Regent: Uma Arquitectura para a Evolução de Micro-Serviços

Álvaro António Santos[1], Mário Pereira[1], João Leitão[1], and João Costa Seco[1]

NOVA LINCS, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Caparica, Portugal
aa.santos@campus.fct.unl.pt
{mjp.pereira,jc.leitao,joao.seco}@fct.unl.pt

Resumo As arquitecturas de micro-serviços são uma escolha popular para a concepção de sistemas distribuídos altamente escaláveis, disponíveis e modulares. Mas, apesar da sua popularidade, este tipo de arquitectura levanta os seus próprios desafios — em particular, a dificuldade de garantir que a evolução de um micro-serviço não leva à quebra da aplicação ao violar as dependências de outros micro-serviços.
Neste artigo apresentamos uma arquitectura de suporte à evolução *correcta* e *robusta* de micro-serviços. Por robusta, entendemos uma evolução que não compromete a correcção e disponibilidade das funcionalidades já existentes; por correcta, entendemos uma evolução cuja verificação de compatibilidade com o sistema actual é baseada em princípios formais. Contrariamente a outros métodos baseados em testes de integração, a nossa abordagem consegue oferecer garantias de correcção ao alavancar em métodos formais baseados em sistemas de tipos.
Neste artigo, apresentamos um protótipo inicial da solução, que recorre a uma componente central para realizar a verificação de código de forma estática em tempo de *deployment*, recorrendo a adaptadores de código automaticamente gerados quando necessário. Resultados preliminares da avaliação validam a solução, demonstrando um custo operacional adicional modesto.

1 Introdução

As arquitecturas de micro-serviços são uma escolha popular para a criação de sistemas distribuídos. Embora a definição exacta de micro-serviço seja contenciosa, é geralmente aceite que se tratam de unidades mutuamente desacopladas e substituíveis, que implementam um pequeno conjunto de funcionalidades bem definidas, e que comunicam entre si através da troca de mensagens em rede [15,5,16,12]. São estas características que tornam os sistemas construídos segundo arquitecturas de micro-serviços altamente escaláveis, disponíveis, e modulares: modificar um destes sistemas implica apenas a substituição de algumas componentes, ao contrário do caso monolítico em que qualquer modificação requer a substituição de todo o monólito.

Uma consequência natural desta modularidade é que se torna possível alocar equipas a trabalhar em micro-serviços distintos, de forma independente e em paralelo. No entanto, estas estratégias de desenvolvimento que advêm da natureza modular dos sistemas, ainda que com potencial para permitirem ciclos de evolução de software mais curtos, ampliam um problema crucial no desenvolvimento de software: como garantir que alterações a uma componente não quebram outra (particularmente, quando as alterações podem ser feitas em paralelo).

Essencialmente, enquanto que num sistema monolítico o ambiente de desenvolvimento (verificador de sintaxe, *linter*, *linker*, compilador) é capaz de detectar a maioria das incompatibilidades entre os vários módulos — em parte graças à maior homogeneidade de tecnologias usadas, em parte graças a todas as componentes estarem sob o seu controlo —, não existem equivalentes directos para sistemas em que cada componente é relativamente independente e opaca (i.e., cujo funcionamento interno é desconhecido ao nosso ambiente local), desenvolvida com tecnologias potencialmente diferentes e em que o padrão de comunicação entre cada par de componentes pode ser completamente único.

No entanto, o problema de garantir a evolução robusta de um sistema — e por "robusta", entenda-se "sem pôr em causa o bom funcionamento actual" — implica saber aquilo que está a ser objecto da nossa evolução. Esta questão leva-nos a um segundo problema associado às arquitecturas de micro-serviços: embora existam mecanismos de especificação do comportamento de interfaces (e.g., abordagens à base de contratos, definições de esquemas XML, sistemas de tipos comportamentais), a capacidade de trabalho paralelo e independente proporcionados pelas arquitecturas de micro-serviços não incentivam a sua utilização na criação de componentes e nas suas interacções, mas sim a criação de protocolos/interfaces *ad hoc* e em constante mutação.

Os problemas que identificamos são já conhecidos, e existem tentativas de os atacar. A título de exemplo, referimos a abordagem de integração contínua (commummente utilizada em arquitecturas de micro-serviços), cujo um dos princípios é o de que todas as alterações a um sistema, por ínfimas que sejam, devem ser submetidas a uma bateria de testes (e.g., testes unitários, testes de integração) [10], e as abordagens de multi-versão, em que após a evolução de uma componente se mantêm versões antigas da mesma (ou temporariamente, para que as componentes que dessa dependem se possam adaptar, ou em permanência) [8]. Mas nenhuma destas abordagens é perfeita: qualquer teste unitário ou de integração está limitado a um subconjunto dos casos possíveis — e, se escrito manualmente, está limitado pela criatividade do seu autor —, e os mecanismos de multi-versão requerem mais recursos máquina para manter versões de cada serviço, mais recursos humanos para corrigir problemas em cada versão do serviço e, possivelmente, interacção humana para decidir que versões manter.

A nossa abordagem visa os mesmos problemas (a definição de interfaces e a sua evolução), mas tentar resolvê-los por uma via diferente. Essencialmente, a nossa abordagem assenta em 2 relações formais: uma relação de "compatibilidade" e uma relação de "conversão". Estas relações conferem à nossa abordagem garantias típicas da utilização de métodos formais, ao invés das garantias tipica-

mente associadas a abordagens de testes empíricos e não-exaustivos: enquanto que as abordagens actuais de detecção de incompatibilidades podem apresentar falsos negativos, a nossa abordagem é conservadora, e apenas pode falhar apresentando falsos positivos — isto é uma consequência típica da utilização de técnicas de sistemas de tipos, em que a informação concreta que o programa manipula (valores) é substituída por informação mais abstracta (tipos), resultando numa análise que tem de assumir o pior (o "pior" valor pertencente a um dado tipo), e que é capaz de fornecer uma aproximação dos possíveis estados do programa durante a sua execução.

Através destas duas relações formais, o nosso sistema rejeita alterações que possam pôr em causa o seu bom funcionamento (usando a relação de compatibilidade), e, ao aceitar alterações às interfaces já existentes, gera automaticamente adaptadores (usando a relação de conversão) que transformam os valores trocados entre componentes, de forma a que nenhuma componente tenha de ser manualmente alterada para lidar com os novos valores. Embora o nosso sistema esteja contextualizado nas arquitecturas de micro-serviços, isto deve-se apenas ao facto de estas arquitecturas promoverem por natureza as evoluções rápidas e *ad hoc* de sistemas. Cremos, no entanto, que os princípios que aqui expomos são ortogonais à arquitectura sob a qual o sistema é desenhado, e que as nossas ideias podem facilmente ser aplicadas em outros contextos.

É ainda importante realçar que os principais conceitos subjacentes ao nosso sistema (a verificação de compatibilidade e a geração de adaptadores) não são, por si só, novos. A nossa contribuição com este artigo é explicitar uma forma de organizar todos estes elementos numa arquitectura/sistema/middleware que suporta a evolução, robusta e tão automática quanto possível, do sistema de base e a criação de um protótipo do nosso sistema.

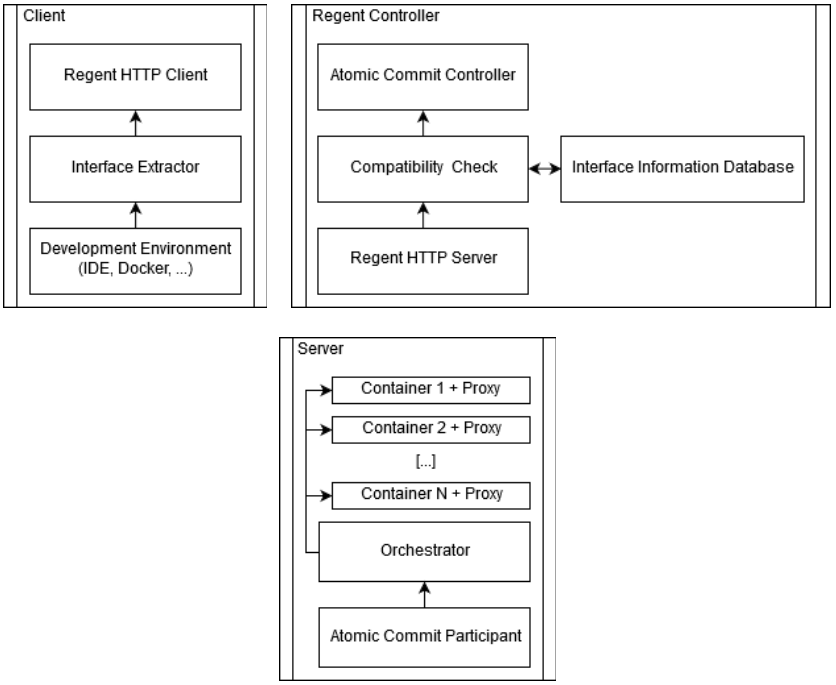
2 Arquitectura e Detalhes Técnicos

A arquitectura que propomos concretiza-se em três tipos de serviços, cuja estrutura é apresentada na Figura 1. Na figura, as setas indicam o sentido principal do fluxo de informação no sistema. Num sistema seguindo a nossa arquitectura, cada cliente/criador de micro-serviços teria uma instância do serviço `Client` a correr localmente, cada máquina física teria uma instância do serviço `Server` a correr localmente, e existiria um controlador central a implementar o serviço `Regent Controller`. Chamamos a atenção para um detalhe crítico: os "serviços" apresentados na Figura 1 não são os "micro-serviços" criados pelos utilizadores do nosso sistema. Os nossos "serviços" são uma camada de gestão para os micro-serviços dos utilizadores, sendo-lhes tão invisíveis quanto possível.

Concretamente, as responsabilidades de cada serviço são:

- **Serviço Client**: tem como função extrair do código-fonte escrito pelo utilizador (posteriormente empacotado num micro-serviço), toda a informação relevante para a definição da interface do serviço. Conceptualmente, a componente "Interface Extractor" deste serviço pode ser compreendida como uma

Figura 1: Estrutura dos serviços do nosso sistema.



função que aceita código-fonte Java, C, C++, (...) e retorna uma especificação de interface agnóstica aos detalhes da linguagem usada para escrever o código-fonte. É importante notar que esta extracção não pode ser completamente automatizada em todos os casos (e.g., extrair informação de tipos numa linguagem dinamicamente tipada), e que o formato das interfaces é facilmente modificável pelo utilizador (como exemplificado na secção 3) para colmatar esta limitação.

Aquando de uma tentativa de *deployment* do utilizador, este serviço envia não só a informação necessária para construir e lançar o micro-serviço — tipicamente, um *container* — para o serviço `Regent Controller`, mas também a informação sobre a interface oferecida por esse micro-serviço.

Este serviço também é responsável por providenciar ao utilizador uma lista das interfaces disponíveis no serviço (i.e., conhecidas pelo serviço `Regent Controller`), para que ele possa descobrir o estado dos outros micro-serviços quando lhe for conveniente, e por mapear os nomes utilizados pelo utilizador para identificadores únicos em todo o sistema. Concretamente, este serviço garante que o utilizador pode, por exemplo, referir-se a uma função remota como `get` mesmo que ela mude de nome para `retrieve`, sendo responsável (em conluio com o serviço `Regent Controller`) por manter um mapeamento de nomes conhecidos localmente para identificadores únicos em todo o sis-

tema e imutáveis.

- **Serviço Regent Controller**: Esta é a componente fulcral da nossa arquitectura, e o seu papel é receber e processar pedidos dos serviços `Client`. Particularmente, a sua tarefa mais importante é o tratamento de pedidos de *deployment* — que requerem a verificação da compatibilidade entre as interfaces dos micro-serviços novos/actualizados e dos já existentes. Caso esta se verifique, o serviço procede à injecção de código capaz de gerar adaptadores nos novos micro-serviços, até que sejam por fim lançados (tipicamente como *containers*) em máquinas reais.
  É ainda de realçar o processo de *deployment* que este serviço efectua: dado que um pedido de *deployment* pode conter múltiplos micro-serviços, possivelmente destinados a correr em máquinas diferentes, e que pode não fazer sentido lançar um dos micro-serviços sem lançar o outro — por exemplo, no caso em que estamos a dar *deploy* a 2 micro-serviços que dependem mutuamente um do outro —, este serviço tem a responsabilidade de coordenar a operação de *deployment* de forma a que ela seja feita atomicamente em todas as máquinas, ou falhe em todas.

- **Serviço Server**: é responsável por gerir a fase de *deployment* nas máquinas físicas, iniciada pelo serviço `Regent Controller`. Essencialmente, este serviço é usado para garantir que o orquestrador de micro-serviços/*containers* não é actualizado com informação incoerente que poderia ter de vir a ser revertida.
  É ainda neste serviço que os adaptadores de código actuam. A sua mecânica base é simples: qualquer micro-serviço que contacte outro usa os seus adaptadores (de forma automática) para garantir que só envia mensagens que o seu destinatário saiba compreender. Dado que os micro-serviços só podem estar a correr no contexto deste serviço se tiverem passado a verificação de compatibilidade do *Regent Controller*, então temos a garantia de que é sempre possível realizar esta adaptação de mensagens.

Exposta a arquitectura, passamos agora a descrever em detalhe o que é a relação de compatibilidade, o que é a relação de conversão, o que constitui "informação relevante" para o extractor de interfaces do serviço `Client` e como representamos esta informação.

A relação de compatibilidade é definida sobre a linguagem que usamos para definir interfaces, e trata-se essencialmente de uma relação clássica de subtipagem. Essencialmente, o que esta relação faz é: para um par de interfaces A e B, em que A consome/depende de algumas funções de B, a relação de compatibilidade verifica que (1) as funções que A crê existirem em B lá existem de facto e (2) os tipos de retorno/aceites pela função que A crê existir são compatíveis com os que existem realmente em B. Por exemplo, supondo que a nossa linguagem de descrição de interfaces é a linguagem `C`, a interface A que depende da função `long f(void)` em B seria compatível, segundo a nossa relação, com a interface B que oferece a função `int f(void)`. No entanto, há que realçar que a relação

de compatibilidade não é estritamente uma relação de subtipagem, podendo ser configurada pelo utilizador para lidar com situações que uma relação clássica de subtipagem rejeitaria (como descrito na secção $3-3^{o}$ *deployment*).

A relação de conversão (que descreve a forma como criamos os nossos adaptadores) assenta num princípio semelhante ao da relação de compatibilidade: se o serviço de interface A está a tentar comunicar com um serviço de interface B, é porque são compatíveis (caso contrário, o serviço `Regent Controller` teria impedido o *deployment* de pelo menos um destes micro-serviços), e portanto os tipos de valores que B espera receber são deriváveis a partir de A.

Há, no entanto, dois aspectos a ter em atenção quanto à relação de conversão: em primeiro lugar, os adaptadores que implementam esta relação têm conhecimento das interfaces cujas comunicações estão a mediar, mas os valores que adaptam existem ao nível dos próprios micro-serviços e não das interfaces. Um micro-serviço de interface A simplesmente envia ao micro-serviço de interface B um qualquer valor e o adaptador tem de ser capaz de interceptar este valor e alterá-lo. Em segundo lugar, os adaptadores têm de garantir que preservam toda a informação que recebem, mesmo que não a usem — para ver um exemplo em que isto é necessário, basta pensar nas interfaces A, B, e C, em que A e C dependem de um valor com um campo `age`, mas do qual B não depende. Se A enviar a B este valor, e B o propagar para C, então é expectável que C possa aceder ao campo `age` criado em A.

Por fim, focamo-nos nas questões referentes à extracção de interfaces. Essencialmente, a relação de compatibilidade informa estas questões, por ser ela a utilizadora principal da informação que estamos a extrair. No nosso caso, as interfaces apenas necessitam de conter as assinaturas das funções que o micro-serviço correspondente expõe para o mundo, e informação sobre quais as funções de outras interfaces esta depende. O processo exacto de extracção varia para cada linguagem de programação que o extractor suporte, mas o funcionamento base deverá ser sempre o mesmo: traduzir o código-fonte para uma AST, extrair anotações sobre os tipos manipulados pelo programa, e representar esses tipos numa linguagem comum definida por nós.

A definição da sintaxe da nossa linguagem de especificação de interfaces está fora do âmbito deste artigo, mas exemplos da mesma podem ser encontrados no nosso protótipo.

No contexto particular do nosso protótipo, é relevante mencionar que trabalhamos com alguns pressupostos clássicos do domínio de linguagens de programação. Nomeadamente, assumimos que os nossos três tipos de serviço e os micro-serviços não falham, que os micro-serviços não estão replicados, que a uma interface corresponde um e somente um micro-serviço, e que as comunicações entre serviços não falham.

No entanto, não consideramos estes pressupostos limitativos, dado que nos encontramos numa fase inicial do projecto. Além disto, estes conceitos são tipicamente difíceis de capturar formalmente, pelo que os relegamos para trabalho futuro.

3 Avaliação Preliminar do Sistema

À data da escrita deste artigo, a implementação do nosso sistema ainda se encontra em curso[1]. Assim, elegemos expor um exemplo nesta secção, discutindo algumas das questões que ele levanta no contexto do nosso trabalho. O exemplo que escolhemos realça principalmente decisões feitas ao nível do serviço `Regent Controller`, dado que é aquele cuja implementação se encontra mais matura.

O nosso exemplo começa como um sistema composto por 2 micro-serviços numa relação típica de servidor – cliente, *deployed* simultaneamente como uma unidade. O micro-serviço servidor disponibiliza uma funcionalidade de geração aleatória de inteiros num intervalo, e o micro-serviço cliente usa esta funcionalidade. Nas listagens 1.1 e 1.2 apresentamos as especificações destas interfaces, como actualmente aceites pelo nosso sistema — note-se que esta especificação é tão agnóstica quanto possível aos detalhes de implementação.

Listing 1.1: Servidor (1º *deployment*)

```
Name: NumberServiceServer

Depends:
        -

Defines:
        @-protocol: http
        @-request-body: -
        @-reply-body: json

        at: /random?l={l}&u={u}
        type: (l: int, u: int) -> int
        $-errors: 400, 500
        $-method: GET
```

Listing 1.2: Cliente (2º *deployment*)

```
Name: NumberServiceClient

Depends:
        from: NumberServiceServer
                @-protocol: http
                @-request-body: -
                @-reply-body: json

                at: /random?l={l}&u={u}
                type: (l: int, u: int) -> int
                $-errors: 400, 500
```

[1] As versões públicas do nosso protótipo pode ser consultadas na página GitHub do projecto. [11]

```
                $-method: GET

Defines:
        -
```

As nossas especificações utilizam "@" para denotar uma propriedade que se aplica a todas as funções disponibilizadas num micro-serviço, e "$" para indicar que uma propriedade depende dos valores de alguma outra (e.g., a propriedade "$-method" só faz sentido no contexto do protocolo `HTTP`). Além deste detalhe notacional, realçamos que a propriedade `at` é um simples *path* URL (correspondente ao *endpoint* em que a função está disponibilizada), com a particularidade de definir identificadores (entre chavetas) para que seja possível escrever anotações de tipos. À excepção destes detalhes, cremos que o formato das nossas especificações é de compreensão imediata.

Num 2º *deployment*, o micro-serviço do servidor é estendido com uma função que retorna uma *string* alfanumérica aleatória de um dado comprimento, e o cliente utiliza também esta função. Por uma questão de brevidade, a partir deste *deployment* mostramos apenas as partes mais relevantes das especificações. Ao não alterar as funções já existentes nas interfaces, este *deployment* é trivialmente aceite pelo `Regent Controller`.

Listing 1.3: Servidor (2º *deployment*)

```
at: /mumbo/jumbo?qty={quantity}
type: (quantity: int) -> str
```

Listing 1.4: Cliente (2º *deployment*)

```
at: /mumbo/jumbo?qty={quantity}
type: (quantity: int) -> str
```

No 3º *deployment*, há uma tentativa de alterar no servidor a função introduzida no 2º *deployment* para que passe a aceitar uma palavra da qual são escolhidos os caracteres (conforme a listagem 1.5). Por defeito, o `Regent Controller` aceita esta alteração, dado que é possível o adaptador do cliente transformar os valores enviados naqueles que o servidor espera receber com uma estratégia simples: passar valores por defeito (i.e., 0, "", nulo) aos parâmetros que o cliente desconhece.

Listing 1.5: Servidor (3º *deployment*)

```
at: /mumbo/jumbo/{word}?qty={quantity}
type: (quantity: int, word: str) -> str
```

No entanto, o nosso sistema admite que esta estratégia pode não ser ideal para todos os casos, e permite que o cliente se sobreponha, indicando que não se considera compatível com o servidor se os parâmetros da função *XYZ* se

alterarem. Esta estratégia poderá fazer sentido quando a função de que o cliente depende é crítica para o seu funcionamento correcto.

De forma análoga, o cliente pode indicar que se considera compatível com o servidor mesmo que este deixe de aceitar parâmetros que aceitava anteriormente (caso em que o cliente não envia os parâmetros inutilizados). Esta estratégia faz sentido, por exemplo, em casos de *logging* e *analytics*, em que o cliente trata o servidor como um *data sink* (i.e., envia-lhe todos os dados que consegue obter e assume que o servidor consegue lidar com falhas ou incorrecções nos dados).

Independentemente do 3º *deployment* ser aceite ou não, consideramos um 4º *deployment*, em que o servidor tenta alterar a definição do seu serviço de geração de números aleatórios por forma a que os limites sejam *strings* em vez de inteiros (listagem 1.6).

Listing 1.6: Servidor (4º *deployment*)

```
at: /random?l={l}&u={u}
type: (l: str, u: str) -> int
```

À semelhança do que se passa no *deployment* anterior, o `Regent Controller` apresenta um comportamento por defeito, mas permite que o programador se sobreponha. Por defeito, o controlador rejeita esta alteração, devido à incompatibilidade entre os tipos que o cliente espera poder enviar (inteiros) e os que o servidor espera receber (string).

No entanto, este comportamento pode ser demasiado restritivo — essencialmente, é possível que o servidor funcione sem problema desde que os valores de entrada possam ser *coagidos* para outros, à semelhança da noção de *type casting* presente em derivados da linguagem `C`. Para este efeito, o nosso sistema permite que o programador defina (na especificação do cliente, do servidor, ou na de ambos) uma função de conversão que pode ser aplicada, se necessário, aos parâmetros de entrada ou de retorno das funções.

No contexto do nosso exemplo, bastaria o programador definir uma função em `Python` como a da listagem 1.7 na especificação do servidor, indicando que esta se aplica a todos os parâmetros da função disponibilizada em `/random`.

Listing 1.7: Função de coerção

```
def int_to_str(param: int) -> str:
        return str(param)
```

O funcionamento do mecanismo de coersão funciona da seguinte forma: no caso normal (sem coersões), os adaptadores gerados em cada micro-serviço "consumidor" da interface de outro enviam os parâmetros necessários ao micro-serviço "produtor". No caso de haver uma incompatibilidade de tipos, o adaptador procura na lista de coersões uma função que seja capaz de resolver essa incompatibilidade, e aplica-a transparentemente.

Realçamos que esta funcionalidade ainda não se encontra implementada no nosso protótipo, embora esteja em curso.

4 Trabalho Relacionado

Não existindo, tanto quanto sabemos, outros sistemas/arquitecturas como a nossa, os trabalhos relacionados com o nosso incidem sobre as técnicas que utilizamos para algumas das componentes do nosso sistema, ou sobre alternativas para atacar os mesmos problemas.

Como já mencionado na secção 1, identificámos duas alternativas para os problemas que descrevemos. Primeiro, as baseadas em testes unitários e de integração típicas das abordagens de integração contínua. Um exemplo desta filosofia é a ferramenta Jenkins [10], que submete automaticamente qualquer *deployment* a uma bateria de testes definidos pelos utilizadores. Depois, as baseadas na manutenção de múltiplas versões dos mesmos serviços — que podem ir de esforços como simplesmente disponibilizar duas *quasi*cópias de uma função, à utilização de ferramentas como a ferramenta Mx [8] para escolher dinamicamente, de entre um conjunto de funções, qual a que deve ser executada.

No entanto, nenhuma destas abordagens é perfeita — requerendo, no caso da integração contínua, a escrita e actualização de testes cuja completude dificilmente pode ser atestada formalmente e, no caso das abordagens multi-versão a manutenção das várias funções que são disponibilizadas e um maior número de recursos para suportar a maior carga computacional.

A nível das componentes do nosso sistema (ver a Figura 1), há que distinguir duas categorias: as componentes *off-the-shelf* e as componentes desenvolvidas por nós. As componentes *off-the-shelf* não são o nosso foco em termos de implementação, dado que que já existem imensas versões testadas e comprovadas destas componentes — as componentes nesta categoria são as da base de dados das interfaces, as que usam o protocolo HTTP para enviar e receber (mas não processar) pedidos ao serviço `Regent Controller`, as componentes que implementam um algoritmo para resolver o problema de *Atomic Commit* (já conhecido da comunidade de sistemas [3]), e as componentes de *containers* e de orquestração (que correspondem a ferramentas como Docker [6] e Kubernetes [14]).

É nas restantes componentes — da extracção de interfaces, da verificação de compatibilidade e da geração/utilização de adaptadores ("proxies" na Figura 1) — que focamos os nossos maiores esforços de implementação. No caso da extracção de interfaces, as técnicas já existentes estão relacionadas com ferramentas de compilação e interpretação de código [1], já que extrair e transformar informação de uma linguagem para outra é o cerne da sua função. No caso da verificação de compatibilidade, o trabalho relacionado depende do que compreendemos por compatibilidade — no nosso caso, como descrito na secção 2, a relação de compatibilidade é essencialmente uma relação de subtipagem clássica como a descrita no livro seminal de Benjamin Pierce sobre sistemas de tipos [13]. A nossa utilização de adaptadores consistente na implementação da relação de conversão, que é, à semelhança da relação de compatibilidade, largamente informada pela noção clássica de subtipagem.

Referimos agora alguns conceitos ortogonais à nossa arquitectura, nomeadamente:

- As noções de contractos [4] e de sistemas de tipos complexos como os tipos comportamentais [9], que são ferramentas utilizadas para descrever as interfaces/formas de comunicação entre programas.
- A lógica de Hoare [7], que permite descrever através de pré e pós-condições o funcionamento de uma função, que tem uma aplicação óbvia na definição de relações de compatibilidade alternativas.

É também interessante mencionar que já conhecemos uma utilização de métodos formais e automáticos para tornar a utilização de micro-serviços mais robusta [2], embora essa utilização se foque em garantir a correcção da construção dos artefactos que descrevem os micro-serviços a ser *deployed* (i.e., a correcção dos "Dockerfile" utilizados para construir *containers* com a ferramenta Docker), ao invés da nossa utilização para garantir a correcção da própria acção de *deployment*.

5 Conclusão e Trabalho Futuro

O nosso trabalho, na sua concepção actual, permite-nos racicionar sobre os *deployments* de micro-serviços com maior grau de robustez: a nível concepcional, nada parece ser impeditivo à construção do serviço que propomos. A nível prático, os resultados de que dispomos ainda são preliminares, mas apontam no sentido de que o custo operacional acrescido pela nossa solução será diminuto face aos custos endógenos à gestão de micro-serviços (concretamente, envio de binários pela rede e a sua instalação).

O nosso trabalho pode ser visto como uma base mínima para a discussão e tratamento da evolução de sistemas. Como trabalho futuro vemos a extensão da arquitectura e ideias aqui apresentadas para lidar com relações de compatibilidade mais ricas — incorporando, por exemplo, noções de pré/pós-condição — e com noções de replicação de micro-serviços — relaxando assim o pressuposto de que os micro-serviços não falham, que cremos ser efectivamente desnecessária no nosso modelo actual. Vemos ainda como trabalho futuro a refactorização do serviço `Regent Controller` para lidar com pedidos de *deploy* de forma concorrente ao invés de serializar a sua análise e para usar estratégias de difusão dos novos micro-serviços pelas máquinas que os correm (ou seja: usar protocolos mais leves que os de Atomic Commit).

Acknowledgements Trabalho financiado por NOVA LINCS UID/CEC/04516/2013, COST CA15123 - FC&T Project CLAY - PTDC/EEI-CTP/4293/2014

Referências

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2Nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)

2. Benni, B., Mosser, S., Collet, P., Riveill, M.: Supporting micro-services deployment in a safer way: A static analysis and automated rewriting approach. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 1706–1715. SAC '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3167132.3167314, http://doi.acm.org/10.1145/3167132.3167314
3. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming. Springer Publishing Company, Incorporated, 2nd edn. (2011)
4. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Trans. Program. Lang. Syst. **31**(5), 19:1–19:61 (Jul 2009). https://doi.org/10.1145/1538917.1538920, http://doi.acm.org/10.1145/1538917.1538920
5. Cloudfare, Inc.: What Is a Serverless Microservice? | Serverless Microservices Explained, https://www.cloudflare.com/learning/serverless/glossary/serverless-microservice/, accessed 2019-06-12
6. Docker Inc.: Enterprise Container Platform | Docker, https://www.docker.com/, accessed 2019-06-12
7. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259, http://doi.acm.org/10.1145/363235.363259
8. Hosek, P., Cadar, C.: Safe software updates via multi-version execution. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 612–621 (May 2013). https://doi.org/10.1109/ICSE.2013.6606607
9. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P.M., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (Apr 2016). https://doi.org/10.1145/2873052, http://doi.acm.org/10.1145/2873052
10. Jenkins: Testing, https://jenkins.io/doc/developer/testing/, accessed 2019-06-12
11. Álvaro António Santos: Regent, https://github.com/Alvaro-Santos/Regent, accessed 2019-06-12
12. Newman, S.: Building Microservices. O'Reilly Media, Inc., 1st edn. (2015)
13. Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
14. The Kubernetes Authors: Production-Grade Container Orchestration - Kubernetes, https://kubernetes.io, accessed 2019-06-12
15. Thones, J.: Microservices. IEEE Software **32**(01), 116–116 (jan 2015). https://doi.org/10.1109/MS.2015.11
16. Tony Mauro: Adopting Microservices at Neflix: Lessons for Architectural Design, https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, accessed 2019-06-12