Geração Automática de Modelos de Desempenho para Arquiteturas de Microsserviços

Lucas Machado Gutierrez[1], **Rafael R. Obelheiro**[1]

[1]Universidade do Estado de Santa Catarina - UDESC
Departamento de Ciência da Computação

lucas.machado.gutierrez@gmail.com, rafael.obelheiro@udesc.br

Resumo. *O presente trabalho propõe a implementação de uma ferramenta capaz de gerar automaticamente modelos analíticos de desempenho para aplicações baseadas na arquitetura de microsserviços (MS) através de monitoramento do sistema. É apresentada, inicialmente, a solução conceitual proposta e a implementação da ferramenta, mostrando como foi feita a extração dos parâmetros de desempenho. Por fim, é mostrado um exemplo de sistema e o modelo gerado.*

1. Introdução

A arquitetura de MS vem se mostrando uma tendência no desenvolvimento de aplicações baseadas em nuvens computacionais. Esta arquitetura estrutura uma aplicação como um conjunto de pequenos serviços, cada um sendo executado como um processo independente e se comunicando através de mecanismos leves, geralmente através de requisições HTTP. A arquitetura tem como princípio dividir o sistema em serviços até que cada unidade tenha apenas uma responsabilidade, atuando de maneira independente e autônoma, buscando pelo baixo acoplamento e alta coesão [Newman 2015].

A arquitetura, entretanto, acrescenta uma complexa rede de comunicações entre os MSs, diminuindo consideravelmente a visibilidade do sistema. Uma forma de lidar com a complexidade e agilidade de arquiteturas de MS é monitorar extensivamente as aplicações diretamente em produção, permitindo assim identificar e corrigir rapidamente eventuais problemas. *Tracing* distribuído permite entender o fluxo de requisições em um sistema baseado em MSs e é usado para propiciar observabilidade de aplicações em MSs [Sridharan 2018].

Embora informações geradas por uma ferramenta de *tracing* distribuído forneçam um instantâneo que descreve onde uma aplicação está gastando seu tempo, elas não são suficientes para raciocinar sobre o desempenho dessa aplicação. Por exemplo, um *trace* (representação de uma propagação de contexto de uma determinada requisição ao sistema) pode indicar os MSs que mais contribuem para a latência fim a fim de requisições, mas é incapaz de apontar se a ativação de novas instâncias desses serviços propiciaria uma redução dessa latência. Modelos analíticos de desempenho, neste contexto, ajudam a analisar o desempenho e explorar soluções para o sistema a nível de componente, de maneira mais rápida e barata. Um exemplo de modelagem analítica é o modelo de redes de filas [Menascé et al. 2004].

As características das arquiteturas de MS, como quantidades potencialmente grandes de serviços e interações entre eles, agilidade e compartimentalização de equipes de desenvolvimento, fazem com que a construção de modelos analíticos torne-se um desafio.

Para contornar este problema, [Mizan and Franks 2012] e [Israr et al. 2007] propuseram um gerador automático de modelos baseados em redes de filas em camadas usando de monitoramento, porém ambos os trabalhos usaram de instrumentação específica e não trabalharam com aplicações em MSs, mas com sistemas distribuídos de maneira geral.

A proposta deste trabalho é gerar automaticamente modelos de redes de filas para aplicações baseadas em MSs através de monitoração, usando uma ferramenta suportada pelo OpenTracing (2018), uma API aberta para instrumentação de aplicações distribuídas.

2. Solução Conceitual e Implementação

Para construir um modelo de rede de filas, é necessário definir as filas e a demanda de serviço em cada uma. Para o problema, cada MS é representado como uma fila no sistema. A demanda em cada fila depende do número de vezes que um MS é invocado no processamento de uma requisição (o seu número de visitas) e do tempo médio dispendido no MS a cada visita (o seu tempo de serviço). A ordem de processamento das requisições nas várias filas não é relevante, pois o modelo trabalha com valores médios.

No contexto de *tracing* distribuído, um *span* representa toda a atuação de um MS em uma determinada requisição e cada *span* informa o momento de início, a duração total e o MS que o invocou. Um *trace*, por sua vez, é um grafo acíclico de *spans* que representa a propagação de contexto de uma requisição no sistema. Com um *trace* de uma aplicação, é possível constatar quantas vezes um serviço foi invocado, através do número de *spans* de um mesmo serviço no *trace*, e encontrar o tempo de serviço deste, que é a diferença da duração total com a soma da duração de todos os *spans* do qual este MS invoca. O tempo de serviço, em outras palavras, é o tempo no qual o serviço esteve efetivamente processando a requisição, e não só aguardando o fim do processamento dos serviços nos quais este invoca. Com essas duas informações, é possível calcular a demanda de cada MS, que é o produto do número de visitas pelo tempo de serviço.

Para a identificação dos tempos de serviço de maneira apropriada, a coleta de *traces* de um determinado sistema deve ser feita a partir de aplicação de uma requisição por vez no sistema, garantindo que nunca terá mais de uma requisição dentro do sistema. Com isso, é garantido a não geração de filas em nenhum MS. Com um número determinado de *traces* coletados, é calculada a média dos tempos de serviço constatados dentre todos os *spans* em todos os *traces* de cada MS.

Uma premissa do trabalho é que a aplicação está instrumentada com o OpenTracing (2018), uma API aberta de *tracing* distribuído, o que permite o uso de qualquer ferramenta suportada por esta API. A ferramenta escolhida para este trabalho foi o Jaeger[1]. O fluxo da solução está representado na Figura 1, onde uma aplicação em MSs, instrumentada pelo OpenTracing, é monitorada pelo Jaeger e este armazena os *traces* desta aplicação no banco de dados Cassandra. Trace2PDQ é a implementação proposta neste trabalho, um *script* Python que busca os *traces* no banco de dados, identifica os microsserviços associados a esta aplicação e gera um *script* PDQ (ferramenta capaz de resolver modelos baseados em redes de filas) [PDQ 2017]. O PDQ, então, resolve o modelo e gera métricas de desempenho para uma determinada taxa de chegada de requisições ao sistema. Foi escolhido a implementação do PDQ para a linguagem R (PDQ-R).

[1]Ferramenta desenvolvida pela Uber para monitoramento de suas aplicações [OpenTracing 2018]

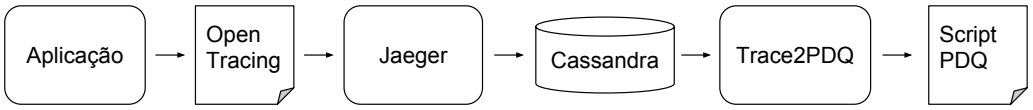

Figura 1. Fluxo da solução.

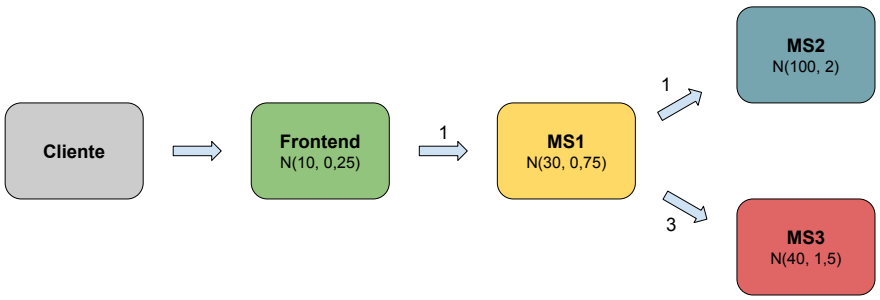

Figura 2. Grafo de requisições do AMS.

A solução gera modelos de rede aberta, com apenas uma classe de requisições, ou seja, cada comunicação entre serviços envolve o mesmo tipo de requisição e associa o mesmo tempo de serviço. A solução resolve apenas modelos com chamadas síncronas e considera que os MSs têm apenas um servidor processando as requisições.

3. Resultados

Foi aplicada a implementação desenvolvida em um caso de teste, uma aplicação em MSs chamada AMS, desenvolvida pelo autor. Foram implementados quatro MSs e um cliente, todos na linguagem Go, se comunicando por chamadas RPC. A Figura 2 ilustra o grafo de requisições do MS para uma requisição do cliente, onde as arestas representam as invocações e o peso é o número de invocações em uma requisição. Cada MS foi implementado com um atraso aleatório normal para simular o tempo de serviço, onde a média e o desvio padrão definido para cada um está representado na mesma figura, respectivamente, abaixo do nome do MS. A comunicação é síncrona, ou seja, o MS1 só invoca o MS3 após a comunicação com o MS2 terminar.

A Figura 3 ilustra o *script* gerado automaticamente para a aplicação MS. O cálculo da demanda foi colocado explicitamente como o produto do número de visitas pelo tempo de serviço para facilitar a visualização do resultado. Como o cliente também foi instrumentado, ele foi encontrado e definido como uma fila do sistema. É possível observar uma coerência com os valores de atraso e número de visitas definidos na Figura 2.

Para a validação de modelos analíticos gerados automaticamente para o AMS, foi construído um modelo de simulação da aplicação AMS e aplicada uma série de simulações com diferentes taxas de chegada. Foram, então, comparados os tempos de resposta do sistema e a utilização de cada MS do modelo gerado e das simulações para cada taxa de chegada. Para a comparação, foi feito o cálculo do erro relativo para cada métrica em cada taxa de chegada. Para os tempos de resposta, o maior erro foi de 5%, enquanto para a utilização foi de 7,63%. Os erros inferiores a 10%, tanto a nível de sistema quanto a nível de MS, mostram que os modelos gerados são consistentes.

```r
# Modelo de Redes de Filas para o AMS

library(pdq)

nReqs <- 1
tempo <- 530.248
lambda <- nReqs / tempo

# Inicializacao

Init("OpenCircuit")
SetComment("Modelo de Redes de Filas para o AMS")

# Definicao das Filas

CreateNode("Frontend", CEN, FCFS)
CreateNode("Ms3", CEN, FCFS)
CreateNode("Hello-World", CEN, FCFS)
CreateNode("Ms2", CEN, FCFS)
CreateNode("Ms1", CEN, FCFS)

# Definicao da Carga de Trabalho

CreateOpen("req", lambda)
SetWUnit("req")
SetTUnit("ms")

# Definicao das Demandas

SetDemand("Frontend", "req", 1 * 10.60065)
SetDemand("Ms3", "req", 3 * 40.1524666667)
SetDemand("Hello-World", "req", 1 * 1.02825)
SetDemand("Ms2", "req", 1 * 100.9822)
SetDemand("Ms1", "req", 1 * 32.0555)

# Soluciona o Modelo

Solve(CANON)
Report()
```

Figura 3. Código PDQ-R gerado para o AMS.

4. Conclusão e Trabalhos Futuros

Este trabalho apresentou uma solução para geração automática de modelos de desempenho para aplicações de MS, usando de ferramentas e instrumentação de monitoramento já existentes. Para os requisitos apresentados, a geração foi possível e gerou modelos consistentes.

Como trabalhos futuros, existe a possibilidade de tratar chamadas assíncronas, considerar mais de uma classe de requisição, identificar a quantidade de servidores que processam requisições que um MS possui e comparar os resultados dos modelos com resultados experimentais de aplicações testadas.

Referências

Israr, T., Woodside, M., and Franks, G. (2007). Interaction tree algorithms to extract effective architecture and layered performance models from traces. *Journal of Systems and Software*, 80(4):474–492.

Menascé, D. A., Almeida, V. A. F., and Dowdy, L. W. (2004). *Performance by Design: Computer Capacity Planning by Example*. Prentice-Hall PTR.

Mizan, A. and Franks, G. (2012). Automated performance model construction through event log analysis. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 636–641.

Newman, S. (2015). *Building Microservices: Designing Fine-grained Systems*. O'Reilly Media.

OpenTracing (2018). Introduction - OpenTracing. `http://opentracing.io/documentation/`.

PDQ (2017). PDQ: *Pretty Damn Quick* performance analyzer. `http://www.perfdynamics.com/Tools/PDQ.html`.

Sridharan, C. (2018). *Distributed Systems Observability*. O'Reilly Media.