Paulo Renato Branco Dias

Licenciatura em Engenharia Informática

Definição e Execução de Computações Dinâmicas numa Linguagem de Programação Orientada-a-Serviços

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador : Hervé Paulino, Prof. Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente: Ana Moreira

Arguente: Francisco Martins

Vogal: Hervé Paulino

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Junho, 2014

Definição e Execução de Computações Dinâmicas numa Linguagem de Programação Orientada-a-Serviços

Copyright © Paulo Renato Branco Dias, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

À minha família

Agradecimentos

Quero agradecer em primeiro lugar ao meu orientador Professor Hervé Paulino por toda a dedicação, passagem de conhecimento e apoio incansável desde o primeiro dia.

Quero também agradecer a todos os meus colegas e amigos de curso, bem como à minha família e namorada pelo apoio ao longo da elaboração deste trabalho.

Resumo

Os serviços inicialmente idealizados para o mundo dos negócios, têm actualmente um espectro de utilização muito mais lato, facilitando assim a incorporação de software do exterior, sob a representação de serviço, por parte das aplicações. Os principais contribuidores para a emergente utilização de serviços são a proliferação dos dispositivos móveis, a crescente popularidade da computação da nuvem e a ubiquidade da Internet.

Apesar deste estado da arte, a abstracção dos serviços continua, maioritariamente, a ser relegada para a camada do *middleware*. Consequentemente, este confinamento obstem o programador de ter privilégios para interagir com os serviços ao nível da linguagem. A inexistência deste nível de abstracção dificulta o *deployment* de aplicações dinâmicas.

Como medida para tal, o objectivo do nosso trabalho é garantir suporte ao dinamismo e *deployment* de arquitecturas orientadas a serviços. Com esse propósito, vamos endereçar os problemas de incorporação dos serviços acessíveis pela Web e permitir operações de reconfiguração dos mesmos, nomeadamente, a ligação dinâmica, substituição do fornecedor de serviços e a gestão dinâmica de conjuntos de fornecedores de serviços.

Palavras-chave: Computação Orientada a Serviços, Ligação Dinâmica, Reconfiguração Dinâmica, Tempo de Execução

Abstract

The service concept has surpassed the business world to interface distributed resources in general. Applications are progressively incorporating external software modules presented as services

The main contributors for this are the proliferation of mobile devices, the growing popularity of cloud computing and the ubiquity of the Internet.

Despite this state of the art, the service's abstraction continues, mostly, relegated for middleware layer, being subsequently represented in mainstream programming languages as one more language entity.

In order to break this status quo recent work proposes the promotion of the service abstraction from the middleware to the language level. The work of this dissertation falls within one of such proposals, Zib, a service-oriented language for dynamic and scalable systems by composing services published by the application, the runtime system and third-parties (such as Web accessible services).

To this extent, the goal of our work is to provide the runtime support for the construction of such service-oriented architectures. For that purpose, we will address the issues of incorporating Web accessible services and enabling reconfiguration operations, such a dynamic binding, service provider replacement and the dynamic management of pools of service providers.

Keywords: Service-oriented Computing, Dynamic Binding, Dynamic Reconfigurations, Runtime

Conteúdo

1 Introdução **1**

1.1 Motivação 1

1.2 Problema 2

1.3 Proposta 3

1.4 Contribuições 4

1.5 Estrutura do documento 5

2 Estado da Arte **7**

2.1 Arquitecturas e computação orientada a serviços 7

2.1.1 Computação orientada a serviços 7

2.1.2 Arquitecturas orientadas a serviços 8

2.2 Programação orientada a serviços 10

2.2.1 Programação orientada a serviços nas linguagens mais comuns . . 11

2.2.2 Composição de serviços 12

2.2.3 Linguagens de programação orientadas a serviços 15

2.2.4 Arquitecturas Dinâmicas Orientadas a Serviços 21

2.3 Análise Crítica 21

3 O Ecosistema Zib+Elina **23**

3.1 O Modelo de Programação Zib 23

3.1.1 Linguagem de especificação de serviços 24

3.1.2 Linguagem de implementação de serviços 25

3.2 Elina 27

3.3 ZibElina - Execução de computações Zib no *middleware* Elina 30

3.3.1 Compilação de Zib para Elina 31

3.3.2 Construção e Lançamento da Aplicação 36

4 Linguagem de Construção e Lançamento de Aplicações Zib **37**

4.1 Linguagem 37

4.2 Tradutor 39
4.2.1 Exemplo de tradução 40
4.2.2 Implementação 42
4.2.3 Incorporação 43
4.3 Ligação aos Serviços da WEB 43

5 Sistema de Execução **45**
5.1 Operações adicionadas aos serviços Zib 45
5.1.1 Descrição das operações 45
5.1.2 Implementação 46
5.2 Suporte para integração de Serviços WEB 49
5.2.1 Detalhes de implementação 50

6 Exemplos de Aplicação **53**
6.1 Aplicação *Weather Information* 53
6.2 Aplicação *Temperature Center* 54
6.3 Aplicação *Ordered Temperature Center* 57

7 Conclusões **61**
7.1 Trabalho futuro 61
7.1.1 Linguagem de Construção e Lançamento de Aplicações Zib 61
7.1.2 Sistema de execução 62

A Anexos **67**
A.1 Classe *WebServiceGenerator* 67
A.2 Classe *WSImport* 69
A.3 Implementação do serviço *Temperature Center* da aplicação *Temperature Center* 71
A.4 *WeatherWebServiceStub* gerado para o serviço WEB *Weather* da aplicação *Temperature Center* 72
A.5 Implementação do serviço *Temperature Center* da aplicação *Ordered Temperature Center* 73
A.6 Implementação com listagem ordenada do serviço *Temperature Center* da aplicação *Ordered Temperature Center* 75
A.7 Cliente da aplicação *Temperature Center* 77
A.8 Cliente da aplicação *Ordered Temperature Center* 78

Lista de Figuras

1.1 Adicionar suporte para integrar a composição de serviços. 4

2.1 Representação de uma arquitectura orientada a serviços básica 9
2.2 Composição de serviços da Web através de uma orquestração [11] 13
2.3 Composição de serviços da Web através de uma coreografia [11] 14

3.1 Ilustração de uma *Pool* de Serviços 29
3.2 Compilação dos ficheiros Zib 30

4.1 Exemplo de um servidor Web 38
4.2 Sistema ZibElina com linguagem de composição e lançamento de aplicação e respectivo tradutor. 40
4.3 Fases da tradução da linguagem de composição e lançamento de aplicações 42

5.1 Vários clientes a acederem ao mesmo serviço. 47
5.2 Vários clientes a acederem, de nós Elina distintos, ao mesmo serviço através de um *stub*. 48

Lista de Tabelas

2.1 Levantamento de funcionalidades das linguagens presentes na secção 2.2.3 22

3.1 Linguagem de especificação de serviços. 25
3.2 Implementação de serviços. 26

4.1 Linguagem de composição de uma aplicação 39

Listagens

2.1 Especificação de directorias REST em NodeJS 10
2.2 Invocação de um método REST em JavaScript, utilizando a biblioteca JQuery 10
2.3 Interface do *endpoint* de um serviço *JAX-WS* 11
2.4 Classe de implementação do *endpoint* de um serviço *JAX-WS* 12
2.5 Classe que publica o serviço *JAX-WS* na web 12
3.1 Exemplo de especificação de um serviço Zib 25
3.2 Declaração de uma operação síncrona em Zib 26
3.3 Declaração de uma operação assíncrona em Zib 26
3.4 Exemplo de implementação de um serviço Zib 28
3.5 Interface `IService` 29
3.6 Compilação para Java do serviço *KeyValueStore* da Listagem 3.1 31
3.7 Compilação para Java do *provider KeyValueStoreProvider* da Listagem 3.4 . 32
3.8 Exemplo de um cliente de um serviço 34
3.9 Resultado da compilação de um cliente de um serviço 35
4.1 Especificação de uma composição 41
4.2 Composição gerada pelo tradutor 41
4.3 Resultado da tradução da estrutura de uma aplicação sem *bundle* 41
4.4 Resultado da tradução da estrutura de uma aplicação com um *bundle* `B` . . 42
4.5 Interface dos *visitors* do tradutor 42
4.6 Especificação de um serviço Web 43
4.7 Resultado da tradução da especificação de um serviço Web 44
5.1 Serviço Zib com suporte para reconfiguração dinâmica 45
5.2 Interface com operações de reconfiguração para *pools* 49
5.3 Interface com operações de reconfiguração para *pools* 49
6.1 Ficheiro de lançamento da aplicação *Weather Information* 54
6.2 Interface do serviço da Web 54
6.3 Cliente da aplicação *Weather Information* 54
6.4 Serviço responsável por interpretar XML 54
6.5 Ficheiro de lançamento da aplicação *Temperature Center* 55

6.6 Interface do serviço *Temperature Center* 56
6.7 Exemplo de execução da aplicação *Temperature Center* 56
6.9 Interface do serviço *Temperature Center* da aplicação *Ordered Temperature Center* 58
6.8 Ficheiro de lançamento da aplicação *Ordered Temperature Center* 58
6.10 Exemplo de execução da aplicação *Ordered Temperature Center* 58

Introdução

1.1 Motivação

A computação orientada a serviços assume-se cada vez mais como o paradigma de referência para a programação de sistemas distribuídos em ambientes heterogéneos, como a *World Wide Web*. A WWW tem evoluído para uma plataforma de provisionamento de serviços. Tal tem tido um impacto considerável no desenvolvimento de aplicações, que crescentemente incorporam serviços externos, disponibilizados na Web. A crescente utilização de serviços externos por parte das aplicações é um factor que deve ser tomado em conta. Um dos principais contribuidores para este crescimento é a proliferação dos dispositivos móveis, cujas características fazem com que estes sejam particularmente interessantes para a utilização de computação de forma remota. A crescente popularidade da computação na nuvem e a ubiquidade da Internet são também dois factores importantes na origem deste crescimento. Cada vez mais a *WWW* tem vindo a revelar uma inerente capacidade para a prestação de serviços, permitindo que estejamos todos conectados e utilizemos serviços sem custos adicionais. Estes serviços podem, por sua vez, ser implementados fazendo referências entre eles, criando assim um grafo de composição de serviços que se pode espalhar através da Internet.

Apesar deste estado da arte, a abstracção de serviço continua, principalmente, confinada ao nível do *middleware*, sendo posteriormente representada nas linguagens de programação de uso geral como sendo apenas mais uma entidade da linguagem. O caso mais comum é a programação orientada a objectos, onde os serviços são apresentados simplesmente como apenas mais um objecto.

De forma a quebrar este *status quo* surgiram algumas propostas de linguagens de programação [3, 4, 13, 20] com base no conceito de serviço. Este é o contexto em que se

insere a linha de investigação onde se encaixa este trabalho. O objectivo principal dessa linha de investigação é fornecer uma abstracção uniforme para a construção de sistemas concorrentes, dinâmicos e escaláveis através da composição de serviços publicados pela aplicação, pelo sistema de execução ou por terceiros (tais como serviços da Web).

Dada a natureza distribuída e dinâmica dos ambientes de execução que se pretende suportar, a linguagem e o seu sistema de execução, têm de incluir mecanismos que garantam escalabilidade e tolerância a falhas. Para tal, assume vital importância a introdução de mecanismos que permitam a adequação, em tempo de execução, da computação à evolução dinâmica das condições da infraestrutura subjacente e dos requisitos da própria aplicação. Exemplos desse tipo de adequação são a comutação dinâmica entre fornecedores do mesmo serviço, adição de novos fornecedores de serviços e pausar ou retomar a execução de um ou mais fornecedores de serviços.

1.2 Problema

O trabalho desta tese tem como objectivo o suporte ao desenvolvimento de arquitecturas de software distribuídas que permita a composição de serviços de diversas origens, nomeadamente de origem interna, do sistema de execução ou mesmo provenientes da Internet. Além disso, existe o objectivo de garantir a gestão dinâmica deste conjunto de serviços e dos seus respectivos fornecedores.

No que se refere à composição de serviços provenientes de origens diversas, existem alguns desafios, tais como a integração de diferentes tecnologias e uniformização de interfaces de serviços. Para tornar possível a integração de diferentes tecnologias é comum o uso de uma camada de adaptação através de intermediários para abstrair a comunicação do resto do programa, um exemplo desta abordagem é o *JAX-WS* [1]. No que diz respeito à uniformização das interfaces de diferentes serviços, há muito trabalho realizado na área das ontologias.No entanto, neste tópico em particular, o principal objectivo deste trabalho é proporcionar uma estrutura capaz de integrar serviços implementados sobre diferentes tecnologias. A uniformização das interfaces de representação é uma preocupação ortogonal que não será abordada no contexto desta dissertação.

Em relação ao suporte para dinamismo, será um desafio incorporar mecanismos do sistema de execução que permitam a reconfiguração de serviços ao nível da linguagem ou garantir que é possível ajustar conjuntos de fornecedores de serviços de forma dinâmica.

Entrando num contexto mais específico, relacionado com a configuração de serviços, existem vários desafios que devem ser superados, tais como:

Ligação dinâmica É importante oferecermos suporte à ligação dinâmica com o intuito de se permitir a comutação dinâmica entre fornecedores de um dado serviço. Este desafio é também uma mais valia, se superado, no contexto das tolerâncias a falhas.

[1] https://jax-ws.java.net/

Adicionar/Remover um fornecedor de serviços Uma aplicação tem requerimentos que devem ser tidos em conta, como tal, nós queremos dar suporte ao programador para adicionar ou remover fornecedores de serviços sempre que for necessário.

Pausar ou retomar um fornecedor de serviços Para garantir que as alterações efectuadas num serviço durante a sua execução são feitas de forma segura e consistente, é necessário garantir suporte para pausar ou retomar o consumo dos pedidos feitos ao fornecedor do mesmo. Estas operações têm particular utilidade no caso de, por exemplo, se querer substituir um fornecedor de um serviço que está a ser acedido por vários clientes, em que antes de se efectuar a operação de substituição pode-se pausar a execução do serviço de forma a que nenhum cliente saia prejudicado com a operação de substituição.

Depois de um estudo aprofundado encontramos apenas uma arquitectura comparável à que nós queremos implementar, o *ServiceJ* [4].

O *ServiceJ* é uma estrutura que permite a ligação dinâmica de serviços. Sempre que é inicializado um serviço é criada, de forma abstracta, uma fila de serviços com suporte para inserção e remoção dinâmica. Nesta fila de serviços podem estar associados fornecedores de serviços provenientes de diferentes origens e em caso de falha de um destes, existe um suporte para tratamento a falhas em que o fornecedor em falha é automaticamente removido ou substituído de uma forma transparente. Isto permite que uma aplicação em execução consiga manter-se sem que haja percepção da ocorrência de uma falha.

Foram estudadas mais algumas arquitecturas com o intuito de encontrar outras soluções possíveis para o nosso problema, contudo as que foram encontradas não preencheram os requisitos necessários para a solução que nós gostaríamos de adoptar. As restantes arquitecturas que foram encontradas, o *Jolie* [13], o *Abacus* [20] e o *S* [3], também permitem a composição de serviços e invocação a serviços externos, porém estas arquitecturas não facultam suporte para configurar serviços durante a sua execução.

O *S* é um caso especial, pois o seu principal objectivo é compor serviços RESTful provenientes da Web. A identificação e composição destes serviços é simplificada, visto que os serviços da Web são identificados por um *URL*. Esta arquitectura contribuiu para engrandecer a nossa visão no que diz respeito a serviços remotos, principalmente, no que diz respeito à composição de serviços Web e construção dinâmica de respostas aos clientes com base na informação proveniente da composição de diferentes serviços remotos.

1.3 Proposta

Este trabalho assenta sobre uma linguagem de implementação de serviços de nome *Zib*[18] e tem dois grandes objectivos:

Primeiro vamos implementar um tradutor simples para uma linguagem de composição previamente especificada. Esta linguagem permite-nos expressar computações baseadas em serviços dos três tipos previamente mencionados: os implementados em *Zib*, os fornecidos pelo sistema de execução e os serviços provenientes da Web. O tradutor a implementar tem como objectivo a geração de um ficheiro *Zib* onde estão definidos todos os serviços que são requeridos pela aplicação. Como forma de permitir que os serviços especificados em *Zib* possam ser integrados no nosso sistema de execução *Elina*, vamos utilizar um compilador *Zib* cujo desenvolvimento já vem de trabalhos anteriores. Este, para além de traduzir as especificações dos serviços *Zib* também traduz as implementações de serviços (fornecedores) em *Java bytecode* com chamadas para o sistema de execução *Elina*, um *middleware* Java para computação paralela em ambientes distribuídos.

A segunda variante será fornecer suporte para reconfigurações nos serviços da linguagem de programação *Zib*, ou seja, queremos abordar as questões de integração externa de serviços, substituição do fornecedor de serviços, pausar/retomar o consumo de pedidos e permitir a gestão dinâmica de conjuntos de fornecedores de serviços. Para este propósito, teremos de modificar o sistema de execução do *Zib*, construído sobre o *Elina*. Particularmente, teremos que estender o sistema de execução do *Zib*, com o objectivo de apoiar as operações de reconfiguração e integrar os serviços de diferentes origens.

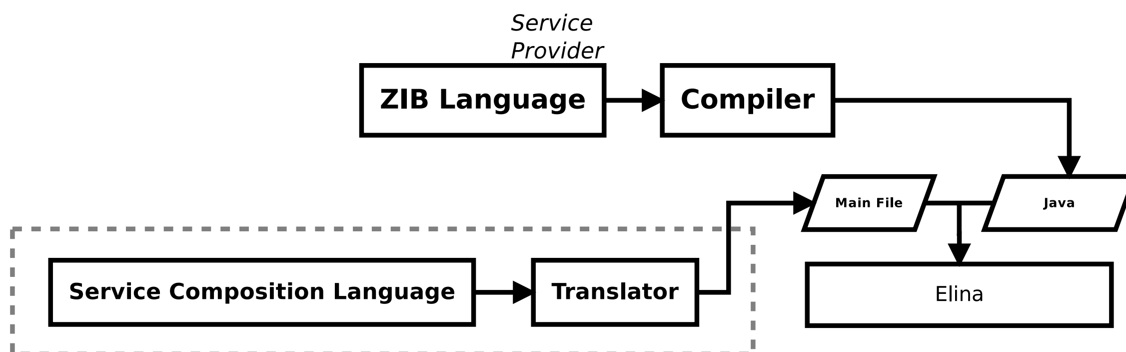

Figura 1.1: Adicionar suporte para integrar a composição de serviços.

1.4 Contribuições

As principais contribuições que queremos atingir são as três seguintes:

1. O tradutor;

2. A introdução das capacidades de reconfigurações do Elina, dando origem a um sistema com suporte para dinamismo ao nível da linguagem;

3. Implementação de um conjunto de pequenas aplicações que ilustram as potencialidades do trabalho desenvolvido.

1.5 Estrutura do documento

O restante deste documento está organizado da seguinte forma:

Capítulo 2 Contém o estado da arte relacionado com a computação e arquitecturas orientadas a serviços, programação orientada a serviços, algumas linguagens de programação orientadas a serviços e finalmente arquitecturas dinâmicas orientadas a serviços.

Capítulo 3 Neste capítulo será feita uma introdução ao Zib e ao seu sistema de execução, Elina. Será também explicado como é que ambos estão interligados.

Capítulo 4 Será feita uma referência à linguagem que permite compor diferentes serviços, como também, ao tradutor que interpreta esta mesma linguagem.

Capítulo 5 Neste capítulo será explicado como foi feita a implementação que dá suporte ao dinamismo, nomeadamente como foram implementadas as operações de reconfiguração de fornecedores de serviços.

Capítulo 6 Será feita uma avaliação do trabalho final através de exemplos de pequenas aplicações que irão demonstrar quais as potencialidades resultantes da implementação desta dissertação.

Capítulo 7 Neste capítulo iram estar presentes todas as conclusões resultantes da elaboração desta dissertação.

Estado da Arte

2.1 Arquitecturas e computação orientada a serviços

O paradigma da computação orientada a serviços [16, 17] consiste num conjunto de conceitos, princípios e métodos que são especificados por uma arquitectura orientada a serviços. As aplicações construídas sobre este paradigma disponibilizam serviços através de interfaces estandardizadas que são independentes dos restantes componentes da aplicação, fornecendo assim de uma forma simplificada um certo nível de modularidade e escalabilidade.

2.1.1 Computação orientada a serviços

Este paradigma utiliza a noção de serviço como base do desenvolvimento de aplicações distribuídas. Uma das mais valias da utilização deste paradigma é a facilidade com que se pode de forma explicita programar sem ter noção da arquitectura ou engenharia de software que está a ser criada ou até já está implementada por detrás da programação. Esta abstracção entre as diferentes camadas de desenvolvimento só é possível devido à divisão que é feita em três grandes áreas distintas:

Fornecedores de serviços Os principais responsáveis por esta área são os programadores que apesar de não necessitarem de ter conhecimento lógico da aplicação, providenciam componentes aplicacionais através de interfaces. A composição de todos estes componentes desenvolvidos pelos programadores é que permite aos engenheiros de software construir aplicações.

Distribuidor de serviços É a entidade responsável por tornar o serviço público e permitir o acesso dos mesmos aos engenheiros de software que desenham uma aplicação.

Construtores de uma aplicação Normalmente esta componente é representada por engenheiros de software ou por profissionais com competências para desenhar toda a lógica de uma aplicação ao mais alto nível, sem nunca terem percepção de qual a implementação existente por detrás de cada implementação.

Existem dois tipos de clientes de serviços, os utilizadores finais ou outros fornecedores de serviços. O último caso deriva cadeias de dependências entre serviços que podem também terminar nos utilizadores finais. Para permitir a comunicação entre estes múltiplos actores algumas restrições devem ser definidas:

Neutro tecnologicamente A invocação de serviços deve seguir padrões para transmitir e manipular toda a informação adjacente necessária. Isto faz com que seja necessário utilizar protocolos de comunicação, descrição de serviços e mecanismos de pesquisa de forma a estabelecer um padrão de comunicação com os serviços. Estas restrições facilitam a publicação de serviços por parte das entidades que utilizam a Internet como meio de exportação de serviços.

Acoplamento fraco O cliente deve estar desagregado do serviço o mais possível de forma a que entre ambos não haja noção do comportamento e estruturas internas de cada um.

Localização transparente Os serviços devem estar disponíveis para uma variedade significante de clientes, que independentemente das suas localizações devem conseguir sempre invocar os serviços que desejam. Para tal, é necessário que os serviços registem a sua localização em repositório públicos e acessíveis pelos utilizadores interessados.

2.1.2 Arquitecturas orientadas a serviços

Uma arquitectura orientada a serviços [15] é uma arquitectura de software distribuída que é construída com base num conjunto de serviços que cooperam entre si com o objectivo de criar uma aplicação.

Uma arquitectura orientada a serviços básica é a forma mais simples de construir uma infraestrutura que permita que um determinado conjunto de serviços seja acessível através de interfaces padrão e protocolos de comunicação. Nesta arquitectura existem três entidades principais:

Agência de descoberta de serviços Entidade responsável por publicar a descrição de um serviço e garantir que esta é detectável.

Fornecedor de um serviço Define a descrição de um serviço e publica-a para um cliente ou para a agência de descoberta de serviços.

Solicitante de um serviço Utiliza a operação *find* para obter a descrição de um serviço da agência de descoberta de serviços. Após obter a descrição pretendida, utiliza-a para efectuar a ligação com o fornecedor do serviço em questão para poder invocar o que desejar do mesmo.

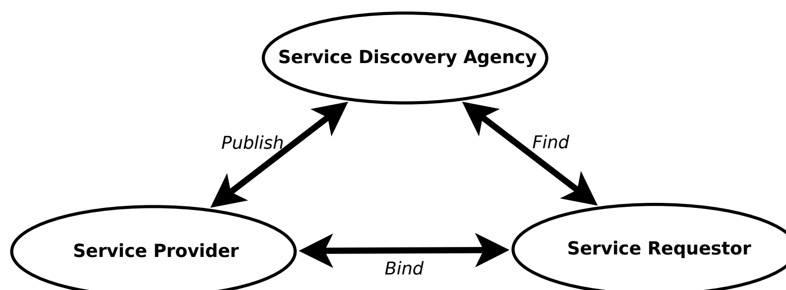

Figura 2.1: Representação de uma arquitectura orientada a serviços básica

2.1.2.1 Serviços da Web

Arquitecturas como a que está na ilustrada na figura 2.1 não são muito comuns nos dias de hoje, pois é difícil gerir um repositório dinâmico e acessível a partir da Internet nos dias de hoje. O mais comum é estabelecer quais os serviços a utilizar na fase de desenho da arquitectura a adoptar, evitando assim, o processo de procura de um serviço em repositórios. O *REST* e o *SOAP* são os protocolos de comunicação mais conhecidos a quando da utilização de serviços e ambos permitem descrever a estrutura de um serviço e as operações da sua respectiva interface. A principal diferença entre este dois protocolos é que o *REST* tem um maior foco na estruturação da informação, representando-a normalmente em *XML* ou *JSON*, enquanto que o *SOAP* permite invocações remotas através da tecnologia *RPC*.

Existem serviços que comunicam entre si com o intuito de fornecer informação combinada e os serviços podem ser combinados de diferentes formas. Um conjunto de serviços pode ser atingível em apenas um computador, como também numa rede local ou até mesmo através da Internet.

2.1.2.2 REST

Lado do servidor A crescente utilização da computação móvel faz com que cada vez mais exista necessidade de criar aplicações em que o servidor possa servir mais do que um tipo de plataforma, como o *REST* é baseado na estruturação da informação em si e não nas tecnologias ou linguagens a serem utilizadas, cada vez mais tem vindo a ser utilizado como um dos suportes para o lado do servidor.

A tendência é para que as ferramentas de trabalho continuem a aumentar o nível a que se programa e que cada vez existam mais bibliotecas e extensões nas ferramentas de trabalho utilizadas e um exemplo disso é o bocado de código em NodeJS 2.1 para definir as directorias de uma interface *REST* e as suas respectivas acções. Como é possível verificar neste último exemplo referenciado o *REST* tem um conjunto de métodos que são suportados e, para além do *GET* e do *POST* também existe o *PUT* e o *DELETE*. Se o servidor for implementado seguindo as normas do protocolo cada um destes métodos deverá desencadear diferentes acções. O *GET*, por exemplo, é utilizado para retornar informação ao cliente, já o *POST* serve para criar novas entradas no lado do servidor. O *DELETE* tal como o nome indica deverá pagar a entidade recebida pela servidor e por fim o *PUT* deverá substituir o conteúdo de uma entidade por outro.

```
1 app.get('/api/users', user.list);
2 app.post('/api/users/register', user.checkLoginParams, user.register);
3 app.post('/api/users/login', user.checkLoginParams, user.login);
4 app.post('/api/users/logout', user.logout);
5 app.get('/api/users/:id', user.get);
6 app.post('/api/users/:id', user.list);
```

Listagem 2.1: Especificação de directorias REST em NodeJS

Lado do cliente Para um cliente lidar com este protocolo não necessita de grandes requisitos, pois é um protocolo bastante leve e ao cliente basta-lhe apenas ter conhecimentos das directorias para efectuar os pedidos remotos e após um retorno positivo fazer o tratamento da informação da forma pretendida.

A maior parte das linguagens do lado cliente hoje em dia também já têm mecanismos que facilitam a invocação de operações no servidor utilizando este protocolo, como exemplo disso temos o bocado de código em JavaScript 2.2.

```
1 $.ajax({
2   type: "GET",
3   dataType: "jsonp",
4   url: "http://someserverurl:8080/api/users",
5   success: function(users){
6       alert(users);
7   }
8 });
```

Listagem 2.2: Invocação de um método REST em JavaScript, utilizando a biblioteca JQuery

O cliente pode optar por efectuar uma invocação síncrona ou assíncrona, a não ser que esteja limitado de alguma forma pela linguagem que esteja a utilizar.

2.2 Programação orientada a serviços

As linguagens de programação mais comuns da actualidade normalmente não têm noção de serviço como sendo mais uma entidade. Isto faz com que os programadores tenham de especificar que protocolos querem utilizar para atingir determinado serviço, sendo o mais utilizado e conhecido o *HTTP*.

Outro cenário possível é a utilização de mecanismos que representem o serviço como sendo mais um objecto e disponibilizem ao cliente uma referência para o mesmo. Neste caso em específico, normalmente é criado um *stub* por onde o cliente pode efectuar invocações remotas. Um exemplo deste último caso é o *JAX-WS* que é uma extensão para dar suporte a serviços remotos em Java.

2.2.1 Programação orientada a serviços nas linguagens mais comuns

Lado do servidor Nesta secção será mostrado como criar um serviço remoto simples, utilizando o já referido *JAX-WS*. O ponto de partida na implementação de um serviço utilizando esta ferramenta é perceber que anotações existem e quais as suas finalidades.

A primeira anotação a ser especificada deverá ser a *@WebService*, que define qual vai ser a classe que vai ser exposta como sendo o *web service endpoint*, cuja finalidade é declarar quais os métodos que os clientes podem invocar no serviço. A implementação destes métodos pode ser feita numa única classe e nesse caso a especificação dos métodos a serem invocados é declarada apenas nessa classe ou pode ser criada uma interface com a especificação dos métodos e respectiva implementação noutra classe. Um exemplo deste último caso está representado nos seguintes pedaços de código, 2.3 e 2.4.

Para todos os métodos que se deseje que sejam exposto publicamente aos clientes deve ser adicionada a anotação *@WebMethod*

Por fim, como é demonstrado no código 2.5, basta criar uma classe responsável por definir qual a localização em que o serviço irá executar, mapear os métodos e publicar o serviço.

No caso de ser adicionado o parâmetro *?wsdl* ao URL de acesso ao serviço, o *JAX-WS* tem a particularidade de retornar um documento em XML com toda a descrição do serviço em questão.

```java
1  package com.thesis.webservice;
2  import javax.jws.WebMethod;
3  import javax.jws.WebService;
4
5  @WebService
6  public interface HelloThesis {
7    @WebMethod public String sayHello(String name);
8  }
```

Listagem 2.3: Interface do *endpoint* de um serviço *JAX-WS*

```
package com.thesis.webservice;
import javax.jws.WebService;

@WebService(endpointInterface="com.thesis.webservice.HelloThesis")
public class HelloThesisImpl implements HelloThesis{
  public String sayHello(String name) {
    return "HelloThesis:" + name;
  }
}
```

Listagem 2.4: Classe de implementação do *endpoint* de um serviço *JAX-WS*

```

package com.thesis.webservice;
import javax.xml.ws.Endpoint;

public class HelloThesisWebService {
  public static void main(String[] args) {
    Endpoint.publish(
      "http://localhost:3000/WS/HelloThesis",
      new HelloThesisImpl()
    );
  }
}
```

Listagem 2.5: Classe que publica o serviço *JAX-WS* na web

Lado do cliente Se o lado do servidor já estiver implementado é bastante fácil implementar a parte do cliente que irá fazer as invocações remotas ao serviço, pois no *SOAP* um serviço tem sempre especificada uma descrição das suas funcionalidades e o cliente pode gerar o seu código em função dessa descrição. Uma ferramenta que já vem incluída no *JDK*, normalmente utilizada para a geração do código do cliente com base no *WSDL*, é o *wsimport*. O cliente também pode ser todo implementado sem ser utilizada qualquer ferramenta de geração de código, existindo também anotações para o lado cliente que devem ser tomadas em consideração. Após concluída a implementação do código responsável pelas ligações ao serviço basta implementar qual o comportamento associado a cada invocação.

2.2.2 Composição de serviços

A composição de serviços permite agregar serviços de forma a que estes sejam representadas como apenas um serviços combinado. Existem duas formas de criar serviços compostos, através de orquestrações ou coreografias.

Uma orquestração tem como referência um executável que especifica acções de processos de negócio com serviços. Numa orquestração existe uma única entidade responsável pelo controlo das acções entre os serviços, o orquestrador.

O *BPEL* [11, 2] é uma linguagem bastante comum para especificar uma orquestração de vários serviços. Esta linguagem permite desenhar uma arquitectura orientada a serviços, através da composição, orquestração e coordenação entre vários serviços. Uma composição de serviços é conhecida como um processo de negócio de serviços. É nesta composição que pode ser especificada qual a ordem de invocação de métodos a cada serviço composto, podendo as invocações serem sequenciais ou em simultâneo. Com esta linguagem também é suportada a criação de ciclos, declaração de variáveis, copiar e atribuir valores e por exemplo ter tratamento para falhas. A combinação de tudo isto faz com que seja possível, do ponto de vista da algoritmia, definir processos de negócios mais complexos. Além de tudo isto este caso de estudo tem uma acrescento de valor por também permitir que os serviços possam estabelecer ligações de forma dinâmica e durante o tempo de execução.

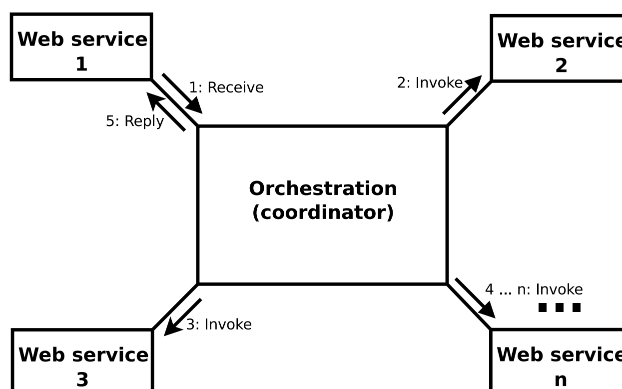

Figura 2.2: Composição de serviços da Web através de uma orquestração [11]

No caso da coreografia, a composição é realizada de uma forma mais colaborativa e permite que cada participante envolvido possa definir qual a sua própria interacção. Este tipo de composição de serviços acompanha as sequências de mensagens entre os vários serviços e fontes sem existir um único ponto de controlo, cada participante tem uma linha de execução de acordo com o comportamento dos restantes participantes. O objectivo é criar uma perspectiva global que permita definir as trocas de mensagens que ocorrem entre os serviços e os demais envolvidos. A linguagem mais comum para definir coreografias entre serviços é o *WS-CDL* [7, 1, 2]. O *WS-CDL* é, um pouco ao estilo do *BPEL*, uma linguagem baseada em XML que define, de um ponto de vista global, qual o comportamento observável dos seus colaboradores. Os objectivos da especificação do *WS-CDL* [7] servem para permitir:

Reutilização, garantido que diferentes participantes de contextos distintos possam utilizar a mesma definição coreografia;

Cooperação, fazendo com que as coreografias definam a sequência de troca de mensagens entre os seus colaboradores;

Semântica, incluindo nas coreografias semânticas e documentação para todos os componentes;

Composição, através da combinação de coreografias existentes com novas coreografias que por sua vez podem ser utilizadas em contextos existentes;

Modularidade, as coreografias poderão ser criadas através de entidades presentes noutras coreografias;

Alinhamento de informação, onde os participantes, intervenientes numa coreografia, podem comunicar e sincronizar a sua informação observável;

Tratamento de excepções, em que os participantes tratam eventos excepcionais ou pouco usuais no decorrer da coreografia;

Transacionalidade em que os participantes de uma coreografia tem a capacidade de coordenar o resultado final de colaborações a longo prazo.

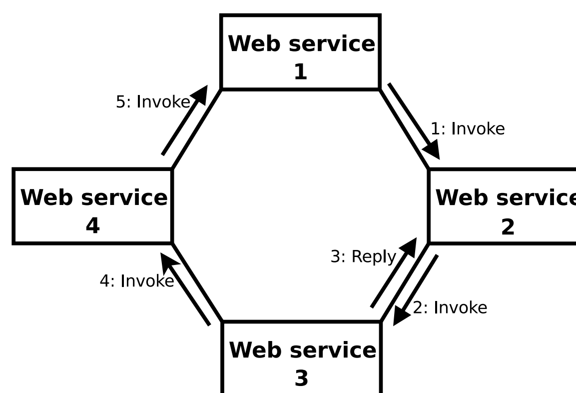

Figura 2.3: Composição de serviços da Web através de uma coreografia [11]

2.2.2.1 Agregação de coreografias e orquestrações

A linguagem WS-CDL tem sido criticada pelo suporte limitado para certos tipos de utilização, separação limitada entre o meta-modelo e sintaxe, e a falta de base formal. Este facto for tomado como motivação para a criação dum nova linguagem coreográfica. Além disso, não é claro se todos os conceitos podem ser mapeados à linguagem BPEL [12].

Uma coreografia captura o processo colaborativo envolvendo múltiplos serviços e especialmente as suas interacções dum ponto de vista global. Por outro lado, as orquestrações lidam com a descrição das interacções entre um conjunto de serviços [1].

Existem dois cenários aplicacionais que utilizam ambos os tipos de composição de serviços (coreografia e orquestração):

- No primeiro cenário as entidades de negócio podem acordar numa coreografia específica, tal como definido no WS-CDL, de forma a conseguir um objectivo comum.

Esta coreografia baseada em WS-CDL é então utilizada para gerar *stubs* do processo do BPEL para cada participante. Neste caso, a coreografia WS-CDL pode ser considerada como um contrato global em que todas as partes devem acordar.

- No segundo cenário uma entidade de negócio pretende publicar a interface dos seus processos disponibilizando-a aos seus parceiros de negócio. Neste cenário, deverá ser gerada uma descrição coreográfica do processo interno. De forma a endereçar este tipo de cenários, foi proposta uma especificação que combina o uso da linguagem WS-CDL com WS-BPEL (*Web Sevice Business Process Execution*)[12].

2.2.3 Linguagens de programação orientadas a serviços

Programação orientada aos serviços ou *SOP* [21] foi uma linguagem proposta para dar suporte á reutilização facilitando o desenvolvimento de sistemas distribuídos. A programação orientada aos serviços expande os conceito do desenvolvimento orientado aos objectos e componentes, para os sistemas de computação de distribuída.

A programação orientada aos objectos ou *OOP* [21] tem-se demonstrado interessante, tendo contribuído de forma notória para a comunidade de computação e para a indústria. Por outro lado, tal paradigma (*OOP*) tem associado um grande nível de abstracção, surgindo assim a necessidade da criação dum paradigma com suporte para algo mais concreto.

A programação orientada aos serviços surgiu como uma forma de implementar serviços utilizando técnicas orientadas aos objectos melhorando a mesma através da incorporação de primitivas mais concretas e operacionais disponibilizadas aos programadores. O uso de *SOP* apresenta diversas vantagens quando comparado com *OOP* [9] nomeadamente na separação de cuidados, grande granularidade nas unidades de processamento, baixo acoplamento e mensagens descritivas. Ou seja, os princípios da programação orientada aos serviços são considerados especializações dos princípios associados ao paradigma orientado aos objectos. Por outras palavras, os princípios da programação orientada aos serviços podem ser determinados pelos seguintes conceitos:

- Cada sistema é composto por um conjunto de serviços e trocas de pedidos;
- Os serviços são implementados e fornecidos por um conjunto de provedores de serviços;
- Os serviços são geridos por repositórios de serviços que mantém uma base de todos os serviços registados;
- Os consumidores de serviços podem invocar serviços aos *proxies* dos fornecedores de serviços;
- Os *proxies* podem encontrar e criar ligações com os fornecedores de serviços;
- Os pedidos devem incorporar todos os dados a serem processados pelos serviços;

- Os consumidores de serviços devem fornecer aos utilizadores ou clientes os serviços;
- Os serviços são utilizados para dividir grandes aplicações em pequenos módulos discretos;
- Os serviços são integrados via mecanismos de composição de serviços de forma a criar aplicações mais complexas.

2.2.3.1 ServiceJ

O ServiceJ [4] é uma extensão do Java com suporte integrado á programação de serviços remotos. O ServiceJ melhora outras soluções orientadas a objectos de duas formas. Primeiro, lida com questões de volatilidade do serviço através do suporte para associação diferida, restrições de QoS, e selecção optimizada de serviços. Segundo, lida com questões de distribuição fornecendo suporte transparente a falhas dos serviços através do uso de construções que permitem a criação de sessões *ad hoc* para serviços remotos. O ServiceJ tem como finalidade a integração de serviços remotos utilizando o modelo de programação Java. Para esse fim, o ServiceJ introduz quantificadores de tipo que permitem a recuperação de falhas transparente, através da introdução de construções declarativas ao nível da linguagem permitindo assim uma selecção fina e dinâmica de serviços, e pelo fornecimento de blocos de sessão que demarcam as transacções *ad hoc* entre serviços remotos.

O ServiceJ retira a necessidade de dependência com a localização efectiva do serviço (determinada pelo URL do mesmo). Em vez de apontar directamente para a localização do serviço, as variáveis do ServiceJ referenciam um conjunto de serviços. Existem duas diferenças essenciais entre as variáveis de conjunto do ServiceJ e as variáveis normais do Java. Visto que uma variável Java está directamente associada a uma instância do serviço, perante a ocorrência de uma falha a variável Java não será capaz de contactar as instâncias replicadas. Por outro lado, a variável de conjunto do ServiceJ, aponta para um conjunto de serviços intermutáveis permitindo assim que o sistema em execução altere de forma dinâmica o serviço em uso.

As variáveis de conjunto são declaradas com um tipo quantificado que indica que a execução do sistema deve injectar de forma não determinística uma referência encontrada nessa mesma variável contendo um conjunto de referências específicas. Uma variável de conjunto está associado a um conjunto de serviços mapeados para um único serviço abstracto, desta forma os programadores poderão tratar um conjunto de serviços através de um *proxy* representando um serviço virtual. O sistema de execução inicializa o serviço virtual injectando um membro do conjunto na variável de conjunto. O serviço injectado chama-se serviço activo. Outra propriedade interessante do com junto de serviços é a partilha dos mesmos entre aplicações, sendo considerado um avanço em relação á aproximação, baseada em anotações, utilizada no Java, que dissemina os URLs do serviço por várias aplicações definidas na arquitectura orientada a serviços.

O sistema de execução do ServiceJ permite associação diferida de serviços através da associação da variável de conjunto a um serviço do grupo correspondente permitindo assim uma recuperação de falhas transparente pela selecção e injecção dum novo membro no conjunto de serviços.

De forma a restringir o conjunto de serviços, o ServiceJ delega a responsabilidade ao sistema de execução reforçando as restrições de associação dos membros. Ao nível do código fonte, os programadores especificam uma condição de associação booleana com uma operação declarativa. O sistema de execução cria uma vista selectiva do conjunto de serviços partilhado através da selecção dum candidato que satisfaça a restrição.

É ainda possível aos programadores especificar atributos de qualidade. Perante a evocação dum método, o sistema de execução acede ao conjunto de serviços injectando o membro que mais aproxima os atributos de qualidade da variável de sequência.

Uma sequência de chamadas a métodos é chamada de sessão, e por questões de consistência, é crucial que a variável de conjunto de serviços esteja associada ao mesmo serviço durante toda a sessão. Aquando a entrada num bloco de sessão, o sistema de execução injecta um serviço na variável de conjunto. A partir desse momento, a variável de conjunto de serviços fica bloqueada até que a sessão termine com sucesso, ou até que o serviço fique inacessível. Caso contrário, o sistema de execução aborta a sessão corrente, desbloqueando a variável de conjunto de serviços, injectando outro membro nessa variável, e reiniciando toda a sessão. Desta forma, através da introdução destes bloqueios com escopo orientado à sessão sobre as variáveis de conjunto, permite que os programadores assumam as sessões como atómicas e tolerantes a falhas nas interacções com serviços remotos.

2.2.3.2 Jolie

O Jolie [13] é o resultado duma tentativa de obter um denominador comum que ofereça coerentemente as principais funcionalidades da computação orientada aos serviços ou *SOC* integradas com as tecnologias existentes. Por outras palavras, o objectivo é oferecer uma linguagem de programação para definir serviços base, a sua organização na arquitectura, e o comportamento dos orquestradores responsáveis pela supervisão de interacções entre os serviços, possivelmente utilizando diferentes tecnologias de comunicação. É esperado que o Jolie seja a primeira linguagem que responde de forma positiva aos problemas de heterogeneidade tanto para as tecnologias de comunicação de serviços como para aspectos de composição.

Um programa Jolie define um serviço e uma composição de duas partes, chamado comportamento e desenvolvimento. A parte comportamental define as acções que deverão ser levadas a cabo pelo serviço, tais como computações internas ou comunicações de input/output. Esta parte abstrai como as comunicações irão ser suportadas. A parte de desenvolvimento complementa a parte comportamental introduzindo a informação necessária para estabelecer a ligação entre dois serviços. Também poderá ser utilizada

para a definição da estrutura de uma arquitectura orientada a serviços.

De forma a gerir os dados, o Jolie suporta tipos de dados básicos tais como inteiros, strings ou booleanos. De forma geral, as variáveis e expressões podem ser utilizadas para suportar árvores de dados utilizando sintaxe poderosa e concisa. As variáveis são alocadas dinamicamente em tempo de execução.

A comunicação entre serviços é definida pelas portas de comunicação. Existem dois tipos de portas. Portas de entrada que lidam com a exposição de operações de input a outros serviços. E portas de output, que definem como invocar operações de outros serviços. Ambos são conceitos duais e por isso a sua sintaxe é semelhante. As portas são baseadas em três conceitos fundamentais de localização, protocolo e interface. A localização exprime o meio de comunicação, em conjunto com os parâmetros de configuração. Este meio de comunicação é utilizado pelos serviços para exporem a sua interface ou para contactarem outro serviço. Um protocolo define como os dados para envio ou recepção deverão ser formatados, nomeadamente o formato de codificação ou descodificação. Finalmente, uma porta deverá especificar a interface acessível através da mesma. Actualmente, o Jolie suporta quatro tipos de comunicações: *Bluetooth L2CAP* , *Unix local sockets*, Java RMI e *sockets* TCP/IP. Os protocolos são referidos pelo nome e a implementação do protocolo HTTP pode detectar dinamicamente invocações do cliente utilizando formatos diferentes.

O Jolie permite que as portas de saída possam ser dinamicamente associadas, ou seja, que as suas localizações e protocolos possam mudar em tempo de execução. A associação dinâmica é obtida tratando as portas de saída como variáveis.

Tendo várias instâncias de uma execução comportamental num serviço, introduz o problema do encaminhamento das mensagens de chegada para as instâncias correspondentes. Em *frameworks* de aplicações Web este problema é abordado através da utilização de um identificador de sessão único armazenado num *cookie* do *browser*.

O Jolie suporta o encaminhamento de mensagens de chegada para instâncias comportamentais através dos identificadores de sessão. Em vez de fazer uso duma única variável para identificar instâncias comportamentais, os identificadores de sessão permitem que o programador refira uma combinação de várias variáveis.

A gestão de falhas no Jolie envolve quatro conceitos base: escopo, falha, terminação e compensação. Uma falha é um sinal levantado por um comportamento, dentro do escopo quando é obtido um estado de erro, para que seja feita a recuperação de tal erro. A terminação é um mecanismo utilizado para recuperar de erros sendo automaticamente chamado quando o escopo é terminado de forma inesperada por um comportamento colateral permitindo assim uma paragem de forma não abrupta. O escopo termina com sucesso se não levantar qualquer sinal e é obtido através da gestão de todas as falhas lançadas por um comportamento interno. Os mecanismos de recuperação são implementados com base no tratamento de erros que contêm código a ser executado perante a ocorrência de alguma falha.

O Jolie suporta dois tipos de arquitectura de composição. A primeira permite que

um serviço execute outros serviços no mesmo motor de execução, de forma a ganhar vantagens em termos de controlo de recursos. O segundo tipo lida com a topologia das conexões entre os serviços e uma arquitectura orientada ao serviço. Os dois representantes são respectivamente, a incorporação e a agregação.

A incorporação é um mecanismo para executar múltiplos serviços na mesma máquina virtual. Um serviço chamado incorporador, pode incorporar outro serviço, chamado serviço embebido. Actualmente o Jolie suporta a incorporação de definições de serviços em Java e JavaScript sendo que este suporte pode ser estendido de forma modular. A incorporação pode especificar opcionalmente uma porta de saída e neste caso, assim que o serviço for carregado, a porta de saída é associada à porta de entrada de comunicação local do serviço incorporado. A hierarquia entre o incorporado e incorporador é também útil para melhorar performance pois os serviços na mesma máquina virtual podem comunicar utilizando canais de comunicação local para comunicarem a baixa latência. A incorporação dinâmica pode ser utilizada para implementar novas funcionalidades tais como a mobilidade e adaptação dos serviços.

A agregação é uma generalização dos *proxies* de rede que permitem que um serviço exponha operações sem as implementar na sua lógica, em vez disso delega essas operações a outros serviços. A agregação também pode ser utilizada para programar vários padrões arquitecturais, tais como avaliadores de carga, *proxies* invertidos, e adaptadores.

2.2.3.3 S

O S [3] é uma extensão da linguagem de programação JavaScript, cuja finalidade é o desenvolvimento e composição de serviços Web *RESTFul*.

O S permite também expor a utilização de múltiplos serviços externos como um serviço composto. Os resultados da informação processada destes diferentes serviços externos pode ser ser adicionada dinamicamente à resposta de um cliente. Como forma de acelerar a composição dos serviços o S pratica um modelo de paralelismo sem restrições de ordem, tratando das operações de entrada e saída em paralelo.

A linguagem S é compilada para ficheiros JavaScript. Visto que é utilizado um sistema de execução assíncrono e orientado a eventos, o compilador contém mecanismos de dessincronização de forma a permitir que o sistema de execução possa praticar invocações, tanto assíncronas como síncronas.

O sistema de execução é baseado na maquina virtual Google JavaScript V8 e no Node.js e permite que de forma transparente e eficiente a execução dos serviços Web RESTFul possa ser paralelizada ao longo de diferentes núcleos de processadores. Estas características mantêm-se mesmo que o sistema de execução tenha de suportar milhares de clientes de forma concorrente. Tudo isto é possível devido ao suporte adicional para distribuição de carga, controlo de processos e comunicação entre processos.

Nesta linguagem houve necessidade de atribuir processos independentes para diferentes cenários que serão detalhados em seguida.

Encaminhar pedidos Existe um processo dedicado que é utilizado pelo sistema de execução para permitir que diferentes entidades responsáveis pelo tratamento de pedidos possam estar associados ao mesmo porto. Este, é também responsável por abrir o porto para as conexões TCP do serviço e por aceitar pedidos HTTP provenientes de clientes externos.

Tabela de encaminhamentos Com a finalidade de se tratar dos recursos aninhados de um serviço, houve a necessidade de se criar um processo responsável por gerir a tabela de encaminhamentos dinamicamente sempre que sejam inseridos novos recursos.

Estados partilhados No processo dedicado para este cenário foi feita uma aproximação ao processo utilizado para o cenário anterior. Visto que cada estado está sujeito a pedidos simultâneos, a consistência destes é garantida através de um processo independente que gere estados autónomos. Os recursos aninhados são também geridos como processos independentes, seguindo os procedimentos normais de qualquer tratador de pedidos, à excepção de que neste cenário os recursos são registados e apagados de forma dinâmica da tabela de encaminhamentos.

2.2.3.4 Abacus

O Abacus [20] é uma linguagem de programação orientada a serviços para aplicações *grid* que abstrai o serviço através da construção da linguagem e permite aos programadores que se foquem na lógica da aplicação, ignorante portanto os detalhes de baixo nível. Esta linguagem tem como objectivos oferecer uma forma mais efectiva de desenvolver aplicações robustas em *grid* e reduzir os custos de desenvolvimento.

O Abacus permite que uma *grid* seja visível como um supercomputador virtual. Cada serviço em *grid* tens dois tipos de operações primitivas, escrever e executar. A operação de escrever significa lançar um serviço enquanto que a operação executar significa invocar um serviço.

O Abacus usa um modelo de endereçamento em *grid* que é dividido em endereços de dois níveis: um para lançamento de serviço e outro para localização. O espaço físico do serviço é representado por um URL enquanto que o espaço virtual é representado por um identificador único. Isto permite que uma aplicação não perca a referência dos seus serviços enquanto estes são movidos entre diferentes localizações.

O compilador e o sistema de execução do Abacus oferece suporte para virtualização e ligações dinâmicas de serviços. Durante o tempo de compilação, o compilador do Abacus atribui um identificador único a cada serviço declarado. Durante o tempo de execução, o sistema de execução do Abacus atribui um espaço físico ao serviço a ser lançado e depois efectua ligações entre o espaço virtual e o correspondente espaço físico. Isto permite que outros programas possam aceder ao endereço físico através dos seus endereços virtuais.

O sistema de execução do Abacus foi desenvolvido com base no Vega Grid Operating System, que é um sistema *grid*. As principais funcionalidades utilizadas pelo Abacus

deste sistema foram traduzir os endereços virtuais em endereços físicos e a gestão do ciclo de vida de um serviço. Este sistema *grid* também oferece suporte para lançar serviços em tempo de execução.

2.2.4 Arquitecturas Dinâmicas Orientadas a Serviços

A crescente utilização da programação orientada a serviços faz com que surjam novas necessidades nas arquitecturas orientadas a serviços dinâmicas. Uma das principais necessidades que tem vindo a emergir é a possibilidade para compor serviços que se liguem dinamicamente [5, 6].

Para aplicar dinamismo a uma arquitectura orientada a serviços é necessário definir restrições e ter garantias de qualidade dos serviços a utilizar. A funcionalidade mais relevante a verificar neste tipo de arquitecturas é que um serviço possa receber uma implementação de forma dinâmica. As ligações dinâmicas expandem as opções disponíveis, o cliente pode escolher que serviço utilizar e o fornecedor de serviços pode actualizar um serviço com uma nova funcionalidade durante o tempo de execução. Como forma de oferecer este tipo de ligações dinâmicas também é importante que o sistema de execução contenha mecanismos de descoberta de serviços, para tal, é necessário especificar os serviços de uma forma mais abstracta de forma a facilitar a procura entre o serviço procurado e os serviços existentes. A procura de serviços pode ser facilitada através da utilização de ontologias mas no nosso caso o esperado é que o sistema de execução seja o principal responsável por isso. No nosso caso só deverá ser tirado partido da abstracção da especificação dos serviços para facilitar na mudança dinâmica de um fornecedor de serviço.

Para além das ligações dinâmicas, existem dois tipos de integrações de serviços dinâmicos, *on-demand*, onde o programador expressa explicitamente que deseja integrar um novo serviço ou, automático, onde o sistema assume que é necessário integrar um novo serviço. No âmbito desta dissertação não será suposto efectuarmos referência à integração automática de serviços, sendo o principal foco em torno das necessidade do programador.

2.3 Análise Crítica

Na secção anterior foi feito um levantamento de tecnologias com propriedades relevantes para a solução a que nos propomos neste trabalho. Com base nas linguagens de programação orientadas a serviços apresentadas na secção 2.2.3 foi feito um levantamento de características como forma de comparação entre estas e a nossa proposta.

Começando por dar foco às origens dos fornecedores de serviços que cada linguagem (listadas na tabela 2.1) permite integrar, apenas o *Jolie* permite integrar serviços de origem local, do sistema de execução e remota. Já o *Abacus* e o *ServiceJ* permitem apenas a integração de serviços de origem local e do sistema de execução. Por outro lado a linguagem

S tem apenas suporte para integração de serviços remotos.

Apesar do *Jolie* contemplar todas as origens de serviços a que nos propomos neste trabalho, a nível de reconfigurações (o segundo grande foco do nosso trabalho), permite apenas a substituição de fornecedores de serviços, tal como o *Abacus*. Por outro lado, o *ServiceJ* suporta apenas reconfiguração de filas de fornecedores de serviços dinâmicas e o *S* não tem suporte para qualquer tipo de reconfigurações.

		S	**Abacus**	**ServiceJ**	**Jolie**
Integração de serviços	Locais	Não	Sim	Sim	Sim
	Sistema de execução	Não	Sim	Sim	Sim
	Remotos	Sim	Não	Não	Sim
Reconfigurações	Pausar/Retomar execução	Não	Não	Não	Não
	Filas de providers dinâmicas	Não	Não	Sim	Não
	Substituição de provider	Não	Sim	Não	Sim

Tabela 2.1: Levantamento de funcionalidades das linguagens presentes na secção 2.2.3

O Ecosistema Zib+Elina

O modelo Zib apresenta-se como um modelo de programação orientado a serviços, em que a computação é definida através da implementação e composição de serviços. O modelo foi concretizado como uma extensão à linguagem de programação Java, tendo o seu sistema de execução por base o *middleware* Elina. Neste capítulo iremos detalhar ambos.

3.1 O Modelo de Programação Zib

O *Zib* [18] é um modelo de programação concorrente de alto nível, onde as aplicações são construídas através da composição de fornecedores de serviços. Este modelo de programação permite o decopolar entre a especificação e a implementação da computação. A ligação aos serviços segue um mecanismo orientado ao contracto, que tem por base a interface do serviço. Como consequência, os fornecedores de serviços não são endereçáveis ao nível da linguagem. Esta desagregação permite o tratamento uniforme de:

Recursos remotos e locais Um serviço pode abstrair um serviço externo acessível através da Web ou até mesmo um módulo de software que tenha sido lançado pela própria aplicação.

Serviços ao nível aplicacional ou do sistema Os serviços podem também ser utilizados para abstrair funcionalidades provenientes de camadas inferiores, permitindo assim uma interacção com, por exemplo, o sistema de execução. O sistema de execução pode controlar o nível de interferência que pretende permitir às camadas superiores, ajustando o número e natureza de serviços que deseja publicar.

Os fornecedores de serviços podem ter noção de estado ou não. Os serviços com estado preservam-no mesmo que hajam interacções com diferentes clientes, sendo que a linguagem não oferece qualquer tipo se suporte para persistência de tal estado. Os serviços também podem ter noção de sessão que pode ser partilhada por múltiplos clientes. As sessões também podem ter estado associado, mas este é descartado quando a sessão termina.

Estas características fazem com que as sessões possam escalar facilmente. Na presença de um serviço suportado por múltiplos fornecedores de serviços, o sistema de execução, pode facilmente distribuir as sessões já existentes por todos esses fornecedores. Os serviços com noção de estado acabam por não ser modulares, pois a partilha de estado requer modelos de consistência que por sua vez prejudicam a escalabilidade.

3.1.1 Linguagem de especificação de serviços

A especificação da interface de um serviço foi elaborada de forma semelhante à sintaxe utilizada nas interfaces do Java ou C++. A língua será detalhada consoante a gramática presente na tabela 3.1. Aquando da especificação de um serviço `s` existe suporte a herança pelo que a interface de um serviço pode estender outras interfaces $\bar{r}$. É também na interface do serviço que as sessões têm de ser especificadas. As sessões também têm suporte a herança, permitindo assim que sejam estendias de outras várias sessões $\bar{n}$.

As sessões são construídas com hipótese de serem parametrizados argumentos ($\bar{a}$) e permitem que as operações $\bar{O}$ de um dado serviço sejam partilhadas por todos os clientes que se liguem ao mesmo.

Uma operação, tanto as que são acessíveis apenas ao serviço como as que estão definidas no contexto de uma sessão, têm de ser tipificadas segundo um tipo `t`, identificadas por um nome `o` e ter a hipótese de receber um conjunto de argumentos ($\bar{a}$). Todas as operações que forem definidas na especificação de um serviço são consideradas públicas por definição.

Um exemplo de especificação de um serviço com noção de sessão está ilustrado na Listagem 3.1, em que é especificado um armazenador que estrutura as suas entradas por chave e respectivo valor. Existe um método `newStore` e um `removeStore` que permitem, respectivamente, criar ou remover armazenadores com o identificador passado em parâmetro. no caso de se pretender efectuar operações sobre um armazenador em específico tem de ser criada uma sessão, devendo o identificador do armazenador pretendido ser passado em parâmetro no construtor da sessão. Uma fez estabelecida a sessão podem ser, podem ser efectuadas várias operações sobre o armazenador em questão, nomeadamente a operação `put` que permite adicionar uma nova entrada com a chave `K` e valor `V` no armazenador, o `get` que permite devolver a última entrada existente no armazenador com a chave `K` passada em parâmetro, o `getAll` que devolve uma colecção com todas as entradas que correspondam à chave `K` passada em parâmetro e finalmente o `delete` que permite remover todas as entradas que correspondam à chave `K` passada em parâmetro.

S	:=	**service** s : [**extends** $\bar{r}$] { [N] $\bar{O}$ }	Especificação do serviço
N	:=	**session** [**extends** $\bar{n}$] { C $\bar{O}$ }	Especificação da sessão
C	:=	($\bar{a}$)	Protótipo do construtor de sessão
O	:=	t o($\bar{a}$)	Protótipo de uma operação

Tabela 3.1: Linguagem de especificação de serviços.

```
import java.util.Collection;
public service KeyValueStore {
  <K, V> void newStore(String id);
  void removeStore(String id);
  session<K, V> {
    (String id);
    void put(K key, V value);
    void put(K key, Collection<V> value);
    V get(K key);
    Collection<V> getAll(K key);
    void delete(K key);
  }
}
```

Listagem 3.1: Exemplo de especificação de um serviço Zib

3.1.2 Linguagem de implementação de serviços

Na tabela 3.2 está especificada a gramática que define a linguagem de implementação de serviços. É através desta linguagem que se pode implementar os fornecedores de serviços a utilizar, bastando para tal utilizar a palavra reservada **provider** e fornecer qual a implementação concreta desejada, de acordo com a interface do serviço.

Um serviço pode conter campos, operações e sessões. Em que tanto os campos como as operações seguem a sintaxe utilizada nas linguagens mais comuns, nomeadamente o Java ou o C++. Como forma de garantir a consistência dos dados e a concorrência ao nível do serviço, cada operação pode ser qualificada como atómica ou síncrona, através da utilização das palavras reservadas **atomic** ou **sync** respectivamente. Todas as operações que forem declaradas além da interface de um serviço serão consideradas privadas por omissão (podendo ser declaradas como **protected**) e acessíveis apenas à implementação do serviço onde estejam declaradas. Utilizando as Listagens 3.2 e 3.3 como exemplo, a invocação da operação **syncOperation()** é sempre síncrona enquanto que a invocação da operação **asyncOperation()** é assíncrona.

As invocações a operações podem ser efectuadas, seguindo a anotação do ponto, ao estilo das invocações de métodos nas linguagens orientadas a objectos. Se na implementação da operação, esta estiver declarada sob a qualificação de **sync**, estamos perante uma operação assíncrona, caso esta operação não esteja qualificada como síncrona por omissão será assíncrona. Neste último caso, o programador que invoca a operação tem

a opção de utilizar explicitamente a palavra chave **sync**, fazendo com que esta seja invocada de forma síncrona, mesmo que esta não esteja assim definida na sua implementação. Uma invocação síncrona à operação assíncrona

```
1 sync String syncOperation() {
2
3 }
```

Listagem 3.2: Declaração de uma operação síncrona em Zib

```
1 String asyncOperation() {
2
3 }
```

Listagem 3.3: Declaração de uma operação assíncrona em Zib

As sessões podem ser criadas através da utilização do construtor **newsession** que devolve um elemento do tipo da sessão do serviço alvo. Estas sessões permitem que os clientes possam preservar os seus dados no serviço durante a utilização de várias operações no decorrer da sua execução.

Os fornecedores de serviços podem ser activos ou passivos. Activos, no caso de definirem o seu próprio fio de execução ou passivos no caso de simplesmente responderem a pedidos externos. A principal diferença entre ambos é as implementações consideradas activas, invocam outros serviços enquanto que as passivas apenas disponibilizam recursos de origem própria. Também for criado um processo a que chamamos de cliente, pois é um módulo que acede a outros serviços mas ele próprio não oferece qualquer serviço. Um cliente pode ser expresso ao nível da linguagem e para tal fim, reservamos a palavra **client**, que será traduzida para Java como sendo um serviço que consideramos vazio. Esta entidade não está representada na gramática por ser derivada de um serviço activo que implementa um `EmptyService` que não oferece quaisquer operações para o exterior, disponibilizando apenas forma para o cliente definir qual a sua linha de execução. O resultado de `client C{...}` será `C implements EmptyService{...}`.

Um exemplo de implementação da especificação do serviço demonstrado anteriormente relativo ao armazenador por relação chave-valor está referenciado no código presente na Listagem 3.4.

S	:=	[**abstract**] **provider** p **implements** s [**extends** $\bar{q}$] { M [N] }	Fornecedor (Implementação do serviço)
N	:=	**session** [**extends** n] { M }	Implementação da sessão
M	:=	$\bar{a}$ [C] $\bar{O}$	Conteúdos: parâmetros, construtor e operações
C	:=	($\bar{a}$) { $\bar{I}$ }	Construtor
O	:=	[**atomic**] [**sync**] [(**E**)] t o($\bar{a}$) { $\bar{I}$ }	Operação
I	:=	i	Instrução da linguagem hóspede
E	:=	**newsession** s($\bar{E}$)	Nova sessão
	|	[**sync**] E.o($\bar{E}$)	Invocação de uma operação
	|	e	Expressão da linguagem hóspede

Tabela 3.2: Implementação de serviços.

Um exemplo de implementação da especificação do serviço demonstrado anteriormente relativo ao armazenador por relação chave-valor está referenciado no código presente na Listagem 3.4.

3.2 Elina

O Elina [19] é um *middleware* Java que tem como finalidade suportar o desenvolvimento de aplicações paralelas sobre uma vasta variedade de arquitecturas, nomeadamente ambientes de memória partilhada e distribuída.

Como forma de garantir heterogeneidade em todos os nós e *clusters* suportados por esta ferramenta de trabalho, a arquitectura adoptada estrutura-se da seguinte forma:

API Centrada no conceito de objecto activo, conhecido como serviço Elina. Interface baseada em anotações que permite induzir uma programação declarativa para expressar tarefas ou paralelismo.

Núcleo Inclui toda a lógica independente da tecnologia do *middleware*

Camada de Abstracção Tecnológica Um nível de adaptação que permite a integração de diferentes tecnologias para ajustar o sistema no seu todo (por exemplo protocolo de comunicação em rede), ou ao nível de uma instância individual (por exemplo gestão de *pools* de *threads*).

O principal objectivo desta ferramenta é oferecer uma arquitectura com uma interface independente que permita a execução do mesmo código em variadas arquitecturas. Como tal o Elina pode ser utilizado como uma biblioteca de execução de computação paralela numa só máquina, como também ser utilizado como um *middleware* responsável pela execução de múltiplas aplicações concorrentes em ambiente distribuídos.

Os serviços Elina, que são programaticamente representados como `Service` e implementam a interface `IService` (Listagem 3.5), são construídos seguindo o padrão de desenho objecto activo [10]. A utilização deste padrão tem como objectivo permitir um modelo de programação mais flexível em que a execução das operações é desacoplado da invocação das mesmas, possibilitando que o resultado destas invocações possa ser retornado tanto de forma síncrona como assíncrona. Na interface `IService` presente na Listagem 3.5 apenas estão presentes os métodos relevantes para este trabalho, nomeadamente o `invoke` que permite fazer invocações a operações de um determinado serviço de forma assíncrona e o `cancel` que permite o cancelamento da execução do serviço.

O Elina permite interligar serviços, independentemente da sua localização física ou virtual, podendo estes comunicar entre si, caso estejam ambos a ser executados na mesma máquina ou até mesmo em máquinas diferentes. Para assegurar a comunicação entre serviços a executar em máquinas distintas, o Elina gera todo o código necessário a esta comunicação. Neste processo incluem-se representantes (stubs) que para além de serem

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
provider KeyValueStoreProvider implements KeyValueStore {

  Map<String, Map> storeMap = new HashMap<String, Map>();
  <K, V> void newStore(String id) {
    if (!storeMap.containsKey(id))
      storeMap.put(id, new HashMap<K,List<V>>());
  }
  void removeStore(String id) {
    if (storeMap.containsKey(id))
      storeMap.remove(id);
  }

  session<K, V> {
    Map<K, List<V>> store;
    (String id) {
      store = storeMap.get(id);
      if (store == null) {
        store = new HashMap<K,List<V>>();
        storeMap.put(id, store);
      }
    }

    atomic void put(K key, V value) {
      List<V> values = store.get(key);
      if (values == null) {
        values = new ArrayList<V>();
        store.put(key, values);
      }
      values.add(value);
    }

    atomic void put(K key, Collection<V> value) {
      List<V> values = store.get(key);
      if (values == null) {
        values = new ArrayList<V>();
        store.put(key, values);
      }
      values.addAll(value);
    }

    atomic V get(K key) {
      return (values == null) ? null : values.get(values.size()-1);
    }

    atomic Collection<V> getAll(K key) {
      return store.get(key);
    }

    atomic void delete(K key) {
      if (store.containsKey(key))
        store.remove(key);
    }
  }
}
```

Listagem 3.4: Exemplo de implementação de um serviço Zib

um importante nível de abstracção na comunicação entre serviços remotos, são também de particular importância nas novas funcionalidades que irão ser desenvolvidas neste trabalho e que iremos abordar mais à frente no capítulo 5.

Debruçando um pouco mais nas propriedades referentes a escalabilidade e escalonamento, o Elina foi concebido com suporte para agregação de conjuntos de serviços e construção de estruturas hierárquicas através da utilização de adaptadores para suportar as funcionalidades específicas para ambientes de memória distribuída. Existem quatro tipo de agregações de serviços possíveis, nomeadamente o *Distribute-Map-Reduce*, *Pool* de Serviços, Memória Particionada e por fim o *Façade*. Destes quatro comportamentos o quem tem particular interesse no âmbito desta dissertação é a *Pool* de Serviços (Figura 3.1). Este tipo de agregação de serviços foi o escolhido pela particularidade de ter uma implementação muito parecida à dos *stubs*, outra vantagem é a possibilidade de escalonar tarefas entre os serviços presentes na *pool*. Existem várias técnicas de escalonamento que podem ser aplicadas, contudo apenas utilizamos a implementação já feita do escalonamento *Round-robin*, em que as tarefas são escalonadas pelos serviços de forma circular.

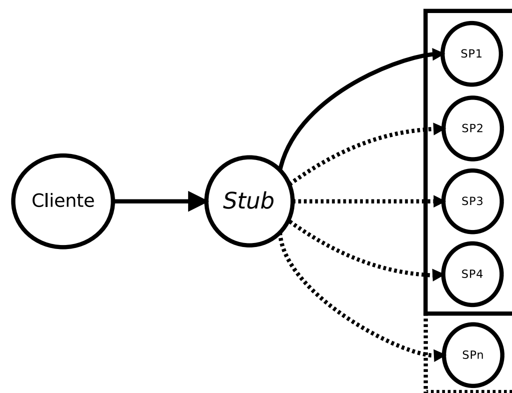

Figura 3.1: Ilustração de uma *Pool* de Serviços

```
1  import java.io.Serializable;
2  import java.util.UUID;
3  import javax.jws.WebMethod;
4
5  public interface IService extends Serializable{
6    <R> IFuture<R> invoke(String methodName, Object[] args)  throws
       NoSuchMethodException;
7    <R> IFuture<R> invoke(String methodName, Object[] args,Class<?>[] types)
       throws NoSuchMethodException;
8    [...]
9    public void cancel();
10 }
```

Listagem 3.5: Interface `IService`

3.3 ZibElina - Execução de computações Zib no *middleware* Elina

O modelo Zib foi concretizado como uma extensão à linguagem de programação Java. A compilação de código Zib em código Java fica a cargo de um compilador implementado para o efeito, o *Zibc*. Este compilador foi implementado sobre o Polyglot [14]. O Polyglot é um compilador de código fonte em código fonte extensível, especialmente desenhado para o desenvolvimento de extensões à linguagem Java. O *Zibc*, é portanto responsável por gerar o código Java correspondente aos vários serviços e respectivos fornecedores definidos nos ficheiros de entrada. Esse código Java faz, naturalmente, chamadas à API da plataforma Elina.

O fluxo de execução adjacente ao lançamento de uma aplicação Zib sobre o Elina, é iniciado pelo programador responsável pela implementação da aplicação Zib. Nesta primeira fase além de serem implementados todos os serviços necessários é também necessário criar um ficheiro (*Deployment.java*) onde se especifica todos os serviços que vão ser utilizados. Após isto, todos os ficheiros são passados para o compilador *Zibc* de onde o resultado será os respectivos serviços mas agora compilados para Java. Por fim para lançar a aplicação sobre o Elina é necessário implementar mais um ficheiro, o *Main.java*, onde é feita a associação entre as interfaces dos serviços e respectivos fornecedores de serviços. O fluxo de execução termina com o Elina a executar o resultado da compilação do ficheiro *Main.java*, utilizando sempre as referência necessárias aos ficheiros *java* resultantes da compilação do *Zibc*.

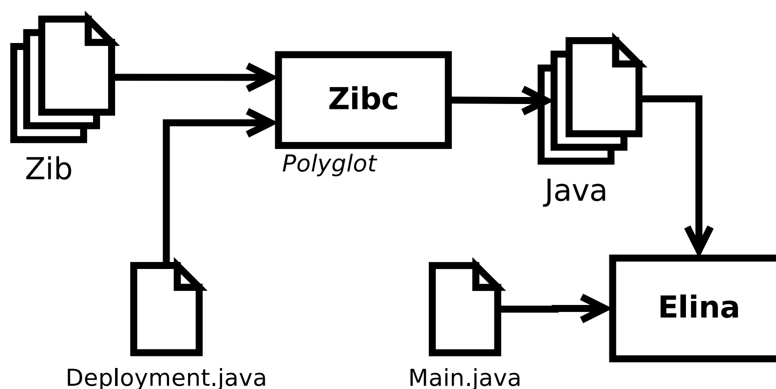

Figura 3.2: Compilação dos ficheiros Zib

3.3.1 Compilação de Zib para Elina

Com o intuito de demonstrar resultados de compilações do *Zibc* estão representados em seguida os resultados das compilações da interface do serviço presente na Listagem 3.1 e da sua respectiva implementação que está presente na Listagem 3.4.

O primeiro exemplo que apresentamos na Listagem 3.6 é o resultado da compilação, de Zib para Java, da interface do serviço *KeyValueStore*. Esta interface tinha definidos tanto operações fora como dentro de uma sessão e no resultado da compilação é possível verificar que tanto o serviço *KeyValueStore* como a sua sessão foram compilados para interfaces Java, em que ambos estendem a interface IService (Listagem 3.5). Isto deve-se ao facto de o serviço e a sua sessão partilharem do mesmo tipo de comportamento, pois ambos têm operações para serem invocadas por clientes.

```
1  import java.util.Collection;
2  import service.IService;
3  import service.aggregator.IReconfigurablePool;
4  public interface KeyValueStore extends IService, IReconfigurablePool {
5    public abstract <K, V> void newStore(String id);
6    public abstract void removeStore(String id);
7    public abstract <K, V> KeyValueStore.session<K, V> createSession(String id);
8    public static interface session<K, V> extends IService {
9      public abstract void put(K key, V value);
10     public abstract void put(K key, Collection<V> value);
11     public abstract V get(K key);
12     public abstract Collection<V> getAll(K key);
13     public abstract void delete(K key);
14   }
15 }
```

Listagem 3.6: Compilação para Java do serviço *KeyValueStore* da Listagem 3.1

De seguida ilustra-se o resultado da compilação da implementação do fornecedor de serviços *KeyValueStoreProvider*. A Listagem 3.4 apresenta a referida implementação, em Zib, do serviço *KeyValueStore* ilustrado na Listagem 3.1. Como resultado da compilação deste fornecedor de serviços para Java (Listagem 3.7) é possível verificar que o compilador converte os *providers* para classes que por sua vez vão estender a classe `Service` e implementar a interface do serviço a que estão associados. Quanto à implementação das sessões, estas tal como aconteceu na especificação dos serviços têm um resultado de compilação com alguma semelhanças ao resultado da compilação do *provider*, pois ambos têm implementações de operações para fornecer.

```java
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import zib.runtime.Service;
public class KeyValueStoreProvider extends Service implements KeyValueStore {
  static Map<String, Map> storeMap = new HashMap<String, Map>();
  public synchronized <K, V> void newStore(String id) {
    if(!storeMap.containsKey(id))
      storeMap.put(id, new HashMap<K, List<V>>());
  }

  public synchronized void removeStore(String id) {
    if (storeMap.containsKey(id)) storeMap.remove(id);
  }

  public <K, V> KeyValueStoreProvider.session<K, V> createSession(String id) {
    return new KeyValueStoreProvider.session<K, V>(id);
  }

  protected static class session<K, V>  extends Service implements KeyValueStore.session<K, V> {
    Map<K, List<V>> store;
    public session(String id) {
      super();
      store = storeMap.get(id);
      if (store == null) {
        store = new HashMap<K, List<V>>();
        storeMap.put(id, store);
      }
    }

    public synchronized void put(K key, V value) {
      List<V> values = store.get(key);
      if (values == null) {
        values = new ArrayList<V>();
        store.put(key, values);
      }
      values.add(value);
    }

    public synchronized void put(K key, Collection<V> value) {
      List<V> values = store.get(key);
      if (values == null) {
        values = new ArrayList<V>();
        store.put(key, values);
      }
      values.addAll(value);
    }
```

```
50
51      public synchronized V get(K key) {
52        List<V> values = store.get(key);
53        if (values == null) return null;
54        return values.get(values.size() - 1);
55      }
56
57      public synchronized Collection<V> getAll(K key) {
58        List<V> values = store.get(key);
59        if (values == null) return null;
60        return values;
61      }
62
63      public synchronized void delete(K key) {
64        if (store.containsKey(key)) store.remove(key);
65      }
66    }
67    public KeyValueStoreProvider() { super(); }
68  }
```

Listagem 3.7: Compilação para Java do *provider KeyValueStoreProvider* da Listagem 3.4

Por fim, passamos a demonstrar um exemplo da compilação de um cliente. Apesar de um cliente ter uma palavra reservada (*client*), o seu comportamento vai ter bastantes semelhanças ao de um serviço à excepção de que um cliente não oferece nenhum recurso para terceiros, ou seja, apenas consome. Portanto como é possível verificar no exemplo presente na Listagem 3.8, foi criado um cliente que irá consumir as operações especificadas e implementadas nos exemplos anteriores. No resultado da compilação do cliente para Java (Listagem 3.9) é então possível verificar as semelhanças já referidas para com o resultado da compilação de um serviço, sendo que a principal diferença está na classe que é estendida pelo cliente, neste caso a *service.ActiveService* que apesar de ser um serviço, não oferece mecanismos que permitam invocar operações. Todas as interfaces de serviços presentes na aplicação estão acessíveis aos clientes através da classe *Deployment*, que é a classe responsável por listar os mesmos. Os clientes de um serviço têm ainda a hipótese de fazer invocações assíncronas e para esse caso a utilização do tipo *IFuture* é mais adequada, visto que permite que uma invocação (*invoke*) seja feita a certo ponto da execução de uma aplicação e o retorno dessa invocação seja só consumido a quando da chamada do método *get()*, como é possível verificar no exemplo da compilação do cliente que se segue. A quando da invocação das operações é possível definir vários argumentos em parâmetro, tal situação é possível verificar no resultado da compilação do cliente na linha 34, nesse exemplo são passados os argumentos *name* e *weight*.

```
package applications.weightCenter;
import java.util.Scanner;
import java.util.Collection;
import zib.library.OutStream;
import utils.KeyValueStore;

public client weightCenterClient {
  public void run() {
    KeyValueStore.session<String, Integer> kv = newsession<String, Integer>
        KeyValueStore("Weight");
      OutStream.println("Options:");
      OutStream.println("add␣[name]␣[weight]\t\t-␣Add␣weight␣and␣person␣name␣to
          ␣store");
      OutStream.println("all␣[name]\t\t\t-␣Get␣all␣weights␣of␣person␣name␣in␣
          store");
      OutStream.println("quit\t\t\t\t-␣Exit␣application");
      Scanner sc = new Scanner(System.in);
    while(sc.hasNext()){
      String s = (String)sc.next();
      if(s.equalsIgnoreCase("quit")) {
        System.exit(0);
        }
      else if(s.equalsIgnoreCase("add") || s.equalsIgnoreCase("all")){
        String name = ((String)sc.next()).trim();
        int weight = sc.nextInt();
          if(s.equalsIgnoreCase("add")) {
            kv.put(name, weight);
            OutStream.println("Weight␣added␣successfully!");
          }
          else if(s.equalsIgnoreCase("all")) {
            String output = "|";
            Collection<Integer> temperatures = kv.getAll(name);
          for(Integer i : temperatures)
            output += i + "|";
          OutStream.println("Weight␣list␣of␣person"+name+":\n"+output);
          }
      }
      else
        OutStream.println("Invalid␣command.");
    }
  }
}
```

Listagem 3.8: Exemplo de um cliente de um serviço

```java
package applications.weightCenter;

import java.util.Scanner;
import java.util.Collection;
import zib.library.OutStream;
import utils.KeyValueStore;
import service.IService;
import service.aggregator.IReconfigurablePool;
import service.ActiveService;
import zib.runtime.Service;
import service.IFuture;
import zib.runtime.MultiArgs;

public class weightCenterClient extends service.ActiveService {
  public void run() {
    utils.KeyValueStore.session<java.lang.String, java.lang.Integer> kv =
        applications.temperatureCenter.Deployment.KeyValueStore
        .<java.lang.String, java.lang.Integer> createSession("Weight");
    applications.temperatureCenter.Deployment.OutStream.println("Options:");
    applications.temperatureCenter.Deployment.OutStream
        .println("add␣[name]␣[weight]\t\t-␣Add␣weight␣and␣person␣name␣to␣store"
            );
    applications.temperatureCenter.Deployment.OutStream
        .println("all␣[name]\t\t\t-␣Get␣all␣weights␣of␣person␣name␣in␣store");
    applications.temperatureCenter.Deployment.OutStream
        .println("quit\t\t\t\t-␣Exit␣application");
    java.util.Scanner sc = new java.util.Scanner(java.lang.System.in);
    while (sc.hasNext()) {
      java.lang.String s = (java.lang.String) sc.next();
      if (s.equalsIgnoreCase("quit")) {
        java.lang.System.exit(0);
      } else if (s.equalsIgnoreCase("add") || s.equalsIgnoreCase("all")) {
        java.lang.String name = ((java.lang.String) sc.next()).trim();
        int weight = sc.nextInt();
        if (s.equalsIgnoreCase("add")) {
          kv.<Void> invoke("put", MultiArgs.mult(name, weight)).get();
          applications.temperatureCenter.Deployment.OutStream
              .println("Weight␣added␣successfully!");
        } else if (s.equalsIgnoreCase("all")) {
          java.lang.String output = "|";
          service.IFuture<java.util.Collection<java.lang.Integer>> temperatures
              = kv
              .<Collection> invoke("getAll", MultiArgs.mult(name));
          for (java.lang.Integer i : temperatures.get())
            output += i + "|";
          applications.temperatureCenter.Deployment.OutStream
              .println("Weight␣list␣of␣person" + name + ":\n"
                  + output);
        }
      } else
```

```
48          applications.temperatureCenter.Deployment.OutStream
49              .println("Invalid␣command.");
50      }
51    }
52
53    public ClientExample() {
54      super();
55    }
56  }
```

Listagem 3.9: Resultado da compilação de um cliente de um serviço

3.3.2 Construção e Lançamento da Aplicação

No início da elaboração desta dissertação o lançamento de uma aplicação Zib sobre o *middleware* Elina estava condicionado pelo facto de que o programador tinha de implementar duas classes associadas aos lançamento da aplicação, sendo necessário para tal que o programador tivesse conhecimento da API do Elina. Um dos ficheiros era o *Deployment.java* onde o programador tinha de listar todos os serviços para que o compilador conhecesse o tipo destes serviços e o outro ficheiro era o *Main.java* que era onde se iniciava a execução da aplicação.

Como forma de automatizar todo este processo, um dos primeiros objectivos a cumprir no âmbito desta dissertação será criar suporte para uma linguagem que permita o *deployment* de serviços Zib, permitindo assim facilitar o processo de lançamento de uma aplicação.

4

Linguagem de Construção e Lançamento de Aplicações Zib

Neste capítulo será introduzida a linguagem responsável por compor serviços e por lançar aplicações desenvolvidas em Zib. Começaremos por apresentar qual a gramática por detrás da linguagem e quais as ferramentas utilizadas para permitir a interpretação e tradução da mesma. Seguido de pequenos exemplos da própria linguagem e de resultados gerados pelo tradutor da mesma. No final deste capítulo serão também evidenciados os procedimentos implementados para permitir que a linguagem integre serviços disponíveis na Web.

4.1 Linguagem

Como foi referido no capítulo anterior, antes de ter sido dado início ao trabalho desta dissertação, para compor vários serviços, era necessário mapeá-los manualmente num ficheiro Java (Secção 3.3.2). Como forma de automatizar esse processo foi proposta a criação de uma linguagem capaz de definir o *deployment* de uma aplicação, ou seja a ligação entre os serviços e os seus respectivos fornecedores (*providers*). Para tal fim, foi criada uma gramática (Tabela 4.1) e respectiva linguagem. Esta linguagem permite que sejam especificados os serviços (**service**) a serem utilizados na aplicação e a sua implementação. Uma propriedade relevante quanto ao dinamismo a que a linguagem dá suporte é o facto de ser permitido definir múltiplos fornecedores para um dado serviço, bastando para tal adicionar **n of** antes do fornecedor pretendido, sendo **n** a quantidade de fornecedores desejada. Um dado serviço pode ser declarado de forma heterogénea por fornecedores

de diferentes tipos. Para tal é necessário conter cada tipo separado por virgulas da seguinte forma **n1 of x, n2 of y**, sendo **x** e **y** o correspondente a diferentes implementações do mesmo serviço. Para além de ser possível ligar interfaces de serviços a implementações disponíveis localmente, podem também ser procuradas (**lookup**) implementações num registo conhecido do sistema de execução ou até mesmo indicar fornecedores de serviços disponíveis na Web, bastando para tal indicar, através de uma `string` o URI da localização do fornecedor desejado.

Dado que o propósito desta linguagem é definir a totalidade dos módulos que compõem uma dada aplicação, também é possível que sejam especificados clientes (**client**) que irão ser implementados de forma a usufruírem das operações disponibilizadas pelos serviços.

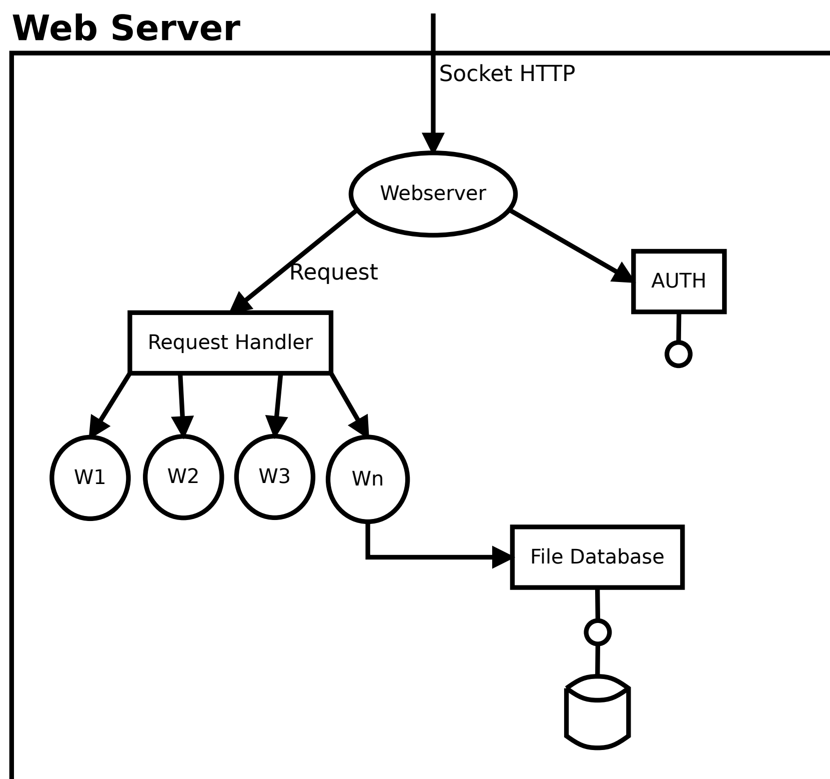

Figura 4.1: Exemplo de um servidor Web

Além de permitir a ligação entre servidores e fornecedores, a linguagem permite a especificação de canais de comunicação entre estes últimos. Nesse sentido, a linguagem também oferece algum suporte para a composição dos serviços requeridos pela aplicação. Os canais de comunicação entre os serviços são especificados através de sessões (**session**). Um bom exemplo em que a composição de serviços exportando uma aplicação útil e bastante frequente é um servidor Web. Tal como é possível ver na Figura 4.1 em que está representado um servidor Web, subdividido por módulos, entenda-se serviços, é simples criar uma aplicação em que se possa combinar tanto a comunicação entre os

vários serviços como ter serviços com políticas de escalonamento. No exemplo desta figura o cliente *Webserver* fica responsável por receber os pedidos dos utilizadores através de um *socket* HTTP, posto isto cabe-lhe a ele ou reencaminhar o pedido para autenticação através do serviço *AUTH* ou reencaminhar para uma *pool* de serviços no caso de o utilizador já estar autentificado e estar tudo correcto para o utilizador ter acesso, por exemplo, à base de dados. A base de dados por sua vez pode ser acessível través de outro serviço que pode oferecer um nível de abstracção para a localização das base de dados, disponibilizando apenas o resultado aos pedidos feitos independentemente da base de dados utilizada, esteja esta disponível localmente ou remotamente.

A linguagem tem também suporte para definir pacotes de configurações que podem ser criados e referenciados através da utilização da palavra reservada **bundle**. Permite definir, de certa forma, uma coreografia, no sentido em que esta facilita a especificação dos serviços que vão compor a aplicação e os seus respectivos canais de comunicação. Dando lugar a que estes saibam como devem interagir entre eles durante a execução da aplicação.

D	:=	$\bar{S}$	Especificação de um *deployment*
	|	**bundle** b($\bar{a}$) $\bar{S}$	Definição de um *bundle*
S	:=	**service** s : $\bar{R}$	Mapeamento de um serviço para os seus fornecedores
	|	**session** n : **newsession** s($\bar{e}$)	Estabelecer um sessão com o serviço em questão
	|	**client** : $\bar{p}(\bar{e})$	Declarar um conjunto de clientes parametrizáveis
	|	**bundle** : $\bar{b}(\bar{e})$	Utilização de um *bundle*
R	:=	[e **of**] P	Definir quantidade e parâmetros de um dado fornecedor
P	:=	$p(\bar{e})$	Fornecedor para ser lançado
	|	e	URI para um fornecedor a correr remotamente
	|	**lookup** [p]	Procurar por fornecedores de uma lista de serviços

Tabela 4.1: Linguagem de composição de uma aplicação

4.2 Tradutor

Nesta secção vamos detalhar como é que incorporámos a linguagem no ecossistema Zib existente. Como forma de permitir que a linguagem abrangida neste capítulo seja depois integrada no sistema ZibElina, houve a necessidade de criar um tradutor responsável por traduzir esta linguagem para Java. Se compararmos a Figura 4.2 com a Figura 3.2, podemos verificar que na primeira referida foi adicionado um tradutor e que este agora é responsável por gerar os ficheiros *Deployment.java* e *Main.java* que por sua vez depois lançam a aplicação sobre o Elina. O tradutor criado só tem fases de análise sintáctica e geração de código, pelo que não é feita qualquer tipo de análise semântica. No caso de o ficheiro a traduzir conter erros, estes serão reportados durante a compilação dos ficheiros gerados. Após o ficheiro Java ser gerado como resultado da tradução da linguagem de

composição de serviços e dos restantes ficheiros Zib relacionados com a aplicação serem compilados também para Java, o ficheiro traduzido com a composição dos serviços lança a aplicação e integra todos os elementos que compõem a aplicação, por sua vez, estes executam e fazem invocações sobre o *middleware* Elina.

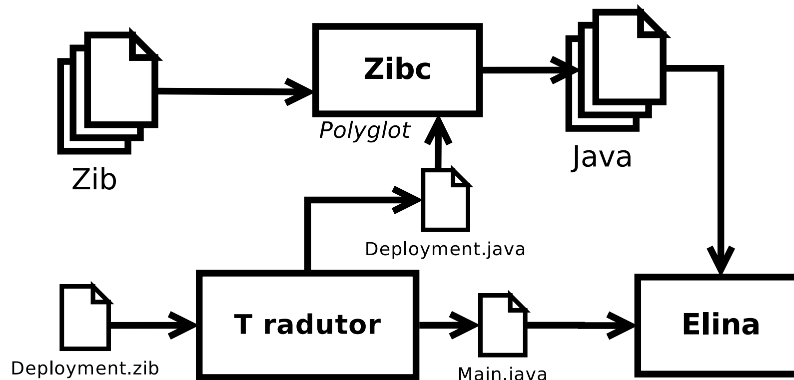

Figura 4.2: Sistema ZibElina com linguagem de composição e lançamento de aplicação e respectivo tradutor.

4.2.1 Exemplo de tradução

No exemplo que se segue (Listagens 4.1 e 4.2) é possível verificar uma composição de uma aplicação, seguida do respectivo resultado da sua tradução. Na Listagem 4.1, mais concretamente na linha 1, foi composta uma aplicação em que é especificado um serviço `Queue` e que tem como fornecedor uma implementação local `QueueProv`. Já na linha 2 a especificação é referente a uma sessão que será estabelecida com o serviço `Queue` previamente já declarado. O cliente da aplicação recebe como argumento a sessão especificada anteriormente, cuja servirá de canal de comunicação entre o cliente e o serviço.

Como resultado do exemplo de composição descrito acima, foi gerado um ficheiro Java que irá lançar a aplicação por intermédio da classe `ApplicationLauncher` que permite inicializar e lançar a aplicação. Por omissão é sempre criado um `Stub`, através da invocação do método `createStub()` para cada fornecedor de serviços presente e um serviço fica sempre a apontar para uma *pool* que contém pelo menos um fornecedor, na Listagem 4.2 na linha 10 o serviço está a ser associado a uma *pool* de serviços constituída pelos fornecedores de serviços presentes na lista `providersList0`. O número de fornecedores presente nesta lista vai depender da parametrização do número de fornecedores desejados pelo programador a quando da declaração do serviço na linguagem de lançamento e composição de aplicações. Todos estes detalhes são alusivos a questões de dinamismo e serão pormenorizados no próximo capítulo.

Quando o programador começa a programar o ficheiro *Deployment.zib*, que é o ficheiro de entrada do tradutor, tem duas opções, ou define um *bundle* de forma a poder reutilizá-lo ou então não define nada e o *Main.java* fica traduzido com uma execução instantânea através do método *main()* que vai inicializar a aplicação, tal e qual como é

```
service Queue : QueueProv() ;
session s : newsession Queue(10) ;
client : C(s) ;
```

Listagem 4.1: Especificação de uma composição

```
public static void main(String args[]){
  try{
    if(args.length == 1)
      ApplicationLauncher.init(args[0]);
    else
      ApplicationLauncher.init();
    Application _app = Application.newInstance();
    java.util.List<Queue> providersList0 = new java.util.ArrayList<Queue>();
    providersList0.add((Queue)new QueueProv().createStub());
    Deployment.Queue = (Queue) ServiceAggregator.<Queue>servicePool((Queue[])
        providersList0.toArray(new Queue[providersList0.size()]));
    _app.addService(Deployment.Queue);
    Queue.session s = Deployment.Queue.createSession(10);
    _app.addService(new C(s));
    ApplicationLauncher.deploy(_app);
  }catch (Exception e) {
    e.printStackTrace();
  }
}
```

Listagem 4.2: Composição gerada pelo tradutor

possível verificar na Listagem 4.3 em que não há nenhum *bundle* definido. No caso de ser definido um *bundle* o resultado da tradução será igual ao da Listagem 4.4, que provém de um *bundle* identificado com o identificador `B`. Neste último caso é criada uma classe com o nome do identificador do *bundle* e é passada a `Application` em parâmetro no construtor de forma a que depois todos os serviços do *bundle* possam ficar acessíveis pelo resto da aplicação, sendo que os serviços do *bundle* serão adicionados à mesma.

```
public static void main(String args[]){
  try{
    if(args.length == 1)
      ApplicationLauncher.init(args[0]);
    else
      ApplicationLauncher.init();

    Application _app = Application.newInstance();
    ApplicationLauncher.deploy(_app);
  }catch (Exception e) {
    e.printStackTrace();
  }
}
```

Listagem 4.3: Resultado da tradução da estrutura de uma aplicação sem *bundle*

```
public class B{
  B(Application _app){

  }
}
```

Listagem 4.4: Resultado da tradução da estrutura de uma aplicação com um *bundle* B

4.2.2 Implementação

Como forma de traduzir a linguagem de composição de serviços em Java, começamos por criar o *parser* representado na Figura 4.3. A criação deste *parser* foi feito, através da utilização de uma gerador de *parsers* em Java, o JavaCC[8]. O *parser* foi gerado com base na gramática definida na Tabela 4.1. Após o *parser* estar gerado conseguimos que dado uma linguagem de entrada para ser analisada resulta-se uma árvore sintáctica, no caso de a análise sintáctica não detectar erros, ou resulta-se uma excepção no caso de existir erros de construção sintáctica na linguagem a validar. Se o resultado final do *parser*, for de facto, uma árvore sintáctica, esta é percorrida e traduzida pela ordem de entrada. Para percorrer a árvore sintáctica foi implementado um sistema de *visitors* como é possível verificar na Listagem 4.5. Para cada nó visitado é traduzido o nó em questão e no caso de existirem mais nós aninhados a este, segue-se o mesmo procedimento que do nó actual para os aninhados de forma recursiva.

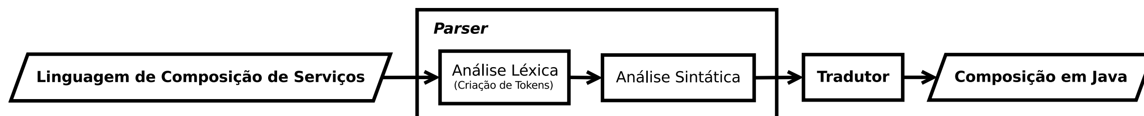

Figura 4.3: Fases da tradução da linguagem de composição e lançamento de aplicações

```
public interface ITranslatorVisitor<T> {
  public void visit(ASTBundle bundle, CompositionCode code);
  public void visit(ASTBundleCall bundlec, CompositionCode code);
  public void visit(ASTClient client, CompositionCode code);
  public void visit(ASTLookup lookup, CompositionCode code);
  public void visit(ASTService service, CompositionCode code);
  public void visit(ASTSession session, CompositionCode code);
  public void visit(ASTCall callid, CompositionCode code);
  public void visit(ASTURL url, CompositionCode code);
  public void visit(ASTMain mainNode, CompositionCode code);
  public void visit(ASTId id, CompositionCode code);
  public void visit(ASTServiceContent sc, CompositionCode code);
  public void visit(ASTSystemService ss, CompositionCode code);
}
```

Listagem 4.5: Interface dos *visitors* do tradutor

4.2.3 Incorporação

A incorporação do tradutor no processo de compilação do Elina foi feita através do *software* Maven[1], que é o responsável pela composição de todos os componentes que permitem o resultado final deste trabalho. Para além de serem especificadas todas as dependências entre cada componente é também através do Maven que é incorporado o processo de testes.

O processo de testes consiste na verificação do resultado da tradução da linguagem especificada neste capítulo para a linguagem Zib. Como tal, foram criados dois cenários de testes, os que simplesmente verificam se o código é traduzido sem ser lançada nenhuma excepção, ou o caso em que se comete um erro despropositado de sintaxe para verificar se o erro despoletado é o esperado. Em ambos os casos é especificado um ficheiro de *input* para ser tratado pelo tradutor durante o processo de teste. No caso de o teste ser o da verificação da mensagem de erro, é especificado um ficheiro contendo o erro esperado, afim de poder ser efectuada a comparação.

4.3 Ligação aos Serviços da WEB

Esta linguagem de composição de serviços, para além de ter suporte para integrar serviços, tanto internos como remotos, tem também suporte para integrar serviços externos. Para tal basta associar à especificação de um serviço, um URI com a localização do serviço Web que se deseja integrar. Este URI é representado na linguagem sobre a forma de `String` e o resultado da tradução deste tipo de serviços é um código Java que executa e gera o lado cliente do serviço Web durante o tempo de execução. Como nos restantes serviços, é criado um *stub* e uma *pool* por questões de dinamismo. Os detalhes da geração do código do lado do cliente de forma a tornar possível o acesso ao fornecedor do serviço da Web serão abordados no próximo capítulo.

```
1 service TemperatureConverter:"http://www.w3schools.com/webservices/tempconvert.
    asmx?WSDL";
```

Listagem 4.6: Especificação de um serviço Web

[1]Ferramenta de automação de compilação e gestão de projectos - *http://maven.apache.org*

```java
public class Composition {
  public static void main(String args[]) {
    try {
      if (args.length == 1)   ApplicationLauncher.init(args[0]);
      else                    ApplicationLauncher.init();

      Application _app = Application.newInstance();

      java.util.List<TemperatureConverter> providersList0 = new java.util.
          ArrayList<TemperatureConverter>();

      providersList0.add((TemperatureConverter) new WebServiceGenerator("zib.
          compiler.tests.webServiceIntegration", "TemperatureConverter").<
          TemperatureConverter>getStub("http://www.w3schools.com/webservices/
          tempconvert.asmx?WSDL"));

      Deployment.TemperatureConverter = (TemperatureConverter)
          ServiceAggregator
          .<TemperatureConverter> servicePool((TemperatureConverter[])
              providersList0
              .toArray(new TemperatureConverter[providersList0
                  .size()]));

      _app.addService(Deployment.TemperatureConverter);
      ApplicationLauncher.deploy(_app);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Listagem 4.7: Resultado da tradução da especificação de um serviço Web

5

Sistema de Execução

Neste capítulo serão listadas, detalhadas e será feita uma referência à implementação das reconfigurações utilizadas para garantir suporte ao dinamismo nos serviços. Estas reconfigurações foram implementadas ao nível do sistema de execução e permitem garantir escalabilidade e tolerância a falhas e suporte para ajuste de fornecedores de serviços consoante as necessidades aplicacionais.

5.1 Operações adicionadas aos serviços Zib

Para que um serviço possa suportar operações de reconfiguração, é necessário a quando da sua declaração, de especificar que este vai ser reconfigurável através da utilização da palavra reservada *reconfigurable* (Listagem 5.1).

```
reconfigurable service X{
    ...
}
```

Listagem 5.1: Serviço Zib com suporte para reconfiguração dinâmica

5.1.1 Descrição das operações

As reconfigurações dinâmicas criadas com o objectivo de oferecer suporte ao dinamismo são as seguintes:

Substituir o fornecedor de um serviço A finalidade desta operação de reconfiguração é permitir que em tempo de execução possa ser substituído o fornecedor de um serviço, ou

seja, o objectivo é permitir que a a interface de um serviço possa trocar a sua implementação, independentemente da sua localização. Deverá ser permitido que a implementação de um serviço seja oriunda da própria aplicação, sistema de execução ou até mesmo da Web. Esta reconfiguração é uma vantagem no cenário de tratamento de falhas provenientes de um dado fornecedor, pois possibilita que caso seja lançada uma excepção causada por uma falha de um fornecedor a aplicação automaticamente possa substituir este por um novo durante a sua execução.

Caso o programador queira utilizar esta reconfiguração pode fazê-lo através da invocação da operação `replace(S newProvider)` para o caso de querer substituir o fornecedor de um serviço por um ao nível da aplicação. Caso o objectivo seja substituir o fornecedor por uma implementação presente ao nível do sistema de execução, basta ao programador utilizar a operação `replaceFromSystem()` e o próprio sistema de execução efectua um *lookup* de forma a atribuir uma implementação do sistema ao serviço. Por fim, caso o programador opte por reconfigurar o serviço de forma a que este seja implementado por um fornecedor proveniente da Web, deverá utilizar a operação `replace(URL url)` em que o parâmetro deverá ser a localização do novo fornecedor.

Parar ou recomeçar a execução de um fornecedor de serviços A utilização desta reconfiguração deverá permitir que o consumo de pedidos por parte de um fornecedor de serviços seja pausado ou recomeçado. Esta ferramenta aplica-se a qualquer um dos três tipos de serviços já referidos anteriormente, serviços do sistema de execução, aplicação ou da Web. Esta reconfiguração tem particular utilidade no caso de, por exemplo, ser necessário substituir um fornecedor de um dado serviço e como forma de prevenção de durante esta substituição não serem perdidos pedidos, deveria-se pausar um fornecedor antes de o substituir e só voltar a recomeçar o consumo de pedidos por parte do mesmo após a substituição estar finalizada com sucesso.

Adicionar ou remover fornecedores de serviços a um agregador Esta reconfiguração permite que seja feita uma gestão do número de fornecedores associados ao mesmo serviço em simultâneo. Esta gestão é permitida durante a execução de uma aplicação de forma a distribuir, por exemplo, os pedidos pelos vários fornecedores. A principal vantagem da utilização desta reconfiguração é a construção de aplicações dinâmicas e escaláveis.

5.1.2 Implementação

Todo o suporte ao dinamismo e reconfigurações relacionadas foram implementadas ao nível do sistema de execução, Elina. Antes do início deste trabalho o Elina já tinha implementado um nível de abstracção entre a interface do serviço e a sua implementação, por meio de um intermediário (*stub*). Contudo, este intermediário só era utilizado no caso de existirem interacções com serviços Elina remotos, sendo que para os serviços Elina locais

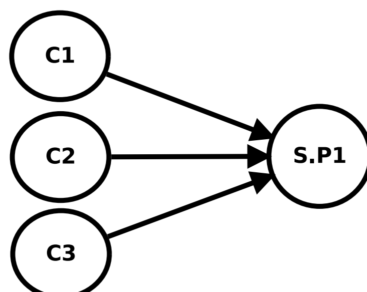

Figura 5.1: Vários clientes a acederem ao mesmo serviço.

a ligação ao serviço era feita de forma directa e sem qualquer tipo de intermediário ou nível de abstracção (Figura 5.1).

Como forma de garantir que as aplicações possam ser lançadas compondo, de forma transparente para o programador, serviços das três diferentes origens, da aplicação, do sistema de execução e da Web, tivemos de passar a aplicar a noção de *stub* para qualquer tipo de serviço (Figura 5.2). Esta abstracção permite desacoplar o cliente do fornecedor, por forma a podermos fazer as reconfigurações planeadas. Um exemplo prático do resultado da implementação da geração de código para que seja sempre gerado um *stub* para cada serviço, está demonstrado na linha 12 da Listagem 4.2 através da utilização da operação `createStub()`.

Para além de ser sempre criado um *stub* para cada serviço, existiu também a necessidade, por questões de dinamismo, de se criar sempre uma fila de fornecedores (*pool*). Mesmo que a um serviço só seja parametrizado um único fornecedor é sempre criada uma *pool*, mesmo que só com um fornecedor. Esta implementação dá um nível de abstracção ao programador, no sentido em que este consegue incrementar ou remover novos fornecedores a um serviço, independentemente da declaração inicial que tenha feito para o mesmo.

Todos os serviços que sejam declarados como reconfiguráveis são compilados para um novo tipo de serviço `zib.runtime.ReconfigurableService` que estende a interface `service.Service` e a interface `service.aggregator.IReconfigurablePool` (Listagem 5.2).

Um fornecedor de serviços que implemente um serviço reconfigurável é compilado para `zib.runtime.ReconfigurableProvider` que implementa o `zib.runtime.ReconfigurableService`.

De modo a aprofundar o nível de detalhe, passamos a explicar as implementações efectuadas nas seguintes reconfigurações:

Parar ou recomeçar a execução de um fornecedor de serviços Como era impossível garantir a pausa do consumo de pedidos por parte de serviços da Web, foi também necessário para esta reconfiguração a existência de um nível intermédio responsável por guardar os pedidos em causa de o fornecedor remoto ser pausado, para posteriormente

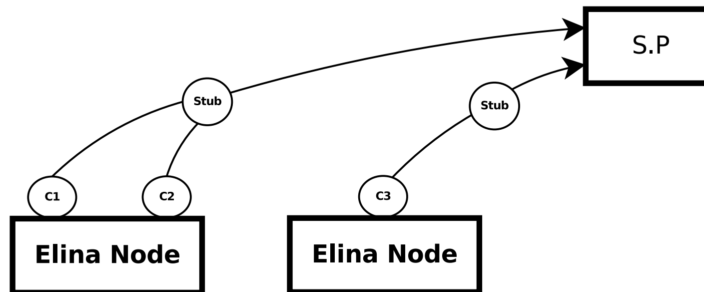

Figura 5.2: Vários clientes a acederem, de nós Elina distintos, ao mesmo serviço através de um *stub*.

serem consumidos, no caso de este fornecedor retomar a sua execução e consecutivamente o consumo dos pedidos guardados pelo *stub*.

Esta operação foi especificada na interface `IReconfigurable` como é possível ver nas linhas 9 e 11 da Listagem 5.3. Como forma de permitir a pausa e respectivo recomeço de consumo de pedidos por parte de todos os tipos de fornecedores de serviços foi adicionada uma variável binária *paused* que caso esteja a `true` significa que os pedidos não podem ser consumidos e caso esteja a `false` o fornecedor está em execução e pronto para consumir pedidos, seguindo o procedimento normal.

Quando é invocado um `pause()` esta variável passa a `true` e qualquer pedido que chegue após isso é guardado numa fila onde são guardados todos os pedidos por parte dos clientes. Caso seja invocado o `resume()` de um fornecedor, então a variável volta a ficar a `false` e todos os pedidos guardados na fila são executados por ordem de chegada e retornados os resultados da execução dos mesmos para os clientes.

Adicionar ou remover fornecedores de serviços a um agregador Utilizando o facto de que já existia um agregador de fornecedores de serviços com o intuito de ser definido o número de fornecedores necessários consoante as necessidades das aplicações, o objectivo desta operação é permitir que este mesmo agregador seja gerido dinamicamente a um nível superior, portanto, ao nível da linguagem. Como tal foi necessário implementar duas operações no sistema de execução e exportá-las para serem acessíveis a qualquer cliente. Estas operações foram definidas na interface `IReconfigurablePool` (Listagem 5.2).

No caso da implementação da operação `add(R provider)`, apenas o fornecedor de serviços passado em parâmetro é adicionado à lista de serviços e o tamanho da lista de serviços da *pool* é actualizado no escalonador para não haver falhas de distribuição. No caso da operação `remove(R provider)` é feito exactamente o contrário, ou seja, é eliminado o fornecedor passado em parâmetro e é actualizado o tamanho da lista de fornecedores do escalonador para um número inferior, evitando também, falhas de distribuição.

Substituir o fornecedor de um serviço Para executar esta operação de reconfiguração foi necessário garantir que o serviço e o seu fornecedor estavam completamente desagregados de forma a que um cliente pudesse substituir o fornecedor de um serviço de uma forma completamente transparente.

Esta operação foi especificada na interface `IReconfigurable`. Tendo em conta que o fornecedor de um serviço está sempre representado por intermédio de um *stub* e de uma *pool*, esta operação irá substituir o fornecedor com a ajuda de escalonador que percorre a *pool* de forma circular. Ao nível do *stub* apenas basta verificar se as interfaces do fornecedor já atribuído ao serviço e do novo fornecedor a atribuir correspondem e caso coincidem, o fornecedor existente na classe `ServiceStub` é atribuído ao novo passado em parâmetro. Durante a execução desta substituição são utilizadas as operações, já referidas, para paragem ou recomeço da execução de um fornecedor, para que quando se proceda à substituição de um *provider* seja garantido que o *provider* a ser substituído pause a sua execução antes de ser substituído evitando assim que execuções fiquem pendentes ou mal resolvidas.

```
1  public interface IReconfigurablePool extends IReconfigurable {
2    @WebMethod(exclude = true)
3    <R> void add(R provider);
4    @WebMethod(exclude = true)
5    <R>  void remove(R provider);
6  }
```

Listagem 5.2: Interface com operações de reconfiguração para *pools*

```
1  public interface IReconfigurable {
2    @WebMethod(exclude = true)
3    void replaceFromSystem();
4    @WebMethod(exclude = true)
5    void replace(URL url);
6    @WebMethod(exclude = true)
7    <S> void replace(S newProvider);
8    @WebMethod(exclude = true)
9    void pause();
10   @WebMethod(exclude = true)
11   void resume();
12 }
```

Listagem 5.3: Interface com operações de reconfiguração para *pools*

5.2 Suporte para integração de Serviços WEB

Existem duas formas de numa aplicação Zib serem integrados serviços da WEB, uma é a quando do lançamento de uma aplicação no caso de no ficheiro de lançamento da aplicação o serviço ter como *provider* um URL para um *WSDL* com a especificação de um serviço remoto ou, quando programaticamente numa aplicação existe um `replace(URL`

url). Apesar de existirem estas duas formas distintas de integrar os serviços WEB, estes utilizam o mesmo mecanismo para serem integrados, havendo uma pequena alteração quanto ao momento em que são integrados. No caso de ser no ficheiro de lançamento e composição da aplicação, o serviço WEB é integrado em tempo de compilação enquanto que se este for integrado programaticamente, este vai ser integrado durante o tempo de execução.

5.2.1 Detalhes de implementação

O processo de integração de serviços da WEB começa com a instanciação da classe *WebServiceGenerator* que está ilustrada no Anexo A.1. É nesta classe que o serviço remoto começa a ser importado e para tal é necessário passar em parâmetro no momento da instanciação, a pasta da aplicação, o nome do pacote para onde será importado o serviço da WEB e o nome do serviço a ser importado. Num cenário em que temos uma aplicação no seguinte pacote `applications.scheduler` e importamos um serviço da WEB com o nome *calendar* para esta aplicação, deverá ser criado o pacote com o nome `applications.scheduler.calendarWsStub` para onde será importado todo o código do cliente do serviço remoto.

Após ser criado o objecto da classe *WebServiceGenerator*, é então invocado o método `getStub(String url)` com o URL do serviço a importar. Posto isto, é utilizada a classe *WSImport* que está ilustrada no Anexo A.2. Quando é criado um objecto desta última classe logo no construtor é feita a validação ao URL do serviço, para verificar se é válido e está activo, caso esteja tudo operacional os atributos do *WSImport* são preenchidos com os valores passados em parâmetro e o objecto fica pronto para que seja chamado o método `importClient()`.

Antes do `importClient()` ser chamado, para optimizar, o *WebServiceGenerator* vai verificar se o serviço a ser importado já se encontra disponível na directoria da aplicação e numa versão não mais recente do que a última alteração feita no ficheiro Zib que utiliza o serviço remoto, como é possível verificar na linha 42 do Anexo A.1. Caso o serviço não exista ou esteja desactualizado, é então chamado o método `importClient()`. Neste método, primeiro é verificado se o serviço já existe e caso exista é eliminado, após isso é executado um ficheiro *Ant*[1] onde está especificada uma tarefa que permite executar a ferramenta *wsimport*[2].

Quando o processo de transferência com o *wsimport* terminar é iniciado o tratamento do código gerado, para tal o *WebServiceGenerator* utiliza a classe *ZibifyClientWS*. Esta última classe, faz *load* dos ficheiros Java gerados pelo *wsimport* e altera os ficheiros de forma a que estes possam ser invocados por intermédio de um *stub* compatível com o sistema de execução Elina. No fim das alterações estarem feitas é gerado um *WebServiceStub* que vai estender um *ServiceStub* do Elina e vai ter um *stub* para fazer a ligação com a interface de comunicação com o serviço remoto gerada pelo *wsimport*. Um exemplo de um

[1] Apache Ant é uma ferramenta utilizada para automatizar a construção de software.
[2] Ferramenta para gerar código Java com base num *WSDL*.

WebServiceStub gerado está ilustrado no Anexo A.4. Para além das hierarquias referidas que permitem que um serviço WEB seja integrado no sistema de execução são também geradas no *WebServiceStub* todas as operações remotas que o serviço oferece.

No fim de toda a geração do código a classe *ZibifyClientWS* tem um método denominado de `getWebServiceStub`, que vai fazer o *load* de todo o código gerado, compilar o mesmo e instanciar o *WebServiceStub* para ser devolvido ao sistema de execução e ficar apto para sofrer invocações dando sempre um nível de abstracção ao invocador.

6

Exemplos de Aplicação

Neste capítulo serão ilustrados e explicados três exemplos de aplicações em Zib compostas por mais do que um serviço. Será também demonstrado para cada aplicação um exemplo de execução e respectivos resultados gerados para o utilizador da aplicação.

6.1 Aplicação *Weather Information*

A aplicação que iremos ilustrar de seguida é composta (Listagem 6.1) por três serviços com fornecedores provenientes de diferentes origens, um serviço é definido ao nível aplicacional (Listagem 6.4), outro é do sistema e por fim o último está implementado na Web (Listagem 6.2). Existe também um cliente (Listagem 6.3) que será o responsável por fazer invocações a estes serviços.

O fluxo de execução desta aplicação é o seguinte:

1. O cliente invoca a operação `Weather.getWeather("Lisboa", "Portugal")` ao serviço da Web, pedindo a temperatura para Lisboa, Portugal;

2. O cliente inicializa o serviço *XMLParser* e passa em parâmetro a resposta, em XML, do serviço da Web;

3. O cliente invoca a operação `print(String s)`, do serviço de sistema *OutStream*, com o valor da temperatura obtido do serviço *XMLParser*

O resultado final da execução desta aplicação foi: `Lisboa:  55 F (13 C)`

```
service Weather : "http://www.webservicex.com/globalweather.asmx?WSDL";
service zib.library.OutStream : system;
service utils.XMLParser : utils.providers.XMLParserProvider();
client : Client();
```

Listagem 6.1: Ficheiro de lançamento da aplicação *Weather Information*

```
service Weather {
  String getWeather(String cityName, String countryName);
  String getCitiesByCountry(String countryName);
}
```

Listagem 6.2: Interface do serviço da Web

```
client Client {
  void run() {
    String _weather = Weather.getWeather("Lisboa", "Portugal");
    XMLParser.session xml = newsession XMLParser(_weather);
    OutStream.println("Lisboa:␣"+ xml.getValue("Temperature"));
  }
}
```

Listagem 6.3: Cliente da aplicação *Weather Information*

```
service XMLParser {
  session{
    (String xml);
    String getValue(String id);
  }
}
```

Listagem 6.4: Serviço responsável por interpretar XML

6.2 Aplicação *Temperature Center*

Com base na aplicação anterior decidimos criar outra aplicação que para além de utilizar o serviço remoto *Weather*, usa também uma *store* que indexa temperaturas por chave e respectivo valor. O serviço que usamos como *store* foi o já ilustrado na Listagem 3.1, o *KeyValueStore*. Foi criado outro serviço que é o responsável por manipular os dados guardados na *store* e disponibiliza-los para clientes por intermédio de operações, o serviço é o *TemperatureCenter* e uma das operações que este disponibiliza é a `startCityListener(String cityName, String countryName)` que adiciona uma cidade à lista de cidades que são para monitorizar em *background*. Existem mais duas operações que são para obter informação relativa ás temperaturas das cidades que estão a ser monitorizadas, as operações em questão são, a `getCityTemperatures(String cityName, String countryName)` e a

`getCityAVG(String cityName, String countryName)`, tanto uma como a outra recebem a cidade e o respectivo país, contudo a a primeira operação retorna uma lista com todas as temperaturas efectuadas da cidade e país em questão e a segunda devolve a média das leituras efectuadas também de uma cidade e país que esteja a ser monitorizado.

Como é possível verificar no Anexo A.3 em que está ilustrada uma implementação do serviço *TemperatureCenter* que implementa a interface *Runnable*, existe um método `run()` que por não ser declarado na interface do serviço não fica acessível do exterior e é considerado privado. Este método é executado no momento em que é feito o *bind* do *provider* à interface do serviço e fica a executar numa *thread* em *background*. Esta *thread* é responsável por iterar de forma síncrona a lista de cidades existentes para serem monitorizadas e preencher a *store* com as novas temperaturas lidas. Ainda no Anexo A.3, na linha 15 é criada a lista de cidades de forma a que esta tenha capacidades de sincronismo, na iteração da mesma também é utilizado um *synchronized*, (linha 37) permitindo assim que os acessos à lista sejam atómicos. As leituras das cidades a serem monitorizadas são feitas de cinco em cinco segundos, como é possível verificar na linha 50, a *thread* que executa em *background* é adormecida durante cinco segundos a cada iteração.

Por fim, foi criado um cliente que é o responsável pelos *inputs* e *outputs* da aplicação. Este permite receber 4 comandos, nomeadamente o *add*, *all*, *avg* e *quit*. Todos os comandos fazem invocações no serviço *TemperatureCenter*, à excepção do *quit* que termina a execução da aplicação. O *add* invoca o `startCityListener`, o *all* invoca o `getCityTemperatures` e converte a lista de temperaturas para texto e o *avg* que invoca o `getCityAVG`. Ao estilo a aplicação anterior, esta usa o serviço do sistema *OutStream* para imprimir os resultados da aplicação para o utilizador.

Na Listagem 6.7 está ilustrado um exemplo de execução efectuado com esta aplicação. Os *inputs* dados pelo utilizador estão representados a verde e os *outputs* da aplicação a preto regular. A execução foi sequencial e sem grandes intervalos de tempo entre cada instrução. Foram adicionas 4 cidades e após isto foi pedido para cada cidade a listagem das temperaturas lidas e a média das mesmas.

```
1 package applications.temperatureCenter;
2 
3 service Weather: "http://www.webservicex.com/globalweather.asmx?WSDL";
4 service zib.library.OutStream: system;
5 service utils.KeyValueStore: utils.providers.KeyValueStoreProvider();
6 service TemperatureCenter: applications.temperatureCenter.TemperatureCenterProv
      ();
7 service utils.XMLParser: utils.providers.XMLParserProvider();
8 client: Client();
```

Listagem 6.5: Ficheiro de lançamento da aplicação *Temperature Center*

```
package applications.temperatureCenter;
import java.util.Collection;
service TemperatureCenter {
  void startCityListener(String cityName, String countryName);
  Collection<Integer> getCityTemperatures(String cityName, String countryName);
  double getCityAVG(String cityName, String countryName);
}
```

Listagem 6.6: Interface do serviço *Temperature Center*

```
Options:
add [city] [country]     - Start listen city
all [city] [country]     - Get all city temperatures
avg [city] [country]     - Get average of a city temperatures
quit         - Exit application
add Lajes Portugal
Lajes added successfully!
add Milan Italy
Milan added successfully!
add Porto Portugal
Porto added successfully!
add Hermosillo Mexico
Hermosillo added successfully!
all Lajes Portugal
Temperatures list of Lajes:
|17|17|17|17|17|16|16|16|16|16|16|16|16|16|16|16|16|16|16|16|16|16|16|16|16|
avg Lajes Portugal
Average temperature of Lajes:
16.2
all Milan Italy
Temperatures list of Milan:
|23|23|23|23|23|23|23|23|23|23|23|23|23|23|23|23|23|23|23|
avg Milan Italy
Average temperature of Milan:
23.0
all Porto Portugal
Temperatures list of Porto:
|18|18|18|18|18|18|18|18|18|18|18|18|18|18|18|18|
avg Porto Portugal
Average temperature of Porto:
18.0
all Hermosillo Mexico
Temperatures list of Hermosillo:
|39|39|39|39|39|39|39|39|39|39|39|39|
avg Hermosillo Mexico
Average temperature of Hermosillo:
39.0
quit
```

Listagem 6.7: Exemplo de execução da aplicação *Temperature Center*

6.3 Aplicação *Ordered Temperature Center*

Esta aplicação é um extensão da aplicação anterior (Secção 6.2), sendo que as principais actualizações efectuadas são que o serviço *TemperatureCenter* agora tem mais duas operações, o `getCityMinimum(String cityName, String countryname)` e o `getCityMaximum(String cityName, String countryname)`, em que a primeira devolve a temperatura mínima guardada na *store* e a segunda a temperatura máxima respectivamente.

No que diz respeito aos *providers*, no *TemperatureCenterProv* que está ilustrado no Anexo A.5 foram implementadas as duas novas operações já referidas, no método `run()` que executa a *thread* de leitura de temperaturas em *background* as leituras passaram a ser efectuadas de hora a hora em vez de serem de cinco em cinco segundos e as restantes operações mantiveram-se iguais. Foi também implementado um segundo *provider* para o serviço *TemperatureCenter*, cujo nome é *OrderedTemperatureCenterProv* e está ilustrado no Anexo A.6. Este último *provider* difere do primeiro apenas na operação `getCityTemperatures(String cityName, String countryName)`, pois devolve a lista de todas as temperaturas lidas de forma ordenada.

O cliente também sofreu algumas alterações nesta aplicação. Passou a ter suporte para receber os comandos *min* e *max* que invocam as duas operações já referidas, o *min* invoca a `getCityMinimum` e o *max* a `getCityMaximum`. No comando *all* foi adicionada a opção de o utilizador introduzir a opção *ordered* como está ilustrado na linha 18 do exemplo de execução da aplicação *Ordered Temperature Center* na Listagem 6.10. No caso de o utilizador utilizar a opção *ordered* é feito um *bind* dinâmico ao serviço *TemperatureCenter*, sendo substituído o *provider* para o *OrderedTemperatureCenterProv*, que por sua vez irá devolver a listagem de leituras de forma ordenada. No exemplo da Listagem 6.10 é possível verificar a diferença entres os resultados gerados para o comando *all* com e sem a opção *ordered* (Linha 15 e 18).

Por fim, como já foi referido existe um exemplo de execução desta aplicação na Listagem 6.10, onde tal como no último exemplo (Listagem 6.7) os *inputs* do utilizador estão representados a verde e os *outputs* da aplicação a preto regular. Os intervalos de inserção dos comandos durante este exemplo sofreram um intervalo de treze horas entre o comando que está na linha 13 e o que está na linha 15, portanto foram adicionadas três cidades para serem monitorizadas pelo *TemperatureCenter* e só foram feitas as leituras passadas treze horas e com resultados satisfatórios.

```
1  package applications.orderedTemperatureCenter;
2  import java.util.Collection;
3
4  service TemperatureCenter {
5    void startCityListener(String cityName, String countryName);
6    Collection<Integer> getCityTemperatures(String cityName, String countryName);
7    double getCityAVG(String cityName, String countryName);
8    int getCityMinimum(String cityName, String countryname);
9    int getCityMaximum(String cityName, String countryname);
10 }
```

Listagem 6.9: Interface do serviço *Temperature Center* da aplicação *Ordered Temperature Center*

```
1  package applications.orderedTemperatureCenter;
2
3  service Weather :  "http://www.webservicex.com/globalweather.asmx?WSDL";
4  service zib.library.OutStream : system;
5  service utils.KeyValueStore : utils.providers.KeyValueStoreProvider();
6  service TemperatureCenter : applications.orderedTemperatureCenter.
       TemperatureCenterProv();
7  service utils.XMLParser : utils.providers.XMLParserProvider();
8  client: Client();
```

Listagem 6.8: Ficheiro de lançamento da aplicação *Ordered Temperature Center*

```
1  Options:
2  add [city] [country]         - Start listen city
3  all [ordered] [city] [country]  - Get all city temperatures
4  avg [city] [country]         - Get average of a city temperatures
5  min [city] [country]         - Get minimum of a city temperatures
6  max [city] [country]         - Get maximum of a city temperatures
7  quit                 - Exit application
8
9  add Lisboa Portugal
10 Lisboa added successfully!
11 add Dresden Germany
12 Dresden added successfully!
13 add Madrid Spain
14 Madrid added successfully!
15 all Lisboa Portugal
16 Temperatures list of Lisboa:
17 |18|19|20|21|20|20|21|20|20|19|18|17|17|
18 all ordered Lisboa Portugal
19 Temperatures list of Lisboa:
20 |17|17|18|18|19|19|20|20|20|20|20|21|21|
21 avg Lisboa Portugal
22 Average temperature of Lisboa:
23 19.2
24 min Lisboa Portugal
25 Minimum temperature of Lisboa:
```

```
26 17
27 max Lisboa Portugal
28 Maximum temperature of Lisboa:
29 21
30 all Dresden Germany
31 Temperatures list of Dresden:
32 |23|24|26|27|27|28|28|28|28|28|27|25|24|
33 all ordered Dresden Germany
34 Temperatures list of Dresden:
35 |23|24|24|25|26|27|27|27|28|28|28|28|28|
36 avg Dresden Germany
37 Average temperature of Dresden:
38 26.4
39 min Dresden Germany
40 Minimum temperature of Dresden:
41 23
42 max Dresden Germany
43 Maximum temperature of Dresden:
44 28
45 all Madrid Spain
46 Temperatures list of Madrid:
47 |20|22|22|23|25|25|27|28|27|27|27|25|23|
48 all ordered Madrid Spain
49 Temperatures list of Madrid:
50 |20|22|22|23|23|25|25|25|27|27|27|27|28|
51 avg Madrid Spain
52 Average temperature of Madrid:
53 24.7
54 min Madrid Spain
55 Minimum temperature of Madrid:
56 20
57 max Madrid Spain
58 Maximum temperature of Madrid:
59 28
60 quit
```

Listagem 6.10: Exemplo de execução da aplicação *Ordered Temperature Center*

Conclusões

Os modelos de programação dos dias de hoje não facilitam a integração de serviços e lançamento de aplicações ao mais alto nível e de uma forma simples. Neste contexto, o trabalho desenvolvido no âmbito desta dissertação foi um contributo visto que no final os resultados foram satisfatórios e as contribuições a que nos propusemos foram cumpridas.

O tradutor que gera os ficheiros de lançamento e composição de aplicações ficou funcional e permite que as aplicações sejam construídas de uma forma mais simples e com mais automatismos. No trabalho relacionado com a implementação de novas operações para os serviços Zib, ficou implementada uma boa base de suporte para trabalho futuro em ambientes distribuídos, nomeadamente todas as reconfigurações dinâmicas a que nos propusemos foram implementadas e concluídas com uma obtenção de resultados satisfatórios. Por fim, foram demonstrados exemplos de aplicações compostas por vários serviços e com propriedades dinâmicas que permitiram contribuir para uma boa ilustração e clarificação do resultado final de todo o trabalho desenvolvido.

7.1 Trabalho futuro

O trabalho desenvolvido foi satisfatório e tem potencialidades para servir como base para trabalho futuro tanto ao nível da linguagem responsável pelo lançamento de aplicações Zib, bem como ao nível do sistema de execução.

7.1.1 Linguagem de Construção e Lançamento de Aplicações Zib

No contexto desta linguagem há trabalho que pode vir a ser estendido sobre o já realizado no decorrer desta dissertação, nomeadamente ser adicionado suporte para quando

o programador está a construir uma aplicação composta por vários serviços, poder especificar que determinado serviço da aplicação vai ser exposto para o exterior como serviço, bastando para tal o programador utilizar uma palavra reservada que desencadeasse este processo e especificar o porto por onde o serviço iria ficar acessível. Outra funcionalidade que pode ser tomada em conta para trabalho futuro é adicionar suporte para que sejam adicionados explicitamente *providers* quando é feita a associação entre um serviço e a sua implementação para no caso de o *provider* associado falhar, um dos *providers* em *failover* poder ser substituído em tempo de execução, evitando o cenário em que serviços de uma dada aplicação possam terminar a sua execução de forma inesperada.

7.1.2 Sistema de execução

Apesar de todas as operações de reconfiguração adicionadas aos serviços Zib por intermédio do sistema de execução apresentarem resultados satisfatórios, houve certas particularidades que não foram implementadas mas que podem vir a ser tomadas em consideração em trabalhos futuros, nomeadamente na operação de substituição de um *provider*, permitir no caso de existir sessão estabelecida, que esta passe de um *provider* para o outro sem que seja reiniciada. No fim deste trabalho conseguimos criar aplicações com diferentes serviços e até mesmo aplicações com serviços provenientes da WEB. Contudo não foi criado suporte para que sejam construídas aplicações com serviços distribuídos por diferentes nós Elina. Esta última funcionalidade também seria um trabalho interessante a ter em consideração no futuro e que iria dar origem a novos desafios.

Bibliografia

[1] A. Barros, M. Dumas e P. Oaks. "A Critical Overview of the Web Services Choreography Description Language (WS-CDL)". Em: (2005). URL: http://www.bptrends.com/publicationfiles/03-05WPWS-CDLBarrosetal.pdf.

[2] A. Barros, M. Dumas e P. Oaks. "Standards for Web Service Choreography and Orchestration: Status and Perspectives". Em: *Proceedings of the Third International Conference on Business Process Management*. BPM'05. Nancy, France: Springer-Verlag, 2006, pp. 61–74. ISBN: 3-540-32595-6, 978-3-540-32595-6. DOI: 10.1007/11678564_7. URL: http://dx.doi.org/10.1007/11678564_7.

[3] D. Bonetta, A. Peternier, C. Pautasso e W. Binder. "S". Em: *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming - PPoPP '12*. Vol. 47. 8. New York, New York, USA: ACM Press, fev. de 2012, p. 97. URL: http://dl.acm.org/citation.cfm?id=2145816.2145829.

[4] S. De Labey, M. van Dooren e E. Steegmans. "ServiceJ A Java Extension for ProgrammingWeb Services Interactions". Em: *Web Services, 2007. ICWS 2007. IEEE International Conference on*. 2007, pp. 505–512. DOI: 10.1109/ICWS.2007.161.

[5] M. Di Penta, R. Esposito, M. L. Villani, R. Codato, M. Colombo e E. Di Nitto. "WS Binder". Em: *Proceedings of the 2006 international workshop on Service-oriented software engineering - SOSE '06*. New York, New York, USA: ACM Press, mai. de 2006, p. 74. URL: http://dl.acm.org/citation.cfm?id=1138486.1138502.

[6] J. L. Fiadeiro e A. Lopes. "A model for dynamic reconfiguration in service-oriented architectures". Em: *Software & Systems Modeling* 12.2 (fev. de 2012), pp. 349–367. ISSN: 1619-1366. URL: http://link.springer.com/10.1007/s10270-012-0236-1.

[7] N. Kavantzas, T. Fletcher, D. Burdett, Y. Lafon, C. Barreto e G. Ritzinger. *Web Services Choreography Description Language Version 1.0*. Candidate Recommendation. http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/. W3C, nov. de 2005.

[8] V. Kodaganallur. “Incorporating language processing into java applications: A JavaCC tutorial”. Em: *Software, IEEE* 21.4 (2004), pp. 70–77.

[9] S. D. Labey, M. V. Dooren e E. Steegmans. “Bridging the Gap Between Object-Oriented Programming and Service Oriented Computing”. Em: *In Proceedings of the 1st International Workshop on Foundations of Service Oriented Architecture, 2007. International Extended Abstracts*. 2007.

[10] R. G. Lavender e D. C. Schmidt. *Active Object – An Object Behavioral Pattern for Concurrent Programming*. 1996.

[11] Matjaz B. Juric. *A Hands-on Introduction to BPEL*. URL: http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html.

[12] J. Mendling e M. Hafner. “From WS-CDL choreography to BPEL process orchestration”. Em: *Journal of Enterprise Information Management* 21.5 (2008), pp. 525–542. ISSN: 1741-0398. URL: http://mendling.com/publications/TR06-CDL.pdf.

[13] F. Montesi, C. Guidi e G. Zavattaro. “Service-Oriented Programming with Jolie”. English. Em: *Web Services Foundations*. Ed. por A. Bouguettaya, Q. Z. Sheng e F. Daniel. Springer New York, 2014, pp. 81–107. ISBN: 978-1-4614-7517-0. DOI: 10.1007/978-1-4614-7518-7_4. URL: http://dx.doi.org/10.1007/978-1-4614-7518-7_4.

[14] N. Nystrom, M. Clarkson e A. Myers. “Polyglot: An Extensible Compiler Framework for Java”. Em: *Compiler Construction*. Ed. por G. Hedin. Vol. 2622. Lecture Notes in Computer Science. Berlin / Heidelberg: Springer, 2003, pp. 138–152. ISBN: 978-3-540-00904-7. DOI: 10.1007/3-540-36579-6_11. URL: http://dx.doi.org/10.1007/3-540-36579-6_11.

[15] D. Orchard, F. McCabe, E. Newcomer, H. Haas, C. Ferris, D. Booth e M. Champion. *Web Services Architecture*. W3C Note. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/. W3C, fev. de 2004.

[16] M. P. Papazoglou, P. Traverso, S. Dustdar e F. Leymann. “Service-Oriented Computing: State of the Art and Research Challenges”. Em: *Computer* 40.11 (nov. de 2007), pp. 38–45. ISSN: 0018-9162. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4385255.

[17] M. Papazoglou. “Service-oriented computing: concepts, characteristics and directions”. Em: *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*. 2003, pp. 3–12. DOI: 10.1109/WISE.2003.1254461.

[18] H. Paulino, A. M. Dias e J. C. Cunha. *A Service Centred Approach to Concurrent and Parallel Computing*. Rel. téc. CITI/Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2012.

[19] J. Saramago, D. Mourao e H. Paulino. "Towards an Adaptable Middleware for Parallel Computing in Heterogeneous Environments". Em: *Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*. 2012, pp. 143–151. DOI: 10.1109/ClusterW.2012.36.

[20] X. Wang, L. Xiao, W. Li 0008 e Z. Xu. "Abacus: A Service-Oriented Programming Language for Grid Applications." Em: *IEEE SCC*. 2005, pp. 225–232. URL: http://dblp.uni-trier.de/db/conf/IEEEscc/scc2005.html\#WangXLX05.

[21] H. Zhu. *From OOP to SOP: What Improved?* URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.124.2449.

Anexos

A.1 Classe *WebServiceGenerator*

```
package zib.runtime;
import japa.parser.ParseException;
import java.io.File;
import java.io.IOException;
import java.net.URL;
import zib.runtime.wsimport.WSImport;
import zib.runtime.wsimport.ZibifyClientWS;

public class WebServiceGenerator implements service.WebServiceStubGenerator {
  private String baseFolder;
  private String packageName;
  private String serviceName;

  public WebServiceGenerator(String baseFolder, String packageName, String
      serviceName){
    this.baseFolder   = baseFolder;
    this.packageName  = packageName;
    this.serviceName  = serviceName;
  }
  public <S> S getStub(String url) throws ParseException, IOException,
      ClassNotFoundException {
    /***********************************
     * WSIMPORT WEBSERVICE
     ***********************************/
      WSImport wsi      = new WSImport(url, this.baseFolder, this.packageName);
      String webServiceDir  = wsi.getWebServiceDir();
```

```
25
26
27
28      /************************************
29       * ZIBIFY WEBSERVICE CLIENT CODE
30       ***********************************/
31
32        ZibifyClientWS<S> zibws = new ZibifyClientWS<S>(webServiceDir,
            serviceName, this.packageName, wsi.getWebServicePackage());
33
34
35        /*********************************
36         * Check if is to update webstub
37         ********************************/
38          File zibFile = new File(webServiceDir+File.separator+".."+File.
              separator+serviceName+".zib");
39          File webServiceDirFile = new File(webServiceDir);
40
41          //If the webservice client was edited we need to update the webservice
              stub
42          if(!webServiceDirFile.exists() || zibFile.lastModified() >
              webServiceDirFile.lastModified()){
43            wsi.importClient();
44            zibws.init();
45          }
46
47      /************************************
48       * RETURN GENERATED WEBSERVICE STUB
49       ***********************************/
50        return zibws.getWebServiceStub(new URL(url));
51    }
52  }
```

A.2 Classe *WSImport*

```
package zib.runtime.wsimport;
import java.io.File;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.apache.tools.ant.Project;
import org.apache.tools.ant.ProjectHelper;

public class WSImport {
  private static final String WSDL_REGEX  = "https?:\\/\\/?[\\da-z\\.-]+\\.[a-z
      \\.]{2,6}[\\/\\w␣\\.-]*\\/(\\w+).asmx\\?(?:W|w)(?:S|s)(?:D|d)(?:L|l)";
  private static final String ANT_FILE = "/build-wsimport.xml";
  private static final String WS_STUB_PACKAGE_SUFIX = "WsStub";
  private String wsdlURL;
  private String webServiceName;
  private String package_path;
  private String outputFolder;
  private String packageName;

  public WSImport(String wsdlURL, String outputFolder, String packageName)
      throws MalformedURLException, IOException  {
    this.wsdlURL    = wsdlURL;
    this.outputFolder   = outputFolder;
    this.packageName  = packageName;

    if (!this.isValidURL())
      throw new MalformedURLException("Not␣a␣valid␣WSDL␣location");

    this.setWebServiceName();
    this.setPackage();
  }
  private void setPackage() {
    this.package_path = packageName.replaceAll("/", ".") + "." +this.
        webServiceName.toLowerCase() + WS_STUB_PACKAGE_SUFIX ;
  }
  private void setWebServiceName(){
    Pattern pattern = Pattern.compile(WSDL_REGEX);
        Matcher matcher = pattern.matcher(this.wsdlURL);
        matcher.find();
        this.webServiceName = matcher.group(1).toLowerCase();
  }
  private boolean isValidURL(){
    return this.wsdlURL.matches(WSDL_REGEX);
  }
  private void deleteOldClient(){
```

```
46      File oldClient = new File(this.getWebServiceDir());
47      if(oldClient.isDirectory()){
48        for (File file : oldClient.listFiles()) file.delete();
49        oldClient.delete();
50      }
51    }
52    public void importClient() throws MalformedURLException, IOException {
53      //Delete the old client
54      this.deleteOldClient();
55      // Check if URL is reachable
56      URLConnection c = new URL(this.wsdlURL).openConnection();
57      c.connect();
58      //Start the import
59      File buildFile = new File(WSImport.class.getResource(ANT_FILE).getFile());
60      Project p = new Project();
61      p.setUserProperty("ant.file", buildFile.getAbsolutePath());
62      p.init();
63      ProjectHelper helper = ProjectHelper.getProjectHelper();
64      p.addReference("ant.projectHelper", helper);
65      helper.parse(p, buildFile);
66      p.setProperty("wsdl.location", this.wsdlURL);
67      p.setProperty("wsimport.package", this.package_path);
68      p.setProperty("wsimport.source", this.outputFolder);
69      p.executeTarget("wsimport");
70    }
71    public String getWebServiceName(){
72      return Character.toUpperCase(this.webServiceName.charAt(0)) + this.
            webServiceName.substring(1);
73    }
74    public String getWebServiceDir(){
75      return this.outputFolder+ File.separator +this.package_path.replace(".",
            File.separator)+File.separator;
76    }
77    public String getWebServicePackage(){
78      return this.package_path;
79    }
80  }
```

A.3 Implementação do serviço *Temperature Center* da aplicação *Temperature Center*

```
1  package applications.temperatureCenter;
2  import utils.KeyValueStore;
3  import utils.XMLParser;
4  import java.util.Collection;
5  import java.util.Collections;
6  import java.util.List;
7  import java.util.ArrayList;
8  import java.lang.InterruptedException;
9  import java.util.Iterator;
10
11
12 provider TemperatureCenterProv implements TemperatureCenter, Runnable{
13
14   static KeyValueStore.session<String, Integer> kv = newsession<String, Integer
       > KeyValueStore("Weather");
15   static List<String> cities = Collections.synchronizedList(new ArrayList<
       String>());
16   synchronized void startCityListener(String cityName, String countryName){
17     cityName = cityName.toLowerCase();
18     countryName = countryName.toLowerCase();
19     if(!cities.contains(cityName+"_"+countryName))
20       cities.add(cityName+"_"+countryName);
21   }
22
23   synchronized Collection<Integer> getCityTemperatures(String cityName, String
       countryName){
24     return kv.getAll(cityName.toLowerCase()+"_"+countryName.toLowerCase());
25   }
26
27   synchronized double getCityAVG(String cityName, String countryName){
28     Collection<Integer> temperatures = getCityTemperatures(cityName,
         countryName);
29     int total = 0;
30     for(Integer i : temperatures)
31       total += i;
32     return total/temperatures.size();
33   }
34
35   void run() {
36     while(true){
37       synchronized(cities) {
38         Iterator<String> i = cities.iterator();
39         while (i.hasNext()){
40           String cityCountry = i.next();
41           String[] cityCountryArr = cityCountry.split("_");
```

```
42          String _weather = Weather.getWeather(cityCountryArr[0],
                cityCountryArr[1]);
43          XMLParser.session xml = newsession XMLParser(_weather);
44          String temperature = xml.getValue("Temperature");
45          int celcius = Integer.parseInt(temperature.substring(temperature.
                indexOf("C") - 3, temperature.indexOf("C")).trim());
46          kv.put(cityCountry, celcius);
47        }
48      }
49      try{
50        Thread.sleep(5000);
51      }
52      catch(InterruptedException e){
53
54      }
55    }
56  }
57 }
```

A.4 *WeatherWebServiceStub* gerado para o serviço WEB *Weather* da aplicação *Temperature Center*

```
1  package applications.temperatureCenter.globalweatherWsStub;
2  import java.net.URL;
3  import java.util.UUID;
4  import service.IFuture;
5  import service.NoSuchMethodException;
6  import core.Level;
7  import core.taskManager.InvokeHandlerTask;
8  import drivers.Adapters;
9  import drivers.TaskExecutorDriver;
10
11 public class WeatherWebServiceStub extends service.ServiceStub<applications.
       temperatureCenter.Weather> implements GlobalWeatherSoap {
12
13     private GlobalWeatherSoap stub;
14
15     private static UUID WebService = UUID.randomUUID();
16
17     TaskExecutorDriver taskExecutor;
18
19     public WeatherWebServiceStub(URL url) {
20         super(WebService, WebService, Level.Cluster);
21         this.stub = new GlobalWeather().getGlobalWeatherSoap();
22         this.taskExecutor = Adapters.getTaskExecutor();
23         this.setService(this.stub);
24     }
25
```

```
26      public String getWeather(String cityName, String countryName) {
27          return this.service.getWeather(cityName, countryName);
28      }
29
30      public String getCitiesByCountry(String countryName) {
31          return this.service.getCitiesByCountry(countryName);
32      }
33
34      public <R> IFuture<R> invoke(String methodName, Object[] args, Class<?>[]
            types) throws NoSuchMethodException {
35          return this.invoke(methodName, args);
36      }
37
38      public <R> IFuture<R> invoke(String methodName, Object[] args) throws
            NoSuchMethodException {
39          if (methodName.equals("getWeather") || methodName.equals("
                getCitiesByCountry")) {
40              if (args == null) args = new Object[]{};
41              InvokeHandlerTask<R> task = new InvokeHandlerTask<R>(this,
                    methodName, args);
42              return this.taskExecutor.execute(task);
43          }
44          throw new NoSuchMethodException(methodName);
45      }
46  }
```

A.5 Implementação do serviço *Temperature Center* da aplicação *Ordered Temperature Center*

```
1   package applications.orderedTemperatureCenter;
2   import utils.KeyValueStore;
3   import utils.XMLParser;
4   import java.util.Collection;
5   import java.util.Collections;
6   import java.util.List;
7   import java.util.ArrayList;
8   import java.lang.InterruptedException;
9   import java.util.Iterator;
10
11  provider TemperatureCenterProv implements TemperatureCenter, Runnable{
12
13    static KeyValueStore.session<String, Integer> kv = newsession<String, Integer
        > KeyValueStore("Weather");
14    static List<String> cities = Collections.synchronizedList(new ArrayList<
        String>());
15    synchronized void startCityListener(String cityName, String countryName){
16      cityName = cityName.toLowerCase();
17      countryName = countryName.toLowerCase();
```

```
18      if(!cities.contains(cityName+"_"+countryName))
19        cities.add(cityName+"_"+countryName);
20    }
21
22    synchronized Collection<Integer> getCityTemperatures(String cityName, String
          countryName){
23      return kv.getAll(cityName.toLowerCase()+"_"+countryName.toLowerCase());
24    }
25
26    synchronized double getCityAVG(String cityName, String countryName){
27      Collection<Integer> temperatures = getCityTemperatures(cityName,
            countryName);
28      int total = 0;
29      for(Integer i : temperatures)
30        total += i;
31      return total/temperatures.size();
32    }
33
34    synchronized int getCityMinimum(String cityName, String countryName){
35      Collection<Integer> temperatures = getCityTemperatures(cityName,
            countryName);
36      int minimum = 1000;
37      for(Integer i : temperatures)
38        if(i < minimum) minimum = i;
39      return minimum;
40    }
41
42    synchronized int getCityMaximum(String cityName, String countryName){
43      Collection<Integer> temperatures = getCityTemperatures(cityName,
            countryName);
44      int maximum = -1000;
45      for(Integer i : temperatures)
46        if(i > maximum) maximum = i;
47      return maximum;
48    }
49
50    void run() {
51      while(true){
52        synchronized(cities) {
53          Iterator<String> i = cities.iterator();
54          while (i.hasNext()){
55            String cityCountry = i.next();
56            String[] cityCountryArr = cityCountry.split("_");
57            String _weather = Weather.getWeather(cityCountryArr[0],
                  cityCountryArr[1]);
58            XMLParser.session xml = newsession XMLParser(_weather);
59            String temperature = xml.getValue("Temperature");
60            int celcius = Integer.parseInt(temperature.substring(temperature.
                  indexOf("C") - 3, temperature.indexOf("C")).trim());
61            kv.put(cityCountry, celcius);
```

```
62          }
63        }
64        try{
65          Thread.sleep(3600000);
66        }
67        catch(InterruptedException e){
68        }
69      }
70    }
71  }
```

A.6 Implementação com listagem ordenada do serviço *Temperature Center* da aplicação *Ordered Temperature Center*

```
1   package applications.orderedTemperatureCenter;
2   import utils.KeyValueStore;
3   import utils.XMLParser;
4   import java.util.Collection;
5   import java.util.Collections;
6   import java.util.List;
7   import java.util.ArrayList;
8   import java.lang.InterruptedException;
9   import java.util.Iterator;
10
11  provider OrderedTemperatureCenterProv implements TemperatureCenter, Runnable{
12
13    static KeyValueStore.session<String, Integer> kv = newsession<String, Integer
        > KeyValueStore("Weather");
14    static List<String> cities = Collections.synchronizedList(new ArrayList<
        String>());
15    synchronized void startCityListener(String cityName, String countryName){
16      cityName = cityName.toLowerCase();
17      countryName = countryName.toLowerCase();
18      if(!cities.contains(cityName+"_"+countryName))
19        cities.add(cityName+"_"+countryName);
20    }
21
22    synchronized Collection<Integer> getCityTemperatures(String cityName, String
        countryName){
23      List<Integer> temperatures = new ArrayList(kv.getAll(cityName.toLowerCase()
          +"_"+countryName.toLowerCase()));
24      Collections.sort(temperatures);
25      return temperatures;
26    }
27
28    synchronized double getCityAVG(String cityName, String countryName){
29      Collection<Integer> temperatures = getCityTemperatures(cityName,
          countryName);
```

```
30      int total = 0;
31      for(Integer i : temperatures)
32        total += i;
33      return total/temperatures.size();
34    }
35
36    synchronized int getCityMinimum(String cityName, String countryName){
37      Collection<Integer> temperatures = getCityTemperatures(cityName,
            countryName);
38      int minimum = 1000;
39      for(Integer i : temperatures)
40        if(i < minimum) minimum = i;
41      return minimum;
42    }
43
44    synchronized int getCityMaximum(String cityName, String countryName){
45      Collection<Integer> temperatures = getCityTemperatures(cityName,
            countryName);
46      int maximum = -1000;
47      for(Integer i : temperatures)
48        if(i > maximum) maximum = i;
49      return maximum;
50    }
51
52    void run() {
53      while(true){
54        synchronized(cities) {
55          Iterator<String> i = cities.iterator();
56          while (i.hasNext()){
57            String cityCountry = i.next();
58            String[] cityCountryArr = cityCountry.split("_");
59            String _weather = Weather.getWeather(cityCountryArr[0],
                  cityCountryArr[1]);
60            XMLParser.session xml = newsession XMLParser(_weather);
61            String temperature = xml.getValue("Temperature");
62            int celcius = Integer.parseInt(temperature.substring(temperature.
                  indexOf("C") - 3, temperature.indexOf("C")).trim());
63            kv.put(cityCountry, celcius);
64          }
65        }
66        try{
67          Thread.sleep(3600000);
68        }
69        catch(InterruptedException e){
70
71        }
72      }
73    }
74  }
```

A.7 Cliente da aplicação *Temperature Center*

```
package applications.temperatureCenter;
import utils.XMLParser;
import java.util.Scanner;
import zib.library.OutStream;
import java.util.Collection;

client Client {
  void run() {
      OutStream.println("Options:");
      OutStream.println("add [city] [country]\t\t- Start listen city");
      OutStream.println("all [city] [country]\t\t- Get all city temperatures");
      OutStream.println("avg [city] [country]\t\t- Get average of a city temperatures");
      OutStream.println("quit\t\t\t\t- Exit application");
      Scanner sc = new Scanner(System.in);
    while(sc.hasNext()){
      String s = (String)sc.next();
      if(s.equalsIgnoreCase("quit")) {
        System.exit(0);
        }
      else if(s.equalsIgnoreCase("add") || s.equalsIgnoreCase("all") || s.equalsIgnoreCase("avg")){
        String city = ((String)sc.next()).trim();
        String country = ((String)sc.next()).trim();
          if(s.equalsIgnoreCase("add")) {
            TemperatureCenter.startCityListener(city, country);
            OutStream.println(city+" added successfully!");
          }
          else if(s.equalsIgnoreCase("all")) {
            String output = "|";
            Collection<Integer> temperatures = TemperatureCenter.getCityTemperatures(city, country);
          for(Integer i : temperatures)
            output += i + "|";
          OutStream.println("Temperatures list of "+city+":\n"+output);
          }
          else if(s.equalsIgnoreCase("avg")) {
            double avg = TemperatureCenter.getCityAVG(city, country);
            OutStream.println("Average temperature of "+city+":\n"+ avg);
          }
      }
      else
        OutStream.println("Invalid command.");
    }
  }
}
```

A.8 Cliente da aplicação *Ordered Temperature Center*

```
package applications.orderedTemperatureCenter;
import utils.XMLParser;
import java.util.Scanner;
import zib.library.OutStream;
import java.util.Collection;

client Client {
  void run() {
      OutStream.println("Options:");
      OutStream.println("add␣[city]␣[country]\t\t-␣Start␣listen␣city");
      OutStream.println("all␣[ordered]␣[city]␣[country]\t-␣Get␣all␣city␣
          temperatures");
      OutStream.println("avg␣[city]␣[country]\t\t-␣Get␣average␣of␣a␣city␣
          temperatures");
      OutStream.println("min␣[city]␣[country]\t\t-␣Get␣minimum␣of␣a␣city␣
          temperatures");
      OutStream.println("max␣[city]␣[country]\t\t-␣Get␣maximum␣of␣a␣city␣
          temperatures");
      OutStream.println("quit\t\t\t\t-␣Exit␣application\n");
      Scanner sc = new Scanner(System.in);
      boolean ordered = false;
    while(sc.hasNext()){
      String s = (String)sc.next();
      if(s.equalsIgnoreCase("quit")) {
        System.exit(0);
        }
      else if(s.equalsIgnoreCase("add") || s.equalsIgnoreCase("all") || s.
          equalsIgnoreCase("avg") ||
          s.equalsIgnoreCase("min") || s.equalsIgnoreCase("max")){
        String city = ((String)sc.next()).trim();
        if(city.equalsIgnoreCase("ordered")){
          ordered = true;
          city = ((String)sc.next()).trim();
        }
        String country = ((String)sc.next()).trim();
          if(s.equalsIgnoreCase("add")) {
            TemperatureCenter.startCityListener(city, country);
            OutStream.println(city+"␣added␣successfully!");
          }
          else if(s.equalsIgnoreCase("all")) {
            if(ordered)
              TemperatureCenter.replace(new OrderedTemperatureCenterProv());
            String output = "|";
            Collection<Integer> temperatures = TemperatureCenter.
                getCityTemperatures(city, country);
          for(Integer i : temperatures)
            output += i + "|";
          OutStream.println("Temperatures␣list␣of␣"+city+":\n"+output);
```

```
43            }
44            else if(s.equalsIgnoreCase("avg")) {
45              double avg = TemperatureCenter.getCityAVG(city, country);
46              OutStream.println("Average␣temperature␣of␣"+city+":\n"+ avg);
47            }
48            else if(s.equalsIgnoreCase("min")) {
49              int min = TemperatureCenter.getCityMinimum(city, country);
50              OutStream.println("Minimum␣temperature␣of␣"+city+":\n"+ min);
51            }
52            else if(s.equalsIgnoreCase("max")) {
53              double max = TemperatureCenter.getCityMaximum(city, country);
54              OutStream.println("Maximum␣temperature␣of␣"+city+":\n"+ max);
55            }
56        }
57        else
58          OutStream.println("Invalid␣command.");
59        if(ordered){
60          ordered = false;
61          TemperatureCenter.replace(new TemperatureCenterProv());
62        }
63      }
64    }
65  }
```