Universidade Federal de Pernambuco
Centro de Informática

Graduação em Engenharia da Computação

Implementação de Middleware baseada em Microserviços

Rubem Fonseca Moura Junior

Trabalho de Graduação

Recife
15 de maio de 2018

Universidade Federal de Pernambuco
Centro de Informática

Rubem Fonseca Moura Junior

Implementação de Middleware baseada em Microserviços

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Nelson Souto Rosa*

Recife
15 de maio de 2018

Dedico este Trabalho de graduação primeiramente a Deus, por ter me guiado e me concedido capacidade para trilhar todo caminho na universidade até chegar aqui. Dedico também aos meu honrosos pais Rubem Fonseca Moura e Elizabeth Ferreira da Rocha Moura que me educaram da melhor maneira que alguém poderia educar, e se sacrificaram em diversas situações para me manter firme na universidade.

Agradecimentos

Agradeço primeiramente a Deus, por toda sabedoria concedida a mim por Ele, aos meu pais e irmãs pelas ajudas e sacrifícios para me manter na universidade. Agradeço a minha noiva pelo apoio e compreensão nesta longa caminhada. Aos meus professores que compartilharam seus conhecimentos comigo para que eu pudesse chegar até aqui, em especial ao professor Nelson Souto Rosa por ter dedicado parte de seu tempo me orientando no desenvolvimento deste trabalho. Agradeço aos meus amigos Rita de Cássia, Tasso Luiz, Paulo Renato, Daniel Alexandro, Rodrigo Bruno e José Renê pelos momentos difíceis compartilhados em companheirismo, fazendo dessa jornada um ambiente mais agradável. Por fim, quero agradecer ao Centro de Informática por me proporcionar uma educação de altíssima qualidade com toda sua infraestrutura e funcionários altamente qualificados. À todos, meu muito obrigado.

Tão-somente esforça-te e tem mui bom ânimo, para teres o cuidado de fazer conforme a toda a lei que meu servo Moisés te ordenou; dela não te desvies, nem para a direita nem para a esquerda, para que prudentemente te conduzas por onde quer que andares.

— JOSUÉ 1:7 (Bíblia)

Resumo

Microserviços têm sido amplamente utilizados para o desenvolvimento de sistemas distribuídos. A arquitetura de microserviço estrutura uma aplicação como uma coleção de serviços fracamente acoplados. Os benefícios de se desenvolver aplicações com a arquitetura de microserviços são variados. Um deles é que através da arquitetura de microserviço um software pode ser desenvolvido como pequenos serviços, cada um executando seu próprio processo e se comunicando através de mecanismos leves. Isso torna o processo de entendimento e desenvolvimento da aplicação mais fácil. Cada serviço pode ser desenvolvido independentemente dos outros serviços tornando mais fácil a implementação de novas versões. O desenvolvimento em escala se torna mais fácil, possibilitando que cada serviço seja desenvolvido independentemente. Isso nos dá a possibilidade de organizar o projeto em torno de várias equipes, onde cada uma será responsável por um ou mais serviços individuais. Cada equipe poderá implementar e escalar seus serviços independentemente de todas as outras equipes. Outra vantagem é a melhoria do isolamento de falhas. Caso ocorra um problema de *memory leak* em um microserviço por exemplo, apenas ele será afetado, evitando que o problema seja propagado para todo o sistema. Os avanços na área de microserviços sugerem o uso desta tecnologia na implementação de sistemas de middleware, uma vez que ambas estão relacionadas à implementação de sistemas distribuídos. Um middleware é um software de distribuição localizado entre a aplicação e o sistema operacional, que abstrai a complexidade e a heterogeneidade de tecnologias envolvidas no desenvolvimento de aplicações distribuídas e facilita enormemente a tarefa dos desenvolvedores de aplicações distribuídas. Para implementar um middleware utilizando os conceitos de microserviços, será necessário criar seus componentes separadamente como microserviços, onde eles irão se comunicar entre si. Neste trabalho de graduação foi desenvolvido um middleware onde os componentes de sua arquitetura são microserviços. Com isso, será apresentada uma avaliação de desempenho onde iremos observar as vantagens e desvantagens do uso de microserviços como base para o desenvolvimento de middleware.

Palavras-chave: Middleware, Microserviços, Docker, *Remoting Patterns*

Abstract

Microservices have been widely used for the development of distributed systems. The microservice architecture structures an application as a collection of loosely coupled services. The benefits of application development with the microservice architecture are varied. One is that through the microservice architecture, software can be developed as small services, each one executing its own process and communicating through light mechanisms. This facilitates the process of understanding and developing the application. Each service can be developed independently of other services, facilitating the deployment of new versions. The development of scales becomes easier, allowing each service to be developed independently. This gives us the possibility to organize the project around several teams, where each one will be responsible for one or more individual services. Each team can deploy and scale their services independently of all other teams. Another advantage is improved fault isolation. If a memory leak problem occurs in a microservice, for example, only it will be affected, preventing the problem from being propagated to the whole system. Advances in the micro-services area suggest the use of this technology in the implementation of middleware systems, since both are related to the implementation of distributed systems. A middleware is distributed distribution software between the application and the operating system, which abstracts the complexity and heterogeneity of the technologies involved in the development of distributed applications and greatly facilitates the task of distributed application developers. In this work of graduation a middleware was developed where the components of its architecture are microservices. Thus, a performance evaluation will be presented where we will observe the advantages and disadvantages of the use of microservices as the basis for the development of middleware.

Keywords: Middleware, Microservices, Docker, *Remoting Patterns*

Sumário

1 Introdução **1**
- 1.1 Contexto e motivação 1
- 1.2 Problema 2
- 1.3 Objetivo 2
- 1.4 Estrutura do Documento 2

2 Conceitos Básicos **3**
- 2.1 Middleware 3
- 2.2 Remoting Patterns 4
- 2.3 Microserviços 5
- 2.4 Considerações finais 6

3 Middleware Baseado em Microserviços **7**
- 3.1 Fundamentação 7
- 3.2 Requisitos e Arquitetura 7
- 3.3 Projeto e Implementação 9
- 3.4 Considerações Finais 18

4 Avaliação de Desempenho **19**
- 4.1 Objetivos 19
- 4.2 Experimentos 19
  - 4.2.1 Métricas 19
  - 4.2.2 Parâmetros 19
  - 4.2.3 Fatores 20
  - 4.2.4 Invocação do método remoto 21
  - 4.2.5 Invocação do método remoto limitando a quantidade de memória 21
  - 4.2.6 Invocação do método remoto limitando a quantidade de CPUs 23
  - 4.2.7 Teste de escalabilidade 25
- 4.3 Considerações finais 32

5 Conclusão e Trabalhos Futuros **33**
- 5.1 Conclusão 33
- 5.2 Trabalhos Futuros 34

Lista de Figuras

2.1 Camadas de um middleware 3
2.2 Remoting Patterns 4

3.1 Diagrama de arquitetura do middleware 8
3.2 Diagrama de sequência do registro do Servidor da Aplicação no Servidor de Nomes 10
3.3 Diagrama de sequência da solicitação do cliente da aplicação 11
3.4 Arquivo *dockerfile* utilizado 12

4.1 Tempo gasto em função da quantidade de invocações realizadas pelos middlewares com microserviço e sem. 22
4.2 Tempo gasto em função da quantidade de invocações de acordo com middleware com limite de 50mb de memória disponível e sem limites. 23
4.3 Tempo gasto em função da quantidade de invocações de acordo com middleware sem limites de memória e com limite de 45mb de memória. 24
4.4 Tempo gasto em função da quantidade de invocações de acordo com middleware sem limites de memória e com limite de 40mb de memória. 25
4.5 Resultado da execução do comando *sudo docker stats*. 25
4.6 Tempo gasto em função da quantidade de invocações de acordo com middleware sem limites de memória e com todos limites anteriores. 26
4.7 Tempo gasto em função da quantidade de invocações de acordo com middleware utilizando 4 CPUs e 3 CPUs. 27
4.8 Tempo gasto em função da quantidade de invocações de acordo com middleware utilizando 4 CPUs e 2 CPUs. 28
4.9 Tempo gasto em função da quantidade de invocações de acordo com middleware utilizando 4 CPUs e 1 CPU. 29
4.10 Tempo gasto em função da quantidade de invocações de acordo com middleware utilizando 4 CPUs e 50% de uma CPU. 30
4.11 Tempo gasto em função da quantidade de invocações de acordo com middleware utilizando 4 CPUs e 25% de uma CPU. 31
4.12 Tempo gasto em função da quantidade de invocações realizadas pelos middlewares com um servidor de *bind* e dois servidores de *bind* 32

CAPÍTULO 1

Introdução

1.1 Contexto e motivação

Middleware é uma infraestrutura de software localizado entre a aplicação e o sistema operacional que tem como finalidade facilitar o desenvolvimento e a execução de sistemas distribuídos [9]. Uma das principais dificuldades ao se construir uma aplicação distribuída é tratar com a heterogeneidade de sistemas operacionais, protocolos de redes, hardware e linguagem de programação. O middleware esconde dos desenvolvedores de sistemas distribuídos a heterogeneidade e a complexidade inerentes à distribuição.

Uma das grandes razões para se usar um middleware é que ele simplifica o processo de comunicação. Com o middleware, sistemas distintos não precisam de integração direta entre eles, toda a comunicação se dá através do middleware. Através de um middleware dois sistemas distintos podem operar independentemente. Se um sistema estiver inativo para manutenção, o outro sistema poderá continuar a operar. Isso significa que quedas de comunicação entre os sistemas não atrapalham seus funcionamentos.

A utilização de novas arquiteturas e tecnologias pode reduzir o tempo de desenvolvimento dos sistemas de middleware. Como por exemplo, utilizando-se a arquitetura de microserviços o desenvolvimento do middleware poderá ser distribuído entre equipes separadas, que irão implementar microserviços de cada componente do middleware. Através do conceito de microserviços uma aplicação pode ser estruturada como uma coleção de pequenos serviços fracamente acoplados. A arquitetura de microserviços permite a entrega/implantação contínua de aplicações grandes e complexas.

Escalabilidade é um dos aspectos chave dos microserviços. Como cada serviço é um componente separado, é possível expandir uma única função ou serviço sem ter que dimensionar toda a aplicação. Os serviços críticos para o negócio podem ser implantados em vários servidores para aumentar a disponibilidade e o desempenho de um serviço sem impactar o desempenho dos outros serviços.

A implementação de um middleware utilizando a arquitetura de microserviços traz a possibilidade do middleware ser facilmente escalável. A disposição de cada componente do middleware como um serviço separado, desacopla seu código evitando assim que possíveis falhas afetem todo o sistema.

1.2 Problema

O problema abordado neste trabalho de graduação é como implementar um middleware baseado em Chamada de Procedimento Remoto (RPC) utilizando microserviços.

Os desafios para implementar um middleware baseado em microserviços são os seguintes: definir quais os componentes do middleware, definir quais destes componentes serão implementados como microserviços, e definir uma estratégia de distribuição e composição dos componentes implementados como microserviços.

1.3 Objetivo

Este trabalho de graduação tem como principal objetivo desenvolver um middleware utilizando conceitos de microserviços. Para alcançar este objetivo principal, alguns objetivos secundários foram definidos: definir o conjunto de requisitos do middleware, propor uma arquitetura que satisfaça os requisitos definidos e implementar os microserviços. Por fim, será feita uma análise com vantagens e desvantagens do uso de microserviços e uma avaliação de desempenho com variações de implementações da arquitetura proposta.

1.4 Estrutura do Documento

O trabalho aqui apresentado está dividido em 5 capítulos, incluindo este capítulo introdutório. No Capítulo 2, serão apresentados os conceitos básicos necessários para o entendimento do trabalho, tais como os conceitos de middleware e microserviços. Em seguida, o Capítulo 3 discorrerá sobre a fundamentação, os requisitos e arquitetura, projeto e implementação do middleware baseado em microserviço. No Capítulo 4 será realizada uma avaliação de desempenho do middleware. Por fim, o Capítulo 5 trará um resumo concluindo o estudo realizado, assim como os desafios encontrados e as propostas para possíveis trabalhos futuros.

CAPÍTULO 2

Conceitos Básicos

2.1 Middleware

O termo middleware se aplica a uma camada de software que fornece uma abstração de programação e mascara a heterogeneidade das redes subjacentes, hardware, sistemas operacionais e linguagens de programação [6].

Além de resolver os problemas de heterogeneidade, o middleware fornece um modelo computacional uniforme usado pelos desenvolvedores de aplicações distribuídas. Os modelos possíveis incluem a chamada remota de objetos, a notificação remota de eventos, o acesso remoto ao SQL e o processamento de transações distribuídas [6]. Por exemplo, um middleware que fornece invocação remota de objetos, permite que um objeto de um programa executado em um computador invoque um método de um objeto de um programa em execução em outro computador. Sua implementação oculta o fato de que as mensagens são passadas pela rede para enviar a solicitação de chamada e sua resposta.

Sistemas de middleware são organizados em camadas [10]. Elas são definidas como Infraestrutura, Distribuição, Serviços Comuns e Serviços Específicos como mostrado na Figura 2.1.

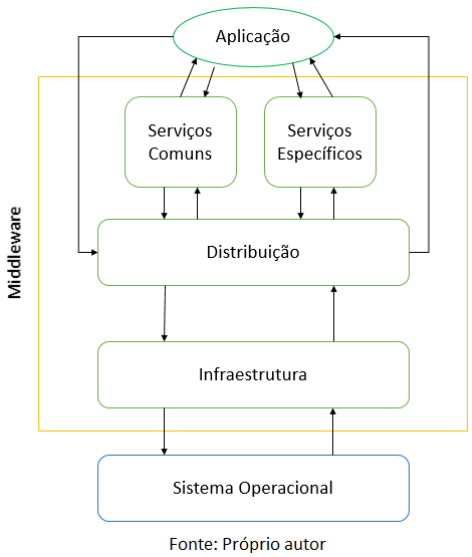

Figura 2.1 Camadas de um middleware

A camada de infraestrutura é responsável pela interação com o sistema operacional, além

de esconder a heterogeneidade de hardware, software e rede. Já a camada de distribuição implementa o modelo de distribuição do middleware (por exemplo o modelo cliente-servidor), define as abstrações usadas para construir os sistemas distribuídos, por exemplo os objetos [10].

Os serviços comuns são serviços providos pelo middleware e usados por aplicações de todos os domínios, como por exemplo os serviços de segurança, nomes, transação e controle de concorrência. Por outro lado, os serviços de middleware específicos são os serviços adaptados aos requisitos de domínios específicos, como telecomunicações, comércio eletrônico, assistência médica, automação de processos ou aeroespacial. [10].

2.2 Remoting Patterns

Para desenvolver as camadas de um middleware podemos utilizar os *Remoting Patterns* [12] que são específicos para projetos de middleware orientado a objetos. Um middleware básico é composto por sete componentes: *Proxy, Requestor, Client Request Handler, Invoker, Server Request Handler, Marshaller* e o *Lookup*. Ao estruturar um middleware utilizando os *Remoting Patterns* teremos sua estrutura definida como na Figura 2.2.

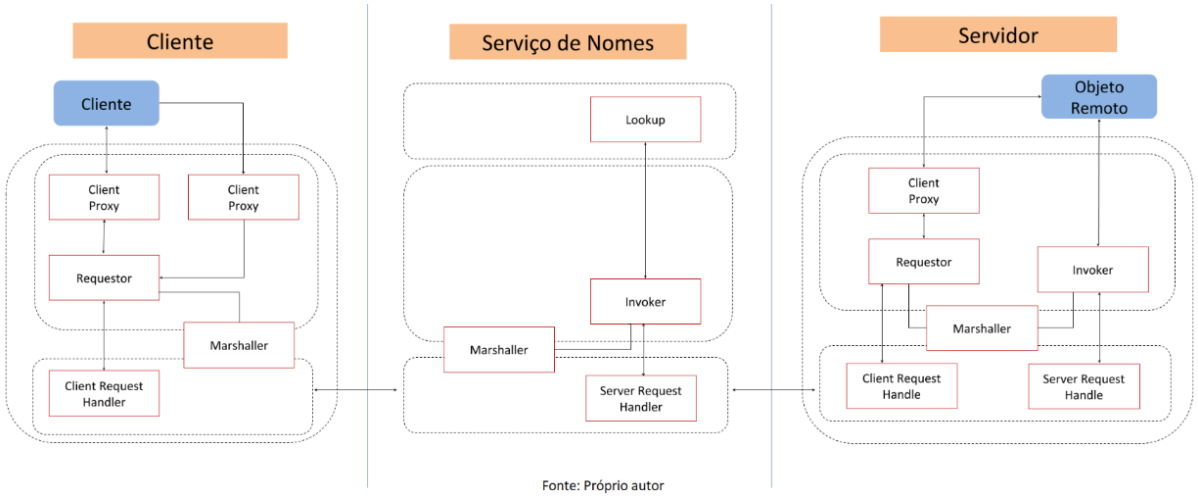

Figura 2.2 Remoting Patterns

Em um middleware, o *client proxy* permite que objetos remotos sejam acessados da mesma forma que objetos locais. Para isso, o *client proxy* deve possuir a mesma interface do objeto remoto [12].

O *requestor* é responsável por invocar possíveis operações do objeto remoto no servidor. Por meio dele a conexão é estabelecida com o servidor, a invocação do objeto é enviada, e o resultado da invocação é recebido. O *client request handler* é responsável por gerenciar conexões de rede, enviar/receber invocações, tratar *timeouts* e erros de invocação [12].

O *invoker* é responsável por fazer a invocação alcançar o objeto remoto depois que ela chega ao servidor. Informações são passadas ao *invoker* para que ele selecione o objeto remoto

apropriado. Já o *server request handler*, é responsável por receber a invocação, combinar fragmentos de mensagens (quando necessário) e encaminhar a invocação para o *invoker* correto [12].

O *marshaller* tem como principal objetivo serializar os dados em uma sequência de bytes que poderá ser transmitida pela rede. Os servidores irão registrar AOR's (*Absolute Object Reference*) por nomes para seu serviço [12]. Uma AOR estende o conceito de identificação de objeto com informações de localização, como o nome do host, a porta e o identificador de um objeto remoto. Geralmente, é importante que a localização dos objetos remotos não esteja conectada ao sistema. Por exemplo, quando objetos remotos precisarem ser movidos para outros hosts, a integridade do sistema distribuído não deve ser comprometida.

O *lookup* é o serviço de nomes do middleware. Ele permite que desenvolvedores de um servidor registrem seus objetos remotos em um serviço central, por nome ou propriedade [12]. Os clientes podem então, descobrir os objetos remotos usando esse serviço. O cliente precisa conhecer apenas a AOR do serviço de pesquisa, em vez de várias AOR's dos objetos remotos com os quais deseja se comunicar. O padrão *lookup* simplifica o gerenciamento e a configuração de sistemas distribuídos, pois os clientes podem encontrar facilmente objetos remotos, evitando o acoplamento entre eles. O *lookup* será responsável por fornecer aos clientes estas AOR's.

2.3 Microserviços

Um microserviço é um processo coeso e independente que interage por meio de mensagens [4]. Por exemplo, considere um serviço destinado a solucionar cálculos. Para chamá-lo de microserviço, ele deve fornecer operações aritméticas solicitáveis por meio de mensagens, mas não deve fornecer outras funcionalidades (que possam ser vagamente relacionadas), como plotagem e exibição de funções [4]. Podemos afirmar também que um microserviço é um pequeno software que pode ser implantado, escalado e testado independentemente, além de ter uma única responsabilidade [11].

A arquitetura de microserviço é um modo utilizado para desenvolver uma aplicação como um conjunto de pequenos serviços independentes. Cada um desses serviços é executado em um processo independente [1].

O Docker [7] é uma plataforma aberta para desenvolvimento, entrega e execução de aplicativos e atualmente tem sido bastante utilizada para desenvolvimento de microserviços. É possível desenvolver e fazer o *deploy* de softwares muito rapidamente, como também é fácil criar os serviços necessários separadamente e gerenciá-los como microserviços sem afetar outros serviços.

Com o Docker a implantação de aplicações é feito em contêineres portáteis que são independentes. Dessa forma o contêiner facilita o processo de implantação e execução de aplicações em ambientes isolados. Os contêineres são geralmente descritos como ambientes leves de execução em tempo real. Muitos de seus principais componentes são equivalentes aos de uma máquina virtual. Um contêiner também possui serviços isolados de um sistema operacional e são projetados para facilitar o empacotamento e execução de softwares. Podemos observar algumas características e requisitos dos microserviços:

1. Cada microserviço possui uma única responsabilidade em um sistema (por exemplo as operações aritméticas fornecidas pelo microserviço de um sistema destinado a solucionar cálculos);

2. Cada microserviço pode ser desenvolvido e implantado de forma independente; e

3. O microserviço melhora o isolamento de falhas.

Os contêineres suportam tais requisitos de uma arquitetura de microserviços. Cada microserviço pode ser implantado sem interromper outros microserviços. Pelo gerenciamento de isolamento, velocidade e gerenciamento de ciclo de vida, os contêineres fornecem um ambiente ideal para a implantação de serviços. Desse modo, torna mais fácil a implantação de novas versões de serviços em contêineres [5].

2.4 Considerações finais

Neste capítulo, podemos entender melhor o que é um middleware e como ele é dividido em camadas, além de compreender a função de cada uma delas. Vimos como implementar um middleware utilizando os *Remoting Patterns*. Também pudemos compreender o que é um microserviço e o que vem a ser uma arquitetura baseada em microserviço. Pudemos entender um pouco sobre a ferramente Docker e vimos que ela é uma ótima ferramenta para desenvolver microserviços em contêineres.

CAPÍTULO 3

Middleware Baseado em Microserviços

3.1 Fundamentação

Como vimos na Seção 2.2, ao estruturar o middleware utilizando o *Remoting Patterns*, podemos notar que seus componentes possuem papéis específicos bem definidos. Desta forma, temos a possibilidade de desenvolvê-los como serviços desacoplados.

Ao desenvolver o middleware utilizando os conceitos de microserviços, poderemos estruturar seus componentes independentes entre si como processos separados, onde cada um irá fornecer um serviço e o sistema completo irá compor o middleware.

Quando pensamos em desenvolver um middleware utilizando os *Remoting Patterns* junto com microserviços, devemos notar que o middleware terá seus componentes desacoplados, possibilitando sua expansão.

3.2 Requisitos e Arquitetura

Podemos definir os requisitos funcionais e não funcionais do middleware desenvolvido como descritos na Tabela 3.1 e Tabela 3.2 .

Dentre os requisitos funcionais, o middleware irá transportar mensagens entre os seus componentes desenvolvidos em microserviços. Observando a transparência de acesso, o middleware irá utilizar *proxies* para intermediar a comunicação entre o cliente e o servidor da aplicação. O middleware irá fornecer o *proxy* correspondente ao servidor que possui o AOR da aplicação solicitada pelo cliente.

O middleware irá fornecer um serviço de nomes para que o cliente possa solicitar o serviço da aplicação pelo nome. Para isso, o middleware irá armazenar todas as AOR's das aplicações

Tabela 3.1 Requisitos Funcionais

#	**Descrição dos Requisitos Funcionais**
RF01	O middleware irá transportar as mensagens entre os seus componentes desenvolvidos em microserviços
RF02	O middleware irá registrar os servidores das aplicações pelo nome
RF03	O middleware irá fornecer as AOR's das aplicações registradas
RF04	O middleware irá utilizar *proxies*
RF05	O middleware irá intermediar a solicitação/resposta entre o cliente e o servidor da aplicação

Tabela 3.2 Requisitos não Funcionais

#	Descrição dos Requisitos Não Funcionais
RNF01	Escalabilidade
RNF02	Testabilidade
RNF03	Modularidade

registradas.

Todas as requisições serão intermediadas pelo middleware, e nunca diretamente entre o cliente e o servidor da aplicação. Por isso o cliente sempre irá solicitar a execução da aplicação através de um microserviço dentro do middleware. Para o cliente este microserviço será o fornecedor da aplicação.

Dentre os requisitos não funcionais listados na Tabela 3.2, temos a escalabilidade. O middleware poderá ser escalável de forma simples, devido a sua arquitetura utilizando microserviços.

Os microserviços irão ser executados em contêineres, e serão serviços desacoplados. Devido a sua arquitetura utilizando microserviços, o middleware será modularizado.

A arquitetura do middleware está descrita na Figura 3.1. O cliente irá consumir os serviços fornecidos por cada servidor de forma direta ou indireta.

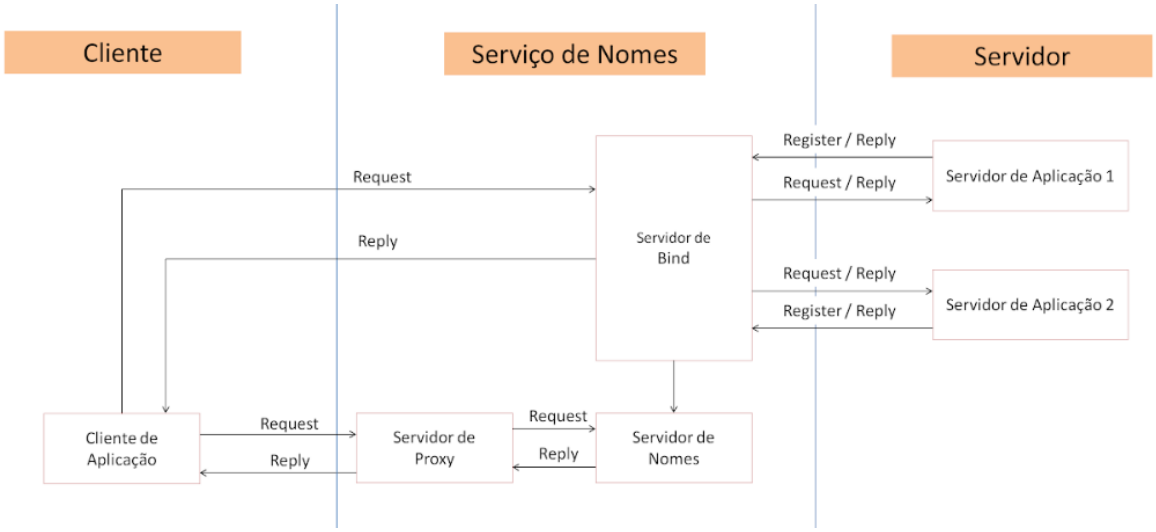

Figura 3.1 Diagrama de arquitetura do middleware

Para compreender a arquitetura do middleware, vamos analisar seu funcionamento. Primeiramente o servidor de nomes é executado. Ele será responsável por registrar as aplicações disponíveis e que serão acessadas a partir dos seus respectivos nomes. Além disso, o servidor de nomes irá fornecer ao cliente, de forma indireta, a referência para o servidor que receberá a solicitação da execução da aplicação oriunda do cliente.

Com o servidor de nomes em execução, o servidor de *bind* é executado, e será responsável por intermediar a solicitação do registro vinda dos servidores das aplicações, e o servidor de

nomes. O servidor de *bind* irá armazenar um registro para cada aplicação de acordo com seu nome e um identificador único. Ao registar as aplicações no servidor de nomes, o servidor de *bind* irá passar como parâmetros o nome da aplicação e sua própria referência como fornecedor daquele serviço. Importante observar neste ponto que, para todos clientes, o servidor de *bind* sempre será a referência para solicitar a execução da aplicação. Ao receber uma solicitação, o servidor de *bind* irá encaminhá-la para o servidor da aplicação correspondente, além de receber o resultado da execução e encaminhá-lo para o cliente.

Após colocar o servidor de *bind* em execução, os servidores das aplicações são executados. Cada servidor irá solicitar ao servidor de *bind* o registro de seu serviço, passando para ele como parâmetros o nome da aplicação, um identificador e a referência para acessar seu serviço.

Com o serviço da aplicação pronto, o servidor de *proxy* é executado. Ele é responsável por fornecer ao cliente, a referência do servidor que irá receber a solicitação da execução da aplicação. Esta referência é obtida de acordo com o nome da aplicação solicitada pelo próprio cliente. O servidor de proxy irá solicitar está referência ao servidor de nomes, e retornar esta resposta ao cliente solicitante. Com isso, todo serviço do lado do cliente está pronto e aguardando uma solicitação de execução.

Agora, com todos os serviços em execução, colocamos o cliente para executar. O cliente irá solicitar ao servidor de *proxy*, a referência para o servidor da aplicação que irá solicitar a execução da aplicação. Assim que o cliente obtém esta referência, a solicitação da execução da aplicação é iniciada. O servidor de *bind* irá encaminhar esta requisição ao servidor da aplicação propriamente dito, para executar a solicitação do cliente.

É importante observar como a arquitetura do middleware atende aos requisitos descritos nas Tabelas 3.1 e 3.2. Para atender ao requisito RF01 os componentes do middleware se comunicam através de endereços IP/PORTA e através disso, se comunicam para transportar as mensagens.

Ao executar o servidor de cada aplicação, o registro de sua aplicação pelo nome é feito no servidor de nomes atendendo ao requisito RF02. Este registro é intermediado pelo servidor de *bind*. Através deste registro, o middleware irá fornecer as AOR's atendendo ao RF03.

Atendendo o requisito RF04, para o cliente obter as AOR's utilizará um servidor de *proxy* que se comunica com o servidor de nomes. Além disso, para o cliente enviar sua solicitação para o servidor da aplicação, sempre será por meio do servidor de *bind*.

Devido às propriedades dos microserviços descritas na Seção 2.3, ao desenvolver os componentes do middleware como microserviços atendemos aos requisitos RNF01, RNF02 e RNF03.

3.3 Projeto e Implementação

Para definir este projeto, primeiramente foi preciso decidir em que plataforma iríamos criar os microserviços. Devido a sua grande popularidade, facilidade e grande comunidade na Internet, escolhemos o Docker para ser a plataforma utilizada para criar os microserviços. Foi preciso definir quais *Remoting Patterns* seriam implementados como microserviço. Decidimos desenvolver o middleware com 3 microserviços tal qual foi descrito na Figura 3.1.

O diagrama de sequência do registro do servidor da aplicação no servidor de nomes pode ser visto na Figura 3.2. Nele, podemos observar o fluxo do registro do servidor da aplicação no servidor de nomes. O servidor da aplicação irá solicitar ao servidor de *bind* o seu registro

através do nome da aplicação. O servidor de *bind* por sua vez, irá se registrar no servidor de nomes, como fornecedor daquele serviço.

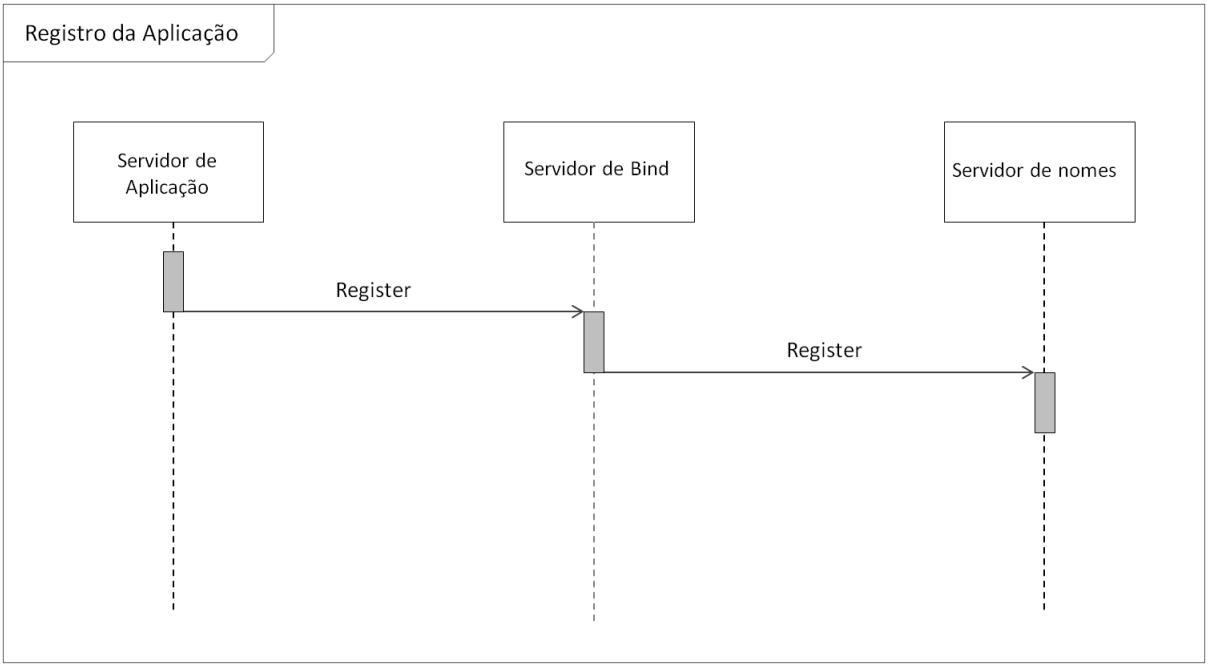

Figura 3.2 Diagrama de sequência do registro do Servidor da Aplicação no Servidor de Nomes

O diagrama de sequência de uma requisição do cliente está descrito na Figura 3.3. Com todos os microserviços em execução, o cliente solicita a execução da operação no servidor da aplicação. Ele irá obter a referência do serviço através do servidor de *proxy*. Por sua vez, o servidor de *proxy* irá solicitar ao serviço de nomes a referência para o serviço solicitado. Com a referência, o cliente solicita a execução da aplicação ao servidor de *bind*, que repassa a solicitação para o servidor da aplicação para executar o serviço. O servidor da aplicação devolve o resultado ao servidor de *bind*, que repassa a resposta para o cliente da aplicação.

Para iniciar o desenvolvimento do middleware criamos um contêiner no *docker* para executar o serviço de nomes. O primeiro passo foi criar um arquivo de configuração *dockerfile* que contém todos os comandos que normalmente são executados manualmente para criar uma imagem *docker*. Através deste arquivo, o *docker* pode criar imagens automaticamente[2]. Este mesmo arquivo foi utilizado para criar todas imagens no *docker* dos microserviços do middleware.

Instruções comuns do *dockerfile* começam com RUN, ENV, FROM, MAINTAINER, ADD, e CMD, entre outros [8]:

1. FROM: Especifica a imagem de base que o *dockerfile* usará para criar uma nova imagem. Para esta trabalho, estamos usando o *phusion / baseimage* como nossa imagem base porque é uma imagem mínima baseada em Ubuntu modificada para a sintetização *docker*.

2. MAINTAINER: Especifica o nome do autor do *dockerfile* e seu e-mail.

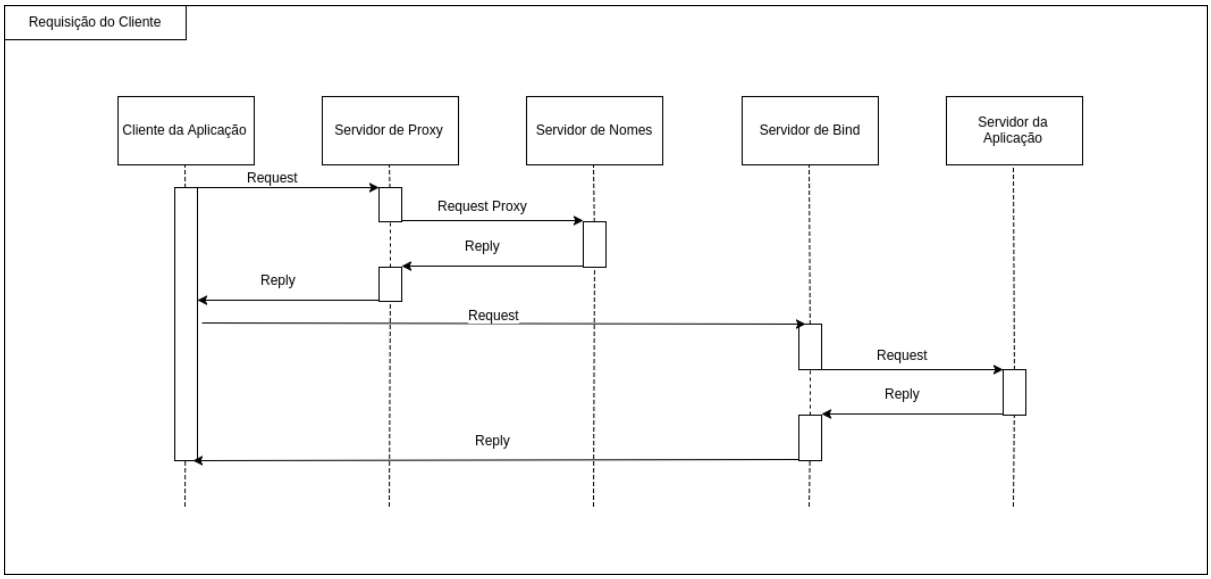

Figura 3.3 Diagrama de sequência da solicitação do cliente da aplicação

3. RUN: Executa algum comando UNIX para construir a imagem.

4. ENV: Define as variáveis de ambiente. Para esta trabalho, *javahome* é a variável que está definida.

5. CMD: Fornece facilidade para executar comandos no início do container. Isso pode ser substituído ao executar o comando *docker run*.

6. ADD: Esta instrução copia os novos arquivos, diretórios no sistema de arquivos de container *docker* no destino especificado.

7. EXPOSE: Esta instrução expõe a porta especificada para a máquina host.

Utilizando estes comandos, foi criado o arquivo *dockerfile* para execução deste projeto, como mostrado na Figura 3.4.

Cada contêiner criado através do *docker* possui um endereço IP específico, e será através da combinação IP/PORTA que os contêineres irão se comunicar.

O servidor de nomes criado no projeto é responsável pelos serviços de *bind* e *lookup* do middleware. Ele é executado como um microserviço, em um contêiner no *docker*. Seu arquivo *main* está descrito no Código 3.1. Em sua execução, este microserviço passa a escutar na porta 1313 como podemos ver na linha 09 do código, por onde as requisições irão chegar. Na linha 06 do código, criamos o objeto responsável por executar o *bind* ou o *lookup* do servidor de nomes. Ao receber uma requisição, o servidor de nomes executa o *bind* ou o *lookup*.

Código 3.1 Servidor de nomes

```
1 import java.io.IOException;
```

```
# Dockerfile

FROM  phusion/baseimage:0.9.17

MAINTAINER  Rubem Moura <rfmj2@cin.ufpe.br>

RUN echo "deb http://archive.ubuntu.com/ubuntu trusty main universe" >
        /etc/apt/sources.list

RUN apt-get -y update

RUN DEBIAN_FRONTEND=noninteractive apt-get install -y -q python-software-properties
    software-properties-common

ENV JAVA_VER 8
ENV JAVA_HOME /usr/lib/jvm/java-8-oracle

RUN echo 'deb http://ppa.launchpad.net/webupd8team/java/ubuntu trusty main' >>
        /etc/apt/sources.list && \

    echo 'deb-src http://ppa.launchpad.net/webupd8team/java/ubuntu trusty main' >>
        /etc/apt/sources.list && \

    apt-key adv --keyserver keyserver.ubuntu.com --recv-keys C2518248EEA14886 && \
        apt-get update && \

    echo oracle-java${JAVA_VER}-installer shared/accepted-oracle-license-v1-1
        select true | sudo /usr/bin/debconf-set-selections && \

    apt-get install -y --force-yes --no-install-recommends
        oracle-java${JAVA_VER}-installer oracle-java${JAVA_VER}-set-default && \

    apt-get clean && \
    rm -rf /var/cache/oracle-jdk${JAVA_VER}-installer

RUN update-java-alternatives -s java-8-oracle

RUN echo "export JAVA_HOME=/usr/lib/jvm/java-8-oracle" >> ~/.bashrc

RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

CMD ["/sbin/my_init"]
```

Figura 3.4 Arquivo *dockerfile* utilizado

```
2
3  public class NamingServer {
4
5    public static void main(String[] args) throws IOException,
        Throwable {
6      NamingInvoker invoker = new NamingInvoker();
7
8      System.out.println("Servidor de nomes esta pronto...");
9      invoker.invoke(1313);
10   }
11 }
```

Ao executar o *bind*, conforme Código 3.2, o servidor de nomes registra o nome do serviço e armazena-o na variável `register p1`, e armazena um *proxy*, associado ao microserviço do requisitante que é obtido na variável `register p2` do código. Em outras palavras, quando um microserviço requisitar a execução do *bind* no servidor de nomes, ele irá passar um objeto *proxy* e o nome do serviço no qual ele está registrando no servidor de nomes.

Código 3.2 Execução do *bind* do Servidor de Nomes

```
1  case "bind":
2    System.out.println("Servidor de nomes - Executando BIND " + i);
3    // @ Invokes the remote object
4    String register_p1 = msgUnmarshalled.getBody().getRequestBody()
5        .getParameters().get(0).toString();
6    ClientProxy register_p2 = (ClientProxy) msgUnmarshalled
7        .getBody().getRequestBody().getParameters().get(1);
8    rObj.bind(register_p1, register_p2);
9    ter.setResult("");
10
11   Message msgToBeMarshalled_register = new Message(
12       new MessageHeader("protocolo", 0, false, 0, 0),
13       new MessageBody(null, null, new ReplyHeader("", 0, 0),
14           new ReplyBody(ter.getResult())));
15
16   // @ Marshall the response
17   msgMarshalled = mrsh.marshall(msgToBeMarshalled_register);
18
19   // @ Send response
20   srh.send(msgMarshalled);
21   i++;
22   break;
```

Ainda no servidor de nomes, conforme o Código 3.3, quando um serviço solicitar a execução do *lookup*, o requisitante passa o nome do serviço desejado. O servidor de nomes obtém este nome na linha 05, e na linha 07 define o resultado com o *proxy* que referencia o servidor da aplicação registrado com aquele nome.

Código 3.3 Execução do *lookup* do Servidor de Nomes

```
case "lookup":

  System.out.println("Servidor de nomes - Executando LOOKUP");
  // @ Invokes the remote object
  String bind_p1 = msgUnmarshalled.getBody().getRequestBody()
       .getParameters().get(0).toString();
  ter.setResult(rObj.lookup(bind_p1));

  // @ Prepare message to be sent back to client
  Message msgToBeMarshalled_bind = new Message(new MessageHeader(
       "protocolo", 0, false, 0, 0), new MessageBody(null,
       null, new ReplyHeader("", 0, 0), new ReplyBody(
            ter.getResult())));

  // @ Marshall the response
  msgMarshalled = mrsh.marshall(msgToBeMarshalled_bind);

  // @ Send response
  srh.send(msgMarshalled);
  break;
```

Após a criação do servidor de nomes, iremos executá-lo como microserviço dentro de um contêiner no *docker*.

O próximo microserviço a ser executado é o servidor de *bind*. Ele é responsável por receber as requisições de solicitações de *bind* vinda dos servidores das aplicações e encaminha-las ao servidor de nomes, assim como também é responsável por receber as requisições de execução da aplicação vinda do cliente da aplicação, encaminhando esta requisição para o servidor da aplicação.

Código 3.4 Servidor de *bind*

```
import java.io.IOException;

public class BindServer {

  public static void main(String[] args) throws IOException,
      Throwable {
    // remote object
    // obtain instance of Naming Service
    // register the application in Naming service
    BindInvoker invoker = new BindInvoker();

    System.out.println("O Bind Server esta em execucao.");
    invoker.invoke(1316);
  }
}
```

O código do servidor de *bind* está representado no Código 3.4. Como podemos ver, ao executar este servidor, seu contêiner irá escutar as requisições na porta 1316. Através dele, um servidor de uma aplicação qualquer, que deseje registrar seu serviço no servidor de nomes, irá solicitar este registro executando o comando *bind* do objeto que aponta para este servidor de *bind*. Na linha 09 do código, criamos o objeto que irá executar os comandos do servidor de *bind*. A execução do comando *bind* está representada no Código 3.5. Na linha 04 criamos a referência para o próprio servidor de *bind*. Já na linha 05 do código obtemos o nome da aplicação solicitante.

Código 3.5 Código da execução do *bind* no servidor de *bind*

```
1  case "bind":
2    // @ Invokes the remote object
3    NamingProxy namingService = new NamingProxy("172.17.0.2", 1313);
4    bindServer = new BindServerProxy("172.17.0.3",1316);
5    String serviceName = (String)
        msgUnmarshalled.getBody().getRequestBody().getParameters()
6       .get(1);
7
8    System.out.println(" - Executando bind no ServerBind da
        aplicacao: " + serviceName);
9
10   if((int)
        msgUnmarshalled.getBody().getRequestBody().getParameters()
11      .get(0) == 1) {
12     System.out.println("Entrou app 1");
13     applicationOne = (ReverserProxy)
          msgUnmarshalled.getBody().getRequestBody().getParameters()
14        .get(2);
15   } else {
16     System.out.println("Entrou app 0");
17     applicationTwo = (ReverserProxy)
          msgUnmarshalled.getBody().getRequestBody().getParameters()
18        .get(2);
19   }
20
21   namingService.bind(serviceName, bindServer);
22
23   //t1 = QoSObserver.getTime();
24   ter.setResult(serviceName);
25   //t2 = QoSObserver.getTime();
26   //QoSObserver.saveTime(t2-t1);
27
28   Message _add_msgToBeMarshalled = new Message(new MessageHeader(
29        "protocolo", 0, false, 0, 0), new MessageBody(null,
```

```
30          null, new ReplyHeader("", 0, 0), new ReplyBody(
31              ter.getResult())));
32
33    // @ Marshall the response
34    msgMarshalled = mrsh.marshall(_add_msgToBeMarshalled);
35
36    // @ Send response
37    srh.send(msgMarshalled);
38    break;
```

Ao executar o *bind*, o servidor de *bind* executa uma chamada para o servidor de nomes na linha de código 21, solicitando que execute o *bind* no servidor de nomes, passando como *proxy* o objeto que referencia o próprio servidor de *bind*, criado na linha de código 04. Em outras palavras, o servidor de *bind* sempre registra um novo serviço no servidor de nomes, passando um objeto *proxy* que aponte para ele mesmo. Ele será o servidor que contém todos os registros de todos servidores de aplicações.

Ainda no servidor de *bind*, quando um cliente da aplicação solicitar a execução do serviço, ele executará serviço através de um objeto de *proxy* que referencia o servidor de *bind*. Este objeto será obtido através do servidor de *proxy*. Quando o servidor de *bind* recebe a solicitação da execução do serviço pelo cliente, o Código 3.6 será executado.

Neste trecho de código na linha 06, o servidor de *bind* irá determinar para qual servidor da aplicação irá encaminhar a chamada do cliente. Em seguida, na linha 08 e 11, ele executa a chamada da aplicação com o objeto *proxy* que contém a referência para o servidor de aplicação correspondente à solicitação do cliente, e o servidor da aplicação executa a aplicação.

Código 3.6 Código da execução do serviço no servidor de *bind*

```
1  case "inverter":
2    // @ Invokes the remote object
3    StringBuffer string = (StringBuffer)
        msgUnmarshalled.getBody().getRequestBody().getParameters()
4       .get(0);
5
6    if((int)
        msgUnmarshalled.getBody().getRequestBody().getParameters()
7       .get(1) == 1) {
8      ter.setResult(applicationOne.inverter(string));
9      System.out.println(" - Invocando Inverter no ServerBind da
           aplicacao: 1");
10   } else {
11     ter.setResult(applicationTwo.inverter(string));
12     System.out.println(" - Invocando Inverter no ServerBind da
           aplicacao: 0");
13   }
14
15   //t1 = QoSObserver.getTime();
```

```
16    //ter.setResult(serviceName2);
17    //t2 = QoSObserver.getTime();
18    //QoSObserver.saveTime(t2-t1);
19
20    Message _add_msgToBeMarshalled2 = new Message(new MessageHeader(
21         "protocolo", 0, false, 0, 0), new MessageBody(null,
22         null, new ReplyHeader("", 0, 0), new ReplyBody(
23              ter.getResult())));
24
25    // @ Marshall the response
26    msgMarshalled = mrsh.marshall(_add_msgToBeMarshalled2);
27
28    // @ Send response
29    srh.send(msgMarshalled);
30    break;
```

Com estes microserviços em execução, uma aplicação pode se cadastrar no servidor de nomes, fornecendo seu serviço.

O próximo microserviço é o servidor de *proxy*, que irá fornecer ao cliente a AOR que foi armazenada no servidor de nomes. O cliente irá solicitar ao servidor de *proxy* o endereço para o qual ele irá solicitar a execução da aplicação. O Código 3.7 mostra a implementação do servidor de *proxy*. Nela podemos ver que ele irá receber as requisições através da porta 1315 na linha 10 do código. Na linha 07 criamos o objeto que irá executar os comandos do servidor de *proxy*.

Código 3.7 Servidor de *proxy*

```
1  import java.io.IOException;
2  import java.net.UnknownHostException;
3
4  public class ProxyClient {
5
6    public static void main(String[] args) throws
        UnknownHostException, IOException, Throwable {
7      ProxyInvoker invoker = new ProxyInvoker();
8
9      System.out.println("Servico de Proxy esta pronto...");
10     invoker.invoke(1315);
11   }
12 }
```

Ao receber uma requisição vinda do cliente da aplicação, o servidor de *proxy* irá fazer uma requisição ao servidor de nomes como podemos ver no Código 3.8 na linha 11 e 13. Neste momento, o servidro de proxy está solicitando a referência para o servidor que, para o cliente, será o fornecedor do serviço. O servidor de nomes irá retornar para o servidor de *proxy* a referência para o servidor de *bind*, que registrou a aplicação.

Código 3.8 Código da execução do servidor de *proxy* ao receber uma requisição

```
case "proxy":
  System.out.println(" - Executando proxy client.");
  // @ Invokes the remote object
  BindServerProxy reverserProxy;
  NamingProxy namingProxy = (NamingProxy)
     msgUnmarshalled.getBody().getRequestBody().getParameters()
     .get(0);
  int applicationType = (int)
     msgUnmarshalled.getBody().getRequestBody().getParameters()
     .get(1);

  if(applicationType == 1) {
    reverserProxy = (BindServerProxy)
       namingProxy.lookup("Reverser");
  } else {
    reverserProxy = (BindServerProxy)
       namingProxy.lookup("ReverserTwo");
  }

  //t1 = QoSObserver.getTime();
  ter.setResult(reverserProxy);
  //t2 = QoSObserver.getTime();
  //QoSObserver.saveTime(t2-t1);

  Message _add_msgToBeMarshalled = new Message(new MessageHeader(
       "protocolo", 0, false, 0, 0), new MessageBody(null,
       null, new ReplyHeader("", 0, 0), new ReplyBody(
            ter.getResult())));

  // @ Marshall the response
  msgMarshalled = mrsh.marshall(_add_msgToBeMarshalled);

  // @ Send response
  srh.send(msgMarshalled);
  break;
```

Com isso, teremos o middleware em execução, aguardando a requisição do cliente para intermediá-la com o servidor da aplicação.

3.4 Considerações Finais

Este capítulo apresentou inicialmente os requisitos do middleware implementado e a arquitetura definida a partir destes requisitos. Em seguida, apresentamos as principais decisões de projeto e detalhes de implementação dos microserviços que compõem o middleware.

CAPÍTULO 4

Avaliação de Desempenho

Neste capítulo será realizada uma avaliação de desempenho do middleware desenvolvido, levando em consideração os impactos que esta arquitetura traz ao projeto. A metodologia utilizada para esses experimentos é baseada em Raj Jain (1991).

4.1 Objetivos

O objetivo desta avaliação é comparar o desempenho do middleware desenvolvido com a arquitetura de microserviços ou não.

Como o objetivo é avaliar o desempenho do middleware, a aplicação usada para avaliá-lo é um inversor de caracteres de um *string*. Esta aplicação é disponibilizada e pode ser acessada remotamente por clientes usando-se o middleware proposto.

4.2 Experimentos

O sistema consiste em 3 microserviços, 2 servidores da aplicação e 1 cliente como foi descrito na Seção 3.2. O cliente da aplicação consome os serviços disponibilizados em cada contêiner ao solicitar a inversão da *string*.

4.2.1 Métricas

A métrica utilizada neste experimento é o tempo decorrido entre solicitar a inversão do *string* e obter o resultado da solicitação.

A métrica será calculada da seguinte forma: duas variáveis locais *t1* e *t2* são criadas no código do cliente. A variável *t1* irá armazenar o valor da hora no exato momento antes de executar as requisições ao serviço. A variável *t2* irá armazenar o valor da hora no momento posterior a execução das solicitações do serviço. Calculando a diferença de tempo entre *t2* e *t1*, obtemos o tempo gasto total na execução do serviço as N vezes (valor que será incrementado no *loop* da solicitação do cliente). Este cálculo representa o *RTT* (*round trip time*) entre o cliente e o servidor da aplicação.

4.2.2 Parâmetros

Para realizar os experimentos, será utilizado um notebook executando 3 microserviços, 2 aplicações e um cliente no *Docker*. As configurações do notebook estão descritas na Tabela 4.1

Tabela 4.1 Configurações do notebook utilizado nos experimentos

Processador	Intel® Core™ i7-2630QM
Memória RAM	8 gb
Sistema Operacional	Ubuntu 16.04.03 LTS de 64 bits
Disco de Armazenamento	SSD MD Radeon M3 series

.

A fim de não levar em consideração fatores como a latência da rede, optou-se por conectar os microserviços através da mesma máquinas utilizando endereços IP/PORTA.

É importante ressaltar que durante a execução do experimento, apenas o middleware e a aplicação estão sendo executados, além do notebook estar sempre conectado à fonte de energia.

Além da configuração da máquina, também é necessário definir os parâmetros que controlam a carga de trabalho realizada pela aplicação. Esses parâmetros são fixos para o experimento. Nesse contexto, são definidos os seguintes parâmetros:

1. Intervalo entre o envio de cada solicitação do serviço pelo cliente: 5 milisegundos

2. Tamanho (em bytes) da *string* enviada pelo cliente ao servidor: 48 bytes

4.2.3 Fatores

Os fatores são aqueles parâmetros variados nos experimentos. Os fatores que iremos analisar neste experimento são:

1. A quantidade de memória RAM utilizada por cada contêiner no momento da execução de cada microserviço: Caso a quantidade de memória RAM disponível para cada contêiner seja completamente utilizada, o *Docker* irá utilizar o disco rígido para fazer paginação.

2. A quantidade de CPUs utilizadas na execução de cada contêiner

É válido salientar que o middleware foi executado sem restrições de quantidade de memória RAM e verificou-se que todos microserviços não necessitavam mais do que 100mb de memória, e por isso, os valores para se variar a quantidade de memória foram todos menores do que 100mb (50mb, 45mb e 40mb). Para todos os valores de memória RAM, limitamos também a quantidade de memória *swap*. A memória *swap* é uma memória virtual criada pelo sistema operacional utilizando o espaço do disco rígido. Quando a quantidade de memória RAM não é suficiente para executar uma aplicação, o sistema operacional utiliza a memória *swap*. Isto também acontece no gerenciamento de memória utilizada por um contêiner no Docker.

Os valores escolhidos para limitar a quantidade de CPUs disponível pelos contêineres foram 4 CPUs, 3 CPUs, 2 CPUs, 1 CPU, 50% de uma CPU e 25% de uma CPU.

Como carga de trabalho utilizamos o número de invocações ao serviço variando entre 1000, 5000, 10000, 20000 e 40000 invocações.

4.2.4 Invocação do método remoto

Para analisarmos o *RTT* da invocação do método remoto pelo cliente, foi adicionado ao código um *loop* onde dentro dele fazemos as requisições ao servidor, conforme podemos ver na Linha 4 do Código 4.1. Neste *loop* incrementamos a quantidade de interações conforme descrito anteriormente em valores de 1000 à 40000 interações na Linha 03, fazendo com que o cliente execute essa quantidade de invocações ao método remoto no servidor da aplicação. Para este experimento, a quantidade limite de memória não foi determinado e com isso, os microserviços utilizavam a quantidade de memória que fosse desejável.

Código 4.1 Invocação do método remoto

```
1 time1 = System.nanoTime();
2 StringBuffer str = new StringBuffer("Programacao concorrente
     Distribuida - Midlleware");
3 for(int i = 0; i<10000; i++) {
4   str = reverserProxy.inverter(str,1);
5   System.out.println(i + " " + str);
6   Thread.sleep(5);
7 }
8 time2 = System.nanoTime();
```

Os resultados com os tempos de execuções para cada middleware de acordo com a quantidade de invocações está descrido na Figura 4.1.

Na Figura 4.1, podemos observar que para todas as quantidades de invocações o middleware com microserviços levou mais tempo para realizar as invocações. Isso ocorreu devido ao número de requisições ter aumentado quando desenvolvemos o middleware com microserviços. Por exemplo, no middleware com microserviços, para o cliente obter o *proxy* para executar o método remoto, precisa passar pelo servidor de *proxy*. No middleware sem microserviço isso não era necessário e o cliente obtinha o *proxy* acessando o servidor de nomes diretamente. Além disso, para o cliente executar o método remoto, ele precisa passar pelo servidor de *bind*, o que antes não era necessário.

4.2.5 Invocação do método remoto limitando a quantidade de memória

Para realizarmos o segundo experimento, foi limitada a quantidade de memória RAM de cada microserviço em execução. A quantidade máxima disponível de memória para eles utilizarem, primeiramente foi de 50mb. Para limitarmos, executamos o contêiner com um parâmetro adicional no comando de execução. O parâmetro passado é o *-m 50m*. Além disso, a quantidade de memória *swap* também foi limitada para mesma quantidade de memória RAM disponível, através do parâmetro *–memory-swap -1* [3].

Para os outros valores das quantidades de memória, foi executado o mesmo procedimento quando limitado à 50mb. Os outros limites foram 40mb e 45mb. A Figura 4.2 mostra a comparação do tempo do middleware com microserviço, sendo executado com limites de memória de 50mb em comparação com toda memória disponível pelo computador (8gb), assim como, o gráfico da Figura 4.3 mostra a comparação com limites de memória de 45mb e o gráfico da

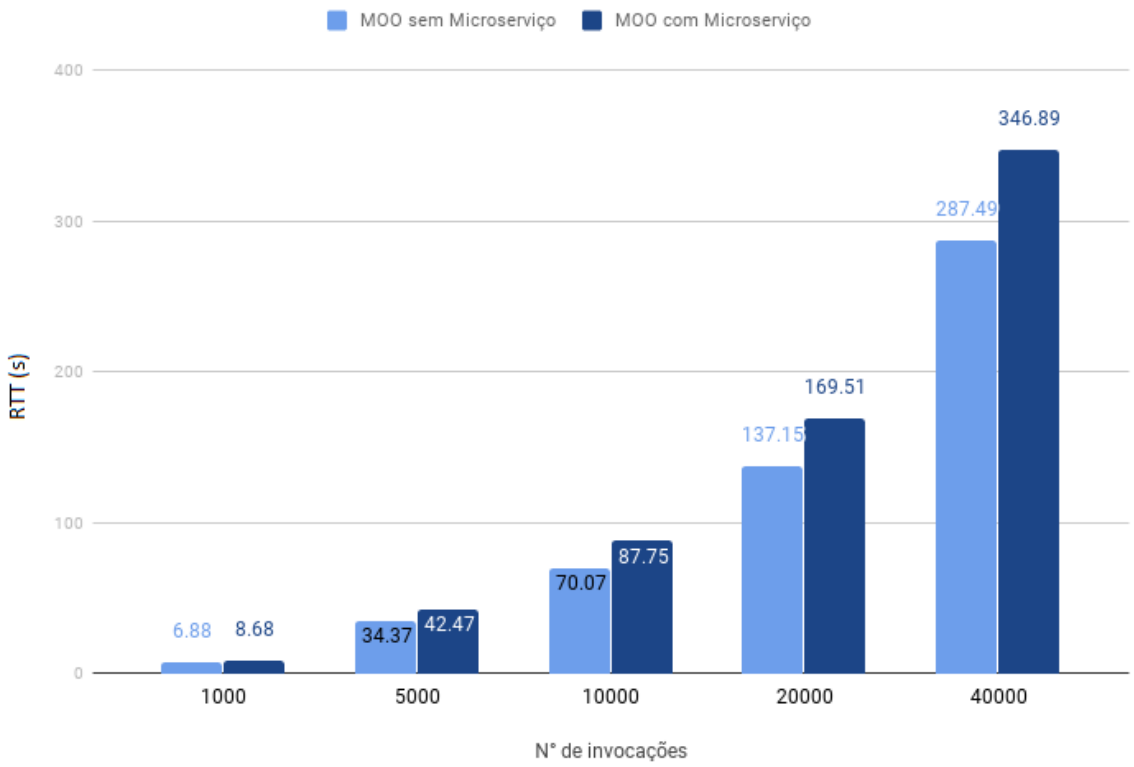

Figura 4.1 Tempo gasto em função da quantidade de invocações realizadas pelos middlewares com microserviço e sem.

Figura 4.4 mostra a comparação com limites de memória de 40mb.

O comando *sudo docker stats* foi utilizado para visualizarmos o uso da memória durante a execução das requisições do cliente da aplicação ao serviço disponível. O resultado deste comando pode ser visualizado na Figura 4.5.

Podemos ter uma melhor visualização da comparação do middleware sendo executado com limites de memória no gráfico da Figura 4.6. Para todos os limites de memória definidos, o middleware com total memória disponível teve um desempenho melhor. Isso acontece devido ao tempo gasto do sistema operacional utilizar a memória *swap* para executar o contêiner. Como a memória *swap* é mais lenta que a memória RAM, devido a utilização do disco rígido, o tempo da execução é maior quando limitamos a quantidade de memória utilizada pelo contêiner.

É importante mencionar que, para valores menores de 40mb, não obtivemos sucesso na execução do middleware. Os microserviços eram sobrecarregados e necessitavam de mais memória, gerando erros de conexão e *IOException* do Java.

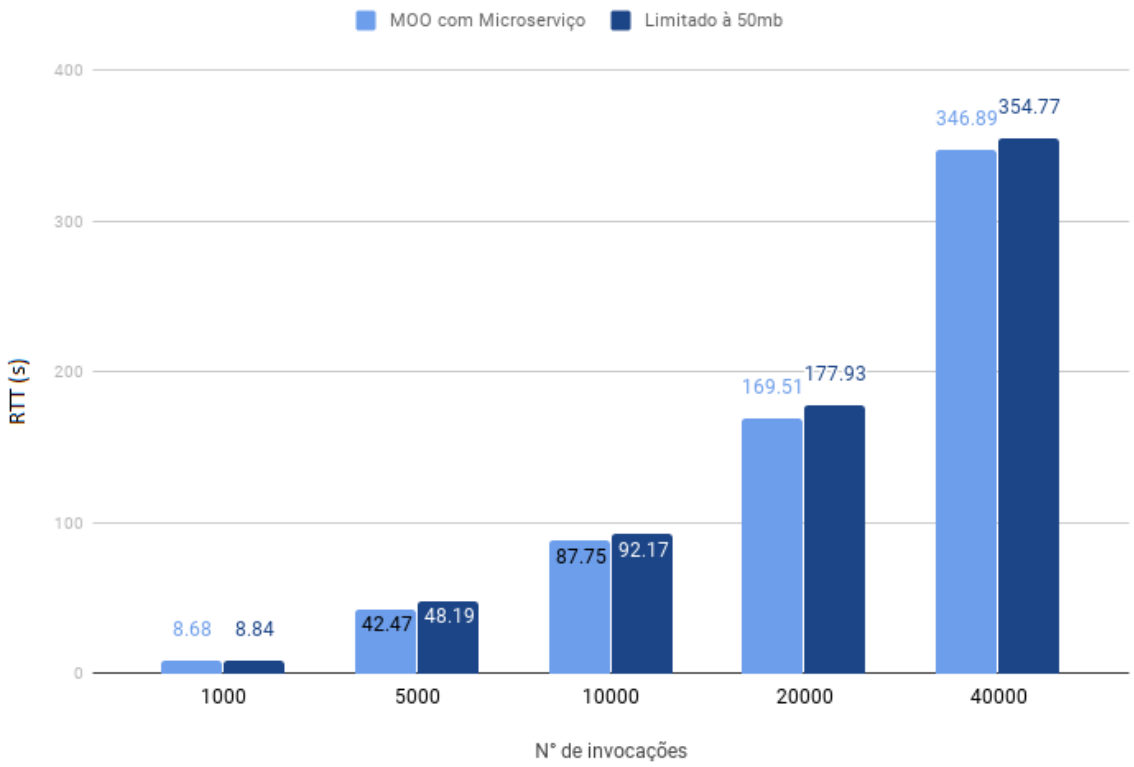

Figura 4.2 Tempo gasto em função da quantidade de invocações de acordo com middleware com limite de 50mb de memória disponível e sem limites.

4.2.6 Invocação do método remoto limitando a quantidade de CPUs

Para realizarmos este experimento, foi limitada a quantidade de CPUs disponíveis para execução do contêiner. A quantidade máxima de CPUs disponíveis pelo processador utilizado foi de 4 CPUs. Para limitarmos, executamos o contêiner com um parâmetro à mais no comando de execução. O parâmetro passado é o *–cpus="4"*, onde 4 é o número de CPUs utilizadas.

Para limitar a quantidade de CPUs disponíveis com os outros valores definidos, foi executado o mesmo comando quando limitado à 4 CPUs, porém com valores de CPUs diferentes. A Figura 4.7 mostra a comparação do tempo de execução do middleware com microserviço utilizando as 4 CPUs e apenas 3 CPUs disponibilizadas pelo processador. Já a Figura 4.8 mostra a comparação do tempo de execução do middleware com microserviço utilizando as 4 CPUs e apenas 2 CPUs disponibilizadas pelo processador, assim como a Figura 4.9 mostra a comparação utilizando as 4 CPUs e apenas 1 CPU. A Figura 4.10 mostra a comparação utilizando 4 CPUs e apenas 50% da capacidade de uma CPU, assim como a Figura 4.11 mostra a comparação utilizando 4 CPUs e apenas 25% da capacidade de uma CPU.

Para todos os valores definidos de limites de CPU, o *RTT* foi maior que utilizando as 4 CPUs disponíveis pelo processador. Isso ocorre devido ao desempenho do processador ser

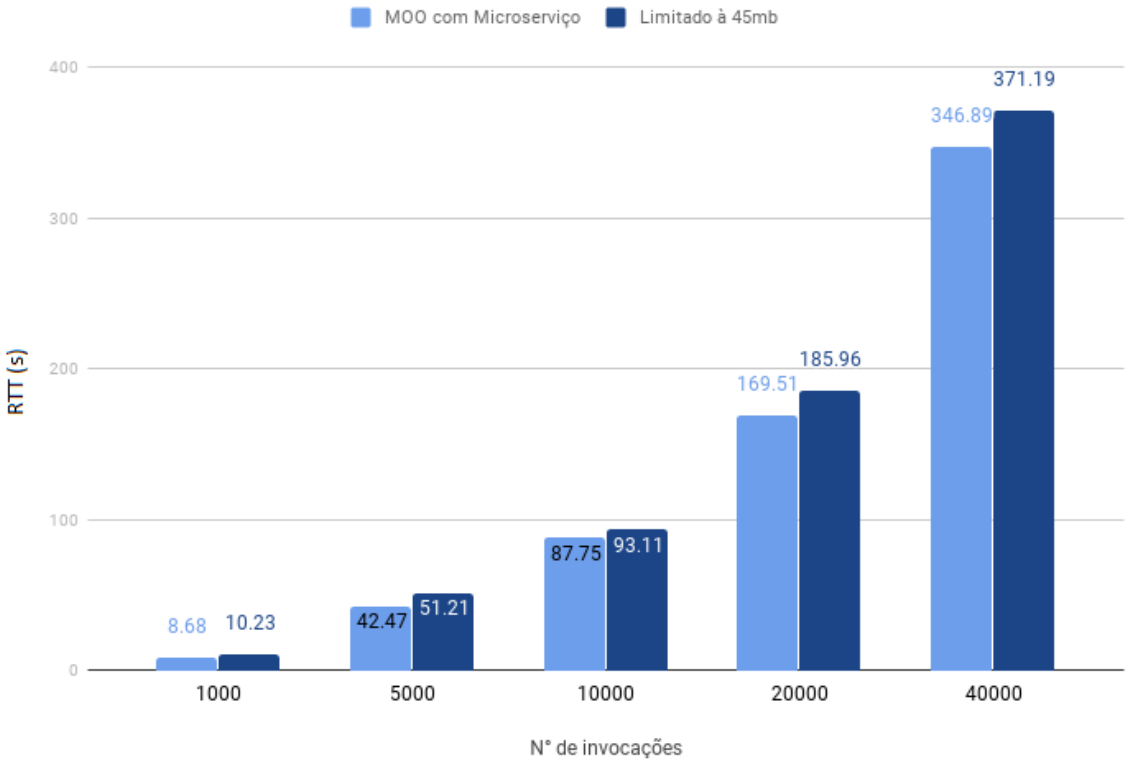

Figura 4.3 Tempo gasto em função da quantidade de invocações de acordo com middleware sem limites de memória e com limite de 45mb de memória.

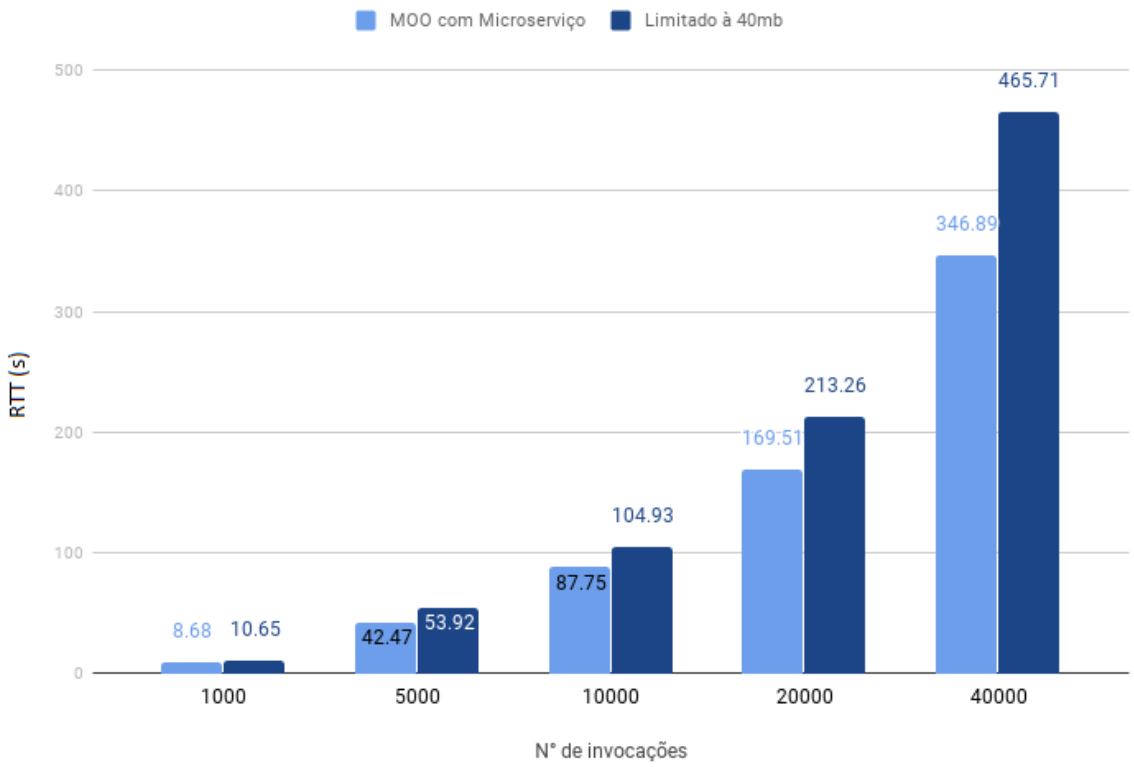

Figura 4.4 Tempo gasto em função da quantidade de invocações de acordo com middleware sem limites de memória e com limite de 40mb de memória.

limitado, e quanto menor o desempenho, maior será o tempo necessário para executar o mesmo processo. É importante mencionar que mesmo com apenas 25% de uma CPU, o middleware conseguiu concluir a requisição do cliente.

4.2.7 Teste de escalabilidade

Para realizarmos o teste de escalabilidade, foi adicionado mais um servidor de *bind* ao middleware. Este segundo servidor de *bind* é iniciado quando simulamos uma sobrecarga no pri-

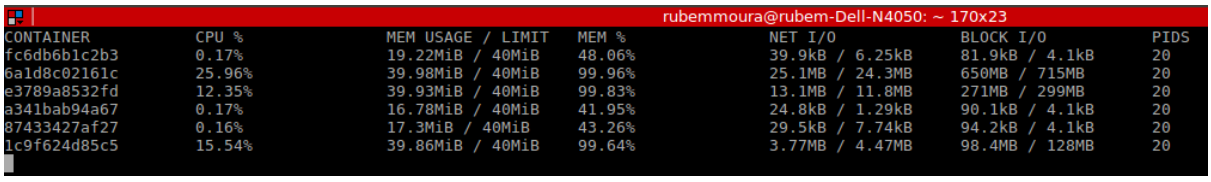

Figura 4.5 Resultado da execução do comando *sudo docker stats*.

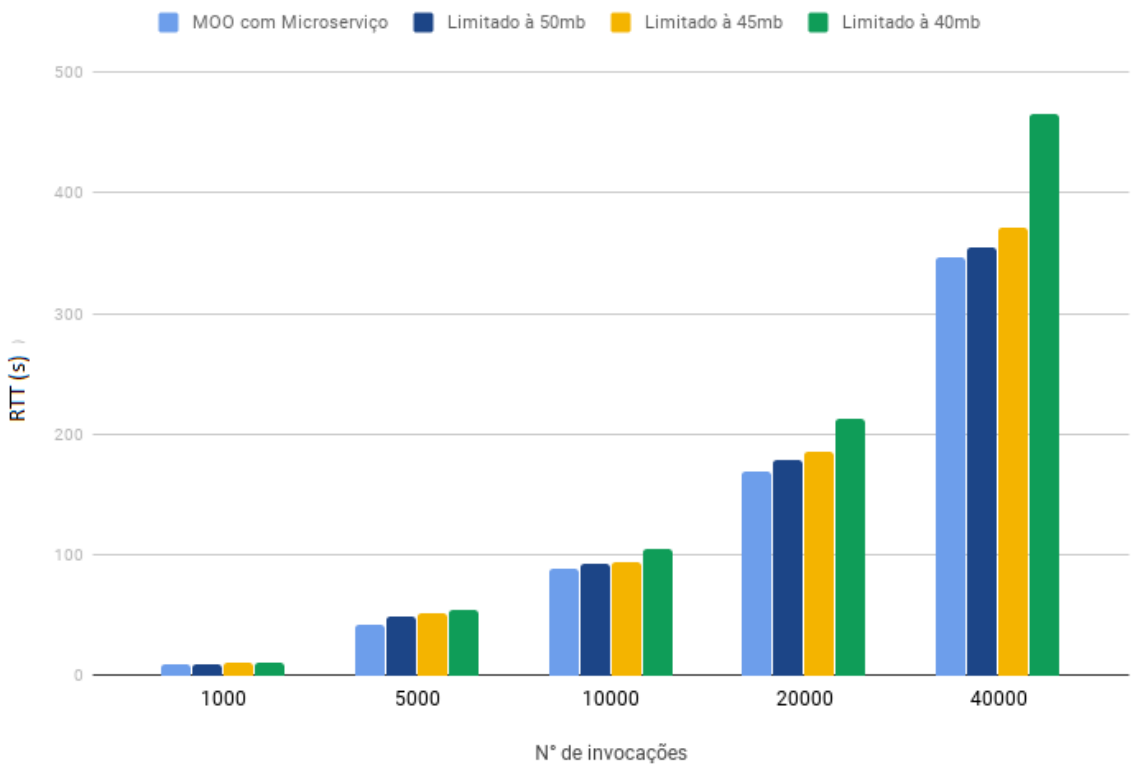

Figura 4.6 Tempo gasto em função da quantidade de invocações de acordo com middleware sem limites de memória e com todos limites anteriores.

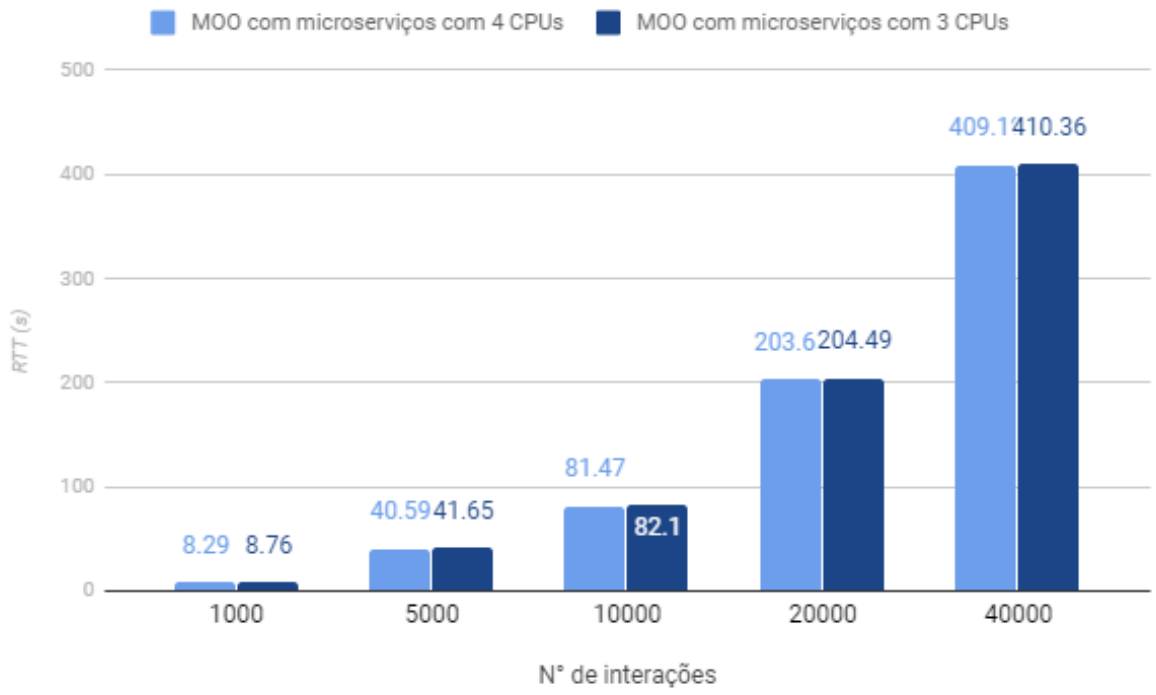

Figura 4.7 Tempo gasto em função da quantidade de invocações de acordo com middleware utilizando 4 CPUs e 3 CPUs.

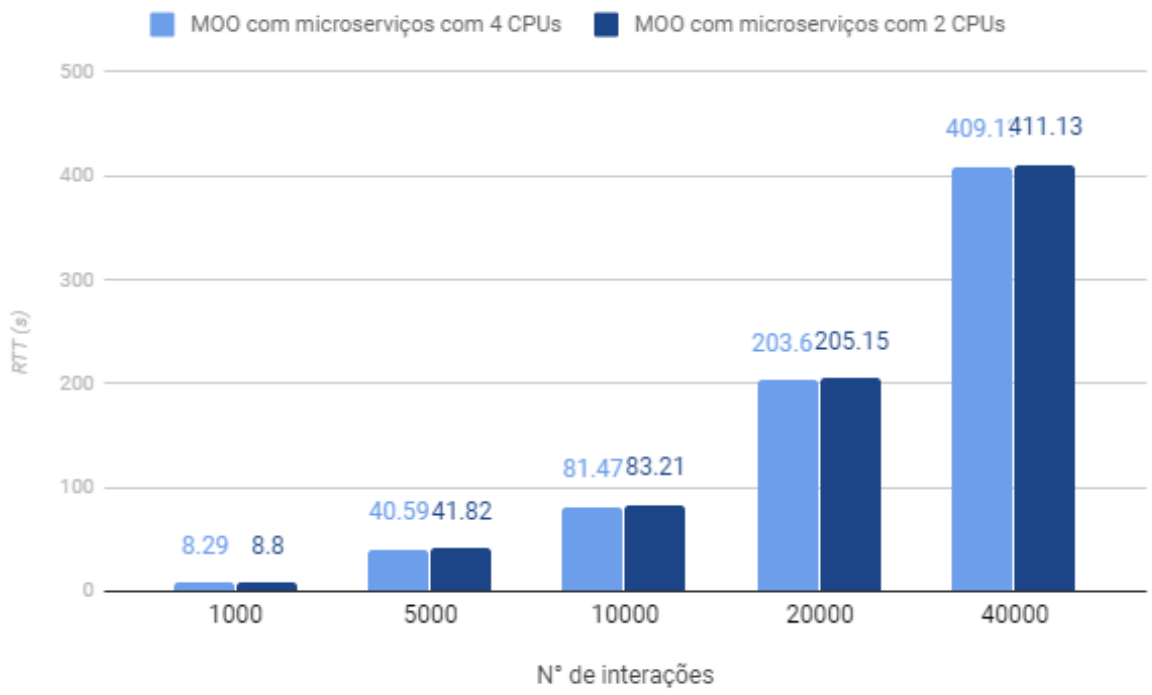

Figura 4.8 Tempo gasto em função da quantidade de invocações de acordo com middleware utilizando 4 CPUs e 2 CPUs.

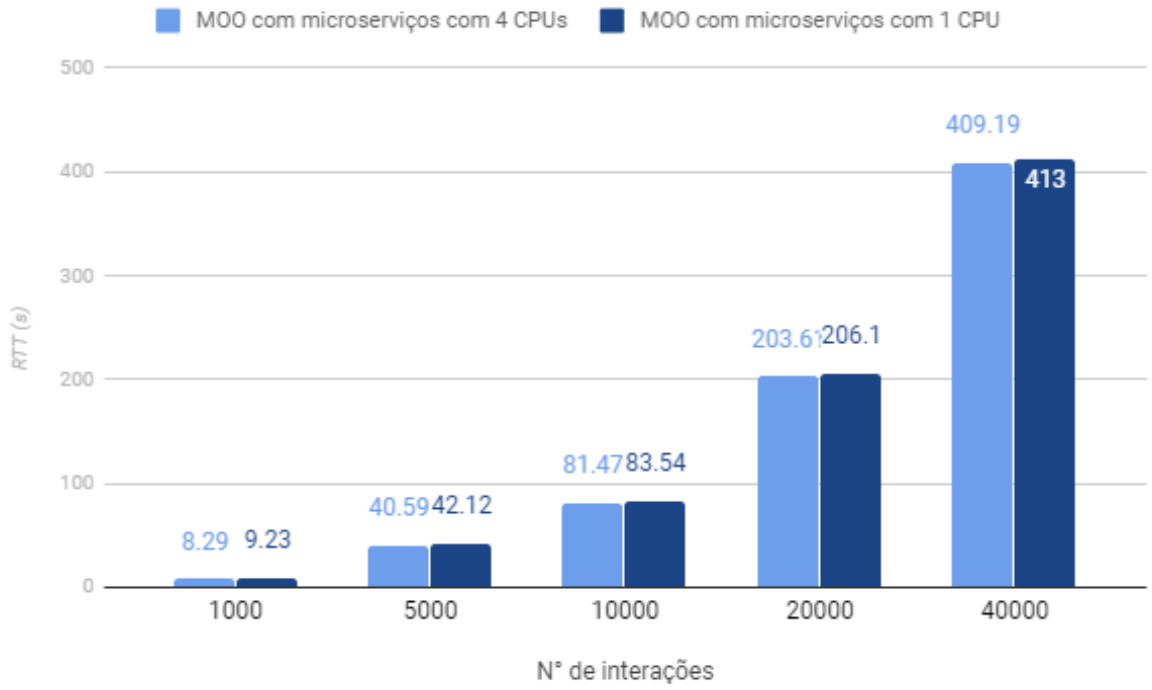

Figura 4.9 Tempo gasto em função da quantidade de invocações de acordo com middleware utilizando 4 CPUs e 1 CPU.

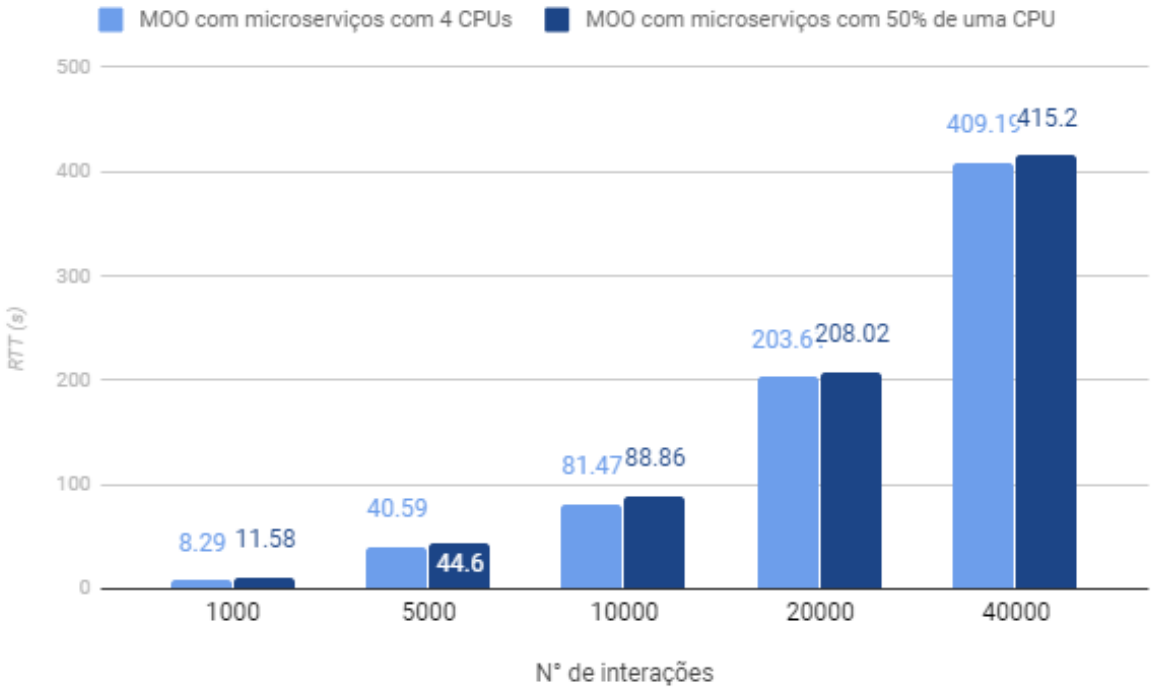

Figura 4.10 Tempo gasto em função da quantidade de invocações de acordo com middleware utilizando 4 CPUs e 50% de uma CPU.

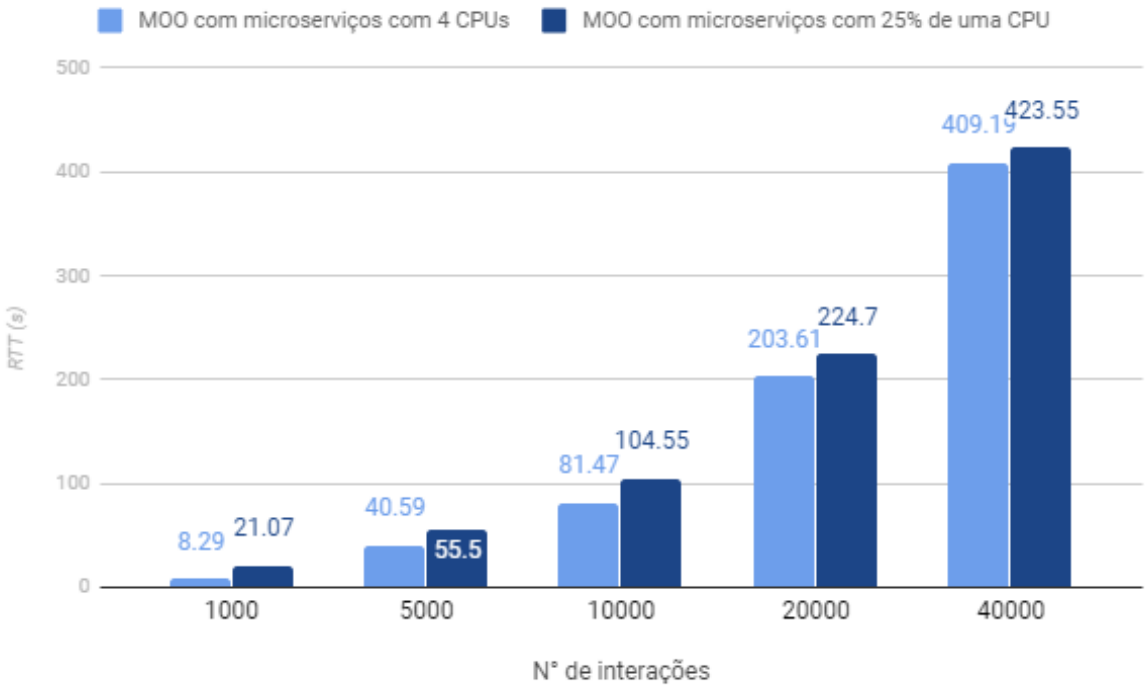

Figura 4.11 Tempo gasto em função da quantidade de invocações de acordo com middleware utilizando 4 CPUs e 25% de uma CPU.

meiro servidor de *bind*. Esta sobrecarga foi simulada determinando um limite de aplicações que um servidor de *bind* conseguiria armazenar ao intermediar seu registro no servidor de nomes.

Ao atingir este limite, que para título de experimento foram 10 aplicações, quando o primeiro servidor de *bind* recebesse mais uma solicitação para registrar uma nova aplicação, essa solicitação era encaminhada para o segundo servidor de *bind*. Os resultados da execução estão descritos na Figura 4.12. O tempo de execução das requisições foram praticamente os mesmos. Isso aconteceu porque para o cliente, quantidade de comunicação para acessar o serviço remoto não mudou.

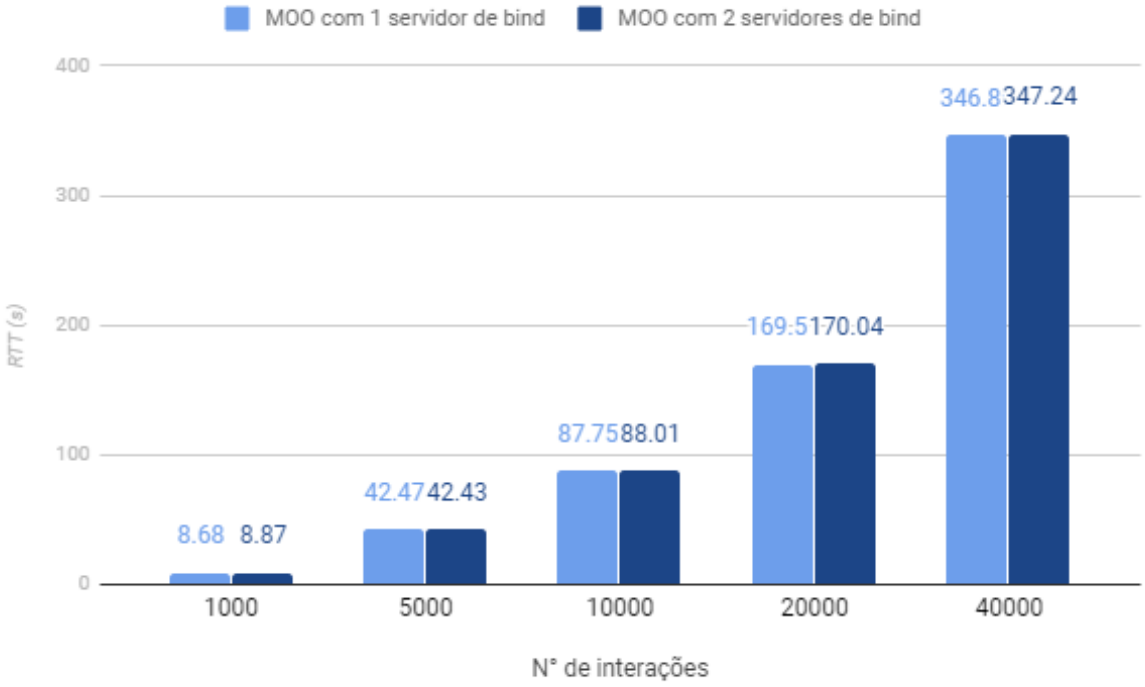

Figura 4.12 Tempo gasto em função da quantidade de invocações realizadas pelos middlewares com um servidor de *bind* e dois servidores de *bind*

4.3 Considerações finais

Este capítulo apresentou inicialmente o objetivo da avaliação de desempenho do middleware implementado. Em seguida, apresentamos o experimento e descrevemos suas métricas, parâmetros e os fatores. Por fim foi apresentado a invocação do método remoto limitando ou não a quantidade de memória, e limitando ou não a quantidade de CPUs disponíveis na execução do contêiner.

CAPÍTULO 5

Conclusão e Trabalhos Futuros

Neste capítulo apresentamos inicialmente as conclusões do trabalho, seguida por possíveis trabalhos futuros.

5.1 Conclusão

Os resultados obtidos nos experimentos mostraram que o tempo gasto pelo middleware com microserviço foi maior que o tempo gasto pelo middleware sem microserviço. Entretanto, antes de descartar esta implementação é importante levar em consideração o tempo em que o middleware com microserviço pode ser desenvolvido. Devemos ressaltar que cada microserviço pode ser desenvolvido por equipes de desenvolvedores à parte o que torna o processo de desenvolvimento mais ágil. O aumento do tempo de execução é devido ao fato de que no middleware com microserviços temos mais comunicação ao executarmos uma requisição. O que antes para o cliente era apenas acessar o servidor de nomes, e já obter o *proxy* para executar a aplicação no método remoto, agora é necessário passar pelo servidor de *proxy* antes de chegar ao servidor de nomes, além de ter que passar pelo servidor de *bind* antes de chegar ao serviço remoto no servidor da aplicação.

O fator memória é um ponto crucial na execução de qualquer aplicação, e neste middleware não seria diferente. Foi possível notar a perda de desempenho à medida que se tem menos memória disponível. O fato de ter um middleware desenvolvido em microserviços, com cada microserviço sendo executado em um *host* separadamente, nos dá a possibilidade de distribuir a carga de execução deste middleware, evitando assim que o middleware sofra com a baixa disponibilidade de memória.

Notamos que, ao executar a aplicação no Docker, em uma situação que a memória disponível pelo sistema está escassa, o contêiner utiliza uma memória *swap* que é reservada no próprio disco rígido do computador. No nosso caso, o fato do disco rígido do computador ser um *solid state drive* (SSD) ajudou a minimizar a diferença de tempo de execução nos casos de diminuição da memória disponível, o que seria provavelmente maior nos casos de um computador com um disco rígido *hard disk drive* (HD).

Os resultado obtidos limitando a quantidade de CPUs disponíveis na execução dos contêineres também eram de se esperar. Quanto menor a quantidade de CPUs, mais tempo o middleware levava para executar sua tarefa.

A arquitetura desenvolvida permite a adição de mais um servidor de *bind* independente do já adicionado. Isso dá a possibilidade do mesmo servidor de nomes dá suporte à mais de um servidor de *bind*, tornando o middleware mais escalável.

5.2 Trabalhos Futuros

Como sugestão de trabalhos futuros, seria interessante implementar o mesmo middleware com outra estrutura de seus componentes em microserviços. Podemos dispôr o middleware em seis microserviços de forma diferentes: cliente, servidor, *proxy*, *requestor* e *marshaller* (cliente), *invoker* e *marshaller* do lado do servidor. Os outros 2 microserviços seriam o *ClientRequestHandler* e o *ServerRequestHandler*.

Em termos de implementação, trechos dos códigos implementados poderiam ser otimizados. O identificador único de cada aplicação ao solicitar o registro ao servidor de *bind*, pode ser substituído pelo próprio nome da aplicação, assim como é utilizado quando o servidor de *bind* registra a aplicação no servidor de nomes.

Outra possibilidade interessante é a utilização da ferramenta Docker *compose*, para executar o middleware utilizando multi-contêineres.

Referências Bibliográficas

[1] DMITRY NAMIOT, M. S.-S. On microservice-architectural.

[2] DOCS, D. Dockerfile reference.

[3] DOCS, D. Limit a container's resources.

[4] DRAGONI, N., GIALLORENZO, S., LAFUENTE, A. L., MAZZARA, M., MONTESI, F., MUSTAFIN, R., AND SAFINA, L. *Microservices: Yesterday, Today, and Tomorrow*. Springer International Publishing, Cham, 2017.

[5] EKICI, E. Microservices architecture, containers and docker.

[6] GEORGE COULOURIS, JEAN DOLLIMORE, T. K., AND BLAIR, G. *DISTRIBUTED SYSTEMS Concepts and Design*, vol. 2012. Addison-Wesley, 1988.

[7] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment.

[8] RUNNABLE. Dockerize your java application.

[9] SCHANTZ, R. E., AND SCHMIDT, D. C. Middleware for distributed systems: Evolving the common structure for network-centric applications. *Encyclopedia of Software Engineering 1* (2001), 1–9.

[10] SCHMIDT, D. C., AND BUSCHMANN, F. Patterns, frameworks, and middleware: Their synergistic relationships. *ICSE '03 Proceedings of the 25th International Conference on Software Engineering* (2003).

[11] THÖNES, J. Microservices.

[12] VÖLTER, M., KIRCHER, M., AND ZDUN, U. Remoting patterns-a systematic approach for design reuse of distributed object middleware solutions. *IEEE Computing magazine* (2004).