cujo objectivo é descrito em seguida:

No capítulo **Estado da Arte - Estilos e tecnologias para serviços web** são contextualizadas as principais tecnologias que resultaram nas tecnologias de serviços actuais. Após isso, são analisadas as actuais soluções para o desenvolvimento de serviços web. É dado ênfase às plataformas C++, onde se descrevem os principais problemas destas, e às plataformas de outras linguagens, onde se destacam ideias e modelos para a criação de uma solução sem essas lacunas.

Segue-se o capítulo **Arquitectura** onde se analisa a arquitectura da *ANSWER*, por forma a compreender os vários constituintes da plataforma e as suas interacções. É descrito o processo de construção onde é observado o processo de publicação do código de serviços, bem como aspectos de configuração dos componentes da plataforma.

Em **Modelo de Programação** é descrito o modelo de programação proposto pela *ANSWER*. São detalhadas as funcionalidades disponibilizadas pela plataforma para a criação de Serviços RPC/SOAP e REST, entre outros componentes associados ao desenvolvimento.

No capítulo **Aspectos de implementação** são descritas as técnicas que suportam a simplificação do modelo de programação proposto, e explicados os detalhes de integração com componentes e bibliotecas externos.

Finalmente apresenta-se o capítulo **Conclusões e Notas finais**, onde são referidos os pontos de desenvolvimento futuro desta plataforma, e é feito um balanço geral do resultado do trabalho de projecto.

Em apêndice encontram-se os ficheiros de configuração utilizados pela plataforma, assim como um exemplo completo da criação e publicação de um serviço web.

Capítulo 2

Estado da Arte - Estilos e tecnologias para serviços web

O presente capítulo tem por objectivo analisar as várias tecnologias e estilos para o desenvolvimento de serviços web.

Na primeira secção estabelece-se uma perspectiva histórica do desenvolvimento dos principais estilos que culminaram nas actuais plataformas.

Na segunda secção descrevem-se as presentes tecnologias para o desenvolvimento de serviços web em C++, detalhando as lacunas existentes nas actuais soluções. A terceira secção foca tecnologias disponíveis para a criação de serviços web em outras linguagens que não C++, identificando pontos de interesse passiveis de utilização na implementação da *ANSWER*.

Por fim, na última secção, realiza-se um resumo dos principais pontos do capítulo.

2.1 Perspectiva histórica dos estilos e paradigmas de serviços

Para compreender melhor o âmbito dos diferentes paradigmas e estilos é útil olhar à evolução tecnológica que os precedeu. Na figura 2.1 apresenta-se a evolução das tecnologias que resultaram nas actuais plataformas de serviços web.

2.1.1 Unix RPC

RPC é o termo que engloba mecanismos e funções cujo espaço de execução do procedimento difere do processo que realiza a chamada a este. Pode ser entendido como uma simples função remota, onde os processos cliente e servidor transmitem a informação necessária através de uma rede, como se ilustra em 2.2. Uma vez

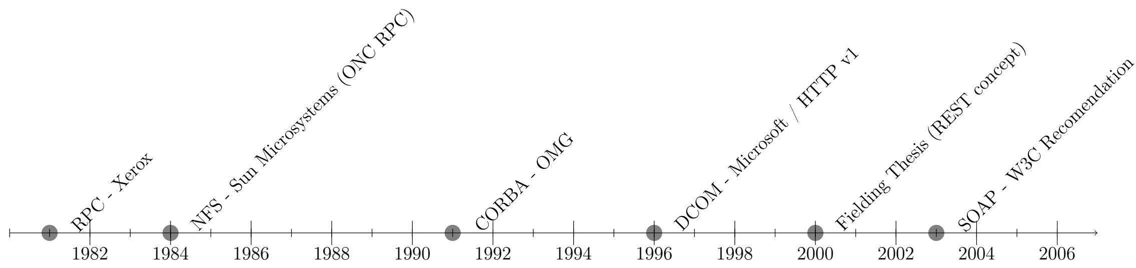

Figura 2.1: Cronologia dos eventos que antecedem as actuais plataformas de serviços web

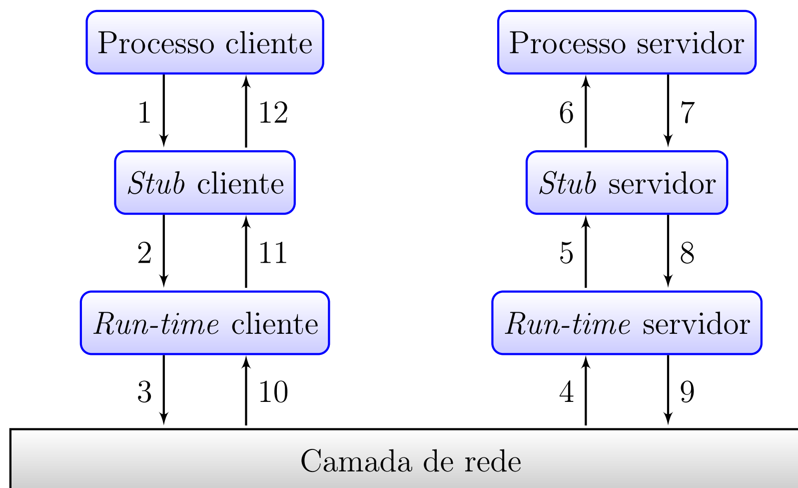

Figura 2.2: Arquitectura RPC

que se trata apenas de um conceito, não formalizado, existem muitas subtilezas nas múltiplas implementações, o que resulta numa incompatibilidade ao nível do protocolo/plataforma ou funcionamento.

Nos anos 80, os protocolos de comunicação eram focados na camada de rede, como é o caso do *Network File System* (NFS) desenvolvido inicialmente pela Sun Microsystems.

Com a chegada dos anos 90 é dado foco à programação orientada a objectos. Esse foco teve também impacto no RPC, resultando na criação da noção de *Object Remote Procedure Call* (ORPC). Os ORPC acrescentam ao RPC mapeamento entre objectos da aplicação aos protocolos de rede.

2.1.2 CORBA

A *Common Object Request Broker Architecture* (CORBA) é uma especificação, criada pelo *Object Management Group* (OMG), cuja primeira versão foi publicada 1991 [Gro91]. Tem como objectivo a interacção de componentes de software em

diferentes linguagens e ambientes de execução, mediante e utilização de uma linguagem intermédia, o IDL. Estes componentes recorrem a um intermediário designado por *Object request broker* (ORB). Ao invocar uma operação o ORB é responsável por encontrar a implementação do objecto, invocando esta de forma transparente que seja local ou remoto.

A comunicação é feita com base na definição da *Interface Definition Language* (IDL), que realiza a interface com os objectos externos da linguagem de programação dos componentes. A especificação CORBA define interfaces para o mapeamento de IDL para Ada, C, C++, LISP, Ruby, Smalltalk, Java, COBOL, PL/I e Python.

Hoje em dia são poucas as aplicações que utilizam CORBA. Dos vários motivos que justificam o insucesso do CORBA cita-se o principal, do documento [Hen06]:

- A especificação era larga e complexa, e muita dela nunca chegou a ser implementada, nem como *proof-of-concept*.

2.1.3 DCOM

O *Distributed Component Object Model* (DCOM) é uma tecnologia proprietária da Microsoft. Esta tecnologia estende o âmbito da tecnologia de componentes binários *Component Object Model* (COM), acrescentando a esta uma camada ORPC responsável pela realização do *Marshaling* e *Garbage Collection* dos dados. Desta forma, clientes interagem com objectos presentes no servidor através da obtenção de um ponteiro para uma das interfaces desse objecto, podendo então chamar métodos sobre esse ponteiro como se o objecto residisse no mesmo espaço de memória do processo do cliente.

World Wide Web

Com a ubiquidade do acesso à Internet, as especificações e protocolos que a compõem ganham grande relevância. Estes protocolos com especificações abertas, obtêm uma grande adopção, em particular o HTTP. Acedido na forma tradicional, o HTTP demonstrou capacidades que resultaram no conceito de arquitectura REST. Por outro lado a necessidade empresarial de convenções formalizadas obteve no HTTP um protocolo simples, que em conjunto com a popularidade do formato XML, utilizado para descrever a linguagem de representação de dados, permitiram o protocolo SOAP.

2.1.4 SOAP

O SOAP como o próprio nome indica é um protocolo. A especificação do protocolo SOAP, presente no documento [GHM+06] define a estrutura das mensagens XML trocadas entre os participantes a chamadas remotas. Uma das principais características é a extensibilidade, através da qual estão definidas especificações adicionais para segurança, *routing*, etc. Destaca-se também a sua neutralidade face ao protocolo de transporte. Desta forma as mensagens são enviadas através de protocolos como HTTP, SMTP ou até TCP. Também não são impostas restrições quanto ao estilo arquitectural, uma vez que se enquadra de forma igual a um estilo RPC ou SOA.

2.1.5 REST

Embora usualmente apontado como a alternativa/oposto do SOAP, o REST descreve não um protocolo mas um estilo arquitectural. Este estilo resulta da análise das melhores características do desenho da World-Wide-Web, proposto por [Fie00]. Central a este estilo é a noção de recurso, elemento de informação, ao qual pode ser associado um identificador único (URI). Clientes e servidores comunicam mediante uma interface padronizada (ex.: HTTP) trocando representações do estado destes recursos, como referido em [RR07].

2.2 Tecnologias C++

A presente secção tem por objectivo a caracterização das plataformas existentes presentemente para o desenvolvimento de serviços web em C++, referindo estilos arquitecturais e protocolos descritos no capítulo de introdução.

2.2.1 Axis2/C

O Axis2/C, documentado em [axi12e], é a principal tecnologia utilizada no desenvolvimento de serviços web em SOAP na linguagem C++. Implementa uma parte significativa do conjunto WS-* e é distribuído sob uma licença livre (*Apache Licence v2*). Não obstante, a complexidade do seu modelo de trabalho dificulta a integração em projectos de desenvolvimento. Este modelo, que define os passos necessários à criação de um serviço web é descrito de seguida:

- Declarar classes em Java, que representam os *endpoints* de serviço, operações e tipos de pedido e resposta.

- Executar um conjunto de programas que processam o ficheiros com o código Java, gerando em C (não C++), e o ficheiro WSDL respectivo.
- Da declaração de cada serviço, resultam 3 ficheiros (*stubs*) onde é possível implementar o código da operação.
- Depois implementada a função para realizar a operação, o projecto é compilado como uma biblioteca dinâmica, para ser utilizado pelo módulo de Apache do Axis2.

Para além do modelo de trabalho, existem também sérios problemas ao nível do próprio código do projecto, entre os quais o grande número de fugas de memória ainda presentes [axi12h], [axi12j], [axi12k], [axi12i], bem como a falta de suporte a IPv6 [axi12g].

Detalhes da arquitectura

Da extensa descrição da arquitectura do Axis2, presente em [axi12c], descrevem-se de seguida os detalhes mais relevantes ao âmbito deste trabalho de projecto.

O módulo Apache de Axis2/C dita a seguinte sequência para o carregamento de serviços:

As operações e serviços são descritos num ficheiro XML denominado services.xml, que é utilizado pelo módulo apache do Axis2/C para a determinação do caminho em ficheiro de sistema onde a biblioteca dinâmica com o código do serviço web se encontra. Por forma a exemplificar alguns dos pontos citados apresenta-se de seguida o modelo de funcionamento:

1. Ao iniciar são iteradas todas as sub-directorias de um directório lido da configuração.
2. Por cada uma das directorias, é lido o ficheiro services.xml (descrito de seguida).
3. A biblioteca é carregada mediante a função *dl_ open* e é invocado o serviço mediante uma chamada *dl_ sym* com o símbolo correcto.

Para compreender não só o Axis2/C mas as várias soluções presentes de seguida é útil caracterizar o ficheiro services.xml. Como se descreve na listagem 2.1, no caso das operações associadas ao método GET, o nome e ordem dos parâmetros é definida através do atributo `RESTLocation`. Este atributo possui o *template* de URI a utilizar quando a invocação à operação é realizada por GET, ao invés de SOAP. Compreende este esquema o que o *Axis2/C* denomina, ainda que incorrectamente, de *RESTful Gateway*, uma vez que são mapeados verbos HTTP em endpoints parametrizáveis.

Listagem 2.1: Exemplo do conteúdo de um ficheiro services.xml

```
<?xml version="1.0"?>
<service name="calculator">
  <parameter name="ServiceClass">calculator</parameter>
  <description>Calculator Service</description>
  <operation name="multiplication" uri_ns="http://calculator" ↩
      uri_prefix="calculator">
    <parameter name="wsamapping">urn:multiplication</parameter>
    <parameter name="RESTMethod">POST</parameter>
    <parameter name="RESTLocation">multiplication/</parameter>
  </operation>
  <operation name="add" uri_ns="http://calculator" uri_prefix="↩
      calculator">
    <parameter name="wsamapping">urn:add</parameter>
    <parameter name="RESTMethod">GET</parameter>
    <parameter name="RESTLocation">soma/{operand1}/{operator}/{↩
      operand2}/</parameter>
  </operation>
</service>
```

Modelo de Programação

Para mostrar o modelo de programação recorre-se ao exemplo da criação de um serviço com uma operação *greet*, com capacidade de receber pedidos como os da listagem 2.2, retornando a resposta listada em 2.3. Os passos e código apresentados foram retirados de [axi12f].

Listagem 2.2: Conteúdo de uma mensagem de pedido à operação *greet*

```
<greet>
Hello Service!
<greet>
```

Listagem 2.3: Conteúdo da mensagem de resposta da operação *greet*

```
<greetResponse>
Hello Client!
<greetResponse>
```

Para implementar o serviço com Axis2/C é necessário:

- Implementar a função correspondente à operação de *greet*.
- Essa função terá o nome `axis2_hello_greet`.
- Implementar as funções *init*, *invoke*, *on_fault* e *free*, definidas pela interface[1]

[1] *interface* neste contexto é uma estrutura C cujos atributos membro são ponteiros de funções

axis2_svc_skeleton.

- Essas funções serão implementadas com os nomes hello_init, hello_invoke, hello_on_fault e hello_free, respectivamente.
- Implementar a função de criação, responsável por criar uma instância do esqueleto de serviço, contendo ponteiros para as funções citadas anteriormente e finalmente retornado-a.
- Implementar as funções axis2_get_instance e axis2_remove_instance, invocadas pelo Axis2/C para realizar a criação e destruição de instâncias de serviço.
- Escrever (manualmente) o ficheiro services.xml para o serviço.

O código resultante do exemplo é listado no anexo 1.

Concluí-se desta forma que a utilização de Axis2/C é bastante complexa.

O modelo de trabalho tipicamente associado a projectos mais complexos envolve programação em Java, o que é contra-intuitivo no desenvolvimento de serviços em C++.

Também o seu modelo de programação é complicado implicando a criação de um conjunto considerável de funções que não a da operação em si.

2.2.2 WSO2/C++

O projecto WSO2/C++ [wso12] tem como base como base o Axis2/C. Tem como objectivo produzir um *wrapper* C++ para o código C gerado pelo Axis2/C. Está aparentemente inactivo, e o código disponibilizado não é compatível com as versões mais recentes das ferramentas de que depende.

Modelo de Programação

Como é possível observar na listagem 2.4, embora o código seja C++, o modelo de programação apresentado não utiliza um nível de abstracção adequado. A utilização de ponteiros sem qualquer semântica de *ownership* torna o código desnecessariamente complicado, como explicado por [Red11]. A utilização da biblioteca xml Axiom como mecanismo de acesso a pedidos e criação de respostas é um processo fastidioso, já que impõe a utilização de funções C, não tirando proveito de vantagens da linguagem C++ como o encapsulamento em classes ou semântica de construtores e destrutores para inicialização e limpeza de recursos.

Listagem 2.4: Definição da função greet

```
OMElement* Hello::greet(OMElement* inMsg)
{
  string greet;
  if(inMsg)
  {
    try
    {
      if(inMsg->getFirstChild() && inMsg->getFirstChild()->nodeType()↩
          == AXIOM_TEXT)
      {
        OMText *text = dynamic_cast<OMText*>(inMsg->getFirstChild());
        greet = text->getValue();
        cout << "Client greeted Saying" << greet << endl;
      }
    }catch (bad_cast)
    {
      return NULL;
    }
    OMElement *helloEle = new OMElement("greetResponse");
    OMElement *text = new OMElement("text");
    helloEle->setText(greet);
    return helloEle;
  }
  return NULL;
}
```

2.2.3 gSOAP

A gSOAP, disponível em [VEG02], é uma das mais antigas plataformas para o desenvolvimento de serviços SOAP em C/C++. Desenvolvida e testada desde 2001, o funcionamento básico recorre à utilização de um programa *wsdl2h* que gera um ficheiro header *.h* a partir de um WSDL.

São providenciados com a plataforma dois programas *wsdl2h* e *soapcpp2*. O primeiro importa um ou mais WSDLs e schemas XML e gera um header C (compatível com C++), de gSOAP com a sintaxe necessária para definir o serviço web e operações e os tipos de dados C/C++. O segundo programa utiliza este *header*, gerando criando os ficheiros *soapH.h* e *soapC.cpp* que definem a seriação XML para os tipos de dados, bem como os ficheiros *soapClient.cpp* e *soapServer.cpp*, *stubs* para a implementação do código de cliente e servidor.

Da compilação de todos os ficheiros necessários resulta um executável, a ser

utilizado num servidor web com suporte a CGI.

Como exemplo concreto, recorre-se à proposta descrita em [gso12b], que consiste na criação de um serviço SOAP que retorna o *quadrado mágico*([Mat12]) de dimensão N, definindo a operação *magicSquare* implementada pela função C `int ns1__magic(int n, matrix *square);`.

Na listagem 2.5 encontra-se a declaração dos tipos necessários para representar as respostas e da função RPC que implementa a operação.

Listagem 2.5: Definição de tipos e funções RPC para parsing pelo gerador de código

```
class vector
{ public:
  int *__ptr;
  int __size;
  struct soap *soap;
  vector();
  vector(int n);
  ~vector();
  void resize(int n);
  int& operator[](int i);
};
class matrix
{ public:
  vector *__ptr;
  int __size;
  struct soap *soap;
  matrix();
  matrix(int n, int m);
  ~matrix();
  void resize(int n, int m);
  vector& operator[](int i);
};
int ns1__magic(int rank, matrix *result);
```

Na plataforma gSOAP o código de seriação e de-seriação é gerado em conjunto com o código esqueleto do serviço. Da execução do programa *soapcpp2* sobre a listagem 2.5 resultam ficheiros de elevadas dimensões, como se pode verificar em [gso12a]. Entre os ficheiros gerados destacam-se quatro ficheiros, necessários ao processo de compilação do serviço. Estes quatro ficheiros, que não incluem o código da implementação do serviço, contam para este exemplo com o total de 2782 linhas de código (incluíndo comentários), tendo o maior deles 2154 linhas. Ao invés de detalhar aqui todos os ficheiros, opta-se por listar em o código da função `ns1__magic` que implementa a operação descrita anteriormente.

Modelo de Programação

Na listagem 2 apresenta-se o código que implementa a operação de cálculo do quadrado mágico, `ns1__magic`, bem como a função `main`, onde se inicia o servidor SOAP. Denota-se que a função que implementa a operação tem realizar retorno de estado à plataforma, o que implica que o parâmetro de resposta tem ser passado como input, modelo que não é tão claro como a correspondência ....

2.2.4 PocoProject - Remote

Produzido pela *Applied Informatics Software Engineering* a Remote, disponível em [Inf12], é uma biblioteca para desenvolvimento de WebServices em C++, sendo uma componente opcional, de licença comercial às bibliotecas Poco, de licença Open-Source que a empresa disponibiliza.

As bibliotecas Poco incluem funcionalidades de baixo nível, como threads, eventos, filesystem, timers e de alto nível como Email, processamento XML e compressão Zip, entre muitas outras.

Detalhes da Arquitectura

A criação de serviços web SOAP na plataforma Remote faz uso de um programa de parsing, utilizando um ficheiro de configuração em XML, um exemplo do qual se mostra em anexo, na listagem 4. Da execução deste programa sobre a definição do serviço e operações são gerados vários ficheiros, que tem que ser incluídos na definição das operações do serviço, criando um executável que implementa um servidor HTTP para responder ao serviço.

Modelo de Programação

Como se exemplifica na listagem 2.6 a componente para geração de serviços Remote propõe no modelo de programação a utilização de *keywords* específicas, tais como `@serialize` e `@remote` em comentários. Estas serão depois utilizadas por um programa de *parsing*, cujos artefactos resultantes são utilizados no processo de compilação. Possui também dependências fortes com restantes componentes da biblioteca PocoProject.

Listagem 2.6: Código de exemplo PocoProject::Remote

```
//@ serialize
struct WeatherData
{
  float windSpeed;
```

```
  short windDirection;
  float windSpeed;
  Poco::Timestamp timestamp;
};

class WeatherStation
{
public:
  //@ remote
  WeatherData currentWeather() const;

  //@ remote
  void pastMeasurements(
    std::vector<WeatherData>& measuments,
    std::size_t maxResults) const;
};
```

2.2.5 Staff

O *Staff*, disponível em [sta12], apresenta-se como o mais recente projecto para o desenvolvimento de serviços SOAP em C++. É bastante recente, motivo pelo qual não era conhecido como uma das alternativas no início deste trabalho de projecto. O surgimento do projecto *Staff* sublinha a importância da plataforma *ANSWER*, demonstrando que existe uma lacuna na criação de serviços web em C++.

Internamente utiliza o *Axis2/C*, pelo que está limitado ao mesmo conjunto de problemas, ainda que disponibilizando uma modelo de C++. Para além da biblioteca o *Staff* é composto pelo programa de gerador / parser, denominado *staff_codegen*, central ao desenvolvimento, responsável pela geração de stubs, parsing de WSDLs, etc.

Modelo de programação

Na listagem 2.7 apresenta-se um exemplo proposto pela documentação do projecto, onde a classe `User` define um objecto "composto"[2].

Listagem 2.7: Modelo de programação do *Staff*

```
namespace sample
{
  //! user
  struct User
  {
```

[2]Também chamado de "complexo", por oposição a objectos simples.

```
    int id; //!< user id
    std::string name; //!< user name
    std::string description; //!< user description
  };

  typedef std::list<User> UsersList; //!< users list

  //! user manager service
  class UserManager: public staff::IService
  {
  public:
    virtual int Add(const std::string& name, const std::string& ↩
        description) = 0;
    virtual void Remove(int id) = 0;
    virtual void Update(const User& user) = 0;
    virtual User Get(int id) const = 0;
    virtual UsersList GetAllUsers() const = 0;
  };
} // namespace sample
```

Denota-se a imposição de herança, na qual a classe de serviço implementa de `IService` e os meta comentários.

2.2.6 AnubisNetworks Webservices Alpha

A empresa *AnubisNetworks* possuía vários serviços web, disponibilizados para facilitar a integração com outros sistemas e produtos. Dado que de todas as opções para desenvolver serviços web em C++ descritas em 2.2 a mais completa era a Framework *Axis2/C* esta foi escolhida para os implementar.

Revendo os passos propostos para definir serviços pela plataforma em 2.2.1, deverá ser claro que estes são manualmente intensivos, o que potência típicos erros de *copy-paste*. A necessidade de editar manualmente os ficheiros em caso de alterações aos parâmetros das operações e encapsulamento (de-seriação) e construção da resposta (seriação) era um trabalho fastidioso. Para dar uma resposta, ainda no contexto profissional da *AnubisNetworks* foi iniciado o esforço de desenvolvimento de uma plataforma que com o objectivo de encapsular o Axis2/C.

Nos seguinte pontos será apresentada a descrição do funcionamento deste modelo, as escolhas realizadas e problemas encontrados durante a sua concepção.

Arquitectura

No contexto profissional da *AnubisNetworks* os serviços web *SOAP compliant*, estavam já a ser desenvolvidos com recurso ao Axis2/C. Sendo este *open-source*, o código analisado para estudar a viabilidade de criar um ponto de partida que evitasse recriar toda a infraestrutura de raiz. Desta forma o código foi alterado, adicionando-se um nível de indirecção ao ponto de onde é realizada a invocação do código de serviço e é criada a resposta a partir dessa chamada. Tal foi possível porque ao invocar o código responsável pela execução da operação desejada, este acede aos parâmetros disponibilizados numa árvore XML, independentemente do tipo de pedido, como se demonstra na figura 2.3.

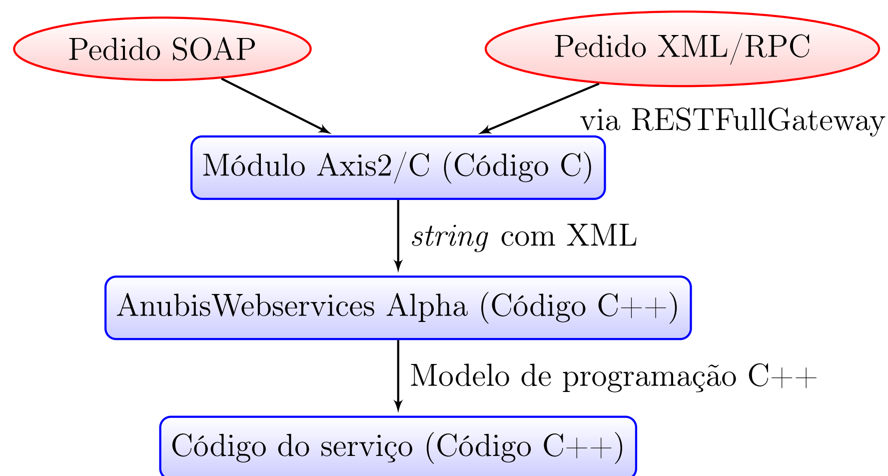

Figura 2.3: Utilização do modelo interno do Axis2/C

Dado isto, foi criada uma indirecção para realizar a chamada de código C++, criando uma fronteira do encapsulamento do código original do Axis2/C.

Após um modelo conceptual ter sido apresentado na empresa, interesse foi demonstrado em que este projecto pudesse vir a substituir a solução que recorria ao Axis2/C, potenciando o desenvolvimento de serviços em C++, de uma forma mais fácil ao modelo de trabalho proposto originalmente pelo Axis2/C.

Tal como é ilustrado na figura 2.4, o módulo Axis2 foi mantido, porque abstrai a integração com o Apache. Fornece também outras funcionalidades, entre as quais destaca-se a capacidade de definir uma interface que invoca as operações SOAP disponíveis mediante um *endpoint* predefinido, funcionalidade que é chamada de *RESTGateway*, incorrectamente já que se trate de um mecanismo de XML/RPC que não impõe qualquer restrição ao estilo arquitectural.

Dado que a plataforma *AnubisNetworks Webservices Alpha* não utiliza o conjunto de programas em Java referidos em 2.2.1, a funcionalidade de geração deste ficheiro

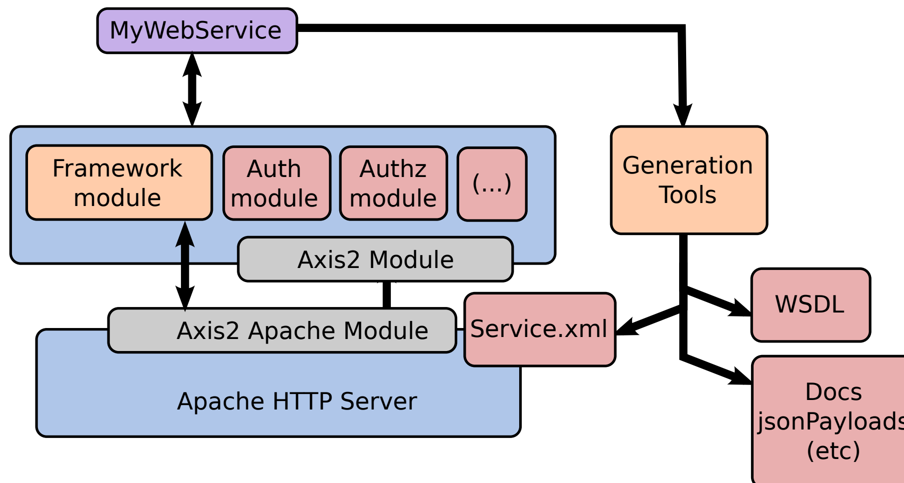

Figura 2.4: Vista geral da arquitectura do modelo base

teve de ser recriada, para integrar os serviços com o Axis2. Para esse fim, foi criado um programa, de nome *cpp_wsdl*, cuja função era processar os *headers* C++ nos quais os tipos dos serviços e operações estão declarados, gerando em função disso o service.xml. O funcionamento deste programa encontra-se descrito no diagrama de blocos da figura 2.4.

Antecipando dificuldade de criar um *parser* / *lexer* que compreenda o idioma C++, optou-se por utilizar uma biblioteca interna do IDE QTCreator, disponível em [qtc12b], que implementa um *parser*, cujo código se encontra em [qtc12a], com capacidade de interpretar a AST de C / C++ / ObjectiveC / ObjectiveC++, mediante o desenho de padrão Visitor.

Eventualmente o gerador foi expandido, para gerar não apenas o ficheiro services.xml mas também o ficheiro de WSDL do serviço, bem como outros necessários ao funcionamento da plataforma.

Modelo de programação

Na listagem 2.8 descreve-se um exemplo da criação de um serviço que implementa uma simples máquina de calcular.

O tipo correspondente ao DTO do pedido é concretizado da seguinte forma:

Listagem 2.8: Definição da classe de pedido

```
struct CalculatorRequest{
  double operand1;
  double operand2;
  char operation;
```

```
  CalculatorRequest(const anubis::webservice::Params& params):
    operand1(params.getParamByType<double>("operand1")),
    operand2(params.getParamByType<double>("operand2")),
    operation(params.getParamByType<char>("operation"))
  {}
};
```

A definição do construtor que recebe *params* é necessária para realizar a deseriação dos múltiplos parâmetros. A interface *Params* (classe abstracta) declara métodos para retornar valores de tipos ou de colecções de tipos. Internamente essa classe é concretizada por XMLParams realizando pesquisas XPATH, sobre o XML de pedido. Estas pesquisas são realizadas recorrendo ao selector "//" concatenado pelo nome do parâmetro, o que levanta problemas com definições de pedidos que possuam elementos com o mesmo nome em várias posições do XML.

A classe que define o tipo de resposta encontra-se listada em 2.9 sendo constituido pelo resultado da operação e uma *string* com a descrição textual da operação realizada ex.: $(4 + 5)$

Listagem 2.9: Definição da classe de resposta e função de seriação manual

```
struct CalculatorResponse{
  double result;
  std::string text;
};

std::ostream& operator<< (
    std::ostream& out,
    const CalculatorResponse &data ){
  out << "<operation>" << data.text << "</operation>";
  out << "<result>" << data.result << "</result>";
  return out;
}
```

Cabe à função global de inserção em `ostream` a seriação do tipo de resposta em XML. Este tipo de seriação é uma solução limitada já que não realiza a separação do conteúdo a seriar do formato em que esta vai ser representada, limitando o formato de respostas ao formato XML. A construção manual dos elementos é também contraproducente, já que pode levar a erros que apenas são detectáveis em tempo de execução.

Findada a criação das classes de pedido e resposta resta definir a classe que representa a operação, e explicitar o registo (configuração) desta na plataforma, tarefa realizada na listagem 2.10

Listagem 2.10: Definição da classe de serviço e registo da operação

```
class Calculator:
  public anubis::webservice::
          WebMethod<CalculatorRequest,CalculatorResponse>{

  public:
    void process(const CalculatorRequest &request,
                 CalculatorResponse &response);
};

anubis::webservice::
  RegisterWebMethod<Calculator> calculator("calculator");
```

Para a criação da operação é necessária a herança da classe *WebMethod* cujos templates serão parametrizados com o Pedido e Resposta (ou *void* caso não existisse). Pronto isso implementa-se a função virtual definida pela herança, onde o pedido é o primeiro parâmetro, e a resposta o segundo)

2.3 Outras linguagens e plataformas

Como principal fonte de inspiração para a criação da plataforma *ANSWER* foram analisadas plataformas de serviços web implementadas em outras linguagens, com especial foco em C# e JAVA. O modelo de *runtime* que suporta estas linguagens possui, nesta instância, vantagens que resultaram na criação de múltiplas soluções. Considerando a diferente natureza da linguagem de programação, pode não ser possível implementar serviços da mesma forma em C++. Ainda assim, o objectivo desta secção é identificar as boas práticas destas plataformas, referindo mecanismos ou processos a ter em conta no desenvolvimento da *ANSWER*.

2.3.1 OpenRasta

A plataforma para desenvolvimento de serviço web em C# OpenRasta define a noção de recurso, no sentido do estilo arquitectural REST como um tipo que possui um URI associado. Ao configurar *URIs* no OpenRasta, associam-se o tipo de recurso com o URI que lhe pertence.

Listagem 2.11: Classe que representa o recurso Customer

```
public class CustomersHandler
{
  public Customers Get()
  {
```

```
    return CustomerRepository.GetAll();
  }
}
```

Para implementar a obtenção de um recurso basta implementar o método público *Get()*. A plataforma não impõe nenhuma cadeia de herança sobre a classe que representa o recurso, tornando o modelo de programação particularmente simples.

Ao processar um pedido HTTP, o URI deste é associado a um controlador (*handler*) de recurso. No *OpenRasta* os controladores são classes POCO, que apenas necessitam de implementar um método com o nome do método HTTP do pedido, ou um método decorado com um atributo para esse nome de método HTTP.

Para gerar a resposta ao pedido o objecto retornado pelo método será processado por um *Codec*, classe que responsável por definir o formato de representação do estado do recurso associado a esse pedido.

Para registar o tipo a um URI com uma representação XML é suficiente utilizar código descrito em 2.12:

Listagem 2.12: Código de registo da classe recurso e configuração do *endpoint*

```
ResourceSpace.Has.ResourcesOfType<Customers>()
  .AtUri("/customers")
  .HandledBy<CustomersHandler>()
  .AsXmlDataContract();
```

É possível registar múltiplos *Codecs* sobre um mesmo URI, para processar o mesmo recurso em diferentes formatos, como XML, JSON, HTML, etc. Quando o pedido é realizado o Codec a ser utilizado é determinado em função do *header* Accept do HTTP explicitado pelo cliente, ou opcionalmente extensões de ficheiros, mecanismo que recorre às classes *UriDecorators*.

2.3.2 Jax-RS

A plataforma Jax-RS é utilizada para o desenvolvimento de web services em Java, recorrendo ao estilo arquitectural REST. Para esse efeito define várias anotações (@GET, @PUT, @POST) que mapeiam os vários métodos de acesso HTTP sobre uma classe que representa o tipo de um recurso web.

Um exemplo de utilização dessas anotações encontra-se listado em 2.13.

Listagem 2.13: Código de registo da classe recurso e configuração do *endpoint*

```
@Path("/request/{id}")
public class RequestResource {
 @GET
```

```
@Produces( { MediaType.APPLICATION_XML })
 public Request getRequestById(@PathParam("id") Long id) {
   RequestDAO requestDAO = new RequestDAO();
   Request request = requestDAO.getRequestById(id);
   return request;
 }
}
```

A contrário do que ocorre com *OpenRasta* a configuração não é centralizada. A associação entre os tipos que implementam o recurso e o URL onde este será disponibilizado ocorre na própria classe. De forma semelhante o tipo de resposta a um pedido é declarado no método que implementa o método de resposta.

2.4 Notas finais

Da análise das plataformas neste capítulo concluí-se a existência de uma lacuna na criação de serviços em C++. Tanto a mais completa, a Axis2/C, como a mais antiga, a gSOAP suportam C++, mas o modelo de programação usa código C. As plataformas que encapsulam a plataforma Axis2/C ou estão desactualizadas, no caso da WSO2/C++, ou recorrem a meta comentários, como a recente Staff. A utilização de geração intermédia de stubs é um mal generalizado de muitas das plataformas que complica o processo de integração.

Na tabela 2.1 são resumidas as principais características e problemas / lacunas das actuais solução de desenvolvimento de serviços web em C++.

Da análise das plataformas em outras linguagens destacam-se as principais considerações para o trabalho de projecto.

A noção de *Codec* é interessante e o seu mapeamento entre o header HTTP Accept é uma forma clara e simples de definir possíveis representações do mesmo recurso. Esta pode ser implementada no modelo REST e estendida no modelo RPC, por forma a permitir */RPC, ao invés de apenas XML/RPC.

É desejável que de forma semelhante ao *OpenRasta* a plataforma *ANSWER* não obrigue a uma cadeira de herança, por uma questão de simplificação do modelo de programação.

O modelo de publicação associado à criação de um serviço em *Jax-RS* é simples e poderoso, evitando os múltiplos passos normalmente associado a geradores/stubs e à necessidade de intervenção manual para compilação.

[1]Suporte é restrito à noção de *RESTFull* Gateway do Axis2/C

	Estilos suportados	Principais problemas
Axis2/C	SOAP e REST[1]	Fraca abstracção Fugas de memória Geração intermédia
WSO2/C++	O mesmo que Axis2/C	Geração intermédia Inactivo
gSOAP	SOAP	Fraca abstracção Geração intermédia
Poco::Remote	SOAP	Licença comercial Meta-comentários Geração intermédia
AnubisNetworks Webservices Alpha	O mesmo que Axis2/C	Modelo de programação pouco intuitivo
Staff	O mesmo que Axis2/C	Geração intermédia Meta-comentários
C2Serve	REST	Modelo insuficiente

Tabela 2.1: Características e principais problemas das plataformas de serviços web C++

Capítulo 3

Arquitectura

O presente capítulo descreve a arquitectura da plataforma *ANSWER*. Dado que nenhum dos modelos para C++ analisados no capítulo anterior são satisfatórios, apresentamos de seguida a solução proposta por este trabalho de projecto.

O diagrama de blocos apresentado em 3.1 mostra a arquitectura geral da solução, com os participantes na sequência de acções que resultam na publicação de um serviço web na plataforma *ANSWER*.

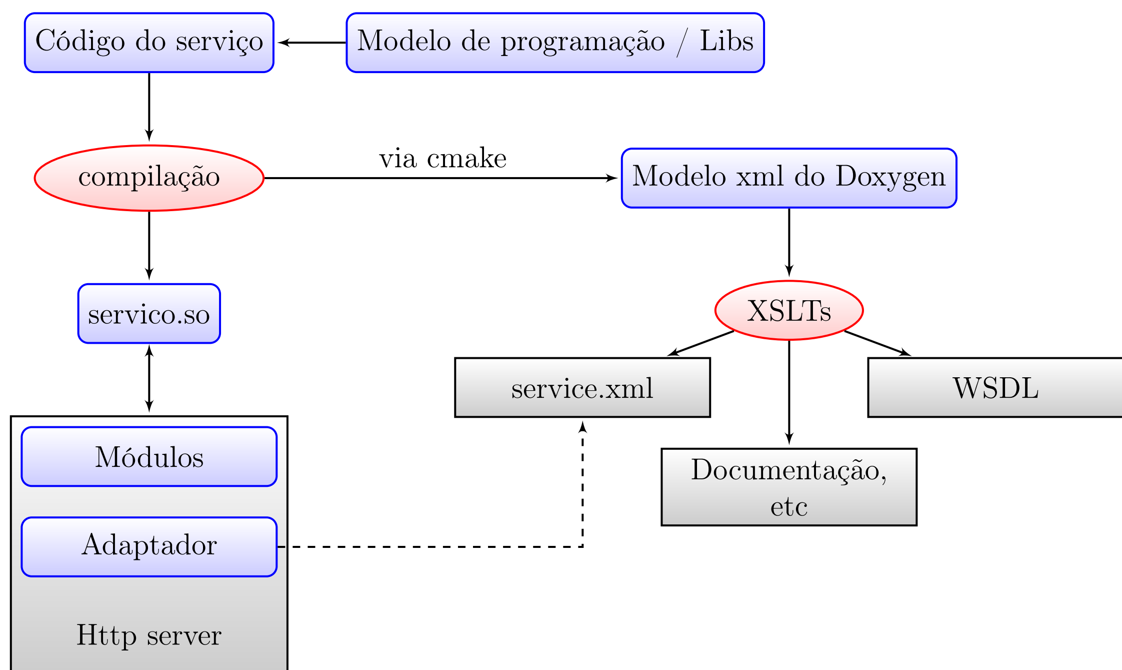

Figura 3.1: Arquitectura Geral da *ANSWER*

Assumindo que o código do serviço está completo e que também estão instalados os módulos que interceptam os vários pontos do ciclo de vida, (o modelo de programação de ambos é analisado em profundidade no capítulo seguinte), destacam-se as

seguintes fases:

- A compilação do código, passo responsável por gerar a biblioteca dinâmica com o código do serviço web e também artefactos tais como WSDL, descritores de serviços, como é o caso do ficheiro services.xml, documentação, entre outros, necessários para o correcto funcionamento sob o servidor HTTP.
- Um adaptador, já sob o contexto de execução de um servidor HTTP, realiza o carregamento do código do serviço e fica à escuta de pedidos.

A fase de compilação é aprofundada na primeira secção, com especial ênfase ao mecanismo que invoca o Doxygen, projecto disponível em [dox12a], e subsequente geração dos vários artefactos, WSDL, services.xml, etc.

A segunda secção descreve as características dos três adaptadores disponibilizados pela *ANSWER* - Axis2CAdapter FastCGIAdapter e ApacheAdapter-, bem como seus os aspectos de configuração.

3.1 Processo de construção

A compilação do código de serviço visa criar a biblioteca dinâmica a ser carregada pela plataforma quando integrada num servidor web, bem como os múltiplos artefactos que são necessários à publicação e funcionamento de serviços web.

Embora não seja imposta a utilização de um sistema específico para a construção do código de serviço o trabalho de projecto incluí um módulo para o sistema CMake, que facilita a integração do processo de geração de artefactos no processo de construção messe sistema. A integração com outro sistema de construção (ANT, JAM, Automake, etc.) é deixado ao critério do utilizador. Seguidamente é descrito o funcionamento do módulo CMake.

3.1.1 Módulo CMake CPP_WSDL

O módulo de CMake disponibilizado pela *ANSWER* é composto por um ficheiro de nome CPP_WSDL.cmake que define a macro CPP_WSDL.

O código da macro CPP_WSDL é apresentada em totalidade no anexo 3, no entanto seguidamente é sucintamente descrito o seu funcionamento.

A macro CPP_WSDL, define um passo adicionar no sistema de construção após a compilação do código do projecto. Este passo executa uma sequência de comandos, recorrendo à variáveis disponibilizadas pelo sistema CMake. Os comandos são:

- Executar o doxygen sobre as directorias de inclusão de headers C++, exportando o modelo semântico XML.
- Processar o XML gerado, iterativamente chamando o executável xsltproc [xsl12] com os vários XSLTs, resultando na geração de artefactos, tais como WSDL, services.xml, etc.

A utilização da macro no projecto é listada em 3.1. É necessária a inclusão o ficheiro que a define, CPP_WSDL.cmake. A macro, CPP_WSDL pode então ser chamada, necessitando que lhe seja passada o nome do projecto. Da chamada à macro resulta a criação da variável `CPP_WSDL_OUTPUT`, que contêm os o caminho para os ficheiros gerados, para que possam ser instalados juntamente com a biblioteca dinâmica do código do projecto.

Listagem 3.1: Utilização da macro CMake CPP_WSDL

```
include (CPP_WSDL)
CPP_WSDL(${LIBRARY_NAME})

install (FILES ${CPP_WSDL_OUTPUT} DESTINATION ${DEST_PATH})
```

3.1.2 Modelo interno do Doxygen

No início do trabalho de projecto a componente CPP_WSDL do AnubisNetworks Webservices Alpha demonstrou ter vários problemas. A complexidade do código ao recorrer a uma solução de *parsing* da AST de C++, agravada pela necessidade de mudança associada à constante evolução do modelo de programação durante o decorrer do trabalho de projecto levaram a que outras alternativas fossem exploradas. Foram analisadas outras bibliotecas de AST, bem como *parsers Flex/Bison*. Uma vez que o Doxygen é *open-source*, este foi também analisado, para perceber resolvia o problema do *parse* de ficheiros C++. Da análise da arquitectura, descrita na figura 3.2, optou-se por utilizar directamente o Doxygen, exportando o modelo semântico XML, ao invés de replicar a funcionalidade de *parsing*. Desta forma é delegado a um projecto maduro a tarefa de *parsing*.

O ficheiro de configuração utilizado com o *Doxygen*, correctamente parametrizado para gerar apenas o modelo XML é apresentado no anexo 5

3.1.3 XSLTs de geração

A *Extensible Stylesheet Language Transformations* (XSLT) é uma linguagem declarativa para a transformação de um XML em outros documentos, XML HTML, texto

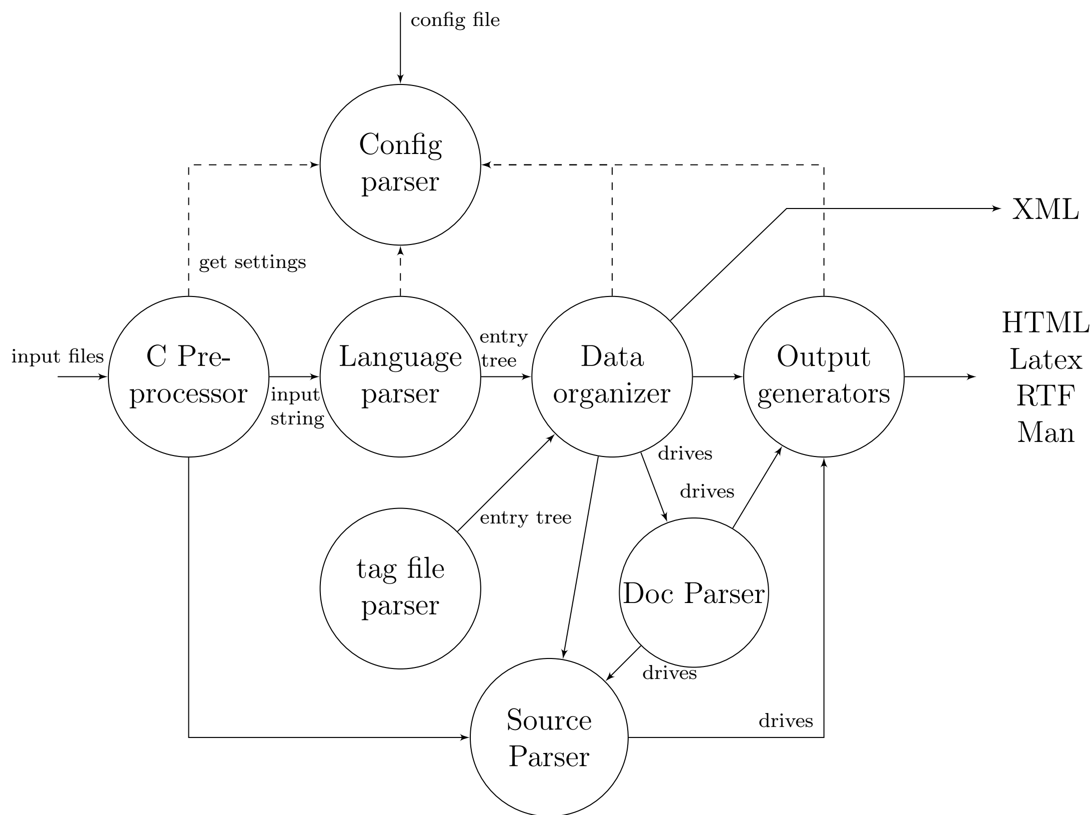

Figura 3.2: Arquitectura do doxygen (adaptado de [dox12b])

simples entre outros. Como tal foi escolhida para implementar a transformação do modelo semântico das classes de serviços do XML gerado pelo Doxygen nos vários ficheiros necessários à publicação do serviço.

Durante a análise do Doxygen, foram testadas várias versões, tendo-se concluído que o modelo semântico era estável e simples o suficiente para evitar a criação de um modelo intermédio próprio. Como exemplo do tipo de código para processar esse modelo, descreve-se o template recursivo `dataElementIsComplex` na listagem 3.2, utilizado pelos vários XSLTs para determinar se um dado tipo é composto ou simples.

Listagem 3.2: Função XSLT `dataElementIsComplex`

```
<!-- Template to identify whether class variables are basic data ↩
    types -->
<!-- Arguments -->
<!--  fileID - id reference of a class or struct file-->
<!-- Return true if type is complex, nothing otherwise-->
<xsl:template name="dataElementIsComplex">
<xsl:param name="fileID"/>
```

```
<xsl:if test="$fileID != ''">
<xsl:for-each select="document( concat( $fileID, '.xml'))//sectiondef↩
    [@kind='public-attrib']/memberdef[@kind='variable']">
<xsl:variable name="type" select="type" />
<!-- Define a var that contains something like std::list or vector --↩
    >
<xsl:variable name="collectionType" select="substring-before(type/↩
    text(),'&lt;')" />
<!-- Use context instead of var -->
  <xsl:choose>
    <xsl:when test="$collectionTypes/item[@value=$collectionType]">
      <xsl:value-of select="true()"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:choose>
        <xsl:when test="$wsdlTypes/pair[@key=$type]">
          <xsl:call-template name="dataElementIsComplex">
            <xsl:with-param name="parameter" select="type/ref[1]/↩
                @refid"/>
          </xsl:call-template>
        </xsl:when>
      </xsl:choose>
    </xsl:otherwise>
  </xsl:choose>
  </xsl:for-each>
</xsl:if>
</xsl:template>
```

Services.xml

O ficheiro services.xml, descrito em 2.2.1 é utilizado pelo módulo do Axis2, através do adaptador Axis2 Adapter para o carregamento das bibliotecas, tal como descrito em 2.2.1. Com esse objectivo foi recriada a geração do ficheiro, através da criação do XSLT services.xml.xslt.

WSDL

O ficheiro wsdl.xslt implementa a geração dos WSDL de serviço. Por questões de tempo, não foi ainda possível implementar opções que permitam controlar os vários *namespaces* presentes no WSDL. De momento é utilizado o namespace *http://answer* em todos elementos que necessitem da definição de um *namespaces*, tais como tipos, *bindings* e *portType*.

Json Payloads

O ficheiro wsdl.xslt implementa a geração dos WSDL de serviço. Por questões de tempo, não foi ainda possível implementar opções que permitam controlar os vários *namespaces* presentes no WSDL. De momento é utilizado o namespace *http://answer* em todos elementos que necessitem da definição de um *namespaces*, tais como tipos, *bindings* e *portType*.

Descrição do serviço

A nível empresarial surgiu a necessidade de exportar a definição dos serviços. Resultou dessa necessidade um XSLT que define uma simples descrição dos serviços e os seus tipos de pedidos/respostas em SOAP ou Recursos e Representação em REST. Embora não tenha presentemente nenhum uso interno, caso se verifique uma alteração significativa do modelo semântico do Doxygen é possível passar a utilizar o modelo gerado por este XSLT como modelo intermédio.

Documentação

O XSLT de documentação de serviços web trata a criação de um ficheiro HTML, por projecto, que descreve os vários tipos e apresenta um formulário para realizar um pedido de teste. Como exemplo do output apresenta-se a figura 3.3 onde se visualiza em *browser* o HTML resultante da documentação do serviço com a uma operação `Login`.

login

Description

Login.

Request Type

POST

Request Payload

<username> string </username>
<password> string </password>

Response Payload

<accepted_terms> boolean </accepted_terms>

Figura 3.3: Exemplo de documentação da uma operação

3.2 Adaptadores e sua configuração

Antes de ter sido introduzida a noção de adaptador, a *ANSWER* apenas encapsulava o módulo Axis2/C, adaptando o modelo de programação de forma semelhante ao que acontecia em 2.2.6. Ao adicionar o suporte ao modelo de programação REST o código foi alterado, criando uma alternativa que utilizava FastCGI. Da reestruturação do código comum a ambos surgiram os dois adaptadores Axis2CAdapter e FastCGIAdapter. Pouco depois, foi adicionado um terceiro, o ApacheAdapter, que implementa um módulo nativo de Apache, evitando alguns dos problemas do FastCGI neste servidor. Desta forma foi cumprido o objectivo do suporte a múltiplos servidores HTTP, recorrendo à utilização do módulo apropriado, como se ilustra na figura 3.4

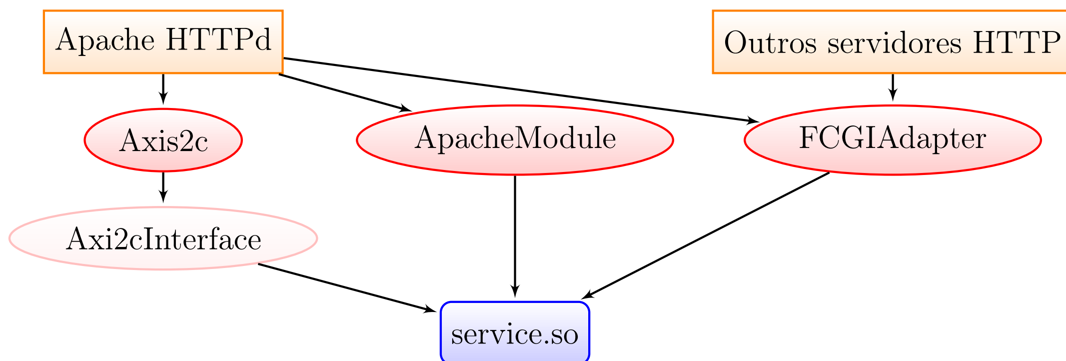

Figura 3.4: Diagrama de correspondência de adaptadores e servidores HTTP

Na tabela 3.1 são descritas as principais diferenças e características entre os vários adaptadores.

	Adaptador		
	Axis2c	FastCGI	ApacheModule
SOAP	Sim	Não	Não
XML/RPC	Sim[1]	Sim	Sim
REST	Parcial[1]	Sim	Sim
Módulos de Fluxo	Sim	Sim	Sim
Suporta IPv6	Não	Sim	Sim
Codecs	Não[2]	Sim	Sim
Servidores HTTPd	Apenas Apache	Múltiplos (FCGI)	Apenas Apache

Tabela 3.1: Capacidades dos adaptadores

As configurações dos adaptadores são apresentadas como configurações de do servidor HTTP Apache. Atendendo às necessidades do projecto e capacidades de

[1] Suporte é restrito à noção de REST Gateway do Axis2
[2] Apenas XML

cada Adaptador, apresentadas na tabela 3.1, apenas é necessário que um deles esteja activo.

Destaca-se que a alteração do adaptador escolhido não implica o passo de recompilação do código de serviço, nem de módulos da plataforma *ANSWER*.

3.2.1 Adaptador Axis2/C

O suporte a SOAP já se encontra implementado no projecto Axis2/C, mas como se descreve no Estado da Arte o modelo de programação é complexo.

O adaptador *Answer-Axis2/C* delega ao módulo para Apache Axis2/C o carregamento das bibliotecas de código de serviços criadas com o modelo de programação *ANSWER*, ao fornecer uma camada de adaptação. O módulo de Axis realiza o *dl_open* sobre o adaptador, que responde correctamente à invocação de *dl_sym*. Internamente o adaptador carrega a biblioteca com o código do serviço.

Para solucionar alguns dos problemas detectados com o Axis2/C foram disponibilizados pela empresa *AnubisNetworks patches* com soluções para alguns dos problemas detectados, entre os quais [axi12a] e [axi12b].

Configuração

Caso se pretenda utilizar o adaptador *Answer-Axis2/C* é necessário configurar o módulo Axis2/C de apache como seria normal a uma instalação de Axis2/C. Um exemplo dessa configuração é listado em 3.3, onde se configura a localização `/services` como base do URI onde são publicados os serviços.

Listagem 3.3: Configuração do Axis2/C

```
LoadModule axis2_module  /usr/axis2c/lib/libmod_axis2.so
<IfModule axis2_module>
  Axis2RepoPath /usr/axis2c
  Axis2LogFile logs/axis2c.log
  Axis2MaxLogFileSize 1000
  Axis2LogLevel warn
  <Location /services>
    SetHandler axis2_module
  </Location>
</IfModule>
```

3.2.2 Adaptador FastCGI

FastCGI é um protocolo [Bro96] que fornece uma interface que é utilizada para a interacção de programas binários com servidores web. A principal diferença do

FastCGI com o protocolo CGI que o antecede é a capacidade de reutilizar o mesmo binário em pedidos subsequentes, evitando a destruição da aplicação e a criação do processo a cada pedido.

O adaptador FastCGI cumpre o objectivo da independência de servidor HTTP, já que entre os servidores que possuem suporte a FastCGI listam-se o Apache Webserver [apa12], NGinx [ngi12], Lighty [lig12]. O adaptador é instalado como um binário FastCgi, a ser carregado pelo servidor web. Como exemplo apresenta-se na listagem 3.4 um exemplo de configuração deste, em apache, recorrendo ao módulo fastcgi. A localização /fcgiservices é configurada como base do URI onde são publicados os serviços.

Configuração

Listagem 3.4: Configuração do adaptador FastCGI em Apache

```
LoadModule fastcgi_module  modules/mod_fastcgi.so
<IfModule fastcgi_module>
  FastCgiServer "/opt/answer/cgi-bin/answer-fcgi" -processes 1
  ScriptAlias /fcgiservices /opt/wps/lib/cgi-bin/answer-fcgi
</IfModule>
```

3.2.3 Adaptador módulo nativo de Apache

A nível empresarial surgiu a necessidade de criar um adaptador, com suporte a REST, sem recurso a FastCGI. Para cumprir esse requisito foi criado o adaptador ApacheAdapter, implementado como módulo nativo de Apache. Possui as mesma capacidades do módulo de FastCGI, ou seja o suporte a RPC e REST, mas tem melhor desempenho, já que evita um nível de indirecção imposto pelo módulo FastCGI.

Configuração

Na listagem 3.5 apresenta-se um exemplo de configuração do adaptador nativo de apache. Destacam-se os parâmetros *ANSWER*ModulesDir e *ANSWER*ServicesDir que indicam a localização das bibliotecas dinâmicas dos módulos e serviços, respectivamente.

Listagem 3.5: Configuração do adaptador módulo nativo de Apache

```
LoadModule answer_module modules/mod_answer.so
<IfModule answer_module>
  ANSWERAxisRequestFormat On
  ANSWERModulesDir /opt/answer/modules
  ANSWERServicesDir /opt/answer/services
```

```
  <Location "/services">
    SetHandler answer_module
  </Location>
</IfModule>
```

3.2.4 Módulos *ANSWER*

Os módulos *ANSWER*, quer os disponibilizados pela plataforma ou os criados pelo utilizador poderão ser instalados enquanto ficheiros em qualquer ponto sistema. Ao ser iniciada a plataforma estes são carregados e registados automaticamente. Não existe presentemente nenhum mecanismo formal aquando do registo para declarar dependências entre módulos ou ordem de carregamento. No entanto os módulos encontrados no directório configurado para o efeito são carregados por ordem alfabética. Tirando partido desta característica, sugere-se que enquanto essa funcionalidade não estiver seja formalizada, se recorra ao renomear de ficheiros ou criação de *links* simbólicos, para forçar a ordem do carregamento, como se exemplifica na listagem 3.6. Recorrendo a uma estrutura de ficheiros semelhante bastaria configurar a directoria de módulos para answer/modules.ordered.

Listagem 3.6: Exemplificação do uso de symlinks para ordenação do carregamento de módulos

```
.
`-- answer
    |-- modules
    |   |-- authorization.so
    |   |-- customModule2.so
    |   `-- customModule1.so
    `-- modules.ordered
        |-- 01_authorization.so -> ../modules/authorization.so
        |-- 02_customModule2.so -> ../modules/customModule2.so
        `-- 03_customModule1.so -> ../modules/customModule1.so
```

Capítulo 4

Modelo de Programação

Como referido nos objectivos, um dos principais requisitos de desenho do projecto é a facilidade de utilização da plataforma. Este requisito reflecte-se fortemente no modelo de programação, analisado em detalhe neste capítulo.

As duas primeiras secções apresentam o modelo de programação para serviços RPC/SOAP e REST. Destaca-se que, ao contrário das alternativas analisadas no capítulo Estado da Arte, a plataforma *ANSWER* suporta simultâneamente a criação serviços em ambos os estilos.

A terceira secção apresenta o modelo de configuração de instanciação, sendo aplicável simultâneamente para serviços implementados em RPC/SOAP ou REST. Esta configuração controla quando é instanciado o objecto que responde a um pedido, e se o mesmo é reutilizado em pedidos subsequentes.

Na quarta secção é descrito o modelo de programação para a definição de Codecs. Os Codecs são os objectos responsáveis por seriar para um formato especifico as instâncias dos objectos de resposta (serviços RPC) ou recursos (serviços REST).

A quinta secção apresenta o ciclo de vida de um pedido e o modelo de programação para a criação de módulos *ANSWER*. Estes módulos executam código do utilizador em pontos específicos do ciclo de vida dos pedidos a serviços.

Na sexta e última secção é analisada a hierarquia de tipos de excepções disponibilizados pela plataforma *ANSWER* e é descrito a sua manipulação.

4.1 Modelo de Programação RPC/SOAP

Em seguida apresenta-se o modelo de programação para RPC/SOAP, recorrendo ao exemplo da calculadora utilizado em capítulos anteriores.

No final são comparadas as diferenças relevantes entre o modelo proposto e os anteriores, por forma a evidenciar as melhorias obtidas.

4.1.1 Definição de Tipos de Mensagens

No exemplo da calculadora os pedidos são representados por instâncias da classe `CalculatorRequest`, presente na listagem 4.1.

Listagem 4.1: Definição da classe de pedido

```
struct CalculatorRequest{
  double operand1;
  double operand2;
  char operation;

  template<class Archive>
  serialize(Archive &ar, const unsigned int /*version*/){
    ar & BOOST_SERIALIZATION_NVP(operand1);
    ar & BOOST_SERIALIZATION_NVP(operand2);
    ar & BOOST_SERIALIZATION_NVP(operation);
  }
};
```

Idealmente nas classes dos tipos de mensagem, como é o caso do tipo de pedido, apenas constariam os campos que compõem o conteúdo das mensagens que representam. Porém a definição da seriação tem de ser explicita, não podendo ser obtida automaticamente com recurso a introspecção, ao contrário do que acontece noutras linguagens. De uma análise das várias bibliotecas de seriação [oth12] optou-se pela integrar a Boost::Serialization. Esta permite seriação de membros de instância, como na função template `serialize` do exemplo listado em 4.1, mas também colecções da biblioteca STL, ponteiros, referências, objectos com hierarquia, e STD como strings, etc.

Foi considerada a possibilidade de recorrer à execução de um qualquer programa externo em tempo de compilação que criaria a definição da função de seriação para que esta estivesse presente para o *linker*. Embora seja princípio de desenho do *ANSWER* a simplicidade do modelo de programação, considerou-se que os problemas levantados ao recorrer a essa técnica, nomeadamente a criação de ficheiros intermédios e abstracção de funcionalidade potencialmente útil ao utilizador não justificaria o retorno em simplicidade. Desta forma as definições dos tipos de pedidos e resposta na plataforma *ANSWER* são feitas em C++ puro, evitando assim quebrar objectivo proposto de não recorrer a técnicas de linguagens intermédias ou ferramentas de *parsing* de código (ou meta-linguagens em comentários) para geração de código intermédio.

Uma análise mais detalhada da integração e extensão da Boost::Serialization na plataforma *ANSWER* será feita no capítulo seguinte.

A definição do tipo de resposta é apresentado na listagem 4.2, na classe *CalculatorResponse*. De forma semelhante à classe de pedido *CalculatorRequest* esta apenas possuí os atributos membro e a função de *template*, que indica à plataforma quais os membros a seriar.

Listagem 4.2: Definição da classe de resposta

```
struct CalculatorResponse{
  double result;
  std::string text;

  template<class Archive>
  serialize(Archive &ar, const unsigned int /*version*/){
    ar & BOOST_SERIALIZATION_NVP(result);
    ar & BOOST_SERIALIZATION_NVP(text);
  }
}
```

4.1.2 Definição de Serviço

A classe `MyService`, que define o comportamento do serviço, encontra-se descrita na listagem 4.3. É responsabilidade desta classe a definição e implementação das várias operações, enquanto funções membro. No exemplo dado, apenas é descrita a função *calculator*. Cada uma destas funções membro é registada na plataforma *ANSWER* utilizando a macro *REGISTER_ OPERATION*, que recebe como parâmetro a função em questão.

Listagem 4.3: Definição da classe de serviço e registo da operação

```
class MyService{
  public:
  CalculatorResponse calculator(const CalculatorRequest &request);
}

REGISTER_OPERATION(MyService::calculator)
```

O registo da função membro implica que esta será disponibilizada enquanto uma operação SOAP/RPC, com nome idêntico à função membro, *calculator* neste exemplo.

É também possível definir que a operação seja disponibilizada sobre um nome que não o da função membro, recorrendo ao código de exemplo da listagem 4.4, que consiste na própria definição da macro `REGISTER_OPERATION`, um pouco alterada.

Listagem 4.4: Definição da classe de serviço e registo da operação

```
answer::RegisterOperation<BOOST_TYPEOF(&MyService::calculator)> ↩
    registrator("Um-Outro.Nome", &MyService::calculator);
```

4.1.3 Comparação com outros modelos de programação

Destaca-se a simplicidade desta versão em comparação ao código necessário para realizar o mesmo serviço com as múltiplas plataformas para a linguagem C++ apresentadas no capítulo **Estado da Arte - Estilos e tecnologias para serviços web**, em especial com a *AnubisNetworks Webservices Alpha* 2.10. Não é colocada nenhuma imposição de herança, e ao invés de ser necessário uma classe para representar uma operação singular esta é agora uma função membro. Esta alteração é possui duas vantagens: A representação de função descreve mais aproximadamente a operação, representando directamente o pedido e resposta como parametro de entrada e retorno da função respectivamente. Torna também possível que um mesmo conjunto de operações partilhe a mesma lógica de construção através construtor da classe de serviço.

Denota-se ainda o facto da definição da função *serialize* ser conceptualmente idêntica entre as classes de resposta e de pedido. É assim possível que o mesmo tipo seja utilizado simultâneamente como pedido (onde é de-seriado) e resposta (onde é seriado).

4.2 Modelo de Programação REST sobre HTTP

A plataforma *ANSWER* suporta a criação de serviços REST recorrendo à representação de tipos de Recursos e Controladores.

A noção de recurso é analoga ao conceito em REST. Os controladores definem quais os métodos HTTP suportados para aceder ao recurso que controlam.

4.2.1 Definição de Recurso

A definição de recurso, central à definição de serviços web REST, é mapeada na plataforma *ANSWER* através de uma classe DTO. A listagem 4.5 define a classe `VCard`, que representa o estado da uma versão simplificada de um cartão vCard [DH98], identificado unicamente por um UUDI.

Listagem 4.5: Definição da classe Recurso VCard

```
#include <boost/serialization/list.hpp>
```

```
class VCard{
public:
  boost::uuids::uuid id;
  std::string name;
  std::list<std::string> email;

  template<class Archive>
  void serialize(Archive &ar, const unsigned int /*file_version*/){
    ar & BOOST_SERIALIZATION_NVP(id);
    ar & BOOST_SERIALIZATION_NVP(name);
    ar & BOOST_SERIALIZATION_NVP(email);
  }
};
```

Denota-se que para realizar a seriação de colecções da `STL`, como é o caso de da lista de *strings* email, apenas é necessária a inclusão do header apropriado: *<boost/serialization/list.hpp>*.

4.2.2 Definição de Controlador

O controlador é na plataforma *ANSWER* a classe que responde ao pedido sobre um determinado recurso, identificado por um URI. É da responsabilidade do controlador a definição e implementação dos métodos HTTP que pretende disponibilizar, implementado-os como funções membro com o mesmo nome. Na listagem 4.6 apresenta-se a classe de exemplo `vCardController`. Nesta listam-se as funções membro `Get`, sobrecarregadas para representar para os métodos HTTP GET de retorno de uma lista de `VCard`, ou apenas um, mediante um identificador. A função membro `Delete` corresponde ao método HTTP DELETE, efectuando a remoção do `VCard` identificado como parâmetro. A função membro `Post` corresponde ao método HTTP POST e criação ou actualiza um dado `VCard`. No exemplo do `VCardController` é presente a função membro `RetrieveAddressBook`, que implementa a listagem dos *vCards* que fazem parte da lista de endereços de um dado *vCard*. Desta forma é possível associar ao mesmo controlador níveis mais profundos do template de URI, aquando do ser registo, como se descreve de seguida.

Listagem 4.6: Declaração do controlador VCardController

```
class VCardController {
  using boost::uuids::uuid;
public :
  const list<VCard> Get () const;
  const VCard Get ( const uuid &id ) const;
```

```
  void Delete ( const uuid &id ) ;
  void Post ( const VCard& teacher );

  const list<VCard> RetrieveAddressBook( const uuid& id ) const;
};
```

Na listagem 4.7 é realizado o registo do controlador *VCardController* através da macro `REGISTER_HANDLER`, que recebe como parâmetros a classe a registar, a classe que representa o recurso, o tipo de identificação do recurso e o template de URI correspondente. O tipo de identificação será utilizado na validação do parâmetro do template de URI. No exemplo apresentado o tipo de identificação é um UUID, o que implica que num acesso GET, realizado ao URI */vcards/{uuid}* apenas são permitidos UUIDs válidos no parâmetro *{uuid}*. Caso o URI acedido contenha um UUID inválido, será retornado um *status* HTTP *400 Bad Request*.

Por questões de desenho é desejável que a um controlador possa ser atribuída a responsabilidade de mais endpoints parametrizáveis. Pretende-se por exemplo que a função `RetrieveAddressBook`, que retorna a lista dos VCards associados a um dado VCard, identificado pelo seu UUID, seja mapeada em */vcard/{num}/addressbook*. Isto é possível utilizando a macro REGISTER_HANDLER_ENDPOINT, como se demonstra no mesmo exemplo, onde passada a função membro e o template de URI. Não é necessário referir o tipo de identificador como acontece no registo do controlador em si, já que este é extraído do(s) parâmetro(s) da função membro registada.

Listagem 4.7: Registo do controlador

```
REGISTER_HANDLER(VCardController,
                 VCard,
                 boost::uuids::uuid,
                 "/vcards/{num}");

REGISTER_HANDLER_ENDPOINT(VCardController::RetrieveAddressBook,
                          "/vcard/{num}/addressbook");
```

De forma semelhante ao modelo de programação RPC/SOAP destaca-se que a plataforma não impõe qualquer herança, quer nos tipos que representam recursos quer nos controladores.

4.3 Controlo da estratégia instanciação

O controlo de instanciação é o mecanismo que define qual a estratégia usada sobre a instância do objecto que atende o pedido. Este controlo é útil tanto para serviços

implementados em RPC/SOAP como em REST.

O trabalho de projecto inclui os seguintes tipos de instanciação:

SingleCall - Por cada pedido é instanciada uma classe e sobre esta é invocada a operação.

Singleton - É criada uma instância assim que o servidor é criado. Esta instância é utilizada em todos os pedidos da suas operações.

LazySingleton - Idêntica ao tipo Singleton, excepto que a instância é criada apenas na primeira invocação de uma operação que lhe pertença.

O tipo de instanciação a utilizar é especificado directamente na classe de serviço, como se exemplifica na listagem 4.8, onde se alterou a classe `MyService` originalmente listada em 4.3, adicionando o typedef InvokationStrategyType para `answer::instantiation::LazySingleton`.

Listagem 4.8: Alteração do tipo de instanciação

```
class MyService{
  public:
  CalculatorResponse calculator(const CalculatorRequest &request);

  typedef answer::instantiation::LazySingleton InvokationStrategyType↩
      ;
}
```

Desta forma apenas quando a plataforma receber o primeiro pedido para a operação *calculator* é que é criada uma instância de MyService, que será reutilizada nos pedidos subsequentes às suas operações.

4.4 Formatação

Por forma a suportar outros tipos de seriação de resposta que não apenas XML, espelhando o modelo de *Codecs* presentes na plataforma *OpenRasta*.

Entre as outras vantagens da utilização da biblioteca de seriação Boost encontra-se a possibilidade de definir novos arquivos, classes que comandam a sintaxe de escrita (ou leitura) da informação a ser alvo de seriação.

Ao nível a plataforma *ANSWER* é possível enumerar dois tipos distintos de Codecs, os genéricos e os específicos. Os genéricos, implementam a noção de arquivo da biblioteca Boost::Serialization e permitem a seriação de qualquer tipo. Presentemente a plataforma *ANSWER* disponibiliza dois destes, XML, para input e output e

JSON, para output. De momento não é possível registar Codecs genéricos adicionais, salvo por inclusão directa e modificação de código na plataforma.

Os codecs *Específicos*, estão directamente associados a uma classe de pedido ou resposta, para serviços RPC, ou classe de recurso, para serviços REST.

Num pedido normal, a seriação em XML de uma instância de VCard resulta um XML semelhante ao apresentado na listagem 4.9.

Listagem 4.9: Exemplo da seriação de um VCard em XML

```
<vCard>
  <id>1553dbda-5f2c-4355-86b0-8ac0938073a1</id>
  <name>Carl Sagan</name>
  <!--zero or more elements of the following-->
  <email>carlsagan@universe.com</email>
</vCard>
```

Porém, pode também ser útil, do ponto de vista aplicacional, dar suporte a pedidos que pretendam enviar ou receber directamente os dados em formato `VCard`. O estilo arquitectural REST refere que para este efeito seja utilizado o atributo HTTP *Accept*, definindo o *mime-type* pretendido. Ao nível da plataforma *ANSWER* isto é concretizado mediante a especialização dos template Coder e Decoder, para dar suporte a pedidos e respostas, respectivamente. Observe-se o exemplo do código da listagem 4.10, onde se demonstra a seriação para suporte a text/vcard e x-vcard[1].

Listagem 4.10: Definição de um Codec para VCard

```
namespace answer {
namespace codec {
  template<>
  bool Coder<VCard>(std::ostream& out, const std::string& accept, ↩
      const VCard& vcard){
    if ( accept == "text/vcard" ) {
      out << "BEGIN:VCARD" << endl
          << "VERSION:3.0" << endl
          << "N:" << vcard.name << endl;
      for (std::list<std::string>::const_iterator it =
            vcard.email.begin();
            it != vcard.email.end(); ++it){
        out << "EMAIL:" << *it << endl;
      }
      out << "END:VCARD" << endl;
      return true;
    }else if ( accept == "text/x-vcard" ){
```

[1] código efectivo encontra-se omisso por questões de brevidade

```
      //(...)
    }
    return false;
  }
}
}
```

De forma semelhante descreve-se o modelo semântico para suportar a leitura do tipo VCard em formato vCard na listagem 4.11.

Listagem 4.11: Definição de um Decoder para VCard

```
namespace answer {
namespace codec {
  template<>
  bool Decoder<VCard>(std::istream& in, const std::string& accept, ↩
     VCard& vcard){
    if ( accept == "text/vcard" ) {
      //Read formated vcard
      vcard = VCard::fromStream(in);
      return true;
    }
    return false;
  }
}
}
```

4.5 Modelo dos programação de módulos

Da caracterização do ciclo de vida de um pedido, ilustrado na figura 4.1, surge a necessidade de suportar a criação de filtros/módulos para a realização de processamento durante a fase de entrada e saída.

Como exemplo da declaração de um módulo apresenta-se a listagem 4.12. Nela encontra-se a classe AuditModuleExample, módulo para auditoria de acesso / erros. Para tal, é necessário herdar de WebModule implementando os métodos que se pretendam que sejam chamados, de acordo com a figura 4.1. A definição correspondente deste módulo encontra-se listada em 4.13.

Listagem 4.12: Declaração de um módulo de fluxo

```
class AuditModuleExample: public anubis::webservice::WebModule{
  memcache::Memcache _memcacheServer;
public:
```

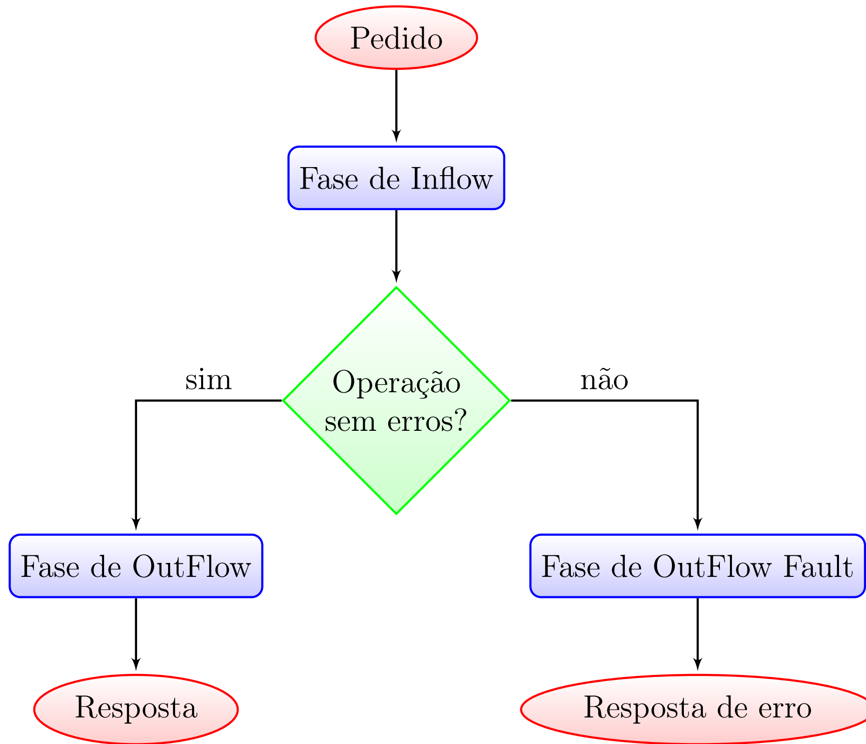

Figura 4.1: Ciclo de vida de um pedido

```
  virtual FlowStatus inFlow(anubis::webservice::InFlowContext &
      context);
  virtual FlowStatus outFlow(anubis::webservice::OutFlowContext &
      context);
  virtual FlowStatus outFlowFault(anubis::webservice::OutFlowContext
      &context);
};
```

Listagem 4.13: Definição de um módulo de fluxo

```
anubis::webservice::WebModule::FlowStatus Audit::inFlow(InFlowContext
    &context){
  cout << "AUDIT: Invoking "<< context.getOperation().getName() <<
      endl;
  return OK;
}
anubis::webservice::WebModule::FlowStatus Audit::outFlow(
    OutFlowContext &context){
  cout << "AUDIT: Done Invoking "<< context.getOperation().getName()
      << endl;
  return OK;
}
anubis::webservice::WebModule::FlowStatus Audit::outFlowFault(
    OutFlowContext &context){
```

```
  cerr << "AUDIT: Error Invoking "<< context.getOperation().getName()↩
      << endl;
  return OK;
}
static RegisterWebModule<Audit> _register("Audit");
```

As classes `InFlowContext` e `OutFlowContext`, disponibilizadas pela plataforma *ANSWER* contêm informação relevante ao pedido. Mediante estas é possível consultar qual a operação a ser executada, as suas parametrizações, qual o método utilizado para a invocação (SOAP, HTTP GET, HTTP POST, etc), bem como consultar e criar cookies, variáveis de ambiente, etc.

4.6 Excepções e tratamento

A *ANSWER* disponibiliza uma hierarquia de classes de excepções que deve ser utilizada pelo programador para que a plataforma realize o correcto processamento das situações de erro. A classe WebMethodException, base das restantes classes de excepção encontra-se descrita na listagem 4.14

Listagem 4.14: Classe de tratamento de excepção

```
class WebMethodException: public std::runtime_error {
  int _error_level;

public:
  WebMethodException(const std::string& message, int error_level);
  int getErrorLevel() const;
};
```

Como exemplo de utilização da hierarquia apresenta-se o código em 4.15, que foca a classe `UserID`, que descreve um utilizador, internamente representado por um UUID. A validação é realizada no construtor, que em caso de *input* inválido notifica a plataforma, utilizado a excepção `WebMethodInvalidInput`

Listagem 4.15: Classe de tratamento de excepção

```
class UserID{
  std::string _userID;
public:
  explicit UserID(const std::string &userID){
    //Check if this is a valid UUID
    try
    {
      boost::uuids::uuid id = boost::uuids::string_generator()(userID↩
          );
```

```
      _userID = boost::uuids::to_string(id);
    } catch (std::runtime_error &ex) {
      throw answer::WebMethodInvalidInput("Invalid scopeId");
    }
  }

  std::string& getUserId() const{ return _userID; }
};
```

Dado que excepções não tratadas resultam na terminação do processo de servidor HTTP os principais pontos de execução de código do utilizador encontram-se rodeados por um bloco *try*. O bloco *catch* associado captura instâncias de `std::exception`, registando a excepção na consola de erro. Por omissão muitos servidores web redireccionam o *stderr* para um ficheiro de auditoria. Embora básico este sistema é simples e eficaz e tem o benefício de não impor nenhum sistema ao utilizador da *ANSWER* ao contrário de plataformas apresentadas no Estado da Arte.

Capítulo 5

Aspectos de implementação

Neste capítulo focam-se aspectos de implementação dos vários mecanismos internos da plataforma. São analisadas as técnicas de programação que resultam na simplificação do modelo de programação para o utilizador final.

Para simplificar o modelo de programação e tendo em conta que o C++ não possui nenhum modelo (nativo) de suporte ao *runtime* que forneça um modelo de introspecção, recorreu-se ao uso de algumas técnicas/considerações da linguagem.

Estas técnicas baseiam-se na sua maioria em características da programação genérica, mais especificamente na capacidades de processamento de *templates* que C++ oferece.

5.1 Utilização de Macros

Sendo na sua essência um mecanismo de substituição, as macros revelam algumas lacunas. Entre as principais encontram-se a potenciação de colisões no espaço de nomes, já que não suportam a noção de *namespace*. Dificultam também a depuração de código, uma vez que realizada a substituição, na compilação não se sabe que foi uma macro que originou o código resultante.

Tendo estes cuidados em mente, foram definidas macros apenas quando estas substituem um troço de código que pretende realizar objectivo claro e conciso, mas cujo verdadeiro código dai resultante seja complexo. Exemplo prático a função de registo de operações, a macro `REGISTER_OPERATION`, cuja definição é listada em 5.1.

Listagem 5.1: Definição da macro `REGISTER_OPERATION`

```
#define REGISTER_OPERATION(ServiceOperation) \
  anubis::webservice::RegisterOperation<BOOST_TYPEOF(&↩
      ServiceOperation)> MAKE_UNIQUE(_registrator_)(#ServiceOperation↩
      , &ServiceOperation)
```

5.2 Registo automático

A especificação do C++ prevê que certos objectos sejam instanciados antes invocado do ponto de entrada de um programa. Tal é necessário para garantir que instâncias de objectos estáticos, ou globais estejam presentes para que possam ser imediatamente utilizados. A estes objectos o *scope* que controla do tempo de vida destes objectos é superior ao do troço principal do programa. Da execução do exemplo listado em 5.2 obtêm-se o seguinte resultado:

```
Constructing A
Entered main
```

Listagem 5.2: Programa exemplo de *scope* de instâncias globais

```
#include <iostream>

class A{
public:
  A(){
    std::cout << "Constructing A" << std::endl;
  }
};

A instA;

int main(){
  std::cout << "Entered main" << std::endl;
  return 0;
}
```

O mesmo ocorre aquando do carregamento uma biblioteca dinâmica, resultando na criação de todos os objectos estáticos ou globais que a biblioteca defina.

Esta característica é utilizada em vários pontos da plataforma, para registar automaticamente classes, evitando que seja necessário à biblioteca da plataforma ter conhecimento prévio de todas as definições a disponibilizar. Como exemplo descreve-se na listagem 5.3 a classe de *template* `RegisterWebModule`. Ao ser definida uma instância `_register` de `RegisterWebModule` como descrita na listagem 5.4 esta vai, aquando do carregamento do código chamar a função `registerModule` sobre o singleton de `WebModuleStore` registando uma instância de `Authentication`. Terminado o *scope* de vida de *_register* o destrutor de `RegisterWebModule` assegura que o módulo é removido da *Store*.

Listagem 5.3: Classe de registo automático `RegisterWebModule`

```
template <class T>
class RegisterWebModule{
  std::string _name;
public:
  RegisterWebModule(const std::string& name): _name(name){
    T *module = new T();
    WebModuleStore::getInstance().registerModule( _name, module );
  }

  ~RegisterWebModule(){
    WebModuleStore::getInstance().removeModule( _name );
  }
};
```

Desta forma não é necessário à plataforma conhecer ou incluir o ficheiro com a definição da classe `Authentication`. Versões mais avançadas desta técnica são utilizadas para facilitar o registo de operações RPC/SOAP e controladores REST, disponíveis na *ANSWER* através das macros `REGISTER_OPERATION`, `REGISTER_HANDLER` e `REGISTER_HANDLER_ENDPOINT`.

Listagem 5.4: Utilização da classe `RegisterWebModule`

```
static RegisterWebModule<Authentication> _register("Authentication");
```

5.3 Integração com Boost::Serialization

Um dos aspectos mais importantes da Boost::Serialization é a separação entre a listagem dos atributos a seriar, e a representação da seriação. Esta separação é concretizada recorrendo às funções de *template* `serialize` que definem quais os parâmetros, e as classes de Arquivo[1] (`Archive`) que definem a formatação. São disponibilizadas com a biblioteca Boost::Serialization modelos de arquivos binários, de texto simples e XML cada modelo contêm duas classes, uma para implementar o input e outra para output.

Tal como descrito em 4.4 são disponibilizadas pela plataforma *ANSWER* dois modelos de arquivo, o de XML, com suporte a input e output, e o de JSON com suporte de output apenas, já que não foi possível por questões de tempo implementar o de input.

A estrutura definida pelo modelo de seriação XML da Boost::Serialization não é compatível com o modelo pretendido, o que motivou a criação de um novo modelo de

[1] Também conhecidos por Codecs, na plataforma *ANSWER*

arquivo. Este modelo encontra-se implementado nos ficheiros ws_xml_iarchive.hpp ws_xml_oarchive.hpp, que implementam respectivamente o input (de-seriação) e output (seriação). De forma semelhante o arquivo criado para suportar JSON encontra-se implementado no ficheiro ws_json_oarchive.hpp.

5.4 Meta programação baseada em *templates*

O uso de Meta programação baseada em *templates* (Template metaprogramming) é utilizado como mecanismo de avaliação e processamento de algoritmos em tempo de compilação.

A biblioteca Boost possui no seu *namespace* MPL um vasto conjunto de funções que encapsula muitas das operações fornecendo uma interface de utilização simplificada.

Em conjunto com outras técnicas de processamento de *templates* é possível ao utilizador referir apenas as operações que pretende que sejam reconhecidas pela plataforma. Desta forma uma só classe define múltiplas operações, simplificando o modelo de programação e atingindo a noção desejada de função membro = operação web.

Definindo apenas a operação mediante *Class::FunctionMember* pode parecer limitativo, uma vez que impossibilita a definição de *overloads* desta mesma funções. No entanto tal nunca seria possível pois a informação necessária para diferenciação de tipos durante a de-seriação pode ser ambígua.

No código do *ANSWER* a utilização de *Template metaprogramming* é evidente no ficheiro OperationStore.hh, no qual está presente o código da listagem 5.5

Listagem 5.5: Código de registo de operação

```
template <typename Operation>
class RegisterOperation{
  std::string _operationName;
public:
  RegisterOperation(const std::string& operationName, const Operation↩
      &op):
    _operationName(operationName)
  {

    typedef typename class_<Operation>::type Type;

    typedef typename boost::function_types::result_type<Operation>::↩
       type
      response;
```

```
typedef typename boost::mpl::at_c<boost::function_types::↩
    parameter_types<Operation>,1>::type response_type;

typedef typename
  boost::remove_reference<
    response_type
  >::type const_request;

typedef typename
  boost::remove_const<
    const_request
  >::type request;

typedef typename ResolveStrategy<Type>::type Strategy;

const unsigned arity = boost::function_types::function_arity<↩
    Operation>::value;

/*
 * Get types of function
 * request (const & request),
 * where request is optional;
 *
 *   response (void) and arity == 2 -> RequestOnly
 *   response (!void) and arity == 2 -> RequestResponse
 *   response (!void) and arity == 1 -> ResponseOnly
 */
typedef BOOST_DEDUCED_TYPENAME boost::mpl::eval_if<
  BOOST_DEDUCED_TYPENAME boost::is_void<response>::type,
  BOOST_DEDUCED_TYPENAME boost::mpl::eval_if_c<
    arity == 2,
    boost::mpl::identity<
      RequestOnly<Type, Operation, request, InstantiationStrategy↩
          <Strategy, Type> >
    >,
    boost::mpl::void_ // for assert
  >,
  BOOST_DEDUCED_TYPENAME boost::mpl::eval_if_c<
    arity == 2,
    boost::mpl::identity<
      RequestResponse<Type, Operation, request, response, ↩
          InstantiationStrategy<Strategy, Type> >
    >,
    boost::mpl::identity<
```

```
        ResponseOnly<Type, Operation, response, ↩
           InstantiationStrategy<Strategy, Type> >
      >
    >
  >::type typex;

  try{
    OperationStore::getInstance().registerOperation(_operationName,↩
        new typex(op));
  }catch (std::exception &ex){
    std::cerr << "Error initializing operation ["<< _operationName ↩
      << ": " << ex.what() << std::endl;
  }
}
```

Em função da aridade da operação e do tipo de retorno é criado em tempo de compilação um *template* de uma das três classes, `RequestOnly`, `RequestResponse` ou `ResponseOnly`. Cada uma destas classes implementam a interface apresentada na listagem 5.7.

No exemplo 5.6 retirado do ficheiro ws_xml_oarchive.hpp, responsável por implementar a seriação XML das respostas utilizadas na plataforma, lista-se a função de *template* `save` que em tempo de compilação determina o tipo do objecto a ser seriado, destingindo entre `enum`, tipos primitivos, ou outros, utilizando uma classe especifica para cada caso durante a seriação.

Listagem 5.6: Classe operation

```
template<class T>
void save(const T &t){
  typedef
    BOOST_DEDUCED_TYPENAME boost::mpl::eval_if<
      boost::is_enum< T >,
      boost::mpl::identity<save_enum_type<ws_xml_oarchive> >,
      //else
      BOOST_DEDUCED_TYPENAME boost::mpl::eval_if<
      // if its primitive
        boost::mpl::equal_to<
          boost::serialization::implementation_level< T >,
            boost::mpl::int_<boost::serialization::primitive_type>
        >,
          boost::mpl::identity<save_primitive<ws_xml_oarchive>
        >,
      //else
    boost::mpl::identity<save_only<ws_xml_oarchive> >
  > >::type typex;
```

```
  typex::invoke(*this, t);
}
```

Listagem 5.7: Classe operation

```
class Operation {
public:
  Operation() {}
  virtual ~Operation() {};
  //The invocation wrapper
  virtual std::string invoke(const std::string&)=0;
};
```

É sobre a *store* destas operações que um adaptador invoca uma operação.

5.4.1 SFINAE

Substitution failure is not an error é um termo introduzido por David Vandevoorde [VJ02]. Descreve um conjunto de técnicas de programação que tira partido da situação em que durante a compilação de um programa a substituição inválida de parâmetros de *template* não constitui um erro.

Ao criar o conjunto de candidatos para resolução de *overload*, alguns (ou todos) os candidatos desse conjunto podem ser o resultado de substituir argumentos de *templates* deduzidos por parâmetros de *template*. Caso exista um erro durante a substituição, o compilador remove o *overload* em causa do conjunto, ao invés de parar a compilação com um erro. Se no final do processamento restar pelo menos um *overload* a resolução é dada como válida.

A importância desta funcionalidade é descrita no código da listagem 5.8, adaptado de [JWHL03].

Listagem 5.8: Exemplo de SFINAE

```
int negate(int i) { return -i; }

template <class F>
typename F::result_type negate(const F& f) { return -f(); }
```

Supondo que o compilador encontra a chamada `negate(1)`, a primeira definição é claramente a adequada, mas o compilador tem necessariamente de considerar (e instanciar os protótipos) de ambas as definições para chegar a essa conclusão. Da instânciação da segunda definição resultaria `int::result_type negate(const int&);` onde o tipo de retorno é inválido. Caso isto fosse um erro, a adição de funções de *template* (mesmo sem serem invocadas) iria quebrar código válido. Mas

dado o SFINAE a segunda definição é simplesmente removida do conjunto de resolução de *overloads*.

No código do *ANSWER* a utilização de SFINAE encontra-se ligada à determinação do tipo de instanciação pretendido, através das classes descritas na listagem 5.9, que são utilizadas em 5.5.

Listagem 5.9: Utilização de SFINAE para determinar estratégia de instânciação

```
template<typename>
struct void_ {
  typedef void check;
};

// The default strategy type
template<typename T, typename X = void>
struct ResolveStrategy{
  typedef instantiation::SingleCall type;
};

template<typename T>
struct ResolveStrategy <T, typename void_<typename T::↩
    InvokationStrategyType >::check> {
  typedef typename T::InvokationStrategyType type;
};
```

Esta técnica é utilizada para criar mecanismos de introspecção em tempo de compilação. Internamente a própria *Boost* utiliza esta técnica na componente de *metaprogramming* [boo12]

5.5 Detecção de tipos complexos

No contexto *RESTGateway* do Axis2/C tipos de pedidos podem ser mapeados como GET ou POST. O método HTTP a utilizar é está dependente da complexidade do tipo do pedido. Um tipo é considerado simples se é composto unicamente por elementos simples ou outros tipos simples.

A lista de tipos simples é listada como atributo chave, nos pares de `wsdlMapping` no xml cpp_dataTypes.xml, como descrito na listagem 5.10. Este utilizado pelos vários XSLTs de geração de ficheiros, não apenas pelo de gerador de WSDL.

Listagem 5.10: Elementos simples e seu mapeamento em tipos xml

```
<wsdlMappings>
```

```
  <pair key="std::string" value="xs:string"/>
  <pair key="char *" value="xs:string"/>
  <pair key="int" value="xs:int" />
  <pair key="double" value="xs:double" />
  <pair key="float" value="xs:float" />
  <pair key="short" value="xs:short" />
  <pair key="long" value="xs:long" />
  <pair key="unsigned int" value="xs:unsignedInt" />
  <pair key="unsigned short" value="xs:unsignedShort" />
  <pair key="unsigned long" value="xs:unsignedLong" />
  <pair key="unsigned" value="xs:unsignedInt" />
  <pair key="bool" value="xs:boolean" />
  <pair key="char" value="xs:string[maxLength=1]" />
</wsdlMappings>
```

Caso o tipo de pedido possua um atributo membro que não esteja listado em como chave em 5.10, este é considerado um tipo complexo. São também complexos o que utilizam colecções de tipos, sendo reconhecidas as colecções listadas em 5.10, presentes no mesmo ficheiro cpp_dataTypes.xml.

Listagem 5.11: Tipos de colecções reconhecidas

```
<collectionTypes>
  <item value="std::vector" />
  <item value="std::list" />
  <item value="std::deque" />
  <item value="std::set" />
  <item value="std::multiset" />
  <item value="std::map" />
  <item value="std::multimap" />
  <item value="[]" />
</collectionTypes>
```

Por vezes é necessário forçar que uma operação seja sempre realizada via POST. Um exemplo típico deste cenário é o da operação de *login* que recebe utilizador e senha. O tipo de pedido é composto por elementos básicos, o que implica que o tipo seja acessível através do método HTTP GET. Mas para integração numa página web é preferível não permitir que o pedido seja realizado por GET, já os dados, potencialmente sensíveis iriam surgir no ecrã. Para esse efeito é disponibilizada a macro `REGISTER_OPERATION_POST`, com a mesma sintaxe que `REGISTER_OPERATION`.

5.6 Considerações de *Threading*

Ao nível da integração da plataforma com o servidor HTTP, apenas uma instância de aplicação deve fazer uso da plataforma por pedido. No Apache esta configuração é denominada por *fork* (ou *pre-fork*, que inicializa um conjunto de instâncias a *à prióri* dos pedidos).

Espera-se que o suporte à execução concorrente seja terminado numa próxima versão da plataforma *ANSWER*. Ainda assim são descritos de seguida alguns detalhes de implementação directamente relacionados com este problema.

O padrão de desenho *singleton* utilizado em vários pontos da *ANSWER* é implementado com recurso à descrição por Scott Meyer's em [Mey95], é conhecido como *Meyers singleton*. Este padrão apenas é *thread-safe* na recente especificação de [iso11] consequência da seguinte regra:

```
§6.7 [stmt.dcl] p4
 If control enters the declaration concurrently while the variable is
 being initialized, the concurrent execution shall wait for
 completion of the initialization.
```

Foram alvo de análise problemas de concorrência em vários propostas implementações do padrão *singleton* em C++, no artigo [MA04]. Presentemente não foi ainda tomada qualquer decisão entre utilizar C++11 para compilar a componente da biblioteca da *ANSWER* ou recorrer a outro padrão.

Capítulo 6

Conclusões e Notas finais

Apresentaram-se neste trabalho aspectos de desenho e implementação da plataforma *ANSWER*, para o desenvolvimento de serviços web. Fez-se uma síntese crítica dos projectos existentes no mercado actualmente, identificando as lacunas que necessitavam de ser colmatadas e olhando a técnicas desejáveis de uma solução. Definiu-se a arquitectura para a plataforma, prestando especial atenção à modularidade, por forma a suportar a utilização em múltiplos servidores web, recorrendo ao conceito de adaptador. Foram descritos os vários Adaptadores, através dos quais se adicionou suporte a múltiplos servidores web. Estabeleceu-se o modelo de programação da *ANSWER*, tendo como principal critério de desenho a facilidade de uso, evitando por completo a utilização de meta-comentários. A plataforma *ANSWER* suporta todos os estilos arquitecturais e protocolos que foram propostos como objectivo, entre XML/RPC, SOAP (via uma abstracção do Axis2/C) e REST. Nenhuma das outras plataformas, nem as de outras linguagens analisadas no estado da arte suportam simultaneamente todos os estilos.

No entanto, ao observar o amplo âmbito do projecto é, natural que existam pontos em que este possa ser melhorado. Perspectivam-se de seguida alguns destes pontos, antecipando os futuros desenvolvimentos da plataforma *ANSWER*.

Embora se considere finalizado o modelo de programação no contexto do trabalho de projecto, a utilização contínua dos padrões e modelos descritos no capítulo **Aspectos de implementação**, poderão levar a subsequentes melhorias na API, facilitando ainda mais a sua utilização. Entre os possíveis alvos de melhorias destacam-se o mecanismo de definição de instanciação, especialmente no modelo para REST, e a definição de *Codecs*.

Ficam ainda por analisar as vantagens oferecidas pela última revisão da linguagem C++, o C++11 [iso11]. É possível que ao migrar o código de C++03 [iso03] sejam encontradas vantagens internamente, ao nível da plataforma, mas também ao

nível do modelo de programação apresentado.

Pela relação com o contexto profissional, centrado em *Linux* e sistemas abertos, não foi possível explorar a utilização da *ANSWER* noutros sistemas operativos. É interessante a criação de um adaptador que suporte o servidor web mais comum nos sistemas operativos *Windows*, o *Internet Information Services* (IIS).

Ainda assim, foram cumpridos e em muitos casos ultrapassados, os requisitos propostos na apresentação do projecto. Tratando-se de um trabalho de projecto que inclui uma implementação é importante destacar que a plataforma *ANSWER* já está a ser utilizada em produtos de mercado empresarial, instalados pela empresa *AnubisNetworks* em vários países do mundo.

Listagens de modelos de programação

.1 Exemplo do modelo de programação do Axis2/C

Listagem 1: Código de exemplo Axis2/C (retirado de [axi12d])

```
/*
 * Copyright 2004,2005 The Apache Software Foundation.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, ↩
    software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or ↩
    implied.
 * See the License for the specific language governing permissions ↩
    and
 * limitations under the License.
 */
#include <axis2_svc_skeleton.h>
#include <axutil_log_default.h>
#include <axutil_error_default.h>
#include <axutil_array_list.h>
#include <axiom_text.h>
#include <axiom_node.h>
#include <axiom_element.h>
#include <stdio.h>

axiom_node_t *axis2_hello_greet(const axutil_env_t *env,
```

```
    axiom_node_t *node);


int AXIS2_CALL
hello_free(axis2_svc_skeleton_t *svc_skeleton,
    const axutil_env_t *env);

axiom_node_t* AXIS2_CALL
hello_invoke(axis2_svc_skeleton_t *svc_skeleton,
    const axutil_env_t *env,
    axiom_node_t *node,
    axis2_msg_ctx_t *msg_ctx);

int AXIS2_CALL
hello_init(axis2_svc_skeleton_t *svc_skeleton,
    const axutil_env_t *env);


axiom_node_t* AXIS2_CALL
hello_on_fault(axis2_svc_skeleton_t *svc_skeli,
    const axutil_env_t *env, axiom_node_t *node);

axiom_node_t *
build_greeting_response(const axutil_env_t *env,
    axis2_char_t *greeting);

axiom_node_t *
axis2_hello_greet(const axutil_env_t *env, axiom_node_t *node)
{
  axiom_node_t *client_greeting_node = NULL;
  axiom_node_t *return_node = NULL;

  AXIS2_ENV_CHECK(env, NULL);

  if (node)
  {
    client_greeting_node = axiom_node_get_first_child(node, env);
    if (client_greeting_node &&
        axiom_node_get_node_type(client_greeting_node, env) == ↩
            AXIOM_TEXT)
    {
      axiom_text_t *greeting = (axiom_text_t *)↩
          axiom_node_get_data_element(client_greeting_node, env);
      if (greeting && axiom_text_get_value(greeting , env))
      {
```

```
            const axis2_char_t *greeting_str = axiom_text_get_value(↩
                greeting, env);
            printf("Client greeted saying \"%s\" \n", greeting_str);
            return_node = build_greeting_response(env, "Hello Client!");
        }
      }
    }
    else
    {
      AXIS2_ERROR_SET(env->error, ↩
          AXIS2_ERROR_SVC_SKEL_INVALID_XML_FORMAT_IN_REQUEST, ↩
          AXIS2_FAILURE);
      printf("ERROR: invalid XML in request\n");
      return_node = build_greeting_response(env, "Client! Who are you?"↩
          );
    }

    return return_node;
}

axiom_node_t *
build_greeting_response(const axutil_env_t *env, axis2_char_t *↩
    greeting)
{
    axiom_node_t* greeting_om_node = NULL;
    axiom_element_t * greeting_om_ele = NULL;

    greeting_om_ele = axiom_element_create(env, NULL, "greetResponse", ↩
        NULL, &greeting_om_node);

    axiom_element_set_text(greeting_om_ele, env, greeting, ↩
        greeting_om_node);

    return greeting_om_node;
}

static const axis2_svc_skeleton_ops_t hello_svc_skeleton_ops_var = {
    hello_init,
    hello_invoke,
    hello_on_fault,
    hello_free
};

axis2_svc_skeleton_t *
axis2_hello_create(const axutil_env_t *env)
```

```
{
  axis2_svc_skeleton_t *svc_skeleton = NULL;
  svc_skeleton = AXIS2_MALLOC(env->allocator,
      sizeof(axis2_svc_skeleton_t));

  svc_skeleton->ops = &hello_svc_skeleton_ops_var;

  svc_skeleton->func_array = NULL;

  return svc_skeleton;
}

int AXIS2_CALL
hello_init(axis2_svc_skeleton_t *svc_skeleton,
    const axutil_env_t *env)
{
  svc_skeleton->func_array = axutil_array_list_create(env, 0);
  axutil_array_list_add(svc_skeleton->func_array, env, "helloString")↩
      ;
  return AXIS2_SUCCESS;
}

axiom_node_t* AXIS2_CALL
hello_invoke(axis2_svc_skeleton_t *svc_skeleton,
    const axutil_env_t *env,
    axiom_node_t *node,
    axis2_msg_ctx_t *msg_ctx)
{
  return axis2_hello_greet(env, node);
}

axiom_node_t* AXIS2_CALL
hello_on_fault(axis2_svc_skeleton_t *svc_skeli,
    const axutil_env_t *env, axiom_node_t *node)
{
  axiom_node_t *error_node = NULL;
  axiom_node_t* text_node = NULL;
  axiom_element_t *error_ele = NULL;
  error_ele = axiom_element_create(env, node, "EchoServiceError", ↩
      NULL,
      &error_node);
  axiom_element_set_text(error_ele, env, "Echo service failed ",
      text_node);
  return error_node;
}
```

```
int AXIS2_CALL
hello_free(axis2_svc_skeleton_t *svc_skeleton,
    const axutil_env_t *env)
{
  if (svc_skeleton->func_array)
  {
    axutil_array_list_free(svc_skeleton->func_array, env);
    svc_skeleton->func_array = NULL;
  }


  if (svc_skeleton)
  {
    AXIS2_FREE(env->allocator, svc_skeleton);
    svc_skeleton = NULL;
  }

  return AXIS2_SUCCESS;
}

AXIS2_EXPORT int
axis2_get_instance(axis2_svc_skeleton_t **inst,
    const axutil_env_t *env)
{
  *inst = axis2_hello_create(env);
  if (!(*inst))
  {
    return AXIS2_FAILURE;
  }

  return AXIS2_SUCCESS;
}

AXIS2_EXPORT int
axis2_remove_instance(axis2_svc_skeleton_t *inst,
    const axutil_env_t *env)
{
  axis2_status_t status = AXIS2_FAILURE;
  if (inst)
  {
    status = AXIS2_SVC_SKELETON_FREE(inst, env);
  }
  return status;
}
```

.2 Modelo de programação do gSOAP

Listagem 2: Código de exemplo para a criação de um serviço em gSOAP (adaptado de [gso12b])

```
// Install as a CGI application.
// Alternatively, run from command line with arguments IP (which must↩
    be the
// IP of the current machine you are using) and PORT to run this as a
// stand-alone server on a port. For example:
// > magicserver.cgi machine 18081 &
// To let 'magic' talk to this service, change the URL in magic.cpp ↩
   into
// "http://machine:18081"
// where "machine" is the name of your machine or e.g. "localhost"

int main(int argc, char **argv)
{ struct soap soap;
  int m, s;
  soap_init(&soap);
  // soap.accept_timeout = 60; // die if no requests are made within ↩
      1 minute
  if (argc < 3)
  { soap_serve(&soap);
    soap_destroy(&soap);
    soap_end(&soap);  // clean up
  }
  else
  { m = soap_bind(&soap, argv[1], atoi(argv[2]), 100);
    if (m < 0)
      exit(-1);
    fprintf(stderr, "Socket connection successful %d\n", m);
    for (int i = 1; ; i++)
    { s = soap_accept(&soap);
      if (s < 0)
        exit(-1);
      fprintf(stderr, "%d: accepted %d IP=%d.%d.%d.%d ... ", i, s, (↩
          int)(soap.ip>>24)&0xFF, (int)(soap.ip>>16)&0xFF, (int)(soap↩
          .ip>>8)&0xFF, (int)soap.ip&0xFF);
      soap_serve(&soap);  // process RPC skeletons
      fprintf(stderr, "served\n");
      soap_destroy(&soap);
      soap_end(&soap);  // clean up
    }
  }
```

```
  return 0;
}

int ns1__magic(struct soap *soap, int n, matrix *square)
{ int i, j, k, l, key = 2;
  if (n < 1)
    return soap_receiver_fault(soap, "Negative or zero size", "The ↩
        input parameter must be positive");
  if (n > 100)
    return soap_receiver_fault(soap, "size > 100", "The input ↩
        parameter must not be too large");
  square->resize(n, n);
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      (*square)[i][j] = 0;
  i = 0;
  j = (n-1)/2;
  (*square)[i][j] = 1;
  while (key <= n*n)
  { if (i-1 < 0)
      k = n-1;
    else
      k = i-1;
    if (j-1 < 0)
      l = n-1;
    else
      l = j-1;
    if ((*square)[k][l])
      i = (i+1) % n;
    else
    { i = k;
      j = l;
    }
    (*square)[i][j] = key;
    key++;
  }
  return SOAP_OK;
}
```

Ficheiros de processo de construção e de configurações

.3 Macro de CMakel CPP_WSDL

Listagem 3: Macro de CMake CPP_WSDL

```
MACRO(CPP_WSDL)

get_property(INCLUDE_DIRS DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR} ↩
    PROPERTY INCLUDE_DIRECTORIES)
message(STATUS "CPP_WSDL ${includeDirs}")
add_custom_command(
  TARGET ${PROJECT_NAME}
  POST_BUILD
  #OUTPUT ${CMAKE_BINARY_DIR}/services.xml ${CMAKE_BINARY_DIR}/${↩
      PROJECT_NAME}.js
  COMMAND PROJECT_NAME=${PROJECT_NAME} INPUT_PATH=${↩
      CMAKE_CURRENT_SOURCE_DIR} doxygen ${CMAKE_CURRENT_SOURCE_DIR↩
      }/../tools/doxygen_cpp_wsdl/Doxyfile 2>&1 >/dev/null
  COMMAND xsltproc --stringparam project_name ${PROJECT_NAME} --path ↩
      "${CMAKE_BINARY_DIR}/xml" ${CMAKE_CURRENT_SOURCE_DIR}/../tools/↩
      doxygen_cpp_wsdl/services.xml.xslt index.xml > ${↩
      CMAKE_BINARY_DIR}/services.xml
  COMMAND xsltproc --path "${CMAKE_BINARY_DIR}/xml" ${↩
      CMAKE_CURRENT_SOURCE_DIR}/../tools/doxygen_cpp_wsdl/↩
      json_payloads.xslt index.xml > ${CMAKE_BINARY_DIR}/${↩
      PROJECT_NAME}.js
  COMMAND xsltproc --stringparam project_name ${PROJECT_NAME} --path ↩
      "${CMAKE_BINARY_DIR}/xml" ${CMAKE_CURRENT_SOURCE_DIR}/../tools/↩
      doxygen_cpp_wsdl/documentation.xslt index.xml > ${↩
      CMAKE_BINARY_DIR}/${PROJECT_NAME}_documentation.html
  COMMAND xsltproc --stringparam project_name ${PROJECT_NAME} --path ↩
      "${CMAKE_BINARY_DIR}/xml" ${CMAKE_CURRENT_SOURCE_DIR}/../tools/↩
      doxygen_cpp_wsdl/serviceDescriptor.xslt index.xml > ${↩
```

```
    CMAKE_BINARY_DIR}/serviceDescription.xml
  COMMAND xsltproc --stringparam project_name ${PROJECT_NAME} --path ↩
    "${CMAKE_BINARY_DIR}/xml" ${CMAKE_CURRENT_SOURCE_DIR}/../tools/↩
    doxygen_cpp_wsdl/services_js.xslt index.xml > ${↩
    CMAKE_BINARY_DIR}/services.js
  ${CMAKE_CURRENT_SOURCE_DIR}/xml/index.xml > ${CMAKE_BINARY_DIR}/${↩
    PROJECT_NAME}.js
)

set(CPP_WSDL_OUTPUT
  ${CMAKE_BINARY_DIR}/services.xml
  ${CMAKE_BINARY_DIR}/${PROJECT_NAME}.js
  ${CMAKE_BINARY_DIR}/services.js
  ${CMAKE_BINARY_DIR}/serviceDescription.xml
)

ENDMACRO(CPP_WSDL)
```

.4 Configuração do build de serviço em PocoProject::Remote

Listagem 4: Ficheiro de configuração para gerador PocoProject::Remote

```
<AppConfig>
    <RemoteGen>
        <files>
            <include>
                ${system.env.POCO_BASE}/Remoting/include/Poco/↩
                    Remoting/RemoteObject.h
                ${system.env.POCO_BASE}/Remoting/include/include/Poco↩
                    /Remoting/Proxy.h
                ${system.env.POCO_BASE}/Remoting/include/include/Poco↩
                    /Remoting/Skeleton.h
                include/WeatherStation.h
            </include>
        </files>
        <output>
            <mode>server</mode>
            <include>include</include>
            <src>src</src>
            <namespace>Sample</namespace>
            <copyright>Copyright (c) 2009</copyright>
        </output>
        <compiler>
```

```
            <exec>g++</exec>
            <options>
                -I${system.env.POCO_BASE}/Foundation/include
                -I${system.env.POCO_BASE}/Remoting/include
                -Iinclude
                -E
                -C
                -o%.i
            </options>
        </compiler>
    </RemoteGen>
</AppConfig>
```

Listagem 5: Configuração do Doxygen

```
PROJECT_NAME           = $(PROJECT_NAME)
XML_PROGRAMLISTING     = NO
EXTRACT_ALL            = YES
INPUT                  = $(INPUT_PATH)
RECURSIVE              = YES
GENERATE_HTML          = NO
GENERATE_LATEX         = NO
GENERATE_XML           = YES
INCLUDE_PATH           = $(INCLUDE_PATH)
CLASS_DIAGRAMS         = NO
```

Referências

[apa12] Apache software foundation http server project. http://httpd.apache.org/, 5 2012.

[axi12a] Anubisnetworks reported issue on axis2/c. https://issues.apache.org/jira/browse/AXIS2C-1282, 5 2012.

[axi12b] Anubisnetworks reported issue on axis2/c (2). https://issues.apache.org/jira/browse/AXIS2C-1283, 5 2012.

[axi12c] Apache axis2 architecture guide. http://axis.apache.org/axis2/c/core/docs/Axis2ArchitectureGuide.html, 4 2012.

[axi12d] Apache axis2/c example hello service. http://axis.apache.org/axis2/c/core/docs/hello/service/hello_svc.c.html, 5 2012.

[axi12e] Apache axis2/c is a web services engine. http://axis.apache.org/axis2/c/core/, 5 2012.

[axi12f] Apache axis2/c quick start guide. http://axis.apache.org/axis2/c/core/docs/axis2c_manual.html#quick_start, 5 2012.

[axi12g] Axis2c doesn't support ipv6. https://issues.apache.org/jira/browse/AXIS2C-1530, 5 2012.

[axi12h] Memory leak in axis2_http_worker_process_request (http_worker.c). https://issues.apache.org/jira/browse/AXIS2C-1586, 5 2012.

[axi12i] Memory leak in axis2_simple_http_svr_conn_read_request() (file core/transport/http/common/simple_http_svr_conn.c). https://issues.apache.org/jira/browse/AXIS2C-1583, 5 2012.

[axi12j] memory leak in code generated by wsdl2c, op not freed in axis2_stub_populate_services_for_myservice. https://issues.apache.org/jira/browse/AXIS2C-1151, 5 2012.

[axi12k] memory leaks in xpath. https://issues.apache.org/jira/browse/AXIS2C-1602, 5 2012.

[boo12] Boost::MPL. http://www.boost.org/doc/libs/1_50_0/libs/utility/enable_if.html, 5 2012.

[Bro96] Mark R. Brown. *FastCGI specification*. Open Market, Inc., 1996.

[DH98] F. Dawson and T. Howes. vCard MIME Directory Profile. RFC 2426 (Proposed Standard), September 1998.

[dox12a] Doxygen. http://www.stack.nl/~dimitri/doxygen, 5 2012.

[dox12b] Doxygen's internals. http://www.stack.nl/~dimitri/doxygen/arch.html, 5 2012.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[GHM+06] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. Soap version 1.2 part 1: Messaging framework. Technical report, 2006.

[Gro91] Object Management Group. Corba component model 1.0 specification. Specification Version 1.0, Object Management Group, August 1991.

[gso12a] gsoap artifact generation for magic squares soap server. http://www.cs.fsu.edu/~engelen/soapdemo.cgi?ex=8, 5 2012.

[gso12b] gsoap building a soap server: Magic squares. http://www.cs.fsu.edu/~engelen/magicserver.html, 5 2012.

[Hen06] Michi Henning. The rise and fall of corba. *Queue*, 4(5):28–34, June 2006.

[Inf12] Applied Informatics. Applied informatics remoting. http://www.appinf.com/en/products/remoting.html, 5 2012.

[iso03] *ISO/IEC 14882:2003: Programming languages: C++*. 2003.

[iso11] *ISO/IEC 14882:2011: Programming languages: C++*. 2011.

[JWHL03] J. Järvi, J. Willcock, H. Hinnant, and A. Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, 21(6):25–32, June 2003. Magazine.

[lig12] lighttpd also known as Lighty. http://www.lighttpd.net/, 5 2012.

[MA04] Scott Meyers and Andrei Alexandrescu. C++ and the Perils of Double-Checked Locking. *Dr. Dobb's Journal*, July 2004.

[Mat12] Wolfram Mathworld. Magic square. http://mathworld.wolfram.com/MagicSquare.html, 5 2012.

[Mey95] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[ngi12] nginx also known as Engine-X. http://nginx.org/, 5 2012.

[oth12] Boost::serialization other serialization implementation analysis. http://www.boost.org/doc/libs/1_51_0/libs/serialization/doc/overview.html#Otherimplementations, 5 2012.

[qtc12a] Qt creator c++ parsing lib. http://www.qt.gitorious.org/qt-creator/qt-creator/trees/master/src/libs/cplusplus, 5 2012.

[qtc12b] Qt creator ide and tools. http://qt.nokia.com/products/developer-tools/, 5 2012.

[Red11] Martin Reddy. *API Design for C++*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[RR07] Leonard Richardson and Sam Ruby. *Restful web services*. O'Reilly Media, Inc., 1 edition, May 2007.

[sta12] Wso2 web services framework for c++. http://wso2.com/products/web-services-framework/cpp/, 5 2012.

[VEG02] Robert A. Van Engelen and Kyle A. Gallivan. The gsoap toolkit for web services and peer-to-peer computing networks. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02, pages 128–, Washington, DC, USA, 2002. IEEE Computer Society.

[VJ02] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[wso12] Open source web services framework for c++ based on axis2/c. http://code.google.com/p/staff/, 5 2012.

[xsl12] Xml stylesheet transformation library. http://xmlsoft.org/XSLT/, 5 2012.