Universidade Federal de Pernambuco

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Luiz Artur Botelho da Silva

"FLIMSY: UM MIDDLEWARE FUNCIONAL EM SCALA"

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre Profissional em Ciência da Computação.

ORIENTADOR(A): ***Prof. Nelson Souto Rosa***

RECIFE 2015

“FLIMSY: UM MIDDLEWARE FUNCIONAL EM SCALA”

Por

Luiz Artur Botelho da Silva

Dissertação de Mestrado Profissional

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE 2015

Catalogação na fonte
Bibliotecário Jefferson Luiz Alves Nazareno CRB4-1758

S586f Silva, Luiz Artur Botelho da.
Flimsy: um middleware funcional em scala / Luiz Artur Botelho da Silva. – Recife: O Autor, 2015.
91 f.: fig., tab.

Orientador: Nelson Souto Rosa.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIN, Ciência da Computação, 2015.
Inclui referências.

1. Sistemas operacionais distribuídos (Computadores). 2. Padrões de software. 3. Software de aplicação- Desenvolvimento. I. Rosa, Nelson Souto (Orientador). II. Titulo.

004.2 CDD (22. ed.) UFPE-MEI 2015-118

Dissertação de Mestrado Profissional apresentada por **Luiz Artur Botelho da Silva** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título, "**FLiMSy: Um Middleware Funcional em Scala**", orientada pelo Professor Nelson Souto Rosa e aprovada pela Banca Examinadora formada pelos professores:

_____

Prof. Paulo Romero Martins Maciel
Centro de Informática / UFPE

_____

Prof. Fernando Antonio Aires Lins
Universidade Federal Rural de Pernambuco

_____

Prof. Nelson Souto Rosa
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 31 de julho de 2015.

_____

Profª. EDNA NATIVIDADE DA SILVA BARROS
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

I dedicate this thesis to all my family, friends and professors
who gave me the necessary support to get here.

Agradecimentos

Gostaria de agradecer primeiramente a Deus por ter me dado forças para concluir esse trabalho. Em seguida, gostaria de agradecer ao professor ***Nelson Rosa*** por ter me guiado tão bem nessa jornada. Não menos importante, gostaria de pedir desculpas a minha mãe ***(Dijanira Botelho Dias da Silva)*** e aos meus irmãos ***(José Dias Da Silva e Melkyadys José da Silva Neto)*** pelas ausências existentes durante a elaboração desse trabalho. Também gostaria de agradecer ao amor da minha vida, minha noiva e futura esposa ***(Anne Mari Miyachi)*** por me aturar de mau-humor e por, as vezes, eu não poder dá a devida atenção. Gostaria de agradecer aos meus colegas de mestrado ***Robson Timóteo***, ***Marcelo Lima***, ***Thiago Vidal*** e ***Emmanuel Tenório*** companheiros que sempre me motivaram e que me ajudaram durante o programa de mestrado. Por fim, gostaria de agradecer as pessoas que diretamente ou indiretamente contribuíram para que esse trabalho fosse concluído. São eles: ***Mauro Silva (CESAR)***, ***Heitor Souza (CESAR)***, ***Rodrigo Lins (CESAR)***, ***Fernando Brayner (AOL)***, ***Lucas Moreno (CESAR)*** e ***José Elias (CESAR)***.

O covarde nunca começa, o fracassado nunca termina, o vencedor nunca desiste.

—NORMAN VICENTE PEALE

Resumo

Sistemas distribuídos têm sido implementados em uma grande diversidade de domínios de aplicação, tais como: Finanças e comércio, com os sistemas de comércio eletrônico para compra de produtos pela Internet; Assistência Médica, que tem como exemplo o uso de prontuários eletrônicos online e a telemedicina no apoio a diagnóstico remoto ou serviços mais avançados, tais como cirurgia remota; Educação, onde podem ser destacadas as ferramentas de ensino baseadas na Web, tais como ambientes virtuais de aprendizagem à distância; Transporte e Logística, que usam tecnologias de localização; Gestão ambiental, que utiliza tecnologia de sensores para controlar e gerenciar o ambiente natural e com isso garantir um alerta precoce de catástrofes naturais, como terremotos, inundações ou tsunamis. Apesar de sua popularidade, existem vários desafios a serem vencidos ao projetar sistemas distribuídos. Dentre eles, podemos citar a heterogeneidade e o controle de concorrência. Alguns desses desafios podem ser solucionados usando-se um *middleware*, que é um software de conectividade que encapsula um conjunto de serviços e que reside entre a aplicação e o sistema operacional. Os sistemas de *middleware* permitem ainda que aplicações distintas se comuniquem abstraindo do usuário final como esse processo é realizado. O desenvolvimento de um *middleware* é uma atividade complexa, pois ao mesmo tempo que ele precisa esconder das aplicações distribuídas a complexidade dos mecanismos de concorrência e comunicação de baixo nível providos pelo sistema operacional, é preciso fornecer um conjunto crescente de serviços, tais como serviços de segurança e transação. Para tratar a complexidade mencionada , sistemas de *middleware* têm sido projetados com o uso de vários padrões de projeto especificamente definidos para a construção deste tipo de software. Além do uso destes padrões, há um esforço crescente para a adoção de novos paradigmas de programação no desenvolvimento de sistemas de *middleware*, em particular o paradigma funcional. Isso porque esses paradigmas oferecem nativamente recursos que lidam melhor com paralelismo e concorrência do que a maioria dos paradigmas presentes em linguagens de programação convencionais. Neste contexto, o objetivo deste trabalho é projetar e implementar um *middleware*, chamado FLiMSy, usando a linguagem de programação funcional Scala. FLiMSy foi projetado utilizando os padrões de projeto de *middleware* conhecidos como *Remoting Patterns*. Para avaliar a implementação proposta, foi realizado um experimento com o objetivo de medir o tempo de reposta do FLiMSy e também analisar as facilidades de se usar os recursos puramente funcionais de Scala no desenvolvimento de um *middleware* orientado a objetos.

Palavras-chave: *Middleware*. Programação Funcional. Scala. Padrões de Projeto.

Abstract

Distributed systems have been implemented in a wide variety of application domains, such as Finance and trade with e-commerce systems to buy products over the Internet; Healthcare, such as the use of online electronic medical records and telemedicine to support remote diagnostics or more advanced services, such as remote surgery; Education, which can be highlighted the teaching tools based on the Web, such as virtual environments distance learning; Transportation and Logistics, using geolocation technologies; Environmental stewardship, which uses sensor technology to control and manage the natural environment and thus provide early warning of natural disasters such as earthquakes, floods or tsunamis. Despite of its popularity, there are several challenges that need to be overcomed when we design distributed system, which include heterogeneity and concurrency control. Some of these challenges can be solved using a middleware, which is a connectivity software that encapsulates a set of services that lies on between the application and the operation system. The development of a middleware is a complex task, because it needs at the same time hide the complexity of the concurrency mechanism and the low level communication provided by the operation system from the distributed application, and needs also provide a set of increasing services, such as security service and transaction. To deal with mentioned complexity, middleware system has been design using specified design pattern to build this kind of software. Aside the usage of these design patterns, there is an increase effort to adopt new paradigm of programming to develop middleware softwares, in particular a funcional paradigm. That is because these paradigms natively offer features that deal better with parallelism and concurrency than most of the paradigms present in conventional programming languages. Within this context, the goal of this project is design and develop a middlware called FLiMSy, using Scala as funcional programming language. FLiMSy was developed using middleware's desing patterns known as Remoting Patterns. To evaluate the proposed implementation, an experiment was conducted in order to measure the response time of FLiMSy and also review the facilities of using purely functional features of Scala in developing a middleware object-oriented.

Keywords: Middleware. Functional Programming. Scala. Design Patterns.

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

Sumário

1 Introdução

Este capítulo inicia contextualizando e apresentando a motivação para este trabalho. Em seguida, o problema a ser tratado na dissertação é identificado e apresenta-se ainda uma breve análise das deficiências das soluções existentes para resolver este problema. Por fim, são apresentados os objetivos e a relevância do que está sendo proposto. O capítulo finaliza com a definição da estrutura do restante do documento.

1.1 Contexto e Motivação

As áreas de aplicação para sistemas distribuídos vêm crescendo bastante no cenário atual (Qilin & Mintian, 2010). Como exemplo disso, pode-se notar a Internet que é um grande sistema distribuído que provê seus serviços por meio de diversos protocolos de comunicação como *Hypertext Transfer Protocol (HTTP)*, *File Transfer Protocol (FTP)* e *Simple Mail Transfer Protocol (SMTP)* (Markus Volter & Zdun, 2005).

É possível observar também que há diversas áreas de aplicação que usam sistemas distribuídos, tais como: Finanças e comércio, com os sistemas de comércio eletrônico para compra de produtos pela Internet; Assistência Médica, que tem como exemplo o uso de prontuários eletrônicos online e da telemedicina no apoio a diagnóstico remoto ou serviços mais avançados, tais como cirurgia remota; Educação, onde podem ser destacadas as ferramentas de ensino baseadas na Web, tais como ambientes virtuais de aprendizagem à distância; Transporte e Logística, que usam tecnologias de localização, como Global Positioning System (GPS) e serviços de mapas baseados na Web, como MapQuest[1], Google Maps[2] e Google Earth[3]; Gestão ambiental,

[1] http://www.mapquest.ca
[2] http://maps.google.com
[3] https://www.google.com/earth/

que utiliza tecnologia de sensores para controlar e gerenciar o ambiente natural e com isso garantir um alerta precoce de catástrofes naturais, como terremotos, inundações ou tsunamis (George Coulouris & Blair, 2011).

O compartilhamento de recursos se destaca como uma das principais motivações para a construção de sistemas distribuídos. Através deste compartilhamento, recursos podem ser geridos por servidores e acessados por clientes (George Coulouris & Blair, 2011). O autor Arno Puder (2006) cita outras razões que podem ser utilizadas durante a concepção de sistemas distribuídos, são elas:

- **Escalabilidade:** garante a normalidade do sistema distribuído mesmo após um aumento significativo do número de recursos e de clientes;
- **Tolerância a falhas:** garante a disponibilidade do sistema mesmo em caso de falhas;
- **Independência de localização do cliente e serviço:** permite que várias máquinas remotas forneçam de forma transparente um único serviço;
- **Manutenção e implantação:** possibilita que alterações feitas nas regras de negócio dos servidores não afetem os clientes;
- **Segurança:** gerência o acesso aos recursos distribuídos;
- **Integração com o negócio:** facilita a comunicação entre sistemas distintos.

Porém, existem vários desafios a serem vencidos ao projetar sistemas distribuídos. Dentre eles, podemos citar o controle de concorrência, que é um problema de difícil solução em sistemas distribuídos, pois é preciso gerenciar de forma coordenada o acesso ao recurso solicitado. Alguns desses desafios podem ser solucionados por meio de *middleware*, que é um software de conectividade que encapsula um conjunto de serviços e que reside entre a aplicação e o sistema operacional. Os sistemas de *middleware* permitem ainda que aplicações distintas se comuniquem abstraindo do usuário final como esse processo é feito (Arno Puder, 2006).

Dada a sua complexidade, sistemas de *middleware* tem sido projetados com base em vários padrões de projeto para sistemas distribuídos (Markus Volter & Zdun, 2005). Estes padrões são utilizados para resolver alguns dos vários desafios presentes em aplicações distribuídas, tais como: Segurança, Transparência de localização, Transparência de acesso, Tolerância a falhas, Escalabilidade.

No entanto, há um esforço crescente para a adoção de novos paradigmas de programação no desenvolvimento de sistemas de *middleware*. Em particular, linguagens funcionais começam a ser utilizadas no desenvolvimento de *middleware* para tratar com a complexidade crescente deste tipo de software (Debasish Ghosh & Vinoski, 2011).

No paradigma funcional, programas são desenvolvidos por meio de funções matemáticas que evitam estados ou dados mutáveis. Além disso, não há nas linguagens funcionais alocação explícita de memória, nem declaração explícita de variáveis (Victor Pankratius & Garreton, 2012). Nativamente também há casos de implementação do modelo de Atores em algumas linguagens funcionais como Erlang que conseguem trabalhar melhor com concorrência do que as linguagens imperativas (Victor Pankratius & Garreton, 2012; Cesarini & Thompson, 2009). Atores são basicamente processos simultâneos que se comunicam através da troca de mensagens (Victor Pankratius & Garreton, 2012). Apesar das linguagens tradicionais como Java darem suporte à concorrência, os desenvolvedores, em alguns casos, precisam criar suas próprias soluções ou usar Application Programming Interface (API) de terceiros para complementar as suas soluções.

Dentre as diversas linguagens funcionais podemos citar Scala que é uma linguagem de programação que faz uso tanto do paradigma orientado a objetos quanto funcional, executa sobre uma Java Virtual Machine (JVM) facilitando sua portabilidade, e que pode ser usada combinada com Java fazendo uso da sua enorme variedade de bibliotecas (Martin Odersky & Venners, 2008). Scala possui uma sintaxe de *script* amigável e bem diferente das linguagens funcionais mais populares como Haskel e Erlang (Cesarini & Thompson, 2009; O'Sullivan *et al.* , 2008).

A programação em Scala se assemelha a programação em algumas linguagens de *script* atuais que são muito populares como Perl, Python e Ruby (Flanagan & Matsumoto, 2008; Van Rossum, 1995; Wall, 2000). Além disso, Scala é uma linguagem de programação extensível, pois ela foi projetada para crescer com as demandas de seus usuários, ou seja, ao invés de fornecer todas as ferramentas necessárias para a construção de um sistema, ela fornece um conjunto mínimo de funcionalidades e permite que a linguagem seja expandida de acordo com as necessidades do programador (Martin Odersky & Venners, 2008).

Por fim, a fusão de programação orientada a objetos e funcional é um dos principais benefícios de Scala, pois permite que desenvolvedores utilizem paradigmas distintos como se fosse apenas um. Também é possível utilizar recursos presentes na programação funcional como o modelo de Atores, assim como fazer uso dos recursos da programação orientado a objetos

como Classes, Herança e Polimorfismo (Martin Odersky & Venners, 2008).

1.2 O Problema

O desenvolvimento de *middleware* é uma atividade complexa, pois ao mesmo tempo que ele precisa esconder das aplicações distribuídas a complexidade dos mecanismos de concorrência e comunicação de baixo nível providos pelo sistema operacional, é preciso fornecer um conjunto crescente de serviços, tais como serviços de segurança e transação.

A dificuldade de implementar um *middleware* fica mais aparente e tem se acentuado por dois fatores principais: necessidade crescente de aplicações distribuídas cada vez mais robustas e complexas (Markus Volter & Zdun, 2005); e o suporte provido para controle de concorrência em linguagens amplamente utilizadas como Java. Java apresenta um modelo de dados compartilhado que faz uso de semáforos com bloqueios para controlar o acesso de recursos. Porém, este modelo muitas vezes se torna difícil de gerenciar quando essas aplicações aumentam de tamanho e complexidade. Isso porque estas aplicações que utilizam muitas *threads* precisam identificar e gerenciar as diversas *threads* que estão tentando modificar ou acessar dados simultaneamente provocando problemas como *deadlocks* e *race conditions*.

Neste contexto, o problema a ser considerado nesta dissertação é como projetar e implementar um *middleware* orientado a objetos utilizando a linguagem de programação funcional Scala.

1.3 Deficiências do Estado da Arte

Linguagens de programação tradicionais como Java e C++ são comumente usadas no desenvolvimento de sistemas *middleware*. No entanto, nelas, há diversos problemas, tais como: problemas de concorrência; paralelismo e tolerância a falhas, que são difíceis de serem resolvidos por meio de linguagens de programação tradicionais (Debasish Ghosh & Vinoski, 2011).

Tolerância a falhas é a capacidade do sistema se recuperar em casos de falhas tanto de hardware quanto de software garantindo, com isso, a disponibilidade do sistema. Java não oferece um suporte a tolerância a falhas que permita a um serviço distribuído continuar funcionando, caso algum de seus componentes falhem (Orne & de Araújo Macedo, 2000). Concorrência é a disputa por execução entre dois ou mais processos (Filman & Friedman, 1984). Já a paralelismo consiste em dividir a execução de uma aplicação em partes, onde essas partes são executadas pelo

processador no mesmo intervalo de tempo, proporcionando com isso um melhor desempenho (Almasi & Gottlieb, 1989).

Problemas de concorrência e paralelismo presentes em linguagens tradicionais de propósito geral como Java são motivados por deficiências nos mecanismos de sincronização utilizados para o acesso a recursos compartilhados originando, em alguns casos, problemas de *deadlocks* e *race conditions*.

Eles também podem provocar não só a degradação do desempenho do sistema em função do aumento do número de *threads* utilizadas, como também a dificuldade de manutenção em aplicações complexas que utilizam diversas *threads*. Em sistemas distribuídos, esses problemas se agravam, pois há um aumento na incidência de acesso a recursos compartilhados entre aplicações dessa natureza (Debasish Ghosh & Vinoski, 2011).

Do ponto de vista de *middleware*, não há muitas implementações de sistemas *middleware* implementados utilizando linguagens funcionais. Isso tem motivado uma nova tendência, visto que essas linguagens oferecem nativamente um maior suporte para problemas bastante comuns no desenvolvimento de aplicações distribuídas, como paralelismo e concorrência (Debasish Ghosh & Vinoski, 2011).

1.4 Solução Proposta

O objetivo deste trabalho é projetar e implementar um *middleware*, chamado FLiMSy, usando a linguagem de programação funcional Scala. Ele será projetado utilizando os padrões de projeto de *middleware* conhecidos como *Remoting Patterns* (Markus Volter & Zdun, 2005). A escolha por estes padrões de *middleware* deveu-se por vários fatores, dentre eles: padrões amplamente utilizados para construção de *middleware*, facilidade de manutenção e extensível.

Os padrões de *middleware* disponíveis nos *remoting patterns* são altamente reutilizáveis. Estes padrões são adotados para solucionar problemas comuns no contexto de sistemas distribuídos minimizando o tempo de desenvolvimento e garantindo um produto com melhor qualidade.

Com relação a facilidade de manutenção, um *middleware* pode ser construído utilizando vários padrões de projeto, onde que cada um deles é responsável por executar uma funcionalidade específica. O fato dos componentes ficarem isolados e possuírem responsabilidade distintas permite aos desenvolvedores identificar, com maior precisão, caso ocorra falhas na aplicação

distribuída.

Quanto à extensibilidade, o uso de *Remoting Patterns* permite que novos componentes sejam integrados ao *middleware* com maior facilidade, pois há padrões específicos para tratar estender partes do *middleware*.

O uso da linguagem de programação Scala na construção do *middleware* Middleware funcional em Scala (FLiMSy) se deu devido a vários fatores, como: facilidade de uso da linguagem durante o processo de desenvolvimento, portabilidade por fazer uso de uma JVM, suporte à concorrência com uso do modelo de Atores e suporte aos paradigmas funcionais e orientado a objetos aproveitando o melhor desses dois mundos. Além disso, Scala tem sido uma linguagem funcional que vem sendo usada por grandes empresas (Martin Odersky & Venners, 2008).

Também é possível observar que, o modelo de Atores de Scala se baseia nos mecanismos de troca de mensagens entre *threads*, ao invés de bloquear e liberar *threads* que é o modelo presente em diversas linguagens tradicionais como Java. Scala também fornece uma forma mais compacta de escrever código do que Java eliminando, com isso, repetições de código desnecessárias e ajudando na sua legibilidade (Martin Odersky & Venners, 2008).

Além disso, é comum desenvolvedores construírem aplicações distribuídas sem fazer uso de padrões de projetos para *middleware* consolidados no mercado. Isso pode ocasionar não só perda de qualidade do produto, pois eles teriam que desenvolver e testar bem todos os componentes usados na construção, como também provocar um aumento no prazo de desenvolvimento, pois é preciso desenvolver todos componentes para que eles atendam os requisitos básicos aplicados a sistemas distribuídos. Os padrões de projeto voltados para construção de *middleware* tem o objetivo de minimizar o processo de desenvolvimento e aumentar a qualidade do produto, já que eles foram testados por várias aplicações distribuídas presentes no mercado.

Dificuldade durante o processo de manutenção da aplicação distribuída seria outro problema ocasionado pela falta de padrões, pois sem eles é mais difícil para o desenvolvedor identificar e corrigir falhas. Também é possível observar a dificuldade de se inserir novos componentes na arquitetura do *middleware*, pois, em muitos casos, os desenvolvedores desenvolvem aplicações com componentes fortemente acoplados.

1.5 Estrutura da Dissertação

Essa dissertação é estruturada em cinco capítulos além da presente introdução:

- **Capítulo 2**: Este capítulo introduz os conceitos básicos necessários ao entendimento da proposta. Em particular, são apresentados os conceitos de *middleware*, modelos e padrões de projeto de *middleware*, programação funcional e Scala.
- **Capítulo 3**: Este capítulo descreve o *middleware* FLiMSy em detalhes. Para isto, o capítulo é estruturado seguindo as etapas de desenvolvimento do *middleware*, ou seja, apresenta inicialmente os seus requisitos, seguido pela definição da arquitetura, projeto e implementação. O capítulo conclui com um exemplo de uma aplicação distribuída construída com o FLiMSy.
- **Capítulo 4** Este capítulo apresenta uma avaliação experimental do *middleware* proposto. A avaliação experimental foca em aspectos de desempenho do FLiMSy.
- **Capítulo 5**: Este capítulo apresenta uma análise comparativa com as soluções existentes na literatura, com ênfase nos avanços propostos pelo FLiMSy.
- **Capítulo 6**: Finalmente, este último capítulo apresenta as conclusões e os trabalhos futuros.

2

Conceitos Básicos

Este capítulo introduz os conceitos básicos necessários ao entendimento do FLiMSy. Inicialmente são apresentados os conceitos de *middleware* e modelos de *middleware*. Em seguida, são introduzidos os padrões de projetos especialmente definidos para a implementação de sistemas de *middleware*. Finalmente, uma vez que FLiMSy é um *middleware* baseado em uma linguagem funcional, são apresentadas as principais características do paradigma funcional e a linguagem Scala.

2.1 *Middleware* e Modelos de *Middleware*

Um sistema distribuído pode ser definido como um conjunto de componentes que executam em computadores em rede e se comunicam e coordenam sua ações somente trocando mensagens (Coulouris *et al.* , 2011). Normalmente complexos, os sistemas distribuídos são também caracterizados por um conjunto de propriedades desejáveis, conhecidas como transparências (Coulouris *et al.* , 2011):

- **Transparência de Acesso**: permite que recursos locais e remotos sejam acessados usando operações idênticas;
- **Transparência de Localização**: permite que um recurso seja acessado sem que se tenha conhecimento sobre sua localização física ou de rede, como exemplo, endereço IP;
- **Transparência de Concorrência**: permite que vários processos executem concorrentemente usando recursos compartilhados sem que haja interferência entre eles;

- **Transparência de Replicação**: permite que várias instâncias de um recurso sejam utilizadas para melhorar a confiabilidade e o desempenho sem que os desenvolvedores das aplicações estejam cientes destas réplicas;
- **Transparência de Falhas**: permite que falhas sejam ocultadas dos usuários das aplicações distribuídas garantindo que tarefas sejam concluídas mesmo que estas falhas ocorram;
- **Transparência de Mobilidade**: permite o deslocamento de recursos e clientes em um sistema sem afetar a operação das aplicações;
- **Transparência de Desempenho**: permite que o sistema se reconfigure, inicializando novas instâncias como exemplo, para melhorar o desempenho quando ocorrem variações de carga na aplicação; e
- **Transparência de Escala**: permite que a aplicação se expanda sem que sua estrutura ou seus algoritmos sejam alterados.

Os sistemas distribuídos normalmente possuem apenas algumas destas características. Além disto, em função de sua complexidade, sistemas distribuídos são normalmente implementados com a ajuda de uma plataforma de *middleware* (ou *middleware* simplesmente). O *middleware* é uma camada de software localizada entre o sistema operacional e a aplicação distribuída que possui três papéis básicos: abstrair dos desenvolvedores de aplicações distribuídas a heterogeneidade dos ambientes de rede, suportar modelos de coordenação avançados entre os componentes das aplicações distribuídas, por exemplo *Publish/Subscribe*, e tornar transparente à estas aplicações a distribuição (Eugster *et al.* , 2003; Issarny *et al.* , 2007).

Plataformas de *middleware* são amplamente conhecidas como sistemas de software complexos (Bernstein *et al.* , 1987; Andrew T. Campbell & Kounavis, 1999; Schmidt *et al.* , 2003; Venkatasubramanian, 2002; Vinoski, 2002). Esta complexidade está relacionada principalmente à necessidade de fornecer um grande número de transparências e serviços para desenvolvedores de aplicações distribuídas. Na prática, os desenvolvedores querem se manter distantes das dificuldades em tratar mecanismos de baixo nível de concorrência, comunicação e distribuição, ao mesmo tempo que precisam de serviços distribuídos para agregar valor às suas aplicações. Portanto, espera-se que as plataformas de *middleware* forneçam transparências de distribuição, tais como acesso, localização, falha, concorrência, migração e assim por diante (ISO, 1995).

Ao mesmo tempo, o conjunto de serviços de *middleware* geralmente inclui serviços de nomes, segurança, concorrência, licenciamento, implantação e assim por diante.

Apesar do grande número de taxonomias de *middleware* existentes (Emmerich, 2000; Bernard & Bernard, 2006; Vinoski, 2002), nesta dissertação adotamos uma extensão da classificação proposta por Emmerich (Emmerich, 2000). Esta classificação foi escolhida porque ela é independente de características particulares de sistemas de *middleware* comerciais existentes.

De acordo com Emmerich, os sistemas de *middleware* existentes seguem quatro modelos básicos: transacional, orientado a mensagem, procedural e orientado a objetos. Esta classificação foi definida adotando-se como único critério a primitiva de comunicação fornecida pelo *middleware* para interação entre os componentes da aplicação distribuída.

- ***Middleware* Transacional**: *middleware* transacional permite que os componentes de uma aplicação distribuída interajam através do uso de transações. Na prática, as interações entre os componentes são tratadas como invocações remotas com as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Tipicamente, o *middleware* utiliza protocolos *two-phase commit*(Bernstein *et al.* , 1987) para implementar transações distribuídas. Aplicações construídas com este modelo são tipicamente cliente/servidor onde o cliente invoca uma transação ao servidor.

- ***Middleware* Orientado a Mensagem**: Mais conhecido como MOM (Eugster *et al.* , 2003), este modelo de *middleware* suporta a comunicação entre componentes distribuídos através da troca de mensagens. Uma característica fundamental deste modelo é o suporte natural à comunicação assíncrona e em grupo. MOMs são normalmente implementados de acordo com o padrão JMS (*Java Message Service*)(Hapner *et al.* , 2013).

- ***Middleware* Procedural**: A primitiva básica de comunicação fornecida por sistemas de *middleware* procedurais é a chamada remota de procedimento. Neste modelo, inicialmente o servidor registra em um diretório (conhecido como serviço de nomes) os procedimentos que ele implementa, em seguida o cliente busca os serviços desejados no diretório e posteriormente invoca este serviços diretamente ao servidor. Conhecido como um dos primeiros sistemas de *middleware*, o RPC (*Remote Procedure Call*) é a mais tradicional implementação do modelo procedural (Birrell & Nelson, 1984).

- ***Middleware* Orientado a Objetos**: *Middleware* orientado a objetos é uma evolução do *middleware* procedural, onde um cliente invoca um método remoto ao servidor ao invés de um procedimento. Esta evolução ocorreu em paralelo ao surgimento das linguagens orientadas a objeto. Sistemas de *middleware* orientado a objetos tornaram-se muito populares com o surgimento do padrão CORBA (*Commom Object Request Broker Architecture*) e do implementação do RMI (*Remote Method Invocation* (Grosso, 2001; OMG, 2004)).

A implementação de sistemas de *middleware* seguindo estes modelos tem sido feita basicamente utilizando padrões de projeto de *middleware*.

2.2 Padrões de Projeto de *Middleware*

A crescente complexidade dos sistemas de *middleware* tem levado à definição de padrões de projeto especialmente concebidos para a implementação deste tipo de software. Em particular, três conjuntos de padrões de projeto tem sido amplamente utilizados no desenvolvimento de *middleware*: POSA (*Pattern-Oriented Software Architecture*) (Schmidt *et al.* , 2006; Buschman *et al.* , 2007), *Remoting Patterns* (Arno Puder, 2006) e *Enterprise Integration Patterns* (Hohpe & Woolf, 2011; Arno Puder, 2006). Entre estes conjuntos, o mais popular na construção de *middleware* orientado a objetos é o *remoting pattern*(Arno Puder, 2006).

O *Remoting Patterns* (Markus Volter & Zdun, 2005; Zdun *et al.* , 2004) consiste de um conjunto de padrões de projetos e uma linguagem de padrões de projeto especialmente desenvolvidos para tratar aspectos importantes da implementação de *middleware* orientado a objetos. Os padrões foram agrupados em seis categorias de acordo com suas funcionalidades no *middleware*: padrões remotos básicos, padrões de identificação, padrões de gerenciamento de ciclo de vida, padrões de extensão, padrões de infraestrutura estendida, e padrões de invocação assíncrona.

Os padrões remotos básicos, como o próprio nome sugere, são aqueles padrões comumente implementados na maioria dos sistemas de *middleware*. Estes padrões implementam as funcionalidades básicas dos sistemas de *middleware* orientados a objetos e são normalmente agrupados no padrão de projeto composto chamado *Broker*. O *Broker* inclui oito padrões básicos: *Interface Description*, *Client Proxy*, *Requestor*, *Client Request Handler*, *Server Request Handler*, *Invoker*, *Marshaller* e *Remoting error*.

Para encontrar objetos remotos dentro de um sistema distribuído, clientes precisam identificar, endereçar e localizar estes objetos. Para isto, os desenvolvedores de aplicações distribuídas associam um identificador ao objeto remoto para identificá-lo (padrão *Object ID*). Como estes identificadores são válidos apenas no servidor onde ele está executando, é preciso estender esta identificação com informações de localização do servidor tais como nome da porta e do *host* (padrão *Absolute Object Reference*). Finalmente, para evitar o acoplamento direto entre clientes e objetos remotos, as informações de identificação são mantidas e gerenciadas por um servidor que implementa o padrão de projeto *Lookup*.

Objetos remotos diferentes normalmente precisam de gerenciamento de ciclo de vida diferentes: alguns precisam existir durante toda a execução do servidor e outros existem apenas durante um período limitado de tempo. Além deste gerenciamento de ciclo de vida, uma outra atividade importante a ser realizada sobre os objetos remotos dizem respeito ao momento da ativação e desativação dos objetos. Para acomodar todos estes elementos, o *remoting pattern* possui três padrões de gerenciamento de ciclo de vida: *Static instance*, *Per-request instances* e *Client-dependent instances*. Além disto, a ativação destes objetos podem ser feitas utilizando três padrões de ativação: *Leasing*, *Pooling*, *Lazy Acquisition* e *Passivation*.

Os padrões de extensão são usados para estender a funcionalidade do *middleware*, por exemplo, para suportar segurança, transações, troca de protocolos de comunicação e assim por diante. Para permitir estas extensões, três padrões podem ser utilizados: *Invocation interceptor* permite a injeção de comportamento adicional à invocação do objeto remoto, como exemplo, criptografando as invocações; *Invocation context* permite a adição de informações às chamadas remotas, exemplo disto, as credenciais do cliente; e, por fim, o padrão *protocol plug-in* que pode estender o protocolo de comunicação usado pelo *middleware*, por exemplo, para suportar múltiplas conexões.

Os padrões de infra-estrutura estendida são usados para implementar características avançadas do *middleware* como o gerenciamento de ciclo de vida (*Lifecycle manager*), a configuração simultânea de múltiplos objetos remotos (*Configuration groups*), monitoramento de atributos de qualidade dos objetos remotos (*QoS Observer*), a definição explicita de objetos locais *Local objects*, encaminhamento automático de invocação de mensagens (*Location forward*).

Finalmente, os padrões de invocação assíncrona permitem que clientes invoquem objetos remotos e continuem executando sua tarefas enquanto a chamada remota é executada. Para isto, quatro padrões podem ser utilizados: *Fire-and-forget*, onde o controle retorna ao cliente

imediatamente após a chamada; *Syncw-with-server*, quando o controle retorna ao cliente logo depois de recebido um reconhecimento que a invocação foi recebida pelo servidor; *Poll-object* permite ao cliente definir um objeto que fica responsável por realizar consultas sobre o término da invocação; e *Result-callback*, onde um objeto é utilizado para notificar o cliente quando a reposta ao método remoto foi concluída.

2.3 Programação Funcional

A programação funcional é um paradigma de programação baseado na avaliação de funções matemáticas. O princípio fundamental da programação funcional é que um cálculo pode ser realizado por funções compondo o sentido matemático, o que significa que uma função sempre produz a mesma saída quando for dada a mesma entrada (Hughes, 1989). Na visão de Sebesta (2011), o objetivo da programação funcional é camuflar funções matemáticas o máximo possível.

Bird & Wadler (1988) define que a programação em uma linguagem funcional consiste na construção de definições que utilizam o computador para avaliar expressões. O principal papel do programador é a de construir uma função para resolver um determinado problema. Esta função, que pode envolver uma série de funções subsidiárias, é expresso em notação que obedece a princípios matemáticos.

Uma das principais características da programação funcional é a transparência referencial. Essa transparência indica que a execução de uma função sempre produz o mesmo resultado quando é executada com os mesmos parâmetros, evitando com isso efeitos colaterais funcionais (Sebesta, 2011). Efeito colateral funcional é quando o resultado da execução de uma função modifica ou é modificada por uma variável externa. Um exemplo de efeito colateral funcional é quando uma função tenta altera um valor de uma variável global ou estática de uma classe que é usada por outra função (Hughes, 1989). A Listagem 2.1 feita em Java mostra um exemplo de efeito colateral funcional.

```
1 public class Test {
2   int x;
3   int y;
4
5   public int soma(int x, int y) {
6     this.x = x;
```

```
7       this.y = y;
8       return this.x + this.y;
9     }
10
11    public static void main(String[] args) {
12      Test test = new Test();
13      test.soma(20, 10);
14    }
15  }
```

Listagem 2.1: Exemplo de efeito colateral funcional em Java

O código define uma classe chamada *Test* que possui as variáveis globais *x* e *y*, Linhas 1 a 3. Nesta classe há também 2 métodos *soma()* e *main()*. O método *main()*, Linhas 11 a 14, é usado para testar o exemplo. Já o método *soma()* , Linhas de 5 à 9, é usado para receber 2 valores passados por argumento e somá-los. A presença de efeito colateral ocorre na Linha 8 do método *soma()*. Esse efeito é provocado porque não é possível prever o resultado esperado do retorno desta função, pois ele pode ser influenciado pelas variáveis globais *x* e *y* que podem ser acessadas externamente.

Outra característica das linguagens de programação puramente funcionais é o fato delas não usarem variáveis para guardar em memória os valores intermediarias das operações ou sentenças de atribuição (Sebesta, 2011). Isso impossibilita o uso de mecanismos de repetição, como laços, pois esses mecanismos são controlados por variáveis. As repetições são especificadas por recursão em linguagens puramente funcionais.

Em linguagens de programação imperativas, o resultado da avaliação de uma expressão é armazenado em uma posição de memória que é representada por uma variável em um programa. Por exemplo, o processo de avaliação da expressão: (A*B)+(C/D) é feito em 3 partes. Primeiramente (A*B) é avaliado e armazenado em memória. Em seguida, o resultado da avaliação de (C/D) também é feita e armazenada em memória. Após isso, os valores armazenados são avaliados gerando o resultado desejado (Pratt & Zelkowitz, 2000).

Já em linguagens puramente funcionais, por não usarem variáveis para guardar os seus resultados intermediários em memória, elas utilizam uma estrutura de dados do tipo *stack*, ou seja pilha, para guardar os resultados intermediários da avaliação de uma expressão. Os resultados armazenados na *stack* ficam em memoria até o termino da execução da expressão, após isso, eles são liberados (Bird & Wadler, 1988). O paradigma funcional é citado por Coutts

& Loh (2012); P. Totoo & Loidl (2012) como uma técnica promissora no desenvolvimento de aplicações que fazem uso de programação concorrente. Isso porque código puramente funcional não possui variáveis e, portanto, não necessitam usar mecanismos de *locks* para gerenciar o acesso a variáveis compartilhadas.

O valor de uma função é o tipo mais importante na programação funcional (Bird & Wadler, 1988). Uma função pode ser passada como argumento para funções e retornada como resultado. Linguagens funcionais podem fazer uso de *Currying* para realizar chamadas encadeadas entre funções (Bird & Wadler, 1988). O *Currying* é um recurso que divide uma função que recebe vários argumentos em várias funções de apenas um argumento que podem ser executas em cadeia onde que a ultima função executa a ação pretendida. A Listagem 2.2 mostra um exemplo de *Currying* em Scala.

```
1 object Test {
2   def main(args: Array[String]): Unit = {
3     print(Test.add(1)(20))
4   }
5
6   def add(x: Int)(y: Int): Int = {
7     x + y;
8   }
9 }
```

Listagem 2.2: Exemplo de *Currying* em Scala

Nesta listagem. a Linha 1 cria um objeto em Scala que possui os métodos *main()* e *add()*. O método *main()*, Linhas de 2 a 4, é usado para testar o exemplo. Já as Linha de 6 a 8 são usadas para demostrar o uso de Currying pelo método *add()*. O método *add()* recebe dois argumentos por parâmetro e retorna a soma deles. O uso do método *add()*, na Linha 3, mostra o encadeamento de chamadas provocado pelas chamadas sucessivas a si mesmo, demostrando o uso de *Currying*.

Por fim, diversas são as linguagens funcionais existentes atualmente, dentre elas podemos citar Lisp (Steele, 1990), Erlang (Cesarini & Thompson, 2009) e Haskel (O'Sullivan *et al.* , 2008; Marlow, 2013). Lisp foi uma das primeiras linguagens funcionais que surgiram. Além disso, ela não era considerada uma linguagem puramente funcional, pois a maioria das suas versões incorporavam recursos imperativos, tais como uso de iteração e sentenças de atribuição (Sebesta, 2011).

Já Haskel, é uma linguagem de programação puramente funcional, de proposito geral. Algumas das características de Haskel são o uso de tipagem forte, recursão, modelo de programação paralela determinísticas que visa eliminar erros tradicionalmente conhecidos que são associados com a programação paralela (Marlow, 2013).

2.4 Scala

Scala (*Scalable Language*) é uma linguagem de programação fortemente tipada que faz uso combinado de recursos do paradigma orientado a objeto e da programação funcional (Debasish Ghosh & Vinoski, 2011; Martin Odersky & Venners, 2008). Scala também compila seu código para bytecodes usando a JVM da mesma forma que Java. Além disso, é possível invocar código Java de Scala e vice-versa.

Programas desenvolvidos em Scala costumam ser simples e ter menos linhas de código do que Java. Martin Odersky & Venners (2008) relata que um programa em Scala tem cerca da metade do número de linhas do mesmo programa feito em Java. Isso porque a sintaxe de Java é mais sobrecarregada do que a de Scala. A Listagem 2.3 faz comparação utilizando como exemplo a criação de classes e construtores em Java e Scala.

```
1  //Java
2  class MyClass {
3      private int index;
4      private String name;
5      public MyClass(int index, String name) {
6          this.index = index;
7          this.name = name;
8      }
9  }
10 //Scala
11 class MyClass(index: Int, name: String)
```

Listagem 2.3: Criação de construtores em Java e Scala

Comparando o primeiro trecho de código escrito em Java, Linhas de 2 a 9, com o segundo trecho de código em Scala, Linha 11, é possível perceber que, para o mesmo exemplo, o código escrito em Java demandou um esforço maior de codificação do programador do que o código escrito em Scala.

Scala destaca-se como uma linguagem com um sistema de tipo estático muito avançado

(Martin Odersky & Venners, 2008). Linguagens de tipagem estática realiza, o processo de verificação dos tipos das variáveis em tempo de compilação eliminando possíveis erros de verificação de tipos em tempo de execução (PIERCE, 2002).

Com relação à fusão de paradigmas, Scala é considerada uma linguagem orientada a objetos, pois cada valor nela é um objeto e cada operação é uma chamada de método. Scala também é uma linguagem funcional completa, pois incorpora a maioria dos recursos presentes no paradigma funcional, tais como: imutabilidade de variáveis, inferência de Tipos, uma biblioteca com estruturas de dados imutáveis, suporte nativo a *currying*, *pattern matching*, modelo de atores e *closure* (Martin Odersky & Venners, 2008). Alguns desses recursos serão detalhados no decorrer deste capitulo.

Martin Odersky & Venners (2008) enfatiza o quanto a paradigma funcional torna fácil e rápida a construção de partes simples de uma aplicação. Já com relação ao paradigma orientado a objetos, ele facilita uma melhor estruturação de sistemas grandes. Além disso, a combinação dos 2 paradigmas torna possível não só expressar novos padrões de programação, como também melhorar o estilo de programação deixando ele legível e conciso.

O estudo de Victor Pankratius & Garreton (2012) fez um comparativo entre a linguagem de programação Scala e a linguagem Java. Esse estudo listou algumas desvantagens de Scala em relação a Java, dentre elas podemos citar:

- Scala foi superior a Java na alocação de 4 *threads* executando em uma arquitetura *multicore*, porém, o mesmo experimento em uma arquitetura com apenas um núcleo os resultados se mostraram semelhantes.
- Projetos em Scala apresentaram um tempo maior durante o processo de compilação do que projetos em Java.
- Apesar de Scala apresentar uma diminuição na quantidade de linhas de código em comparação a Java, às vezes códigos em Scala precisam de um esforço maior do programador para entendê-lo.
- Códigos em Scala exige um maior tempo de teste e *debug* do que o códigos feitos em Java

Já a inferência de tipos é um recurso de Scala que permite ao programador omitir a definição de tipos em qualquer lugar do código transferindo a responsabilidade de deduzir o tipo

para o compilador (Martin Odersky & Venners, 2008). Bem diferente de linguagens como Java, que para se criar uma variável é necessário previamente definir o seu tipo. O uso de inferência de tipos também reduz o trabalho repetitivo de definição de tipos tornando o programa menos confuso e mais legível. A Listagem 2.4 mostra possíveis formas de criação de variáveis em Scala demostrando o uso de inferência de tipos.

```
1 object Test {
2   def main(args: Array[String]): Unit = {
3     var x = 1 + 2 * 3;
4     var y = x.toString();
5   }
6 }
```

Listagem 2.4: Exemplo de inferência de tipos em Scala

Neste código é mostrada a criação das variáveis *x* e *y* sem a necessidade de determinar o tipo delas, Linhas 3 e 4. A variável *x* recebe um valor numeral inteiro como reposta. Já a variável *y* recebe como reposta a conversão do valor da variável *x* em uma sequência de caracteres. Em linguagens de programação como Java isso não é possível, pois o compilador exige que os tipos das variáveis criadas sejam configurados previamente.

Outro recurso do paradigma funcional usado por Scala é o *Pattern Matching* que é um sofisticado sistema de detecção de padrões, que chama um método após avaliar a sua estrutura (Martin Odersky & Venners, 2008). Este recurso permite localizar um tipo determinado em um conjunto composto por diversos tipos. Na Listagem 2.5 é apresentado um exemplo de *Pattern Matching* feito em Scala.

```
1 object Test {
2   def main(args: Array[String]): Unit = {
3     println(matchTest("two"))
4   }
5
6   def matchTest(x: Any): Any = x match {
7     case 1 => "one"
8     case "two" => 2
9     case y: Int => "scala.Int"
10  }
11 }
```

Listagem 2.5: Exemplo de *Pattern Matching* em Scala

Neste exemplo, o método *matchTest()* recebe como parâmetro a variável de um tipo genérico chamada *x*, Linha 6. Dentro deste método o valor correspondente à variável *x* é avaliado por meio das cláusulas *cases*, Linhas de 7 à 9, até achá-lo. Após isso, o método retorna o valor presente dentro da cláusula *case* que corresponde ao valor da variável *x* passada como argumento do método *matchTest()*. Note que o método *matchTest()* recebe um argumento definido em *x* de qualquer tipo e que as cláusulas *cases* avaliadas dentro do método também são de tipos diferentes, Linhas de 7 à 9. Isso é a essência do *Pattern Matching* que permite achar correspondências mesmo em tipos diferentes.

Não menos importante, *closure* é um recurso presente no paradigma funcional e também suportado por Scala. *Closure* é uma função cujo valor de retorno depende do valor de uma ou mais variáveis declaradas fora dela (Martin Odersky & Venners, 2008). A Listagem 2.6 mostra um exemplo de *closure* feito em Scala.

```
1 object Test {
2   def main(args: Array[String]): Unit = {
3     println(addMore(10));
4   }
5   var more = 1;
6   val addMore = (x: Int) => x + more;
7 }
```

Listagem 2.6: Exemplo de *Closure* em Scala

A Listagem 2.6 apresenta o *Closure* em um exemplo de código feito em Scala. A Linha 6 cria o método chamado *addMore()* que recebe como parâmetro a variável *x* do tipo *Int*. O objetivo do método *addMore()* é somar o valor da variável *x* passada por parâmetro com o valor da variável *more* que é definida externamente a este método, Linha 5.

Linguagens imperativas como Java possuem bibliotecas de suporte à programação concorrente baseada em *threads*. Scala pode fazer uso dessas bibliotecas como qualquer outra de Java. Porém, por suportar o paradigma funcional, Scala possui nativamente o suporte ao modelo de ator da linguagem funcional Erlang.

Atores possuem um modelo de concorrência que evita muitas dificuldades, como gerenciar variáveis que estão sendo manipuladas por outras *threads* presentes em modelos suportados por linguagens como Java. Isso porque o modelo de Atores se baseia em um mecanismo de troca de mensagens assíncronas entre atores, partes integrantes de uma comunicação, que minimiza problemas de concorrência dessa natureza (Debasish Ghosh & Vinoski, 2011; Martin Odersky &

Venners, 2008). Um ator pode realizar duas operações básicas, envio e recebimento de mensagem. A operação de envio, denotada por um ponto de exclamação (!), envia uma mensagem para um ator. Cada ator tem uma caixa de correio na qual as mensagens recebidas são enfileiradas. Um ator manipula mensagens que chegam em sua caixa de correio por meio da palavra reservada *receive* ou *react*.

A diferença entre *receive* e *react* é que *receive* cria sempre uma nova *thread* para tratar as mensagens recebidas enquanto que *react* reusa as *thread* já criadas para tratar as mensagens recebidas. A Listagem 2.7 mostra um exemplo de uso de Atores em Scala.

```
1  object Test {
2    def main(args: Array[String]): Unit = {
3      var actor: MyActor = new MyActor();
4      actor.start() ! "msg Test Actor!"
5    }
6  }
7  class MyActor extends Actor {
8    def act() {
9        receive {
10         case msg =>
11           println("received message: " + msg);
12       }
13   }
14 }
```

Listagem 2.7: Exemplo de atores em Scala

Para construir o Ator, foi criada a classe *MyActor* que estende da classe *Actor* da API de Scala, Linha 7. O método *act()* da classe *Actor* precisou ser sobrescrito na classe *MyActor*, Linhas de 8 à 13, para atender as mensagens recebidas por este Ator. Dentro do método *act()* da classe *MyActor*, a palavra reservada *receive*, Linha 9, é usada para criar uma nova *thread* para cada mensagem recebida pelo Ator. A mensagem recebida é avaliada pelas cláusulas *cases* presentes dentro de *receive* a fim de encontrar uma correspondência para ela, Linha 10. A mensagem só é processada por *receive*, caso seja encontrada uma correspondência dela em uma das suas cláusulas *cases*. Para iniciar o Actor, é preciso criar uma instancia de *MyActor* e ,em seguida, invocar o método *start()* de *Actor* passando a mensagem desejada, Linhas 3 e 4.

2.5 Considerações Finais

Este capítulo apresentou conceitos básicos *middleware* e modelos de *middleware*. Além disso, foi descrito também as tecnologias utilizadas no desenvolvimento do FLiMSy. Com relação as tecnologias utilizadas, foram expostos conceitos básicos sobre a linguagem de programação Scala e o uso de *remoting patterns* que são padrões de projeto que fornecem uma infraestrutura básica para aplicações distribuídas.

3

FLiMSy: *Middleware* Funcional em Scala

Este capítulo apresenta detalhadamente o *middleware* FLiMSy. Inicialmente são apresentados os requisitos do *middleware*. Com os seus requisitos definidos, em seguida apresentamos a arquitetura do FLiMSy e suas camadas. Também serão mostrados quais padrões de projeto foram utilizados e como eles foram implementados utilizando a linguagem de programação Scala.

3.1 Visão Geral

FLiMSy é um *middleware* orientado a objetos implementado na linguagem de programação Scala. Ele foi construído com a finalidade de prover uma infraestrutura de distribuição para a construção de aplicações cliente/servidor. Para isso, foram definidos inicialmente um conjunto de requisitos a serem atendidos pelo FLiMSy. Esses requisitos são relacionados à comunicação, ao controle de concorrência e as transparências de acesso e localização. Eles estão definidos no *Reference model of open distributed processing (RM-ODP)*(1995) e servem como base o desenvolvimento de sistemas distribuídos. Estes requisitos são detalhados a seguir:

- **[RF01] Comunicação confiável**: O *middleware* deverá fornecer comunicação confiável entre os componentes da aplicação distribuída. Este requisito deve garantir que requisições feitas pelos clientes sejam recebidas pelo servidor e que respostas à estas requisições sejam recebidas pelos clientes.

- **[RF02] Serialização de dados**: O *middleware* deverá ser capaz de serializar diversos tipos de dados. Este requisito garante que o *middleware* seja capaz de converter objetos de tipos genéricos em fluxo de bytes e também reverter esse processo. Como exemplo, se o *middleware* receber como argumento um objeto do tipo *string*, esse objeto deve ser convertido em uma sequência de bytes antes de ser enviado ao destino.

Após isso, quando a sequência de bytes chegar ao seu destino, o *middleware* pede ver capar de convertê-la novamente em um objeto do tipo *string*.

- **[RF03] Uso de padrões de projeto de *middleware***: O *middleware* deverá ser desenvolvido utilizando padrões de projetos especificamente projetados para a implementação deste tipo de software. Este requisito garante ao *middleware* um bom projeto de software, uma vez que estes padrões já são bem consolidados e testados.

- **[RF04] Transparência de localização**: O *middleware* deverá abstrair do cliente a localização dos objetos remotos distribuídos na rede. Este requisito deve garantir que serviços sejam descobertos na rede sem que o cliente saiba onde eles estão localizados.

- **[RF05] Transparência de acesso**: O *middleware* deverá ocultar diferenças na representação de dados e como um recurso é acessado. Ele tem que ser capaz de fornecer interfaces com as assinaturas das operações dos objetos remotos disponíveis na rede fazendo com que chamadas remotas se pareçam com chamadas locais.

- **[RF06] Controle de Concorrência**: O *middleware* deverá permitir o acesso concorrente aos objetos remotos de forma transparente. Este requisito garante que será transparente as aplicações o acesso concorrente aos objetos remotos.

- **[RF07] Comunicação baseada em RPC**: FLiMSy deverá permitir a interação entre os componentes da aplicação distribuída baseada no modelo de interação do RPC. Neste caso, um componente cliente invoca remotamente um método em um objeto remoto implementado pelo servidor.

3.2 Arquitetura

A arquitetura do FLiMSy é subdividida em 4 camadas com responsabilidades distintas: infraestrutura, distribuição, serviços e aplicação. Essa subdivisão faz com que cada camada possua uma responsabilidade única de fácil entendimento minimizando dificuldades de manutenção, pois o desenvolvedor saberá em qual camada mexer, caso haja algum problema. Essa subdivisão também propícia uma melhor visão global da arquitetura facilitando a inclusão de novos módulos.

A camada de infraestrutura abstrai todo o processo de comunicação feito dentro do *middleware*. Já a camada de distribuição é responsável por permitir o acesso aos recursos distribuídos na rede como se eles estivessem disponíveis localmente. Na prática, os clientes invocam objetos locais ou remotos de uma maneira semelhante.

Com relação à camada de serviço, ela é responsável por prover a transparência de localização dos objetos remotos, ou seja, os clientes não precisam saber onde estão localizados os objetos remotos na rede sendo papel desta camada fornecer um serviço que entregue a localização deles para os clientes. Já a camada de aplicação, ela prover uma interface de acesso aos recursos distribuídos.

Além da arquitetura do FLiMSy ser dividida em camadas propiciando um melhor entendimento da arquitetura e facilitando o processo de manutenção, FLiMSy apresenta em suas camadas, para satisfazer o [RF03], o uso dos *remoting patterns* (ver Capítulo 2). Como mencionado anteriormente, os *remoting patterns* são padrões testados e consolidados pela indústria e que tem como objetivo atender requisitos comuns presentes em sistemas *middleware*. Para cada requisito de *middleware* é necessário o uso de um ou mais padrões específicos a fim de atendê-lo. A arquitetura do FLiMSy fez uso de alguns padrões de projeto tanto no lado do cliente, quanto no lado servidor para atender os seus requisitos. A Figura 3.1 apresenta a divisão dessas camadas pela arquitetura do FLiMSy.

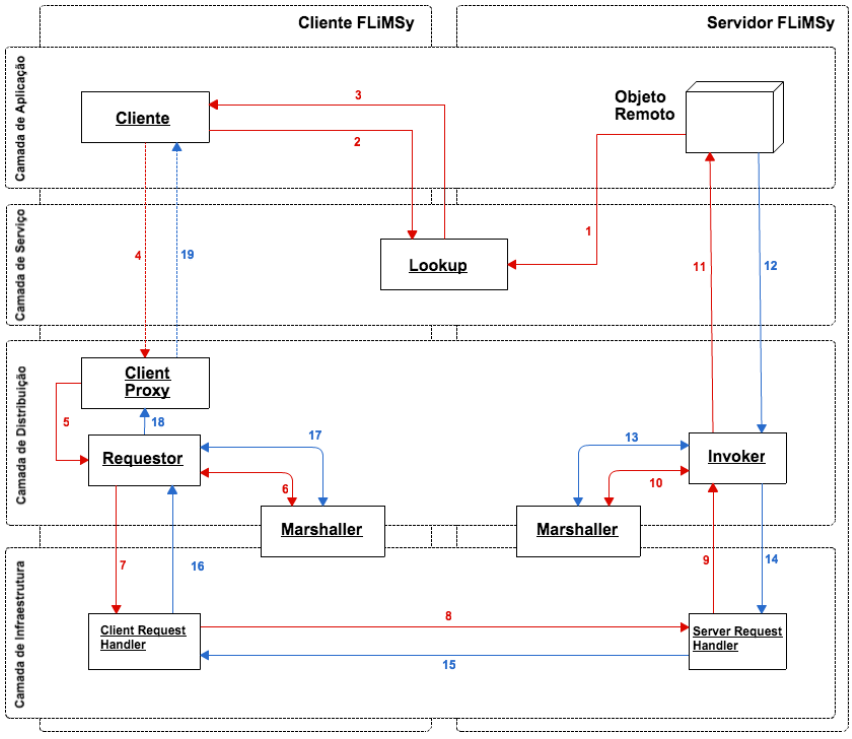

Figura 3.1: Arquitetura FLiMSy. **Fonte:** Elaborado pelo autor.

A comunicação entre os padrões que compõem a arquitetura do FLiMSy é baseada em RPC, satisfazendo o requisito [RF07]. O processo inicial de invocação remota é feito pelo servidor que o faz registando o seu objeto remoto no *Lookup* (1). Em seguida, o cliente (2) faz uma pesquisa ao *Lookup* fornecendo como parâmetro o nome do objeto remoto a ser invocado.

O padrão *Lookup* (3) envia como resposta para o cliente uma instância de *ClientProxy* que será usada para invocar os métodos desejados do objeto remoto. A seguir, o cliente (4) faz uso do *ClientProxy* para realizar a chamada remota.

Em seguida, o *ClientProxy* (5) encaminha os dados recebidos para o *Requestor* (6) que por meio do padrão *Marshaller* os converte em uma sequência de bytes. O *Requestor* (7) invoca o padrão *ClientRequestHandler* informando a localização do objeto remoto desejado e a sequência de bytes gerada pelo *Marshaller*. Depois disso, o *ClientRequestHandler* (8) estabelece um canal de comunicação TCP com o *ServerRequestHandler* e encaminha os bytes para ele.

Ao receber os bytes, o *ServerRequestHandler* (9) invoca o *Invoker* e os passa por parâmetro. O *Invoker* (10) faz uso do *Marshaller* para converter a sequência de bytes recebida em dados. Esses dados vão ser utilizados pelo *Invoker* para criar o objeto remoto. Após isso, o *Invoker* (11) faz a invocação da operação solicitada pelo cliente no objeto remoto criado.

O resultado da operação realizada pelo objeto remoto é repassado para o *Invoker* (12) que por meio do *Marshaller* (13) o converte em fluxo de bytes. Esses bytes são encaminhados do *Invoker* (14) para o *ServerRequestHandler*. Após receber os dados encaminhados, o *ServerRequestHandler* (15) os envia para o *ClientRequestHandler* pelo canal de comunicação TCP/IP.

O *ClientRequestHandler* (16) encaminha os bytes recebidos do *ServerRequestHandler* para o *Requestor*. Este (17), por sua vez, usa o *Marshaller* para transformar os bytes recebidos em dados. Em seguida, o *Requestor* (18) invoca o *ClientProxy* passando como parâmetro os dados. Por fim, os dados recebidos pelo *ClientProxy* (19) são encaminhados para o cliente.

Esse ciclo se repete pra cada chamada remota realizada pelo cliente. Porém, as fases que compõe o processo de localização, passos (2 e 3), só acontecem uma vez, caso o mesmo Cliente realize várias invocações para o mesmo objeto remoto.

3.3 Projeto

O lado cliente do FLiMSy implementa os padrões *ClientProxy*, *Marshaller*, *Requestor* e *ClientRequestHandler*, enquanto o lado servidor tem os padrões *Invoker*, *Marshaller* que possui o mesmo comportamento do usado no cliente e *ServerRequest Handler*. Eles serão descritos a seguir.

Com o objetivo de atender o requisito [RF07] que trata de transparência de acesso foi implementado na arquitetura do FLiMSy o padrão *ClientProxy*. Ele fornece para os clientes do FLiMSy uma interface contendo todos os métodos que serão acessados remotamente.

Os clientes do FLiMSy apenas precisam interagir com o *ClientProxy* quando desejam realizar invocações aos métodos do objeto remoto. O *ClientProxy* utiliza o padrão *Requestor* para acessar o objeto remoto de um servidor do FLiMSy que esteja disponível na rede. Isso porque os clientes fazem uso do *ClientProxy* como uma interface local cuja implementação está disponível em um objeto remoto na rede. Com o *ClientProxy*, o cliente invoca métodos remotos como se estivesse invocando-os localmente. Todo processo de comunicação do *ClientProxy* é mostrado na Figura 3.2.

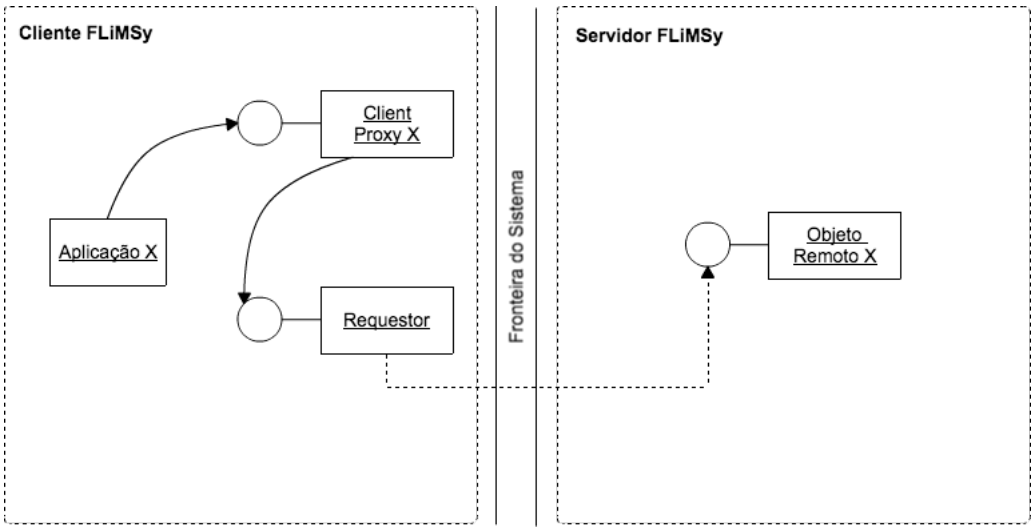

Figura 3.2: ClientProxy. **Fonte:** Elaborado pelo autor.

O processo de comunicação do *ClientProxy* é feito, conforme Figura 3.2: A *Aplicação X* realiza, por meio do *ClientProxy*, uma chamada remota de forma semelhante a uma local; Em seguida, o *ClientProxy* repassa a chamada remota para o *Requestor* que se encarrega de executar a chamada desejada no objeto remoto.

O padrão *Marshaller* foi implementado na arquitetura do FLiMSy a fim de satisfazer o

requisito [RF02] que trata do processo de serialização de dados. Este padrão é responsável por transformar diferentes tipos de objetos em sequência de bytes, tais como: *Long*, *String*, *Float*, *Double* e *Object*.

O *Marshaller* é um padrão que está presente não só no lado do cliente, como também no lado do servidor. O lado do cliente invoca o *Marshaller* para o processo de conversão de dados, por meio do padrão *Requestor*. Já o lado do servidor, faz uso do *Invoker* pra realizar esse mesmo trabalho. Um exemplo de funcionamento do padrão *Marshaller* é mostrado pela Figura 3.3.

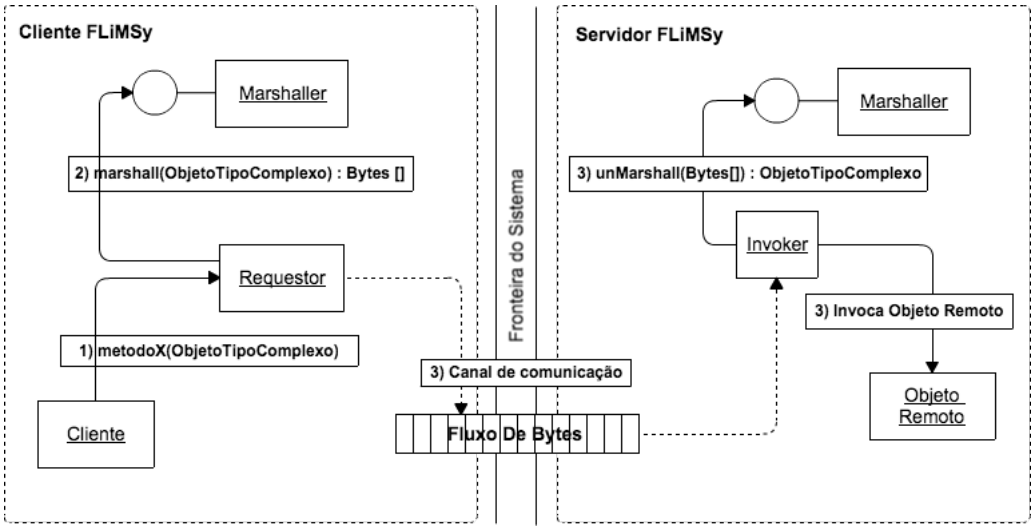

Figura 3.3: Marshaller. **Fonte:** Elaborado pelo autor.

Neste exemplo, o cliente do FLiMSy requisita uma operação remota invocando o *Requestor*. Em seguida, o *Requestor* converte o objeto da requisição feita em uma sequência de bytes utilizando o padrão *Marshaller*. Após converter os dados, o *Requestor* os envia até o *Invoker* presente no lado do servidor por meio do canal de comunicação.

O *Invoker* converte os bytes recebidos em objeto usando o *Marshaller*. O objeto é executado pelo *Invoker* e a sua resposta é convertida em fluxo de bytes pelo *Marshaller*. Os bytes gerados são encaminhados pelo *Invoker* até o *Requestor* presente no lado do cliente. O *Requestor* usa *Marshaller* para converter os bytes da resposta em objeto e o entrega para o cliente.

Já o padrão *Requestor* está localizado no lado do cliente e é usado para ocultar dos clientes do FLiMSy detalhes de comunicação necessários para invocar um objeto distribuído, atendendo com isso o [RF05] que trata sobre transparência de acesso. O *Requestor* faz uso de algumas informações, como endereço IP e porta, para estabelecer comunicação com o processo que contém o objeto remoto que se pretende invocar.

Além dos dados utilizados no processo de comunicação, o *Requestor* também é respon-

sável por enviar os dados da requisição como nome e identificador do objeto remoto, nome da operação solicitada e lista de argumentos. O funcionamento desse padrão é mostrado pela Figura 3.4.

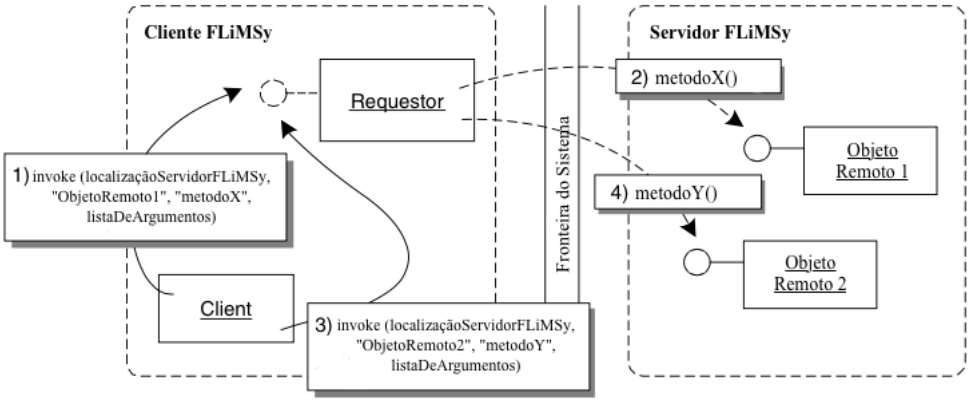

Figura 3.4: Requestor. **Fonte:** Elaborado pelo autor.

Na Figura 3.4, o cliente realiza a invocação dos métodos *X* e *Y* presentes nos objetos remotos 1 e 2 respectivamente. Para isso, o cliente solicita ao *Requestor* a invocação dos métodos *X* e *Y*.

O *Requestor* por meio dos dados de localização do servidor remoto (IP e porta) faz uma chamada ao *Invoker* passando como parâmetro os dados da requisição (nome e identificador do objeto remoto, nome da operação remota pretendida e sua lista de argumentos). O *Invoker* recebe a solicitação do cliente, instancia o objeto remoto e realiza a operação desejada. Basicamente esse padrão é usado para fazer com que a requisição chegue ao objeto remoto especificado, seja processado e retorne o resultado da operação para o cliente.

Os padrões *ClientRequestHandler* e *ServerRequestHandler* foram adicionados à arquitetura do FLiMSy para atender aos requisitos [RF01] que trata sobre comunicação confiável e [RF06] que trata sobre controle de concorrência. O padrão *ClientRequestHandler* está presente no lado do cliente, enquanto o *ServerRequestHandler* fica localizado no lado do servidor. Ambos os padrões são responsáveis por estabelecer uma comunicação entre um cliente e um servidor por meio de *socket* Transmission Control Protocol (TCP) e por meio dela trafegar as requisições dos clientes, em forma de *bytes*, até o servidor e vice-versa atendendo o requisito [RF01].

Como apresentado na Seção 2.3, Scala é uma linguagem de programação funcional que possui nativamente o suporte ao modelo de atores. Nesse modelo, os Atores trocam informações entre processos por meio de mensagens. O modelo de atores de Scala faz uso de algumas características bastante convenientes para programação paralela como ausência de dados

mutáveis. Se um dado é imutável, ele pode ser compartilhado ou copiado sem se preocupar com quem o está utilizando, ou seja, valores imutáveis são naturalmente *thread-safe* evitando com isso *deadlocks* e condições de corrida. *Thread-safe* [1] é um mecanismo de proteção usado para evitar que objetos mutáveis sejam modificados por múltiplas *threads* simultaneamente.

A fim de tratar um maior número de requisições e realizar o requisito [RF06], o *Server-RequestHandler* cria um Ator para cada solicitação do cliente. O funcionamento desses padrões é apresentado pelas Figuras 3.5 e 3.6.

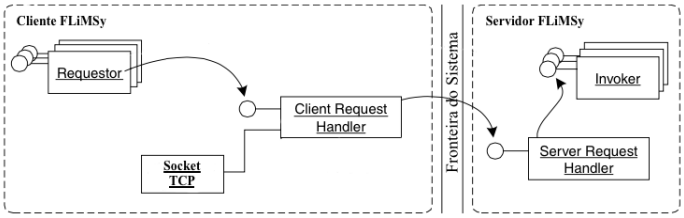

Figura 3.5: ClientRequestHandler. **Fonte:** Elaborado pelo autor.

A Figura 3.5 mostra o funcionamento do padrão *ClientRequestHandler*. Primeiramente o cliente faz uma solicitação ao padrão *Requestor* informando para ele o nome da operação desejada. O *Requestor* realiza uma chamada ao *ClientRequestHandler* informando os dados, como IP, porta, nome do objeto remoto, identificador do objeto remoto, nome da operação que o cliente deseja solicitar, lista de parâmetros da operação solicitada, que são necessários para invocar a chamada remota ao servidor.

O *ClientRequestHandler* estabelece uma comunicação com o servidor por meio de um *Socket* TCP e em seguida envia os bytes da requisição por meio do canal de comunicação com servidor. O *ClientRequestHandler*, ao receber a resposta do servidor, fecha o *Socket* usado durante o processo de comunicação e envia os *bytes* da resposta vinda do servidor para o *Requestor* que é responsável por processar esses dados e enviá-los para o cliente.

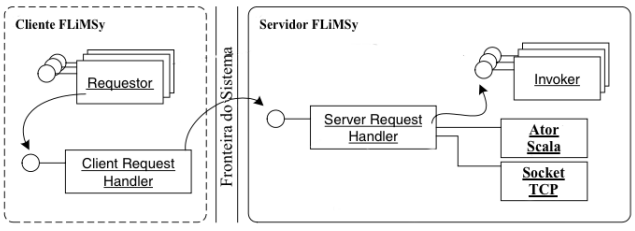

Figura 3.6: ServerRequestHandler. **Fonte:** Elaborado pelo autor.

Com relação à Figura 3.6, ela mostra o funcionamento do *ServerRequestHandler*. O

[1] `http://publib.boulder.ibm.com/html/as400/ic2924/info/RZAHWM40.HTM`

cliente faz uma solicitação de invocação a um objeto remoto no servidor ao *Requestor*. O *Requestor* encaminha os *bytes* desta solicitação para o *ClientRequestHandler* que se encarrega de estabelecer uma comunicação, por meio de *socket* TCP, com *ServerRequestHandler* e envia esses dados para ele. Os *bytes* são recebidos por um Ator criado pelo *ServerRequestHandler* e encaminhados para o *Invoker* que se encarrega de realizar a operação solicitada. Após isso, o *ServerRequestHandler* recebe a resposta do *Invoker* e a encaminha, em forma de *bytes*, para o *ClientRequestHandler*.

O FLiMSy, no lado servidor, faz uso do padrão *Invoker* para invocar as operação aos objetos remotos solicitados pelos clientes. Para isso, ele usa os dados vindos nas solicitações feitas pelos clientes. Os dados presentes nessas solicitações são compostos por: identificador do objeto remoto; nome do objeto remoto; nome da operação que se deseja executar; lista de parâmetros pertencentes a operação. O *Invoker* utiliza esses parâmetros para realizar chamadas as operações do objeto remoto pretendido. A Figura 3.7 demonstra como esse padrão funciona.

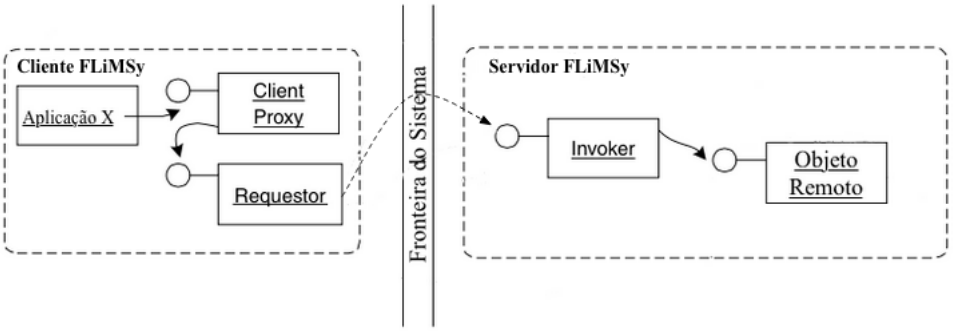

Figura 3.7: Invoker. **Fonte:** Elaborado pelo autor.

A Figura 3.7 mostra como o *Invoker* executa uma chamada remota por meio de uma solicitação do cliente. Primeiramente o cliente faz uma chamada local a uma operação presente no *ClientProxy*. Em seguida, o *ClientProxy* encaminha os dados da solicitação do cliente para o *Requestor* que realiza uma chamada ao *Invoker* passando os dados necessários para realizar a chamada remota. Esses dados são utilizados pelo *Invoker* para instanciar o objeto remoto e depois invocar a operação desejada. O *Invoker* utiliza tanto o parâmetro nome do objeto remoto quanto o identificador do objeto remoto para diferenciar as diversas solicitações realizadas pelos clientes ao mesmo objeto remoto.

Por fim, para atender o requisito [RF04] que trata sobre transparência de localização, foi criado um serviço de nomes com base no padrão *Lookup* (Markus Volter & Zdun, 2005). Esse padrão é utilizado para gerenciar as referências aos objetos remotos. O serviço de nomes usa uma

estrutura de dados em memória do tipo *Map*, que armazena uma chave e um valor, para gravar as informações de localização de um servidor na rede (IP e porta). O campo chave da estrutura de dados é representado por um "String"referente ao nome do objeto remoto. Já o campo valor da estrutura de dados, se refere a um tipo de dado que possui os dados de localização (IP e porta) do servidor no qual está presente o objeto remoto. A Figura 3.8 mostra como o cliente obtém a referência do objeto remoto.

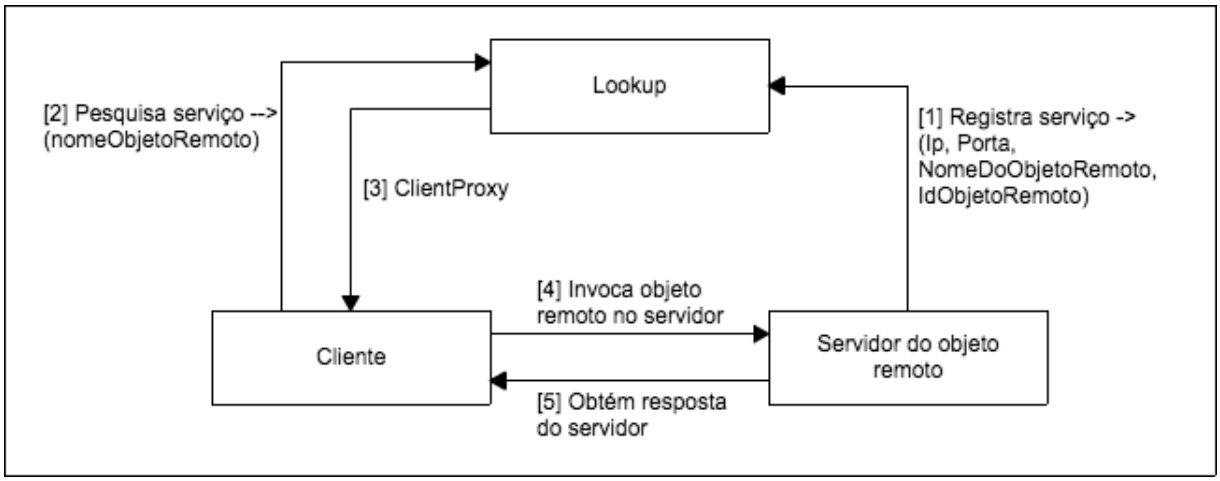

Figura 3.8: Processo de invocação de um objeto remoto no FLiMSy. **Fonte:** Elaborado pelo autor.

Primeiramente o servidor ao ser iniciado se registra no serviço de *Lookup* informando os seguintes dados: IP, porta, nome do objeto remoto, identificador do objeto remoto (1). Esses dados são guardados na estrutura de dados em memória. Após isso, quando o cliente deseja invocar um objeto remoto, ele inicialmente faz uma consulta pelo nome do objeto remoto ao serviço de *Lookup* (2). Caso o nome do objeto exista, é retornado para o cliente um objeto contendo uma interface de acesso ao objeto no servidor (3). O cliente invoca o objeto remoto no servidor (4) por meio da interface. Por fim, o servidor responde a solicitação do cliente (5). A Tabela 3.1 consolida de forma sintetizada como cada requisito listado na Seção 3.1 foi atendido pelo FLiMSy.

Tabela 3.1: Matriz de rastreabilidade dos requisitos do FLiMSy

Requisitos	Atendido por
[RF01] Comunicação confiável	Pelos padrões *ClientRequestHandler* e *ServerRequestHandler*
[RF02] Serialização de dados	Padrão *Marshaller*
[RF03] Uso de padrões de projeto de *middleware*	Uso dos *remoting patterns* na concepção da arquitetura
[RF04] Transparência de localização	Serviço de *Lookup*
[RF05] Transparência de acesso	Padrão *Requestor*
[RF06] Controle de Concorrência	Modelo de atores utilizado pelo padrão *ServerRequestHandler*
[RF07] Comunicação baseada em RPC	Invocação dos objetos remotos

A Tabela 3.1 apresenta em sua estrutura duas colunas cuja primeira coluna representa os requisitos suportados pelo FLiMSy e a segunda coluna como esses requisitos foram atendidos por ele.

3.4 Implementação

Os componentes que compõem a arquitetura do FLiMSy foram implementados em Scala Versão 2.11.6. Também foi utilizado o *Eclipse* como ferramenta de desenvolvimento. Como mencionado anteriormente, a implementação do FLiMSy foi feita com base nos *remoting patterns*. Dentre os vários padrões usados, iremos detalhar o código fonte do *ClientRequestHandler* e do *ServerRequestHandler* que fazem parte da camada de infraestrutura e são responsáveis pelo transporte dos dados do FLiMSy. Também será apresentado o código dos padrões *Requestor* e *Invoker* que estão presentes no lado do cliente e do servidor respectivamente. Os padrões *Requestor* e *Invoker* são os mais importantes na arquitetura do FLiMSy, pois eles são responsáveis por realizar invocações as operações presentes nos objetos remotos. Além deles, também iremos detalhar o código responsável pela transformação de dados em fluxo de bytes que é representado pelo padrão *Marshaller*.

Primeiramente, iremos detalhar o código em Scala utilizado para implementar o padrão *Requestor*. Ele é um padrão presente no lado do cliente cujo objetivo é invocar um objeto remoto no servidor. O código referente a este padrão é representado pela Listagem 3.1.

```
1 class Requestor {
2   def execRemoteObject(request: Request): Any = {
3     var marshaller: IMarshaller = new Marshaller();
4     var clientRequestHandler: ClientRequestHandler = new
    ClientRequestHandler();
```

```
5
6       var inetAddress:InetAddress = InetAddress.getByName(request.ip);
7       var socket: Socket = new Socket(inetAddress, request.port);
8
9       var data: Array[Byte] = marshaller.marshall(request);
10
11        var result: Array[Byte] = clientRequestHandler.execRemoteObject(data,
        socket);
12
13        var response:Response = castFromAnyToResponse(marshaller.unMarshall(
      result));
14
15        response.result;
16    }
17
18    def castFromAnyToResponse(obj: Any): Response = obj match {
19      case obj: Response => obj
20      case _ => throw new ClassCastException
21    }
22  }
```

Listagem 3.1: Exemplo do padrão *Requestor* em Scala

Como apresentado na Listagem 3.1, a classe *Requestor* possui 2 métodos: *execRemote-Object()* e *castFromAnyToResponse()*. O método *castFromAnyToResponse()* , Linhas de 18 à 21, é usado para converter um objeto de um tipo genérico em um objeto do tipo *Response*. Já o método *execRemoteObject()*, Linhas de 2 à 16, é usado para realizar uma invocação ao objeto remoto presente no servidor. Analisando internamente, o método *execRemoteObject()* recebe por parâmetro um objeto do tipo *Request*. Esse objeto possui algumas variáveis que são usadas durante o processo de invocação remota realizada pelo *Requestor*. Essas variáveis são: nome do objeto remoto, identificador do objeto remoto, nome da operação do objeto remoto que se deseja executar e sua lista de parâmetros. Além disso, o objeto *Request* também possui variáveis usadas para estabelecer uma conexão com o servidor, como: IP e a porta do servidor que detém o objeto remoto. As variáveis IP e porta do objeto *Request* são usados para instanciar o objeto *socket* que será usado para se comunicar com o servidor, Linhas 6 e 7. Por fim, na Linha 11, o método *execRemoteObject()* do padrão *ClientRequestHandler* é chamado passando para ele os bytes da requisição convertidos pelo padrão *Marshaller*, Linha 9, e o objeto *socket* criado para se

comunicar com o servidor.

Em seguida, iremos observar o código em Scala utilizado para implementar o padrão *ClientRequestHandler*. Ele é usado para transportar os dados do cliente para o servidor. O código usado para implementar esse padrão pode ser visto na Listagem 3.2.

```
1  object ClientRequestHandler {
2
3    def execRemoteObject(data: Array[Byte], socket: Socket): Array[Byte] = {
4        var is:BufferedInputStream = new BufferedInputStream(socket.
     getInputStream());
5          var os:BufferedOutputStream = new BufferedOutputStream(socket.
     getOutputStream());
6
7      sendData(data, os);
8
9      var result: Array[Byte] = receiveDataFromServer(is);
10     closeConnection(socket, is, os);
11
12     result;
13   }
14
15   def sendData(data: Array[Byte], os:BufferedOutputStream) {
16       os.write(data, 0, data.length);
17       os.flush();
18   }
19
20   def receiveDataFromServer(is:BufferedInputStream): Array[Byte] = {
21     var byteOutput: ByteArrayOutputStream = new ByteArrayOutputStream();
22       var buffer: Array[Byte] = Array.ofDim[Byte](4096);
23
24       var len: Int = is.read(buffer);
25       while (len > 0) {
26         byteOutput.write(buffer);
27         if (is.available() > 0) {
28           len = is.read(buffer);
29         } else {
30               len = 0;
31           }
32       }
```

```
33
34        var resultData: Array[Byte] = byteOutput.toByteArray();
35        byteOutput.close();
36        resultData;
37    }
38
39    def closeConnection(socket: Socket, is:BufferedInputStream, os:
       BufferedOutputStream) {
40      if (is != null) {
41        is.close();
42      }
43
44      if (os != null) {
45        os.close();
46      }
47
48        if (socket != null) {
49          socket.close();
50        }
51    }
52 }
```

Listagem 3.2: Exemplo do padrão *ClientRequestHandler* em Scala

Na Listagem 3.2, Linha 1, a classe *ClientRequestHandler* é definida como um objeto por meio da palavra reservada *object*. Isso é um comportamento de Scala que permite que haja apenas uma instância desta classe. Esta classe possui 4 métodos, são eles: *sendData()*, *receiveDataFromServer()*, *execRemoteObject()* e *closeConnection()*.

O método *sendData()* é usado para escrever os dados da requisição do cliente, convertida em fluxo de bytes, no objeto de escrita do *socket*. Já o método *receiveDataFromServer()* é responsável por ler a resposta do servidor por meio do objeto de leitura do *socket* ((*BufferedInputStream*)). Ao analisarmos internamente este método, na Linha 21 é criada a variável *byteOutput* que é responsável por armazenar todos os bytes da resposta do servidor. A Linha 22 cria um *buffer* de 4K. Esse *buffer* será usado para preencher a variável *byteOutput*. As Linhas de 24 à 32 são usadas para preencher variável *byteOutput* com a resposta vinda do servidor.

Como o tamanho da resposta vinda do servidor pode ser grande, o processo de leitura é feito de forma paginada em blocos de informação de no máximo 4K, tamanho do *buffer* criado.

O tamanho do *buffer* é opcional e utilizou-se um tamanho de 4k por comodidade e por ser bastante usado por programadores durante o processo de leitura de dados. Os dados do buffer são escritos na variável *byteOutput* a cada iteração do laço até que não haja mais dado para ser lido. Na Linha 34 é criada uma variável chamada *resultData* que irá armazenar todos os bytes da variável *byteOutput*. Após isso, na Linha 35, é feita a chamada ao método *close()* do *byteOutput* para liberar o buffer de leitura dos dados. A Linha 36 retorna a leitura dos bytes da resposta do servidor.

Com relação ao método *closeConnection()*, Linhas de 39 a 51, ele é usado não só para liberar os objetos de leitura e escrita do *socket*, como também o próprio objeto *socket*. Não menos importante, o método *execRemoteObject()*, Linhas de 3 à 13, é usado para invocar o objeto remoto no servidor. Ao analisarmos este método, as Linhas 4 e 5 são usadas para obter acesso aos objetos de escrita e leitura do *socket*. A Linha 7 faz uma chamada ao método *sendData()* para escrever uma sequência de bytes no objeto de escrita do *socket*. Já na Linha 9, é criada uma variável chamada *result* para armazenar o resultado vindo da chamada ao método *receiveDataFromServer()*. Após isso, o método *closeConnection()* será chamado para liberar o *socket* e os seus objetos de leitura e escrita. Por fim, o conteúdo da variável *result* é retornado.

Em seguida, iremos ver o código em Scala utilizado para implementar o padrão *Marshaller*. A Listagem 3.3 apresenta como esse padrão foi implementado.

```
1  class Marshaller {
2
3    def marshall(obj: Any): Array[Byte] = {
4    var byteOutput: ByteArrayOutputStream = new ByteArrayOutputStream();
5    var stream: ObjectOutputStream = new ObjectOutputStream(byteOutput);
6
7    stream.writeObject(obj);
8    stream.flush();
9    stream.close();
10
11   var result:Array[Byte] = byteOutput.toByteArray();
12   byteOutput.close();
13   result;
14 }
15
16 def unMarshall(byteArray: Array[Byte]): Any = {
17   var byteInput: ByteArrayInputStream = new ByteArrayInputStream(
```

```
    byteArray);
18    var stream: ObjectInputStream = new ObjectInputStream(byteInput);
19
20    var result: Any = stream.readObject();
21    stream.close();
22    byteInput.close();
23    result;
24  }
25 }
```

Listagem 3.3: Exemplo do padrão *Marshaller* em Scala

O código referente ao *Marshaller* pode ser observado na Listagem 3.3. Nele há a presença dos métodos *marshall* e *unMarshall*. O primeiro recebe como parâmetro um objeto de um tipo genérico e converte-o para um fluxo de bytes. Já o segundo recebe como parâmetro um fluxo de bytes e converte-o para um objeto genérico. No método *marshall()*, Linhas 4 e 5, são criados os objetos *ByteArrayOutputStream* e *ObjectOutputStream*.

O objeto *ObjectOutputStream* é responsável por transformar um objeto de um tipo genérico em um fluxo de bytes: Integer, Long, Float e String. Após isso, esses bytes são escritos no objeto *ByteArrayOutputStream*, conforme Linhas de 7 a 9. Ao terminar a escrita dos bytes, o conteúdo do objeto *ByteArrayOutputStream* é repassado para variável *result*. Depois, Linha 12, o método *close* do objeto *ByteArrayOutputStream* é chamado para liberá-lo da memória. A Linha 13 retorna o valor da variável *result* como resposta à invocação do método.

Já o método *unMarshall()*, nas Linhas 16 e 24, é usado para transformar uma sequência de bytes que foi passada por parâmetro para ele em um objeto de um tipo genérico. No interior deste método, há a criação dos objetos *ByteArrayInputStream* e *ObjectInputStream*, Linhas 17 e 18. O objeto *ByteArrayInputStream* lê a sequência de bytes assada por parâmetro para o método *unMarshall* e em seguida repassa os bytes lidos para o objeto *ObjectInputStream* para que ele converta-os em objeto, Linha 20. Após o processo de conversão de um fluxo de bytes para um objeto do tipo genérico, os objetos *ByteArrayInputStream* e *ObjectInputStream* são liberados da memória, Linhas 21 e 22, e o resultado do processo de conversão é retornado, Linha 23.

O *Invoker* é usado para invocar um objeto remoto no servidor dada uma solicitação do cliente. A Listagem 3.4 apresenta o código usado para implementar esse padrão.

```
1 object Invoker {
2     var echo: Echo = null;
3     var marshaller: IMarshaller = null;
```

```
4     var response: Response = null;
5
6   def execRemoteObject(data: Array[Byte]): Array[Byte] = {
7       if (marshaller == null) {
8         marshaller = new Marshaller();
9       }
10
11      var requestObject = castFromAnyToRequest(marshaller.unMarshall(data))
      ;
12
13     if (requestObject.remoteObjectName == "Echo" && requestObject.
      remoteObjectId == 1) {
14        if (echo == null) {
15          echo = new Echo();
16        }
17
18      if (requestObject.remoteObjectOperationName == "msg") {
19         if (response == null) {
20           response = new Response();
21         }
22           response.result = echo.msg(requestObject.
      remoteObjectArgumentList(0).asInstanceOf[String]);
23      }
24     }
25     marshaller.marshall(response);
26   }
27
28   def castFromAnyToRequest(obj: Any): Request = obj match {
29     case obj: Request => obj
30     case _ => throw new ClassCastException
31   }
32 }
```

Listagem 3.4: Exemplo do padrão *Invoker* em Scala

No código apresentado pela Listagem 3.4, Linha 1, a classe *Invoker* é definida como um objeto por meio da palavra reservada *object*. A classe *Invoker* possui 2 métodos: *execRemoteObject()* e *castFromAnyToRequest()*.

Inicialmente analisaremos o método *execRemoteObject()*. O código das Linhas de 7 à 9

é usado para criar uma instância do objeto *Marshaller*, caso ele não exista. O objeto *Marshaller* será usado para converter bytes em objeto e vice-versa. O código da Linha 11 usa o método *unMarshall()* do objeto *Marshaller* para converter os bytes da requisição do cliente em um objeto genérico do tipo *Any* em Scala. Após isso, o objeto genérico é convertido para o tipo *Request* pelo método *castFromAnyToRequest()* e armazenado na variável *requestObject*.

O objeto *Request* possui um conjunto de variáveis que serão utilizadas para instanciar o objeto remoto solicitado pelo cliente. Essas variáveis correspondem ao nome e identificador do objeto remoto solicitado, nome da operação desejada e sua lista de parâmetros. O código das Linhas de 13 a 24 é utilizado para criar uma instância do objeto remoto, executar a operação desejada no objeto remoto, criar um objeto do tipo *Response* que será responsável por encapsular o resultado da invocação do objeto remoto.

O objeto *Response* possui apenas uma variável chamada *result* que será usada para guardar o resultado da invocação do objeto remoto no servidor. Após isso, o objeto *Response* será convertido em fluxo de bytes pelo método *marshall()* do objeto *Marshaller* e em seguida esses bytes são retornados para quem invocou o método *execRemoteObject()*. Com relação ao método *castFromAnyToRequest()*, Linhas de 28 à 31, ele é usado para converter um objeto do tipo genérico *Any* em Scala para o objeto do tipo *Request*.

Assim como o *ClientRequestHandler*, o *ServerRequestHandler* também é usado para realizar o transporte de dados do servidor até o cliente e vice-versa. Porém, o *ServerRequestHandler* fica localizado no lado do servidor ao contrário do *ClientRequestHandler* que fica no cliente. Além disso, ele cria Atores para atender as solicitações dos clientes. O código utilizado para implementar esse padrão pode ser visto na Listagem 3.5.

```
1  case class ManagerSocket(socket: Socket)
2
3  class ServerRequestHandler extends Actor {
4
5    def callInvoker(data: Array[Byte]) : Array[Byte] = {
6      Invoker.execRemoteObject(data);
7    }
8
9    @Override
10   def act() {
11     while (true) {
12       receive {
```

```
13        case ManagerSocket(socket) =>
14          processRequestFromClient(socket)
15      }
16    }
17  }
18
19  def processRequestFromClient(socket: Socket) {
20    var is: BufferedInputStream = new BufferedInputStream(socket.
    getInputStream());
21    var os: BufferedOutputStream = new BufferedOutputStream(socket.
    getOutputStream());
22
23    var data: Array[Byte] = callInvoker(readRequestFromClient(is));
24      writeResponseToClient(data, os);
25      closeConnection(socket, is, os);
26  }
27
28  def readRequestFromClient(is: BufferedInputStream) : Array[Byte] = {
29    var byteOutput: ByteArrayOutputStream = new ByteArrayOutputStream();
30      var buffer: Array[Byte] = Array.ofDim[Byte](4096);
31
32      var len: Int = is.read(buffer);
33      while (len > 0) {
34        byteOutput.write(buffer);
35        if (is.available() > 0) {
36          len = is.read(buffer);
37        } else {
38              len = 0;
39          }
40      }
41
42      var resultData: Array[Byte] = byteOutput.toByteArray();
43      byteOutput.close();
44      resultData;
45  }
46
47  def writeResponseToClient(data: Array[Byte], os: BufferedOutputStream) {
48    Thread.sleep(50);
49    os.write(data, 0, data.length);
```

```
50            os.flush();
51    }
52
53    def closeConnection(socket: Socket, is: BufferedInputStream, os:
       BufferedOutputStream) {
54      if (is != null) {
55        is.close();
56      }
57
58      if (os != null) {
59        os.close();
60      }
61
62        if (socket != null) {
63          socket.close();
64        }
65
66        this.exit()
67    }
68  }
```

Listagem 3.5: Exemplo do padrão ServerRequestHandler em Scala

O código do *ServerRequestHandler* da Listagem 3.5 mostra a presença de 2 classes, a classe *ManagerSocket* e a *ServerRequestHandler* que estende da classe *Actor* de Scala, Linhas 1 e 3. A classe *ManagerSocket* é usada para encapsular o objeto *socket* em uma mensagem que será lida pelo método *act()* do Ator.

Já a classe *ServerRequestHandler*, por estender da classe *Actor*, tem que obrigatoriamente fazer a sobrescrita do método *act()*. Esse método funciona de forma similar ao método *run()* quando desejamos usar *Threads* em Java. Ao analisarmos internamente o método *act()*, Linhas de 10 a 17, notamos a presença da palavra reservada *receive*. O *receive* é usado pelo modelo de Atores de Scala para criar uma nova *thread* para processar o método *processRequestFromClient()* para cada mensagem do tipo *ManagerSocket* que é recebida, conforme Linhas de 12 à 15.

Também é observado no método *act()* a presença de um laço infinito, Linha 11. Isso é usado para manter o Ator vivo durante o processo de escrita e leitura de informações feito entre o cliente e o servidor. Apesar de haver esse laço, Scala gerencia a execução do método *process-RequestFromClient()* permitindo que ele execute caso uma mensagem do tipo *ManagerSocket*

seja recebida. A mensagem do tipo *ManagerSocket* encapsula um objeto *socket* que será passado por parâmetro para o método *processRequestFromClient()*, Linha 14, para ser processado. Ao final do processamento do método *processRequestFromClient()*, Scala libera a *thread* utilizada pelo Ator.

Além do método *act()*, no *ServerRequestHandler*, há também a presença de mais 5 métodos, são eles: *callInvoker()*, *processRequestFromClient()*, *readRequestFromClient()*, *writeResponseToClient()* e *closeConnection()*. O método *closeConnection()*, Linhas de 53 à 67, de forma similar ao mesmo método usado pelo *ClientRequestHandler*, é usado para liberar os recursos utilizados no processo de leitura e escrita do *socket*. Com relação ao método *writeResponseToClient()*, ele recebe por parâmetro um fluxo de bytes referente à resposta do servidor para o cliente. Antes desses bytes serem escritos para o cliente, este processo é posto para dormir por um determinado intervalo de tempo em milissegundos (ms), Linha 48. Isso é feito pra fixar um tempo constante na resposta do serviço. Após o processo dormir, os bytes são escritos para o cliente, Linha 49.

Não menos importante, o método *receiveRequestFromClient()* é responsável por ler a solicitação do cliente por meio do objeto de leitura do *socket*. Ao analisarmos internamente este método, na Linha 29, é criada a variável *byteOutput* que é responsável por armazenar todos os bytes da solicitação do cliente. A Linha 30 cria um buffer de armazenamento de 4K. Esse buffer será usado para preencher a variável *byteOutput*. As Linhas de 32 à 40 são usadas para preencher variável *byteOutput* com os dados da solicitação do cliente.

Como o tamanho da solicitação do cliente pode ser grande, o processo de leitura é feito de forma paginada em blocos de informação de no máximo 4K, tamanho do buffer criado. Os dados do buffer são escritos na variável *byteOutput* a cada iteração do laço até que não haja mais dado para ser lido. Na Linha 42, é criada uma variável chamada *resultData* que irá armazenar todos os bytes de *byteOutput*. Após isso, na Linha 43, é feita a chamada do método *close()* do *byteOutput* para liberar o buffer de leitura dos dados. A Linha 44 retorna em forma de array de bytes a leitura da solicitação do cliente.

O método *callInvoker()*, presente nas Linhas de 5 à 7, recebe como parâmetro um fluxo de bytes da solicitação do cliente e os repassa para o *Invoker*, Linha 6, para que ele execute o objeto remoto. Já o método *processRequestFromClient()*, Linhas de 19 à 26, é usado pelos Atores para processar as solicitações dos clientes.

Este método recebe como parâmetro o objeto *socket* que está encapsulado na mensagem

do tipo *ManagerSocket* recebida pelo Ator. Dentro do método *processRequestFromClient()* , ao receber o objeto *socket*, são criados os objetos *BufferedInputStream* e *BufferedOutputStream*, Linhas 20 e 21, usados para ler e escrever os bytes do *socket* utilizado na comunicação do cliente com o servidor.

Os bytes da solicitação do cliente são lidos pelo método *readRequestFromClient()* e passados por parâmetro para o método *callInvoker()* para que ele invoque o objeto remoto, conforme Linha 23. Os bytes do resultado da chamada ao método *callInvoker()* são armazenados na variável *data*, Linha 23, e em seguida essa variável é passada como parâmetro para o método *writeResponseToClient()*, Linha 24, para que ele escreva os bytes da resposta para o cliente. Por fim, o método *closeConnection* é chamado para liberar os recursos utilizados durante o processo de leitura e escrita dos dados vindos pelo *Socket* de comunicação.

3.5 Exemplo de Uso do FLiMSy

Como exemplo de uso do FLiMSy, foi implementada uma calculadora. Para demostrar isso, primeiramente descreveremos o fluxo básico da invocação remota da operação de somar (*sum*). Em seguida, mostraremos o código fonte em Scala utilizado no desenvolvimento deste exemplo. A Figura 3.9 mostra o fluxo de execução feito pelo cliente ao invocar o método *sum()* do objeto remoto *Calculator*.

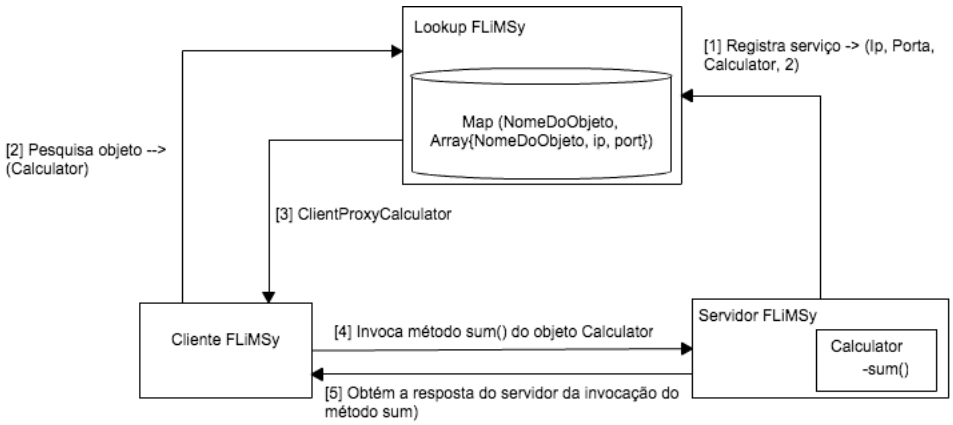

Figura 3.9: Exemplo de uma calculadora usando o FLiMSy. **Fonte:** Elaborado pelo autor.

Na Figura 3.9 é mostrado o fluxo utilizado para invocar o método *sum()* do objeto remoto *Calculator*. Primeiramente o servidor FLiMSy registra o objeto remoto *Calculator* no serviço de nomes Lookup FLiMSy (1). Para isso, o servidor FLiMSy precisa realizar uma invocação

ao método *register()* do *Lookup* FLiMSy passando os seguintes parâmetros: IP, porta, nome e identificador do objeto remoto *Calculator*. Em seguida, o *Lookup* armazena os dados de registro enviados pelo servidor em uma estrutura de dados do tipo *map* composta por chave e valor. No *map*, o campo chave será gravado com o nome do objeto remoto e no campo valor será gravado o objeto da requisição vinda do servidor FLiMSy durante o registro.

Para realizar a invocação do método *sum()* do objeto remoto *Calculator*, inicialmente o cliente FLiMSy faz um pesquisa pelo nome do objeto remoto *Calculator* no *Lookup* (2). O *Lookup* FLiMSy utiliza o nome do objeto passado por parâmetro durante a pesquisa feita pelo cliente para localizar o objeto remoto na estrutura de dados do tipo *map*. A pesquisa no *map* é feita comparando o seu campo chave com o nome do objeto remoto informado pelo cliente FLiMSy. O objeto *ClientProxyCalculator* é retornado como resultado da pequisa, caso o nome do objeto remoto exista no *map* (3). O *ClientProxyCalculator* retornado na pesquisa possui a assinatura de todos os métodos do objeto remoto *Calculator* e é por meio dele que o método *sum()* é invocado pelo cliente FLiMSy (4). Por fim, o cliente recebe a resposta da invocação do método *sum()* do objeto remoto *Calculator* presente no servidor FLiMSy (5).

Com relação ao código fonte em Scala, usado para implementar este exemplo, poucas mudanças foram realizadas. No lado do servidor, foi criada a classe *Calculator* e modificado o *Invoker* para invocá-la. As Listagens 3.6 e 3.7 demonstram o código da classe *Calculator* criada e a modificação que foi feita no código do *Invoker* para suportá-la.

```
1 class Calculator {
2   def sum(x: Int, y: Int): Int = x + y;
3 }
```

Listagem 3.6: Classe calculator em Scala

A Listagem 3.6 apresenta a classe *Calculator* implementada em Scala. Nela há a presença do método *sum()* que recebe as variáveis *x* e *y* por parâmetro e retorna a soma delas, Linhas de 1 a 3.

Com relação as modificações feitas no *Invoker*, para que ele consiga invocar a classe *Calculator*, elas são apresentadas pela Listagem 3.7.

```
1 object Invoker {
2     var echo: Echo = null;
3     var calculator: Calculator = null;
4     var marshaller: IMarshaller = null;
5     var response: Response = null;
```

```
6
7   def execRemoteObject(data: Array[Byte]): Array[Byte] = {
8       if (marshaller == null) {
9         marshaller = new Marshaller();
10      }
11
12      var requestObject = castFromAnyToRequest(marshaller.unMarshall(data))
    ;
13      if (requestObject.remoteObjectName == "Calculator" && requestObject.
    remoteObjectId == 2) {
14        if (calculator  == null) {
15          calculator  = new Calculator();
16        }
17
18        if (response == null) {
19          response = new Response();
20        }
21
22        var x:Int = requestObject.remoteObjectArgumentList(0).asInstanceOf[
    Int];
23        var y:Int = requestObject.remoteObjectArgumentList(1).asInstanceOf[
    Int];
24
25        if (requestObject.remoteObjectOperationName == "sum") {
26            response.result = calculator.sum(x, y);
27        }
28     }
29     marshaller.marshall(response);
30  }
31
32  def castFromAnyToRequest(obj: Any): Request = obj match {
33    case obj: Request => obj
34    case _ => throw new ClassCastException
35  }
36 }
```

Listagem 3.7: Exemplo do padrão *Invoker* em Scala

Na Listagem 3.4, os códigos das Linhas 13 a 24 foram substituídos pelos códigos das Linhas 13 a 30 da Listagem 3.7. Essa mudança foi necessária para fazer com que o *Invoker* crie

uma instância da classe *Calculator* e execute o método especificado na requisição feita pelo cliente.

Já no lado do cliente, foi criada a classe *CalculatorClientProxy* usada para realizar uma invocação ao método *sum()* do objeto remoto *Calculator*. O código em Scala da classe *CalculatorClientProxy* é apresentado pela Listagem 3.8.

```
1  CalculatorClientProxy {
2    var request: Request = null;
3
4    def sum(x: Int, y: Int): Int = {
5      var requestor: Requestor = new Requestor();
6      if (request == null) {
7        createRequest("Calculator", "localhost", 9090);
8      }
9
10     request.remoteObjectArgumentList = Array.ofDim[Any](2);
11     request.remoteObjectArgumentList(0) = x;
12     request.remoteObjectArgumentList(1) = y;
13     request.remoteObjectOperationName = "sum";
14
15     requestor.execRemoteObject(request).asInstanceOf[Int];
16   }
17
18   def createRequest(remoteObjectName: String, ip: String, port: Int) {
19     request = new Request();
20     //network informations
21     request.ip = ip;
22     request.port = port;
23
24     //remote object informations
25     request.remoteObjectId = 2;
26     request.remoteObjectName = remoteObjectName;
27   }
28
29 }
```

Listagem 3.8: Exemplo de calculadora em Scala

No código da Listagem 3.8, há uma variável de classe chamada *request*, Linha 2. Essa variável é responsável por encapsular todas as informações que são necessárias, como: IP, porta,

nome e identificador do objeto, para invocar um objeto remoto. A inicialização da variável *request* é feita pela chamada ao método *createRequest()*, Linhas de 6 à 8. Também são configurados na variável *request* informações (nome do método e lista de argumentos) referentes ao método a ser invocado no objeto remoto *Calculator*. Por fim, o método *execRemoteObject()* de *Requestor* é invocado passando por parâmetro a variável *request*.

O objeto remoto do serviço de nomes *Lookup* implementa 2 métodos *register()* e *bind()*. O método *register()* é usado pelos servidores para registrar seus objetos remotos em uma estrutura de dados do tipo *map*. Já o método *bind()* é usado pelos clientes para pesquisar e obter uma instância de uma interface remota que será usada pelos clientes para invocar os métodos de um objeto remoto. O exemplo de código do objeto remoto presente em *Lookup* é vista na Listagem 3.9.

```
1  class Lookup {
2    def register(request: Request) {
3        var remoteObjectName: String = request.remoteObjectArgumentList(0).
     asInstanceOf[String];
4        if (!LookupTableList.lookupTable.contains(remoteObjectName)) {
5          LookupTableList.lookupTable.put(remoteObjectName, request.
     remoteObjectArgumentList);
6        }
7    }
8
9    def bind(remoteObjectName: String): Response = {
10       var response: Response = new Response();
11       response.result  = LookupTableList.lookupTable.get(remoteObjectName).
     get;
12       response
13   }
14
15   object LookupTableList {
16     val lookupTable = new HashMap[String, Array[Any]] with SynchronizedMap[
     String, Array[Any]]
17   }
18 }
```

Listagem 3.9: Exemplo do objeto remoto do serviço de *Lookup* presente no FLiMSy feito em Scala

Na Listagem 3.9, há não só a classe chamada *Lookup* que contém os métodos *register()* e

bind(), como também o objeto estático *LookupTableList* utilizado para gerenciar a estrutura de dados do tipo *map*. Na classe *Lookup* o método *register()*, Linhas de 2 à 7, recebe uma requisição do servidor por parâmetro contendo as informações (nome do objeto remoto, IP, porta) que serão gravadas no *map*. Para gravar essas informações no *map*, primeiramente checa-se a existência do nome do objeto na estrutura de dados. Caso não exista, as informações vindas na requisição são gravadas.

O método *bind()*, Linhas 9 a 13, possui o nome do objeto remoto a ser consultado na estrutura de dados. Esse método retorna um objeto do tipo *Response* que encapsula os dados de localização (nome do objeto remoto, IP, porta) do objeto remoto consultado.

Já o objeto *LookupTableList* possui uma variável chamada *lookupTable* que contém uma estrutura de dados sincronizada do tipo *map*. Essa estrutura de dados armazena 2 campos. O primeiro referente a chave que identifica o objeto remoto na estrutura sendo representado pelo nome do objeto remoto. E o segundo referente aos dados de localização do objeto remoto (nome do objeto remoto, IP, porta) que são gravados na estrutura de dados na forma de um *array* de tipos genéricos.

A invocação dos métodos *register()* e *bind* no Lookup é feita pela classe *LookupClientProxy*. O código dessa classe usada no exemplo da calculadora é mostrado na Listagem 3.10.

```
1  LookupClientProxy {
2      def register(remoteObjectName: String, ip:String, port:Int) {
3        var requestor: Requestor = new Requestor();
4
5      var request: Request = new Request();
6
7      //network informations
8      request.ip = "localhost";
9      request.port = 9080;
10
11     //remote object informations
12     request.remoteObjectId = 2;
13     request.remoteObjectName = "Lookup";
14     request.remoteObjectOperationName = "register";
15     request.remoteObjectArgumentList = Array.ofDim[Any](3);
16     request.remoteObjectArgumentList(0) = remoteObjectName;
17     request.remoteObjectArgumentList(1) = ip;
```

```
18      request.remoteObjectArgumentList(2) = port;
19
20      requestor.execRemoteObject(request);
21    }
22
23    def bind(remoteObjectName: String): Any = {
24       var requestor: Requestor = new Requestor();
25
26      var request: Request = new Request();
27
28      //network informations
29      request.ip = "localhost";
30      request.port = 9080;
31
32      //remote object informations
33      request.remoteObjectId = 2;
34      request.remoteObjectName = "Lookup";
35      request.remoteObjectOperationName = "bind";
36      request.remoteObjectArgumentList = Array.ofDim[Any](1);
37      request.remoteObjectArgumentList(0) = remoteObjectName;
38
39      var result:Array[Any] = requestor.execRemoteObject(request).
      asInstanceOf[Array[Any]];
40
41      var clientProxy:Any = null;
42      if (remoteObjectName == "Calculator") {
43           clientProxy = new CalculatorClientProxy();
44           clientProxy.asInstanceOf[CalculatorClientProxy].createRequest(
      result(0).asInstanceOf[String], result(1).asInstanceOf[String], result
      (2).asInstanceOf[Int]);
45      }
46      clientProxy;
47    }
48  }
```

Listagem 3.10: Exemplo de uma interface remota do serviço de *Lookup* presente no FLiMSy feito em Scala

Na Listagem 3.10, há a presença dos métodos *register()* e *bind()*. O método *register()* da classe *LookupClientProxy*, Linhas 2 a 21, recebe por parâmetro as variáveis *remoteObjectName*,

ip e *port* que se referem ao nome do objeto remoto, IP e porta do servidor onde está localizado o objeto remoto. O valor dessas variáveis são encapsuladas em um objeto do tipo *Request*, Linha 5 a 18. Após isso, o método *execRemoteObject()* do *Requestor* é executado passando como parâmetro os dados da requisição criada, Linha 20. O *Requestor* encaminha a requisição até chegar ao método *register()* do objeto remoto presente em Lookup FLiMSy. Ao chegar no Lookup FLiMSy os dados da requisição são gravados na estrutura de dados do tipo *map*.

Com relação ao método *bind()* da classe *LookupClientProxy*, ele recebe a variável *remoteObjectName* por parâmetro que se refere ao nome do objeto remoto que se pretende consultar no Lookup FLiMSy que no nosso exemplo é *"Calculator"*. Assim como no método *register()*, o método *bind()* também cria um objeto do tipo *Request* para encapsular os dados que serão usados durante o processo de invocação de um objeto remoto. Após criar a requisição, ela é passada por parâmetro para o método *execRemoteObject()* do *Requestor* que encaminha a requisição até o método *bind()* do objeto remoto no Lookup FLiMSy. Caso exista o nome do objeto remoto na estrutura de dados do tipo *map*, um array contendo os dados de localização (nome do objeto remoto, IP, porta) do servidor que contém o objeto remoto desejado é retornado e armazenado na variável *result*. Em seguida, os valores presentes na variável *result* são usados para criar a instancia da interface remota *CalculatorClientProxy* e retorna ela para o cliente.

3.6 Considerações Finais

Neste capítulo foram mostrados detalhes arquiteturais e de implementação usados na construção do FLiMSy. Inicialmente foram apresentados os requisitos desejáveis do *middleware* proposto. Em seguida, foram introduzidos detalhes da arquitetura, projeto e implementação do FLiMSy. Finalmente, para ilustrarmos o uso prático do *middleware*, foi apresentado o exemplo de uma calculadora distribuída implementada com o uso do FLiMSy.

4

Avaliação Experimental

Este capítulo descreve todo o processo de avaliação experimental do FLiMSy. Nele, serão mostrados as métricas e os parâmetros utilizados na avaliação. Em seguida, serão mostrados os passos necessários para executar o experimento. Por fim, os dados coletados durante o processo de avaliação experimental serão interpretados e apresentado os seus resultados.

4.1 Objetivos

Para avaliar o FLiMSy, foi realizada uma avaliação experimental com uma aplicação cliente-servidor. O objetivo da avaliação é entender o impacto do FLiMSy sobre o desempenho da aplicação. Para isto, a métrica utilizada na avaliação foi o *tempo de resposta* que é definido como o intervalo de tempo entre a requisição e a resposta fornecida pelo sistema. No experimento, foram considerados quatro parâmetros: número de clientes, tempo de serviço, número de invocações remotas e número de máquinas. Esses parâmetros foram utilizados a fim de avaliar o impacto deles sobre a métrica considerada. A Tabela 4.1 apresenta os parâmetros e os níveis utilizados pelo processo de avaliação experimental.

Tabela 4.1: Parâmetros e níveis de avaliação do FLiMSy

Parâmetros	**Níveis**
Número de clientes (C)	1, 50, 100
Tempo do serviço (TS)	10ms, 50ms, 100ms
Número de invocações remotas	5000
Número de máquinas	2

A técnica utilizada durante o processo de avaliação experimental do FLiMSy foi a medição. Essa abordagem é utilizada para avaliar sistemas já existentes por meio da técnica de observação direta. A carga de trabalho (*Workload*) selecionada foi um objeto do tipo "String". O

(*Workload*) corresponde ao tipo informação trocada entre o cliente e o servidor.

4.2 Experimentos

O experimento foi feito com base em uma aplicação cliente-servidor que trocava mensagens do tipo "String" funcionando como uma aplicação de eco. Para o experimento, três elementos foram utilizados (Cliente, Servidor e *Lookup*). Antes de computar os dados dos experimentos, um processo de aquecimento chamado *warmup* era iniciado com o objetivo de inicializar todas as variáveis que pertenciam ao contexto do experimento. O *warmup* descartava as 1500 amostras iniciais durante cada experimento. A interação básica entre esses serviços é mostrada pela Figura 4.1.

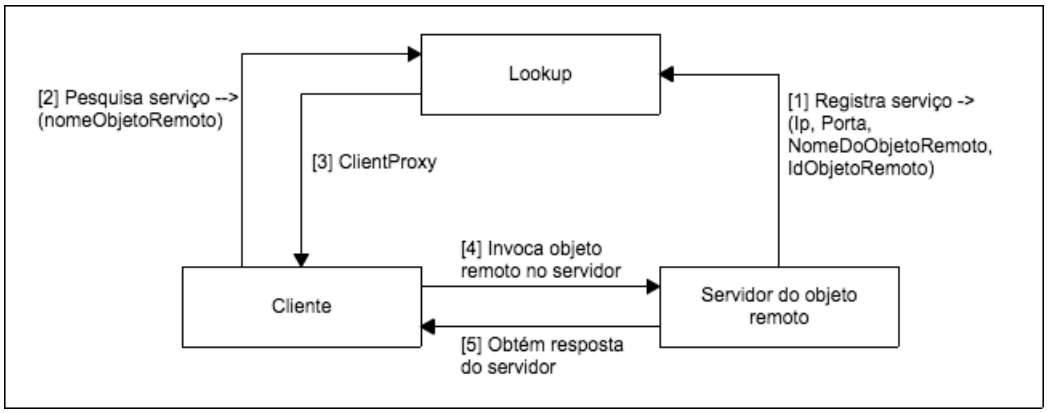

Figura 4.1: Processos presentes na avaliação experimental do FLiMSy. **Fonte:** Elaborado pelo autor.

O fluxo básico de interação entre os serviços apresentados na Figura 4.1 se inicia pelo servidor (1) que faz uma chamada ao serviço de *Lookup* informando para ele não só seus dados de localização, como ip e porta, como também o nome do objeto remoto que será usado como chave única para identificar a referencia ao objeto remoto em uma futura consulta ao serviço de *Lookup*. Em seguida, o cliente (2) faz uma consulta ao serviço de *Lookup* informando o nome do objeto remoto desejado. O serviço de *Lookup* procura na estrutura de dados persistente a referência ao objeto por meio do nome informado na solicitação do cliente. O serviço de *Lookup* retorna (3) uma interface de acesso ao objeto remoto para o cliente, caso o nome do objeto solicitado exista na estrutura de dados. O cliente (4), por meio da interface remota recebida como resposta pelo serviço de *Lookup*, realiza a invocação de um método do objeto remoto no servidor. O servidor (5) recebe a requisição do cliente, invoca o objeto localmente e retorna a reposta para o cliente.

O cliente só faz a pesquisa ao serviço de *Lookup*, enquanto não estiver de posse da interface do objeto remoto desejado. Durante o processo de avaliação experimental, cada cliente realizou 5000 chamadas ao servidor.

Para o experimento foram utilizadas 2 máquinas com as seguintes características: computador modelo MacBook Pro com 1 processador Intel Core i7 2.9 GHz com 2 núcleos, e memória 8 GB de RAM. Além disto, estavam instalados na máquina Scala Version 2.11.2, Java Version 1.7.0-60, Java (TM) SE Runtime Environment (build 1.7.0-60-b19) e Java HotSpot (TM) 64-Bit Server VM (build 24.60-b09, mixed mode). A disposição dos elementos utilizados para compor a infraestrutura necessária para realizar o experimento pode ser observada pela Figura 4.2.

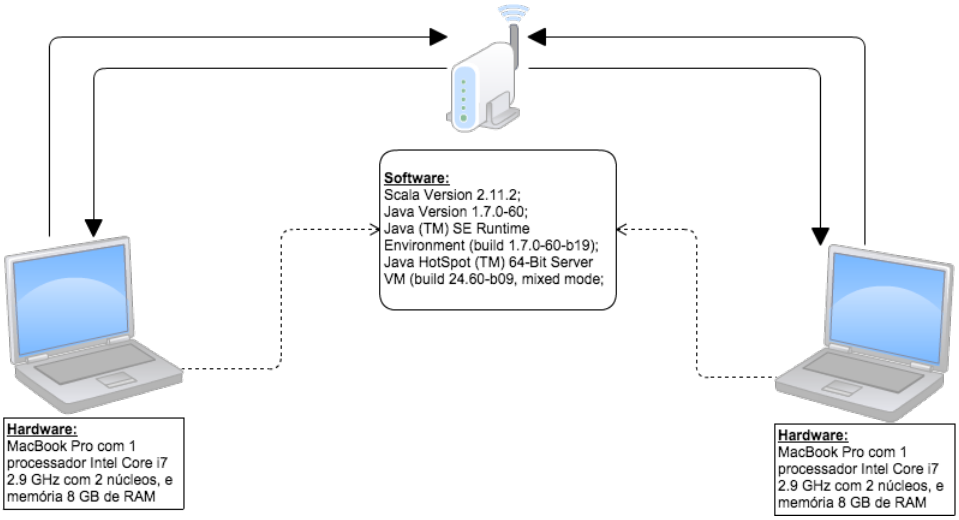

Figura 4.2: Infraestrutura utilizada pelo FLiMSy durante o processo de avaliação experimental. **Fonte:** Elaborado pelo autor.

Na Figura 4.2, as máquinas utilizadas durante o experimento trocavam informações por meio de uma rede *WiFi*. Já os serviços (Cliente, Servidor e Lookup) usados durante o experimento foram inciados nas 2 máquinas da seguinte forma. Em uma máquina estava localizado o cliente e em outra o servidor e o serviço de *Lookup*. Além disso, o cliente utilizou várias *threads* simulando diversas chamadas simultâneas ao servidor.

4.3 Resultados

Para o primeiro experimento fixou-se o valor do tempo de serviço em 10ms e variou-se o número de cliente em 1, 50 e 100. Para fixar os tempos de serviços foi adicionado um *sleep* com o tempo configurado para o experimento pretendido (10ms, 50ms ou 100ms) na função do servidor que escreve a reposta para cliente. Cada cliente realizou 5000 invocações remotas

ao servidor e a carga de trabalho utilizada foi o envio e o recebimento de um objeto do tipo "String" do cliente para o servidor e vice-versa. O resultado desse experimento é mostrado pelo gráfico da Figura 4.3

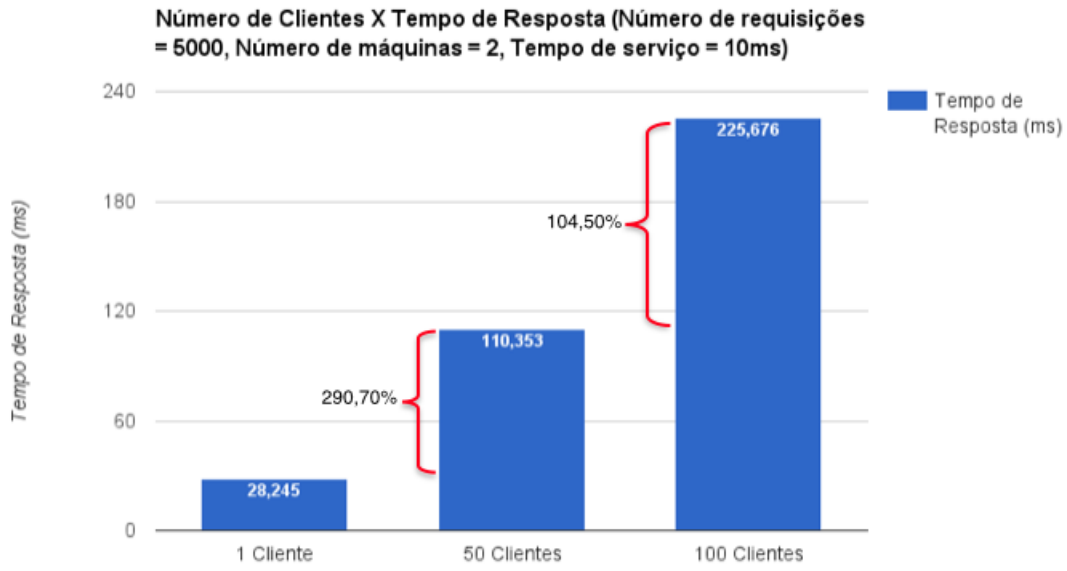

Figura 4.3: Gráfico de experimento do FLiMSycom tempo de serviço de 10ms. **Fonte:** Elaborado pelo autor.

O gráfico apresentado na Figura 4.3, apresenta uma tendência de crescimento nos tempos de respostas obtidos durante uma variação no parâmetro relacionado a número de clientes. Porém, esse crescimento não acontece na mesma proporção. Isso pode ser observado, ao compararmos o tempo de resposta de 1 cliente com o de 50 clientes. O tempo de resposta de um cliente foi de 28,245 (ms), já o tempo de 50 clientes foi de 110,353 (ms) e não de 1412,25 (ms) que seria o valor esperado, caso o tempo de resposta fosse diretamente proporcional a quantidade de clientes. Observa-se que houve um crescimento de 290,70% para 50 clientes em relação a 1 cliente. Esse comportamento também se repete ao compararmos o tempo de resposta de 50 clientes com o de 100 clientes. Nesse caso, o crescimento foi de 104,50% para 100 clientes em relação a 50. Pode ser observado também por meio disso, que houve um crescimento bem menor nos percentuais em comparação a análise anterior. Portanto, esse experimento mostrou que, à medida que há um aumento no número de clientes, o tempo de resposta do FLiMSy não cresce na mesma proporção, mostrando um certo grau de escalabilidade da implementação em Scala. A escalabilidade apresentada pelo FLiMSy mostra que ele tem a capacidade de suportar 50 clientes simultaneamente realizando 5000 requisições cada.

Já no segundo experimento, ele foi feito da mesma forma que o experimento anterior com a exceção de uma mudança. Nele foi alterado o valor do tempo de serviço de 10ms para 50ms. a Figura 4.4 mostra os valores obtidos por este experimento.

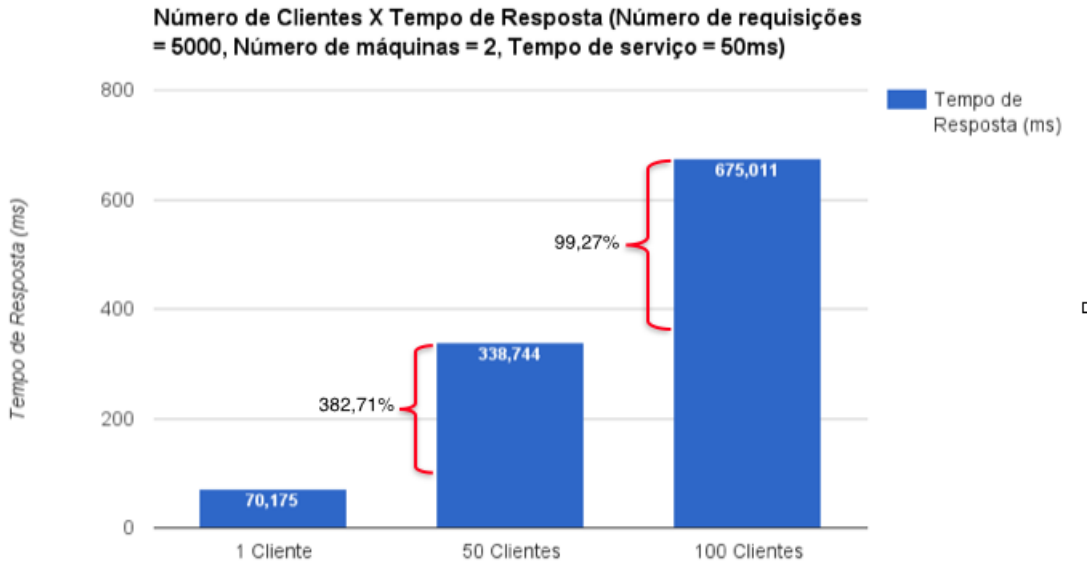

Figura 4.4: Gráfico de experimento do FLiMSy com tempo de serviço de 50ms. **Fonte:** Elaborado pelo autor.

O gráfico do segundo experimento apresentado na Figura 4.4 mostra um comportamento similar ao primeiro experimento. Nele também há uma tendência de crescimento quando há uma variação no número de clientes para um determinado tempo de processamento de serviço, mas esse crescimento não ocorre de forma proporcional. Nota-se, para um tempo de serviço de 50ms, crescimentos de 382,71% e 99,27% quando aumentamos o número de clientes em 50 vezes (1 para 50 clientes) e 2 vezes (50 para 100 clientes), respectivamente.

Com relação ao terceiro experimento, ele foi feito da mesma forma que os experimentos anteriores tendo como mudança apenas o valor do tempo de serviço que ficou fixado em 100ms. Os resultados obtidos por esse experimento são apresentados pela Figura 4.5.

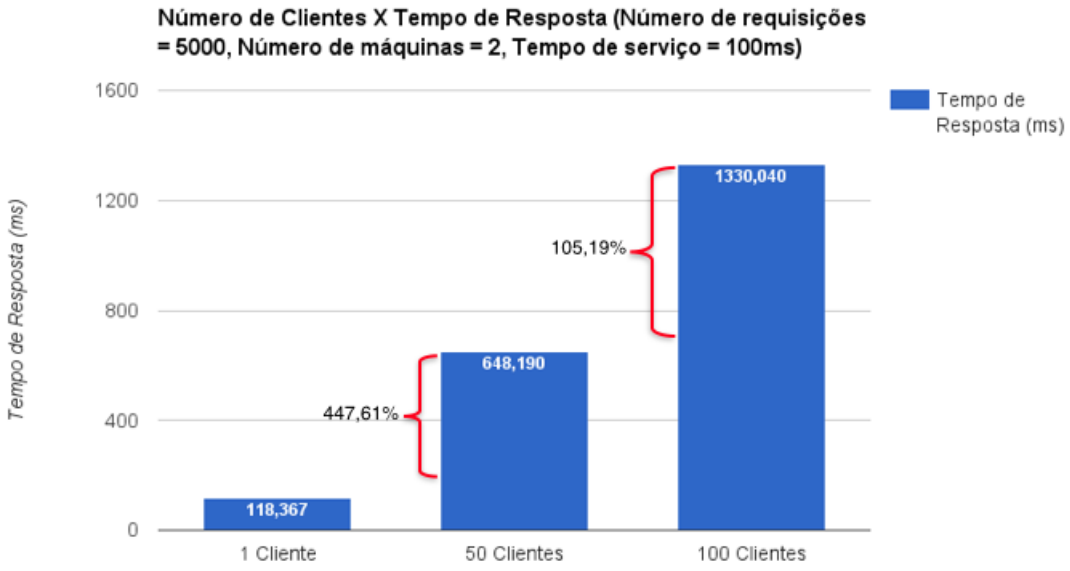

Figura 4.5: Gráfico de experimento do FLiMSy com tempo de serviço de 100ms. **Fonte:** Elaborado pelo autor.

Para o terceiro experimento, o gráfico apresentado pela Figura 4.5 exibiu o mesmo padrão de resultado mostrado pelo primeiro e segundo experimentos. Nesse experimento, notou-se um aumento de 447,61% quando o número de clientes é aumentado em 50 vezes (de 1 para 50 clientes), e um aumento de 105,19% quando dobramos o número de clientes (de 50 para 100 clientes).

Comparando-se os resultados dos três experimentos, é possível observar um aumento médio de 373,67% no tempo de resposta quando há uma aumento de 1 para 50 clientes, e um aumento de 102,99% quando duplicamos o número de clientes de 50 para 100. Isto mostra que o FLiMSy possui um certo grau de escalabilidade até que o número de clientes chegue a 50. Por fim, é importante observar que o impacto do *middleware* no tempo de resposta vai reduzindo a medida que o tempo de serviço aumenta: 64,60% (TS = 10ms), 28,75% (TS = 50ms) e 15,52% (TS = 100ms). Conforme mostrado pelo gráfico apresentado pela Figura 4.6.

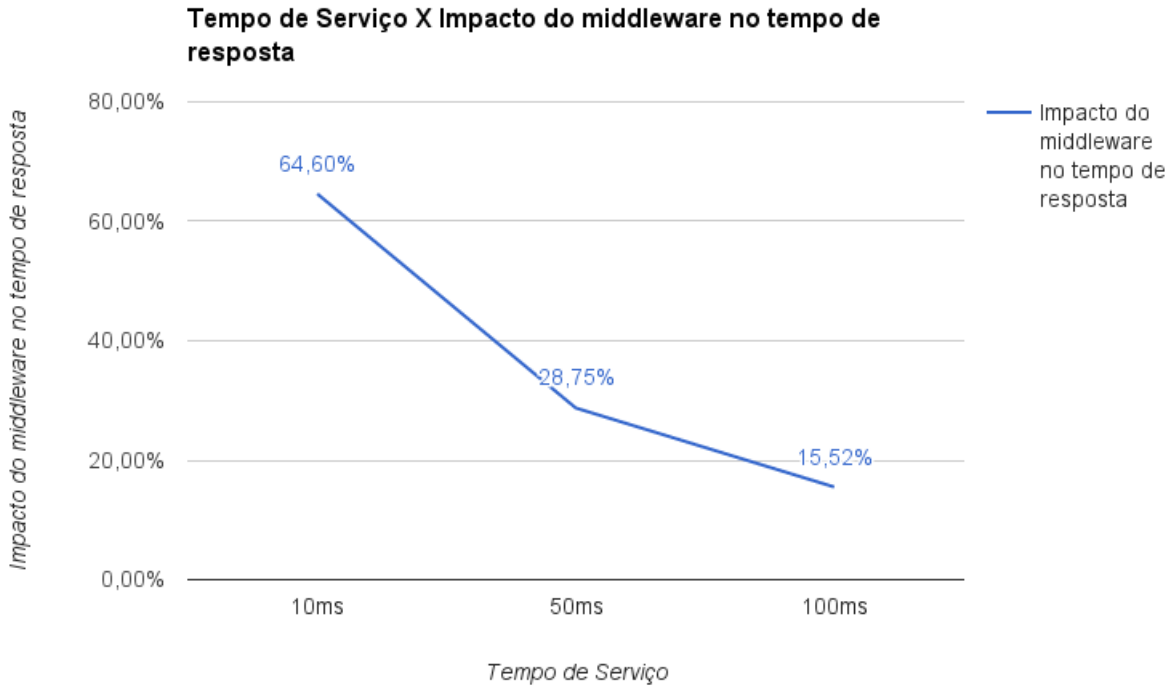

Figura 4.6: Gráfico de desempenho do FLiMSy. **Fonte:** Elaborado pelo autor.

O gráfico mostrado pela Figura 4.6 indica que o FLiMSy tem um melhor desempenho para suportar aplicações com tempo de serviços mais longos. Já com relação ao intervalo de confiança dos dados obtidos por meio da realização dos experimentos, eles podem ser observados nas Tabelas 4.2, 4.3 e 4.4.

Tabela 4.2: Intervalo de confiança do experimento do FLiMSy com tempo de serviço de 10ms.

Clientes	Tempo de Resposta	Desvio Padrão	Limite Inferior	Limite Superior
1	28,245	9,545	27,982	28,512
50	110,353	579,069	108,096	112,636
100	225,676	1208,610	222,355	229,056

Tabela 4.3: Intervalo de confiança do experimento do FLiMSy com tempo de serviço de 50ms.

Clientes	Tempo de Resposta	Desvio Padrão	Limite Inferior	Limite Superior
1	70,175	10,055	69,898	70,455
50	338,744	3,791	338,642	338,907
100	675,011	40,738	674,968	675,194

Tabela 4.4: Intervalo de confiança do experimento do FLiMSy com tempo de serviço de 100ms.

Clientes	Tempo de Resposta	Desvio Padrão	Limite Inferior	Limite Superior
1	118,367	7,354	118,163	118,571
50	648,190	45,787	648,074	648,433
100	1330,040	55,224	1330,022	1330,328

As Tabelas 4.2, 4.3 e 4.4 apresentam a média do tempo de resposta, o desvio padrão e o intervalo de confiança para as medições realizadas. O intervalo de confiança foi calculado utilizado o ambiente R (R Development Core Team, 2008) no nível de 95% (95% I.C) supondo uma distribuição normal das medições. Isso foi feito com o objetivo de validar a confiabilidade dos dados obtidos durante a realização da avaliação experimental do FLiMSy.

4.4 Considerações Finais

O FLiMSy foi submetido a um processo de avaliação experimental cujo objetivo era avaliar como métrica o seu tempo de resposta. Sobre essa métrica, o FLiMSy apresentou resultados compatíveis demostrando indícios de que ele atende as solicitações dos clientes em um tempo de resposta satisfatório. Isso foi confirmado, após a análise dos dados de 3 experimentos que apresentaram o mesmo padrão de resultado.

5

Trabalhos Relacionados

Este capítulo tem o objetivo de apresentar alguns sistemas *middleware*, tais como: *middleware* orientado a objetos, *middleware* orientado a mensagem, *middleware* baseado em espaço de tuplas e *middleware* funcional em Haskel, que se assemelham em algum aspecto com o *middleware* proposto.

5.1 *Middleware* Orientado a Objetos

Um *middleware* orientado a objetos tem como principal característica fazer com que processos acessem de forma transparente objetos que estão fisicamente distribuídos na rede como se estivessem acessando-os localmente. Este tipo de *middleware* se assemelha com o FLiMSy que é o *middleware* proposto por essa dissertação. Isso porque há varias características que estão presentes neste tipo de *middleware* que também estão presentes nos requisitos do FLiMSy, como exemplo: uso de serviço de nomes que prover o requisito de transparência de localização permitindo que objetos remotos sejam acessados como se eles estivessem sendo acessados localmente; uso do padrão *Marshaller* para converter dados em fluxo de dados e vice-versa satisfazendo o requisito de serialização de dados; e uso de interfaces remotas com as assinaturas das operações que estão disponíveis no objeto remoto a ser acessado.

O *middleware* orientado a objetos é uma evolução do *middleware* procedural. Processos que utilizam um *middleware* procedural fazem uso de uma tecnologia de comunicação entre processos chamada *Remote Procedure Call (RPC)* para realizar invocações a métodos remotos como se os invocasse localmente. *Middleware* orientado a objetos também fazem uso de RPC só que com os mecanismos do paradigma orientado a objetos, como herança, polimorfismo e encapsulamento.

Há diversas tecnologias baseadas em *middleware* orientado a objetos. Dentre elas, podemos citar: *Remote Method Invocation (RMI)* e *Common Object Request Broker (CORBA)*. RMI é uma plataforma Java para acesso a objetos distribuídos. O processo de invocação de objetos remotos é feito através de chamadas RPC entre aplicações desenvolvidas em Java (Cade & Roberts, 2002; Grosso, 2001). Por meio do uso de RMI, objetos que residem em uma JVM podem interagir com objetos residentes em outras JVM independente de localização. RMI é dependente da linguagem de programação Java, portanto aplicações que usam RMI para realizar invocações a objetos remotos precisam ser escritas em Java (Cade & Roberts, 2002; Grosso, 2001).

O processo de comunicação usando RMI é feito normalmente por uma aplicação cliente e outra servidora. Primeiramente o cliente realiza invocações remotas ao servidor. Em seguida, o servidor trata as requisições dos clientes e gera as respostas para eles. Todo esse processo de comunicação é feito de forma síncrona onde o cliente que realiza uma requisição a um servidor remoto fica bloqueado esperando pela resposta desta requisição (Grosso, 2001). A Figura 5.1 mostra um exemplo de comunicação usando RMI.

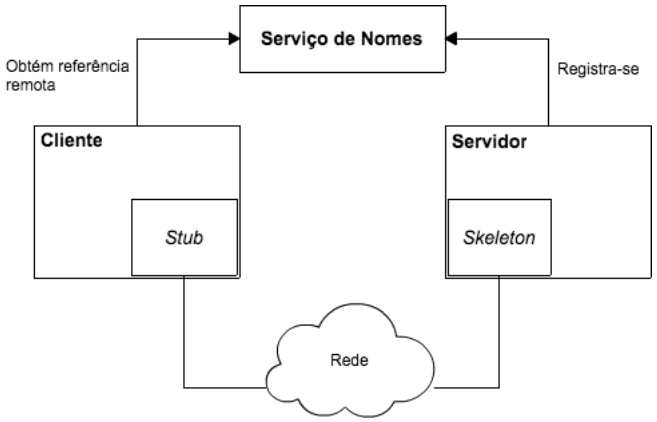

Figura 5.1: O modelo de comunicação RMI para um *middleware* orientado a objetos.
Fonte: Elaborado pelo autor.

A Figura 5.1 mostra o processo de invocação remota realizado entre um cliente e um servidor utilizando a tecnologia Java RMI. Primeiramente é necessário que o servidor se registre no serviço de nomes informando não só seus dados de localização, como *IP* e porta, como também o nome do objeto que irá identificá-lo pelos clientes. Em seguida, o cliente faz uma pesquisa ao serviço de nomes informando para ele o nome do objeto que se deseja invocar. O serviço de nomes retorna para o cliente um *stub* que é a referência remota para o objeto no servidor. O cliente, por meio do *stub*, realiza a invocação de uma operação do objeto remoto no servidor e fica bloqueado aguardando a resposta dele. No servidor, a requisição remota é

tratada pelo *skeleton* e a operação desejava é chamada no objeto presente no servidor. Após isso, o resultado da operação é encaminhado para o *stub* do cliente chamador por meio do *skeleton*.

CORBA é uma arquitetura padrão utilizada para simplificar a troca de informações entre aplicações distribuídas heterogêneas. Além disso, CORBA fornece uma arquitetura independente de linguagem de programação específica (Arno Puder, 2006). Para isso, CORBA utiliza uma Interface Definition Language (IDL) que é uma linguagem declarativa usada para descrever um modelo de objeto CORBA. Uma IDL serve para especificar um contrato entre o cliente e o servidor. O contrato estipula quais operações estão disponíveis no servidor e o que o cliente pode esperar em termo de funcionalidades (Arno Puder, 2006).

Na arquitetura CORBA está presente um módulo chamado Object Request Broker (ORB) que é usado para transmitir invocações de um cliente para um servidor distribuído garantindo, com isso, que a comunicação entre objetos em um ambiente distribuído seja feita de forma transparente (Arno Puder, 2006). O processo de invocação de um objeto remoto é feito pelo ORB por meio de duas interfaces: *invocation adapter* localizada no lado do cliente e *object adapter* no lado do servidor. O *invocation adapter* habilita um operação para ser gerada e invocada. Já o *object adapter*, garante a entrega da invocação para implementação do objeto no servidor (Arno Puder, 2006). O ORB utiliza durante o seu processo de comunicação um *stub* no lado do cliente e um *skeleton* no lado do servidor da mesma forma que o modelo RMI de Java. No entanto, o *stub* e o *skeletton* do ORB são construídos em tempo de compilação por meio de uma IDL.

CORBA e RMI são soluções para implementar sistemas *middleware* orientado a objetos que proveem uma infraestrutura para suportar aplicações distribuídas. Eles fornecem um modelo de infraestrutura semelhante a proposta apresentada pelo FLiMSy. Porém, RMI é restrita a linguagem de programação Java e CORBA é uma especificação que define um modelo de arquitetura para *middleware* orientado a objetos independente de tecnologia. Já o FLiMSy foi implementado em Scala e é dependente de um paradigma funcional.

5.2 Middleware Orientado a Mensagem

Um MOM (*Message Oriented Middleware*) é um tipo de *middleware* que provê a comunicação entre componentes distribuídos (transmissor e receptor) através da troca de mensagens. Esta troca normalmente se dá de forma assíncrona e quase sempre ordenada. A arquitetura

fornecida por este tipo de *middleware* garante um baixo acoplamento permitindo que processos executem de forma independente. Isso também permite que processos se comuniquem sem que precisem ser previamente conhecidos (Eugster *et al.* , 2003). Existem vários tipos de modelos de troca de mensagens para sistemas de *middleware* orientado a mensagens, dentre eles podemos citar: *message queuing* e *message publish/subscribe*.

No *message queuing*, o envio de mensagens é feita de um emissor para um único receptor (Eugster *et al.* , 2003). Todas as mensagens enviadas pelo emissor são enfileiradas em uma estrutura de dados do tipo *Queue* e elas permanecem na estrutura de dados até que o destinatário as consuma. Uma *Queue* ou fila pode ser vista como espaços globais, que são alimentados com mensagens dos emissores. A Figura 5.2 mostra o envido de mensagens usando um *middleware* orientado a mensagens do tipo *message queuing*.

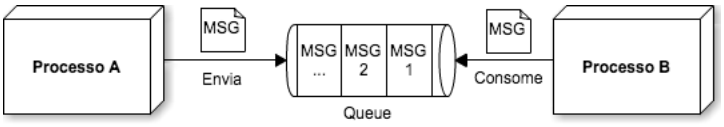

Figura 5.2: *Middleware* orientado a mensagens do tipo *message queuing*. **Fonte:** Elaborado pelo autor.

Esta figura mostra o envio de uma mensagem do processo *A* até o processo *B*. Para isso, o processo *A* envia uma mensagem que será enfileirada em uma estrutura de dados do tipo *Queue*. A mensagem fica armazenada na *Queue* aguardando até que o processo *B* a consuma. O processo *B* remove a mensagem da *Queue* após lê-la.

Já no *message publish/subscribe* a troca de mensagens é feita de um emissor para vários receptores. O *middleware* do tipo *message publish/subscribe* possui uma estrutura conhecida como tópicos (*topic*) que é responsável por guardar as mensagens dos emissores (Eugster *et al.* , 2003). Para que um processo consiga ler as mensagens em um *topic* ele precisa se inscrever nele. A Figura 5.3 mostra um exemplo de um *middleware* orientado a mensagens do tipo *message publish/subscribe*.

Na Figura 5.3, as mensagens são publicadas pelo processo *A* em um tópico. Em seguida,

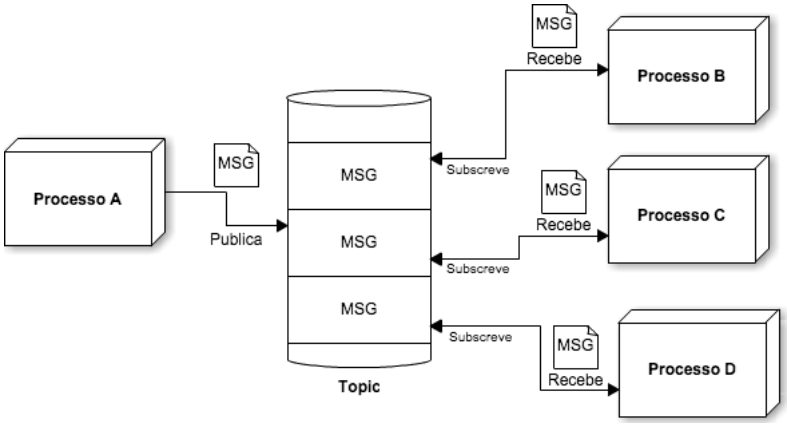

Figura 5.3: *Middleware* orientado a mensagens do tipo *message publish/subscribe*.
Fonte: Elaborado pelo autor.

os processos *B*, *C* e *D* que se subscreveram para as mensagens do tópico em questão poderão receber as mensagens publicadas. Com relação a implementação de um *middleware* orientado a mensagens, Java possui a especificação JMS (Java Message Service) que é usada para fazer com que aplicações se comuniquem por meio de mensagens (Hapner *et al.* , 2013). O JMS implementa tanto o modelo de troca de mensagens *message queuing*, quanto *message passing*. A Listagem 5.1 mostra um exemplo de código do modelo *message queuing* implementado em Java utilizando a especificação JMS.

```
1  public class JMSExample {
2    public static void main(String[] args) {
3          try {
4              ConnectionFactory myConnFactory = new com.sun.messaging.
     ConnectionFactory();
5
6        Connection myConn = myConnFactory.createConnection();
7
8        Session mySess = myConn.createSession(false, Session.AUTO_ACKNOWLEDGE
     );
9
10       Queue myQueue = new com.sun.messaging.Queue("world");
11
12       MessageProducer myMsgProducer = mySess.createProducer(myQueue);
13
14       TextMessage myTextMsg = mySess.createTextMessage();
15             myTextMsg.setText("Hello World");
16             System.out.println("Sending Message: " + myTextMsg.getText());
```

```
17              myMsgProducer.send(myTextMsg);
18
19              MessageConsumer myMsgConsumer = mySess.createConsumer(myQueue);
20              myConn.start();
21
22              Message msg = myMsgConsumer.receive();
23              if (msg instanceof TextMessage) {
24                  TextMessage txtMsg = (TextMessage) msg;
25                  System.out.println("Read Message: " + txtMsg.getText());
26              }
27
28              mySess.close();
29              myConn.close();
30      } catch (JMSException e) {
31        e.printStackTrace();
32      }
33    }
34 }
```

Listagem 5.1: Exemplo de *middleware* orientado a mensagem feito em Java

Nesta Listagem 5.1 é mostrado um exemplo de um *middleware* orientado a mensagens do tipo *message queuing*. Este exemplo foi implementado em Java fazendo uso da especificação JMS. Basicamente nele é demostrado a criação de uma estrutura de dados do tipo *Queue* chamada "world"que recebe uma mensagem cujo valor é "Hello World". Por fim, a mensagem gravada na estrutura é lida e o seu resultado mostrado no console da aplicação de exemplo.

Neste exemplo, primeiramente cria-se uma instância de *ConnectionFactory*, Linha 4, que será responsável por construir uma estrutura de gerenciamento de mensagens do tipo *message queuing*. Depois disso, a Linha 6 é usada para criar uma conexão de acesso à estrutura criada. Após isso, a Linha 8 é usada para criar uma sessão dentro da conexão criada. Em seguida, a Linha 10 é usada para criar a *Queue* que irá armazenar as mensagens envidas. As Linhas de 12 à 17 são usadas para criar uma mensagem e enviá-la para a estrutura do tipo *message queuing* que foi criada. Já as Linhas de 22 à 26 são usadas para ler as mensagens gravadas na estrutura e exibir os resultados no console da aplicação. Com relação a Linha 20, ela é usada para inicializar a estrutura do tipo *message queuing*. Por ultimo, as Linhas 28 e 29 são usadas para fechar a sessão e a conexão com a estrutura do tipo *message queuing*.

De forma semelhante ao FLiMSy, os sistemas *middleware* orientados a mensagem

fornecem um modelo arquitetural com suporte para o desenvolvimento de aplicações distribuídas. No entanto, eles possuem um modelo de comunicação diferente do FLiMSy. Isso porque, os sistemas *middleware* orientados a mensagem se comunicam por meio de troca de mensagens assíncronas e utilizam uma estrutura de dados intermediaria do tipo *Queue* que é responsável por enfileirar as mensagem trocadas entre os processos. Já o FLiMSy utiliza o modelo tradicional cliente/servidor. A comunicação desse modelo é feita por meio do mecanismo de *request/reply* onde o cliente envia uma requisição e fica aguardando uma resposta do servidor.

5.3 Middleware baseado em Espaço de Tuplas

Sistemas *middleware* baseados em espaço de tuplas são modelos computacionais que fazem uso de memória compartilhada para trocar informações entre processos que estão distribuídos na rede. O uso de espaço de tuplas propicia uma abstração da memória compartilhada em um sistema distribuído (Eric Freeman, 1999). Isso é possível porque o modelo de programação de memória compartilhada baseado em espaço de tuplas cria um repositório central que compartilha recursos, por meio de tuplas, entre os diversos processos.

Uma tupla é representada por uma sequência ordenada de campos que possuem um tipo definido de dado e um valor. As operações essenciais em um espaço de tuplas são escrita e leitura. A escrita é utilizada pelo processo para criar um tupla na memória compartilhada e a leitura é usada pelo processo para ler uma tupla presente na memória compartilhada (Eric Freeman, 1999). A Figura 5.4 mostra um exemplo de compartilhamento de recursos entre processos utilizando espaço de tuplas. Os retângulos representam os nós envolvidos no processo de comunicação, a nuvem representa o espaço de memória compartilhado, os círculos ilustram os processos que estão se comunicando por meio do mesmo espaço de memória compartilhada e a explosão a tupla que é o recurso compartilhado entre os processos.

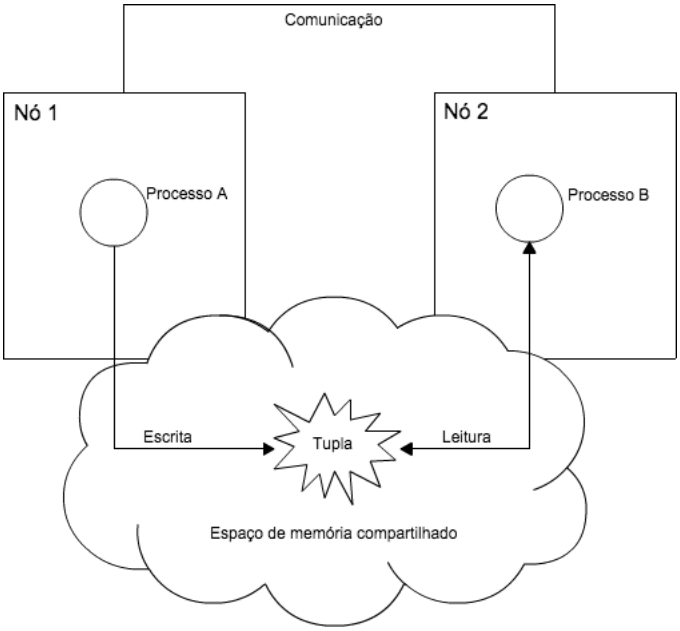

Figura 5.4: Compartilhamento de recursos utilizando espaço de tuplas. **Fonte:** Elaborado pelo autor.

A Figura 5.4 também ilustra a comunicação feita entre 2 processos. Primeiramente o processo *A* compartilha um recurso com o processo *B*. Para isso, o processo *A* escreve em forma de tupla o recurso no espaço de memória compartilhado. Em seguida, o processo *B* faz a leitura da tupla que contém o recurso compartilhado. Para localizar a tupla que contém o recurso, o processo *B* deve informar a mesma sequência de campos da tupla presente no espaço memória.

JavaSpaces é uma especificação Java que implementa o mecanismo de espaço de tupla (Eric Freeman, 1999). Para isso, o JavaSpaces não só faz uso de RMI para acessar o espaço de memória compartilhado, como também utiliza serialização de objetos para armazenar as tuplas. Uma tupla em JavaSpaces é representada pela interface *Entry* do pacote *net.jini.core.entry*. A interface *entry* é utilizada para definir um objeto *entry* que será gravado no repositório compartilhado (Eric Freeman, 1999). Essa interface estipula uma série de características para tornar um objeto elegível de ser gravado no espaço compartilhado. Algumas dessas características são: todos os campos de uma *entry* devem ser públicos, não estáticos, não transientes, não constantes (*finals*), deve conter um construtor público e sem argumentos e não são permitidos campos de tipos primitivos como (*int, long, float, double*) (Eric Freeman, 1999). Essas características são necessárias porque o JavaSpaces faz uso de uma técnica de *lookup* associativo para identificar as tuplas no espaço compartilhado.

As principais operações utilizadas pelo JavaSpaces são *read()*, *write()* e *take()*. A operação *write()* escreve um objeto *Entry* no espaço de memória compartilhado. Com relação a

operação *read()*, ela é usada para ler o objeto *entry* gravado no espaço de memória compartilhado sem remove-lo. Já a operação *take()* funciona de forma similar a operação *read()* sendo que o objeto *entry* gravado é removido do espaço de memória compartilhado. Para buscar uma tupla no espaço compartilhado do JavaSpaces é preciso criar um template do mesmo objeto gravado no espaço compartilhado e passá-lo como argumento nas operações de *read()* e *take()*. Um *template* consiste de uma *entry* de campos nulos ou contendo valores. O lookup associativo, durante o processo de busca, faz a comparação exata dos campos não nulos do template com os respectivos campos no objeto *entry* gravado no espaço (Eric Freeman, 1999). Caso haja uma relação entre todos campos durante a comparação, uma cópia do objeto *entry* gravado no espaço compartilhado é retornado. A Figura 5.5 mostra um exemplo de compartilhamento de recursos entre processos utilizando JavaSpaces.

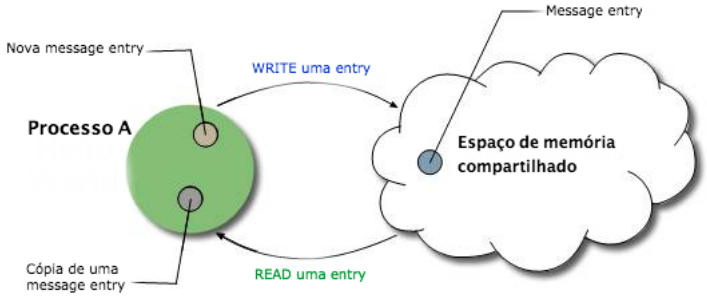

Figura 5.5: Compartilhamento de recursos utilizando JavaSpaces. **Fonte:** Elaborado pelo autor.

Basicamente o processo *A*, representado por um círculo, utiliza a operação de *write()* para escrever no espaço de memória, representado por uma nuvem, um objeto *entry* cujo conteúdo é uma mensagem. Após isso, o processo faz uso da operação *read()* passando como argumento um template do objeto *entry* localizado no espaço de memória. Em seguida, o JavaSpaces, por meio do mecanismo de *lookup* associativo, localiza o objeto *entry* no espaço de memória e retornar uma cópia dele para o processo solicitante. A Listagem 5.2 mostra o código, em Java, utilizado para implementar o exemplo do *hello world*.

```
1 public class HelloWorld {
2  public static void main (String[] args) {
3    try {
4        Message msg = new Message();
5        msg.content = "Hello World";
6
7        JavaSpace space = (JavaSpace)SpaceFinder.find("jini://*/*/mySpace");
8        space.write(msg, null, Lease.FOREVER);
```

```
9
10          Message template = new Message();
11          Message result = (Message)space.read(template, null, Long.MAX_VALUE)
        ;
12          System.out.println(result.content);
13      } catch (Exception e) {
14          e.printStackTrace();
15      }
16    }
17 }
18
19 class Message implements Entry {
20      public String content;
21
22      public Message() {}
23 }
```

Listagem 5.2: Exemplo de JavaSpaces

A classe *Message* mostrada na Listagem 5.2 é usada para definir os atributos e operações do objeto *Entry* que será gravado no espaço de memória. Para o nosso exemplo, a classe *Message* irá conter apenas o atributo *content* do tipo *String*, Linha 20, que será usado para armazenar uma mensagem que será compartilhada no espaço de memória. Para que a classe *Message* seja um objeto elegível para ser gravado em memória é preciso que ela não só implemente a classe *Entry*, Linha 19, como também defina um construtor sem parâmetros, Linha 22.

No método *main()* da classe *HelloWorld*, as Linhas 4 e 5 foram usadas para criar um objeto *Entry* com uma mensagem qualquer em seu conteúdo. Em seguida, a Linha 7 foi usada para obter acesso ao espaço de memória compartilhado. Após isso, a Linha 8 utiliza o método *write()* para escrever o objeto *Entry* criado no espaço de memória.

Para acessar a informação de um objeto *Entry* gravado no espaço de memória, é preciso criar um *template* do objeto *Entry* já gravado, Linha 10, e ,em seguida, é preciso executar o método *read()*, Linha 11, passando como argumento esse template. O resultado da execução do método *read()* é uma cópia do objeto *Entry* gravado no espaço de memória. A Linha 12 imprime, no console, o valor da variável *content* presente no objeto *Entry* gravado no espaço de memória.

Por fim, sistemas de *middleware* baseados em RPC ou que utilizam comunicação por meio de troca de mensagens fazem uso de uma forma de comunicação fortemente acoplada,

pois processos que enviam as mensagens e os que recebem estão diretamente ligados. Em contrapartida, sistemas *middleware* que utilizam o espaço de tuplas tendem a simplificar o processo de comunicação entre processos. Isso porque eles utilizam o espaço de tuplas como um meio comum de comunicação diminuindo o acoplamento entre os processos que enviam e recebem mensagens (Eric Freeman, 1999).

Da mesma forma que o *middleware* orientado a mensagem, os sistemas *middleware* baseados em Espaço de Tuplas fornece uma infraestrutura com suporte ao desenvolvimento de aplicações distribuídas. Porém, esse modelo possui um mecanismo de comunicação entre processos baseado no uso de memória compartilhada, modelo esse bem diferente do suportado pelo FLiMSy que utiliza o modelo tradicional cliente/servidor, por meio do mecanismo de *request/reply*.

5.4 Middleware Funcional

O FIrM é um *middleware* orientado a objetos com suporte para computação em nuvem construído em uma linguagem de programação funcional e que utiliza chamadas RPC para prover a comunicação entre processos (Silva & Rosa, 2015). A arquitetura do FIrM, assim como a do FLiMSy, foi construída baseada nos *remmoting patterns* a fim de prover uma arquitetura extensível, robusta e de fácil manutenção. Este *middleware* fornece uma infraestrutura básica para suporte a *multi-tenancy*. *Multi-tenancy* é o mecanismo que permite que múltiplas empresas compartilhem uma mesma plataforma de software de modo que cada uma fique isolada das outras.

Haskel foi a linguagem de programação utilizada na construção do FIrM. Ela foi escolhida por apresentar nativamente um conjunto de bibliotecas com suporte a diversos estilos de programação concorrente, além de outros recursos como: tipagem estática, ausência de efeitos colaterais e semântica concisa, que ajudaram durante a sua implementação (Silva & Rosa, 2015).

Em linguagens que possuem tipagem dinâmica, os tipos dos resultados das expressões executadas pelo software apenas podem ser conhecidos quando o software for executado. Isso tem a desvantagem de que nunca se saber ao certo, até depois do momento da execução, o que as expressões irão retornar (se será um número, um caractere ou uma outra expressão). Essa incerteza leva a necessidade de se implementar mecanismos que previnam o recebimento e processamento de valores indevidos para diminuir a possibilidade de erros. Já na tipagem

estática, a possibilidade de processar valores indevidos é diminuída pois todos os tipos são definidos e conhecidos antes da execução do software, o que implica em um código mais seguro (Lipova, 2011).

Normalmente, se diz que uma expressão ou função tem efeito colateral quando, além do resultado, é produzido uma alteração no estado geral da aplicação através de mudanças nas variáveis do software (Hughes, 1989). Devido à inexistência do conceito de variável em Haskell, o resultado das expressões ou funções dependem somente das entradas o que torna o resultado mais previsível reduzindo assim a possiblidade de erros (Hughes, 1989).

Ausência de variáveis, presente em Haskell, elimina também a necessidade de codificação de mecanismos que protegem elas de alteração indevida, os chamados mecanismos de *locking*. A tipagem estática e a ausência de efeitos colaterais implicam redução de código devido à ausência da necessidade de implementação de mecanismos de proteção (Pierro & Skinner, 2012; Hinsen, 2009). Isso favorece a criação de sistemas *multi-tenant* na medida em que ela favorece concisão de código e redução da necessidade de codificação de mecanismos de proteção.

A arquitetura do FIrM é dividida em três: camadas distribuição, serviço e infraestrutura. Cada camada oferece um tipo diferente de transparência, conforme mostra a Figura 5.6.

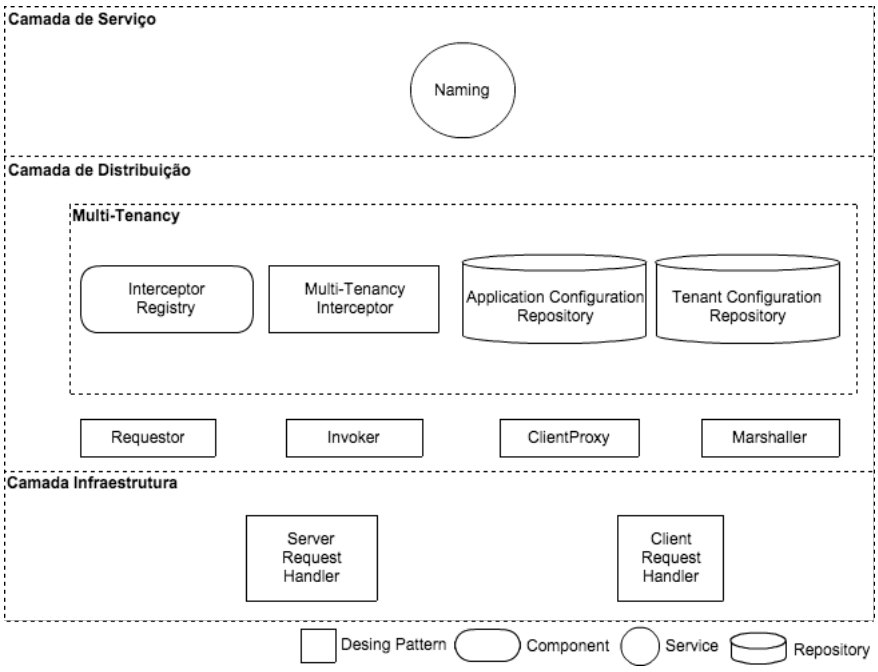

Figura 5.6: Arquitetura do FIrM (Silva & Rosa, 2015)

Na Figura 5.6, a camada de infraestrutura é utilizada para prover a comunicação entre o cliente e o servidor. Ela permite que o *middleware* gerencie a sua comunicação usando algum protocolo de rede. Além disso, por não fazer nenhum tratamento nos dados, esta camada não

é considerada multi-tenancy, pois ela é apenas responsável pelo processo de comunicação. Os *remote patterns ServerRequestHandler* e *ClientRequestHandler* estão presentes nesta camada. O *ServerRequestHandler* está localizado no lado do servidor e é responsável por receber uma invocação enviada pelo cliente. Já o *ClientRequestHandler* está localizado no lado do cliente e é responsável por enviar uma requisição para o servidor.

Com relação a camada de distribuição, ela é responsável por prover a transparência de acesso permitindo que uma chamada a um procedimento remoto seja semelhante a uma chamada a um procedimento local. No lado do cliente, o processo de invocação remota no FIrM começa por meio do *ClientProxy*. Este padrão possui a assinatura das operações presentes no objeto remoto, de modo que o *ClientProxy* sirva de ponto de acesso para as operações presentes no objeto do servidor. O *ClientProxy* chama ao padrão *Requestor*, que é responsável por invocar uma chamada remota, informando para ele os dados necessários para realizar a invocação remota, tais como: nome da operação desejada e sua lista de parâmetros. Em seguida, o *Requestor* faz uma chamada a camada de *multi-tenancy* a fim de obter os dados de configuração relacionados ao *tenant* que está realizando a invocação remota (Silva & Rosa, 2015). Após isso, o *Requestor* encapsula os dados da requisição em um objeto e utiliza o padrão *Marshaller* para converter este objeto em uma sequência de bytes. A sequência de bytes é repassada para camada de infraestrutura para que ela a envie para o servidor.

No servidor, a camada de infraestrutura recebe a sequência de bytes do cliente e repassa para o padrão *Invoker* presente na camada de distribuição. O padrão *Invoker* por meio do padrão *Marshaller* converte a sequência de bytes recebida no objeto da requisição. Em seguida, O *Invoker* faz uma chamada a camada de *multi-tenancy* informando os dados de configuração do *tenant* presentes no objeto da requisição. A camada de *multi-tenancy* aplica as configurações baseado nos dados do *tenant* recebido pelo objeto da requisição e habilita o *Invoker* para que ele realize a chamada ao objeto remoto (Silva & Rosa, 2015).

Por fim, a camada de serviço fornece um serviço de nomes, chamada *naming*, responsável por prover a transparência de localização dos recursos distribuídos. Esta camada faz uso do padrão *Lookup* usado para gerenciar referências a objetos. O *Lookup* utiliza um mecanismo de pesquisa para localizar a referencia a um objeto distribuído. Para obter uma referencia de um objeto, primeiramente o servidor informar seus dados de localização (IP e porta) e o nome do objeto remoto para o serviço de nomes. O serviço de nomes guarda esses dados em uma estrutura de dados persistente para eventuais consultas. Em seguida, o cliente faz uma pesquisa ao serviço

de nomes informando o nome do objeto remoto desejado. O serviço de nomes responde a solicitação do cliente fornecendo para ele uma interface de acesso ao objeto remoto. O cliente utiliza essa interface remota para invocar as operações do objeto no servidor.

FIrM e FLiMSy compartilham duas características em comum: o uso dos *remoting patterns* e o uso de uma abordagem funcional para a implementação. No entanto, o FIrM possui um foco no desenvolvimento de aplicações *multi-tenancy* e nos seus mecanismos, e foi implementado em uma linguagem puramente funcional (Haskell). A opção por Scala reflete uma maior preocupação com a disseminação do uso do FLiMSy em ambientes corporativos onde o Java já é amplamente disseminado.

5.5 Considerações Finais

Neste capítulo foram apresentados alguns exemplos de tipos de *middleware* que possuem alguma relação com a proposta desta dissertação. Também foram apresentados alguns sistemas *middleware* que usaram os *remoting patterns* na construção da sua arquitetura e utilizaram uma linguagem de programação funcional, como Haskell, no seu processo de desenvolvimento.

6

Conclusões e Trabalhos Futuros

Este capítulo apresenta as contribuições desta dissertação, assim como os passos que precisam ser realizados no futuro com o objetivo de estendê-la.

6.1 Conclusões

Nessa dissertação, foi apresentado todo o processo de análise e implementação do FLiMSy, assim como todos os componentes que compõe a sua arquitetura. Foram também mostrados quais são os requisitos atendidos pelo FLiMSy e os padrões para implementação de *middleware*, também conhecidos como *remoting patterns*, que foram utilizados para compor a sua arquitetura. Por fim, foi realizada uma avaliação para mostrar o impacto do FLiMSy no desempenho de aplicações distribuídas implementadas sobre ele.

Como contribuições desta dissertação podemos incluir:

- A implementação de um *middleware* orientado a objetos em Scala é a principal contribuição da dissertação;
- Uso do modelo de atores na implementação dos padrões de projeto de *middleware* "remoting patterns";
- A utilização dos *remoting patterns* na arquitetura do FLiMSy na implementação da transparência de acesso, transparência de localização, controle de concorrência, e serialização de dados.

Durante a avaliação do FLiMSy, que foi feita por meio de 3 experimentos objetivando avaliar o tempo de resposta das requisições enviadas de um cliente para um servidor, foram

observadas evidencias que comprovem o grau de escalabidade do FLiMSy para atender 50 clientes simultaneamente utilizando como carga de trabalho (*workload*) objetos do tipo "String".

Apesar do resultado da avaliação do FLiMSy serem satisfatórios, é preciso realizar novos experimentos com cargas de trabalhos e parâmetros diferentes a fim de confirmar com maior exatidão os resultados alcançados. Além disso, é preciso analisar melhor o impacto da linguagem de programação Scala no desenvolvimento de sistemas *middleware* orientado a objetos em comparação com linguagens convencionais. O código fonte do FLiMSy está disponível no site de compartilhamento de projetos *GitHub*[1].

6.2 Trabalhos Futuros

Como trabalhos futuros, podemos mencionar:

- Estender a arquitetura do FLiMSy adicionando os padrões básicos de gerenciamento de ciclo de vida: *Static Instance*, *Per-Request Instance* e *Client-Dependent Instance* (Markus Volter & Zdun, 2005). A adoção desses padrões pela arquitetura do FLiMSy iria ajudar no controle e gestão dos objetos criados pelo servidor para atender as requisições dos clientes. Isso porque o uso deles iria permitir ao servidor do FLiMSy gerenciar o ciclo de vida de seus objetos melhorando o tempo de resposta para atender as solicitações dos clientes, pois o servidor não precisaria instanciar um objeto a cada requisição feita por eles. Não obstante, o uso deles pelo FLiMSy também iria minimizar o uso de memoria pelo servidor, pois o servidor poderia reusar uma instancia já criada pra atender outros clientes.

- Adicionar a arquitetura do FLiMSy o requisito de transparência de replicação. Esse requisito vai permitir a redundância de instancias do recurso que serão usadas em caso de falhas evitando, com isso, a paralisação do sistema. Isso será feito de forma transparente para o cliente que não terá conhecimento destas replicas.

- Adicionar a arquitetura do FLiMSy um módulo de segurança para garantir a proteção dos recursos compartilhados. Além disso, é preciso também criar mecanismos de segurança para evitar ataque de negação de serviço que comprometam a disponibilidade da aplicação.

[1] https://github.com/labs2/FLiMSy

- Medir o esforço de implementação, como exemplo quantidade de linhas de código, relacionado a implementação de sistemas *middleware* usando paradigmas de programação convencionais e usando o paradigma funcional.

- Comparar o desempenho do FLiMSy com sistemas *middleware* equivalentes desenvolvidos em outras linguagens de programação, como: Java, C, C#.

Referências

Almasi, G. S., & Gottlieb, A. 1989. Highly Parallel Computing. *Benjamin-Cummings Publishing Co., Inc.*

Andrew T. Campbell, Geoff Coulson, & Kounavis, Michael E. 1999. Managing Complexity: Middleware Explained. *IT Professional*, **1**(5), 22–28.

Arno Puder, Kay Romer, Frank Pilhofer. 2006. *Distributed System Architecture: A Middleware Approach*. 2006.

Bernard, G., & Bernard, G. 2006. Middleware for Next Generation Distributed Systems: Main Challenges and Perspectives. *Pages 237–240 of: Proc. 17th International Conference on Database and Expert Systems Applications DEXA '06*.

Bernstein, Philip A., Hadzilacos, Vassco, & Goodman, Nathan. 1987. *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Bird, Richard, & Wadler, Philip. 1988. *Introduction to Functional Programming*.

Birrell, Andrew D., & Nelson, Bruce Jay. 1984. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, **2**(1), 39–59.

Buschman, Frank, Henney, Kevlin, & Schmidt, Douglas C. 2007. *Pattern-Oriented Software Architecture - A Pattern Language for Distributed Computing*. Vol. 4. John Wiley & Sons Ltd.

Cade, Mark, & Roberts, Simon. 2002. *Sun Certified Enterpreise Architect for J2EE Technology*. Sun Microsystems Press.

Cesarini, Francesco, & Thompson, Simon. 2009. *ERLANG Programming*. 1st edn. O'Reilly Media, Inc.

Coulouris, George, Dollimore, Jean, Kindberg, Tim, & Blair, Gordon. 2011. *Distributed Systems: Concepts and Design*. 5th edn. USA: Addison-Wesley Publishing Company.

Coutts, D., & Loh, A. 2012. Deterministic Parallel Programming with Haskell. *Computing in Science & Engineering*.

Debasish Ghosh, Justin Sheehy, Kresten Krab Thorup, & Vinoski, Steve. 2011. Programming Language Impact on the Development of Distributed Systems. *J Internet Serv Appl*, **3**(3), 23–30.

Emmerich, Wolfgang. 2000. Software Engineering and Middleware: A Roadmap. *Pages 117–129 of: Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. New York, NY, USA: ACM.

Eric Freeman, Susanne Hupfer, Ken Arnold. 1999. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley Professional.

Eugster, Patrick Th., Felber, Pascal A., Guerraoui, Rachid, & Kermarrec, Anne-Marie. 2003. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, **35**(2), 114–131.

Filman, Robert E., & Friedman, Daniel P. 1984. Coordinated Computing: Tools and Techniques for Distributed Software. *McGraw-Hill, Inc.*

Flanagan, David, & Matsumoto, Yukihiro. 2008. *The Ruby Programming Language*. First edn. O'Reilly.

George Coulouris, Jean Dollimore, Tim Kindberg, & Blair, Gordon. 2011. *DISTRIBUTED SYSTEMS Concepts and Design*. 2011.

Grosso, William. 2001. *Java RMI*. 1st edn. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

Hapner, Mark, Burridge, Rich, Sharma, Rahul, Fialli, Joseph, & Stout, Kate. 2013. *Java Message Service - Version 2.0*.

Hinsen, K. 2009. The Promises of Functional Programming. *Computing in Science & Engineering*.

Hohpe, Gregor, & Woolf, Bobby. 2011. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

Hughes, John. 1989. Why Functional Programming Matters. *The University, Glasgow*.

ISO. 1995 (july). *ISO 10476-1: Reference Model of Open Distributed Processing (Part I)*.

Issarny, Valerie, Caporuscio, Mauro, & Georgantas, Nikolaos. 2007 (23–25 May). A Perspective on the Future of Middleware-based Software Engineering. *Pages 244–258 of: Proc. Future of Software Engineering FOSE '07*.

Lipova, M. 2011. Learn You a Haskell for Great Good! *Thinking*.

Markus Volter, Michael Kircher, & Zdun, Uwe. 2005. *Remoting Patterns: Foundations of Enterprise, Internet and Real Time Distributed Object Middleware*. 2005.

Marlow, Simon. 2013. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media.

Martin Odersky, Lex Spoon, & Venners, Bill. 2008. *Programming in Scala: A comprehensive step-by-step guide*. 2008.

OMG. 2004 (Mar.). *Common Object Request Broker Architecture: Core Specification*. Object Management Group.

Orne, Lilianne Dantas, & de Araújo Macedo, Raimundo Jose. 2000. Uma Abordagem para Tolerância a Falhas em JAVA atraves de Comunicação em Grupo. *XVIII Simpósio Brasileiro de Redes de Computadores*.

O'Sullivan, Bryan, Goerzen, John, & Stewart, Don. 2008. *Real World Haskell*. 1st edn. O'Reilly Media, Inc.

P. Totoo, P. Deligiannis, & Loidl, H.-W. 2012. Haskell vs. F# vs. Scala,. *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing - FHPC*.

PIERCE, BENJAMIN C. 2002. *TYPES AND PROGRAMMING LANGUAGES*.

Pierro, Massimo Di, & Skinner, David. 2012. Concurrency in Modern Programming Languages. *Computing in Science and Engineering*.

Pratt, Terrence W., & Zelkowitz, Marvin V. 2000. *Programming Languages: Design and Implementation*. Pearson.

Qilin, Li, & Mintian, Zhou. 2010. The State of the Art in Middleware. *International Forum on Information Technology and Applications*.

R Development Core Team. 2008. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Schmidt, D.C., Schmidt, D.C., & Buschmann, F. 2003. Patterns, Frameworks, and Middleware: Their Synergistic Relationships. *Pages 694–704 of:* Buschmann, F. (ed), *Proc. 25th International Conference on Software Engineering*.

Schmidt, Douglas, Stal, Michael, Rohnert, Hans, & Buschmann, Frank. 2006. *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*. Vol. Volume 2. John Wiley & Sons Ltd.

Sebesta, Robert W. 2011. *Conceitos de Linguagens de Programação*. 9.

Silva, Andre, & Rosa, Nelson. 2015. FIrM: Functional Middleware with Support to Multi-tenancy. *Advanced Information Networking and Applications (AINA)*.

Steele, Jr., Guy L. 1990. *Common LISP: The Language (2Nd Ed.)*. Newton, MA, USA: Digital Press.

Van Rossum, Guido. 1995 (Apr.). *Python reference manual*. Report CS-R9525.

Venkatasubramanian, Nalini. 2002. Safe 'composability' of Middleware Services. *Commun. ACM*, **45**(6), 49–52.

Victor Pankratius, Felix Schmidt, & Garreton, Gilda. 2012. Combining Functional and Imperative Programming for Multicore Software: An Empirical Study Evaluating Scala and Java. *ICSE*, 123–133.

Vinoski, S. 2002. Where is middleware. *IEEE Internet Computing*, **6**(2), 83–85.

Wall, Larry. 2000. *Programming Perl*. 3rd edn. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

Zdun, U., Kircher, M., & Volter, M. 2004. Remoting patterns: design reuse of distributed object middleware solutions. *IEEE Internet Computing*, **8**(6), 60–68.