Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Implementação e Análise de um *middleware* baseado em RPC utilizando Erlang

Miguel Rodrigues Araújo

Trabalho de Graduação

Recife, Dezembro de 2016.

Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Implementação e Análise de um *middleware* baseado em RPC utilizando Erlang

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.

Aluno: Miguel Rodrigues Araújo

(mra2@cin.ufpe.br)

Orientador: Prof. Dr. Carlos André Guimarães Ferraz

(cagf@cin.ufpe.br)

Recife, Dezembro de 2016.

Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Implementação e Análise de um *middleware* baseado em RPC utilizando Erlang

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: Prof. Dr. Carlos André Guimarães Ferraz

_____

Avaliador: Prof. Dr. Nelson Souto Rosa

Recife, Dezembro de 2016.

Para Miguel, Minervina, Síria e Amora.

Agradecimentos

Antes de tudo, quero agradecer a Deus pela oportunidade concedida. Na realidade em que vivo isto é incomum. Por isso, sou grato por tudo o que aconteceu nesta etapa da minha vida. Eu levarei os ensinamentos para sempre.

Agradeço a toda minha família por todo o esforço desprendido para que isto se tornasse realidade. Foram incontáveis os momentos que eu precisei da ajuda de todos, e sempre consegui manter o foco em um dos meus sonhos porque todos eles abriram mão, de certa forma, para me ajudar. Sem vocês eu não teria conseguido chegar até aqui. Eternamente grato.

Agradeço ao meu orientador Prof. Dr. Carlos Ferraz por ter sido mais do que um professor nesta jornada. Seu apoio foi além dos limites acadêmicos, e eu me sinto felizardo em ter a oportunidade de aprender com ele. Além do mais, sem o seu apoio muito provavelmente este trabalho não teria existido. Ademais, agradeço a oportunidade no programa de monitoria e orientação em estágio.

Reconheço, sem dúvidas, que o suporte dos professores e funcionários do Centro de Informática da Universidade Federal de Pernambuco durante minha graduação foi essencial. Graças a todos os envolvidos, tive a chance de aprender em um lugar de excelência.

Por último, mas não menos importante, eu agradeço aos amigos feitos nessa caminhada. Uns mostraram que eu era capaz de fazer além do que eu imaginava. Outros abriram meus olhos para as oportunidades que a vida nos permite. Paulo, Rafael, Franclin, Adaílson e Rubens até a próxima!

Ademais, sou grato a todos que porventura eu tenha me esquecido de mencionar, mas que de alguma forma, direta ou indiretamente, contribuíram para este trabalho de graduação.

01000101 01100011 01101100 01100101 01110011 01101001 01100001 01110011 01110100
01100101 01110011 00101100 00100000 00110011 00111010 00110001 00001010

Resumo

A demanda por processamento e armazenamento de dados cresce constantemente a cada ano. O aumento de usuários com acesso a Internet em 2015 chegou a expressivos 43%, quase metade da população mundial [1]. A construção de sistemas distribuídos vem para suprir estas necessidades, muitas vezes, geradas por esse aumento. O E*stado da Arte,* no contexto de construção de um Sistema Distribuído, é concretizado ao não ser necessário que o programador se preocupe com tarefas ortogonais à sua aplicação, como problemas de rede ou localização. O objetivo deste trabalho é construir um *middleware* baseado em chamadas remotas de procedimento que se destina a facilitar o desenvolvimento de aplicações distribuídas. Chamadas Remotas de Procedimento estendem a conhecida abstração de chamadas de procedimento para sistemas distribuídos. Eles visam permitir que uma chamada remota de procedimento se comportasse como se fosse uma invocação local (VÖLTER; KIRCHER; ZDUN, 2004). A tecnologia para o desenvolvimento do projeto será Erlang porque é uma linguagem de programação voltada para sistemas distribuídos que possui algumas características interessantes como, por exemplo, tolerância a falhas e suporte ao paradigma funcional. Além disto, será feita no trabalho uma análise quanto ao desempenho do *middleware*.

Palavras-chave: Sistemas Distribuídos, *Middleware*, Erlang, Chamadas Remotas de Procedimento, Concorrência, Tolerância a Falhas, Paradigma Funcional.

Abstract

Data and storage processing demand is constantly growing every year. User penetration into Internet at 2015 reached expressive 43%, almost half the world population. Distributed Systems came to supply these needs, many times, created by this growth. The State of Art at distributed systems context is realized so that the programmer does not need to worry about orthogonal tasks in his application, such as network or location issues. The aim of this work is to build a remote procedure call *middleware* that is intended to facilitate the development of distributed applications. Remote procedure calls extend the well-known procedure call abstraction to distributed systems. They aim at letting a remote procedure invocation behave as if it were a local invocation (VÖLTER; KIRCHER; ZDUN, 2004). The implementation will be done in Erlang because it is a programming language focused on Distributed Systems that has some interesting characteristics, such as fault tolerance and functional paradigm. In addition, a performance analysis will be made on the *middleware*.

Keywords: Distributed Systems, *Middleware*, Erlang, Remote Procedure Call, Concurrency, Fault Tolerance, Functional Paradigm.

Lista de Abreviações

API: Application Programming Interface (Interface de Programação de Aplicação)

ERTS: Erlang Run-Time System Application

IoT: Internet of Things (Internet das Coisas)

IP: Internet Protocol (Protocolo de Internet)

KVS: Key-Value Server (Servidor Chave-Valor)

NFS: Network File System (Sistema de Arquivos Distribuídos)

OO: Orientação a Objetos

P2P: Peer-to-Peer (Par-a-Par)

RF: Requisito Funcional

RNF: Requisito Não Funcional

RPC: Remote Procedure Call (Chamada Remota de Procedimento)

SO: Sistema Operacional

TCP: Transmission Control Protocol (Protocolo de Controle de Transmissão)

URL: Uniform Resource Locator (Localizador Uniforme de Conteúdo)

WWW: World Wide Web

Lista de Ilustrações

Sumário

Introdução

Segundo o relatório do *State of Connectivity* (INTERNET.ORG, 2015), um estudo anual requisitado pelo *Facebook*[1], no final de 2015 pouco mais de três bilhões de pessoas estão conectadas a Internet. Dentro deste universo, existe a *WWW* (também conhecido por Web), um espaço para troca de informações via *URL*, que é utilizada por milhões de pessoas todos os dias para os mais variados propósitos. Utilizando um navegador padrão, o usuário tem acesso a informações armazenadas em servidores espalhados pelo globo. Isto permite a ilusão de que tudo está disponível localmente no computador. Na verdade, a Web é um grande sistema distribuído que está acessível como uma fonte única de informação para o usuário a partir de um clique (KAUR, MADHURIMA; MADHULIKA, 2013, p. 35).

Existem muitas definições para o conceito de sistema distribuído, mas todos eles se complementam. COULOURIS, DOLLIMORE e KINDBERG (2011) definem um sistema distribuído como "uma coleção de computadores autônomos ligados por uma rede e equipado com um software distribuído"; TANENBAUM e VAN STEEN (2001) expressam como "uma coleção de computadores independentes que aparecem aos seus usuários como um sistema único e coerente". Já PUDER, RÖMER e PILHOFER (2005) especificam da seguinte forma: "um sistema distribuído é um sistema de informação que possui um conjunto de computadores independentes que cooperam entre si através de uma rede de comunicação para alcançar um objetivo específico". Por fim, Leslie Lamport, uma vez cravou que "um sistema distribuído é aquele em que uma falha de um computador que você sequer sabia da existência faz com que o seu não funcione", refletindo a quantidade imensa de desafios que existem para se programar um sistema distribuído.

ZDUN, KIRCHER e VÖLTER (2004) explicaram que, em geral, há dois motivos para se utilizar sistemas distribuídos: (i) inerentes ao problema e (ii) propriedade ao problema. Para o primeiro caso, por exemplo, se um computador no Centro de Informática[2] da Universidade Federal de Pernambuco deseja acessar uma página hospedada em um servidor no Departamento de Física[3] da Universidade de Harvard, a página precisa ser transportada pela rede, não há outra forma de resolver isto. Por outro lado, no segundo

[1] https://www.facebook.com/
[2] http://www2.cin.ufpe.br/site/index.php
[3] https://www.physics.harvard.edu/

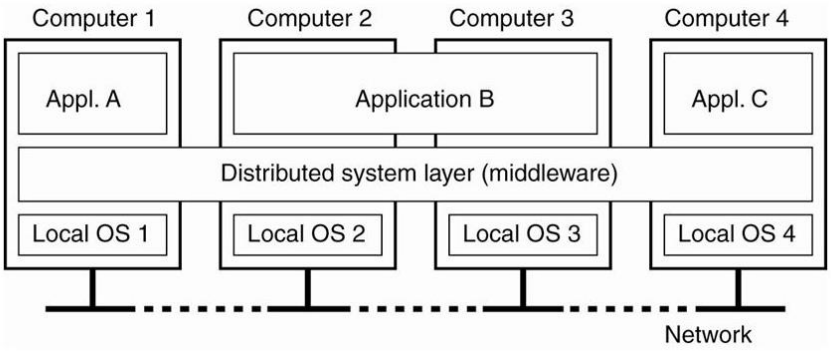
caso, quando se deseja melhorar um sistema adicionando características de sistemas distribuídos, por exemplo, tolerância a falhas e/ou garantir escalabilidade.

Um equívoco comum entre pessoas, enquanto discutem sobre sistemas distribuídos, é definir isto como um novo nome para rede de computadores (KAUR, MADHURIMA; MADHULIKA, 2013, p. 36). No entanto, há uma importante distinção. Um sistema distribuído é construído no topo da rede e tenta esconder a existência de múltiplos computadores. Aparecem como uma única entidade que disponibiliza os serviços requisitados. Uma rede de computadores é um mediador de entidades que provê a troca de mensagens baseada em protocolos conhecidos.

Maior parte das pessoas, quando constroem uma aplicação distribuída pela primeira vez, assumem oito suposições elencadas por ROTEM-GAL-OZ (2008). Primeiramente elucidadas por PETER DEUTSCH (1980), todas elas provam serem falsas no longo prazo, causam transtornos e experiências ruins.

Os desafios durante o desenvolvimento de sistemas distribuídos são enormes. Um sistema deve se tornar distribuído, somente e se somente, houver necessidade real. Distribuição adiciona complexidade, concorrência e outros problemas em potencial para a aplicação. Uma solução comum para estes problemas é adicionar uma camada de *software* chamada *middleware*, que esconde heterogeneidade e provê transparência para os desenvolvedores da aplicação. Neste contexto, transparência significa permitir chamadas remotas de procedimentos idênticas às invocações locais (ZDUN; KIRCHER; VOLTER, 2004). No entanto, realizar este tipo de chamada introduz novos erros, latência e outras adversidades que incapacita o *middleware* de ser uma infraestrutura perfeitamente transparente ao desenvolvedor.

Figura 1: Middleware, uma camada entre os Sistemas Operacionais e as Aplicações.



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

O *middleware* construído neste trabalho utiliza a abordagem definida por BIRREL e NELSON (1984). O escopo é permitir que chamadas remotas de procedimentos se comportassem como chamadas locais, na perspectiva do desenvolvedor. Esta abordagem é semelhante ao *modelo cliente-servidor*[4], no qual o programa que invoca o procedimento é representado pelo cliente e o computador que processa a requisição é o servidor.

O projeto utiliza Erlang (ARMSTRONG, 2003) como linguagem de programação porque ela foi construída para suportar aplicações distribuídas e tolerantes a falhas num ambiente de tempo real e ininterrupto. *Amazon*[5], *Facebook* [6]e *Yahoo*[7], empresas sólidas utilizam Erlang para prover concorrência e distribuição. Esta linguagem é multi-paradigma, suportando os paradigmas funcional e concorrente.

Este trabalho está dividido em cinco capítulos, incluindo este. No capítulo seguinte serão discutidos os fundamentos necessários para ter conhecimento necessário para entender o projeto, tais como definições e problemáticas inerentes à implementação de *middleware*, além dos trabalhos relacionados a esta pesquisa. Adiante, o capítulo três discorrerá sobre os requisitos, arquitetura e implementação desenvolvidas neste trabalho, explicando as decisões tomadas e como cada componente funciona. No capítulo quatro será feita uma análise profunda sobre a eficiência e disponibilidade do *middleware*. Por fim, o capítulo cinco contém as principais conclusões desta pesquisa, dificuldades encontradas e possíveis trabalhos futuros.

[4] https://pt.wikipedia.org/wiki/Cliente-servidor
[5] https://www.amazon.com
[6] https://www.facebook.com/
[7] https://br.yahoo.com/

Fundamentos

Adiante serão apresentados neste capítulo os assuntos que o leitor precisa entender para ser capaz de compreender o trabalho aqui desenvolvido. Primeiramente, será introduzido o que é sistema distribuído, *middleware* e chamadas remotas de procedimentos, considerando suas características e benefícios. Em seguida, haverá uma introdução ao paradigma funcional e, por conseguinte, a linguagem de programação Erlang que será utilizada no desenvolvimento do projeto.

Sistemas Distribuídos

Um sistema distribuído é um conjunto de computadores ligados em rede, com um software permita a distribuição de recursos e a coordenação de atividades oferecendo, idealmente, um sistema integrado (COLOURIS el al., 2011). Os componentes desse sistema se comunicam através de mensagens - não existem variáveis globais compartilhadas. Este definição acarreta em algumas consequências: concorrência, sistema assíncrono e falhas independentes (MADEIRA, 2010).

Para a primeira consequência, isto significa que o sistema é capaz de compartilhar recursos e realizar múltiplas tarefas num dado momento. Isto implica da necessidade de coordenar o acesso aos recursos compartilhados (*hardware*, *software* e dados). Para o segundo ponto, sistema assíncrono, por não existir um relógio global, característica inerente de um sistema distribuído, não haverá um limite para o tempo de comunicação. Por fim, falha independente indica que se um nó falhar, isso não deve impedir dos outros componentes de funcionar (seja por perda de mensagens, duplicação, reordenação, *etc*.).

Sistemas distribuídos existem para os mais variados motivos: seja melhor relação custo/benefício, maior capacidade de processamento, maior confiabilidade e disponibilidade, crescimento gradativo de capacidade ou compartilhamento de recursos. Eles podem ter propósitos individuais ou não, como por exemplo, a Internet.

Atualmente há uma gama gigante de tipos de sistemas distribuídos, como *Clusters*[8], *Grids*[9], P2P, *NFS*, entre outros. Um *Cluster* é um dedicado grupo de computadores

[8] https://en.wikipedia.org/wiki/Data_cluster
[9] https://en.wikipedia.org/wiki/Grid_computing

interconectados que aparecem como um único supercomputador, geralmente utilizado em engenharia científica de alto desempenho e aplicações de negócios. *Grid* é um tipo de sistema distribuído que permite compartilhamento coordenado, comumente usado para ajudar aplicações que estão emergindo da ciência e negócios. Redes *P2P* são sistemas distribuídos descentralizado que possibilitam aplicações de troca de arquivos através da Internet. Por fim, *NFS* são sistemas de arquivos distribuídos que provê ao usuário uma visão unificada dos dados armazenados em diferentes arquivos de sistema e computadores, que podem estar na mesma rede ou não.

Este conceito pode ser confundido com outros conceitos: redes de computadores ou computação paralela (BARNEY, 2000). Uma rede de computadores é um mediador de entidades que provê a troca de mensagens baseada em protocolos conhecidos. Computação Paralela é uma forma de computação em que vários cálculos são realizados simultaneamente.

Historicamente, durante o período de transição entre a primeira e segunda onda computacional definida por WEISER (1999), surgiu o desejo de descentralizar o processamento via programação concorrente, baixo custo de microprocessadores e aos avanços tecnológicos em redes de computadores de alta velocidade, iniciou-se os estudos e trabalhos sobre sistemas distribuídos. Após uma década de estudos científicos, definição de algumas arquiteturas iniciais para sistemas distribuídos, nos anos 90 foi possível começar a desfrutar dos benefícios que estes tipos de sistemas oferecem.

Estado da Arte e tendências futuras para o campo de sistemas distribuídos se une fortemente com a Computação Ubíqua. RAZZAQUE (2016) mostrou como é possível perceber a quantidade imensa de trabalhos voltados para *IoT*.

É necessário ter cuidado sobre as promessas de sistemas distribuídos. Algumas falácias são propagandas através da rede e é importante informar sobre estas falácias para o leitor não cair em armadilhas. PETER DEUTSCH (1980) descreveu oito pontos que são oferecidos, mas que em longo prazo se mostram falsos: (i) a rede é confiável, (ii) não há latência, (iii) a banda, onde os dados são transportados, é infinita, (iv) a rede é segura, (v) a topologia não varia, (vi) só existe um administrador, (vii) o custo para transportar é zero e (viii) a rede é homogênea.

Um sistema distribuído pode existir sobre diferentes aspectos: diferentes tipos de rede, diferentes tipos de hardware (*e.g.* representações de dados), sistemas operacionais distintos, diferentes plataformas de execução ou linguagens de programação discrepantes. Dado que o sistema distribuído se apresenta aos usuários como um sistema único e coerente (TANENBAUM; VAN STEEN, 2001), os desenvolvedores também almejam lidar com um ambiente distribuído como se ele fosse centralizado. Isto é uma característica difícil de lidar, mas para tornar isso possível existe o *middleware*.

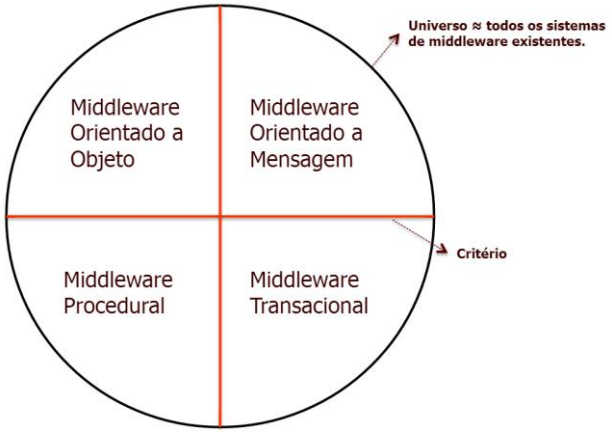
Middleware

O termo *middleware* ganhou popularidade durante a década de 80 como solução para o problema de conectar novas aplicações para sistemas antigos (GALL, 2005). Ele representa a confluência de duas áreas importantes: sistemas distribuídos e engenharia de software. Ela foi inventada na tentativa de simplificar o desenvolvimento de software em sistemas distribuídos e trazer essas capacidades inerentes aos desenvolvedores que possuem pouca expertise (SCHANTZ; SCHIMIDT, 2007). Geralmente, *middleware* tem o propósito de abstrair complexidades de um sistema, permitindo que o desenvolvedor concentre todo seu esforço na tarefa a ser resolvida, sem se distrair com problemas que não são relacionados ao problema propriamente dito (OTHMAN; SCHMIDT, 2001, p.205). A partir da perspectiva da computação, um *middleware* provê uma camada entre a aplicação de software e o sistema de software e tem como objetivo esconder aspectos de distribuição, de forma que a aplicação não tenha que se preocupar com isso (BERNSTEIN, 1996).

Há vários casos de sucesso em *middleware* para sistemas distribuídos: *middlewares* voltados a objetos distribuídos (por exemplo, CORBA e Java RMI), *middleware* como um componente (*e.g.* Java Beans e .NET) ou Web, que permite conectar navegadores com páginas Web que podem ser definidas como portais para sistemas de informarão poderosa.

Existem muitos tipos de *middleware*, entretanto podemos classificar o universo de todos os sistemas de *middleware* existentes em quatro subconjuntos (considerando como critério o tipo de primitiva de comunicação): orientados a objetos, orientados a mensagens, procedural e transacional. Além do tipo de primitiva de comunicação, as características que podem ser elencadas são: aridade da comunicação, (as) síncrono e/ou tipo de protocolo.

Figura 2: Universo dos sistemas de middleware existentes



Fonte: sites.google.com/a/cin.ufpe.br/if711, aula 06, slide 61.

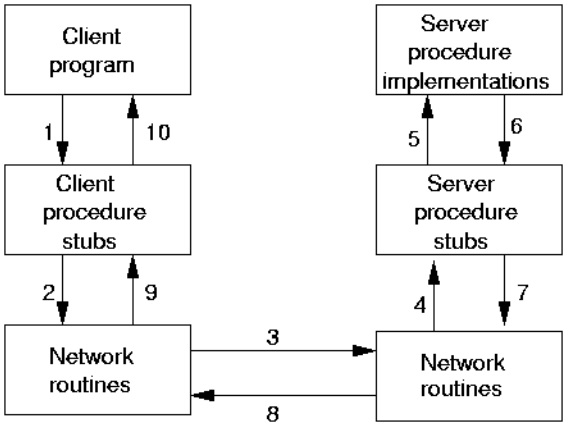
Chamadas Remotas de Procedimento

Neste trabalho, o tipo de utilizado é o *Remote Procedure Calls* (BIRREL; NELSON, 1984), Chamadas Remotas de Procedimento em Português. RPC é um protocolo que um programa pode usar para requisitar serviço de outro programa localizado em outro computador na rede, sem ter nenhum entendimento sobre os detalhes de localização. Uma chamada de procedimento é também conhecida como uma chamada de função ou uma chamada de sub-rotina. Este tipo de protocolo utiliza o *modelo cliente-servidor*, onde um programa (cliente) solicita o serviço em outra máquina (servidor). Esta operação é síncrona, mas o uso *threads*[10] que compartilham o mesmo espaço de endereço permite que RPC seja realizado concorrentemente.

RPC é um tipo de protocolo *requisição-resposta*[11]. Ele é iniciado por um cliente que envia uma requisição em formato de mensagem para um servidor remoto. Lá é executada, a partir de um procedimento já conhecido entre as partes, a requisição com os argumentos. O servidor remoto envia para o cliente a resposta do que foi processado. Enquanto o servidor está processando o conteúdo, o cliente fica bloqueado esperando o resultado (enquanto o servidor executa o procedimento e retorna o valor), a não ser que este cliente envie uma requisição assíncrona ao servidor, como uma chamada XHTTP[12].

[10] https://en.wikipedia.org/wiki/Thread_(computing)
[11] https://en.wikipedia.org/wiki/Request%E2%80%93response
[12] http://www.xhttp.org/

Figura 3: Diagrama para o fluxo de comunicação em uma Chamada Remota de Procedimento



Fonte: Jan Newmarch, Remote Procedure Call, cap. 13.

Há dois tipos, popularmente conhecidos de RPC: (i) são aqueles que puramente usam a abordagem, no qual o servidor provê somente operações específicas para o cliente e (ii) que usam o conceito de orientação a objetos, onde um servidor hospeda um conjunto de objetos remotos que possibilitam operações para o cliente como parte da sua interface pública.

Os mecanismos internos são semelhantes para ambos os casos. No entanto, para o caso de sistemas que usam o conceito de orientação a objetos, é possível trabalhar com a noção de "identificar o objeto", significando que o cliente pode solicitar objetos separadamente.

Erlang

Erlang foi criada com o propósito de resolver problemas em empresas de telefonia. Logo, ela foi desenhada para escrever programar concorrentes que "rodam para sempre" (ARMSTRONG, 1997). Erlang usa processos concorrentes para estruturar os programas. Estes processos não possuem memória compartilhada e se comunicam através de trocas de mensagens assíncronas. A partir de 1993, foi adicionado o conceito de distribuição para a linguagem, no qual torna possível rodar um sistema Erlang homogêneo em *hardware* heterogêneo. Por fim, Erlang também possui um mecanismo para alterar código sem que seja necessário parar o programa, conhecido por *on the fly*.

A primeira versão de Erlang foi desenvolvida em 1986 nos laboratórios de ciência da computação da Ericsson[13]. A linguagem foi moldada com o seguinte objetivo em mente: “prover uma forma melhor de desenvolver aplicações para o meio telefônico” (ARMSTRONG, 2007, p. 3). Naquela época, criar aplicações para este meio era algo atípico porque este ambiente em concorrente por natureza. Além do mais, os programas deveriam operar em domínio de tempo real.

Esta linguagem foi construída sobre o conceito de que tudo é um processo. O meio de comunicação que Erlang suporta é a troca de mensagens entre processos via *Pid*[14] (abreviação para Identificador de Processo em inglês). Logo, não há a característica de memória compartilhada, diminuindo bastante os problemas no universo concorrente. Além do mais, a partir do exemplo abaixo é possível identificar que a linguagem é expressiva e concisa.

echo_server.erl

```
-module(echo_server).
-export([start/0, loop/1]).


start() ->
        socket_server:start(?MODULE, 7000, {?MODULE, loop}).
loop(Socket) ->
    case gen_tcp:recv(Socket, 0) of
        {ok, Data} ->
            gen_tcp:send(Socket, Data),
            loop(Socket);
        {error, closed} ->
            ok
    end.
```

Erlang é utilizando em empresas de sucesso no mercado atual, como o *WhatsApp*[15]. Em nível de informação, em 2011 a empresa conseguiu atingir a marca de um milhão de conexões *TCP* num único servidor, e com memória e *CPU*[16] sobrando [23].

[13] https://www.ericsson.com/
[14] http://stackoverflow.com/questions/243363/can-someone-explain-the-structure-of-a-pid-in-erlang
[15] https://www.whatsapp.com/

Paradigma Funcional

Erlang dá suporte ao paradigma funcional. Programar neste paradigma significa que o desenvolvimento é baseado no conceito matemático de função, em que para elemento do seu conjunto domínio (entrada) há apenas um elemento no seu conjunto contradomínio (saída). Além disso, as funções são normalmente expressas por meio de outras funções – de modo que obter o valor da função para um determinado conjunto de parâmetros envolve não só aplicar as regras daquela função, mas também fazer uso de outras funções (Erlang permite isto via *Funs*[17]).

Por essa razão, a programação funcional não possui o conceito de "variáveis de estado"; se um valor qualquer, independente do tipo, aparece num momento qualquer da computação, considera-se que aquele valor é único e imutável durante todo o processo.

Em uma linguagem funcional pura, como Haskell[18], maiorias das funções são garantidas pelo sistema de tipo para não executar outras ações. Em uma linguagem funcional impura, como Erlang, uma função pode ter outros efeitos colaterais, como consultar um banco de dados ou servidor, ler ou gravar em disco, entre outras coisas. Uma das principais desvantagens de programação funcional é pela falta de efeitos colaterais no sistema. É muito difícil escrever *software* válido sem *IO*. Muitas linguagens de programação modernas possuem elementos de linguagem de programação funcional. Swift[19] e C#[20], por exemplo, tem recursos de programação funcional.

Trabalhos Relacionados

Com o propósito de entender o problema do desenvolvimento e validação do *middleware* baseado em chamadas remotas de procedimento, foram analisadas algumas abordagens já presentes no meio acadêmico.

BORGES (2016) desenvolveu um *middleware* do mesmo perfil, mas usando a linguagem de programação Elixir (VALIM, 2012). Esta linguagem é executada pela máquina

[16] https://en.wikipedia.org/wiki/Central_processing_unit
[17] http://erlang.org/documentation/doc-5.2/doc/extensions/funs.html
[18] https://www.haskell.org/
[19] https://swift.org/
[20] https://msdn.microsoft.com/en-us/library/kx37x362.aspx

virtual de Erlang, conhecida por BEAM[21], que se situa entre sistema operacional e a aplicação em Erlang. Por isso, Elixir herda facilidades providas por Erlang, como processos *lightweight* e foco em aplicações concorrentes e tolerantes a falhas. Elixir é usado na indústria por companhias de sucesso, como por exemplo, *Pinterest*[22] e *Moz*[23]. Além disso, é possível desenvolver aplicações voltadas para Web e para sistemas embarcados utilizando Elixir.

Este trabalho serve como um norte para as etapas de arquitetura e avaliação do *middleware* aqui proposto. Também é possível observar como as tomadas de decisão, na etapa de desenvolvimento, influenciaram bastante os resultados obtidos durante as baterias de teste de desempenho.

[21] http://stackoverflow.com/questions/16779162/what-kind-of-virtual-machine-is-beam-the-erlang-vm
[22] https://br.pinterest.com/
[23] https://moz.com/

Implementação

Neste capítulo serão tratados os aspectos de requisitos, arquitetura e implementação do *middleware*. Ao final deste capítulo, serão apresentadas as considerações finais.

Requisitos

Todas as transparências "comuns", exceto transparência de tecnologia, apontadas em [25] foram adicionadas ao projeto deste trabalho. Além disto, foram incluídas algumas *features* ao *middleware* como recurso extra.

Seguem os requisitos funcionais:

- **RF-01**: transparência de acesso. As chamadas remotas de procedimento devem ser realizadas de maneira semelhante às invocações locais. Logo, acesso local e remoto é programado de forma similar. A programação do cliente não muda caso o serviço esteja executando local ou remotamente.
- **RF-02**: transparência de localização. O cliente não precisará saber informações quanto à localização dos servidores. Para permitir isso, o cliente conhecerá de antemão a localização de um serviço de nomes que é capaz de informar onde o servidor com a função remota desejada está situado.
- **RF-03**: serialização das mensagens. Os dados serão transformados em *byte streams*[24], para que em seguida, sejam conduzidos pela rede. O cliente não precisa se atentar ao fato de traduzir os dados para um formato viável de ser transferido pela rede.
- **RF-04**: segurança de informação. O solicitante não tem a obrigação de garantir segurança dos dados através da Internet porque o *middleware* tratará de criptografar os dados antes da etapa de serialização.

[24] https://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html

Seguem os requisitos não funcionais:

- **RNF-01**: transparência de concorrência. Componentes são compartilhados sem que isto afete os resultados produzidos. Logo, o servidor pode suportar múltiplos clientes, e de forma escalável, sem que isso afete o desempenho do sistema.
- **RNF-02**: Padrões de projetos definidos em (Völter et al. 2004). Estes padrões servirão como guia para a modularização do *middleware*.
- **RNF-03**: transparência de falha: O sistema não para se algum componente falhar, nem o cliente percebe a ocorrência do problema. Portanto, as falhas serão escondidas e os componentes que caíram serão reestabelecidos.
- **RNF-04**: Homogeneidade. Todos os componentes serão desenvolvidos na mesma tecnologia por questões de praticidade. No caso, a tecnologia em questão é Erlang, como já foi destacado em momentos anteriores.

Arquitetura

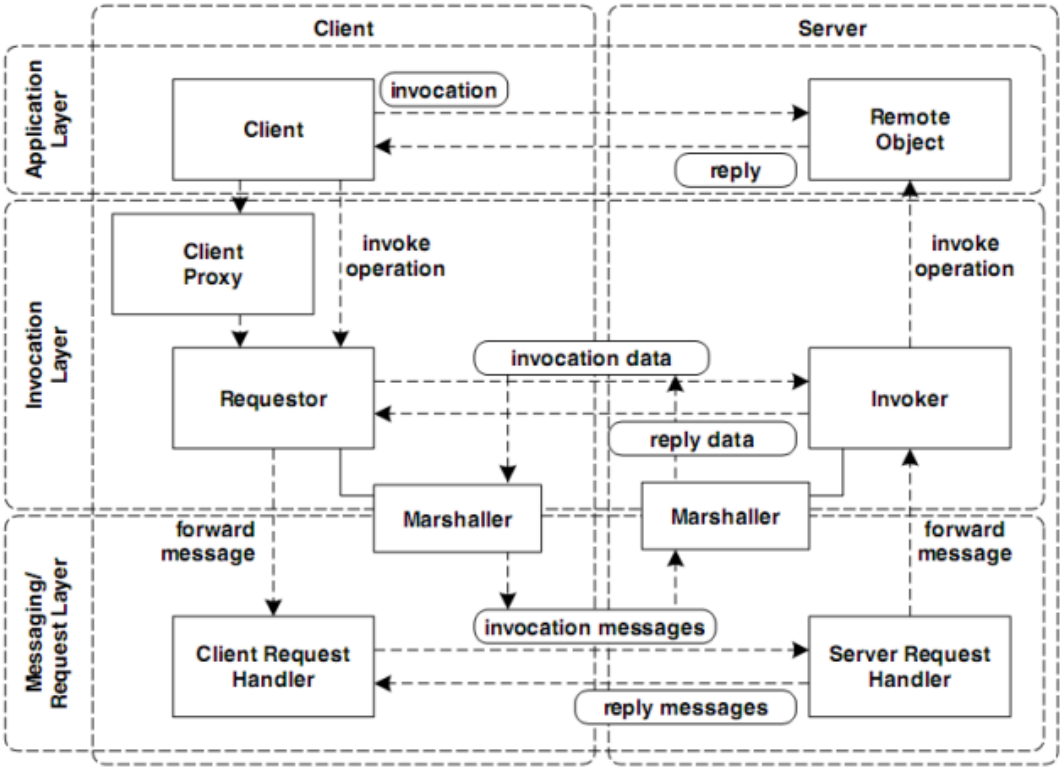
A arquitetura do *middleware* será baseada na **RNF-02**. O fluxo de relacionamento dos padrões neste requisito não funcional são bem definidos e ajudam na modularização e separação lógica dos componentes que englobam a maior parte dos aspectos em sistemas de *middleware*.

À priori, todos os padrões remotos definidos em (VÖLTER; KIRCHER; ZDUN, 2004), mostrados na figura 4, foram programados, mais um padrão - módulo em Erlang – para criptografar os dados por opção do autor. Ademais, pela natureza do paradigma utilizado, o objeto remoto deu lugar à função remota.

Ainda sobre a figura 4, é possível observar compreende as camadas de distribuição (*invocation layer*) e infraestrutura (*messaging layer*). Para a parte de distribuição, etapa em que o *middleware* se comunica diretamente com cliente e servidor (*application layer*), é garantido às transparências de acesso, tornando invocação local semelhante à invocação remota, e localização, tratando de encontrar a localização dos servidores via serviço de nomes. Para a camada de infraestrutura, serão gerenciadas as conexões *TCP*[25] dialogando diretamente com as ferramentas disponibilizadas pelo SO.

[25] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

Figura 4: Arquitetura dos padrões remotos

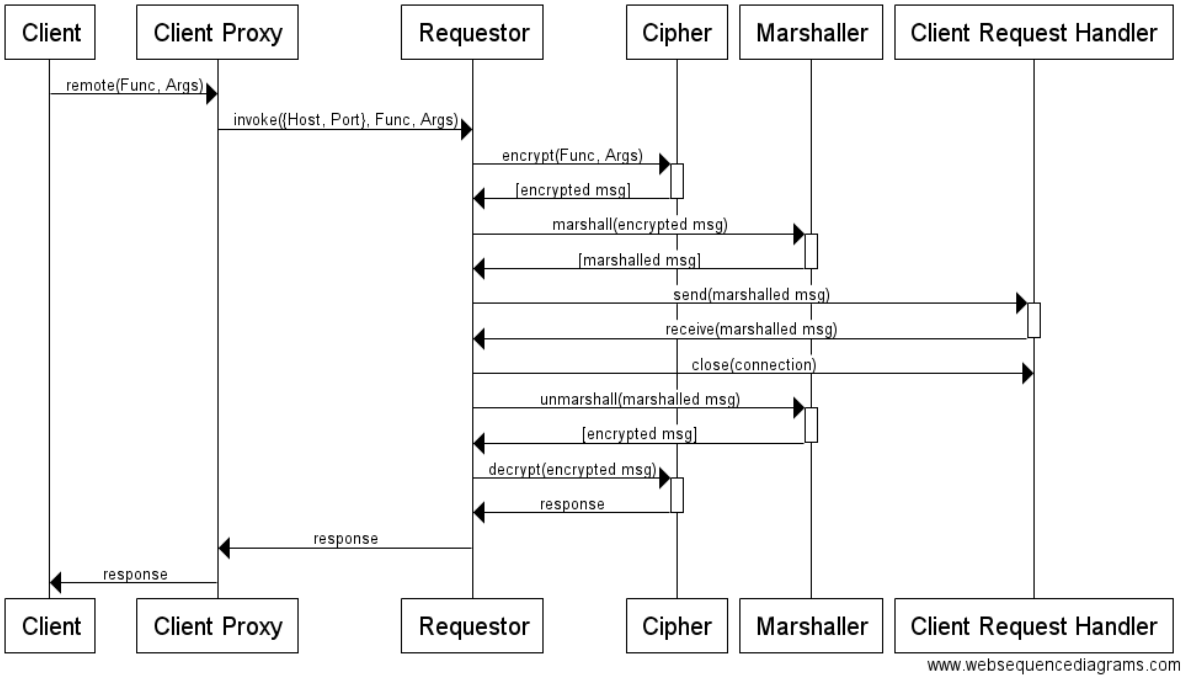


Fonte: VÖLTER; KIRCHER; ZDUN, 2004, p.66.

A arquitetura deste projeto ainda conta com um serviço de nomes do tipo *Key-Value Server (KVS*[26]*)*. Um serviço deste tipo é bastante simples porque armazena os dados em um banco de dados que usa um dicionário como modelo de estrutura de dados fundamental, onde cada chave (*key*) é associada a uma, e somente uma, coleção (*value*). Esta relação é referida como um par. Em Erlang, cada processo possui um dicionário local pré-estabelecido[27], que é gerado no mesmo momento que o processo é criado. Em cada par, chave e valor são representados por uma sequência arbitrária.

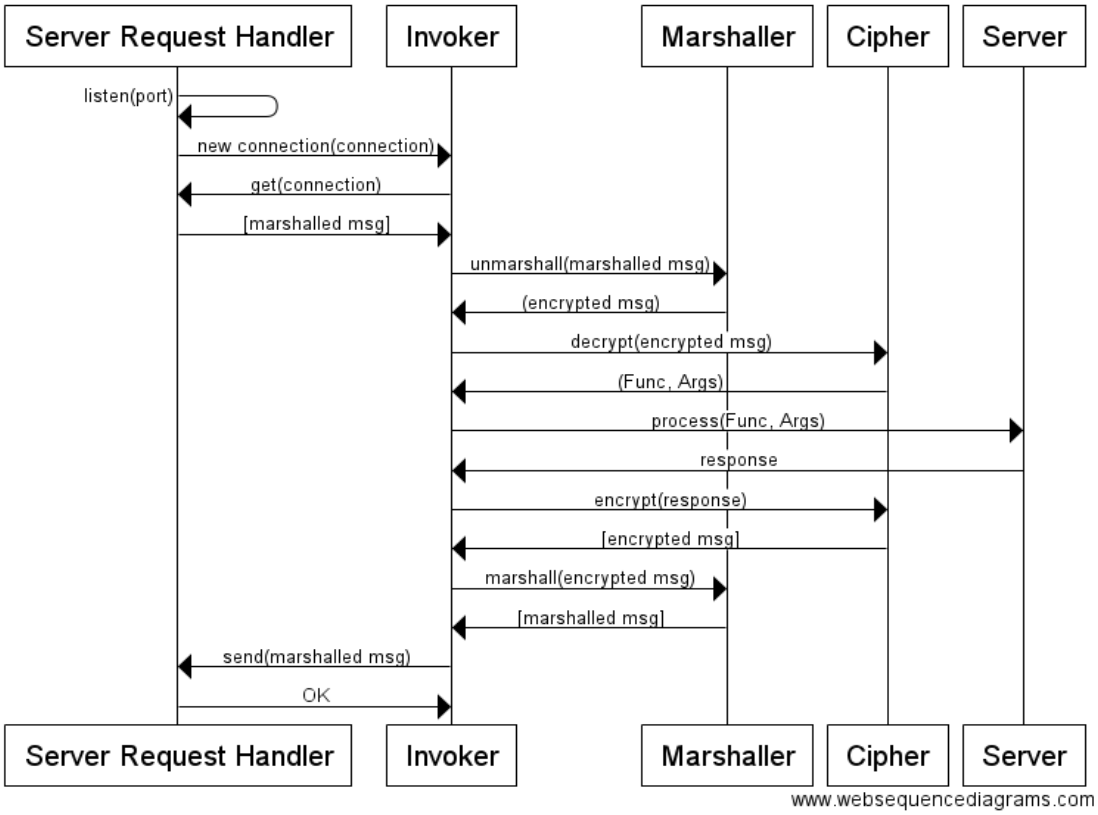
[26] http://www.aerospike.com/what-is-a-key-value-store/
[27] http://www.erlang.org/course/advanced#dict

Figura 5: Diagrama de sequência no lado cliente



Fonte: Elaborado pelo autor.

Figura 6: Diagrama de sequência no lado servidor



Fonte: Elaborado pelo autor.

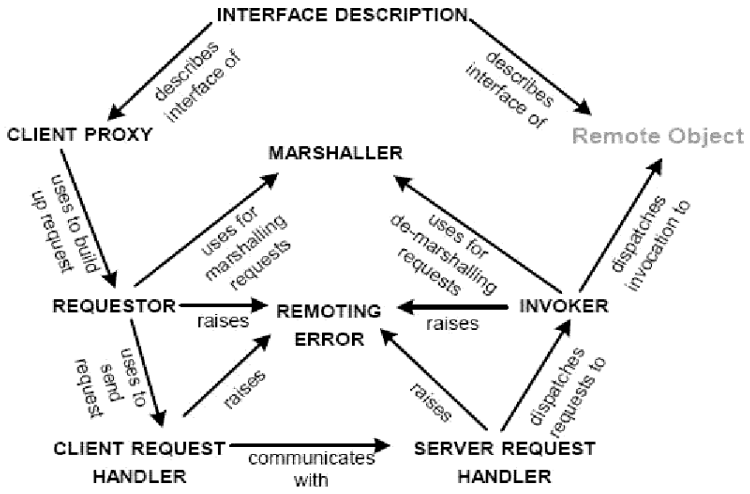
A partir dos diagramas de sequência das Figuras 5 e 6, é possível entender a interação entre os componentes do sistema, tanto no lado do cliente quanto no lado do servidor. Este sistema de *middleware* provê uma infraestrutura para clientes se comunicarem com funções em servidores remotos. Funções remotas são programadas no servidor. O cliente invoca uma operação de uma função local e espera que o conteúdo siga para a função remota.

Para propiciar isto, a invocação precisa atravessar os limites da máquina. O REQUESTOR constrói a invocação remota no lado do cliente a partir dos parâmetros (localização, função e argumentos). O cliente pode se comunicar diretamente com o REQUESTOR ou usar o CLIENT PROXY. Este componente, se o *middleware* fosse orientado a objetos, deveria possuir uma mesma interface que o objeto remoto, mas já que o paradigma é o funcional, o CLIENT PROXY apenas descobre a localização do servidor que contém a função remota, junta este dado ao que foi recebido pelo cliente e repassa ao REQUESTOR. Internamente, o CLIENT PROXY utiliza o REQUESTOR para construir invocações remotas.

No lado cliente, o REQUESTOR manuseia o CLIENT REQUEST HANDLER para gerenciar a comunicação na rede. No lado do servidor, a invocação remota é recebida pelo SERVER REQUEST HANDLER. Este padrão lida com a mensagem de forma eficiente e escalável repassando o conteúdo para o INVOKER, dado que a mensagem chegou totalmente. O INVOKER despacha a chamada remota para o servidor usando a informação conhecida no conteúdo da mensagem.

Os parâmetros passados entre o cliente e servidor são serializados e deserializados usando o MARSHALLER. Além disso, antes de serializar as informações, o conteúdo é criptografado pelo componente CIPHER. Se o cliente tiver algum problema enquanto se comunica com o servidor ou se o servidor tiver problemas internos, este fato é repassado ao CLIENT REQUEST HANDLER e sinalizado ao cliente usando um tipo de erro remoto. Todo esse fluxo é apresentado na figura abaixo.

Figura 7: Relacionamento dos padrões remotos



Fonte: VÖLTER; KIRCHER; ZDUN, 2004, p.31.

Segue abaixo a explicação detalhada da atuação de cada componente:

- **CLIENT PROXY**: cliente interage diretamente com middleware a partir deste componente. O propósito do CLIENT PROXY é permitir que ocorra [**RF-01**]. O CLIENT PROXY usa o REQUESTOR para construir e enviar invocações através da Internet, coletando via serviço de nomes a localização (endereço IP e porta) por onde o servidor está escutando as invocações remotas.
- **REQUESTOR**: este componente tem como objetivo coordenar o envio das chamadas sobre a rede até o INVOKER. Primeiramente, ele repassa parte do conteúdo (função e argumentos) para ser criptografado pelo CIPHER, adicionando uma camada de segurança aos dados. Em seguida, o REQUESTOR pede ao MARSHALLER que os dados, já criptografados, sejam serializados. Após esta etapa, o REQUESTOR solicita ao CLIENT REQUEST HANDLER para que abra uma conexão a partir da localização, que foi informada previamente pelo CLIENT PROXY. Por fim, depois de receber a resposta, o REQUESTOR solicita ao MARSHALLER que ele deserialize o pacote, pede ao CIPHER que descriptografe o conteúdo. Com a resposta em mãos, o REQUESTOR repassa o resultado para o CLIENT PROXY, e este último repassa ao Cliente.
- **INVOKER**: o INVOKER recebe a requisição, vinda do lado cliente pelo REQUESTOR através dos *Request Handlers*, envia ao MARSHALLER para que ele deserialize os

dados. Com o retorno, solicita ao CIPHER que descriptografe a requisição. Depois, envia o conteúdo (função e argumentos) ao servidor, para que este processe os dados. O servidor retorna o resultado ao INVOKER, que criptografa e serializa o resultado a partir do CIPHER e MARSHALLER, respectivamente. Com o pacote serializado, ele repassa ao SERVER REQUEST HANDLER, que envia ao CLIENT REQUEST HANDLER.

- **CIPHER**: este componente é uma dos módulos adicionais para aumentar a segurança, que não está presente na arquitetura pensada por [VÖLTER; KIRCHER; ZDUN, 2004]. REQUESTOR e INVOKER utilizam este componente para criptografar/descriptografar informação.
- **MARSHALLER**: requisições e respostas precisam ser transportadas através da Internet pelo canal criado entre *Request Handlers*. Este componente é responsável por serializar/deserializar as mensagens que serão trocadas. O cliente invoca uma operação via função e seus argumentos. O REQUESTOR utiliza o MARSHALLER para transformar o conteúdo em *byte streams*, tipo de informação que trafega na rede.
- **CLIENT REQUEST HANDLER**: o CLIENT REQUEST HANDLER é responsável por gerenciar a comunicação pela rede dentro da aplicação do cliente. Quando um cliente solicita uma requisição usando um REQUESTOR, o CLIENT REQUEST HANDLER estabelece uma conexão *TCP*, envia o pedido e espera pela resposta, trazida pelo SERVER REQUEST HANDLER. Após isso, retorna o resultado ao REQUESTOR e fecha a conexão.
- **SERVER REQUEST HANDLER**: as mensagens recebidas pelo SERVER REQUEST HANDLER, no lado do servidor, usando API's providas pelo Sistema Operacional, são despachadas ao INVOKER e fica esperando a resposta, que é uma mensagem serializada. Em seguida, o conteúdo é enviado de volta para o CLIENT REQUEST HANDLER.

O serviço de nomes, que segue o padrão remoto LOOKUP [2] tem o objetivo de prover transparência de localização ao *middleware*. Este serviço roda em um processo separado, onde possui um dicionário que associa uma chave (função) a um, e somente um, valor (tupla com a seguinte configuração: {*Host*, *Port*}). A comunicação deste serviço é

baseada no conceito primitivo de processos em Erlang, onde cada processo possui um *Pid* (abreviação para "identificador de processo" em inglês). Este identificador é utilizado para o servidor de nomes se comunicarem com as requisições que chegam do middleware, a partir do CLIENT PROXY.

O servidor, ao ser inicializado, envia ao serviço de nomes, a partir da função `bind()`, uma tupla da seguinte forma: {*Function*, {*Host*, *Port*}}. Também é possível que o servidor remova uma dada função previamente armazenada por ele no serviço de nomes utilizando a função `unbind()`.

Desta forma, o cliente não será incomodado com a problemática de descobrir a posição do servidor que contém a função remota desejada. No entanto, este cliente precisa saber a localização do serviço de nomes. Ele pede ao serviço, via função `lookup()` desenvolvida no serviço de nomes, a localização do servidor passando a função como argumento.

Implementação

O código produzido em Erlang é dividido em módulos. Quando se cria um módulo, é possível declarar dois tipos de coisas: funções e atributos. Atributos são *metadados* descrevendo o módulo, as funções que o autor declara como interface pública ou outras coisas. Normalmente, um módulo agrupa funções de mesmo objetivo. Logo, cada padrão será um módulo em Erlang (o MARSHALLER será um módulo que contém duas funções, `marshall/1` e `unmarshall/1`). Segue abaixo como ficará o MARSHALLER mapeado em Erlang:

marshaller.erl

```
-module(marshaller).
-export([marshall/1, unmarshall/1]).
marshall(Msg) -> term_to_binary(Msg).
unmarshall(Msg) -> binary_to_term(Msg).
```

Sobre o trecho de código acima, analisando somente os aspectos inerentes a linguagem. Esta é a estrutura básica de um módulo em Erlang, que possui, pelo menos em todos os módulos, os atributos `module` e `export` (interface que torna a função pública) e

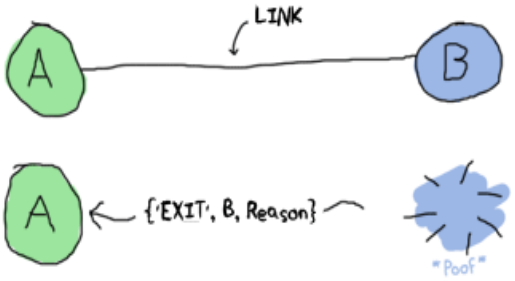
as funções que representam este módulo. A numeração ao lado da assinatura da função (`marshall/1`) indica a aridade (quantos argumentos a função recebe como entrada) da função.

Os padrões remotos definidos em (Völter et al. 2004) foram originalmente idealizado para um middleware orientado a objetos. Pela lógica, o esperado seria usar uma tecnologia que tivesse suporte ao paradigma orientado a objetos. Porém, Erlang dá suporte somente aos paradigmas concorrente e funcional. Logo, foi necessário esquematizar o *middleware* com esses requisitos.

Tolerância a falhas é um conceito maleável, depende diretamente da estratégia, da linguagem. Em alguns casos, ser tolerante a falhas significa "nenhuma informação é perdida, nenhuma tarefa falha", em outros casos significa que "o usuário nunca vê a falha ocorrer". Em Erlang, o conceito geral significa manter o sistema funcionando. Como exemplo dado em por Joe Armstrong em sua tese de doutorado: "*tudo bem se a chamada de um usuário cair, ao invés de cair todas as chamadas de todos os usuários*" [28]. Logo, a ideia de permitir um nó cair, mas manter todo o restante do sistema funcionando é visto com bons olhos em sistemas em Erlang. Portanto, ser tolerante a falhas neste *middleware* significa que o componente vai ser reestabelecido e os outros componentes não vão descobrir que o erro ocorreu.

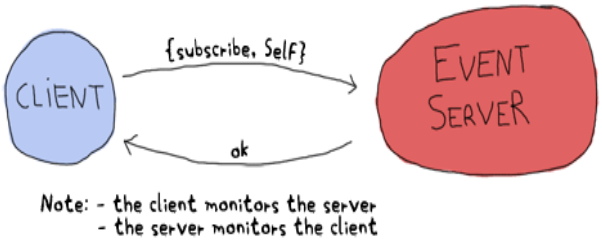
Para dar suporte ao [**RNF-03**] foi preciso criar um módulo adicional para monitorar os componentes desejados. Erlang permite duas formas para notificar quando um processo morre: links e monitores. Links permitem uma ligação de duplo sentido entre dois processos, caso um dos processos morra, o outro processo é notificado sobre o ocorrido. Monitores permite que, dado um processo A, se B for um monitor dele, se A morrer, o processo B será notificado. Entretanto, nada será feito se o processo B morrer [29].

Figura 8: Fluxo de funcionando de notificação usando Links



Fonte: HÉBERT, 2013, Error and Exceptions.

Figura 9: Fluxo de funcionando de notificação usando Monitors



Fonte: HÉBERT, 2013, Error and Exceptions.

Abaixo segue a descrição de cada componente desenvolvido:

- **MARSHALLER**: a implementação deste componente foi trivial. Já existem na biblioteca padrão de Erlang funções para transformação de dados em *byte streams*. Logo, o MARSHALLER é uma interface para as funções `binary_to_term/1`[28] e `term_to_binary/1`[29]. Além disso, pelo o *middleware* ser homogêneo, não houve necessidade de criar um tipo de dados intermediário para que o sistema se comunique com outras tecnologias.
- **CIPHER**: Assim como o MARSHALLER, este componente também é uma interface para funções já desenvolvidas em Erlang. Neste caso, `public_encrypt/4`[30] e `public_decrypt/4`[31].

[28] http://erlang.org/doc/man/erlang.html#binary_to_term-1
[29] http://erlang.org/doc/man/erlang.html#term_to_binary-1
[30] http://erlang.org/doc/man/crypto.html#public_encrypt-4
[31] http://erlang.org/doc/man/crypto.html#public_decrypt-4

- **MONITOR**: Este é um componente adicional que provê tolerância a falhas ao sistema a partir das funções `process_flag/2`[32] e `link/1`[33]. Baseado no conceito de monitores, previamente explicado, este componente é criado em conjunto com o serviço de nomes. O dever dele é observar este serviço e, caso ele caia, o componente levanta uma nova versão deste serviço com as informações com o estado anterior.
- **CLIENT REQUEST HANDLER**: Aqui, é necessário gerenciar conexões enviando a requisição até o servidor que contém a função. Após receber o resultado, ele retorna o conteúdo ao REQUESTOR e fecha o socket. O módulo `gen_tcp`[34] foi utilizado porque já possui uma interface para os sockets *TCP/IP*, necessários para executar essa tarefa.

Figura 10: interface para sockets TCP/IP

```
do_recv(Sock, Bs) ->
    case gen_tcp:recv(Sock, 0) of
        {ok, B} ->
            do_recv(Sock, [Bs, B]);
        {error, closed}  >
            {ok, list_to_binary(Bs)}
    end.
```

Fonte: ERLANG MANUAL, gen_tcp.

- **SERVER REQUEST HANDLER**: este componente foi adaptado da versão sequencial `gen_tcp:recv`[35], adicionando capacidade de trabalhar concorrentemente. Logo, a cada nova conexão, o processo atual do SERVER REQUEST HANDLER, cria um segundo processo para escutar no mesmo canal e, após executar a requisição vinda do CLIENT REQUEST HANDLER, este processo morre. Enfim, se houver um milhão de clientes solicitando este componente, será criado um milhão de versões deste componente para que cada um funcione 1:1.

[32] http://erlang.org/doc/man/erlang.html#process_flag-2
[33] http://erlang.org/doc/man/erlang.html#link-1
[34] http://erlang.org/doc/man/gen_tcp.html
[35] http://erlang.org/doc/man/gen_tcp.html#recv-2

- **INVOKER**: o INVOKER solicita ao componente anterior que abra uma conexão em uma determinada porta (informada pelo servidor) e fique escutando novas conexões. Quando uma nova conexão chega, este componente se comunica com o MARSHALLER e o CIPHER para deserializar e descriptografar, respectivamente. Em seguida, repassa a requisição ao servidor, para que este processe a informação. No retorno, ele requisita aos mesmos componentes que façam a tarefa inversa. Assim, o conteúdo poderá ser enviado pela rede.
- **CLIENT PROXY**: dos elementos do *middleware*, este fica com a tarefa de se comunicar com o serviço de nomes para que o REQUESTOR possa abrir a conexão corretamente, além de que ele é a interface do *middleware* com a aplicação do cliente. Em OO, o CLIENT PROXY possui a interface do objeto remoto no lado do cliente. Assim, cada cliente possui uma versão deste componente. Isso não acontece aqui porque, pelo fato do paradigma ser funcional, a função é genérica.
- **REQUESTOR**: este elemento tem o papel de construir a mensagem codificada e, com a ajuda do CLIENT REQUEST HANDLER, envia a requisição através da Internet para a localização esperada.

Esta etapa especificou os componentes presentes no middleware, definindo os requisitos, a arquitetura e os detalhes de implementação. De posse disto, é possível perceber que é viável desenvolver middleware usando linguagens com suporte ao paradigma funcional. Os recursos presentes em Erlang para tratamento de erros e concorrência foram primordiais. Em seguida haverá uma avaliação, tanto sobre a capacidade de paralelismo quanto da tolerância a falhas.

Análise

Neste capítulo será realizada uma análise sobre a capacidade de desempenho do middleware e o requisito [**RNF-03**]. O tempo de resposta das chamadas remotas de procedimento por vários clientes seja concorrente ou não, será comparado. Além disso, é possível replicar todos os experimentos realizados neste trabalho seguindo as instruções presentes no Guia do Usuário.

Para executar a bateria de experimentos, foram utilizadas duas máquinas: MacBook Pro macOS 10.12 (2.8 GHz Intel Core i5 com 4 núcleos, 500 GB SSD e 8 GB 1600 MHz DDR3) interpretando o papel de cliente e um desktop Ubuntu 16.04 (i5-4590 3.3 GHz com 4 núcleos, 1 TB SSD e 16 GB 1600 MHz DDR3) atuando como servidor. Obviamente, a máquina com maior poder de processamento será o servidor porque ela terá a obrigação de lidar com as requisições e o gerenciamento de conexões simultâneas. Para a etapa de tolerância a falhas, foi usado o computador que servirá de cliente no ciclo de paralelismo por escolha do autor. É importante frisar que cada medição de tempo nos experimentos foi realizada cinco vezes, e somente a média foi reportada no relatório.

Método de análise

Antes de executar cada bateria de testes, foi realizada uma invocação remota para que o servidor, nas etapas seguintes, tivesse a função avaliada na *cache*[36]. A ferramenta de comparação foi à combinação de duas funções: `erlang:statistics`[37] e `timer:tc`[38]. Ambas se complementem para fornecer dados sobre o estado do sistema, tempo total de decorrido para entre requisição e retorno de função, entre outras coisas.

O método de avaliação usado neste trabalho se aproxima da proposta apresentada por (JAIN; 1991, p.26). Alguns dos pontos não puderam ser avaliados por questões de falta de disponibilidade, como por exemplo, avaliar a velocidade de rede, que propõe verificar em duas distâncias (curta, na rede local; longa, atravessando o país) para invocações remotas. Dentre alguns traços do *middleware* a serem analisados, estão tolerância a falhas e paralelismo. Logo, os métodos de avaliação levaram em conta esses dois fatores.

[36] https://en.wikipedia.org/wiki/Cache_(computing)
[37] http://erlang.org/doc/man/erlang.html#statistics-1
[38] http://erlang.org/doc/man/timer.html#tc-1

Sobre o teste voltado ao requisite de tolerância a falhas, o foco para permitir este tipo de requisite ficou restrito ao serviço de nomes. Este componente precisa estar sempre disponível porque mantém a ponte de comunicação entre cliente e servidor a partir do momento que ele é quem informa a localização da função remota ao *middleware*.

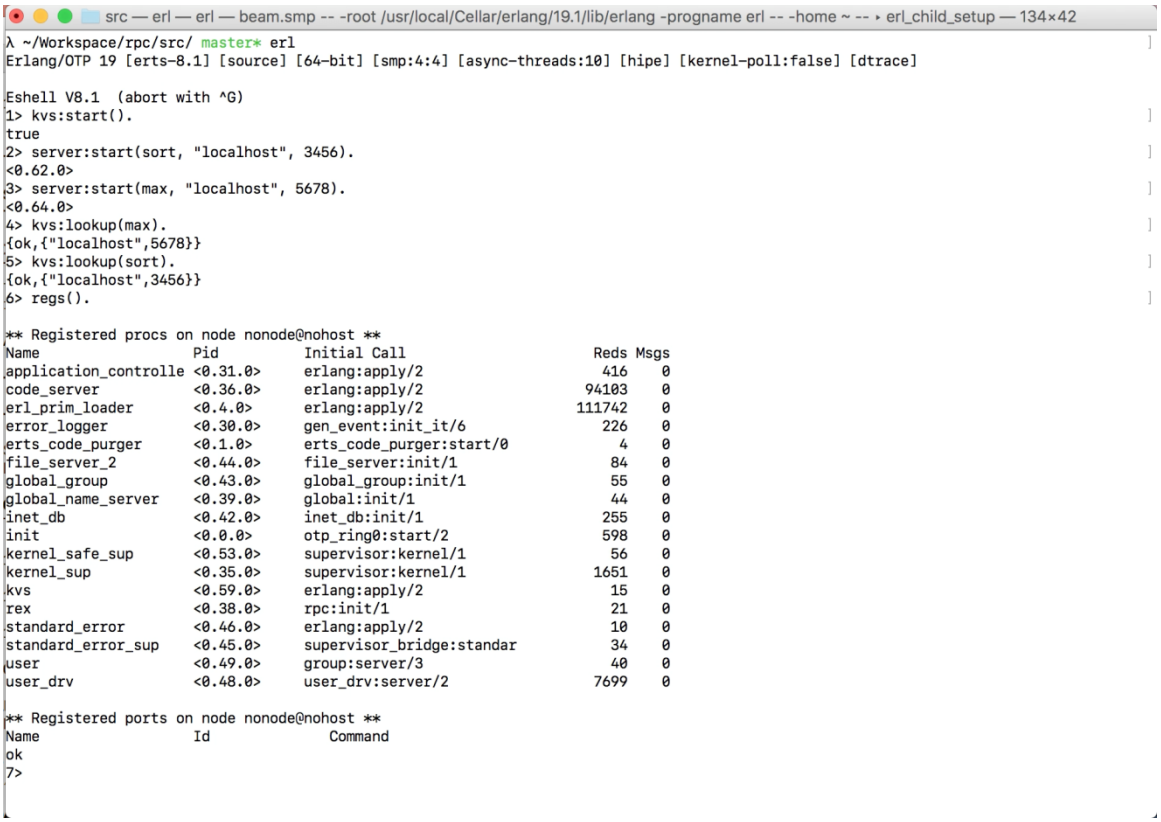
Para avaliar a capacidade de paralelismo, tanto no middleware quanto nas invocações remotas, foi utilizada uma variação de função de ordenação conhecida: `mergesort`[39] (há uma versão *single-threaded* e outra multi-processos). Em um momento será possível avaliar a desempenho do *middleware* com o aumento de clientes, determinando a escalabilidade do mesmo. Em seguida, para as invocações remotas, o `mergesort` que trabalha com múltiplos processos avaliará a capacidade do servidor. Em nível de informação, a multiplicidade dos clientes é dada a partir da criação de múltiplos processos. A métrica usada é um tempo gasto para o *middleware* devolver uma requisição.

Tolerância a falhas

Após configurar o ambiente utilizando o Guia do Usuário, abre sua aplicação de linha de comandos e navegue até a pasta `/src` do projeto. Rode `` `erl` `` para iniciar o *Erlang runtime system*[40]. Caso usuários do sistema operacional *Windows*[41] tenha algum problema, tente `` `werl` `` ou acesse a documentação do *ERTS*. Abaixo segue imagens do processo de execução:

[39] https://en.wikipedia.org/wiki/Merge_sort
[40] http://erlang.org/documentation/doc-4.8.2/doc/system_architecture_intro/sys_arch_intro.html
[41] https://www.microsoft.com/en-us/windows/

Figura 11: Fluxo para testes de tolerância a falhas (etapa um)



Fonte: Elaborado pelo autor.

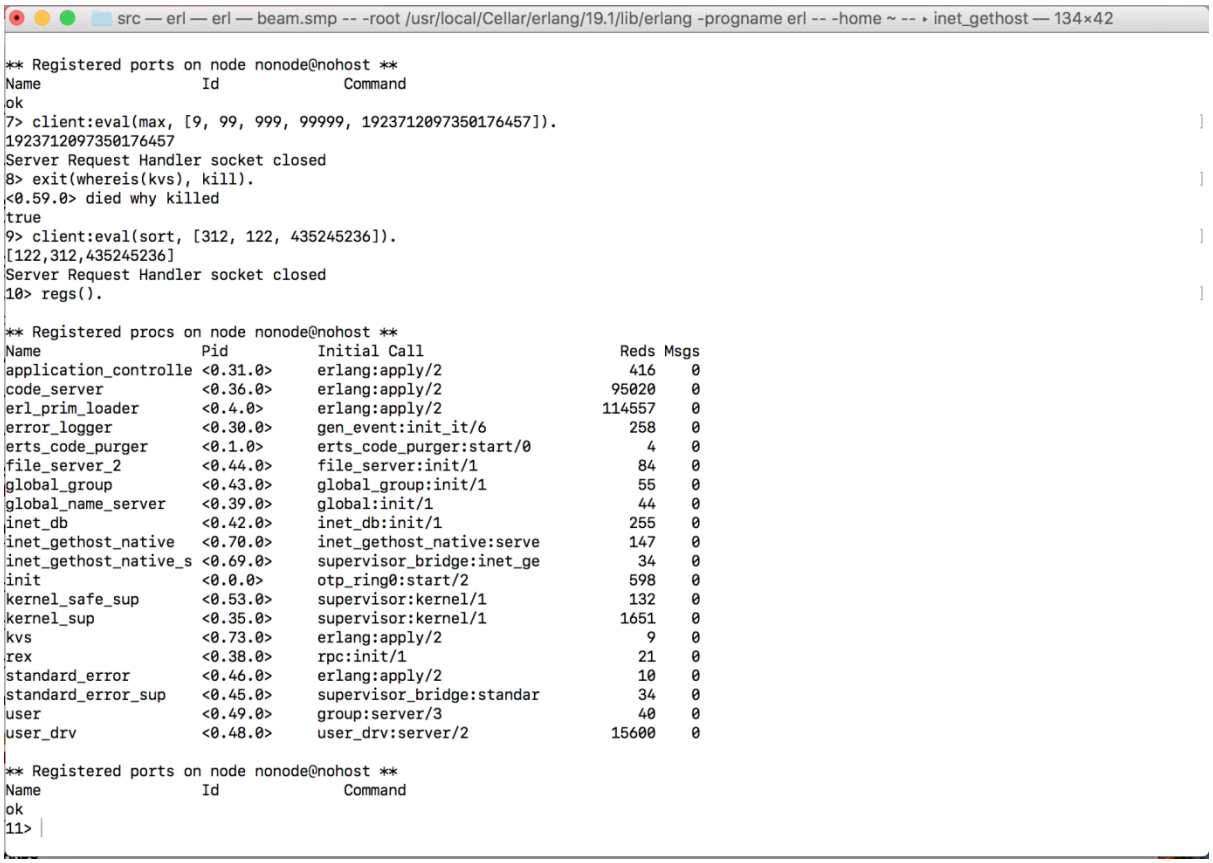
Todo o processo pode ser observado pelas figuras 4.1 e 4.2. Cadastraremos alguns servidores no serviço de nomes responsável por manter a localização das funções remotas. O processo será eliminado e, em seguida, poderá ser observado que o serviço ainda estará disponível para consulta e cadastro.

Na linha um, iniciaremos um novo processo do serviço de nomes a partir da função `start()`. Neste caso, este processo está sendo registrado no processo *shell* criado pelo *ERTS*. Em seguida, a partir da função `start()` do módulo `server`, iniciaremos dois processos de servidores são criados: o servidor que executa a função `sort`, que escuta na porta 3456 e possui o *Pid* igual a `<0.62.0>` e outro servidor que executa a função `max`, escutando na porta 5678 e de *Pid* equivalente a `<0.64.0>` . Apenas para mostrar que o serviço de nomes está funcionando, na quarta e quinta etapas da execução, foi executado a função `lookup()` do componente que retorna uma tupla contendo localização da função solicitada. Depois, foi executada a função `regs()`[42], já presente na biblioteca padrão de

[42] http://erlang.org/doc/man/c.html#regs-0

Erlang que imprime as informações sobre todos os processos registrados. Observe que é possível conferir a identificação do serviço de nomes, `<0.59.0>`.

Figura 12: Fluxo para testes de tolerância a falhas (etapa dois)



Fonte: Elaborado pelo autor.

Na sétima etapa, uma amostra do funcionamento do sistema é verificada. O cliente avalia a função `max` e passa uma lista de inteiros aleatórios como argumentos. Na fase oito, o processo que contém o serviço de nomes é eliminado pela função `exit()`[43], assim como a função `regs()` já está disponível em Erlang, `exit()` também está disponível. Esta função recebe dois argumentos como entrada: o processo, que neste caso foi utilizado o recurso `whereis` que permite passar o nome como argumento ao invés do *Pid*, e o motivo pelo o qual este processo está sendo finalizando. Adiante, na etapa nove, mais uma vez, avaliamos o sistema, onde o cliente solicita a função `sort` e informa outra lista de inteiros. No retorno da avaliação é possível observar que o resultado obtivo foi o que estava sendo esperado. Por fim, a função `regs()` é executada para observar os processos que estão registrados.

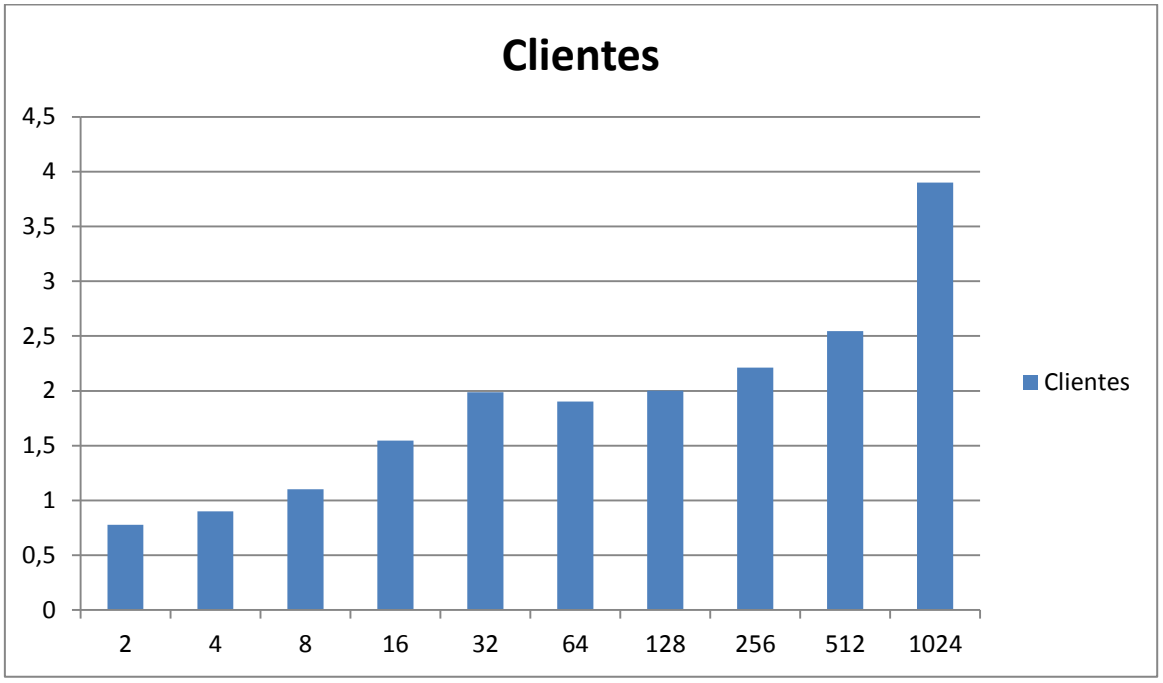
[43] http://erlang.org/doc/man/erlang.html#exit-2

Observe que o processo referente ao serviço de nomes tem um novo *Pid*, `<0.73.0>`. Após eliminarmos este processo, o MONITOR recebeu uma notificação que este processo tinha morrido e o motivo pelo o qual ele foi finalizado, criou outro processo para o mesmo serviço, e repassou o conteúdo para o novo processo (copiando o dicionário do processo antigo para o novo processo) mantendo os dados que poderiam ser perdidos.

Paralelismo

Na primeira etapa de testes para o aspecto de paralelismo no *middleware*, foi utilizada a função `div`, que já está presente na biblioteca padrão de Erlang, já que o propósito deste teste é avaliar a estabilidade do *middleware*. Como pode ser observado na figura abaixo, o tempo médio de processamento de uma invocação não varia bastante com o aumento de clientes usando ele simultaneamente. Portanto, isto mostra que o *middleware* se mantém escalável com um aumento do número de clientes.

Figura 13: tempo de processamento (ms) com variação de quantidade de clientes.

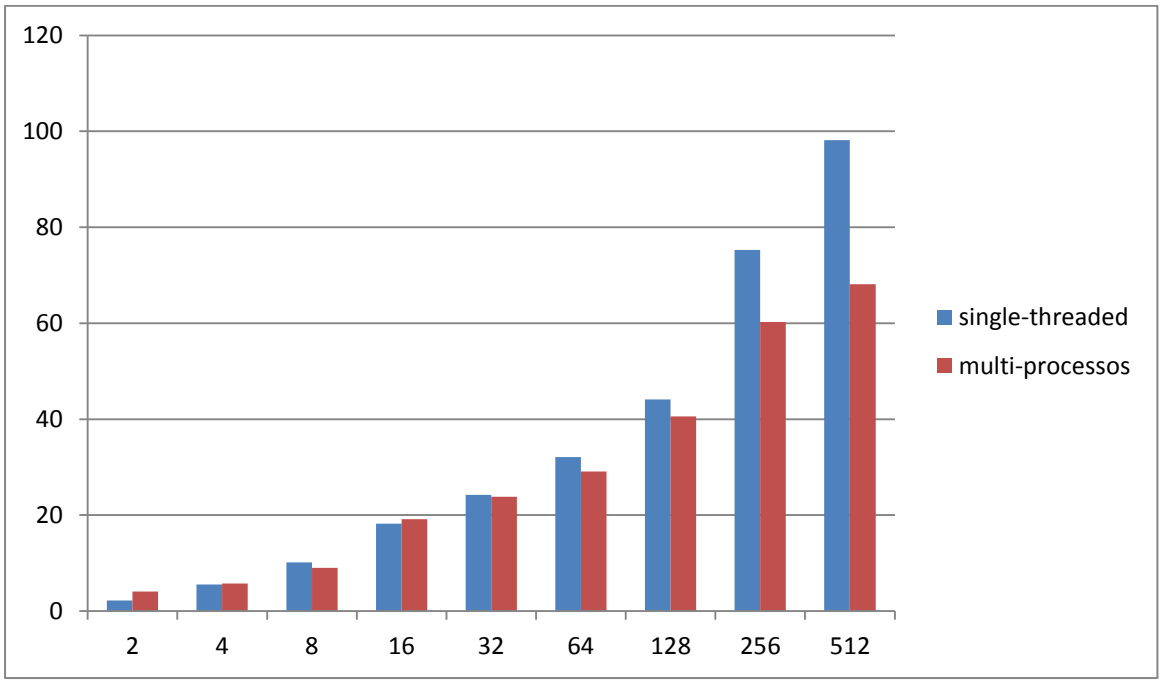


Fonte: Elaborado pelo autor.

Em seguida, foi avaliado o aspecto de paralelismo no servidor. Utilizando a versão *single-threaded* versus *multi-processos* do `mergesort` como medida, o foco é observar o *overhead* que Erlang possibilita em funções remotas. Este ciclo de testes utiliza apenas um

cliente solicitando *n* requisições. Dado que o propósito é entender a partir de que situação vale utilizar paralelismo no servidor.

Figura 14: impacto do paralelismo



Fonte: Elaborado pelo autor.

A partir da Figura 14 ficou claro que não é válido utilizar paralelismo para intervalos pequenos porque o esforço gasto para criar os processos que executam a função paralela acaba gastando mais tempo do que o custo de processar a versão *single-threaded*. Portanto, combinar paralelismo fornecido pelo middleware com o paralelismo proveniente da linguagem traz resultados gratificantes, se tratando de eficiência, nas invocações remotas em um sistema distribuído.

Neste capítulo foram gerados os testes para realizar uma análise do middleware. Concentrando os esforços na característica de tolerância a falhas e paralelismo, foi provado ser possível construir um sistema tolerante a falhas e concorrente utilizando linguagem funcional. Ademais, é importante destacar as dificuldades enfrentadas ao desenvolver um projeto que, inicialmente, foi proposto para uma linguagem orientada a objetos em outro

tipo de paradigma. Em contrapartida, os padrões remotos utilizados aqui se mostraram independentes de paradigma e/ou linguagem de programação, mostrando que a arquitetura destes padrões, pela visão do autor, deve ser considerada quando tratamos deste tipo de problemática.

Conclusões

Permitir que o desenvolvedor projete aplicações sem a preocupação com o fato de ser distribuída ou não, e problemas atrelados a essa característica, como problemas de rede ou localização, é algo bastante desejado. O *middleware* surgiu para tratar destes problemas ortogonais que não estão diretamente ligados a aplicação.

Neste trabalho de graduação foi apresentada uma versão de *middleware* baseado em chamadas remotas de procedimento utilizando Erlang para prover suporte a aplicações distribuídas. A linguagem permitiu de algumas características, como mecanismos de suporte a falhas e primitivas que facilitam paralelizar o sistema. Além disso, a arquitetura foi fortemente baseada nos padrões remotos para o desenvolvimento, com algumas modificações porque o conceito de objeto remoto foi substituído pela função remota.

O *middleware* obteve êxito ao prover os requisitos funcionais e não funcionais, como as transparências de acesso e localização. Como foi destacado no capítulo anterior, o componente de serviço de nomes ajudou a atingir alguns desses objetivos. Portanto, o projeto cumpriu com o seu dever.

Dificuldades encontradas

Durante a fase de implementação, a principal barreira para o progresso do projeto foi à falta de um bom ambiente voltado para o desenvolvimento em Erlang. O *Erlide*[44], uma variação de *Eclipse*[45] para Erlang, este ambiente levantava falhas frequentemente.

Para a etapa de análise, não foi possível encontrar uma ferramenta de testes que se encaixasse com o que o projeto propôs avaliar ou, pelo menos, se adequasse a arquitetura. *Tsung*[46], biblioteca que simula clientes com o foco em teste de escalabilidade e desempenho, foi considerado num primeiro momento, mas ela só permite testes unidirecionais.

[44] http://erlide.org/
[45] https://eclipse.org/
[46] http://tsung.erlang-projects.org/

Trabalhos Futuros

Nem todos os componentes do *middleware* estão paralelizados. O MARSHALLER, componente que serializa e deserializa os dados que são transportados pela rede através do CLIENT REQUEST HANDLER e SERVER REQUEST HANDLER, pode ser paralelizado a fim de obter maior velocidade no *middleware*.

O INVOKER também poderia sofrer melhorar consideráveis. Este elemento age sozinho no lado do servidor, algo passível de multiplicação. Assim, múltiplos INVOKERS devem, provavelmente, melhorar o desempenho do *middleware*.

Guia do Usuário

Primeiramente, baixe o Erlang no site oficial[47]. Nesta linguagem, maior parte das coisas pode ser feita a partir de um emulador (ele vai rodar os *scripts*, carregar módulos, *etc.*). Para iniciar o *shell* no *Linux/macOS* (para usuários *Windowns*, observe este link[48] para mais informações), abra um terminal, vá até a pasta `/src` do projeto e execute `$ erl`. Se tudo ocorrer bem, deve aparecer um texto parecido com o mostrado abaixo:

```
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-
threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
```

Experimentos de tolerância a falhas

1. Carregue os componentes: `c(marshaller)` (repita para todos os componentes).
2. Inicie o serviço de nomes: `kvs:start().`
3. Inicie o servidor: `server:start(mergesort, "localhost", 3456).`
4. Verifique o Pid do serviço de nomes: `regs().`
5. Execute a requisição: `client:eval(mergesort, [3, 1, 2, 0]).`
6. Elimine o processo do serviço de nomes: `exit(whereis(kvs), kill).`
7. Execute uma requisição: `cliente:eval(mergesort, [1, -1, 0, 3]).`
8. Verifique o Pid do serviço de nomes: `regs().`

Experimentos de paralelismo

1. Repita os três primeiros passos do experimento anterior
2. Inicie o servidor: `server:start(merge_sort, "localhost", 5676).`
3. Execute a função `spawn_n` passando a quantidade desejada de clientes e chamadas
4. Utilize a função no passo anterior como argumento da função `timer:tc`.

[47] https://www.erlang.org/downloads
[48] http://learnyousomeerlang.com/starting-out#the-shell

Referências Bibliográficas

[1] CONSORTIUM, World Wide Web. **Internet Live Stats: Real Time Statistics Project**. Organizado por World Wide Web Foundation. Disponível em: <http://www.internetlivestats.com/>. Acesso em: 02 dez. 2016.

[2] VÖLTER, Markus; KIRCHER, Michael; ZDUN, Uwe. **Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware**. [S. I.]: Wiley, 2004. 424 p. (Software Design Patterns).

[3] INTERNET.ORG. **State of Connectivity 2015: A Report on Global Internet Access**. United States of America: International Telecommunication Union, World Telecommunication/ict Indicators Database, 2015. 49 p. Annual study by Facebook. Disponível em: <https://fbnewsroomus.files.wordpress.com/2016/02/state-of-connectivity-2015-2016-02-21-final.pdf>. Acesso em: 02 nov. 2016.

[4] KAUR, Amandeep; MADHURIMA; MADHULIKA. **Recent innovations in Distributed Systems: Challenges and Benefits**. IJCEM International Journal of Computational Engineering & Management. Noida, Uttar Pradesh, India, p. 35-38. jun. 2013.

[5] COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim**. Distributed Systems: Concepts and Design**. 5. ed. USA: Addison-Wesley Publishing Company, 2011. 1008 p.

[6] TANENBAUM, Andrew S.; VAN STEEN, Maarten. **Distributed Systems: Principles and Paradigms**. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. 840 p.

[7] PUDER, Arno; RÖMER, Kay; PILHOFER, Frank. **Distributed Systems Architecture: A Middleware Approach**. [S.I.]: Morgan Kaufmann, 2005. 344 p. (The MK/OMG Press).

[8] LAMPORT, Leslie. **Time, clocks, and the ordering of events in a distributed system**. Communications of the ACM, New York, USA, v. 21, n. 7, p.558-565, 1 Jul. 1978. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/359545.359563.

[9] DEUTSCH, Peter. **The Eight Fallacies of Distributed Computing**. Disponível em: <https://blogs.oracle.com/jag/resource/Fallacies.html>. Acesso em: 05 dez. 2016.

[10] FACTOR, Phil. **The Eight Fallacies of Distributed Computing**. 2014. Disponível em: <https://www.simple-talk.com/blogs/the-eight-fallacies-of-distributed-computing/>. Acesso em: 05 dez. 2016.

[11] ROTEM-GAL-OZ, Arnon. **Fallacies of Distributed Computing Explained**. 2008. Disponível em: <https://www.cse.unsw.edu.au/~cs9243/16s1/papers/fallacies.pdf>. Acesso em: 05 dez. 2016.

[12] KAUR, Amandeep; MADHURIMA; MADHULIKA. **Recent innovations in Distributed Systems: Challenges and Benefits**. IJCEM International Journal of Computational Engineering & Management. Noida, Uttar Pradesh, India, p. 35-38. jun. 2013.

[13] BIRRELL, Andrew D.; NELSON, Bruce Jay**. Implementing remote procedure calls**. ACM Transactions on Computer Systems, [S.I.], v. 2, n. 1, p.39-59, 1 fev. 1984. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/2080.357392. Disponível em: <http://www.cs.virginia.edu/~zaher/classes/CS656/birrel.pdf>. Acesso em: 06 dez. 2016.

[14] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC 11578:1996**: Remote Procedure Call. 1 ed. [S.I.]: ISO/IEC JTC 1, 1996. 570 p.

[15] MADEIRA, Frederico. **Introdução aos Sistemas Distribuídos**. Recife, Pernambuco: Frederico Madeira, 2010. 21 slides, color, 25 cm x 20 cm. Introdução aos Sistemas Distribuídos. Disponível em: <http://www.slideshare.net/fred_m/introduo-aos-sistemas-distribudos>. Acesso em: 08 dez. 2016.

[16] BARNEY, Blaise. **Introduction to Parallel Computing**. Elaborada por Lawrence Livermore National Laboratory. Disponível em: <https://computing.llnl.gov/tutorials/parallel_comp/>. Acesso em: 08 dez. 2016.

[17] WEISER, Mark. **The computer for the 21st century**. Acm Sigmobile Mobile Computing And Communications Review,[S. I.], v. 3, n. 3, p.3-11, 1 jul. 1999. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/329124.329126.

[18] SCHANTZ, Richard E.; SCHIMIDT, Douglas C.. **Middleware for Distributed Systems**. Disponível em: <https://www.dre.vanderbilt.edu/~schmidt/PDF/middleware-encyclopedia.pdf>. Acesso em: 08 dez. 2016.

[19] OTHMAN, Ossama; SCHMIDT, Douglas C.. **Issues in the Design of Adaptive Middleware Load Balancing**. Proceedings of The 2001 ACM Sigplan Workshop on Optimization of Middleware and Distributed Systems - Om '01, New York, NY, USA, v. 36, n. 8, p.205-213, 2001. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/384198.384225.

[20] RAZZAQUE, Mohammad Abdur et al. **Middleware for Internet of Things: A Survey**. IEEE Internet of Things Journal, [S. I.], v. 3, n. 1, p.70-95, fev. 2016. Institute of Electrical and Electronics Engineers (IEEE). http://dx.doi.org/10.1109/jiot.2015.2498900.

[21] GALL, Nick. **Origin of the term "middleware".** 2005. Disponível em: <http://ironick.typepad.com/ironick/2005/07/update_on_the_o.html>. Acesso em: 08 dez. 2016.

[22] ARMSTRONG, Joe. **A history of Erlang**. Proceedings Of The Third Acm Sigplan Conference On History Of Programming Languages - Hopl Iii**,** [S. l.], p.1-26, 09 jun. 2007.

Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/1238844.1238850. Disponível em: <http://webcem01.cem.itesm.mx:8005/erlang/cd/downloads/hopl_erlang.pdf>. Acesso em: 10 dez. 2016.

[23] WHATSAPP. **1 million is so 2011.** 2011. Disponível em: <https://blog.whatsapp.com/196/1-million-is-so-2011?>. Acesso em: 06 jan. 2012.

[24] BORGES, Mateus Moury Fernandes da Rosa. **Uso de Elixir para a construção de um middleware baseado em RPC**. 2016. 46 f. TCC (Graduação) - Curso de Ciência da Computação, Centro de Informática, Universidade Federal de Pernambuco, Recife, 2016. Disponível em: <http://www.cin.ufpe.br/~tg/2016-1/mmfrb.pdf>. Acesso em: 29 out. 2016.

[25] ROSA, Nelson. **Sistemas Distribuídos:** Transparências. Slide 8/23. Disponível em: <https://www.dropbox.com/s/5jt91nbdr1oknn3/programacao-aul02-sistemas-distribuidos-middleware.pdf?dl=0>. Acesso em: 12 dez. 2016.

[26] AB, Ericsson. **Erlang Reference Manual User's Guide.** Version 8.1. Disponível em: <http://erlang.org/doc/reference_manual>. Acesso em: 13 dez. 2016.

[27] NYSTRÖM, Jan Henry. **Analysing Fault Tolerance for ERLANG Applications.** Disponível em: <https://uu.diva-portal.org/smash/get/diva2:213697/FULLTEXT01.pdf>. Acesso em: 01 dez. 2016.

[28] ARMSTRONG, Joe. **Making reliable distributed systems in the presence of sodware errors.** 2003. 295 f. Tese (Doutorado) - Curso de Computer Science, Department Of Microelectronics And Information Technology, The Royal Institute Of Technology, Stockholm, Sweden, 2003. Disponível em: <http://erlang.org/download/armstrong_thesis_2003.pdf>. Acesso em: 02 nov. 2016.

[29] WIGER, Ulf. **Monitor vs link.** [mensagem pessoal] Mensagem recebida por: <serge@hq.idt.net>. em: 02 abr. 2005.

[30] JAIN, Raj. **The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling**. [S. l.]: Wiley, 1991. 685 p. (Programming).