Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Uso de Elixir para a construção de um middleware baseado em RPC

Mateus Moury Fernandes da Rosa Borges

Trabalho de Graduação

Recife
Julho de 2016

Universidade Federal de Pernambuco
Centro de Informática

Mateus Moury Fernandes da Rosa Borges

Uso de Elixir para a construção de um middleware baseado em RPC

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

Orientador: *Nelson Souto Rosa*

Recife
Julho / 2016

Agradecimentos

Primeiramente, agradeço a todos que compõem o quadro de funcionários do Centro de Informática da Universidade Federal de Pernambuco. Graças aos trabalhadores da portaria e de limpeza do centro, tive a honra de conviver, por quase cinco anos, em um ambiente seguro e propício à troca de conhecimento. Esses funcionários colaboram, diariamente, para que os alunos tenham a oportunidade de estudar em um centro de excelência.

A todos os docentes com quem tive o prazer de aprender. Em especial, agradeço ao professor Dr. Nelson Souto Rosa, que se disponibilizou a me orientar na construção deste trabalho, e ao professor Dr. Paulo G. S. da Fonseca, que além de ter ajudado no desenvolvimento das minhas habilidades cognitivas e de raciocínio lógico durantes suas aulas, lecionadas com maestria, se mostrou um grande tutor, compartilhando sua vasta experiência e me aconselhando em algumas das minhas decisões.

À Maratona de Programação, o programa extracurricular oferecido pelo Centro de Informática que indubitavelmente mais me fez crescer. Através desse programa, pude construir grandes amizades e absorver um enorme conhecimento, além de adquirir habilidades de criatividade e trabalho em equipe. Agradeço a professora Dr. Liliane Salgado, que esteve a frente do projeto enquanto participei, e a todos os que foram treinadores no projeto.

À minha turma, composta de pessoas extremamente inteligentes e prestativas, sem as quais eu certamente não teria sido capaz de ter o mesmo aproveitamento durante o curso. Foram várias as emoções vividas nessa época que se encerra, e sempre estivemos juntos para encará-las e compartilhá-las. Em especial, agradeço a Duhan Caraciolo, Guilherme Peixoto, Lucas Lima e Vinícius Cousseau, pessoas com quem tive o prazer de trabalhar em equipe com maior frequência.

A todos os meus grandes amigos, que conheci no prédio onde resido, no Colégio Equipe ou nas aventuras da vida, bem como a todas as pessoas que conviveram comigo ou cruzaram com a minha jornada. Acredito que todos que passam por mim sempre tem algo a ensinar e algo a aprender. Alguns mais e outros menos. Aprendi, até hoje aprendo e continuarei aprendendo com eles, e espero ter dado a minha parcela de retribuição.

À minha mãe, Roseane, e ao meu pai, Otávio, por sempre terem acreditado em mim e batalhado dia após dia para que eu me tornasse uma boa pessoa. Dedico a vocês tudo de grandioso que eu conseguir no meu caminho. Agradeço também aos meus irmãos: Bruno, Fabiana, Leonardo e Marcela, por serem exemplos de pessoa e me inspirarem a lutar pelos meus sonhos. Amo todos vocês.

Vivemos em um país no qual, infelizmente, pessoas têm oportunidades diferentes apenas por nascerem em condições diferentes. Agradeço a Deus pelas oportunidades que me foram cedidas, e espero que, no futuro, possa ajudar aqueles que não tiveram as mesmas chances que eu.

Resumo

De acordo com o que se tem visto nos últimos anos, a capacidade de processamento de um único computador se mostra cada vez mais insuficiente para a construção de aplicações escaláveis e que demandam alto desempenho. Por isso, há uma necessidade cada vez maior de se construírem sistemas distribuídos. O ideal, na construção de um sistema distribuído, é que o programador se preocupe apenas com a sua aplicação, e não com complicações inerentes ao fato de fazer com que sua aplicação seja distribuída, como problemas na rede e de localização. Esse trabalho propõe a construção de um *middleware* baseado em chamadas remotas de procedimentos que visa facilitar o desenvolvimento de aplicações distribuídas. Elixir, uma nova linguagem funcional que é executada na máquina virtual de Erlang, será usada para o desenvolvimento do *middleware*, já que vem se mostrando uma linguagem que possui qualidades desejáveis ao desenvolvimento de um *middleware*, como facilidade para comunicação entre processos e recuperação de falhas, além de ser usada comercialmente em aplicações de grande porte como o *Pinterest.* Erlang, linguagem que deu origem a Elixir, também é usada em sistemas distribuídos, como o *Messenger* e o *WhatsApp*, por exemplo. O trabalho trará, ainda, resultados quanto ao desempenho do *middleware* desenvolvido, levando em conta o uso de técnicas de paralelismo e concorrência, bem como as vantagens e desvantagens do uso de Elixir para a implementação de um middleware.

Palavras-chave: Elixir, *Middleware*, Sistemas Distribuídos*,* Tolerância a Falhas, Programação Funcional.

Abstract

As time goes by, the processing capacity of a single computer is being proved increasingly insufficient to build efficient and scalable applications. Consequently, there is a rising demand for deploying distributed systems. Ideally, when building a distributed system, the developer's concerns should only regard the application's logic, and not complications inherent to distributing the application itself, such as connection problems. Therefore, this work's goal is to build a middleware based in remote procedure calls that aims to facilitate the development of distributed applications for programmers. Elixir, a fairly new functional programming language that runs on the Erlang Virtual Machine, is going to be used in the project, since it provides desirable features to construct a middleware, namely failure recover and inter-process communication, among others. Besides, Elixir is already being used in huge real world applications, like Pinterest. Erlang, the father of Elixir, is also used at the development of distributed systems, such as WhatsApp and Messenger. This work will also show the middleware's accomplishments in terms of efficiency, taking into consideration the use of parallelism and concurrency, in conjunction with analyzing the benefits and weak points of using the chosen programming language.

Key-words: Elixir, Middleware, Distributed Systems, Failure Recovery, Functional Programming.

Sumário

CAPÍTULO 1

Introdução

Hoje em dia, as áreas de aplicações para sistemas distribuídos são as mais diversas possíveis. Vários dos maiores e mais complexos sistemas que utilizamos são sistemas distribuídos. O exemplo mais claro de um sistema distribuído que usamos é a Internet. A Internet pode ser vista como um grande sistema distribuído com um imenso número de servidores que se comunicam com um número ainda maior de clientes através de vários protocolos. Várias partes desse grande sistema distribuído estão em diferentes lugares do mundo.

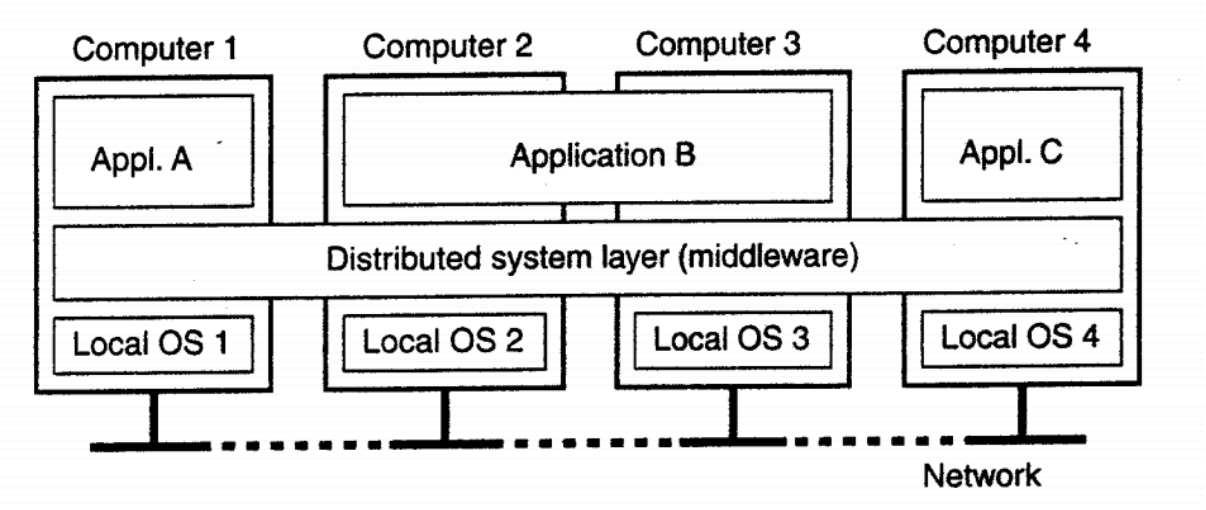
Segundo Völter, Kircher e Zdun (2004), existem várias razões para que sistemas distribuídos sejam usados. Algumas dessas razões são motivadas pela própria natureza do problema que se quer resolver. Por exemplo, na Internet, caso um usuário queira requisitar uma página a um servidor distante, ou bater papo com um amigo, não existe outro caminho a não ser o de transportar dados de um ponto a outro. Outras razões são motivadas pelas propriedades do problema. Se um problema pode ser quebrado em vários subproblemas que podem ser resolvidos independentemente, por exemplo, provavelmente dividir esses subproblemas em várias máquinas pode ser uma boa solução. Adicionalmente, o fato de fazer com que uma aplicação seja distribuída pode fazer com que ela se torne mais tolerante a falhas: distribuir a aplicação evita que ela tenha um ponto único de falha, já que os componentes da aplicação estão em nós diferentes do sistema.

Obviamente, construir um sistema distribuído é uma tarefa complexa, e existem vários desafios no projeto desse tipo de sistema. Um destes desafios é a heterogeneidade, ou seja, num sistema em que várias máquinas estão presentes, provavelmente a comunicação será feita por diferentes sistemas operacionais, com diferentes estruturas de comunicação e hardwares variados. Além disso, componentes da própria aplicação podem ser escritos em linguagens de programação diferentes. Dessa forma, existe um esforço extra para que um sistema heterogêneo funcione. Um outro exemplo de desafio é o de transparência: o sistema tem que ser elaborado de forma que o usuário final não perceba que o sistema é distribuído (MISHRA; TRIPATHI, 2014). Na Internet, por exemplo, o usuário final, que digita uma URL no *browser*, não precisa saber onde está o servidor que traz a informação para seu computador.

Num cenário ideal é desejável que o programador, ao desenvolver uma aplicação, não se preocupe com os desafios inerentes ao fato dessa aplicação ser distribuída. E para isso serve o *middleware* de distribuição: uma camada extra de infraestrutura, que fica entre a aplicação e o sistema operacional, construída para poupar o desenvolvedor de resolver problemas que não são diretamente do escopo da sua aplicação (BERNSTEIN, 1996). Claramente, nenhum *middleware* vai ser capaz de prover uma infraestrutura totalmente transparente para o desenvolvedor. Como um

sistema distribuído utiliza comunicação em rede, latência e falhas de conexão são problemas que o usuário final frequentemente percebe e tem que contornar.

Figura 1.1: Middleware como uma camada extra entre aplicações e sistemas operacionais



Fonte: TANENBAUM; VAN STEEN, 2006, p.3

Existem várias abordagens para a implementação de um *middleware*. Historicamente, a primeira abordagem foi a de transferência de arquivos, através do *File Transfer Protocol*, explicado em Parker (2005). Quando programas em máquinas diferentes queriam se comunicar, eles faziam o *download* de arquivos que continham informações de interesse, possivelmente modificavam ou criavam novos arquivos, e faziam o *upload* destes. Muito embora essa abordagem tenha resolvido vários problemas, várias outras surgiram para resolver novos problemas.

A abordagem usada para a construção do *middleware* deste trabalho é a de *Remote Procedure Call*s (BIRREL; NELSON, 1984). O objetivo da abordagem é fazer com que invocar um procedimento remoto (isto é, uma função a ser executada em um outro ambiente) pareça o mesmo que invocar uma função local, do ponto de vista do desenvolvedor da aplicação. Desta maneira, o programador escreveria o código sem nenhuma mudança, apesar do procedimento não estar localizado na sua máquina. O RPC funciona como o modelo cliente/servidor: o programa que invoca o procedimento é o cliente, e o que o executa, o servidor. Toda essa operação é síncrona, ou seja, o andamento do cliente é bloqueado até que o procedimento remoto dê a resposta à invocação (ROUSE, 2009).

Todo o software desenvolvido durante este trabalho utiliza Elixir (VALIM, 2012) como linguagem de programação. Elixir tem como objetivo prover facilidades para a construção de sistemas distribuídos e tolerantes a falha. Elixir estende Erlang (ARMSTRONG, 1997) - utilizando a *Erlang Virtual Machine* - para adicionar uma sintaxe mais agradável e maior expressividade.

Erlang, que deu origem a Elixir, foi uma linguagem criada para construir sistemas altamente escaláveis e que exigem disponibilidade. Exemplos de sistemas desse tipo são os sistemas de telecomunicação, bancos e troca de mensagens. Essa

linguagem dá um suporte muito grande para concorrência e distribuição. Por conta disto, ela é usada por sistemas como o WhatsApp[1] e o Messenger[2].

Além de prover facilidade de comunicação entre processos utilizando mecanismos de transferência de mensagens, Elixir é uma linguagem funcional e engloba vários benefícios desse paradigma, como a imutabilidade de dados, que associada à capacidade da linguagem de criar *threads* leves faz com que sistemas altamente concorrentes e escaláveis possam ser desenvolvidos. A linguagem, apesar de criada recentemente, já é usada em sistemas distribuídos e com vários usuários, como o Pinterest[3].

Dessa forma, pode-se observar que Elixir é uma linguagem bastante compatível para preencher os requisitos da construção de um *middleware*. Com cada vez mais aplicações distribuídas, é indiscutível a importância de se tentar construir sistemas de *middleware* mais robustos e eficientes. Por isso, este trabalho analisa, também, as vantagens e desvantagens de se construir um *middleware* usando a linguagem.

1.1 Objetivos

O objetivo deste trabalho é, primeiramente, usar Elixir como linguagem de programação para construir um *middleware* que comportará invocações a procedimentos remotos. Além disso, o trabalho objetiva fazer uma análise sobre quais as vantagens e desvantagens de se ter utilizado essa linguagem, possivelmente analisando quais seriam as diferenças, perdas e ganhos se fosse usada uma linguagem com o paradigma de orientação a objetos. A eficiência do sistema será mensurada levando em conta técnicas de concorrência e paralelismo.

1.2 Estrutura do Trabalho

Este trabalho estará dividido em 5 capítulos, incluindo este capítulo introdutório. No Capítulo 2, será discutido o que é um sistema distribuído e que tipo de problemas ele resolve, passando por alguns exemplos populares de sistemas distribuídos. Além disso, introduzirá conceitos básicos sobre o que é um *middleware* e falará de sistemas de *middleware* bastante usados atualmente. O Java RMI (SUN, 2000) é um deles. Por último, introduzirá conceitos sobre a linguagem de programação escolhida, Elixir, apresentando-a com mais detalhes. Em suma, o Capítulo 2 traz os conceitos básicos necessários ao entendimento do projeto. Em seguida, o Capítulo 3 discorrerá sobre a arquitetura e a implementação do *middleware* desenvolvido neste projeto, explicando como cada componente do *middleware* funciona e mostrando trechos de código em Elixir. No Capítulo 4, serão dispostas diferentes análises sobre a implementação, como

[1] https://www.whatsapp.com
[2] https://www.messenger.com/
[3] https://www.pinterest.com

a eficiência do sistema e os benefícios e malefícios de se ter usado Elixir. Na conclusão, no Capítulo 5, será feito um resumo sobre o estudo realizado, bem como uma explicitação de quais foram os desafios encontrados, as limitações do sistema e a realização de trabalhos futuros.

CAPÍTULO 2

Conceitos Básicos

Antes de implementar um *middleware*, é necessário entender o seu conceito, bem como o conceito do que é um sistema distribuído, para que finalidades ele é usado, quais problemas ele resolve e quais desafios surgem com seu uso. Esse capítulo detalha esses fatores, bem como a evolução dos sistemas distribuídos e sistemas de *middleware*. Além disso, o capítulo discorre sobre Elixir, a linguagem de programação escolhida no projeto, elencando alguns pontos considerados importantes nessa linguagem para o desenvolvimento de um *middleware*.

2.1 O que é um sistema distribuído?

Um sistema distribuído é aquele em que computadores conectados a uma rede se comunicam e coordenam suas ações através apenas da transferência de mensagens. Essa definição leva a algumas características importantes dos sistemas distribuídos: concorrência de componentes, ausência de um *clock* global e independência de falha (COLOURIS et al., 2011, p.1).

Concorrência é a capacidade de um sistema de compartilhar recursos e realizar tarefas diferentes simultaneamente. Numa aplicação web de troca de mensagens, por exemplo, um sistema distribuído mantém, simultaneamente, várias conexões entre os seus servidores e os diferentes clientes. Coordenar aplicações concorrentes e que partilham recursos é algo desafiador e importante.

A ausência de um *clock* global é uma consequência direta do sistema ser distribuído. Ao contrário de uma aplicação centralizada, em que o *clock* dá uma ideia do momento em que cada ação ocorre, um sistema distribuído tem tantos *clocks* quantos forem os componentes desse sistema. Como há limitações para sincronização de *clocks* em diferentes máquinas conectadas, não existe uma noção global de tempo. Por isso que toda a comunicação é feita através da passagem de mensagens.

Sistemas distribuídos podem falhar de novas maneiras que sistemas centralizados não falham, como por exemplo falhas por problemas na rede. Quando acontece uma falha em um componente de um sistema distribuído, nada impede que os outros componentes continuem sendo executados. Por isso que os projetistas do sistema também têm que atentar à independência de falha: não deixar que o sistema inteiro seja sacrificado por conta de uma falha em um de seus componentes.

O propósito da cooperação num sistema distribuído pode variar. Alguns sistemas são feitos com o objetivo de explorar a capacidade de processamento conjunta de vários processadores para atacar problemas grandes e complexos. Mas, muitas vezes, os sistemas distribuídos têm propósitos “individuais”: servir usuários que estão mais interessados em suas atividades particulares, e que as vezes podem

requerer a comunicação com outros usuários ou com os servidores (PELEG, 2000, p. 1). A Internet, ou aplicativos de troca de mensagens, são exemplos desse caso.

É importante ressaltar a diferença entre computação paralela e distribuída. Enquanto que na computação paralela todos os processadores têm acesso a recursos compartilhados de memória, e os utilizam inclusive para trocar informações, na computação distribuída cada processo tem seu próprio espaço de memória. Por isso a informação é trocada através de mensagens.

São vários os exemplos de sistemas distribuídos, que estão em várias aplicações diferentes. Redes de telecomunicação, bancos de dados distribuídos e jogos *multiplayer* online são só algumas das aplicações em que se podem utilizar os sistemas distribuídos. Por exemplo, em jogos de tiro online, cada jogador joga em seu computador, e jogadores de times diferentes travam um combate. Informações são trocadas ininterruptamente entre as máquinas que compõem o sistema distribuído (que no caso, são as próprias máquinas usadas pelos jogadores) para que o estado do jogo seja o mesmo para todos os seus jogadores.

2.1.1 Histórico

A ideia de dividir um problema em várias tarefas e resolvê-las de forma independente surgiu, primeiramente, com a programação concorrente. Essa necessidade veio pelo desejo de desenvolver sistemas operacionais mais robustos, confiáveis e eficientes, nos anos 60 (HANSEN, 2001). Sendo possível executar um processo apenas ao término de outro, caso um processo tivesse que parar por uma interrupção (por exemplo, acesso ao disco), todo o hardware ficaria parado enquanto ela não acabasse. Com a execução simultânea de processos, esse problema não teria o mesmo impacto.

O estudo de processos concorrentes e a comunicação entre eles pela troca de mensagens deu espaço para o surgimento dos primeiros sistemas distribuídos. Pode-se dizer que o primeiro sistema distribuído em larga escala foi o sistema de e-mail da ARPANET, a antecessora da Internet. No primeiro e-mail enviado, as máquinas estavam posicionadas uma ao lado da outra e ambas conectadas à ARPANET. Ray Tomlison, que enviou o e-mail, fala mais sobre isso em Tomlinson (2016).

Com o interesse crescente em sistemas distribuídos em 1982, começaram a surgir as primeiras arquiteturas de sistemas paralelos e centros de pesquisa começaram a dar maior atenção a sistemas distribuídos. (JIA, 2013).

Finalmente, nos anos 90, grandes empresas começaram a utilizar sistemas distribuídos para melhorar a eficiência dos seus serviços, que começaram a precisar de uma quantidade de processamento muito maior do que a oferecida por um único computador. Um exemplo é o Google, que utilizou sistemas distribuídos para otimizar seu algoritmo de busca.

Hoje em dia, empresas que oferecem serviços a uma grande quantidade de usuários não o fazem sem utilizar sistemas distribuídos. A era de hoje é marcada por

empresas que possuem grandes centros de dados e de processamento, que proveem, além de maior eficiência, tolerância à falha e outras características importantes.

2.2 O que é um *middleware* de distribuição?

Uma característica crucial dos sistemas distribuídos é que, apesar de serem uma coleção independente de computadores, se apresentam aos usuários como um sistema único e coerente (TANENBAUM; VAN STEEN, 2006, p. 2). Ou seja, num bom sistema distribuído, pessoas (ou programas) pensam que estão lidando com um sistema centralizado. Os desenvolvedores, por sua vez, também querem ser capazes de desenvolver sua aplicação distribuída como se ela fosse centralizada, isso é, sem se preocupar com as dificuldades impostas pelo fato da aplicação ser distribuída. E é para isso que serve o *middleware* de distribuição.

O *middleware* é uma camada de software que fica entre o sistema operacional e a aplicação e tem o objetivo de esconder aspectos da distribuição, de forma que a aplicação não tenha que se preocupar com isso (BERNSTEIN, 1996). Isso é, um *middleware* faz com que algumas dificuldades geradas pela ação de se distribuir o sistema sejam invisíveis à aplicação - ele provê vários tipos de transparências.

Um tipo de transparência é o de localização: a aplicação conhece o nome do serviço que quer usar, mas não precisa saber onde, fisicamente, aquele serviço está localizado. Além disso, um *middleware* faz com que acessar um serviço remoto seja semelhante a acessar um serviço local, através da transparência de acesso. Existem ainda outros tipos de transparência, como a transparência de falha, em que a falha de um serviço remoto não é percebida pela aplicação; a transparência de tecnologia, que faz com que uma aplicação não tenha seu código alterado pelo fato de fazer uma chamada a procedimentos que estão sendo executados em diferentes sistemas operacionais ou diferentes linguagens de programação, etc.

Existem vários tipos de *middleware* que servem de infraestrutura para aplicações distribuídas. Cada tipo difere em várias características, e, por isso, decidir que tipo de *middleware* construir para uma certa aplicação, ou conjunto de aplicações, é algo que deve ser profundamente estudado pelo desenvolvedor que toma as decisões acerca da arquitetura do projeto.

O tipo do *middleware* implementado neste projeto é de um *middleware* baseado em *remote procedure calls* (BIRREL; NELSON, 1984). Esse *middleware* se baseia na observação de que uma invocação a um procedimento, numa única máquina, é apenas um mecanismo interno de transferência de dados e de controle. A ideia é que, com esse sistema, essa transferência de dados e controle possa ser estendida para ser transmitida em uma rede de computadores. Assim, quando uma chamada é feita, o ambiente que fez a chamada é suspenso, os parâmetros são passados pela rede para o ambiente que executará o procedimento, que retorna, então, a resposta desejada, fazendo com que o ambiente que invocou o procedimento volte à sua execução normal. A ideia é que um *middleware* que provê esse serviço faz com que, por parte do desenvolvedor, invocar um procedimento local ou remoto seja feito da mesma forma.

Utilizando o tipo de *middleware* mencionado como base, surgiram outros tipos mais complexos, como os sistemas de *middleware* orientados a objetos. A ideia é que chamadas a métodos de objetos, bem como alteração em seus estados, poderiam ser feitas de maneira remota de forma semelhante a uma chamada ou alteração local. Vários desses sistemas contam ainda com características mais complexas, como a presença de um *garbage collector* remoto, por exemplo. Existem vários padrões estabelecidos para a construção de sistemas de *middleware* orientados a objeto, como o CORBA (OMG, 1991), bem como implementações desse tipo de *middleware*, como o Java RMI (SUN, 2000).

Existem ainda outros tipos de *middleware*, como o *middleware* orientado a mensagens (EUGSTER et al., 2003). Nesse tipo, clientes enviam e *subscribers* recebem mensagens, sendo esta a única forma de comunicação entre eles. Cada cliente envia suas mensagens a um ou mais servidores, que agem como intermediários na transmissão de mensagens, chamados de servidores de filas. Isso faz com que, diferentemente de no modelo RPC, a comunicação seja assíncrona, ou seja, os componentes do software não têm que esperar respostas para seguir a execução, já que após entregar a mensagem ao servidor de filas, o cliente continua sua execução. Geralmente, clientes enviam mensagens sobre certos tópicos, ou que contém certos padrões, e *subscribers* que se inscrevem nesses tópicos recebem as mensagens através do servidor de filas.

Um outro tipo de *middleware* que vale a pena mencionar é o *middleware* baseado em espaço de tuplas (LI; HUDAK, 1989). Tal modelo consiste em um espaço compartilhado por todos os clientes do *middleware* onde estão dispostas tuplas de dados. Toda a comunicação entre os clientes é feita por esse espaço compartilhado, onde eles inserem e removem tuplas. Diferentes implementações desse paradigma estão disponíveis, como o JavaSpaces (SUN, 2002), uma implementação para Java.

Como já dito anteriormente, os vários tipos diferentes de *middleware* possuem características diferentes. Podem ser síncronos ou assíncronos, a sua comunicação pode ser de 1 para 1 ou 1 para n, etc. Escolher o tipo de *middleware* correto para uma certa aplicação ou conjunto de aplicações exige um estudo sobre a natureza da aplicação que o utilizará.

2.3 Elixir

A linguagem de programação utilizada nesse projeto foi escolhida com base em vários fatores. Elixir é uma linguagem que permite ao desenvolvedor a implementação de sistemas bastante escaláveis, já que é uma linguagem orientada a processos, que se comunicam via mensagens. Além dos processos serem leves a ponto de uma única máquina suportar centenas de milhares de processos sendo executados concorrentemente, processos entre máquinas diferentes também podem se comunicar de forma fácil.

Em se tratando de software distribuído, é fato que a ocorrência de erros acontece com mais frequência, principalmente por problemas na rede. Elixir traz um

sistema de supervisores - processos que supervisionam a execução de outros processos - permitindo que, em caso de falha, os processos que estão sendo supervisionados sejam reiniciados com um estado escolhido pelo desenvolvedor. Isso traz a tolerância à falha a preço baixo para o *middleware*.

Aliado a esses fatores, vem o *Mix*, que é uma ferramenta de *build* para Elixir que torna simples o processo de criação de projetos grandes, com vários arquivos que se relacionam. O *middleware* pode ser compilado, testado e executado sem maiores problemas através dessa ferramenta.

2.3.1 Erlang

Erlang foi uma linguagem criada originalmente para melhorar o desenvolvimento de sistemas telefônicos. Esse tipo de sistema era diferente dos demais porque era, por natureza, um sistema altamente concorrente (já que *switches* nas redes de telefonia tinham que suportar milhares de conexões simultaneamente) e com alta tolerância a falhas, já que era bastante indesejável que um sistema telefônico saísse do ar (ARMSTRONG, 2007). Por isso, Erlang também disponibilizava um mecanismo para mudar código *on the fly*, isso é, sem que o programa parasse.

Um pouco depois da idealização de Erlang, começaram os estudos para que fosse desenvolvida a *Erlang Virtual Machine*, chamada de BEAM. Essa máquina virtual é executada como um processo do sistema operacional e fica entre o sistema operacional e a aplicação em Erlang. Os processos de Erlang são implementados inteiramente pela BEAM e não tem nenhuma conexão com os processos ou threads do sistema operacional. Então, por mais que vários processos de uma aplicação de Erlang sejam criados, para o sistema operacional, só há, ainda, um processo. Essa criação e gerenciamento de processos pela BEAM foram feitos de forma que os processos são leves e sua comunicação é simples.

Aplicações em Elixir também são executadas pela BEAM. Por isso, Elixir herda a facilidade para criação e comunicação de processos, que são extremamente *lightweight*. Numa única máquina podem ser criados milhões de processos Elixir. Por ser uma linguagem fortemente baseada em Erlang, Elixir também é voltada para aplicações concorrentes e que exigem tolerância a falhas, características desejáveis em um middleware.

O criador de Elixir, José Valim, criou essa linguagem para utilizá-la em contextos parecidos com Erlang, porém trouxe várias melhorias em relação a essa linguagem em termos de expressividade e organização de código. Os planos do autor foram para que essa linguagem fosse mais legível, com uma sintaxe mais amigável e expressiva, sem perder as características de Erlang. Por isso, Elixir herda todas as bibliotecas disponíveis em Erlang.

2.3.2 Paradigma de programação funcional

Elixir traz várias vantagens provenientes do fato de ser uma linguagem funcional. Primeiramente, sobre coesão de código, já foram realizados estudos que comprovam que linguagens funcionais fazem com que os códigos sejam expressivamente mais concisos do que em linguagens com orientação a objetos, que geralmente são mais verborrágicas e menos expressivas (NANZ; FURIA, 2015).

Uma segunda vantagem das linguagens funcionais, que inclusive é bastante desejável num contexto de construção de um middleware, são as funções de alta-ordem. Essas funções podem receber outras funções como argumentos, e também retornar funções. Isso acrescenta muito em poder à linguagem.

Uma terceira característica é a imutabilidade dos dados. Os dados são imutáveis uma vez que são criados, e isso torna extremamente mais fácil a implementação do paralelismo e da concorrência, uma vez que variáveis em espaços de memória que são compartilhados não podem ser modificadas. Isso faz com que o código seja bem menos suscetível a erros.

Por fim, Elixir traz ainda várias *features* que geralmente estão presentes em linguagens funcionais, como *pattern matching*, *guard clauses* e outras, e combina bibliotecas de Erlang para criar novas funções ainda mais poderosas.

2.4 Considerações Finais

Neste capítulo apresentamos as definições básicas sobre sistemas distribuídos, falando um pouco sobre o histórico, e também sobre *middleware*, camada de infraestrutura extra que possibilita a construção de sistemas distribuídos complexos. Falamos sobre alguns sistemas de *middleware* disponíveis no mercado e quais as diferenças entre os tipos de *middleware* mais famosos. Além disso, explicamos as características principais de Elixir, além de explicitar a motivação sob a qual a linguagem foi criada.

CAPÍTULO 3

Arquitetura e implementação do *middleware*

Este capítulo discorrerá sobre o *middleware* elaborado pela proposta deste trabalho. Descreveremos os requisitos elencados no projeto, a arquitetura do projeto e a sua implementação. Mostraremos também alguns exemplos de como utilizá-lo.

3.1 Requisitos

Aqui, apresentaremos os requisitos funcionais e não funcionais do *middleware* construído, que, como dito anteriormente, se baseará no estilo RPC para suportar aplicações funcionais distribuídas, fornecendo transparências de acesso, localização, concorrência e falha. Pretendemos também que características funcionais sejam preservadas, de forma que invocações remotas possam ser feitas com um maior grau de abstração (trazendo funções como argumentos, por exemplo).

3.1.1 Requisitos Funcionais

Os requisitos funcionais do *middleware* serão os seguintes:

- **RF01**: Transparência de Acesso. Realizar invocações remotas a procedimentos deve ser feito de maneira semelhante a realizar invocações locais. Informações quanto à localização das funções remotas são abstraídas do cliente.
- **RF02**: Transparência de Localização. A aplicação do cliente não deve precisar saber em que servidores estão localizadas as funções remotas para poder usá-las. Para prover essa transparência, será utilizado um servidor de nomes, também construído como parte do projeto. Dessa forma, funções remotas que mudem de localização, por exemplo, não deverão impactar o funcionamento do cliente.
- **RF03**: Permitir funções remotas de alta ordem. Funções remotas poderão receber outras funções em seus argumentos, o que faz com que padrões de códigos já implementados possam ser executados remotamente. Por exemplo, será possível aplicar um *reduce* remoto - função que recebe uma lista e outra função como argumentos e utiliza um acumulador para combinar a aplicação da função passada como argumento em todos os elementos da lista.
- **RF04**: Ambiente de programação interativo: utilizando Elixir, é possível para o programador usar o *middleware* sem nenhum pré-processamento ou compilação prévia, executando comandos e analisando os resultados desses comandos

prontamente. Isso é útil para o programador se familiarizar com o *middleware* antes de escrever sua aplicação.

3.1.2 Requisitos Não Funcionais

No caso de requisitos não funcionais, o *middleware* implementado terá:

- **RNF01**: Utilizar o máximo poder de processamento possível. Elixir é uma linguagem orientada a processos, de forma que é inerente a linguagem a habilidade de paralelizar tarefas em processos distintos.
- **RNF02**: Tolerância a falhas. Elixir conta com uma estrutura que dá suporte ao controle de falhas, possivelmente reiniciando aplicações com um estado definido dinamicamente. Isso fará com que erros de conexão entre clientes e servidores ou erros de execução no servidor, por exemplo, não deixem o sistema, como um todo, parar.
- **RNF03**: Transparência de concorrência. O servidor poderá aceitar invocações concorrentes de vários clientes diferentes e de forma escalável, ou seja, sem prejudicar o desempenho geral do sistema.
- **RNF04**: A definição da arquitetura será feita de acordo com padrões conhecidos de projetos de *middleware,* os Padrões Remotos, definidos em Völter, Kircher e Zdun (2004). Esses padrões servirão de base para a construção do nosso *middleware* em linguagem funcional, e facilitará o desenvolvimento e entendimento do código.
- **RNF05**: Esse *middleware* não provê heterogeneidade. Para esse projeto, tanto o cliente quanto o servidor deverão ser desenvolvidos na mesma linguagem do *middleware* por questões de simplificação.

3.2 Arquitetura

Como definido no **RNF04**, utilizaremos padrões de projeto de *middleware* já conhecidos para pensar a nossa arquitetura. Os padrões servem para modularizar componentes que tratam de aspectos comuns à maioria dos sistemas de *middleware*, como o gerenciamento e a sincronização de conexões, a serialização de mensagens e a necessidade de prover transparências conhecidas, como de acesso e localização.

No caso deste projeto, escolhemos utilizar os Padrões Remotos, definidos em Völter, Kircher e Zdun (2004). Esses padrões foram pensados para sistemas de *middleware* orientados a objeto, ou, mais especificamente, para invocação de métodos em objetos. Por conta disso, nossa arquitetura é, na verdade, uma adaptação bastante simplificada dos Padrões Remotos, visto que trata com funções remotas, e não objetos.

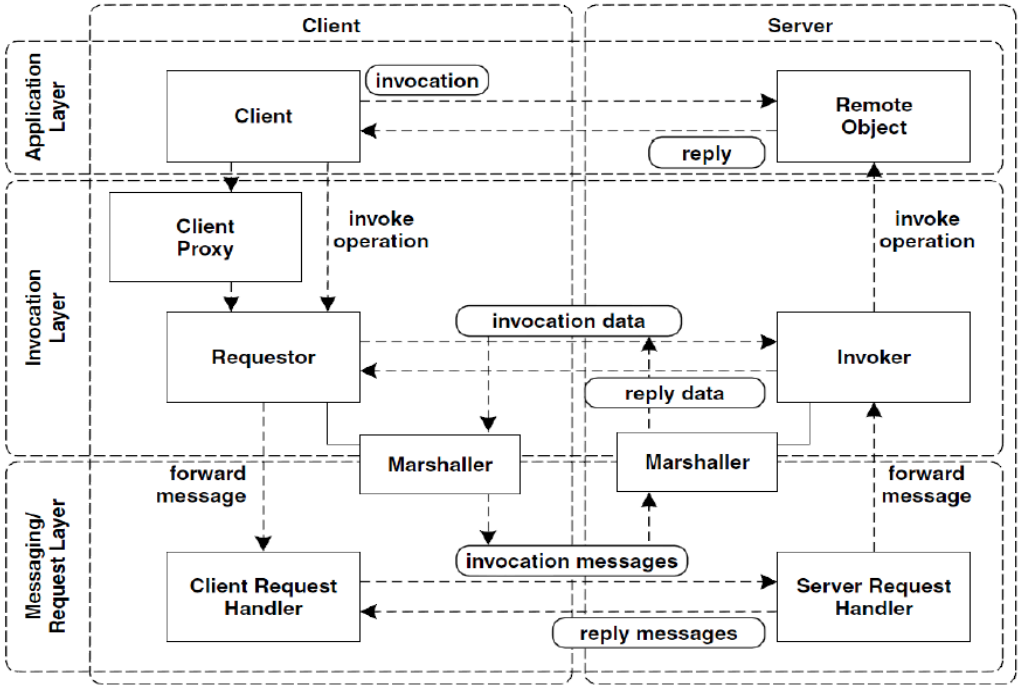
A Figura 3.1 mostra a arquitetura dos Padrões Remotos. Todos os componentes dessa arquitetura serão reutilizados pelo *middleware* implementado neste projeto, menos, por óbvio, o objeto remoto, que dará lugar à função remota. O significado desses componentes será mantido, porém eles serão adaptados para se comportarem no contexto de um *middleware* RPC, e não de um orientado a objeto. Essas

adaptações serão discutidas nesta seção e também na seção de implementação. Ao longo do texto, faremos referência aos componentes dos Padrões Remotos em inglês, para que o entendimento não seja dificultado.

Como pode ser visto na Figura 3.1, os componentes do *middleware* serão divididos nas duas camadas abaixo da *Application Layer*. A primeira, chamada de *Messaging Layer*, trata do gerenciamento das conexões do *middleware*, ou seja, da interação do *middleware* com as ferramentas de rede disponibilizadas pelo sistema operacional. A segunda camada, chamada de *Invocation Layer*, tem o papel de prover a transparência de acesso, transformando invocações escritas como locais em invocações de fato remotas, seguindo os princípios do RPC.

Vale ressaltar que, além do servidor e do cliente, nossa arquitetura conta ainda com um serviço de nomes. O serviço de nomes é um servidor à parte que funciona para nos prover a transparência de localização aos servidores do *middleware*. Esse servidor guarda uma tabela que contém, para cada função remota, a sua localização e a sua interface, que será descrita posteriormente nesta seção.

Figura 3.1: Arquitetura dos Padrões Remotos



Fonte: VÖLTER et al, 2004, p. 66.

É importante frisar que, apesar da aplicação do servidor de nomes ser diferente da dos demais servidores, ele usa os mesmos componentes do nosso *middleware* de um servidor comum.

3.2.1 *Messaging Layer*

Essa camada utiliza a API de rede do sistema operacional para prover a comunicação de rede entre o cliente e o servidor. Dessa forma, todo o envio ou recebimento de mensagens (que são tratadas como sequências de *bytes*) são feitos por essa camada, usando um protocolo de rede pré-estabelecido pelo desenvolvedor do *middleware*, que no caso deste projeto, foi o TCP. Portanto, a camada provê, ao resto do *middleware* e também à aplicação, transparência quanto a comunicação entre diferentes nós do sistema distribuído.

Além do estabelecimento da conexão entre os nós, essa camada também é responsável por algumas configurações, como o estabelecimento de *timeouts* e número de tentativas para se enviar informações, podendo cuidar do tratamento de alguns erros antes de repassá-los às camadas superiores. Essa camada consiste dos componentes:

- ***Client Request Handler**:* faz as funções citadas acima para o cliente. No momento em que invocação remota é feita pelo cliente, esse componente envia a mensagem para o *Server Request Handler*, e espera a resposta. Ao recebê-la, transfere-a para as camadas superiores e fecha a conexão com o servidor.

- ***Server Request Handler**:* faz as funções citadas acima no lado do servidor. Escuta por invocações dos clientes, e ao recebê-las, avisa às camadas superiores quanto à mensagem. Depois que as camadas superiores processam a invocação, dão a resposta, já serializada, a esse componente, que envia de volta, na conexão que ainda está aberta, para o cliente. Logo após enviar a mensagem, o servidor fecha a conexão.

- ***Marshaller***: esse componente, na verdade, atua entre as duas camadas do *middleware*. Apesar de, algumas vezes, desenvolvedores do *middleware* incorporarem esse componente à camada de invocação, o autor deste trabalho preferiu o dispor nesta camada. Isso porque o *Marshaller* não depende de nenhum outro componente, portanto pode ficar na camada mais inferior. Esse componente é responsável por serializar e desserializar as mensagens que vão ser trocadas entre os *request handlers*, ou seja, transformar as mensagens no formato aceito pela rede, que é o de sequência de *bytes*, e também fazer a transformação inversa.

3.2.2 *Invocation Layer*

Essa camada existe para garantir a transparência de acesso à camada de aplicações, onde se encontram o cliente e o servidor. Dessa forma, o desenvolvedor da aplicação não precisará se preocupar, no momento da invocação a um procedimento, se ele é remoto ou local, já que essa camada garante que, do ponto de vista do cliente,

as ações tomadas são as mesmas. Para que isso aconteça, essa camada conta com os seguintes componentes:

- ***Client Proxy**:* esse componente é a porta de entrada do cliente para o *middleware*, ou seja, a interface do *middleware* para o cliente. A ideia é que o *Client Proxy* gere funções cuja assinatura é idêntica à da função remota, para que o cliente possa utilizar essas funções sem se dar conta que não são locais. Essas funções servem também para fazer uma checagem de tipos (e mesmo outras checagens de mais alto nível, que serão explicadas posteriormente) nos argumentos passados para a função. A partir do momento que essas funções são invocadas, o *Client Proxy* incorpora, aos parâmetros passados pelo cliente, outros parâmetros que dizem respeito à localização da função remota, como o endereço IP e a porta pela qual o servidor está escutando as invocações remotas. Todo esse conhecimento é agrupado e repassado ao *Requestor*.

- ***Requestor**:* de posse de todos os dados da invocação remota (localização da função e argumentos), o *Requestor* tem o papel de coordenar o envio dessa invocação até que ela chegue no *Invoker*, componente presente no lado do servidor. Primeiramente, este componente pede ao *Marshaller* para que o nome da função e seus argumentos sejam serializados. Após isso, o *Requestor* pede ao *Client Request Handler* para que ele abra uma conexão com a porta e o endereço IP da função remota, e que, após isso, envie as informações ao servidor. De posse da resposta, o *Requestor* faz um novo pedido ao *Marshaller*: para que desserialize a resposta recebida. Depois disso, a resposta é repassada ao *Client Proxy*, que então a repassa para o cliente, que recebe o retorno da invocação remota como se ela fosse uma invocação local.

- ***Invoker***: Assim que o servidor é iniciado, esse componente pede ao *Server Request Handler* para que inicialize um processo que escute por pedidos de conexões em uma porta predefinida. Para cada requisição que o *Server Request Handler* aceita, haverá um processo gerado, que controlará um *socket*. Esses processos, criados pelo *Server Request Handler*, recebem ordens do processo do *Invoker* através da troca de mensagens. Primeiramente, o *Invoker* ordena que o processo receba a mensagem do cliente, que corresponde à invocação remota. De posse da informação, o *Invoker* cria um novo processo para executar a requisição: este processo usa o *Marshaller* para desserializar a mensagem e invoca a função pedida. Tendo o resultado, o processo novamente utiliza o *Marshaller* para serializá-lo, e então envia a resposta ao *Invoker*, que ordena ao processo do *Server Request Handler* que a envie para o cliente. É válido notar que a criação de processos para diferentes conexões torna o sistema mais eficiente.

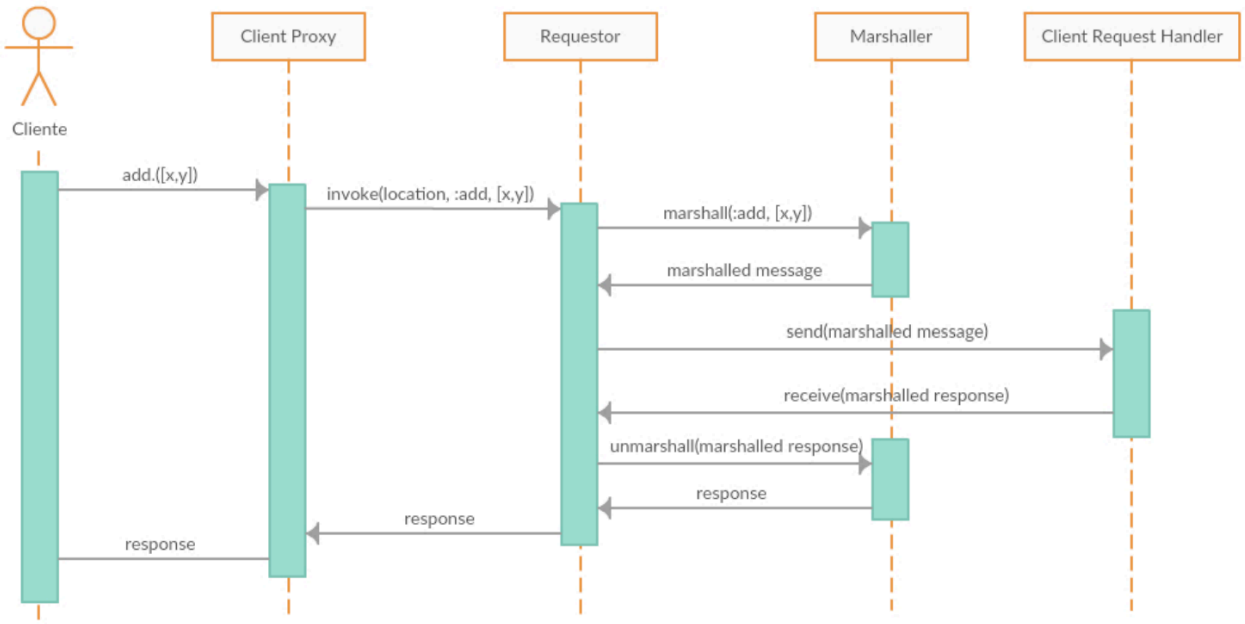
3.2.3 Interação entre os componentes

A Figura 3.2 mostra, através de um diagrama de sequência, a interação entre os componentes do cliente quando ele faz uma invocação para adicionar os números x e y de forma remota. O processo se inicia quando o cliente faz uma invocação a uma função utilizando sua assinatura como se ela fosse uma função local. A partir daí os componentes do *middleware* entram em ação para garantir os tipos de transparências prometidos nos requisitos.

Após receber a invocação do cliente, a função gerada pelo *Client Proxy* junta, aos argumentos x e y, informações relativas à localização do servidor (endereço IP e porta) e da função dentro do servidor (nome do módulo). Todos esses dados relativos à localização na rede e à localização no próprio servidor são obtidos através do serviço de nomes, que será descrito na próxima seção. O *Client Proxy* envia as informações ao *Requestor*.

O *Requestor* utiliza o *Marshaller* para serializar as informações que serão utilizadas no servidor (nome do módulo, nome da função e seus argumentos). Assim, logo após requisitar ao *Client Request Handler* que abra uma conexão com o servidor, o *Requestor* utiliza o mesmo componente para enviar a mensagem. Quando ele recebe a resposta, também através do *Client Request Handler*, desserializa-a com a ajuda do *Marshaller* e a envia para o *Client Proxy,* como retorno da função *invoke*. O *Client Proxy,* então, direciona a resposta para o cliente, como retorno a invocação da função de adição.

Figura 3.2: Interação dos componentes - Cliente



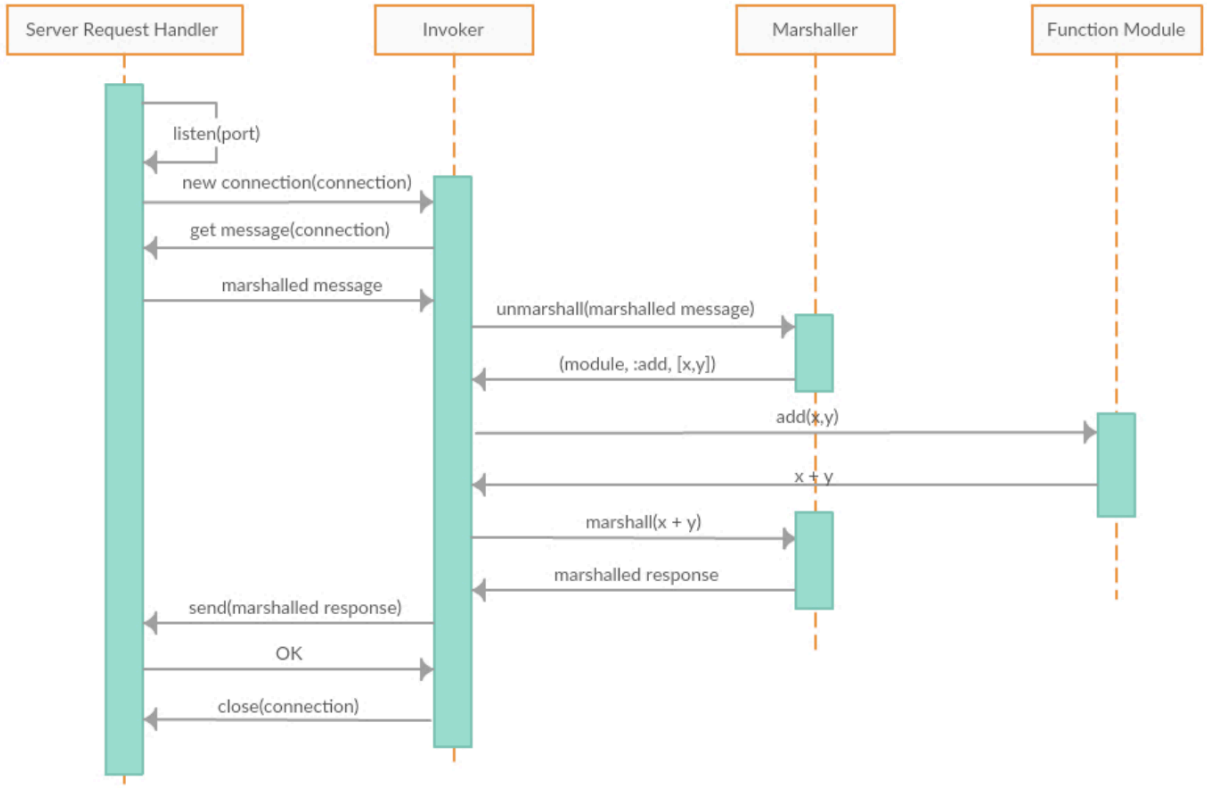
Fonte: Elaborada pelo autor

A Figura 3.3 mostra o diagrama de sequência do servidor a partir do momento em que a invocação do cliente é escutada. O fluxo acontece de forma que, no final, a resposta é enviada ao *Client Request Handler*. Tudo começa no *Server Request Handler*.

O *Invoker* pede ao *Server Request Handler* para abrir um processo que escuta por uma porta predefinida. No momento em que uma conexão é aceita, um novo processo é alocado, e esse processo avisa ao *Invoker* que está responsável por uma nova conexão. O *Invoker* ordena que esse processo receba a mensagem referente ao cliente que solicitou que essa conexão fosse aberta.

O processo alocado pelo *Server Request Handler*, após receber a mensagem a mando do *Invoker*, a envia para esse componente. O *Invoker,* então*,* cria um novo processo, que será responsável por processar a requisição que o cliente fez através da mensagem. Este processo desserializa a mensagem com a ajuda do *Marshaller,* para fazer a localização interna de onde está a função. O módulo da função nada mais é, no contexto de Elixir, do que um *namespace* onde, possivelmente, podem ser encontradas várias funções diferentes. De posse dessas informações, o processo criado pelo *Invoker* pede ao módulo da função para que ela seja executada. O processo, então, serializa o retorno da função com a ajuda do *Marshaller* e o envia, em forma de mensagem, para o *Invoker*.

Figura 3.3: Interação dos componentes- Servidor



Fonte: Elaborada pelo autor

Tendo a resposta serializada, o *Invoker* envia uma mensagem ao processo responsável pela conexão, dizendo que ele deve enviar essa resposta ao cliente.

Quando a resposta é enviada, o processo informa ao *Invoker* que a enviou, e o *Invoker*, finalmente, ordena que esse processo feche a conexão e seja desalocado.

3.2.4 Serviço de nomes

Com a intenção de prover a transparência de localização, o nosso *middleware* também dispõe de um serviço de nomes, que segue o padrão remoto *Lookup*. Originalmente, o serviço de nomes é um servidor a parte que mantém uma tabela, a qual relaciona nomes com localizações de objetos. No nosso caso, ela relaciona nomes com a localização e a assinatura das funções remotas.

Os servidores que implementam as funções remotas podem cadastrar novas funções na tabela do serviço de nomes. Para isso, eles invocam a função *bind* do serviço de nomes, que requer o nome sob o qual a função será cadastrada, informações de localização e informações da função. As informações de localização englobam endereço IP e porta do servidor onde está localizada a função. Quanto a função, será passado o nome do módulo da função, o nome da função e, ainda, para cada parâmetro da função remota será passada uma função para fazer checagem de tipos e conferir se o argumento passado está nos conformes da função remota. Por exemplo, o servidor pode cadastrar a função de *fibonacci* na tabela de nomes mas indicar que ela só pode ser usada por números inteiros entre 0 e 50. A liberdade de argumentos serem checados por funções dá muito mais poder ao desenvolvedor.

Assim, no cliente, funções remotas poderão ser usadas sem que ele saiba as localizações delas. O cliente só precisa saber, à priori, a localização do serviço de nomes. Ele requisita as funções remotas ao serviço de nomes usando a função *lookup*, e passando o nome do serviço como parâmetro. De posse das informações retornadas por ele, começa a interação citada nas seções anteriores através do *Client Proxy*.

3.3 Implementação

Os Padrões Remotos, originalmente, foram pensados para um *middleware* orientado a objetos, e por isso geralmente são desenvolvidos utilizando linguagens que deem suporte a esse paradigma. Por essa razão, é necessário fazer um mapeamento entre os componentes originais e os que utilizaremos aqui, seguindo o paradigma funcional.

Cada componente da nossa implementação será representado por um módulo de Elixir. Um módulo nada mais é do que um *namespace* onde são dispostas funções com associação semântica, ou seja, que servem o mesmo propósito. O *Client* ou *Server Request Handler*, por exemplo, serão módulos.

Na ferramenta de organização de projetos de Elixir, a *Mix build tool*, projetos podem ser representados como um conjunto de *apps*. O código foi organizado de forma que as camadas são divididas em *apps* diferentes, até que se chegue na camada de aplicação, em que cada aplicação será um *app: Server, Naming Service* e *Client.*

As subseções desta seção descreverão a implementação do *middleware*, mostrando trechos de códigos de seus principais componentes.

3.3.1 *Client Proxy*

Uma das qualidades do paradigma funcional é permitir que funções gerem funções como resultado. No contexto do projeto, isso ajuda porque permite que seja criado um *Client Proxy* genérico. Não é necessário que haja, para cada função remota, uma função implementada no *Client Proxy* que provê a transparência de localização equivalente, do jeito que é feito em algumas implementações. Apenas uma função genérica é implementada, e essa função retorna uma função que, ao receber os argumentos, faz a invocação remota desejada.

O Código 3.1 mostra uma versão simplificada do *Client Proxy* para que possamos entender a lógica utilizada para retornar uma função que fará o papel de função remota. É uma função que, ao receber os argumentos *args* (que serão uma lista contendo os argumentos da função remota), fará a verificação dos argumentos para investigar se estão em conformidade com a assinatura da função remota, e caso estejam, ativará o *Requestor* passando os parâmetros de interesse.

Código 3.1: Função retornada pelo *Client Proxy*.

```
1.  def remote_function({{hst, prt},{mod_nm, func_nm, args_checker}}) do
2.    fn args ->
3.      # Se args estiver em conformidade com a assinatura da função
4.      # depois de ser checado por args_checker, o requestor é chamado
5.      Requestor.invoke({hst, prt}, {mod_nm, func_nm, args})
6.    end
7.  end
```

Vale ressaltar que, em Elixir, não existe a *keyword return*. A função retorna a última linha que ela avaliou. No caso, o retorno daí será o mesmo do *Requestor*.

Código 3.2: Uso do Serviço de Nomes

```
1.  lookup = InvocationLayer.ClientProxy.remote_function({
2.    {naming_service_ip, naming_service_port},
3.    {NamingService.LookupTable, :lookup, [&is_bitstring/1]}})
4.  {:ok, add_desc} = lookup.(["add"])
5.  add_func = InvocationLayer.ClientProxy.remote_function(add_desc)
6.  add_func.([3,4]) # = 7
```

O parâmetro *args_checker* é, na verdade, uma lista de funções. Cada função fará a checagem de um argumento passado na lista de parâmetros da função remota. Essa checagem pode dizer respeito ao tipo do argumento, mas também a qualquer outro interesse do desenvolvedor, como se um número respeita um certo limite, por exemplo. O Código 3.2, que mostra o uso do *Client Proxy* para consultar o serviço de nomes, e, posteriormente, recuperar a descrição da interface da função remota *add*, especifica, através do *args_checker*, na Linha 3, que para usar o *lookup*, só deve existir um argumento, que deve ser uma *string* (o que é verificado pela função *is_bitstring/1).*

3.3.2 *Requestor*

O papel central do *Requestor*, como vimos anteriormente, é o de construir a mensagem codificada, com a ajuda do *Marshaller*, e enviar essa mensagem com a ajuda do *Client Request Handler*. Se não houver nenhum tipo de erros de conexão, o *Requestor* desserializa a resposta e entrega ao *Client Proxy*. Em caso de erros de conexão, como a tentativa de tratar esses erros já é feita pela camada de transmissão de mensagens, esse erro é propagado para que o cliente seja avisado que houve um erro.

A função apresentada no Código 3.3 é invocada no *Requestor* logo após ele pedir para o *Client Request Handler* abrir a conexão com o *host* e o *port* que foram passados pelo *Client Proxy*. Utilizando esses dois argumentos, o *Client Request Handler* retorna ao *Requestor* o *socket* da conexão que foi aberta.

Código 3.3: Função *handle_communication* do *Requestor*

```
1.  defp handle_communication(sckt, mod_nm, func_nm, args) do
2.    marshalled_message = Marshaller.marshall({mod_nm, func_nm, args)}
3.    response_data =
4.      case ClientRequestHandler.send_message(sckt, marshalled_message) do
5.        :ok ->
6.          ClientRequestHandler.receive_message(sckt)
7.        error ->
8.          error
9.      end
10.   return_to_proxy(response_data)
11. end
```

Disso, o *Requestor* coordena toda a troca de mensagens através do *socket*. Caso tenha havido erro no estabelecimento da conexão ou na troca de mensagens, o *Client Request Handler* reporta o ocorrido para o *Requestor*, que apenas transmite o erro para as camadas de cima.

3.3.3 *Invoker*

O *Invoker,* assim que é inicializado (o que acontece no momento em que o lado do servidor da aplicação é inicializado), solicita ao *Server Request Handler* a criação de um processo que ficará escutando por requisições de conexões de possíveis novos clientes. Em seguida, o *Invoker* entra na função *manage_connections*, exposta no Código 3.4, que é executada enquanto o servidor estiver funcionando.

A função do Código 3.4, que o processo do *Invoker* executa infinitamente, é responsável por trocar mensagens com os processos do *Server Request Handler* que gerenciam as conexões. Essas mensagens são trocadas através das funções *send* e *receive* de Elixir. Além disso, essa função também coordena o processo de execução das funções remotas em si.

Código 3.4: Função *manage_connections* do *Invoker*

```
1.   def manage_connections do
2.     receive do
3.       {:new_connection, handler_pid} ->
4.         send handler_pid, :receive
5.       {:received, handler_pid, data} ->
6.         me = self
7.         spawn(fn ->
8.           send me, {:processed, handler_pid, process_request(data)}
9.         end)
10.      {:processed, handler_pid, reply} ->
11.        send handler_pid, {:send, reply}
12.      {:sent, handler_pid} ->
13.        send handler_pid, :close
14.    end
15.    manage_connections
16.  end
```

Basicamente, quando um processo diz que uma nova conexão chegou, o *Invoker* ordena que esse processo receba a mensagem enviada pelo cliente. Recebida a mensagem, ela é repassada ao *Invoker*, que cria um novo processo para obter a resposta da invocação através da função *call_function* mostrada no Código 3.5, obviamente com a ajuda do *Marshaller* para o tratamento de serialização e desserialização, que não está mostrada nos códigos expostos. Quando esse processo criado pelo *Invoker* para processar a requisição acaba a execução da função remota, ele envia a resposta ao *Invoker*. O *Invoker* então ordena que o processo do *Server Request* Handler a envie ao cliente. Tendo o processo enviado a resposta, o *Invoker*, finalmente, ordena que esse processo encerre a conexão.

Código 3.5: *Invoker* escolhe a função em tempo de execução

```
1.   def call_function({mod_nm, func_nm, args}) do
2.     apply(mod_nm, func_nm, args)
3.   end
```

A função *apply*, de Elixir, permite que escolhamos uma função de um dado módulo em tempo de execução. Em outras linguagens de programação, chamar a função correta normalmente exigiria *Reflection* (SMITH, 1982) ou uma sequência de controles de fluxo até ser achada a função correta.

3.3.4 *Marshaller*

A implementação do *Marshaller* no *middleware* é bastante simples. Ela apenas oferece uma interface (com as funções *marshall* e *unmarshall*) para que possamos interagir com as funções de Erlang *term_to_binary* e *binary_to_term*. Essas funções fazem a transformação de tipos de dados do Elixir para *bytes* e vice-versa.

Não houve uma preocupação em estabelecer transparência de heterogeneidade para que esse *middleware* fosse integrado com aplicações escritas em diferentes

linguagens de programação. Por isso, não é feita nenhuma transformação para um tipo de dados intermediário antes que os dados sejam transformados em *bytes* para serem transmitidos pela rede. Os tipos de Elixir são os únicos usados.

3.3.5 *Client Request Handler*

O *Client Request Handler* é o responsável pelo estabelecimento da conexão e envio da invocação remota para o servidor alvo. Por isso, ele é construído para lidar com a API oferecida por Elixir para gerenciar conexões no sistema operacional.

Além disso, o *Client Request Handler* também cuida de *timeouts* e tenta se recuperar de alguns tipos de erro de conexão. Mais especificamente, selecionamos como 10 segundos o *timeout* para o *Client Request Handler* esperar por respostas do servidor, tanto para estabelecimento de conexões quanto para recebimento das respostas das invocações remotas.

Código 3.6: Função *send_message* do *Client Request Handler*

```
 1.   defp _send_message(sckt, message, attempt) do
 2.     case :gen_tcp.send(sckt, message) do
 3.       error ->
 4.         if attempt == @max_attempts do
 5.           :gen_tcp.close(sckt)
 6.           {:error, :unable_to_send_message}
 7.         else
 8.           _send_message(sckt, message, attempt +1)
 9.         end
10.       _ ->
11.         :ok
12.     end
13.   end
```

Um exemplo de tratamento de erro de conexão está na função *send_message*, disposta no Código 3.6. Um número de tentativas predefinido é estabelecido antes que o *Client Request Handler* desista de enviar a mensagem e propague o erro para o *Requestor*. Caso a mensagem não possa ser enviada, como visto na Linha 6 do Código 3.6, essa função retorna o erro que informa que não foi possível enviar a mensagem.

3.3.6 *Server Request Handler*

No *Server Request Handler*, ao invés de tratar diretamente com a API oferecida por Erlang para lidar com conexões, optamos por usar a biblioteca *Ranch* para Erlang, detalhada em Nine Nines (2012). Essa biblioteca é um *Socket Acceptor Pool* para protocolos TCP. Isso quer dizer que, iniciado o servidor, essa biblioteca cria um processo para escutar por pedidos de conexão e mais vários processos (um *pool* de processos) que aceitam as conexões escutadas concorrentemente. Isso faz com que o servidor seja muito mais escalável, sendo capaz de aceitar várias conexões em

intervalos de tempo muito pequenos, o que não seria possível se houvesse apenas um processo aceitando conexões por vez.

Código 3.7: Função que recebe invocações no *Server Request*

```
1.   def process_request(sckt, transport, invoker_id) do
2.     receive do
3.       :receive ->
4.         {:ok, data} = transport.recv(sckt, 0, @timeout)
5.         send invoker_id, {:received, self, data}
6.         process_request(sckt, transport, invoker_id)
7.       {:send, data} ->
8.         :ok = transport.send(sckt, data)
9.         send invoker_id, {:sent, self}
10.        process_request(sckt, transport, invoker_id)
11.      :close ->
12.        :ok = transport.close(sckt)
13.    end
14.  end
```

Sempre que uma nova conexão é aceita, o processo que estava no *pool* e a aceitou chama a função *process_request*, definida no *Server Request Handler* e mostrada no Código 3.7. Essa função interage justamente com a função do *Invoker* exposta no Código 3.4 para coordenar o envio e recebimento das mensagens.

A lógica é complementar àquela discutida no Código 3.4. Ao receber, do processo do *Invoker*, o comando *receive*, a função do Código 3.7 recebe a invocação remota serializada enviada pelo cliente e a envia para o *Invoker*. Já quando recebe o comando *send*, acompanhado da resposta serializada, esse processo envia a resposta ao cliente. Finalmente, ao receber o comando *close*, esse processo fecha a conexão e é encerrado.

3.3.7 Aplicações: cliente, servidor e serviço de nomes

Utilizando a implementação dos componentes do *middleware* explicada nas seções anteriores como camadas de infraestrutura, é possível implementar as aplicações desejadas no nosso sistema distribuído: a do serviço de nomes, que guardará os serviços disponíveis nas diferentes máquinas do sistema e proverá a transparência de localização ao usuário; o cliente, que buscará as funções no serviço de nomes e as utilizará, e o servidor, que cadastrará as funções no serviço de nomes, e as implementará para atender requisições do cliente.

A interação entre o cliente e o serviço de nomes é a única parte do código em que o desenvolvedor deve se preocupar com o fato da aplicação ser distribuída, já que cabe a ele saber onde está esse serviço e como usá-lo. Essa interação foi mostrada na Seção 3.4.1, quando falávamos sobre o *Client Proxy*. Uma vez obtendo funções no serviço de nomes, o cliente poderá usá-las de forma semelhante a invocações locais, já que o serviço de nomes abstrai do cliente a localização das funções remotas.

Código 3.8: Servidor cadastrando funções no serviço de nomes

```
1.  naming_service_address = {localhost, 5050}
2.  naming_service_bind = {
3.    NamingService.LookupTable,
4.    :bind,
5.    [&is_bitstring/1, &is_tuple/1]
6.  }
7.  bind = ClientProxy.remote_function({
8.    naming_service_address,
9.    naming_service_bind
10. })
11. bind.([
12.   "add",
13.   {
14.     {server_ip, port},
15.     {Server.Application, :add, [&is_number/1, &is_number/1]}
16.   }
17. ])
```

Já o servidor, na sua inicialização, deve cadastrar as funções que ele implementa no serviço de nomes. Por isso, ele também precisa saber a localização desse serviço. O servidor também usa o *Client Proxy* e os componentes que integrariam, normalmente, a parte do cliente na aplicação. Isso acontece porque, ao se comunicar com o serviço de nomes, ele precisa usar tais funcionalidades, já que ele estabelece a conexão e faz uma invocação remota (para cadastrar suas funções).

O Código 3.8 mostra com mais detalhes como é feito o cadastro da função *add*, implementada no servidor, no serviço de nomes. As Linhas 1-6 trazem informações primeiramente a respeito da localização do serviço de nomes e depois sobre a assinatura da função *bind* que lá está: uma função que recebe dois parâmetros, sendo o primeiro uma *string* e o segundo uma tupla, e que está localizada no módulo *NamingService.LookupTable*. Nas Linhas 7-10, o *Client Proxy* é usado para que a função remota *bind* possa ser utilizada como uma função local, e as Linhas 11-17 usam essa função para cadastrar a função *add*, passando como parâmetros a sua assinatura e sua localização. Note que os argumentos passados à função *bind* estão de acordo com a definição da assinatura dessa função: o primeiro elemento é uma *string* e o segundo, uma tupla.

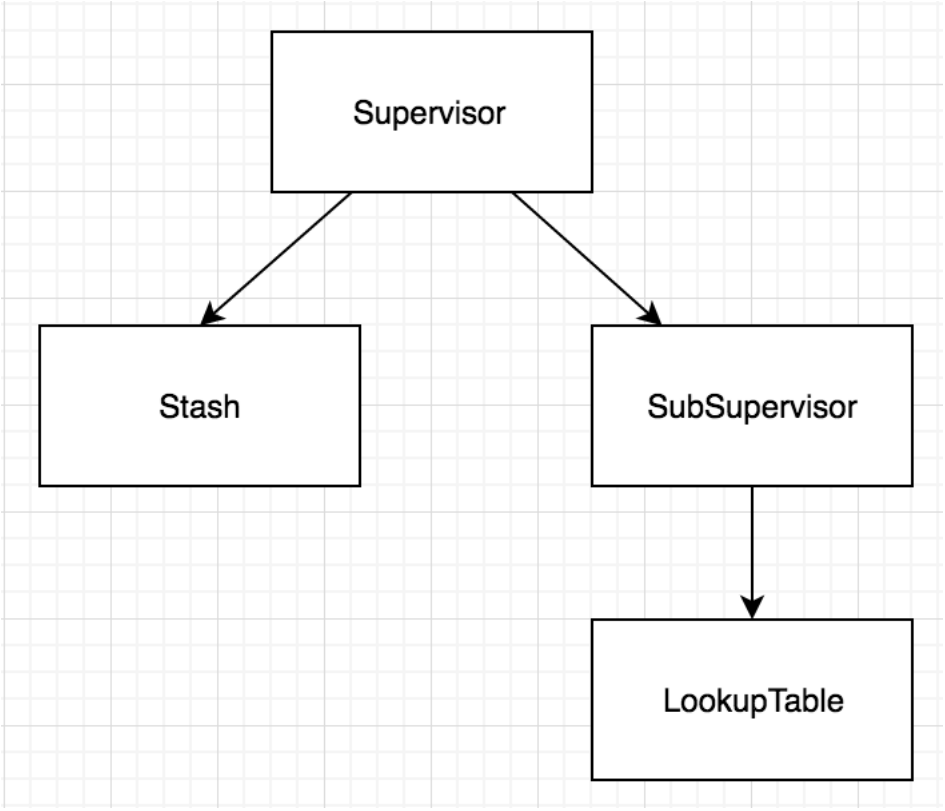
O serviço de nomes foi implementado levando em conta a tolerância a falhas. Ele conta com uma árvore de supervisores para garantir que haja recuperação de erros nessa aplicação, de forma que partes da aplicação que quebrem sejam reiniciadas.

Mais especificamente, como pode ser visto na Figura 3.4, o *Supervisor* da aplicação inicializa dois processos: um deles é o *Stash*, e o outro um *SubSupervisor*, que supervisionará a *LookupTable*. O *Stash* funciona, no contexto do serviço de nomes, como um *backup* da *LookupTable* – toda vez que esse processo for inicializado, ele carrega a tabela que está guardada no *Stash*, e toda vez que for terminado, salva a tabela lá. A inicialização e reinicialização (em caso de erros) da *LookupTable* é feita pelo *SubSupervisor*, e o semelhante para o *Stash* é feito pelo

Supervisor. Esse esquema garante que não há maneira de se perder informação, a não ser que a máquina que execute esses processos pare de funcionar em sua totalidade.

Vale ainda ressaltar que para implementar a *LookupTable* e o *Stash* usamos o *GenServer*[4], um módulo de Elixir que define o comportamento de um servidor numa aplicação cliente/servidor. A vantagem de usar esse módulo é que ele já contém um conjunto de funções padrão e funcionalidades desejáveis num servidor, como a de reportar erros. Além disso, ele consegue manter estado, o que é uma *feature* desejável no nosso contexto, já que precisamos manter, no serviço de nomes, a tabela atualizada de serviços. O serviço de nomes também usa o *Invoker* e os componentes de um servidor, afinal, é isso que ele é no fim das contas: um servidor do sistema distribuído que recebe invocações para atualizar sua tabela.

Figura 3.4: Árvore de supervisores no serviço de nomes



Fonte: Elaborada pelo autor

Agora que demonstramos como construir essas três aplicações, definiremos, no próximo capítulo, quais funções serão cadastradas no serviço de nomes, a fim de analisar o desempenho do *middleware*.

[4] http://elixir-lang.org/docs/stable/elixir/GenServer.html

3.4 Considerações Finais

Este capítulo detalhou o *middleware* implementado, explorando aspectos como arquitetura e implementação. Através dele, pudemos perceber como é realmente possível escrever código mais coeso usando uma linguagem funcional, além da facilidade que essa linguagem provê para concorrência e tolerância a falhas.

O próximo capítulo explora os resultados em termos de escalabilidade, mostrando vantagens e desvantagens de se ter feito essa implementação. Pontos a serem melhoradas na implementação vista também serão comentados posteriormente.

CAPÍTULO 4

Avaliação Experimental do *Middleware*

Neste capítulo, avaliaremos o desempenho do *middleware* desenvolvido, investigando o impacto causado pelo uso do paralelismo e mostrando o funcionamento de alguns requisitos funcionais. Iremos comparar o tempo de resposta de funções invocadas remotamente e de forma concorrente por vários clientes. Vale ressaltar que os códigos utilizados por todos os experimentos feitos pode ser encontrado no GitHub do projeto[5].

4.1 Hardware Utilizado

Para realizar as avaliações, utilizamos duas máquinas conectadas na mesma rede local. Uma das máquinas executou os clientes realizando as invocações remotas de forma concorrente, e a outra executou o servidor que processava as invocações.

Escolhemos a máquina com mais recursos para executar o servidor porque essa máquina teria que lidar, além da execução das funções requisitadas pelos clientes, com o gerenciamento de conexões paralelas, o que exige um bom poder de processamento.

Tabela 4.1: Configuração das duas máquinas utilizadas

Computadores					
	RAM	HD	Núcleos	SO	Processador
Clientes	4GB	1 TB	2	Windows 10	Intel i3
Servidor	8GB	250 GB	4	OS X Yosemite	Intel i5

Fonte: Elaborada pelo autor

Na Tabela 4.1 é possível observar mais detalhadamente as configurações das máquinas usadas nos experimentos, de forma a se perceber que a máquina do servidor possui mais recursos, em termos de capacidade de processamento, do que a que simula os clientes.

4.2 Método de Avaliação

Dentre algumas características a serem avaliadas estão o paralelismo e a tolerância a falhas. Nosso método de avaliação foi definido, portanto, levando em conta esses dois fatores principalmente.

[5] https://github.com/mmfrb/tg

Sobre o paralelismo, queremos investigar o desempenho ao aplicá-lo na linguagem de programação escolhida, tanto no *middleware* em si, que será responsável por gerenciar conexões simultâneas com invocações concorrentes por parte de diferentes clientes; quanto nas funções remotas, que foram pensadas para realizarem tarefas com diferentes granularidades, que sabemos que é um fator que influi no desempenho de aplicações paralelas.

Para avaliar o desempenho do paralelismo no *middleware*, que é feito através da criação de processos que gerenciam as conexões de forma concorrente, usamos uma função que exige pouco processamento por parte do servidor, e nos ativemos ao desempenho quando a quantidade de clientes varia. Isso nos ajudará a medir em quanto o *middleware* é impactado quando quantidade de clientes que o usa aumenta, determinando se o *middleware* é escalável nos limites determinados pelo experimento.

Já para avaliar o paralelismo nas funções remotas, escolhemos uma função que exige um bom nível de processamento e que pode ser paralelizada, e observamos como o desempenho se comporta quando um número fixo de clientes invoca tal função, alterando os seus parâmetros de forma que ela exija níveis de processamento diferentes do servidor. Esse experimento visa analisar o uso do paralelismo na linguagem em si, e o quanto esse uso traz de *overhead*.

Sobre a tolerância a falhas, pudemos ver no Capítulo 3 que ela foi implementada com maior esforço no serviço de nomes, para que, numa eventual falha no processo responsável por manter a tabela de serviços, a informação não seja perdida, e sim recuperada quando o processo for reiniciado. Mostraremos um caso em que o serviço de nomes falhe durante uma série de invocações remotas, que continuarão conseguindo obter a localização dos serviços depois que ele falhou e foi devidamente reiniciado.

4.2.1 Avaliando o Paralelismo

Para avaliar o paralelismo, tendo em mente a estratégia explicada na seção 4.2, elencamos as seguintes funções:

- **Operação Aritmética***:* Essa é a função que exige pouco processamento. Basicamente, vai receber, como argumentos, dois números e uma função para aplicar nesses dois números, como por exemplo a multiplicação. Como dito previamente, mediremos o tempo de resposta do servidor ao variar a quantidade de clientes fazendo invocações remotas de forma concorrente. Essa função também prova o cumprimento do requisito **RF03**, que permite a invocação remota de funções de alta ordem.

- **Números Primos**: Essa função exige um pouco mais de processamento. Ela receberá, como argumento, um intervalo de números, e deverá achar, nesse intervalo, quais os números primos. Dependendo do tamanho do intervalo, essa função pode usar muito processamento. Ela será implementada tanto de forma paralela (múltiplos processos serão usados para achar primos em subintervalos

que não se interceptam) quanto na forma normal (apenas um processo para todo o intervalo), e haverá uma comparação de desempenho.

É válido ressaltar que, na máquina que representa os clientes, utilizamos processos para simular clientes diferentes, de forma que criamos vários processos, sendo cada um deles um cliente.

A métrica que utilizamos para medir o desempenho foi o tempo decorrido para o *middleware* servir uma requisição. Em cada processo alocado pelo *Server Request Handler* para tratar uma nova conexão, medimos o tempo que ele leva desde o momento em que recebeu a mensagem até quando o *Invoker* dá a resposta à mensagem recebida.

Como existem múltiplas invocações, vindas de diferentes clientes, extraímos, ao final, a média do tempo decorrido para servir todas as invocações. Os tempos de resposta foram desconsiderados para uma parte pequena de invocações que não obtiveram sucesso por prováveis problemas de limitações nos *sockets* e da rede em geral.

4.2.2 Avaliando a Tolerância a Falhas

Para avaliar a tolerância a falhas no serviço de nomes, queremos pensar uma maneira de causar a falha no processo responsável por manter a tabela com os serviços cadastrados. O supervisor desse processo, então, fará com que ele seja reiniciado, recuperando a tabela de um outro processo que ele usou para salvá-la assim que foi terminado.

Por isso, implementamos, no serviço de nomes, além das funções *bind* e *lookup*, uma nova função: a função *destroy*. Essa função, que poderá ser invocada remotamente por outros nós do sistema distribuído, exposta no Código 4.1, faz com que o processo responsável por manter a tabela seja destruído.

Código 4.1: Função que destrói tabela de serviços na aplicação do serviço de nomes

```
1.   def destroy do
2.    GenServer.stop NamingService.LookupTable, :shutdown
3.   end
```

É válido lembrar que o servidor de nomes utiliza o módulo GenServer, de Elixir, explicado na Seção 3.4.7. A função *stop*, desse módulo, faz com que o processo passado como argumento seja encerrado. No contexto da nossa aplicação, estamos encerrando o processo que mantém a tabela de serviços cadastrados.

Para realizar esse experimento, só utilizaremos a máquina que fez o papel de servidor no experimento do paralelismo. Utilizaremos duas abas do terminal – na primeira, executaremos o serviço de nomes, e na segunda, utilizaremos o modo interativo de Elixir para, com a ajuda *Client Proxy*, executarmos os seguintes passos:

1. Cadastrar uma função de nome “Fake Function” no serviço de nomes através da operação *bind*.

2. Utilizar a função *destroy* para encerrar o processo que mantém a tabela no serviço de nomes. Nesse passo, o serviço de nomes vai emitir, no seu Log, uma mensagem, que explica que o processo que mantém a tabela foi encerrado.
3. Checar se a tabela foi recarregada na reinicialização do processo. Faremos isso utilizando a operação *lookup* para observarmos se a função “Fake Function” está na tabela.

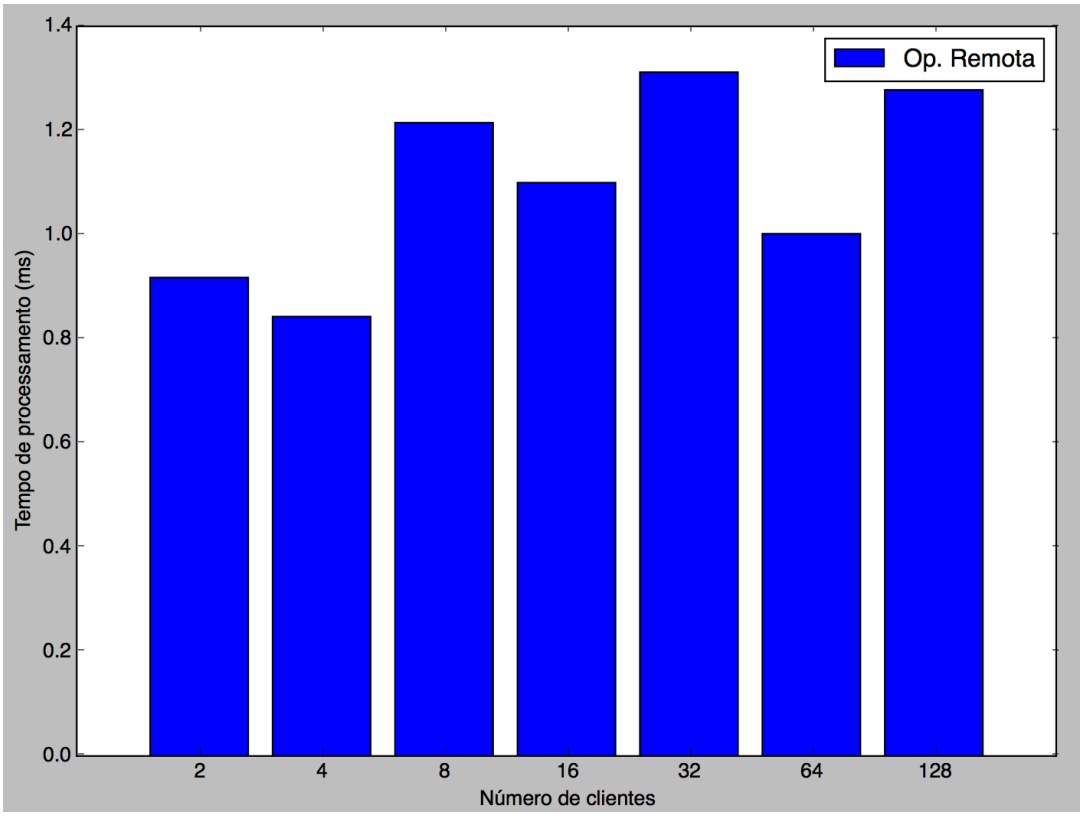
4.3 Resultados

Essa seção mostra os resultados obtidos nas avaliações quanto ao paralelismo e também quanto à tolerância a falhas, utilizando o método de avaliação explicado na Seção 4.2.1.

4.3.1 Paralelismo

Primeiramente, a Figura 4.1 explicita como varia o tempo de processamento médio no servidor ao receber múltiplas invocações de um número variado de clientes, de forma simultânea. Cada invocação usa a função Operação Aritmética previamente definida, fazendo a multiplicação dos números 3 e 5. É válido ressaltar que cada cliente realiza 3 mil invocações remotas consecutivas, a uma taxa de 4 requisições por segundo. Em cada uma delas, a conexão é aberta, e ao final, fechada.

Figura 4.1: Função remota de pouco processamento com variação no número de clientes



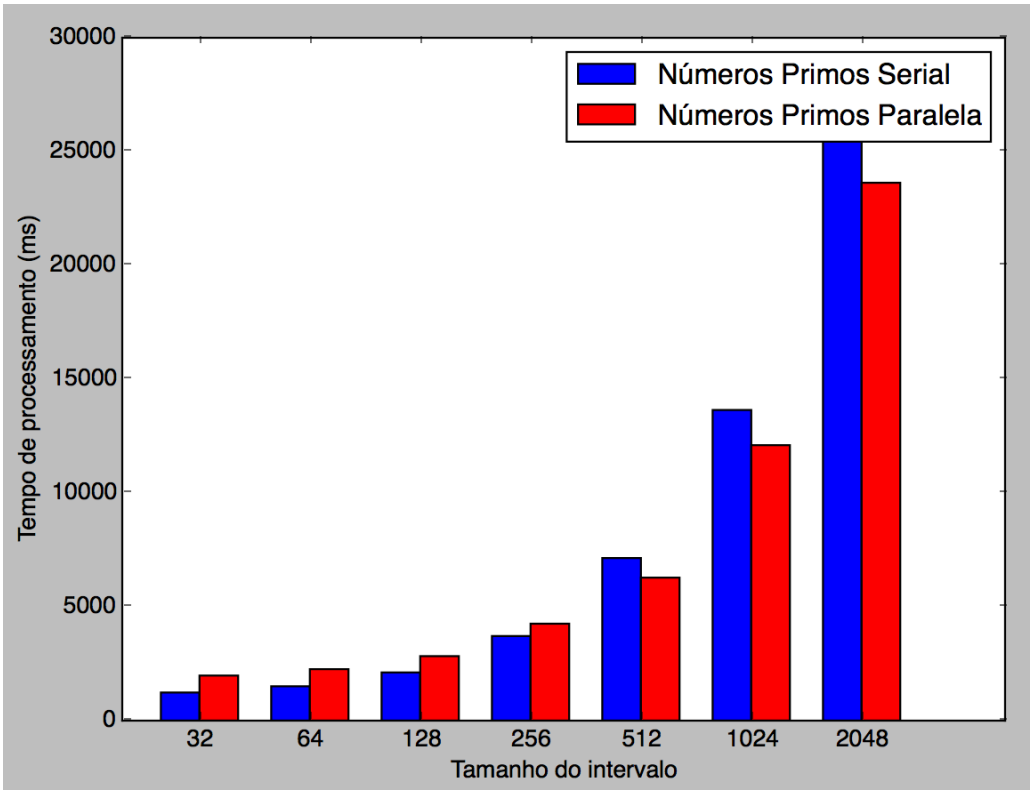
Fonte: Elaborada pelo autor

Como pode ser visto na Figura 4.1, testamos 7 números de clientes diferentes. Podemos perceber que o tempo médio de processamento de uma requisição no servidor se mantém relativamente estável com o aumento do número de clientes utilizando o *middleware* para fazer chamadas a procedimentos remotos simultaneamente.

O fato do tempo de processamento não variar bruscamente quando alteramos o número de conexões simultâneas mostra que o *middleware* está, de fato, utilizando de forma bastante eficiente os recursos da máquina, atendendo o requisito **RNF01**. Novos processos são alocados para gerenciar novas conexões, e o *overhead* de controlar esses processos não afeta de uma forma agressiva o desempenho do *middleware*. O *middleware*, portanto, se mostrou escalável quando aumentamos o número de clientes de 2 para até 128, com cada cliente fazendo até 4 requisições por segundo.

A Figura 4.2 mostra o quanto o paralelismo usado nas funções impacta na capacidade do servidor de processar as invocações, avaliando o *overhead* que a linguagem traz para o uso de paralelismo nas funções remotas. Para medir tal desempenho, utilizamos a função Números Primos. Os resultados são mostrados como a média dos tempos de processamento no servidor quando 16 clientes invocam essa função simultaneamente. Cada cliente invoca a função por 3 mil vezes consecutivas, enviando 4 requisições por segundo. A versão paralela dessa função é feita de forma que 32 processos ficam responsáveis por subintervalos disjuntos. Cada valor do eixo das abcissas indica o último valor analisado pela função no intervalo, que começa sempre em 1.

Figura 4.2: Impacto do paralelismo na execução de funções



Fonte: Elaborada pelo autor

Para intervalos menores, podemos notar que é melhor usar a versão da função que não é executada paralelamente. Isso ocorre porque, para esses valores, a granularidade da função é muito fina, de forma que o *overhead* de criar e gerenciar os processos que executam a função paralela supera o custo de executar a função de forma sequencial. Para tamanhos de intervalo acima de 256, porém, a tarefa ganha granularidade, e usar os 32 processos paralelos para executá-la faz com que os 16 clientes recebam suas repostas de forma mais rápida. Para valores maiores do que 512, a diferença aumenta: é claramente melhor usar a função Números Primos de forma paralela, já que com seu uso, os clientes recebem as respostas das invocações remotas mais rapidamente. É esperado que a diferença continue aumentando caso sejam testados intervalos maiores. Obviamente, outras abordagens, como variar o número de processos que realizam a tarefa de forma paralela, pode melhorar ainda mais a eficiência.

Os resultados da Figura 4.1 e 4.2 nos mostram, portanto, que por mais que o paralelismo no *middleware* implementado, por si só, já apresente um bom nível de escalabilidade para 2 até 128 clientes que fazem 4 requisições por segundo, utilizar paralelismo nas funções, em Elixir, além de ser bastante prático, pode trazer resultados bastante desejados em termos de eficiência nas respostas às invocações dos clientes em um sistema distribuído.

4.3.2 Tolerância a falhas

O primeiro passo é abrir uma aba na linha de comandos do computador para executar o serviço de nomes. O serviço de nomes oferece uma interface para uso na linha de comandos. Tudo que precisamos fazer, para a aplicação, é informá-la qual a porta que ela deverá usar para escutar requisições tanto para o cadastramento de funções quanto para *lookup*. Isso está exposto no Código 4.2, que escolheu a porta 5050.

Código 4.2: Ativação do serviço de nomes

```
1.  $./naming_service --port 5050
```

Numa segunda aba da linha de comandos, executamos os passos descritos na Seção 4.2.2.

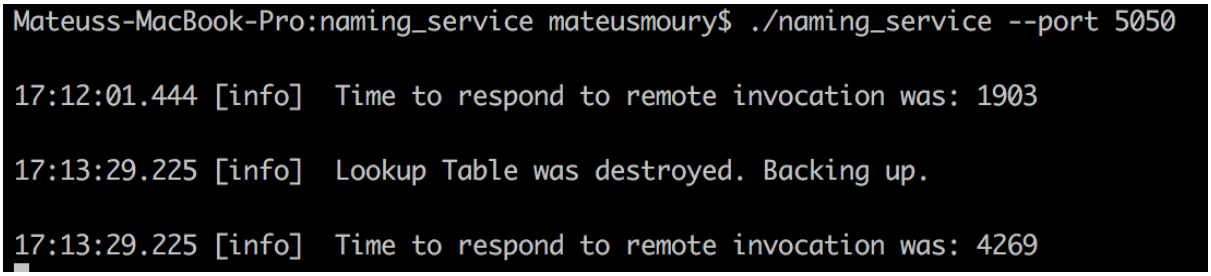
Código 4.3: Modo interativo: cadastrando função no serviço de nomes e o destruindo

```
 1.  naming_service_loc = {:localhost, 5050}
 2.  bind_loc = {
 3.    NamingService.LookupTable, :bind, [&is_bitstring/1, &is_tuple/1]}
 4.  dest_loc = {NamingService.LookupTable, :destroy, []}
 5.  fake_function_server_loc = {:fake_host, :fake_port}
 6.  fake_function_loc = {Fake.Module, :fake_function, []}
 7.  bind = ClientProxy.remote_function({naming_service_loc, bind_loc})
 8.  dest = ClientProxy.remote_function({naming_service_loc, dest_loc})
 9.  bind.(["Fake Function", {fake_function_server_loc, fake_function_loc}])
10.  dest.([])
```

Cadastraremos uma função, destruiremos o processo responsável por manter a tabela e após isso, checaremos se a tabela vinda do processo que foi reiniciado ainda contém a função cadastrada. O Código 4.3 mostra os dois primeiros passos.

A Linha 1 do Código 4.3 guarda uma tupla que diz onde está o servidor de nomes. Nesse caso, ele está na própria máquina, escutando pela porta 5050. As linhas 2-4 dizem respeito à localização das funções *bind* e *destroy*, que serão usadas nos passos 1 e 2, respectivamente, dentro do serviço de nomes, isso é, quais os módulos, os nomes das funções dentro desses módulos e que argumentos essas funções aceitam. As Linhas 5 e 6 definem o servidor e a localização dentro do servidor da função “Fake Function”. Como a função é falsa, já que não é importante de fato para o que queremos analisar, toda a informação dessas linhas é também falsa. As Linhas 7 e 8 usam o *Client Proxy* para gerar as funções *bind* e *destroy*, podendo invocá-las como se fossem funções locais. Finalmente, a Linha 9 cadastra a função e a Linha 10 destrói o serviço de nomes. A Figura 4.3 mostra o log gerado pela execução do serviço de nomes na linha de comandos do computador.

Figura 4.3: Log gerado pelo serviço de nomes após os passos 1 e 2 do experimento



Fonte: Elaborada pelo autor

A Figura 4.3 mostra, no seu primeiro log, as 17h12min, que o serviço de nomes demorou 1903ms para responder a invocação do *bind*. Os dois outros logs dizem respeito a invocação da função *destroy* (note que a hora dos acontecimentos é igual). Ao receber a chamada à função *destroy*, o serviço de nomes informa que a tabela foi destruída e que está prestes a fazer o *backup* dessa tabela em outro processo. O *backup* é feito, e o último log diz respeito ao tempo demorado para responder à invocação *destroy* do cliente.

Após isso, como é de se esperar, já que se utilizam supervisores para tratar tolerância a falhas, a tabela é reiniciada e copia de volta os dados da “Fake Function” que estavam armazenados no *backup*. Por isso, quando executamos a função *lookup*, conforme exposto no Código 4.4, conseguimos de volta o endereço da função.

Código 4.4: Recuperando a função depois da tabela ter sido destruída

```
1.   lookup_loc = {NamingService.LookupTable, :lookup, [&is_bitstring/1]}
2.   lookup = ClientProxy.remote_function({naming_service_loc, lookup_loc})
3.   lookup.(["Fake Function"]
4.   # = {{:fake_host, :fake_port}, {Fake.Module, :fake_function, []}
```

Quando o *lookup* é executado, são retornadas, pelo serviço de nomes, exatamente as informações que utilizamos quando fizemos o *bind* da “Fake Function”.

Isso prova que, apesar do processo ter sido reinicializado, nenhuma informação foi perdida. É muito importante ter essa persistência de informações em qualquer tipo de sistema distribuído de grande porte.

Como dito no início do capítulo, todo o experimento pode ser repetido através da utilização do código-fonte, seguindo as instruções no *readme* do projeto.

4.4 Vantagens e Desvantagens no Uso de Elixir

No começo da elaboração deste projeto, foi necessário um tempo extra para que se pudesse aprender uma nova linguagem de programação. Elixir, por seguir o paradigma funcional e ser uma linguagem orientada a processos, que podem se comunicar de forma simples, ofereceu uma nova maneira de pensar a resolução de problemas.

Apesar do *overhead* requerido para aprender a utilizar a linguagem, ela se provou de fácil aprendizado, com boas fontes e com uma comunidade na Web bastante prestativa. Todas as dúvidas que surgiram na implementação do projeto foram sanadas de forma rápida com consultas às fontes disponibilizadas.

Elixir, por ser uma linguagem nova, ainda peca por não ter algumas funcionalidades que estão para ser implementadas. Durante a implementação deste projeto, por exemplo, a versão usada da linguagem foi atualizada 2 vezes para que novas funcionalidades pudessem ser aproveitadas.

Apesar disso, Elixir apresenta uma sintaxe bastante amigável e de fácil aprendizado. Para atender aos requisitos que estávamos buscando, que eram os de paralelismo e tolerância a falhas, Elixir se provou uma linguagem surpreendente. Com poucas linhas de código é possível paralelizar qualquer tipo de aplicação, e criar processos que se comunicam.

O fato de Elixir ser uma linguagem funcional e, portanto, não armazenar estado e contar com a imutabilidade, fez com que os problemas de concorrência fossem amenizados ou não existissem, o que não seria necessariamente verdade com o uso de uma linguagem de orientação a objetos.

No geral, a linguagem atendeu o propósito de fazer um *middleware* de distribuição baseado em RPC com sucesso. Talvez não seria tão fácil fazer algo do tipo com uma linguagem orientada a objetos, justamente por não contar com várias das características de uma linguagem de paradigma funcional, como ausência de estado e facilidade para a implementação de sistemas concorrentes.

Por outro lado, elaborar um *middleware* orientado a objetos com Elixir certamente traria um esforço bastante maior do que se o mesmo *middleware* fosse pensado com uma linguagem orientada a objetos. Por não existir o conceito de objeto em uma linguagem funcional, mas existir, obviamente, o conceito de função em uma linguagem orienta a objetos, a adaptação de Elixir para construir um *middleware* orientado a objeto seria mais complicada do que o processo inverso. Conclui-se que, quando se desenvolvendo um *middleware*, é uma boa ideia pensar no que cada linguagem de programação pode trazer de benefício ao tipo específico de *middleware.*

Os requisitos do *middleware* podem dar dicas sobre quais seriam as melhores linguagens para que se fizesse a implementação.

Por experiência do autor, se for vontade do desenvolvedor implementar um *middleware* que suporta concorrência, capaz de estabelecer várias conexões simultaneamente, tendo facilidades para o uso de *sockets* e tolerante a falhas, utilizando uma linguagem expressiva e inovadora, que conta com uma comunidade bastante prestativa, Elixir é uma séria opção a se levar em conta.

4.5 Considerações Finais

Neste capítulo, foram expostos os resultados das análises feitas no *middleware* desenvolvido. As análises focaram no uso do paralelismo, tanto para o *middleware* em si quanto para as funções remotas utilizadas no sistema. Além disso, mostrou-se que foi possível construir um componente tolerante a falhas, que foi o serviço de nomes. Todas essas análises foram feitas usando o hardware cujas características foram expostas no próprio capítulo. Ao final do capítulo, foram expostas, ainda, as vantagens e desvantagens na utilização de Elixir.

CAPÍTULO 5

Conclusão

Cada vez mais, sistemas distribuídos estão presentes no desenvolvimento de aplicações utilizadas na vida cotidiana, como redes sociais e jogos, por exemplo. Nos dias de hoje, é muito difícil desenvolver um sistema escalável sem que ele seja distribuído, já que o processamento requisitado por várias aplicações atuais é maior do que aquele oferecido por apenas um computador.

Implementar aplicações sem ter que se preocupar com o fato de serem distribuídas é uma característica almejada por qualquer desenvolvedor de um sistema distribuído, já que novos problemas surgem quando a aplicação passa a ser distribuída, como por exemplo problemas na rede. Por isso existe o *middleware*: para resolver os problemas de uma aplicação distribuída que não dizem respeito à lógica da aplicação em si, mas sim ao fato de distribuí-la.

Neste trabalho, apresentamos um *middleware* construído através da utilização de Elixir, uma linguagem funcional baseada em Erlang, para dar suporte a aplicações funcionais que utilizam chamadas a procedimentos remotos. A linguagem traz algumas características, como paralelismo a custo baixo e tolerância a falhas, que consideramos interessantes e resolvemos utilizar para implementar o sistema. O uso de Elixir foi considerado, no geral, vantajoso. O desafio em aprender uma linguagem funcional nova foi compensado pela qualidade da linguagem e pelas boas fontes de aprendizado, além de uma comunidade ativa para ajudar na web.

A arquitetura do nosso *middleware* foi baseada nos Padrões Remotos para a construção de sistemas de *middleware* orientados a objetos. A adaptação realizada para a construção de um *middleware* baseado em RPC foi simples, já que o fato de existirem objetos acaba trazendo uma complexidade maior a um projeto de *middleware*, principalmente no sentido de gerenciar os recursos do sistema. O objeto remoto deu lugar à função remota. O significado e o uso de todos os Padrões Remotos utilizados foram mantidos, e isso resultou numa facilidade maior para implementar e modificar o sistema.

Nosso *middleware* teve sucesso em prover ao desenvolvedor alguns tipos de transparência, como a de acesso e de localização. Para conseguir isso, além dos componentes usuais (o cliente e o servidor), nosso *middleware* contou com a implementação de um serviço de nomes tolerante a falhas. Através da utilização de uma linguagem de programação de paradigma funcional, também foi possível realizar a invocação de procedimentos remotos de mais alta ordem: invocar funções passando outras como parâmetro.

Foi possível, ainda, verificar, através dos resultados mostrados, como o uso do paralelismo no *middleware* para fazer a gerência de múltiplas conexões simultâneas fez com que o tempo médio de resposta para os clientes se mantivesse estável quando se aumentava a quantidade de clientes fazendo as invocações, até 128 clientes. Dessa

forma, pode-se dizer que o *middleware* cumpriu o objetivo de tentar utilizar grande parte do poder de processamento oferecido para melhorar seu desempenho, se tornando um *middleware* escalável nos limites estabelecidos pelos experimentos.

Apesar do *middleware* ter apresentado uma boa escalabilidade para um número crescente de clientes até certo ponto, também foi provado através dos experimentos que o uso de paralelismo nas funções remotas, dependendo da granularidade delas, pode trazer um ganho muito bom no tempo médio de resposta aos clientes, o que significa que, numa aplicação distribuída que leva em conta a eficiência e rapidez de resposta aos múltiplos clientes, o uso do paralelismo em Elixir pode trazer ganhos relevantes.

Mesmo usando paralelismo para gerenciar conexões no *Server Request Handler*, nosso *middleware* disponibilizava de apenas um *Invoker* no servidor, que se comunicava com todos os processos controladores das conexões. Uma possível melhoria, portanto, seria a de disponibilizar mais *Invokers* para que existisse um paralelismo ainda maior, considerando que esse, possivelmente, foi o maior gargalo da nossa implementação. Uma outra forma de melhorar o *middleware* através do paralelismo seria o de implementar um *Marshaller* paralelo, que serializasse e desserializasse mensagens de forma concorrente, o que não foi feito no caso deste trabalho.

De uma forma geral, o projeto cumpriu o seu objetivo, que era o de utilizar Elixir para construção de um *middleware* de distribuição para comportar invocações a procedimentos remotos, analisando a eficiência do sistema, o que foi feito levando-se em conta técnicas de concorrência e paralelismo.

Referências Bibliográficas

ARMSTRONG, Joe. A History Of Erlang. ***Proceedings Of The Third ACM SIGPLAN Conference On History Of Programming Languages***. New York, NY: N.p., 2007. 1-26.

ARMSTRONG, Joe. The development of Erlang. **Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming**, 2., 1997, Amsterdam. New York, Ny: Acm, 1997. v. 2, p. 196 - 203.

BERNSTEIN, Philip A.. **Middleware: A model for distributed system service**s. Communications of the ACM, New York, NY, v. 39, n. 2, p. 86-98. 02/1996.

BIRRELL, ANDREW D.; NELSON, BRUCE JAY**. Implementing Remote Procedure Calls.** ACM Transactions on Computer Systems. Palo Alto, CA, 02/1984. Caderno p. 39-59. Disponível em: http://www.cs.virginia.edu/~zaher/classes/CS656/birrel.pdf. Acesso em: 14/04/2016

COLOURIS, George et al. **Distributed Systems: Concepts and Design**. Boston: Addison-Wesley, 2011.

CROWCROFT, John**. Open Distributed Systems**. Framingham, MA: Artech House Publishers, 1996.

EUGSTER, Patrick Th. et al. **The many faces of publish/subscribe.** Acm Computing Surveys, New York, v. 35, n. 2, p.114-131, jun. 2003.

HANSEN, Per Brinch**. The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls**. New York: Springer-Verlag, 2001.

Jia, Law. **The History, Evolution & Trends in Distributed Computing.** Southern Mississippi: Law J, 2013. Color. Disponível em: <https://prezi.com/9gobleqbzgp-/the-history-evolution-trends-in-distributed-computing/>. Acesso em: 14 jun. 2016.

LI, Kai; HUDAK, Paul. **Memory Coherence in Shared Virtual Memory Systems.** ACM Transactions On Computer Systems,New York, Ny, v. 7, n. 4, p.321-359, nov. 1989. Disponível em: <http://www.cs.virginia.edu/~zaher/classes/CS656/li.pdf>. Acesso em: 16 jun. 2016.
MICHIE, Donald. Memo Functions and Machine Learning. Nature. Edinburgh, p. 19-22. abr. 1968.

MISHRA, Kamal Sheel; TRIPATHI, Anil Kumar. **Some Issues, Challenges and Problems of Distributed Software System**. International Journal of Computer Science and Information Technologies. Varanasi, India, 07/2014. Caderno p. 3. Disponível em: http://www.ijcsit.com/docs/Volume%205/vol5issue04/ijcsit2014050420.pdf. Acesso em: 11/04/2016

NANZ, S.; FURIA, C. A. **A comparative study of programming languages in Rosetta Code**. In: International Conference On Software Engineering - Vol 1, 37. Proceedings. . [S.l.: s.n.], 2015. p.778–788.

NINE NINES (France). **Ranch**. 2012. Disponível em: <https://github.com/ninenines/ranch>. Acesso em: 21 jun. 2016.

OMG. **Common Object Request Broker: Architecture and Specification.** OMG Document Number 91.12.1 (1991).

PARKER, Don. **Understanding the FTP Protoco**l. 2005. Disponível em: <http://www.windowsnetworking.com/articles-tutorials/network-protocols/Understanding-FTP-Protocol.html>. Acesso em: 16 jun. 2016.

PELEG, David. **Distributed Computing: A Locality-Sensitive Approach**. Philadelphia, PA: SIAM, 2000. Disponível em: https://books.google.com.br/books?redir_esc=y&hl=pt-BR&id=T1hFWuDi1CsC&q=. Acesso em: 26/04/2016

ROUSE, Margaret**. Remote Procedure Call (RPC).** 2009. Disponível em: http://searchsoa.techtarget.com/definition/Remote-Procedure-Call. Acesso em: 10/04/2016

SMITH, Brian Cantwell. **Procedural reflection in programming languages**. 1982. 762 f. Tese (Doutorado) - Curso de Ciência da Computação, Massachusetts Institute Of Technology, Massachusetts, 1982. Disponível em: <http://dspace.mit.edu/handle/1721.1/15961#files-area>. Acesso em: 14 jun. 2016.

SUN. 2000. **Java Remote Method Invocation Specification**. Sun Microsystems, Santa Clara, CA.

SUN. 2002. **JavaSpaces Service Specification**. Sun Microsystems, Santa Clara, CA.

TANENBAUM, Andrew S.; STEEN, Maarten Van. **Distributed Systems:Principles and Paradigms**. 2. ed. Upper Saddle River, NJ: Pearson, 2006.

TOMLINSON, Ray. **The First Network Email**. Disponível em: <http://openmap.bbn.com/~tomlinso/ray/firstemailframe.html>. Acesso em: 29 maio 2016.

VALIM, José. **Elixir.** 2012. Disponível em: <https://github.com/elixir-lang/elixir>. Acesso em: 11 jul. 2016.

VÖLTER, Markus; KIRCHER, Michael; ZDUN, Uwe**. Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware**. Chichester: Wiley, 2004.