Uma infraestrutura para aplicações distribuídas baseadas em atores Scala

Thiago Henrique Coraini

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Francisco Carlos da Rocha Reverbel

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo, janeiro de 2012

Uma infraestrutura
para aplicações distribuídas
baseadas em atores Scala

Esta dissertação contém as correções e alterações
sugeridas pela Comissão Julgadora durante a defesa
realizada por Thiago Henrique Coraini em 28/11/2011.
O original encontra-se disponível no Instituto de
Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Francisco Carlos da Rocha Reverbel (orientador) - IME-USP
- Prof. Dr. Marco Dimas Gubitoso - IME-USP
- Profa. Dra. Graça Bressan - EP-USP

aos meus pais e
à minha querida Marcela,
este trabalho não seria possível sem vocês

Agradecimentos

Agradeço primeiramente aos meus pais, pelos valores de honestidade, humildade e respeito que hoje estão gravados perpetuamente em meu caráter. Minha trajetória de estudos, da qual este mestrado é o ponto culminante, tem suas raízes nas lições de matemática elementar que aprendi com minha mãe, que mesmo com seu pouco estudo esforçou-se para me passar seu conhecimento. Muito mais importante do que a matemática, passou-me a vontade de aprender cada vez mais. Esse esforço jamais será esquecido.

Agradeço à minha namorada Marcela, minha companheira inseparável desde o começo deste mestrado. Muito mais que uma namorada, uma verdadeira amiga, sua importância em minha vida nesses anos não pode ser expressa em palavras. Seu amor e seu carinho foram essenciais durante essa jornada, em especial nos momentos difíceis em que mais precisei de apoio.

Aos amigos de IME, também, por todos os momentos maravilhosos vividos desde a graduação. Companheiros de cervejas e de livros, de viagens e de EPs, nesse instituto conheci pessoas dos mais diferentes tipos com as quais me identifiquei de diferentes formas. O carinho que sinto por cada uma delas é proporcional à saudade que começa a aparecer agora que cada uma toma seu rumo e os encontros se tornam menos frequentes.

Agradeço de maneira geral aos professores do IME com os quais tive contato e com quem aprendi tanto. É uma honra ter estudado numa instituição repleta de profissionais tão competentes. Um agradecimento especial ao professor Francisco Reverbel, meu orientador neste trabalho. Primeiramente por ser o responsável por eu ter conhecido as ferramentas que hoje são tema deste trabalho, que até então eram completamente desconhecidas por mim. E, posteriormente, por todos os conselhos e orientações que permitiram moldar este trabalho. Agradeço também aos professores Gubi e Graça por terem aceitado participar de minha banca e pelos valiosos comentários feitos durante minha qualificação.

Finalmente, agradeço à CAPES pelo apoio financeiro durante a maior parte deste projeto. Não poderia deixar de agradecer ainda à empresa Neolog, onde comecei a trabalhar nos últimos meses deste mestrado e que me permitiu uma carga horária diferenciada por algum tempo, o que me proveu um tempo vital para a conclusão deste trabalho.

Resumo

Uma infraestrutura para aplicações distribuídas baseadas em atores Scala

Escrever aplicações concorrentes é comumente tido como uma tarefa difícil e propensa a erros. Isso é particularmente verdade para aplicações escritas nas linguagens de uso mais disseminado, como C++ e Java, que oferecem um modelo de programação concorrente baseado em memória compartilhada e travas. Muitos consideram que o modo de se programar concorrentemente nessas linguagens é inadequado e dificulta a construção de sistemas livres de problemas como condições de corrida e *deadlocks*. Por conta disso e da popularização de processadores com múltiplos núcleos, nos últimos anos intensificou-se a busca por ferramentas mais adequadas para o desenvolvimento de aplicações concorrentes.

Uma alternativa que vem ganhando atenção é o modelo de atores, proposto inicialmente na década de 1970 e voltado especificamente para a computação concorrente. Nesse modelo, cada ator é uma entidade isolada, que não compartilha memória com outros atores e se comunica com eles somente por meio de mensagens assíncronas. A implementação mais bem sucedida do modelo de atores é a oferecida por Erlang, a linguagem que (provavelmente) explorou esse modelo de forma mais eficiente.

A linguagem Scala, surgida em 2003, roda na JVM e possui muitas semelhanças com Java. No entanto, no que diz respeito à programação concorrente, os criadores de Scala buscaram oferecer uma solução mais adequada. Assim, essa linguagem oferece uma biblioteca que implementa o modelo de atores e é fortemente inspirada nos atores de Erlang.

O objetivo deste trabalho é explorar o uso do modelo de atores na linguagem Scala, especificamente no caso de aplicações distribuídas. Aproveitando o encapsulamento imposto pelos atores e a concorrência inerente ao modelo, propomos uma plataforma que gerencie a localização dos atores de modo totalmente transparente ao desenvolvedor e que tem o potencial de promover o desenvolvimento de aplicações eficientes e escaláveis.

Nossa infraestrutura oferece dois serviços principais, ambos voltados ao gerenciamento da localização de atores: distribuição automática e migração. O primeiro deles permite que o programador escreva sua aplicação pensando apenas nos atores que devem ser instanciados e na comunicação entre esses atores, sem se preocupar com a localização de cada ator. É responsabilidade da infraestrutura definir onde cada ator será executado, usando algoritmos configuráveis.

Já o mecanismo de migração permite que a execução de um ator seja suspensa e retomada em outro computador. A migração de atores possibilita que as aplicações se adaptem a mudanças no ambiente de execução. Nosso sistema foi construído tendo-se em mente possibilidades de extensão, em particular por algoritmos que usem o mecanismo de migração para melhorar o desempenho de uma aplicação.

Palavras-chave: Computação Distribuída, Scala, Modelo de Atores

Abstract

An infrastructure for distributed applications based on Scala actors

Writing concurrent applications is generally seen as a difficult and error-prone task. This is particularly true for applications written in the most widely used languages, such as C++ and Java, which offer a concurrent programming model based upon shared memory and locks. Many claim that the way concurrent programming is done in these languages is inappropriate and makes it harder to build systems free from problems such as race conditions and deadlocks. For that reason, and also due to the popularization of multi-core processors, the pursuit for tools better suited to the development of concurrent applications has increased in recent years.

An alternative that is gaining attention is the actor model, originally proposed in the 1970s and focused specifically in concurrent computing. In this model, each actor is an isolated entity, which does not share memory with other actors and communicates with them only by asynchronous message passing. The most successful implementation of the actor model is likely to be the one provided by Erlang, a language that supports actors in a very efficient way.

The Scala language, which appeared in 2003, runs in the JVM and has many similarities with Java. Its creators, however, sought to provide a better solution for concurrent programming. So the language has a library that implements the actor model and is heavily inspired by Erlang actors.

The goal of this work is to explore the usage of the actor model in Scala, specifically for distributed applications. Taking advantage of the encapsulation imposed by actors and of the concurrency inherent to their model, we propose a platform that manages actor location in a way that is fully transparent to the developer. Our proposed platform has the potential of promoting the development of efficient and scalable applications.

Our infrastructure offers two major services, both aimed at managing actor location: automatic distribution and migration. The first one allows the programmer to write his application thinking only about the actors that must be instantiated and about the communication among these actors, without being concerned with where each actor will be located. The infrastructure has the responsibility of defining where each actor will run. It accomplishes this task by using some configurable algorithm.

The migration mechanism allows the execution of an actor to be suspended and resumed in another computer. Actor migration allows applications to adapt to changes in the execution environment. Our system has been built with extension possibilities in mind, and particularly to be extended by algorithms that use the migration mechanism to improve application performance.

Keywords: Distributed Computing, Scala, Actor Model

Sumário

Lista de Figuras

Capítulo 1

Introdução

A computação distribuída pode ser entendida como aquela realizada por diferentes computadores, interagindo por meio de uma rede para coordenar a execução de uma mesma aplicação (chamada aplicação distribuída). São inúmeros os exemplos de aplicações que rodam de maneira distribuída, seja por sua própria natureza ou seja na busca de escalabilidade, tolerância a falhas ou ganhos de desempenho.

No entanto, desenvolver uma aplicação distribuída apresenta desafios que não existem em aplicações sequenciais típicas. Um desses desafios é lidar com a concorrência inerente da computação distribuída: para que o sistema seja eficiente, é necessário que diversas ações estejam sendo executadas simultaneamente nos diferentes computadores.

Atualmente, a concorrência é cada vez mais vista como uma realidade não só nos sistemas distribuídos, mas também em sistemas tradicionais rodando num único computador. Os processadores com vários núcleos estão cada vez mais difundidos e somente aplicações concorrentes serão capazes de aproveitar plenamente esse tipo de *hardware*.

Porém, desenvolver programas concorrentes é uma tarefa tida como difícil. Mais do que isso, o modelo de concorrência existente nas linguagens mais populares da indústria, como C++ e Java, é tido por muitos como bastante suscetível a erros [30, 38, 39]. Por outro lado, certos conceitos existentes em algumas linguagens funcionais, como a ausência de estado mutável, podem ser grandes aliados no desenvolvimento de aplicações concorrentes.

Com base nisso, uma linguagem relativamente nova surge como uma boa alternativa para muitas situações. A linguagem Scala [30] mistura conceitos existentes tanto em linguagens funcionais como em linguagens orientadas a objetos. Assim, ao mesmo tempo em que ela apresenta uma sintaxe muito familiar para programadores de Java ou C#, ela oferece ferramentas como variáveis imutáveis e funções de ordem superior.

No que diz respeito em particular à programação concorrente, uma alternativa que apresentou-se bastante efetiva quando implementada na linguagem Erlang [6] é o modelo de atores [4]. Esse modelo, que elimina a existência de memória compartilhada, torna a implementação de aplicações concorrentes menos propensa a erros em comparação com mecanismos mais tradicionais de sincronização, como travas, *mutexes*, etc.

No modelo de atores, cada ator é uma entidade isolada, que se comunica com outros unicamente via troca de mensagens assíncronas. Um ator encapsula seus dados e as instruções de como processar mensagens. A inexistência de memória compartilhada e a comunicação exclusivamente por passagem de mensagens tornam os atores entidades bastante adequadas para a programação concorrente.

1.1 Objetivos

O objetivo deste trabalho é desenvolver uma infraestrutura para a execução de aplicações distribuídas baseadas em atores. A ideia é que essa infraestrutura rode num aglomerado de computadores e gerencie os atores da aplicação, oferecendo:

- **Distribuição automática:** O aglomerado onde a aplicação será executada deve ser descrito num arquivo de configuração, listando o endereço de todas as máquinas disponíveis. A partir daí, o código da aplicação poderá usar primitivas genéricas de instanciação de atores, sem especificar onde cada ator deverá ser executado. A infraestrutura escolhe, com base em algum algoritmo, onde colocar cada ator. A manipulação dos atores e sua comunicação ocorrem de forma totalmente transparente.

- **Migração:** Nossa infraestrutura implementa um tipo particular de ator, denominado *ator móvel*. Este se comporta exatamente como um ator normal, porém possui a capacidade adicional de, ao receber uma mensagem específica, interromper sua execução e migrar para um computador diferente, continuando dali a sua execução.

O desenvolvimento de um sistema complexo usando o modelo de atores pode ser um desafio para programadores acostumados com as tecnologias e os paradigmas predominantes do mercado. Em particular, raciocinar sobre uma aplicação distribuída construída usando troca de mensagens assíncronas pode ser uma tarefa complexa.

No entanto, acreditamos que essas dificuldades iniciais são compensadas pelas vantagens que o modelo de atores traz consigo. Uma aplicação escrita usando atores não só pode ser executada concorrentemente numa máquina com vários processadores, como pode rodar num ambiente distribuído sem exigir grandes alterações. Do ponto de vista da arquitetura do sistema, pouca diferença existe na interação entre atores em núcleos diferentes do mesmo processador ou em computadores totalmente distintos, por conta em particular da inexistência de memória compartilhada e do assincronismo na troca de mensagens. Naturalmente que, no caso distribuído, alguma infraestrutura se faz necessária para permitir a comunicação entre computadores remotos.

Assim, acreditamos que existem plenas justificativas para que o uso de atores seja amplamente aceito no desenvolvimento de aplicações. Por isso um de nossos principais objetivos é que as facilidades da distribuição automática e migração de atores possam ser usadas de modo simples, sem requerer mudanças significativas nas aplicações distribuídas que as empregarem. A ideia é que nossa infraestrutura ofereça serviços de distribuição automática e migração de atores a aplicações distribuídas baseadas no modelo de atores, sem exigir praticamente nenhum esforço adicional por parte dos desenvolvedores dessas aplicações.

Além disso, acreditamos que nossa infraestrutura será aproveitada ainda melhor se, em vez de ser usada diretamente pelos desenvolvedores de aplicações, for usada como base para a construção de outras plataformas para computação distribuída. Uma possibilidade seriam plataformas que usem o mecanismo de migração para efetuar reconfigurações na aplicação com a finalidade de, por exemplo, alcançar uma melhor distribuição da carga nos nós do aglomerado.

Tendo em vista casos de uso como o mencionado acima, atenção especial foi dada para que nossa implementação seja facilmente extensível. O sistema deve possuir uma interface que permita consultar informações sobre os atores rodando no aglomerado. Com isso, algoritmos de tomada de

decisão podem realizar a migração de atores quando julgarem vantajoso, seguindo alguma política previamente definida.

Nosso sistema foi inteiramente desenvolvido usando a linguagem Scala, cuja distribuição oficial inclui uma biblioteca que implementa o modelo de atores. No entanto, decidimos não usar a biblioteca de atores de Scala neste trabalho. Ao invés disso, nossa implementação é feita sobre uma infraestrutura já existente, denominada Akka, também escrita em Scala e voltada ao desenvolvimento de sistemas distribuídos. O Akka possui uma ótima implementação do modelo de atores, o que nos levou a decidir por usar tal sistema como base para nossa infraestrutura.

1.2 Motivação

As aplicações podem ser distribuídas por diferentes razões. Muitas vezes uma aplicação precisa ser distribuída devido à própria localização dos recursos com os quais ela lida. Um programa de bate-papo, por exemplo, pode estar em execução em computadores de diferentes continentes, permitindo que pessoas a milhares de quilômetros de distância se comuniquem.

Além das aplicações inerentementes distribuídas, é cada vez mais comum que certos tipos de aplicações que exijam um grande poder computacional sejam executadas de maneira distribuída em aglomerados de computadores. Esses aglomerados muitas vezes são formados por computadores comuns, de maneira a diminuir seu custo. Muito se ouve falar, por exemplo, dos *datacenters* do Google, contendo centenas ou milhares de máquinas comuns que, juntas, são capazes de processar quantidades astronômicas de dados.

Todavia, não são apenas aplicações desse porte que podem se beneficiar da distribuição. Mesmo aplicações mais comuns podem apresentar ganhos de desempenho e vazão ao serem distribuídas num conjunto de computadores. Além disso, sistemas distribuídos também permitem uma maior tolerância a falhas e escalabilidade.

Dessa forma, a grande motivação deste trabalho é a necessidade de uma infraestrutura que efetivamente facilite a implementação de aplicações distribuídas eficientes, escaláveis e reconfiguráveis. Essa necessidade não nos aponta um objetivo imediato, mas um rumo. Supri-lo completamente seria um alvo ambicioso demais. Por isso, concentramo-nos em duas funcionalidades principais: distribuição automática e migração de atores.

1.2.1 Distribuição automática

A principal motivação para a distribuição automática dos atores é facilitar o trabalho dos desenvolvedores de sistemas distribuídos, desobrigando-os de ter que lidar, no código da aplicação, com detalhes como a localização das máquinas.

Além disso, uma aplicação cujo código não possui referências explícitas aos endereços das máquinas nas quais ela será executada é naturalmente adaptável a variações no número de nós do aglomerado, característica que favorece a escalabilidade. Para que uma mesma aplicação rode em 10 ou 100 computadores, por exemplo, basta que o arquivo de configuração que descreve o aglomerado seja modificado. Nosso sistema cuidará de distribuir os atores nas máquinas disponíveis.

A distribuição automática possibilita ainda que se efetue balanceamento de carga no momento da criação dos atores. O algoritmo de distribuição inicial, que decide onde colocar cada ator instanciado,

poderia levar em conta informações como as taxas de utilização dos recursos nos computadores do aglomerado. Assim, os atores seriam inicialmente colocados nos nós mais ociosos.

Uma discussão que merece ser levada em conta é sobre a transparência quanto aos atores remotos da aplicação. Ainda que o usuário possa controlar explicitamente em que máquina ele deseja rodar cada ator, espera-se que a atitude típica seja a de delegar a maior parte dos casos de instanciação de atores para a infraestrutura.

Além disso, é sempre possível consultar se um determinado ator é local ou remoto. Todavia, a ideia é que esse tipo de consulta não seja necessária na maioria das vezes. O desenvolvedor escreve a aplicação como se ela fosse ser executada num único nó, e o sistema se encarrega da colocação dos atores em diferentes máquinas. Essa colocação pode, inclusive, ser posteriormente modificada usando o mecanismo de migração.

O acima exposto deixa claro que, de maneira geral, nossa arquitetura aposta nas vantagens da transparência de localização de atores. Cabe observar, entretanto, que o conceito de transparência de localização foi questionado por vários autores, especialmente quando aplicado a sistemas distribuídos orientados a objetos ou baseados em chamadas remotas de procedimentos. Waldo *et al.* [37] consideram que é um erro tentar construir aplicações distribuídas com total transparência de localização. Esses autores afirmam que estão fadados ao fracasso os sistemas desenvolvidos seguindo um modelo que não diferencie o acesso a objetos locais do acesso a objetos remotos.

Waldo *et al.* relacionam diferenças fundamentais entre computação local e remota, que tornam, segundo eles, a abstração de localização inviável na prática. No entanto, cremos que nossa infraestrutura, e em parte o modelo de atores em si, lidam de maneira satisfatória com essas diferenças e portanto permitem um alto grau de transparência na construção de aplicações distribuídas. Estes são os principais pontos problemáticos relacionados por Waldo *et al.*:

- **Latência:** A diferença de latência entre uma chamada local e uma remota pode ser de algumas ordens de magnitude. Ignorar esse tipo de diferença pode levar a sérios problemas de desempenho [37]. Waldo *et al.* mencionam a importância de ferramentas que detectem padrões de comunicação entre objetos para tentar amenizar os problemas ocasionados por essa diferença de latência. Nos próximos capítulos, veremos um componente de nossa infraestrutura que tem exatamente essa função: registrar os padrões de comunicação entre os atores, permitindo que algoritmos de reconfiguração dinâmica coloquem num mesmo nó atores que realizam uma troca intensa de mensagens. Implementamos ainda o conceito de *atores co-locados*, que são instanciados necessariamente no mesmo nó e podem ser migrados sempre em conjunto.

- **Falhas parciais:** Lidar com falhas em sistemas distribuídos pode ser bem mais complicado que no caso de aplicações locais. Isso porque pode acontecer de apenas uma parte da aplicação parar de funcionar, devido a um problema em uma das máquinas que a executam. Muitas vezes o restante da aplicação não terá sequer como saber a natureza da falha ocorrida. A linguagem Erlang apresenta uma solução bastante interessante para tolerância a falhas em sistemas distribuídos baseados em atores. Essa solução, que usa um esquema de árvores de supervisão, pode ser empregada juntamente com nossa infraestrutura.

- **Concorrência:** A concorrência inerente a aplicações distribuídas é mais difícil de ser controlada do que aquela existente em aplicações com diversas *threads* rodando numa mesma máquina. Isso porque, no segundo caso, existem primitivas de sincronização que permitem

restringir os possíveis entrelaçamentos das execuções das *threads*. No entanto, o modelo de atores é particularmente adequado ao desenvolvimento de aplicações concorrentes, atacando o problema sob um novo paradigma que não o de acesso a memória compartilhada. Como cada ator possui sua própria linha de execução e acessa apenas seu estado interno, depois que a aplicação for modelada usando atores, a distribuição destes em diversos computadores exige pouco esforço adicional.

1.2.2 Migração de atores

No fim da década de 1990, a pesquisa na área de mobilidade de código estava extremamente aquecida. Buscava-se definir melhor a arquitetura e identificar as principais aplicações para os chamados *agentes móveis* [9, 16, 22]. Muitos acreditavam, inclusive, que tais agentes teriam um papel importante no futuro da Internet, que à época se firmava como uma verdadeira revolução nos sistemas computacionais.

No entanto, ao longo da última década, a utilização de agentes móveis e similares não fez jus às expectativas de muitos. Possivelmente devido à complexidade e ao custo adicional que ocasiona, o uso de mobilidade de código simplesmente não tenha sido atrativo o suficiente. Para justificar tais custos, muitos procuraram uma "aplicação matadora" para agentes móveis. Todavia, já em 1995, Harrison *et al.* declararam que "não há nada que possa ser feito com agentes móveis que não possa também ser feito de algum outro modo" [11]. Assim, ao longo dos anos o que se viu foi que a arquitetura cliente-servidor continuou a ser predominante na construção de sistemas distribuídos [10].

No entanto, desconsiderar completamente o uso de tecnologias que propiciem mobilidade de código pode ser uma decisão precipitada. Mesmo sem uma "aplicação matadora" que necessite de mobilidade, aplicações distribuídas podem se beneficiar da mobilidade de código em diferentes níveis [10, 23].

Nosso trabalho explora o conceito de mobilidade sob a perspectiva do modelo de atores. O uso de atores por si só é bastante atraente, em particular na construção de sistemas concorrentes. A mobilidade de tais atores aparece não como uma necessidade, mas como uma ferramenta adicional disponível aos desenvolvedores de aplicações distribuídas. Este trabalho certamente não é o primeiro a implementar atores móveis[1], porém acreditamos que fazer isso usando a promissora linguagem Scala é uma motivação suplementar.

Dentre as vantagens que aplicações baseadas em atores móveis podem apresentar, destacamos as seguintes:

- **Diminuição do tráfego de informação:** Dependendo das características da aplicação, pode ser mais vantajoso para um determinado ator mover-se até o nó onde se encontram os dados que ele precisa processar. Isso pode diminuir a quantidade de informação trafegando na rede, potencialmente aumentando a eficiência da aplicação.

- **Balanceamento de carga:** Uma aplicação distribuída baseada em atores, rodando sobre um aglomerado de computadores, pode ter sua carga balanceada por meio da migração de atores de computadores sobrecarregados para computadores com disponibilidade de recursos.

[1] No Capítulo 8 estudaremos alguns desses trabalhos que exploraram a ideia de atores móveis.

- **Tolerância a falhas:** Com a mobilidade de atores, uma aplicação distribuída pode se adaptar melhor a falhas nos computadores que a executam. Caso a queda iminente de um nó seja detectada, por exemplo, é possível migrar todos os atores em execução nesse nó antes que ele pare de funcionar.

- **Adaptabilidade:** De forma geral, uma aplicação baseada em atores móveis pode se adaptar, segundo critérios específicos, a seu ambiente de execução. Uma aplicação para grades oportunistas[2], por exemplo, pode efetuar a migração de todos os atores em execução num determinado computador quando seu dono voltar a usá-lo.

1.3 Organização do texto

Os próximos dois capítulos apresentam de modo sucinto os temas que consideramos ser pré-requisitos importantes para o entendimento do resto do texto. No Capítulo 2, descreveremos brevemente a linguagem Scala, concentrando-nos principalmente nos pontos em que ela mais difere de Java. No Capítulo 3, apresentaremos o modelo conceitual de atores, além de mostrar as implementações desse modelo nas linguagens Erlang e Scala.

No Capítulo 4, começaremos a descrever a infraestrutura desenvolvida. Nesse capítulo apresentaremos uma visão mais conceitual do sistema, descrevendo seus aspectos funcionais. O Capítulo 4 é essencialmente um "manual de usuário", que apresenta as funcionalidades presentes em nossa implementação.

Nos três capítulos seguintes daremos os detalhes da implementação da infraestrutura. No Capítulo 5, descreveremos a plataforma Akka, na qual nossa implementação é baseada. Os Capítulos 6 e 7 mostram efetivamente os detalhes da implementação dos componentes e dos serviços da infraestrutura, respectivamente.

O Capítulo 8 estuda trabalhos relacionados com o nosso. Finalmente, o Capítulo 9 conclui este trabalho com algumas considerações finais e propostas de trabalhos futuros.

[2]Nesse tipo de arquitetura, a grade é formada por computadores "emprestados" por seus proprietários. Os computadores só devem compartilhar seus recursos com a grade enquanto estiverem ociosos.

Capítulo 2

Scala

A linguagem Scala começou a ser desenvolvida em 2001, pelo professor Martin Odersky e sua equipe na EPFL[1], e foi lançada publicamente em 2003. O nome, segundo os próprios autores, significa linguagem escalável (***Sca****lable* ***La****nguage*). Escalável, nesse caso, tem a ver com a sua capacidade de "crescer" junto com a aplicação, permitindo por exemplo a criação de tipos ou estruturas de controle que se comportam de modo muito semelhante a recursos nativos da linguagem.

Para obter esse resultado, os criadores de Scala buscaram mesclar diferentes conceitos já existentes em diversas linguagens. É daí que advém uma de suas principais características: ao mesmo tempo em que é uma linguagem orientada a objetos, ela possui um lado funcional bastante marcante, o que inclusive leva muitos a chamá-la de uma *linguagem híbrida*.

Um outro fato relevante, e que certamente tem ajudado a impulsionar o sucesso de Scala, é o fato de ela ser compilada para a JVM, a máquina virtual Java. Além disso, um programa em Scala pode facilmente fazer chamadas a classes escritas em Java, facilitando muito a interoperabilidade com sistemas já existentes.

A seguir, daremos uma visão geral sobre algumas das principais características de Scala, de modo a permitir que leitores não familiarizados com essa linguagem acompanhem o restante deste trabalho.

2.1 Onde Scala difere de Java

Scala, em muitos aspectos, assemelha-se bastante com Java, e portanto um entendimento básico de seu código não é, em geral, muito difícil para pessoas com alguma fluência em Java. Todavia, em alguns pontos a sintaxe é bastante divergente. A seguir, serão expostas as principais diferenças entre as duas linguagens.

2.1.1 Declarações

Sintaxe

Nas declarações de variáveis, parâmetros ou métodos, o tipo do que está sendo declarado aparece depois de seu identificador, e não antes como em Java. Por exemplo:

```
var name: String = "Scala"
```

[1] École Polytechnique Fédérale de Lausanne, na Suíça.

Essa sintaxe funciona particularmente bem com outra característica muito interessante de Scala: na maioria das situações práticas, ela é capaz de fazer inferência de tipos, ou seja, deduzir o tipo do valor sendo declarado, desobrigando o programador de explicitá-lo. O trecho acima, por exemplo, seria perfeitamente válido se escrito como:

```
var name = "Scala" // O compilador infere o tipo como sendo String
```

Variáveis e métodos

Todas as declarações em Scala começam com uma palavra reservada: para métodos, essa palavra-chave é **def**, e para variáveis ela pode ser **var** ou **val**. Variáveis declaradas com **val** são imutáveis, comportando-se de maneira similar às declaradas com o modificador `final` de Java, ou seja, o valor atribuído a elas não pode ser posteriormente alterado. Isso combina bem com o lado funcional de Scala, ajudando a prevenir efeitos colaterais indesejados. Já variáveis declaradas com **var** tem comportamento igual às variáveis tradicionais de Java. O trecho a seguir mostra esses três tipos de declarações:

```
def square(x: Int) = x * x // Definição de método

var resultVar = square(3) // Variável que varia
val resultVal = square(4) // Variável imutável

resultVar = square(10) // OK
resultVal = square(15) // Erro de compilação!
```

2.1.2 Ponto-e-vírgula opcional

Nos exemplos anteriores, pode-se notar a ausência de um separador entre os comandos, como o ponto-e-vírgula em Java. Em Scala, essa separação também é inferida e o ponto-e-vírgula pode ser omitido na maioria dos casos (porém seu uso é permitido, caso desejado).

2.1.3 Chamada de métodos

Em Scala, sempre que um método receber apenas um parâmetro, os parênteses usados para chamá-lo podem ser substituídos por chaves. Uma chamada de impressão, por exemplo, pode ser feita como:

```
println { "Olá, Mundo!" }
```

Essa característica, peculiar à primeira vista, serve muito bem à proposta de permitir que bibliotecas acrescentem construções que pareçam fazer parte da linguagem. A biblioteca de atores de Scala (Seção 3.2.2), por exemplo, utiliza muito bem esse recurso, permitindo que a utilização de atores seja semelhante à de construções implementadas nativamente pela linguagem de programação.

2.1.4 Igualdade entre objetos

Em Scala, o operador == não representa a igualdade entre as referências de dois objetos, como em Java. Na verdade, o operador == usa o método `equals()` (existente para todas as classes) para decidir a igualdade entre os objetos. De maneira geral, pode-se dizer que esse método dá uma comparação "semântica" dos objetos, com base no que eles representam (por exemplo, `1.0 == 1` devolve `true`). Caso se deseje um comportamento como o de Java, a comparação entre referências, pode-se usar o método `eq()`, existente para todas as classes exceto as que representam tipos primitivos da JVM (como `Int`, `Double`, etc.).

2.1.5 Construtores

Em Scala, existe uma forma mais concisa e direta de se definir construtores para classes. Uma classe pode definir os chamados *parâmetros de classe* que, juntamente com qualquer código que não fizer parte de definições de campos ou métodos, formarão o *construtor primário* da classe. O trecho de código a seguir exemplifica o uso de um construtor primário:

```
class Fraction(numerator: Int, denominator: Int) {

  // Construtor primário
  println("Fracao sendo criada...")
  if (denominator == 0)
    println("O denominador nao pode ser 0!")
  else
    println("Fracao criada: " + numerador + "/" + denominador)

  /*
  * Definição dos atributos e métodos da classe
  * */
}
```

Como visto, na própria definição da classe já são explicitados os tipos dos parâmetros do construtor primário, e o código que aparece "solto" na definição da classe também entra nesse construtor. Como em Java, outros construtores podem ser criados: um método do tipo **`def this`**`(...)` é considerado um *construtor secundário*. Contudo, uma regra deve ser observada: a primeira ação a ser executada por todo construtor secundário é chamar algum outro construtor.

2.1.6 Parâmetros repetidos

Em Scala, é possível indicar que um método irá receber um número variável de parâmetros de um determinado tipo. Esse recurso, que também existe em Java sob o nome de *varargs*, em Scala chama-se parâmetros repetidos (*repeated parameters*). A ideia é essencialmente a mesma em ambas as linguagens, porém a sintaxe é diferente.

Um parâmetro repetido é indicado com o uso de um asterisco após o tipo desse parâmetro. Além disso, é obrigatório que tal parâmetro seja o último na lista de parâmetros que um método recebe. Dentro do método, essa lista de parâmetros (que pode ter qualquer tamanho a partir de zero) pode ser manipulada como um vetor do tipo indicado. O seguinte trecho de código exemplifica

a utilização de parâmetros repetidos:

```
def echoStrings(args: String*) = {
  for (arg <- args) println(arg)
}

echoStrings(``Uma'')
echoStrings(``Passando'', ``algumas'', ``palavras'')
// Passando zero parâmetros
echoStrings()
```

Como vemos no exemplo acima, a sintaxe do comando `for` em Scala também difere da usada em Java. Na Seção 2.3.4 explicaremos em mais detalhes o `for` de Scala.

2.1.7 Parâmetros com valor padrão

Um recurso interessante que Scala implementa, existente em C++ porém não em Java, é a possibilidade de especificar um valor padrão para um ou mais parâmetros de um método. Parâmetros com valor padrão podem ser omitidos no momento da chamada do método. Em muitas situações esse recurso dispensa o uso de sobrecarga de métodos para atingir o mesmo objetivo, deixando o código bem mais limpo e claro.

O seguinte trecho de código exemplifica o uso de parâmetros com valor padrão:

```
def printValues(i: Int = 42, b: Boolean = true, s: String = "Scala") = {
  println("Inteiro: " + i)
  println("Booleano: " + b)
  println("String: " + s)
}

// Todos os parâmetros com valor padrão
printValues()
// Especificando um ou mais parâmetros
printValues(84)
printValues(84, false)
printValues(84, false, "Java")
```

Um detalhe importante, no entanto, é que caso um dos parâmetros seja omitido na chamada do método, todos os parâmetros seguintes também terão de ser omitidos. A seguinte chamada, por exemplo, causa um erro de compilação:

```
// Tentando omitir o 1o. e o 3o. parâmetros
printValues(false)
```

2.1.8 Parametrização de tipos

A linguagem Scala oferece suporte para a parametrização de tipos, de maneira semelhante aos tipos genéricos de Java [1]. A diferença mais básica no que diz respeito à sintaxe é o uso de colchetes

em Scala para delimitar os parâmetros de tipos. A instanciação de uma tabela que associa cadeias de caracteres a inteiros, por exemplo, pode ser feita da forma:

```
val map = new HashMap[String, Int]
```

Além disso, a definição de um tipo parametrizado limitado por algum outro tipo também é diferente em Scala, com relação a Java. Em Scala, é usado o símbolo `<:`, da seguinte forma:

```
def decorate[T <: Component](comp: T): T = {
  ...
}
```

O método acima, por exemplo, recebe como parâmetro um valor de algum tipo `T`, desde que esse tipo seja uma subclasse de `Component`. Além disso, o valor devolvido pelo método também será do tipo `T`.

Na realidade, a parametrização de tipos em Scala é mais poderosa que a existente em Java. Em Scala, por exemplo, existem as anotações de variância, que permitem controlar as relações de herança entre tipos parametrizados. Contudo, esses são tópicos mais avançados, que fogem da proposta desta breve introdução à linguagem.

2.1.9 Modificadores de acesso

Assim como em Java, a linguagem Scala permite aplicar modificadores de acesso a classes, métodos e atributos. Dessa forma, é possível restringir a visibilidade de certos elementos e, com isso, fazer com que nem todas as classes tenham acesso a eles.

Em Scala, os únicos modificadores de acesso existentes são `private` e `protected`, que significam exatamente o mesmo que em Java. Além disso, qualquer membro sem um modificador de acesso explícito será considerado público, ou seja, acessível em qualquer parte do código. Esse comportamento difere do apresentado por Java, no qual elementos sem um modificador são considerados *package-private*, ou seja, são visíveis apenas dentro do pacote em que foram definidos.

Porém, Scala também permite que membros sejam definidos como *package-private*. Em Scala, existem qualificadores que podem ser aplicados aos modificadores de acesso, permitindo um controle muito maior sobre a visibilidade dos membros de classes. De maneira resumida, um qualificador indica que um determinado elemento aplique o modificador de acesso *até certo nível*. Esse nível, expresso no qualificador, será um pacote ou uma classe externa à classe onde o membro é definido. Um exemplo simples pode ser visto a seguir:

```
// Arquivo A.scala:
package somepackage
import other.B

class A {
  val b = new B
  // Chamada legal em Scala
  b.doSomething()
}
```

```
// Arquivo B.scala:
package somepackage.other

class B {
  private[somepackage] def doSomething() = { ... }
}
```

O qualificador `somepackage` no modificador de acesso `private` indica que o método `doSomething()` *será visível apenas no pacote* `somepackage`. Perceba que tal comportamento não seria possível em Java, onde:

- Se o método fosse feito `private`, seria visível apenas na classe `B`.
- Se o método não recebesse nenhum modificador de acesso (*package-private*), seria visível apenas no pacote `somepackage.other`.

Esse é apenas um exemplo de como a existência dos qualificadores de modificadores de acesso em Scala dão mais controle ao programador sobre como expor os membros de suas classes para serem usados pelos outros componentes do sistema.

2.1.10 O tipo `Option`

Scala possui um tipo particular para representar valores opcionais, como por exemplo o resultado de uma consulta a um `HashMap`. Em Java, nesse tipo de situação, a prática comum é usar o valor `null` para indicar a ausência de valor e, invariavelmente, o código acaba sendo povoado por testes do tipo **`if`** `(obj != null){ ... }`. Além disso, deixar de fazer uma dessas verificações pode fazer com que um erro do tipo `NullPointerException` passe despercebido pelos testes, possivelmente manifestando-se apenas depois, numa situação específica de uso da aplicação.

A abordagem de Scala ajuda na prevenção desse tipo de erro ao definir um novo tipo, chamado `Option`. Esse tipo é abstrato e a aplicação manipula suas duas subclasses concretas: o tipo `Some` e o tipo `None`. Portanto, uma variável do tipo `Option[String]` pode assumir os valores `Some(str)`, para indicar a existência de uma cadeia de caracteres `str`, ou `None`, caso não exista uma referência válida. O exemplo a seguir mostra como fica a consulta a uma estrutura de registros em Scala:

```
val capitals = HashMap[String, String]()
(...)
capitals.get(country) match {
  case Some(city) => println("A capital de " + country + " é " + city)
  case None => println("Não conheço a capital de " + country)
}
```

Por ora, pode-se pensar no funcionamento do **`match`** no código acima como análogo ao da estrutura `switch/case` existente em C e Java. Na Seção 2.3.6, o funcionamento das expressões **`match`** será explicado em mais detalhes.

Uma vantagem importante desse tipo de abordagem é que, sempre que o valor de uma variável ou método for de alguma forma opcional (como na consulta a um mapa, que pode ou não ter um valor associado à chave passada), essa condição fica explícita no tipo da variável. Em Java, é possível

que o programador esqueça de verificar se um valor devolvido por um método é válido ou é `null`. Em Scala, por outro lado, caso o valor seja do tipo `Option[T]`, o programador será obrigado a fazer a verificação, já que ele não poderá simplesmente usá-lo como sendo do tipo `T`.

2.1.11 O método `apply()`

Uma característica importante de Scala, inexistente em Java, é o comportamento especial do método `apply()`. Esse método, que pode aparecer na definição de qualquer classe e com número e tipos de parâmetros arbitrários, possui uma maneira particular de ser chamado. Dada uma instância `ref` de uma classe `C`, a expressão `ref(PARAMS)` faz com que o método `apply()` da classe `C` seja chamado passando-se `PARAMS` como parâmetros. O trecho de código a seguir exemplifica o emprego de `apply()`:

```
class IntGenerator {
  val randomGenerator: Random = new Random()

  def apply(multiplier: Int): Int = {
    randomGenerator.nextInt() * multiplier
  }
}

val generator = new IntGenerator()
val random = generator(17) // Na verdade chama o método apply()
```

O uso do método `apply()` é muito frequente em bibliotecas de Scala. É esse método que dá suporte a outra característica muito importante da linguagem: em Scala, funções também são objetos, e aplicar uma função a seus parâmetros nada mais é do que passar estes parâmetros ao método `apply()` da classe que representa aquela função.

2.2 Orientação a objetos em Scala

Scala adota um modelo de orientação a objetos mais puro do que o de Java, e nesse ponto a linguagem assemelha-se mais a Smalltalk. Em Scala, todo valor é um objeto e toda operação é, na verdade, uma chamada de método [29]. Consequentemente:

- Os tipos primitivos como inteiros, valores booleanos, etc., também são representados por classes;
- Mesmo operações como soma ($+$), multiplicação ($*$), etc., são chamadas de métodos, e podem ser definidas para classes criadas pelo usuário.

Esse modelo unificado promove uma padronização da linguagem e, em particular, facilita a extensão de Scala por meio de bibliotecas de terceiros. Como praticamente tudo são classes e métodos, torna-se possível estender as funcionalidades já existentes de maneira muito natural.

2.2.1 Operadores que são métodos

Vimos que em Scala não existem operações primitivas como soma ou subtração, mas sim métodos para representar essas ações. A regra geral para o uso de métodos como operadores estabelece que qualquer método que receber apenas um parâmetro pode ser chamado dispensando: *(i)* o ponto antes do nome do método; e *(ii)* os parênteses envolvendo o parâmetro passado. Portanto, temos que `4 + 2` na verdade é traduzido para `4.+(2)`.

É importante ressaltar dois pontos:

1. Essa regra serve para qualquer chamada de método. Assim, temos que `str.charAt(2)` e `str charAt 2` são absolutamente iguais.

2. Os nomes dos métodos devem ser exatamente o símbolo do operador desejado. Portanto, dentro da classe `Int`, por exemplo, deve existir algo como:

```
def +(arg: Int) = { ... }
```

Existe um caso especial nessa regra: métodos com o nome terminando em dois-pontos (`:`) são aplicados ao operando da direita, e não da esquerda; além disso, esses métodos possuem associatividade à direita, enquanto para todos os outros casos a associatividade é à esquerda. Assim, tem-se que `x :: y :: z` na verdade é traduzido para `z.::(y).::(x)`. O método `::`, existente na biblioteca de listas de Scala, é usado na construção de listas. Na Seção 2.3.5 veremos como essa exceção à regra torna a manipulação de listas bastante intuitiva.

2.2.2 Princípio do acesso uniforme

O princípio do acesso uniforme, definido por Meyer [26], afirma que deve existir uma sintaxe unificada que não diferencie um atributo implementado por uma computação (método) de um implementado por um valor armazenado (campo). Na prática, isso diz que o código cliente não deve ter de ser alterado com a decisão de se trocar um campo pré-computado em uma classe por um método de mesmo nome.

Scala implementa esse princípio por meio da possibilidade de se chamar métodos sem o uso de parênteses. Na verdade, qualquer método que não receba parâmetros pode ser chamado sem o uso de parênteses, mesmo que a definição do método tenha sido escrita com parênteses vazios (como é obrigatório em Java). Além disso, é possível omitir os parênteses mesmo na definição do método. O seguinte trecho de código exemplifica essas situações:

```
def sayHello = "Hello, world!" // Definição de método sem parênteses

val str = sayHello // Chamada de método sem parênteses
val length = str.length // Omitindo parênteses em método de Java
```

Dessa forma, caso desejássemos que o método `sayHello` fosse trocado por um campo (um **`val`** ou **`var`**), bastaria trocar sua implementação, e os clientes não precisariam ser alterados.

2.2.3 Feições

As feições (*traits*) de Scala desempenham um papel fundamental na construção de componentes reutilizáveis que podem ser aproveitados em diferentes aplicações e combinados entre si. Uma feição, a princípio, poderia ser vista como análoga a uma interface de Java, mas ela vai muito além. Uma das principais diferenças é que uma feição pode conter implementações de métodos.

A definição de uma feição é similar à de uma classe, porém trocando-se a palavra-chave **`class`** por **`trait`**. Ela deve ser combinada com classes utilizando-se a palavra-chave **`with`**, de maneira similar ao uso de `implements` em Java. A combinação de feições e classes, na terminologia de Scala, é conhecida como *mixin*.

A seguir discutiremos dois dos principais usos de feições em Scala [30], ambos bastante adequados ao modelo de orientação a objetos.

Interfaces ricas

Em linguagens como Java, a definição de interfaces costuma representar um dilema: interfaces com muitos métodos (ricas) são boas para os clientes, porém sobrecarregam aqueles que as implementarão; por outro lado, interfaces mais enxutas costumam exigir várias chamadas para se obter resultados que poderiam ser obtidos diretamente se a interface fosse rica.

As feições, por permitirem a existência de métodos concretos, ajudam a resolver essa questão. A feição pode ser enriquecida com métodos já implementados, que fazem uso de alguns poucos métodos abstratos, os quais serão definidos nas classes que forem combinadas com essa feição.

Modificações empilháveis

Feições também podem ser usadas para definir modificações sobre o comportamento padrão de uma dada classe. Começaremos mostrando um exemplo completo desse uso, para depois explicá-lo devidamente:

```
abstract class Formatter {
  def format(str: String): String
}

trait Capitalize extends Formatter {
  abstract override def format(str: String) =
    super.format(str.capitalize)
}

trait AddFrame extends Formatter {
  abstract override def format(str: String) = {
    "*" * (str.length + 4) + "\n" +
    "* " + super.format(str) + " *" + "\n" +
    "*" * (str.length + 4) + "\n"
  }
}

class SimpleFormatter extends Formatter {
  def format(str: String) = str
```

```
}

val formatter = new SimpleFormatter with Capitalize with AddFrame
formatter.format("hello, World!")
```

A última linha acima gera uma cadeia de caracteres com a primeira letra maiúscula, envolta por uma moldura de asteriscos. A primeira coisa a se notar é que uma feição pode estender uma superclasse: isso sinaliza que classes que forem combinadas com tal feição devem ter essa mesma superclasse em comum.

Outra característica especial de feições é que elas podem sobrescrever métodos abstratos da classe pai usando chamadas a `super`. Em classes comuns isso não é permitido, pois essa chamada certamente falharia (já que a superclasse não implementa tal método). Já no caso de feições, é adotado um esquema de linearização das classes [28, 30] que define quem responderá à chamada a `super` em tempo de execução. O importante é que exista uma classe que implemente de fato o método (no caso do exemplo, a classe `SimpleFormatter`).

Assim, diversas feições podem ser "empilhadas" sobre uma classe concreta, modificando o comportamento dessa classe. No exemplo dado, classes que combinassem apenas uma das feições teriam apenas as modificações dessa feição aplicadas.

O interessante é que, dessa forma, podemos definir modificações que podem ser combinadas sem que uma prevaleça sobre a outra (como aconteceria no caso de herança múltipla). Em vez disso, os efeitos delas são somados.

2.2.4 Objetos *singleton*

Buscando implementar um modelo de orientação a objetos mais puro [30], a linguagem Scala não possui o modificador `static`, que em Java indica que um atributo ou método pertence à classe, e não às instâncias dessa classe. Ao invés disso, em Scala existem classes que possuem uma única instância, denominada objeto *singleton*.

A definição de um objeto *singleton* é feita de maneira muito semelhante à definição de uma classe normal, porém trocando-se a palavra **`class`** por **`object`**. Além disso, ao se definir, num mesmo arquivo, uma classe e um objeto *singleton* de mesmo nome, este é denominado objeto acompanhante (*companion object*) da classe, e um consegue ter acesso aos campos privados do outro.

Portanto, em Scala, ao invés de se usar atributos e métodos de classe (com o modificador `static`), como em Java, usam-se atributos e métodos definidos no objeto acompanhante da classe. O acesso a esses campos, porém, é bastante semelhante ao feito a campos `static` de Java:

```
1 object Counter {
2   var value = 0
3
4   def increment: Int = {
5     value = value + 1
6     value // Não necessita de 'return'
7   }
8 }
9
```

```
10 println(Counter.increment) // Imprime 1
11 println(Counter.increment) // Imprime 2
```

Listagem 2.1: *Acesso a campos de objetos singleton em Scala.*

Um detalhe adicional é que podemos definir métodos `apply()` em objetos *singleton*. A chamada ao método, como seria de se esperar, é feita passando os parâmetros diretamente ao nome da classe. O seguinte exemplo mostra como usar essa característica para implementar um método de fábrica para uma determinada classe:

```
class IntGenerator private (seed: Int) {
  val randomGenerator: Random = new Random(seed)

  def apply(multiplier: Int): Int = {
    randomGenerator.nextInt() * multiplier
  }
}

// Objeto acompanhante da classe:
object IntGenerator {
  def apply(seed: Int): IntGenerator = new IntGenerator(seed)
}

// Uso por parte do cliente:
val generator = IntGenerator(19) // Chamando apply() do objeto
val random = generator(17) // Chamando apply() da classe
```

Listagem 2.2: *Usando o método `apply` em objetos singleton.*

O modificador `private`, na posição em que aparece, torna privado o construtor primário da classe. Portanto, a única maneira de instanciar objetos dessa classe será por meio do método `apply()` do objeto acompanhante da classe `IntGenerator`. Na prática, para os clientes da classe, isso significa essencialmente que objetos podem (e devem) ser instanciados usando apenas o nome da classe, sem o emprego de **`new`**.

2.2.5 Definições implícitas

Scala possibilita a definição de elementos implícitos, que podem ser de dois tipos: *conversões implícitas* e *parâmetros implícitos*.

Conversões implícitas

O uso de conversões implícitas permite uma grande extensibilidade à linguagem. Com esse tipo de conversão, é possível por exemplo fazer com que uma classe pré-existente incorpore novos métodos em sua interface, ainda que sem modificar nenhuma linha de código na classe em si.

Uma conversão implícita é definida como um método que recebe um argumento do tipo A e devolve um resultado do tipo B. Esse método será usado para converter valores do tipo A em valores do tipo B. Além disso, para ser usado como uma conversão implícita, a definição desse método deve começar com a palavra **`implicit`**.

Considere, por exemplo, a classe `Fraction` definida abaixo:

```
class Fraction(val n: Int, val d: Int) {
  def *(other: Fraction): Fraction = {
    new Fraction(this.n * other.n, this.d * other.d)
  }

  def *(i: Int) = new Fraction(this.n * i, this.d)
}
```

Nessa classe, que representa frações, são definidos dois métodos: multiplicação por outra fração e multiplicação por um inteiro. Poderíamos então fazer o seguinte uso da classe:

```
val f1 = new Fraction(2, 3)
val f2 = new Fraction(5, 7)

f1 * f2 // Devolve 10/21
f1 * 5 // Devolve 10/3
```

Todavia, seria bastante natural imaginar trocar a ordem dos operadores na segunda forma de uso, em que ocorre a multiplicação por inteiro. Ao menos do ponto de vista matemático, fazer `5 * f1` seria completamente aceitável e deveria produzir o mesmo resultado. Contudo, isso implicaria em chamar o método `*` em um objeto da classe `Int` passando como parâmetro uma instância de `Fraction`. Naturalmente a classe `Int`, de Scala, não implementa nenhum método que lide com objetos do tipo `Fraction`.

Porém, o uso de conversões implícitas permite "estender" a classe `Int`, adicionando um método que realize a multiplicação por frações. O seguinte trecho mostra como isso pode ser feito:

```
implicit def intToFraction(i: Int): Fraction = new Fraction(i, 1)

5 * f1 // Devolve 10/3
```

Caso a conversão implícita tenha sido definida em um escopo diferente do escopo no qual será usada, ela deverá ser trazida para o escopo de uso, com o emprego de `import`. De maneira geral, as conversões implícitas serão aplicadas:

1. Quando o tipo de um parâmetro passado numa chamada a método não condiz com o tipo esperado. Antes de considerar isso um erro, o compilador tentará achar uma conversão implícita que converta o tipo passado naquele esperado.

2. Quando um método não implementado por uma classe é chamado numa instância dessa classe (é o caso do exemplo de frações). Nesse caso, o compilador tentará achar uma conversão implícita que transforme o tipo do objeto num tipo que ofereça o método chamado.

Parâmetros implícitos

Parâmetros implícitos permitem que toda uma lista de parâmetros de um método não seja passada explicitamente, mas sim preenchida por valores definidos como implícitos no escopo da

chamada ao método. O seguinte método, por exemplo, é definido com uma lista de parâmetros implícitos:

```
class Greeting(val greeting: String)

object Login {
  def login(user: User)(implicit greet: Greeting) {
    println(greet.greeting + user.name)
    (...)
  }
}
```

Chamadas ao método poderiam ser feitas então omitindo-se o parâmetro implícito, desde que algum valor implícito estivesse disponível ao compilador no momento da chamada:

```
implicit val greeting = new Greeting("Bom dia, ")

// Passagem implícita do parâmetro
Login.login(someUser)

// A passagem explícita também é permitida
Login.login(otherUser)(greeting)
```

Esse exemplo procura apenas ilustrar o que são parâmetros implícitos. Na Seção 5.2.4, quando discutirmos a seriação de atores no Akka, veremos uma utilização mais interessante de parâmetros implícitos.

Considerações sobre definições implícitas

Uma primeira questão sobre conversões implícitas é: quando o compilador irá utilizar alguma delas? As regras mais importantes a se ter em mente são: *(i)* somente declarações marcadas com **`implicit`** e disponíveis no escopo atual serão utilizadas; *(ii)* caso haja duas conversões implícitas que poderiam ser usadas num certo ponto, o compilador escolherá a conversão mais específica; e *(iii)* apenas uma conversão implícita é utilizada por vez, ou seja, não se pode esperar *transitividade* na aplicação de conversões implícitas.

É importante usar sempre tipos bastante específicos em definições implícitas. Por isso, no exemplo anterior foi definida uma nova classe `Greeting`, apesar de esta na prática ser apenas uma `String`. Caso fosse usado o tipo `String` diretamente, muito possivelmente ocorreriam problemas com diferentes valores implícitos disponíveis para uso num mesmo escopo.

Por fim, vale ressaltar que, apesar de ser um recurso interessante e poderoso, o uso de definições implícitas pode obscurecer o código, dificultando o entendimento e a resolução de erros. Mesmo que se use tipos bastante específicos nas definições implícitas, o programa final que será executado não corresponde exatamente ao código escrito, pois certas partes foram preenchidas com valores implícitos escolhidos pelo compilador. Portanto, a recomendação é que o uso de definições implícitas seja feito com moderação, apenas em situações especiais.

2.3 O lado funcional de Scala

Os criadores de Scala costumam descrevê-la como uma linguagem que mistura orientação a objetos com programação funcional. Isso se justifica: embora essa linguagem implemente um modelo bastante puro de orientação a objetos, ela também possui diversos conceitos típicos de linguagens funcionais.

Assim, muitas das vantagens propiciadas pelas linguagens funcionais acabam se integrando bem com os objetos de Scala. O suporte a funções de primeira classe (que também são objetos) facilita bastante a resolução de certos tipos de problemas. Já a existência de estruturas imutáveis (cujos campos são declarados com **`val`**) ajuda o programador a se proteger de efeitos colaterais indesejados.

2.3.1 Funções de primeira classe

O suporte a funções de primeira classe é provavelmente a característica principal do paradigma funcional. Em Scala, os chamados literais funcionais definem funções anônimas que podem ser armazenadas em variáveis, passadas como parâmetros ou devolvidas por outras funções. A Figura 2.1 exemplifica a sintaxe de um literal funcional.

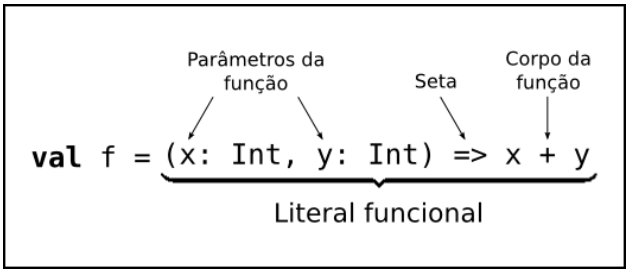


Figura 2.1: *Sintaxe para literais funcionais em Scala.*

Além das funções criadas usando-se essa sintaxe, métodos definidos da maneira tradicional, dentro de classes, também podem ser usados como funções de primeira classe e manipulados da mesma maneira que os literais funcionais. Na Seção 2.3.5 será apresentada uma biblioteca de Scala que define diversas funções de ordem superior (que recebem outras funções como parâmetros) para operar sobre listas.

2.3.2 Passagem de parâmetros por nome (*by-name parameters*)

Algumas vezes, pode ser interessante passar uma expressão como parâmetro a um método de tal forma que sua avaliação seja preguiçosa (*lazy*). Isso significa que essa expressão só deve ser avaliada dentro do corpo do método, no momento em que for referenciada (ou seja, é possível que ela sequer seja avaliada). Tomemos como exemplo o seguinte método:

```
def optionalEval(eval: Boolean, expr: Boolean) = {
  if (eval) expr
  else true
}

// Lança exceção de divisão por zero
optionalEval(false, (5/0) == 0)
```

Essa solução não surte o efeito desejado, já que os parâmetros são avaliados antes de serem passados ao método. Caso seja importante atrasar a avaliação de uma expressão passada como parâmetro a um método, podemos recorrer aos literais funcionais. Assim, esta seria uma forma correta de escrever o método `optionalEval`:

```
def optionalEval(eval: Boolean, expr: () => Boolean) = {
  if (eval) expr()
  else true
}

// Roda normalmente, devolve 'true'
optionalEval(false, () => (5/0) == 0)
```

Nessa versão, o problema é resolvido. Aqui, o método recebe um literal funcional, que só é executado dentro do corpo do método. Assim, o método tem controle sobre quando e se a expressão passada será avaliada. Todavia, essa sintaxe torna o código um pouco confuso e menos intuitivo.

Para esses casos, Scala dispõe de uma alternativa mais interessante, denominada *passagem de parâmetros por nome* (*by-name parameters*). A seguinte versão de `optionalEval` mostra o uso dessa forma de passagem de parâmetros:

```
def optionalEval(eval: Boolean, expr: => Boolean) = {
  if (eval) expr
  else true
}

// Roda normalmente, devolve 'true'
optionalEval(false, (5/0) == 0)
```

Como vemos, a sintaxe para indicar que um parâmetro é passado por nome é semelhante à do exemplo anterior, que recebia um literal funcional, porém dispensa o uso de `()`. Além disso, na chamada do método os parênteses e o => também desaparecem, passando-se diretamente a expressão desejada (que, no entanto, só será avaliada posteriormente). Esse mecanismo propicia bastante clareza ao código, além de facilitar a implementação de funcionalidades que se assemelham a recursos nativos da linguagem.

2.3.3 Valor devolvido por funções

Em Java, todo método cujo tipo do valor devolvido não seja `void` deve executar um comando **`return`** para indicar qual valor será devolvido. Já em Scala, o uso de **`return`** é opcional e, caso seja omitido, o último valor calculado pelo método será devolvido (como na linha 6 da Listagem 2.1).

Com isso, Scala possui uma abordagem mais uniforme com relação aos comandos, pois todos são na verdade expressões que possuem um valor. É o caso por exemplo do **`if`**, que tem um comportamento semelhante ao do operador ternário de Java[2]:

```
val inputFile =
  if (args.length > 0)
```

[2] `int max = (a > b)? a : b`

```
    args(0)
  else
    "default.txt"
```

Essa abordagem, bastante comum em linguagens funcionais, levanta a questão de qual seria o valor de laços **while** ou de chamadas a funções que não devolvem informação útil (o tipo `void` não existe em Scala). Para esses casos, Scala define o tipo `Unit`, que possui apenas um valor, representado por `()`. Assim, dizer que uma expressão devolve um valor do tipo `Unit` é o mesmo que dizer que ela devolve um valor irrelevante, que pode ser descartado.

2.3.4 O comando `for` de Scala

O comando **for**, muito comum em linguagens imperativas, também está presente em Scala, porém com um funcionamento diferente do tradicional: ele possui uma sintaxe declarativa que permite iterar sobre uma coleção e selecionar itens dessa coleção.

Esse **for** pode ser usado basicamente de duas maneiras: como o comando **for** de C ou Java, que avalia repetidamente o corpo do **for** porém não devolve um valor relevante, ou como uma expressão que produz um valor (o qual será outra coleção).

De maneira geral, a sintaxe do **for** em Scala é

```
for ( SEQUENCIA ) CORPO
```

onde `SEQUENCIA` é uma sequência de:

- **Geradores:** Definem variáveis que iteram sobre os elementos de uma coleção, utilizando a sintaxe `VARIAVEL <- COLECAO`. Exemplo:

  ```
  for (element <- list) println(element)
  ```

- **Filtros:** Definem regras que filtram os elementos de coleções percorridas por geradores. Exemplo:

  ```
  for (
    arg <- args
    if arg.startsWith("-")
  ) println("Argumento passado: " + arg)
  ```

- **Atribuições:** Criam variáveis internas ao **for**, evitando recalcular certas expressões. Exemplo:

  ```
  for (
    file <- filesList;
    filename = file.trim
    if filename.endsWith(".txt")
  ) println(filename)
  ```

Em todos os exemplos mostrados acima, o corpo do **for** assemelha-se ao que costuma ser usado em linguagens como C ou Java: um ou mais comandos executados apenas por seus efeitos colaterais (no caso, imprimir algo na tela). Essa é uma situação bastante comum em linguagens imperativas, porém vai de encontro aos princípios do modelo funcional.

Por outro lado, Scala oferece uma outra forma de uso do **for**, que está em maior concordância com os preceitos funcionais. Ao substituir-se o corpo do laço por '**yield** EXPR', será gerada uma nova coleção, com elementos produzidos pelas sucessivas avaliações de EXPR. Isso permite que o **for** seja usado não por seus efeitos colaterais, mas sim como uma expressão que produz um valor, como no exemplo abaixo:

```
val positives =
  for (
    number <- List(-3, 1, 4, 0, -2, 18, 42, -13)
    if (number > 0)
  ) yield number

positives == List(1, 4, 18, 42) // TRUE
```

2.3.5 Listas

Listas em Scala são coleções imutáveis de objetos de um mesmo tipo. Elas são implementadas pela classe `List`, parte da biblioteca padrão de Scala. Mesmo assim, os recursos da linguagem fazem-nas parecer um tipo de dados nativo, como é o caso em muitas linguagens funcionais.

A manipulação básica de listas é feita utilizando-se os seguintes elementos:

- O operador `::`, também conhecido como *cons*. Ele realiza a construção de uma lista não vazia dado seu primeiro elemento e o restante da lista (que pode ser vazio).
- A lista vazia, representada por `Nil` ou `List()`.
- Os métodos `head` e `tail`, que devolvem a cabeça da lista e o restante dela, respectivamente[3], e o método `isEmpty`, que informa se a lista está vazia.

A classe `List` define ainda diversos métodos que recebem literais funcionais para operar sobre os elementos da lista. Tais métodos, como `map`, `filter` e `exists`, são bem comuns em linguagens funcionais. Na Listagem 2.3, podemos ver a criação de uma lista e o uso de dois destes métodos. Vale observar ainda, na linha 2, o encadeamento do operador `::`, na criação de uma lista. Como vimos na Seção 2.2.1, esse encadeamento só é possível pois métodos com nomes terminados em `:` têm associatividade à direita em Scala, além de serem aplicados ao operando da direita.

```
1 // List(-2, -1, 0, 1, 2)
2 val numbers = -2 :: -1 :: 0 :: 1 :: 2 :: Nil
3
4 // List(4, 1, 0, 1, 4)
5 val squares = numbers.map(x => x * x)
6
7 // List(1, 2)
8 val positives = numbers.filter(x => x > 0)
```

Listagem 2.3: *Funções de ordem superior da classe `List`.*

[3] Esses métodos são equivalentes às primitivas `car` e `cdr`, de Lisp.

2.3.6 Casamento de padrões e *case classes*

Algumas linguagens funcionais possuem suporte nativo para casamento de padrões. Em Erlang, por exemplo, o uso de casamento de padrões é tão comum que mesmo uma atribuição à variável, como `X = 42`, é na verdade uma tentativa de casamento de padrão que, para ser bem sucedida, deve vincular o valor da direita à variável da esquerda[4] [6].

Em Scala, o casamento de padrões é feito por meio de expressões **`match`**, que à primeira vista assemelham-se muito ao tradicional comando `switch/case` presente em linguagens como C e Java. A forma geral de uma expressão **`match`** é:

```
VALUE match {
  case PATTERN_1 => EXPR_1
  case PATTERN_2 => EXPR_2
  ...
  case PATTERN_N => EXPR_N
  case _ => DEFAULT
}
```

Nessa expressão, o programa tentará fazer um casamento entre `VALUE` e cada um dos padrões, de cima para baixo. Caso algum casamento seja bem sucedido, a expressão à direita da seta é avaliada e seu valor é devolvido como valor da expressão **`match`**. O *underscore* é um padrão "coringa", que casará com qualquer valor.

O grande poder dessas expressões está nos diferentes tipos de padrões que podem ser construídos, como os exemplificados no trecho abaixo:

```
def describe(x: Any): String = x match {
  case 10 => "Número 10"
  case true => "Booleano verdadeiro"
  case Nil => "Lista vazia"
  case List(x) => "Lista com um elemento: " + x
  case head :: tail => "Lista com cabeça " + head + " e cauda " + tail
  case s: String => "Uma String de tamanho " + s.length
  case x: Int if x > 0 => "Um número positivo"
  case anything => "Alguma outra coisa: " + anything
}
```

Além desses, existe ainda um tipo de padrão muito usado em Scala, que permite fazer o casamento com instâncias de classes definidas pelo programador. Essa forma de casamento de padrões requer que as classes envolvidas sejam *case classes*, isto é, que as definições dessas classes sejam precedidas pela palavra reservada `case`. Considere, por exemplo, a definição de uma classe para a representação de expressões matemáticas e a definição de um método para avaliar essas expressões:

```
// Definicao da classe
case class Expression(operator: String, op1: Int, op2: Int)

def eval(expr: Expression): Int = expr match {
```

[4] Em Erlang, todas as variáveis são imutáveis, como as declaradas com **`val`** em Scala.

```
  case Expression("+", x, 0) => x
  case Expression("+", x, y) => x + y
  case Expression("*", x, y) => x * y
  case Expression("/", x, y) if x == y => 1
  case _ => 0
}
```

Apesar da simplicidade do exemplo, não é difícil perceber que a combinação de *case classes* e casamento de padrões pode ser muito útil, diminuindo drasticamente a quantidade de código necessário para a resolução de certos tipos de problemas.

Além de funcionarem muito bem em casamentos de padrões, as *case classes* também possuem características peculiares, que facilitam sua manipulação. Talvez a mais notável delas é que uma *case class* define automaticamente um *método de fábrica* com o nome da classe, o que permite a instanciação de *case classes* sem o uso de **new**. Uma expressão poderia ser instanciada da seguinte forma:

```
val expr = Expression("+", 21, 21)
```

Outra característica interessante das *case classes* é que os parâmetros de uma *case class* classe (`operator`, `op1` e `op2`, em nosso exemplo) são automaticamente considerados como `vals`. Assim, instâncias de uma *case class* definida apenas com os parâmetros de seu construtor primário (como em nosso exemplo) serão sempre objetos imutáveis, já que todo o estado das instâncias estará armazenado em campos imutáveis.

2.3.7 Variáveis preguiçosas

Scala permite que variáveis sejam definidas como preguiçosas (*lazy*). Uma variável preguiçosa tem atribuída a ela uma expressão no momento de sua declaração. Essa expressão, no entanto, só será avaliada no momento em que a variável for usada pela primeira vez.

Deve-se ressaltar que apenas variáveis imutáveis (aquelas declaradas com **val**) podem ser definidas como preguiçosas. O exemplo a seguir mostra como esse tipo de variável pode ser usado:

```
var a = 1
var b = 0
lazy val c = a / b;

def increment() = {
  b = b + 1
  println(c)
}
```

Caso a expressão atribuída a `c` fosse avaliada logo após a inicialização das outras variáveis (o que aconteceria se `c` não fosse *lazy*), um erro de execução ocorreria por conta da divisão por 0. No entanto, como a variável é preguiçosa, seu valor só será calculado no momento de seu primeiro uso. Isso ocorre quando imprimimos a variável dentro do método `increment()`. Nesse instante,

porém, a variável `b` já não vale mais 0, e portanto a execução ocorrerá normalmente.

Capítulo 3

Atores

Por muitos anos, a maneira como a indústria de processadores cumpria a chamada lei de Moore[1] tinha como consequência aumentos sucessivos na frequência de relógio das CPUs. Cada vez que um desses aumento ocorria, as aplicações se tornavam naturalmente mais velozes, sem que nenhum esforço adicional fosse exigido dos desenvolvedores. A melhora no desempenho vinha como uma consequência natural da maior rapidez com que as instruções eram executadas pelo processador.

No entanto, nos últimos anos o que se tem observado é uma tendência totalmente diferente na evolução dos processadores. A lei de Moore ainda é válida, e deverá continuar valendo por uma ou duas décadas, porém ela não se manifesta mais pelo aumento da frequência de relógio, mas sim pelo número de núcleos de um processador. Os processadores de computadores pessoais já há alguns anos não costumam ter uma frequência maior que 2 ou 3 GHz. A maioria deles possui, porém, pelo menos 2 núcleos de processamento.

Essa revolução na maneira de se construir *hardware* implica, no entanto, numa revolução ainda mais profunda e importante na maneira de se construir *software*. Os desenvolvedores não podem mais simplesmente se acomodar e esperar que o aumento na tecnologia dos processadores torne suas aplicações cada vez mais rápidas. Em outras palavras, o "almoço grátis" acabou [33].

Para se adaptar a essa nova realidade, é preciso construir aplicações concorrentes. Somente uma aplicação preparada para executar diversas tarefas simultaneamente poderá explorar o poder das CPUs com vários núcleos. Todavia, ainda são raros os exemplos de programas realmente capazes de tirar pleno proveito dos benefícios da computação concorrente.

Isso se dá em parte pela dificuldade em escrever programas concorrentes com as tecnologias mais populares atualmente. Em Java, por exemplo, é tido como difícil e altamente sujeito a erros o uso de travas para sincronizar diferentes *threads* que fazem acesso à memória compartilhada. Fazem-se necessários novos conceitos, abstrações e ferramentas mais apropriadas ao desenvolvimento de sistemas concorrentes [34].

A seguir, estudaremos uma dessas possíveis soluções, o modelo de atores. Na Seção 3.1, descreveremos de maneira sucinta o modelo proposto inicialmente, numa abordagem mais teórica. Já na Seção 3.2 mostraremos como algumas linguagens implementaram de fato esse modelo e como é possível usá-lo na resolução de problemas.

[1] Em termos gerais, a lei de Moore prevê um crescimento exponencial no número de transistores em um *chip* ao longo dos anos, mantendo-se o custo de fabricação.

3.1 Modelo de atores

O modelo de atores, proposto inicialmente por Hewitt *et al.* [21], formaliza uma maneira de se escrever programas concorrentes. A unidade básica de programação são os atores, entidades que possuem um comportamento bem definido e se comunicam exclusivamente por meio de troca de mensagens assíncronas.

Um ator encapsula seus dados, de maneira que somente ele pode ter acesso a esses dados. A menos que um desenvolvedor exponha indevidamente o estado de algum ator ao definir como esse ator processará as mensagens recebidas, o conceito de memória compartilhada deixa de existir no modelo de atores. Dessa forma, a execução concorrente ou paralela dos atores fica trivial, já que eles podem rodar simultaneamente sem o risco do sistema atingir um estado inconsistente devido à alguma intercalação imprópria de ações.

Gul Agha [4] defendeu o uso desse modelo no desenvolvimento de linguagens voltadas à implementação de sistemas concorrentes, argumentando que a utilização de atores tornaria mais fácil a criação de aplicações concorrentes e eliminaria problemas comumente existentes nesse tipo de cenário.

No modelo de Agha, cada ator possui uma fila de mensagens recebidas. A única garantia oferecida pelo modelo é que toda mensagem seja entregue a seu destinatário em algum momento do futuro. Um ator segue um comportamento previamente definido, que diz como ele deve reagir às mensagens recebidas. Naturalmente, todas as mensagens são enviadas assincronamente. Ao processar uma mensagem, um ator pode:

1. Criar um número finito de novos atores;
2. Enviar um número finito de mensagens para outros atores;
3. Modificar o seu próprio comportamento.

Na formalização de Agha, após o processamento de uma mensagem um ator deve necessariamente trocar seu comportamento antes de aceitar a próxima. Na prática, porém, muitas vezes esse novo comportamento será igual ao anterior, e a implementação da linguagem de atores pode tratar isso de forma a evitar um custo adicional desnecessário.

3.2 Implementação

Juntamente com a formalização de seu modelo, Gul Agha definiu algumas construções básicas para uma linguagem minimal de atores [3, 4]. Ele definiu ainda uma linguagem de atores "pura", na qual os tipos primitivos e até mesmo as mensagens são atores [4].

Linguagens definidas subsequentemente implementaram de diferentes formas uma abstração de atores similar à do modelo de Agha. Em algumas, os atores são o elemento básico da linguagem, como no trabalho de Agha, porém em outras eles são apenas uma dentre as várias ferramentas disponíveis ao programador.

Nas próximas seções, serão apresentadas as implementações de atores que tem maior relação com nosso trabalho.

3.2.1 Erlang

A linguagem Erlang foi criada dentro da empresa Ericsson, com um objetivo bastante claro: desenvolver aplicações distribuídas, concorrentes, altamente tolerantes a falhas e que rodem de forma praticamente ininterrupta.

Erlang é uma linguagem funcional, com tipagem dinâmica e com variáveis de atribuição única, que não podem ter seus valores modificados. Essa última característica elimina a existência de estado mutável nos programas e, portanto, promove a criação de aplicações paralelizáveis.

Na terminologia de Erlang, o conceito de ator é denominado *processo*. Os processos de Erlang assemelham-se, de certa forma, às *threads* de linguagens como Java, porém são muito mais leves: num computador comum, é possível criar milhares ou até milhões de processos.

Um processo funciona avaliando uma função, passada no momento da criação do processo (como um literal funcional). Essa função, portanto, representa o comportamento do ator, de acordo com o modelo de Agha. Além disso, processos interagem exclusivamente por meio da troca assíncrona de mensagens, e portanto não existe nenhum tipo de memória compartilhada entre eles.

Existem três primitivas básicas que permitem a manipulação de processos:

1. `Pid = spawn(Fun)` - Cria um novo processo que irá avaliar a função anônima `Fun`, um literal funcional passado na sintaxe de Erlang. O valor `Pid` devolvido é um identificador único desse processo e será usado no envio de mensagens a ele.

2. `Pid ! Message` - Envia a mensagem `Message` ao processo `Pid`, retornando imediatamente.

3. **`receive`** `PATTERNS` **`end`** - Recebe mensagens enviadas ao processo. `PATTERNS` define, por meio de uma sequência de padrões, as mensagens que esse processo deseja tratar. Essa sequência de padrões tem uma sintaxe muito semelhante à das expressões *match* de Scala (Seção 2.3.6): para cada padrão é definida uma ação, que será executada quando a mensagem recebida casar com aquele padrão.

Essa sintaxe reduzida permite escrever aplicações de modo conciso porém com grande potencial, principalmente no que diz respeito à execução concorrente dos programas. A própria máquina virtual de Erlang irá se encarregar de escalonar os processos de maneira eficiente, inclusive fazendo uso de múltiplos processadores e/ou múltiplos núcleos quando em máquinas com tais características.

Atores distribuídos

A combinação do modelo de atores com a inexistência de estado mutável faz de Erlang uma linguagem bastante apropriada para o desenvolvimento de aplicações concorrentes. Todavia, é quando os atores são distribuídos em diferentes computadores que o poder dessa linguagem torna-se realmente notável.

Basicamente, existem duas maneiras de se programar aplicações distribuídas em Erlang:

- **Erlang distribuído:** Trata-se de uma abordagem que é usada quando se deseja executar a aplicação num conjunto de nós de confiança (não-maliciosos). Nesse caso, cada nó deve iniciar uma máquina virtual com algumas especificações particulares, para que se forme um aglomerado com todos os computadores envolvidos. Além disso, a função `spawn` pode receber um parâmetro adicional, informando em que nó do aglomerado um determinado processo deve ser

iniciado. Assim, processos podem ser criados remotamente, em qualquer máquina do aglomerado. A criação de um processo remoto produz um PID, de maneira idêntica ao `spawn` de um processo local. O envio e a recepção de mensagens podem ser feitos como no caso local.

- ***Sockets:*** É uma abordagem que emprega *sockets* TCP/IP para a comunicação entre processos. Dessa forma, a comunicação remota entre processos não é tão transparente como no caso de Erlang distribuído, já que algumas bibliotecas adicionais devem ser usadas. Por outro lado, é uma solução mais segura e apropriada para o caso de ambientes onde nem todos os nós são confiáveis, como a Internet.

Uma importante diferença entre os atores distribuídos de Erlang e os atores de nossa infraestrutura é que, em Erlang, na criação de um ator deve-se sempre especificar em qual nó aquele ator será executado. Além disso, vale ressaltar que não é possível realizar migração dos atores distribuídos de Erlang.

3.2.2 Scala

Como vimos no Capítulo 2, programas escritos em Scala podem utilizar de maneira bastante direta bibliotecas escritas em Java. Dessa forma, é plenamente possível programar concorrentemente usando o modelo de Java, ou seja, empregando diretivas de sincronização (`synchronize`) para coordenar o acesso a dados compartilhados por diferentes *threads*.

Porém, essa abordagem para lidar com concorrência é amplamente questionada. Muitos consideram bastante difícil escrever programas concorrentes em Java que estejam garantidamente isentos de problemas como condições de corrida ou *deadlocks* [30, 38, 39].

Assim, os autores de Scala buscaram oferecer uma alternativa mais conveniente e menos sujeita a erros para se programar concorrentemente. Inspirados em Erlang, desenvolveram uma biblioteca que implementa o modelo de atores e oferece primitivas muito semelhantes às existentes naquela linguagem.

Programando com atores em Scala

Os atores de Scala [2], por serem implementados numa biblioteca e não como uma funcionalidade nativa, não garantem o fim dos problemas com concorrência existentes em Java. Um ator continua sendo um objeto, passível de conter estado mutável e de interagir com outros objetos por outros meios que não a troca de mensagens.

Portanto, como a linguagem não garante o cumprimento de certas exigências do modelo de atores (diferentemente de Erlang), é importante que o programador adote as seguintes práticas de programação [30]:

- **Comunicação somente via mensagens:** A linguagem não impede que um ator manipule direta ou indiretamente os atributos de outro, já que atores são objetos comuns. Essa prática, porém, deve ser evitada, ou o programa volta a ficar suscetível aos problemas de concorrência relacionados à memória compartilhada.

- **Mensagens imutáveis:** As mensagens que trafegam entre os atores podem ser objetos quaisquer de Scala. O ideal, no entanto, é que elas sejam objetos imutáveis, de modo a evitar o

compartilhamento de estado mutável. Mensagens imutáveis facilitam a paralelização e evitam condições de corrida. Uma forma de garantir a imutabilidade das mensagens é fazer com que elas sejam instâncias de classes que possuam apenas campos do tipo **`val`**. Além disso, é muito comum a utilização de *case classes* (Seção 2.3.6) como mensagens, já que estas em geral também são imutáveis.

Mantendo esses cuidados especiais em mente, a abstração de atores em Scala permite o desenvolvimento de sistemas concorrentes menos suscetíveis a erros. A sintaxe, como já foi dito, é fortemente inspirada na de Erlang. Graças ao modelo unificado de Scala, que não faz distinção entre métodos e operadores, os atores de Scala têm a aparência sintática de um recurso nativo da linguagem.

Um ator em Scala é definido por uma classe que é combinada com a feição `Actor` e que implementa o método abstrato `act()`, o qual define o comportamento do ator. Dentro desse método, utiliza-se o método `receive` para receber e processar mensagens. A possibilidade de chamar `receive` passando um argumento entre chaves (Seção 2.1.3) torna o código bastante claro, como mostra o exemplo da Listagem 3.1.

```
class MyActor extends Actor {
  def act() {
    println("Ator iniciado!")
    receive {
      case msg => println("Recebi: " + msg)
    }
  }
}
val actor = new MyActor
actor.start
```

Listagem 3.1: *Exemplo bastante simples de um ator de Scala.*

Ao ser iniciado, mediante uma chamada ao método `start`, um ator começa a executar o método `act()` e torna-se apto a receber mensagens. No caso padrão, cada ator de Scala é executado em uma *thread* diferente, e portanto todos os atores trabalham concorrentemente.

A Listagem 3.2 exibe um exemplo típico da manipulação de atores em Scala. Ela mostra uma forma mais concisa para se instanciar um ator sem ter de escrever uma classe para ele. Nas linhas 6 e 18, atores com implementações anônimas são criados por meio do método `actor`, cujo argumento é o corpo do método `act()` do novo ator. Nessas duas chamadas a `actor`, tal argumento é um bloco de código envolvido por chaves. Um detalhe importante é que os atores criados por `actor` são iniciados automaticamente, dispensando o uso de `start`.

Ainda na Listagem 3.2, nota-se o uso de *case classes* como mensagens, uma prática comum em Scala. Além disso, o **`while`** da linha 9 faz com que o ator fique repetidamente esperando uma mensagem do tipo `ResolveName`. Por outro lado, na linha 22, o fato do `receive` não estar dentro de um laço significa que a execução do ator se encerra após o processamento da primeira mensagem recebida.

```
1  case class Address(val host: String, val port: Int)
2  // Mensagens:
3  case class ResolveName(name: String, replyTo: Actor)
4  case class ReplyAddress(name: String, address: Option[Address])
5
6  val nameServerActor = actor {
7    var namesTable = HashMap[String, Address]()
8
9    while (true) {
10     receive {
11       case ResolveName(name, sender) => {
12         sender ! ReplyAddress(name, namesTable.get(name))
13       }
14     }
15   }
16 }
17
18 val clientActor = actor {
19   val nameToResolve = "someOtherActor"
20   nameServerActor ! ResolveName(nameToResolve, self)
21
22   receive {
23     case ReplyAddress(nameToResolve, optAddress) => optAddress match {
24       case Some(address) => ... // fazer algo com o endereço recebido
25       case None => // nome não encontrado
26     }
27   }
28 }
```

Listagem 3.2: *Utilização de atores em Scala.*

Implementação da biblioteca

No exemplo da Listagem 3.2, cada ator é associado a uma *thread* da máquina virtual Java, que por sua vez corresponde a uma *thread* do sistema operacional. Isso garante a concorrência, mas gera uma séria limitação, já que essas *threads* não são tão leves quanto se gostaria. Erlang, por exemplo, só é capaz de permitir a criação de milhões de atores por não associar cada um deles a processos ou *threads* do sistema operacional. Porém, dada a implementação da JVM, a quantidade de atores que se pode criar fica extremamente limitada em Scala, em geral não passando da ordem de alguns milhares.

Para resolver essa limitação, Haller [17, 18] propôs uma implementação de atores que respondessem a eventos e não precisassem estar atrelados a uma mesma *thread* durante toda sua execução. Mesmo sendo baseados em eventos, esses atores são escritos num estilo que não envolve inversão de controle e é basicamente igual ao dos atores baseados em *threads*.

Na implementação de Haller, a função de recebimento de mensagens (equivalente ao `receive`) nunca termina normalmente. Ao invés disso, a função sempre lança uma exceção, que será silenciosamente capturada pela biblioteca de atores e tomada como um sinal para dissociar o ator de sua *thread* de execução. Antes de lançar tal exceção, porém, ela salva, como um atributo do ator, um objeto com informações que conceitualmente correspondem à lista de padrões de mensagens esperadas e suas respectivas ações. Posteriormente, quando uma mensagem for enviada a esse ator, ela será comparada com os padrões naquela lista e, caso exista um casamento, a ação associada será

agendada para ser executada por alguma *thread* de um conjunto (*pool*) de *threads*. Dessa forma, um número limitado de *threads* se revezam no processamento das mensagens de um número possivelmente bem maior de atores, superando portanto a limitação da implementação baseada em *threads*.

Atualmente, a biblioteca padrão de atores de Scala oferece suporte tanto para atores baseados em *threads* como para atores baseados em eventos [19]. Sob o ponto de vista do programador, a diferença básica entre esses dois tipos de atores está na função usada para receber mensagens. Enquanto para atores baseados em *threads* usa-se o comando `receive`, os atores baseados em eventos usam o comando `react`, ilustrado no exemplo abaixo:

```
val eventActor = actor {
  loop {
    react {
      case msg: String => {
        println("Receive string: " + msg)
      }
    }
  }
}

eventActor ! "Scala" // Imprime a string
eventActor ! "Actors" // Imprime novamente
```

Um detalhe importante aqui é que, como a função `react` lança uma exceção que libera a *thread* que a executa, ela nunca retorna normalmente, e portanto nenhum código que vier depois da chamada a `react` será executado. Dessa forma, colocar uma chamada a `react` dentro de um **`while`**, como fizemos com a chamada a `receive` no primeiro ator do exemplo da Listagem 3.2, não faria com que o ator permanecesse recebendo mensagens indefinidamente. Para resolver esse problema, a biblioteca de atores de Scala inclui um método `loop`, que força a execução repetida do trecho compreendido entre as chaves mesmo na presença de `react`.

Por fim, vale ressaltar que embora os atores baseados em eventos sejam muito mais leves e escaláveis, algumas situações costumam exigir o uso dos atores baseados em *threads*. Uma dessas situações ocorre quando atores fazem chamadas bloqueantes (de entrada/saída, por exemplo). Assim, é interessante que a biblioteca de Scala ofereça ambas as opções para os programadores.

Atores distribuídos

Assim como em Erlang, é possível escrever em Scala aplicações que usem atores espalhados em diferentes computadores. A biblioteca padrão de Scala possui também suporte para a comunicação com atores remotos de maneira razoavelmente transparente.

A implementação de atores distribuídos usa o protocolo TCP/IP, por meio das classes `Socket` e `ServerSocket` de Java, mas permite também o emprego de outros protocolos sem a necessidade de grandes modificações [18].

O fragmento de código na Listagem 3.3 modifica os atores da Listagem 3.2 para que eles se comuniquem remotamente. Esse fragmento mostra o uso de três funções que devem ser empregadas quando se trabalha com atores distribuídos: `alive`, `register` e `select`.

```
1  val nameServerActor = actor {
2    ...
3    alive(8001)
4    register('server, self)
5
6    while (true) {
7      receive { ... }
8    }
9  }
10
11 val clientActor = actor {
12   val nameServerActor = select(Node("somehost", 8001), 'server)
13   val nameToResolve = "someOtherActor"
14   nameServerActor ! ResolveName(nameToResolve)
15
16   receive { ... }
17 }
```

Listagem 3.3: *Utilização de atores remotos em Scala.*

Nas linhas 3 e 4, o ator servidor torna-se remotamente acessível, especificando qual a porta que deseja utilizar para a comunicação e registrando-se sob um nome arbitrariamente escolhido. Já na linha 12, o ator cliente ganha acesso a um representante (*proxy*) do ator remoto, fornecendo para tanto o *hostname* (`somehost`) e porta daquele ator e o nome sob o qual ele está registrado.

O modo de utilização de atores distribuídos em Scala é mais restritivo do que em Erlang. Isso ocorre porque não é possível realizar a instanciação remota de um ator, ou seja, não existe uma diretiva como `spawn` que permita especificar o nó onde o ator deverá ser executado. Os atores remotos de Scala podem ser instanciados apenas pelo servidor que irá hospedá-los.

Na Seção 5.2, iremos estudar uma outra implementação de atores, também escrita em Scala. Essa implementação permite que atores distribuídos sejam instanciados remotamente. Em outras palavras, ela permite algo semelhante ao `spawn` remoto de Erlang, que recebe o nó que deverá hospedar o ator. Essa outra implementação de atores em Scala servirá de base para os atores móveis desenvolvidos neste trabalho.

Capítulo 4

Funcionamento da infraestrutura

A infraestrutura que desenvolvemos neste trabalho tem como objetivo principal facilitar a tarefa de distribuir, numa rede de computadores, uma aplicação baseada em atores. Basicamente, nosso sistema ocupa-se com a administração desses atores, gerenciando em particular a localização dos atores no conjunto de nós onde a aplicação é executada. Esse gerenciamento ocorre em dois momentos distintos:

1. Inicialmente, quando os atores são criados, a infraestrutura decide, de maneira autônoma, em que nó da rede cada ator será executado. Dessa forma, os atores se distribuem automaticamente no conjunto de computadores, sem a necessidade de medidas adicionais por parte do desenvolvedor.

2. Posteriormente, durante a execução de um ator, ele será capaz de migrar para um computador diferente daquele onde ele está atualmente sendo executado. A infraestrutura implementa esse mecanismo de migração de forma transparente, ou seja, o programador não precisa se preocupar com a localização de um determinado ator após uma migração. Basta que ele continue usando o ator normalmente, e a plataforma se encarregará de entregar as mensagens corretamente ao ator no seu novo endereço.

Dessa forma, desenvolvedores podem criar aplicações distribuídas preocupando-se apenas com a definição dos atores participantes, e o sistema se encarregará de distribuí-los automaticamente nos nós disponíveis. Isso contribui, em particular, para a escalabilidade da aplicação, já que o sistema espalha automaticamente os atores por todos os nós disponíveis, sem a necessidade de alterações no código da aplicação. Isso é especialmente interessante no cenário atual de aplicações de Internet, onde muitas vezes as taxas de utilização de um serviço aumentam rapidamente num curto espaço de tempo.

Além disso, o mecanismo de migração de atores permite a construção de sistemas capazes de se adaptarem dinamicamente a mudanças em seu ambiente de execução. Algoritmos de distribuição de carga, por exemplo, poderiam ser implementados sobre nossa infraestrutura, usando a migração de atores para modificar o modo como os recursos (computadores) são usados e assim tentar aumentar o desempenho das aplicações.

Neste capítulo, primeiramente daremos uma introdução bastante sucinta à arquitetura da infraestrutura, mostrando quais são seus principais componentes. Em seguida, mostraremos os principais passos envolvidos na utilização da infraestrutura, de forma semelhante a um “manual de

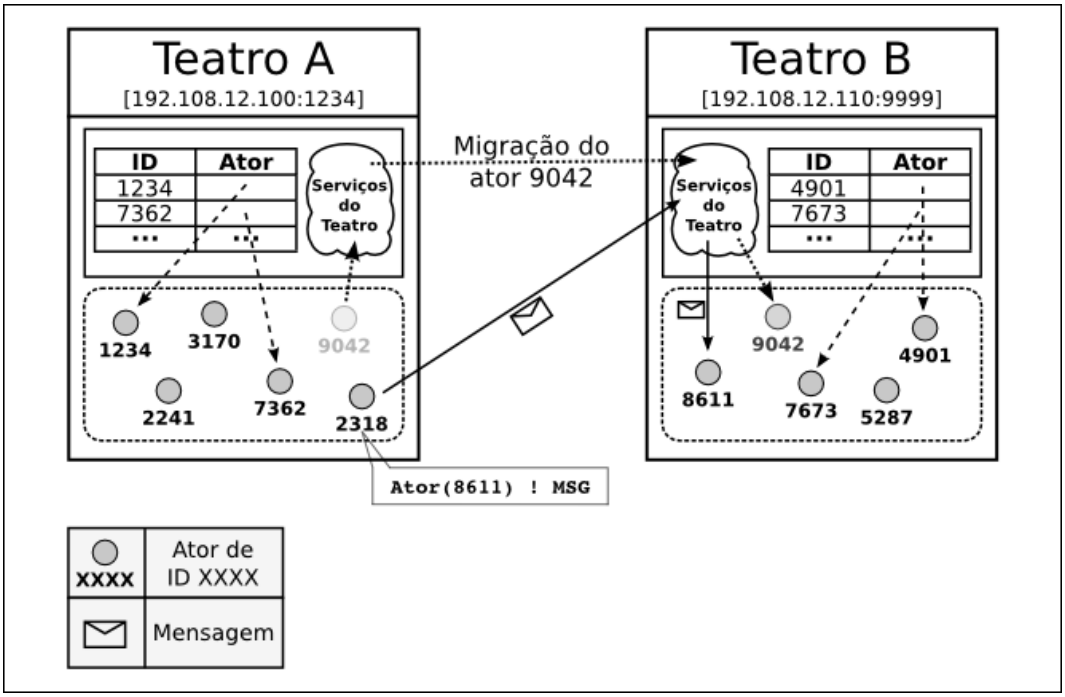


Figura 4.1: *Arquitetura simplificada da infraestrutura, contendo os dois componentes principais: atores móveis e teatros.*

usuário". Isso nos permitirá conhecer todos os serviços oferecidos para, nos próximos capítulos, estudarmos detalhes de sua implementação.

4.1 Principais componentes

A Figura 4.1 exibe uma versão bastante simplificada da arquitetura do sistema, dando destaque para seus dois componentes principais: os atores móveis (representados pelos círculos) e os teatros.

Atores móveis

Um ator móvel é, essencialmente, um ator normal (Capítulo 3), porém com a capacidade adicional de, após ter sido iniciado, interromper sua execução, migrar para outro nó, e então passar a rodar nesse novo computador, retomando a execução de onde havia parado.

Como observaram El Maghraoui *et al.* [25], a estrutura de um ator o torna um ótimo candidato para a migração, já que ele mantém seus dados encapsulados e responde apenas a mensagens assíncronas. Assim, a qualquer momento a migração pode ser efetuada, bastando que todo o estado do ator (incluindo sua caixa de mensagens) seja salvo e enviado ao nó de destino.

A migração de código em sistemas computacionais já foi bastante estudada na literatura. Em geral, ela costuma ser classificada em dois tipos [16]:

- **Migração forte:** Envolve a migração completa de um determinado processo. Isso envolve migrar o código, dados (como valores de variáveis) e o estado da execução. Este último diz respeito ao contexto do processo, isto é, tudo o que é necessário para que o processamento continue, no nó de destino, exatamente do ponto em que ele foi interrompido no nó de origem. Tipicamente, isso inclui dados como a pilha de execução e o apontador de instruções, entre outros.

A vantagem desse tipo de migração é que ela pode interromper a execução de um programa em qualquer ponto, sem a necessidade de interferência do desenvolvedor. Além disso, a aplicação continua sua execução no nó de destino exatamente de onde parou. Por outro lado, implementar esse tipo de migração pode ser bastante complexo, e a migração em si pode ter um custo superior do que uma forma menos restritiva de migração. Na JVM, por exemplo, não temos acesso ao conteúdo da pilha de execução, e por esse motivo a migração forte costuma ser implementada por meio de instrumentação de *bytecode*.

- **Migração fraca:** Nesse caso, apenas o código e os dados são migrados. Isso significa, em geral, que o código da aplicação deverá conter chamadas explícitas à infraestrutura de migração. Tais chamadas permitem identificar pontos no código onde a migração pode ocorrer. Já que o estado da execução (como a pilha) não será salvo, não é possível que a migração ocorra em qualquer instante da execução. Os momentos em que ela pode ocorrer devem ser demarcados explicitamente pelo programador.

  Além disso, deve-se de alguma maneira especificar como a execução será retomada no nó de destino. Como o estado da execução é "perdido" no nó de origem do processo, é impossível continuar a execução exatamente de onde ela foi interrompida. Ao invés disso, a execução continuará a partir de algum ponto definido pelo desenvolvedor, como por exemplo uma função `afterMigration()`.

  Apesar de não oferecer a mesma transparência que a migração forte, a implementação da migração fraca é muito mais simples e direta. Na JVM, por exemplo, ela pode ser realizada usando-se apenas o mecanismo de seriação de objetos.

Como o processamento de mensagens de um ator ocorre de forma sequencial, o instante em que a migração ocorre é bem definido: assim que o ator processar uma mensagem especial que solicite sua mudança de nó. Assim, é como se os atores definissem naturalmente *checkpoints* nos quais a execução pode ser interrompida para dar início à migração. Ao chegar no novo nó, o ator retoma o processamento de mensagens exatamente do ponto em que havia parado.

Dessa forma, ainda que a migração de atores implementada em nosso modelo seja do tipo fraca, já que ela só pode ocorrer em momentos específicos, o código do ator não precisa conter definições explícitas dos pontos nos quais a migração pode ocorrer nem daqueles a partir dos quais a execução deve ser retomada. Do ponto de vista do programador, o código de um ator móvel é similar ao de um ator normal. Assim, os benefícios de transparência típicos da migração forte são alcançados com a eficiência e a simplicidade da migração fraca.

Como se observa na Figura 4.1, cada ator rodando na infraestrutura possui associado a si um campo que o identifica. Esse campo é um `UUID` (*universally unique identifier* – identificador único universal), único em toda a aplicação, que permite identificar o ator durante etapas como o envio de mensagens remotas ou a migração. Vale ressaltar que esse campo é de uso interno da infraestrutura e, portanto, o usuário não precisa manipulá-lo explicitamente (ainda que seja possível ter acesso a ele, caso se deseje).

Teatros

Já os teatros são os responsáveis pela "hospedagem" de um ator num determinado nó. Cada nó deve, obrigatoriamente, executar um teatro. Na Figura 4.1, vemos os teatros A e B, cada um deles

respondendo a requisições num endereço e porta previamente estabelecidos. Os teatros executam serviços como migração, entrega de mensagens remotas e registro de informações sobre troca de mensagens (*profiling*), entre outros.

Internamente, cada teatro deve manter uma tabela que contém todos os atores atualmente residentes naquele nó, e que tem como chave o campo `UUID`. Dessa forma, é possível o encaminhamento de mensagens vindas de outros nós. Podemos observar um exemplo disso na Figura 4.1, onde o ator de `UUID 2318` envia uma mensagem ao ator de `UUID 8611`, que no momento encontra-se num computador diferente. A mensagem é enviada então para o teatro que atualmente hospeda aquele ator. Ao recebê-la, esse teatro procura em sua tabela o ator destinatário da mensagem e a encaminha para ele.

Além disso, a migração de atores ocorre por meio da troca de mensagens entre os teatros envolvidos (origem e destino). Na Figura 4.1, o ator de `UUID 9042` está migrando do teatro A para o teatro B. O processo de migração, na realidade, envolve a troca de mais de uma mensagem entre os teatros, de modo a garantir que o estado do sistema seja mantido corretamente. Isso é importante para que não seja perdida nenhuma mensagem que tenha como destinatário o ator que está mudando de endereço.

Serviços adicionais

Além dos atores móveis e teatros, alguns serviços adicionais também existem na infraestrutura, com o propósito de dar apoio à correta execução da própria infraestrutura e/ou para uso por aplicações. Temos, por exemplo, os seguintes serviços:

Serviço de nomes: Armazena uma tabela relacionando cada ator atualmente em execução na aplicação com sua localização atual. É essencial para a entrega correta de mensagens a atores remotos, e precisa ser atualizado a cada migração. No caso de serviços de nomes distribuídos, essa tabela pode ser dividida entre diferentes nós. Nossa infraestrutura inclui um serviço de nomes distribuído, que pode ser configurado pelos usuários.

Serviço de monitoramento: Permite registrar informações sobre as trocas de mensagens entre os atores rodando no aglomerado, identificando assim aqueles atores que mais trocam mensagens entre si. Com essas informações, seria possível construir, sobre nossa infraestrutura, um sistema de distribuição de carga que, usando a migração de atores, buscasse juntar (colocando no mesmo nó) atores que trocam muitas mensagens, de modo a diminuir assim a quantidade de dados trafegando pela rede.

4.2 Preparação do aglomerado

O cenário mais típico de utilização de nossa infraestrutura, de acordo com nossa concepção, é a execução de aplicações em um aglomerado (*cluster*) de computadores, ou seja, um conjunto de máquinas, muitas vezes com configurações comuns, trabalhando juntas na resolução de um mesmo problema. Na verdade, a principal exigência de nosso sistema é que o desenvolvedor tenha algum controle sobre os computadores que serão utilizados para executar a aplicação. Isso porque a infraestrutura precisa conhecer todos os nós participantes, e é necessário que um teatro seja iniciado em cada um desses nós antes que a aplicação possa ser submetida à execução.

Assim, o primeiro passo para utilizar os serviços da infraestrutura é a preparação do aglomerado de computadores onde essa aplicação será executada. Essa preparação envolve duas etapas: escrita do arquivo de configuração e iniciação dos teatros nos nós.

4.2.1 Arquivo de configuração

O usuário deve preparar um arquivo de configuração, contendo uma descrição do aglomerado de computadores onde a aplicação será executada. Basicamente, essa descrição deve conter o *hostname* e a porta de cada um dos nós a serem utilizados. A Listagem 4.1 exibe um exemplo de um arquivo de configuração válido.

```
cluster {
  nodes = [ "node_1",
    "node_2"]

  node_1 {
    hostname = "example.host.com"
    port = 1234
    name-server on
  }

  node_2 {
    hostname = "192.108.12.110"
    port = 9999
    profiling off
  }

  # Configurações adicionais (opcionais)
}
```

Listagem 4.1: *Exemplo de arquivo de configuração que descreve os nós de um aglomerado.*

Nesse arquivo, diferentes blocos com configurações específicas são agrupados usando-se chaves, de maneira semelhante a blocos de código em muitas linguagens de programação. Todas as configurações relativas ao aglomerado devem estar contidas dentro de um bloco mais externo, denominado `cluster`. Dentro desse bloco, deve obrigatoriamente aparecer um campo chamado `nodes`, contendo uma lista com os nomes (arbitrários) de cada um dos nós a serem utilizados pela aplicação.

Em seguida, é necessário que, para cada um dos nomes de nós definidos, exista um bloco com esse mesmo nome para descrever aquele nó. Nessa descrição, dois campos são obrigatórios: `hostname`, com o nome da máquina ou seu endereço IP, e `port`, com o número da porta pela qual ocorrerão as comunicações.

Além dessas duas informações, dentro desse bloco podem aparecer também outros campos, opcionais, que especificam configurações particulares daquele nó. O campo `name-server`, por exemplo, indica se o nó deve ou não executar uma parte do serviço de nomes distribuído[1]. A omissão de algum campo opcional no arquivo de configuração faz com que um valor padrão seja assumido para o nó.

Após a descrição de cada um dos nós, ainda podem aparecer configurações adicionais, que permitem modificar diferentes aspectos da execução da infraestrutura. Essas configurações são

[1] Na Seção 6.3 discutiremos mais sobre o serviço de nomes distribuído de nossa infraestrutura.

opcionais e, caso não apareçam, valores padrões serão usados. A Seção 4.6 contém uma relação completa dos campos que podem aparecer no arquivo de configuração da infraestrutura.

4.2.2 Iniciação dos teatros

Com o arquivo de configuração pronto, o próximo passo para se executar uma aplicação na infraestrutura será a iniciação de um teatro em cada um dos nós especificados no arquivo de configuração. Isso permitirá que todos os teatros se comuniquem e os atores sejam espalhados por todo o aglomerado.

Dentro de cada máquina virtual Java deve existir apenas um teatro sendo executado. Esse teatro, também chamado de *teatro local*, poderá ser referenciado, dentro do código, por meio do objeto *singleton* `LocalTheater`. Mais de um teatro pode ser executado num mesmo nó, desde que em máquinas virtuais diferentes e associados a portas diferentes. Todavia, acreditamos que o caso mais comum será o de apenas um teatro por máquina.

A forma mais fácil de iniciar um teatro é usando uma das variantes do método `startTheater`, dentro do objeto *singleton* `Mobile`. A Listagem 4.2 mostra as três versões desse método, que diferem pelos tipos de parâmetros que recebem:

- No primeiro caso, o teatro é iniciado de acordo com o nome do nó passado como parâmetro. Esse nome deve corresponder ao nome de algum dos nós descritos no arquivo de configuração.
- No segundo caso, o teatro é iniciado no *hostname* e porta passados como parâmetros. É necessário que essas informações correspondam a algum dos nós descritos no arquivo de configuração.
- No terceiro caso, nenhum argumento é passado ao método. Ele tentará definir qual teatro iniciar com base no nome da máquina onde o sistema está rodando. Caso exista a descrição de algum nó com esse *hostname* no arquivo de configuração, esse nó (ou algum deles, no caso de existir mais de um) será iniciado.

```
def startTheater(nodeName: String): Boolean

def startTheater(hostname: String, port: Int): Boolean

def startTheater(): Boolean
```

Listagem 4.2: *Métodos que permitem iniciar um teatro no nó local.*

O método `startTheater` devolverá um valor de verdadeiro ou falso, indicando se o teatro foi iniciado corretamente ou não, respectivamente. É importante ressaltar que esse método deve sempre ser usado para iniciar um teatro local, ou seja, ele deve ser chamado com parâmetros que identifiquem, no arquivo de configuração, a máquina onde o método será executado. Portanto, para cada nó do aglomerado, uma máquina virtual deverá ser iniciada e pelo menos uma chamada de método para iniciar um teatro deverá ser feita.

4.3 Criação de atores

Após as etapas de preparação do aglomerado, o desenvolvedor pode começar a escrever a aplicação que será executada sobre a infraestrutura. A ideia é que o código da aplicação não precise fazer referências diretas à localização dos atores ou mesmo ao aglomerado em si. Tudo que o programador deverá fazer é escrever o código dos atores e utilizá-los normalmente na resolução de seu problema. Todas as questões referentes à localização desses atores serão tratadas automaticamente. Nessa seção veremos como o desenvolvedor deve proceder para criar atores móveis.

4.3.1 A feição `MobileActor`

Para que os atores criados pelos usuários da infraestrutura tenham a capacidade de migrar, o primeiro passo é que eles sejam implementados como classes que estendam a feição `MobileActor`. Essa feição implementa funcionalidades que tornam possível o mecanismo de migração (cobriremos essas funcionalidades em mais detalhes na Seção 6.1.1).

Podemos resumir em três passos o processo de criação de um ator migrável:

1. Definir uma nova classe, que estenda `MobileActor`.
2. Fazer essa classe seriável, usando para isso a anotação `@serializable`.
3. Definir o comportamento desse ator, por meio do método `receive`, que deve cobrir todos os tipos de mensagens que aquele ator deseja tratar.

A Listagem 4.3 mostra um exemplo bastante simples da execução desses três passos. Como pode-se observar, a definição do comportamento do ator (método `receive`) é feita de forma bastante semelhante ao casamento de padrões de Scala (Seção 2.3.6).

```
@serializable
class MyMobileActor extends MobileActor {
  override def receive = {
    case str: String => println("Received a string: " + str)

    case _ => println("Unknown message received...")
  }

  override def beforeMigration() {
    println("Mobile actor preparing to migrate...")
  }

  override def afterMigration() {
    println("Mobile actor just migrated...")
  }
}
```

Listagem 4.3: *Criando um ator móvel que estende a classe `MobileActor`.*

Um detalhe adicional a se observar é a presença, no exemplo, dos métodos `beforeMigration()` e `afterMigration()`. Esses métodos são *callbacks*, e serão chamados automaticamente pela infraestrutura imediatamente antes e depois da migração, respectivamente. Por meio deles o desenvolvedor pode definir tarefas a serem executadas nos momentos adequados (por exemplo, fechar e

abrir arquivos ou conexões com um banco de dados). Caso o usuário não deseje executar nenhuma tarefa desse tipo, ele pode simplesmente omitir esses métodos.

4.3.2 Instanciação de atores

Tendo escrito o código dos atores móveis, o usuário deverá ser capaz de criar instâncias desses atores dentro da aplicação. Essa instanciação, porém, não deve ser feita da maneira tradicional de Scala, usando a palavra `new` seguida do nome da classe. O seguinte trecho de código, por exemplo, lançaria uma exceção:

```
val actor = new MyMobileActor
```

Ao invés disso, a criação de atores deve ser feita de forma indireta, por meio de métodos disponibilizados pela infraestrutura. Essa restrição tem duas motivações:

1. Ela permite uma separação entre a instância do ator propriamente dita e a referência para esse ator. Ao se criar um ator utilizando um dos métodos da infraestrutura, o que é devolvido é na verdade apenas uma referência para ele. A partir daí, toda a manipulação daquele ator (como o envio de mensagens) é feita utilizando-se essa referência. Isso é desejável segundo o modelo de atores, no qual o estado de um ator é completamente encapsulado e fica acessível exclusivamente via troca de mensagens.

2. No caso do gerenciamento da localização dos atores, a restrição permite que, no momento da criação, a infraestrutura decida autonomamente em que nó aquele ator deverá ser iniciado. Assim, o usuário apenas define os atores que deseja criar, sem precisar se preocupar com a distribuição desses atores no aglomerado, que ocorrerá de forma transparente e automática.

Para efetuar a instanciação de novos atores, o usuário possui duas alternativas: os métodos `launch` e `spawn`. O primeiro deles, `launch`, solicita à infraestrutura a instanciação de um novo ator sem especificar em que nó ele deverá ser executado. Caberá a infraestrutura escolher um nó, iniciar o ator e devolver uma referência para ele. Com essa referência, será possível enviar mensagens ao ator de forma totalmente transparente, não importando em que nó ele foi colocado.

O método `launch` existe em dois formatos, como mostra a Listagem 4.4. No caso do formato 1, a utilização é feita parametrizando-se a chamada ao método com o tipo de ator a ser criado[2]. O ator será instanciado com o seu construtor padrão, ou seja, sem argumentos.

Já o formato 2 é utilizado quando se deseja criar um ator por meio de um construtor que receba argumentos. Para isso, um trecho de código que instancie o ator deve ser passado como parâmetro ao `launch`. Essa abordagem deve ser usada apenas quando de fato necessária, já que ela pode ser mais custosa à infraestrutura[3].

A Listagem 4.5 mostra exemplos da criação de atores usando os dois modos descritos. Observe que, no caso 2, aparentemente está sendo feita uma instanciação de ator com **new**. No entanto, como mostra a Listagem 4.4, o método em questão recebe um parâmetro por nome (Seção 2.3.2). Dessa

[2] Não entraremos em detalhes sobre a palavra `Manifest`, que aparece na parametrização de tipo do método, por esse ser um recurso razoavelmente avançado da linguagem. De maneira resumida, porém, esse mecanismo permite driblar as limitações do *type erasure* de Java, que faz com que informações sobre tipos parametrizados não existam na JVM, em tempo de execução.

[3] A razão desse possível custo adicional será explicada na Seção 7.1.3.

forma, a chamada a **new** só será avaliada de fato dentro do método, e tal instanciação ocorrerá no único momento em que ela é permitida (sob o controle da infraestrutura).

```
1. def launch[T <: MobileActor : Manifest]: MobileActorRef

2. def launch(factory: => MobileActor): MobileActorRef
```

Listagem 4.4: *Assinaturas do método* `launch`.

```
1. val actorRef = launch[MyMobileActor]

2. val anotherRef = launch(new StatefulMobileActor(12))
```

Listagem 4.5: *Exemplos da criação de atores usando o método* `launch`.

Por outro lado, é possível também instanciar um ator especificando em que nó ele deverá ser colocado. Para isso, deve ser usado o método `spawn`. Esse método recebe parâmetros exatamente iguais aos do método `launch` (nome da classe ou construtor com argumentos), porém após a chamada ao método deve aparecer um especificador de localização, que pode ser `here` ou `at NODE`. No primeiro caso, o ator é instanciado localmente, enquanto que no segundo ele será colocado no teatro em execução no endereço `NODE`.

A Listagem 4.6 mostra exemplos de uso do método `spawn`. Essa listagem exibe também a definição da classe `TheaterNode`, usada para especificar endereços de teatros. Deve-se observar que a sintaxe de Scala permite a omissão de pontos e parênteses em chamadas de métodos (Seção 2.2.1), porém os pontos e parênteses poderiam ter sido incluídos sem qualquer alteração de resultado. A chamada com `at`, por exemplo, poderia ser trocada por `spawn(..).at(node)`.

```
case class TheaterNode(hostname: String, port: Int)

// Instancia ator no nó local
val localRef = spawn[MyMobileActor] here

// Instancia ator no nó especificado
val node = TheaterNode("tcoraini.mobileactors.com", 9999)
val remoteRef = spawn(new StatefulMobileActor(42)) at node
```

Listagem 4.6: *Exemplos de instanciação de atores especificando o nó onde colocá-los.*

4.3.3 Atores co-locados

No momento da instanciação dos atores, o programador pode desejar que um determinado grupo de atores sejam colocados num mesmo nó. Esses atores, chamados de *co-locados*, possivelmente trocarão muitas mensagens entre si, e portanto mantê-los num mesmo computador pode melhorar o desempenho da aplicação, já que diminuirá a quantidade de dados trafegando pela rede.

A instanciação de atores co-locados também é feita com os métodos `launch` e `spawn`. Assim como no caso de atores individuais, a diferença entre os dois é a especificação ou não do nó onde o

grupo de atores será executado, respectivamente.

A Listagem 4.7 exibe os dois tipos de parâmetros que podem ser passados ao método `launch` para realizar a instanciação de grupos de atores co-locados. No primeiro caso, é passado ao método o tipo de ator a ser instanciado e o número de atores a serem criados. Todos os atores serão instanciados de maneira exatamente igual, por meio de seu construtor padrão. O método devolve uma lista, de tamanho `number`, com as referências para todos os atores do grupo, que estarão necessariamente no mesmo nó.

Já no segundo caso, é passada ao método uma sequência de trechos de código que, ao serem avaliados, geram instâncias de atores móveis. Observe que o asterisco indica que o método recebe uma lista de parâmetros de tamanho variável (Seção 2.1.6). Como tal lista poderia ter tamanho zero, os dois primeiros parâmetros garantem que sejam passados ao menos dois construtores ao método. Essa versão do método `launch` permite que atores de tipos diferentes façam parte de um mesmo grupo, diferentemente do que ocorre com a primeira versão do método.

```
1. def launch[T <: MobileActor : Manifest](number: Int): List[
   MobileActorRef]

2. def launch(factory1: () => MobileActor,
              factory2: () => MobileActor,
              factories: (() => MobileActor)*): List[MobileActorRef]
```

Listagem 4.7: *Assinaturas do método `launch` usadas para instanciar grupos de atores co-locados.*

A Listagem 4.8 mostra exemplos de utilização do método `launch` na instanciação de atores co-locados. Usuários de Scala mais experientes perceberão um aparente problema na utilização do segundo tipo de `launch`, que não parece passar parâmetros do tipo correto (os parâmetros deveriam ser literais funcionais). Porém, o comando `import Mobile._` coloca no escopo uma conversão implícita que resolve esse problema (Seção 2.2.5) e torna mais amigável o uso do método. O mecanismo de instanciação de atores co-locados será discutido em detalhes na Seção 7.1.4.

```
import Mobile._

// Devolve lista de tamanho 4
1. val colocatedActors = launch[MyMobileActor](4)

// Devolve lista de tamanho 3, com posições relativas aos parâmetros
// preservadas
2. val someOtherActors = launch(
     new StatefulMobileActor(10),
     new StatefulMobileActor(100),
     new StatefulMobileActor(1000))
```

Listagem 4.8: *Exemplos da criação de grupos de atores co-locados usando o método `launch`.*

Ao instanciar um grupo de atores co-locados, além de fazer com que todos eles sejam executados num mesmo nó, a infraestrutura também atribui a todos eles um mesmo identificador de grupo, único em toda a aplicação. Esse identificador é um atributo das referências para atores, chamado `groupId`. O `groupId` é do tipo `Option[String]` e, em atores instanciados individualmente,

seu valor é `None`.

O uso de `spawn` com atores co-locados

Assim como a instanciação individual de atores, a criação de grupos de atores co-locados também pode ser feita com o método `spawn`. Também nesse caso o método `spawn` recebe os mesmos argumentos que o método `launch`, e deve ser completado com os especificadores de localização `here` ou `at NODE`.

Além desses tipos de instanciação, por vezes pode ser interessante solicitar que um grupo de atores co-locados seja instanciado no mesmo nó que algum ator já existente. Isso pode ser útil quando sabemos que os novos atores irão se comunicar intensamente com o ator já existente.

Para esses casos, a infraestrutura dispõe de um terceiro tipo de especificador de localização: `nextTo REF`. O uso de `nextTo` é restrito a chamadas de `spawn` para atores co-locados. Ao passar como parâmetro ao método `nextTo` uma referência `REF` a um ator móvel A, a infraestrutura tomará duas providências:

1. Ela instanciará os atores do grupo no mesmo teatro que hospeda A.
2. Se A já fizer parte de um grupo de atores co-locados, a infraestrutura inserirá nesse mesmo grupo todos os atores instanciados. Caso contrário, ela criará um novo grupo contendo os novos atores e o ator A.

A Listagem 4.9 mostra um exemplo de instanciação de atores co-locados usando o especificador `nextTo`. O grupo de atores será instanciado no exato nó onde o ator de referência `ref` foi instanciado. Além disso, após a instanciação todos os quatro atores farão parte de um mesmo grupo (ou seja, terão o mesmo valor no atributo `groupId`).

```
// Instancia ator em algum nó
val ref = launch[MyActor]

// Instancia 3 atores juntos de 'ref'
val group = spawn[SomeActor](3) nextTo ref
```

Listagem 4.9: *Exemplo de uso de `spawn` para atores co-locados com o especificador `nextTo`.*

4.4 Migração de atores

4.4.1 A mensagem `MoveTo`

Após uma chamada ao método `launch` (ou `spawn`), um novo ator será iniciado em algum nó da rede, e ficará acessível por meio da referência devolvida por essa chamada. O envio de mensagens para esse ator, por exemplo, é feito usando-se o operador `!`, exatamente como nos atores originais de Scala (Seção 3.2.2).

Uma diferença importante em relação aos atores de Scala, no entanto, é a capacidade que os atores da infraestrutura têm de migrar de um nó para outro. Para isso, basta que se envie para um ator uma mensagem do tipo `MoveTo`, contendo o endereço (nome da máquina e porta) do teatro

```
val actorRef = spawn[MyMobileActor] here

actorRef ! "Mensagem para o ator no nó local"

actorRef ! MoveTo("tcoraini.mobileactors.com", 9999)

actorRef ! "Mensagem para o ator em seu novo endereço"
```

Listagem 4.10: *Envio de mensagem `MoveTo` a um ator, solicitando sua migração.*

para onde esse ator deve migrar. A Listagem 4.10 mostra um exemplo bastante simples da solicitação de migração de um ator. No caso, `tcoraini.mobileactors.com` é um nome hipotético para um nó que executa um teatro atendendo a requisições na porta `9999`.

Como toda a manipulação de atores, a solicitação de migração é feita também por meio do envio de uma mensagem de forma assíncrona ao ator. Uma diferença importante, no entanto, é que essa mensagem é tratada de forma prioritária: ela será a próxima mensagem a ser processada pelo ator, dando início à migração. Caso já existam outras mensagens na caixa de mensagens do ator, elas serão seriadas e enviadas juntamente com ele para o novo nó.

Além disso, é importante ressaltar que o tratamento de mensagens do tipo `MoveTo` é automaticamente efetuado pela feição `MobileActor`. Assim, o desenvolvedor não precisará se preocupar com mensagens desse tipo quando estiver escrevendo o método `receive` de seus atores.

Como vemos no exemplo da Listagem 4.10, logo após a solicitação de migração, uma nova mensagem é enviada ao ator, de forma exatamente igual à usada antes da migração. Isso demonstra que o processo de migração ocorre de maneira transparente para o usuário. A atualização da referência do ator, para que ela aponte para o novo endereço, é feita de forma automática pela infraestrutura. Assim, o envio de mensagens a um ator ocorre sempre da mesma maneira, independentemente de onde ele esteja localizado atualmente.

4.4.2 A mensagem `MoveGroupTo`

Como já foi explicado, certas vezes pode ser interessante especificar grupos de atores que devem ser instanciados num mesmo nó. Esse será o caso quando o desenvolvedor já sabe, de antemão, que um certo conjunto de atores trocará uma grande quantidade de mensagens, e portanto que será interessante que essas mensagens sejam locais.

No caso de atores co-locados, a migração deve ser analisada com mais cuidado. Por um lado, migrar um ator sozinho (desvinculando-o do grupo, portanto) pode levar a um aumento significativo no número de mensagens trafegando pela rede. Afinal, era justamente esse o problema que a instanciação de atores co-locados procurava resolver. Por outro lado, decidir migrar todo o grupo de atores pode acabar resultando num processo bastante custoso, dependendo do tamanho do grupo e da quantidade de informação que cada ator deverá levar consigo.

Portanto, a decisão de migração de atores integrantes de grupos co-locados deve ser tomada cuidadosamente. Em nossa infraestrutura, existem duas possibilidades: *migração individual* e *migração conjunta*. No caso da migração individual, o ator será removido do grupo e migrado sozinho para o nó especificado. Para obter esse tipo de comportamento, basta enviar uma mensagem do tipo `MoveTo` para o ator, exatamente como exemplificado na seção anterior.

Já no caso da migração conjunta, a infraestrutura irá migrar todos os atores participantes de um determinado grupo para o nó especificado. Para que isso ocorra, é necessário enviar para algum ator do grupo uma mensagem do tipo `MoveGroupTo(hostname, port)`, de maneira análoga à mensagem `MoveTo`. Basta que essa mensagem seja enviada apenas uma vez, para qualquer um dos atores do grupo, que todos os demais atores desse grupo também receberão uma solicitação de migração.

No caso do processo de migração conjunta, o sistema envia mensagens para que todos os atores do grupo se preparem para migrar e espera que todos estejam prontos, para só então enviá-los juntos ao nó de destino. Todavia, é possível que algum dos atores esteja efetuando um processamento pesado e demore muito para responder à mensagem de migração. No pior caso, esse ator pode até mesmo estar preso em um *deadlock*. Para evitar que os outros atores esperem indefinidamente até que todo o grupo esteja pronto para migrar, o processo de migração conjunta emprega um tempo máximo de espera (*timeout*). Após o envio da mensagem `MoveGroupTo`, a migração ocorrerá quando todos os atores estiverem prontos para serem migrados ou quando o tempo de espera máxima se esgotar, o que ocorrer primeiro.

Caso o tempo máximo de espera se esgote, a migração ocorrerá mesmo sem que todos os atores estejam prontos. Os atores que porventura ficarem para trás serão migrados assim que terminarem o seu processamento atual e responderem à mensagem de preparação para migração. Em ambas as situações (*timeout* alcançado ou não), todos os atores migrados mantêm o valor de seu atributo `groupId`, de forma que o grupo é mantido exatamente igual no nó de destino.

4.5 Monitoramento de atores

Nossa infraestrutura oferece a possibilidade de cada teatro registrar informações sobre a troca de mensagens realizada pelos atores em execução no teatro. Ainda que os dados coletados não sejam utilizados em nossa implementação atual, eles podem ser importantes para algoritmos de balanceamento ou distribuição de carga que porventura sejam implementados sobre nossa infraestrutura e que usem o mecanismo de migração para buscar continuamente uma melhor distribuição dos atores no aglomerado.

A classe `Profiler` é a responsável por monitorar as trocas de mensagens, e por meio dela os desenvolvedores podem obter informações sobre quais atores mais recebem mensagens de nós remotos. Para cada ator atualmente sendo executado no teatro local, essa classe contabiliza quantas mensagens esse ator recebeu de atores localizados em outros nós do aglomerado. Dessa forma, um ator que esteja recebendo muitas mensagens de um teatro remoto específico pode ser identificado como um bom candidato a ser migrado para aquele nó.

O monitoramento também é feito com respeito às mensagens locais trocadas pelos atores. Assim, o número de mensagens que um determinado ator recebeu de outros atores localizados no mesmo teatro também é contabilizado. Esse número poderia ser comparado aos contadores de mensagens recebidas de teatros remotos, de modo a evitar que um ator que receba muitas mensagens locais seja migrado, ainda que ele também receba muitas mensagens de algum nó remoto. Afinal, nesse caso, a migração provavelmente não surtiria efeito algum, e possivelmente até aumentaria a quantidade de mensagens trafegando pela rede.

Existe um `Profiler` associado a cada teatro, e o usuário pode obter esse `Profiler` a partir do

teatro local, por meio do atributo `LocalTheater.profiler`. De posse desse objeto, é possível ter acesso aos dados capturados pelo monitoramento. Um detalhe importante é que o monitoramento é opcional em cada teatro, de forma a evitar uma possível sobrecarga daquele nó. Por padrão, o monitoramento está desativado, podendo ser ativado no arquivo de configuração, conforme será visto na Seção 4.6.

A classe `IMRecord` (*Incoming Messages Record*) é usada para registrar o número de mensagens que um determinado ator recebeu de um teatro específico. A Listagem 4.11 mostra os principais campos dessa classe. Os atributos `uuid` e `from` identificam, respectivamente, o ator e o teatro em questão. Ou seja, aquele `IMRecord` irá contabilizar quantas mensagens o ator identificado por `uuid` recebeu de algum ator rodando no teatro localizado em `from`. Além disso, o método `count` devolve a contagem atual daquele registro, ou seja, o número de mensagens contabilizadas. O usuário consegue apenas ler essa informação, mas não modificá-la. A tarefa de atualizar esse dado é de responsabilidade exclusiva da infraestrutura.

```
case class IMRecord(uuid: String, from: TheaterNode) {
  private var _count: Int = 0
  def count = _count

  ...

}
```

Listagem 4.11: *Principais campos da classe `IMRecord`.*

A classe `Profiler` implementa uma API com métodos que permitem obter as informações registradas durante o monitoramento da troca de mensagens, como mostra a Listagem 4.12. O método `firstInQueue` devolve o registro que representa o ator que mais recebeu mensagens de algum teatro remoto. O nome do método deve-se ao fato de, internamente, a classe usar uma fila de prioridade ordenada pelo número de mensagens recebidas. Entram na fila apenas registros em que esse número ultrapassa um limiar mínimo. Esse limiar pode ser definido no arquivo de configuração, como explica a Seção 4.6.

```
def firstInQueue: Option[IMRecord]

def localMessagesCount(uuid: String): Int

def incomingMessagesRecords(uuid: String): HashMap[TheaterNode, IMRecord]
```

Listagem 4.12: *API da classe `Profiler`, que permite obter dados sobre a troca de mensagens entre atores.*

O método `localMessagesCount` permite obter o número de mensagens que um determinado ator, especificado pelo parâmetro `uuid`, recebeu de outros atores locais. Como dito anteriormente, essa informação ajuda a decidir se aquele ator é de fato um bom candidato à migração ou se ele recebe mais mensagens locais do que remotas.

Já o método `incomingMessagesRecord` devolve uma informação bem mais detalhada e sem praticamente nenhum tratamento. O valor devolvido, uma tabela associativa, contém todos os registros de número de mensagens recebidas (`IMRecords`) existentes para o ator especificado

pelo parâmetro `uuid`, classificados por teatro. Dessa forma, é possível saber quantas mensagens um determinado ator recebeu de cada um dos nós existentes no aglomerado.

Além desses métodos, a API da classe `Profiler` oferece ainda o método `reset()`. Esse método permite apagar todos os registros sobre mensagens recebidas existentes na classe. Isso pode ser desejável de tempos em tempos, já que informações muito antigas talvez já tenham perdido sua relevância no que diz respeito à tomada de decisões de migração. Atualmente, a única possibilidade é que todos os dados sejam apagados de uma vez, mas uma importante melhoria seria a exclusão desses dados continuamente, apagando sempre as informações mais antigas quando estas atingissem um determinado tempo de existência.

O apagamento dos dados monitorados, além de ser feito pelo método `reset()`, também pode ser realizado automaticamente pela infraestrutura, sempre após um intervalo de tempo pré-determinado. Esse tipo de comportamento pode ser configurado diretamente na classe `Profiler` ou por meio do arquivo de configuração (Seção 4.6).

4.6 Configuração da infraestrutura

Como explicamos na Seção 4.2.1, a execução da infraestrutura depende da existência de um arquivo de configuração. Nesse arquivo deve constar, pelo menos, a lista de todos os nós que compõe o aglomerado onde a aplicação será executada. Caso deseje, no entanto, o usuário pode usar esse arquivo de configuração para especificar diversas outras propriedades que irão alterar o funcionamento da infraestrutura. Toda propriedade possui um valor padrão associado a ela, que será usado caso tal propriedade seja omitida no arquivo de configuração.

As próximas subseções contém a relação completa dessas configurações adicionais disponíveis aos utilizadores de nosso sistema. No arquivo de configuração, as propriedades são agrupadas em blocos, definidos entre chaves (de forma similar a blocos em linguagens de programação). No código, uma propriedade é acessada por seu "nome completo", composto pelo nome de todos os blocos a qual ela pertence, separados por ponto (.), mais o nome da propriedade em si. Dessa forma, na Listagem 4.1, por exemplo, a propriedade `name-server` do primeiro nó seria acessada pelo nome `cluster.node_1.name-server`. Nas próximas seções, usaremos essa notação para nomear as propriedades que descreveremos.

4.6.1 Serviço de nomes

Para funcionar corretamente, nossa infraestrutura depende da existência de um serviço de nomes que associe cada ator à sua localização atual. A implementação padrão desse serviço é distribuída, sendo que cada nó participante armazena uma tabela com o endereço de alguns dos atores atualmente em execução na aplicação. A Seção 6.3 dará mais detalhes sobre o serviço de nomes da infraestrutura. As seguintes propriedades podem ser usadas para configurar o serviço de nomes:

Nome da propriedade: `cluster.NODE_NAME.name-server`
Tipo de valor: Booleano
Valor padrão: Falso

Descrição:	Define se aquele determinado nó (de nome `NODE_NAME`) irá executar uma parte do serviço de nomes distribuído da infraestrutura. Caso tenha valor *verdadeiro*, aquele nó irá armazenar uma tabela associando atores à sua localização atual. No caso da infraestrutura estar usando um serviço de nomes distribuído (como é o caso padrão), ao menos um nó deve ter essa propriedade com valor *verdadeiro*.

Nome da propriedade:	`cluster.name-service.class`
Tipo de valor:	Nome de classe (String)
Valor padrão:	`DistributedNameService`
Descrição:	Define a classe a ser usada como implementação do serviço de nomes. O valor a ser passado deve ser o nome completo da classe (incluindo os nomes dos pacotes aos quais ela pertence). Além disso, a classe especificada deve necessariamente implementar a feição `NameService`.

Nome da propriedade:	`cluster.name-service.hash-function`
Tipo de valor:	Nome de classe (String)
Valor padrão:	`DefaultHashFunction`
Descrição:	Caso o usuário decida utilizar a implementação padrão de serviço de nomes distribuído, ele ainda pode configurar uma característica particular desse serviço. Cada par (ator, localização) deve ser guardado num determinado nó participante do serviço de nomes. O nó onde essa informação será guardada é determinado usando-se uma função de espalhamento (*hash*). Caso o usuário deseje, ele pode implementar uma função de espalhamento diferente da padrão e especificá-la com essa propriedade, cujo valor deve ser o nome completa de uma classe que implemente a feição `HashFunction`.

4.6.2 Protocolo inter-teatros

O protocolo inter-teatros define como será implementada de fato a troca de mensagens entre os teatros. Uma possibilidade é usar algum arcabouço de comunicação remota, por exemplo. Na Seção 6.2.3 descreveremos em detalhes o protocolo inter-teatros. As seguintes propriedades podem ser usadas para a configuração desse protocolo:

Nome da propriedade:	`cluster.theater-protocol.class`
Tipo de valor:	Nome de classe (String)
Valor padrão:	`NettyTheaterProtocol`

Descrição:	Define qual a implementação de protocolo inter-teatros deverá ser usada na infraestrutura. A classe deve ser especificada por seu nome completo, e deve necessariamente implementar a classe abstrata `TheaterProtocol`.

Nome da propriedade:	`cluster.theater-protocol.port`
Tipo de valor:	Inteiro
Valor padrão:	1985
Descrição:	Define a porta padrão a ser usada pelo protocolo inter-teatros. Vale notar que essa propriedade não precisa necessariamente ser usada pelo protocolo. O protocolo padrão, no entanto, utiliza essa porta para realizar a comunicação remota.

4.6.3 Monitoramento

O monitoramento de troca de mensagens pode ser configurado com as seguintes propriedades:

Nome da propriedade:	`cluster.NODE_NAME.profiling`
Tipo de valor:	Booleano
Valor padrão:	Falso
Descrição:	Define se aquele determinado nó (de nome `NODE_NAME`) irá executar o serviço de monitoramento de troca de mensagens. Essencialmente, esse serviço contabiliza todas as mensagens que um ator recebeu de um determinado nó (incluindo o nó local). Dessa forma, é possível detectar casos em que um ator esteja recebendo muitas mensagens de um nó remoto. Esse tipo de informação pode ser útil a algoritmos de balanceamento de carga. Por padrão, esse monitoramento está desligado, de modo a evitar uma sobrecarga no processamento daquele nó.

Nome da propriedade:	`cluster.profiling.queue-threshold`
Tipo de valor:	Inteiro
Valor padrão:	100
Descrição:	Define o valor mínimo que um determinado registro de mensagens recebidas deve contabilizar para que entre na fila de prioridade existente na classe `Profiler`. Nessa fila estão os registros de maior incidência de mensagens, ordenados pelo número de mensagens recebidas.

Nome da propriedade:	`cluster.profiling.reset-mode`
Tipo de valor:	`[MANUAL | AUTOMATIC]`

Valor padrão:	`MANUAL`
Descrição:	Define se o apagamento completo dos dados monitorados será feito manualmente pelo usuário (chamando o método `reset()`) ou automaticamente pela infraestrutura. Essa reinicialização dos dados permite eliminar informações muito antigas, que possivelmente já perderam sua relevância (em particular para fins de balanceamento de carga). Essa propriedade pode ser alterada em tempo de execução, diretamente na classe `Profiler`.

Nome da propriedade:	`cluster.profiling.reset-interval`
Tipo de valor:	Inteiro
Valor padrão:	60
Descrição:	Define o intervalo, em minutos, para que a infraestrutura realize o apagamento dos dados monitorados sobre troca de mensagens. Essa propriedade só faz sentido quando usada juntamente com o valor `AUTOMATIC` para a propriedade `cluster.profiling.reset-mode`. Nesse caso, a cada intervalo do tempo definido, a própria infraestrutura se encarregará de chamar o método `reset()`, apagando todos os dados monitorados. Essa propriedade pode ser alterada em tempo de execução, diretamente na classe `Profiler`.

4.6.4 Configurações adicionais

Nome da propriedade:	`cluster.distribution-algorithm`
Tipo de valor:	Nome de classe (String)
Valor padrão:	`RoundRobinAlgorithm`
Descrição:	Define qual algoritmo deve ser usado pelo método `launch` para a escolha do nó onde o ator sendo instanciado será colocado. A classe deve ser especificada por seu nome completo, e deve necessariamente implementar a feição `DistributionAlgorithm`.

Nome da propriedade:	`cluster.colocated-actors.migration-timeout`
Tipo de valor:	Número do tipo Long
Valor padrão:	5000

Descrição: Define o tempo máximo, em milissegundos, que um determinado grupo de atores irá esperar, durante o processo de migração, até que todos os atores do grupo estejam prontos para migrar. Esse tempo máximo é estabelecido de forma a não atrasar indefinidamente a migração, levando em conta que algum dos atores do grupo pode inclusive estar preso em algo como um *deadlock* ou um laço infinito. Caso esse tempo seja atingido e nem todos os atores estejam prontos para migrar, a migração é efetuada com o grupo incompleto, e os atores remanescentes no teatro de origem serão migrados para junto do grupo tão logo respondam à mensagem de preparação para migração.

Capítulo 5

Akka

O sistema Akka[1] é uma plataforma voltada para aplicações concorrentes, escaláveis e tolerantes a falhas. Fortemente baseado no conceito de atores, esse sistema inspirou-se no Erlang OTP, uma plataforma para aplicações distribuídas que utilizam atores e comunicação assíncrona e são escritas em Erlang [6].

O código do Akka é praticamente todo escrito em Scala, mas para algumas tarefas utiliza arcabouços já existentes (tipicamente desenvolvidos em Java), como por exemplo no suporte à comunicação remota. O sistema oferece APIs em Scala e Java, de modo a permitir interoperabilidade com sistemas já existentes, em que a linguagem Java é predominante.

Este capítulo descreve apenas os atores e os servidores remotos do Akka, pois são esses os elementos usados diretamente em nossa implementação. No entanto, vale mencionar que o Akka contempla uma extensa gama de funcionalidades, como memória transacional em software (*software transactional memory*, ou STM), suporte a bancos de dados não-relacionais (NoSQL), suporte ao protocolo AMQP e integração com diferentes arcabouços, como Camel[2] e Spring[3].

Atualmente, o Akka encontra-se na versão 1.2, lançada em setembro de 2011. No entanto, nossa implementação baseou-se na versão 0.10, lançada em agosto de 2010. Em particular antes do lançamento da versão 1.0 (ocorrido em fevereiro de 2011), a implementação do Akka sofria constantes mudanças por parte de seus desenvolvedores, muitas vezes tornando uma versão mais nova incompatível com código escrito para a versão anterior. Essa mudanças motivaram nossa decisão de fixar a versão do Akka que serviria de base para nossa infraestrutura. Pudemos assim concentrar os esforços no desenvolvimento de nosso trabalho, sem ter que lidar com questões de integração com as novas versões do Akka.

Portanto, tudo que será descrito neste trabalho, em particular nas próximas seções, diz respeito à versão 0.10 do Akka. Muita coisa mudou nas versões mais recentes, e certamente melhorias foram feitas. Um trabalho futuro interessante seria portar nossa implementação para a versão mais atual do Akka.

[1] http://akka.io
[2] http://camel.apache.org/
[3] http://www.springsource.org/

5.1 Servidores remotos

As aplicações distribuídas construídas sobre a plataforma Akka usarão essencialmente dois componentes: atores e servidores remotos. Os servidores remotos funcionam como contêineres que abrigam atores que podem ser acessados remotamente. Cada servidor remoto possui uma tabela com as referências para os atores nele registrados. Na próxima seção, daremos mais detalhes sobre como instanciar e registrar atores num servidor remoto.

Os servidores remotos são implementados pela classe `RemoteServer`. Um servidor remoto deve ser iniciado com um nome de máquina (*hostname*) e uma porta, que representam o endereço ao qual o servidor estará associado. A partir daí, torna-se possível a comunicação remota com aquele servidor por meio daquele endereço. A Listagem 5.1 mostra um exemplo de iniciação de um servidor remoto, no nó `example.host.com` e na porta 9999.

```
val server = new RemoteServer
server.start("example.host.com", 9999)
```

Listagem 5.1: *Criando um servidor remoto e associando-o a um endereço local.*

Vale ressaltar que o nome da máquina passado deve corresponder ao do computador onde foi feita a chamada, ou seja, deve ser local. Além disso, apenas um servidor remoto pode estar associado a um determinado endereço. Ou seja, pode existir mais de um servidor remoto em execução num mesmo nó, mas eles deverão ser iniciados em portas diferentes.

Como veremos mais adiante, os teatros existentes em nossa infraestrutura são implementados sobre os servidores remotos do Akka. Internamente cada teatro possui uma instância de `RemoteServer`, que é quem de fato realiza a comunicação remota entre os atores. Dada a importância dos servidores remotos para nossa implementação, nesta seção estudaremos um pouco mais a fundo o funcionamento de tais servidores.

5.1.1 A comunicação usando o Netty

Dentro dos servidores remotos do Akka, a camada mais baixa de comunicação remota usa o arcabouço Netty[4]. O projeto Netty oferece toda uma gama de recursos que dão apoio à construção de aplicações distribuídas que se comunicam assincronamente, de maneira eficiente e escalável.

Ainda que no Netty exista o suporte para o uso de diferentes protocolos, toda a comunicação remota no Akka é feita por meio do protocolo TCP/IP. Ao se iniciar um servidor remoto num endereço específico, uma instância de um servidor Netty é vinculada àquele endereço. É essa instância que se encarrega de criar e administrar os canais de comunicação que farão o envio e o recebimento de mensagens entre diferentes computadores.

Interceptação de mensagens no `ChannelPipeline`

Quando um cliente remoto abre uma conexão com um servidor do Netty, é criado um canal de comunicação. Esse canal entrega as mensagens ao servidor, e também pode ser usado para que o servidor responda ao cliente.

[4] http://www.jboss.org/netty

Ao chegar num servidor remoto, uma mensagem passa por uma série de manipuladores de canal (*channel handlers*), que constituem o chamado *channel pipeline*. Este nada mais é que uma cadeia de interceptadores. Cada interceptador dessa cadeia efetua algum tratamento sobre a mensagem e a encaminha ao próximo interceptador. Um interceptador pode também escolher não passar a mensagem para frente, encerrando ali o caminho dela dentro do *pipeline*.

Uma instância da classe `ChannelPipeline` define, dentro de um servidor, o conjunto de interceptadores pelos quais as mensagens que chegam devem passar. Esses manipuladores podem interceptar tanto as mensagens que chegam em um canal como as que saem dele. Assim, um protocolo que deseje criptografar mensagens, por exemplo, deve possuir dois manipuladores: um para codificar as mensagens que saem do canal, e outro para decodificar aquelas que chegam.

Os servidores remotos do Akka definem uma série de manipuladores para os canais de comunicação que serão usados na troca de mensagens. A instância de `ChannelPipeline` usada possui, em sua maioria, manipuladores implementados pelo próprio Netty, que já vêm prontos para uso. Eles permitem, por exemplo, comprimir mensagens ou criptografá-las com o protocolo SSL, caso assim se deseje.

Além desses, um manipulador específico do Akka também é adicionado ao *channel pipeline*. Esse interceptador, denominado `RemoteServerHandler`, mantém uma tabela de atores e é responsável pela lógica do contêiner de atores propriamente dito. Ao interceptar uma mensagem, o `RemoteServerHandler` verifica para qual ator ela se destina, busca esse ator em sua tabela e, encontrando o destinatário, encaminha a mensagem a ele.

O entendimento desse esquema de interceptadores será importante na Seção 6.2, que descreve a implementação dos teatros em nossa infraestrutura. Tivemos de criar um novo manipulador de canal, específico para os teatros, para que as mensagens a atores móveis fossem identificadas e encaminhadas corretamente.

5.1.2 Formato das mensagens

No Akka, todas as mensagens remotas trocadas entre diferentes nós são codificadas com o auxílio do Protobuf[5], uma caixa de ferramentas para a seriação de dados estruturados desenvolvida pelo Google e empregada na implementação da maioria dos protocolos e formatos de arquivos adotados internamente por essa empresa. O Protobuf oferece uma série de ferramentas que permitem ao desenvolvedor de aplicações fazer a seriação de seus dados de maneira eficiente, extensível e independente de linguagem de programação.

O Protobuf possui uma linguagem própria que permite definir os tipos de objetos que serão seriados. O Akka define, num arquivo `.proto` escrito nessa linguagem, todos os formatos das mensagens que trafegam entre os nós. Esse arquivo é posteriormente compilado com uma ferramenta do Protobuf, que gera código executável capaz de efetuar a seriação dos objetos na linguagem de programação escolhida (no caso do Akka, essa linguagem é Java, pois até o momento o Protobuf não tem suporte para geração de código Scala).

Assim, para as mensagens enviadas a atores remotos, por exemplo, esse arquivo `.proto` define uma mensagem chamada `RemoteRequestProtocol`. Além do corpo da mensagem propriamente dito, essa mensagem contém outras informações, como o `UUID` do ator destinatário. A presença de

[5] http://code.google.com/p/protobuf

tal identificador permite que o servidor remoto busque o ator para o qual a mensagem deve ser encaminhada.

Vimos na seção anterior que cada servidor remoto possui um conjunto de interceptadores (*channel pipeline*) para processar as mensagens que chegam e saem dele. Entre os interceptadores já implementados pelo Netty, e usados no Akka, estão dois que realizam, automaticamente, a codificação e decodificação de mensagens no formato Protobuf. Isso torna bastante vantajoso o uso do Netty em conjunto com o Protobuf.

5.2 Atores

Mesmo sendo escrito em Scala, o Akka não usa a implementação de atores incluída na biblioteca padrão dessa linguagem. Os criadores do Akka optaram por desenvolver sua própria implementação de atores. Possivelmente essa decisão levou em conta certos debates ocorridos na lista de discussão *Scala Internals*, bem como as informações, veiculadas nessa mesma lista, sobre problemas na implementação de atores da biblioteca de Scala. Esses problemas poderiam levar a vazamentos de memória e/ou a *deadlocks* [31, 32].

Aproveitando-se do fato de Scala ser uma linguagem "escalável", na qual usuários são capazes de criar componentes que se pareçam com recursos nativos, os desenvolvedores do Akka criaram mais uma biblioteca que permite a utilização de atores com funcionalidade e aparência sintática bastante similares às dos *processos* de Erlang.

5.2.1 Criação de atores

De maneira geral, a utilização dos atores do Akka é muito semelhante à dos atores da biblioteca de Scala. As mensagens também são enviadas com `!` e a descrição das mensagens aguardadas por um ator também é feita com uma sequência de cláusulas `case`.

Toda implementação de um ator do Akka deve:

- Estender a feição `Actor`, da infraestrutura Akka, ou alguma classe que por sua vez estenda `Actor`;
- Definir o método `receive`, que deve ser composto por uma sequência de cláusulas `case`, cada uma especificando um tipo de mensagem que esse ator deseja tratar e a ação correspondente ao tratamento daquela mensagem.

A Listagem 5.2 mostra a definição de um ator do Akka que funciona como um dicionário simples, permitindo a inserção, consulta e remoção de valores associados a chaves.

Uma diferença importante entre os atores do Akka e os de Scala é que, no caso dos primeiros, cada ator implementa o método `receive`, responsável pelo tratamento das mensagens destinadas ao ator, mas nunca faz chamadas a esse método. As chamadas a `receive` ocorrem sempre por iniciativa do próprio Akka. Por meio de uma chamada a `receive`, o ambiente de execução do Akka notifica um ator da existência de uma próxima mensagem que tem esse ator como destinatário e deve ser consumida por ele. No caso dos atores de Scala, o método `receive` é implementado pela biblioteca de atores e é chamado por um ator para aguardar a próxima mensagem e especificar o tratamento dessa mensagem.

```
case class Put(key: String, value: Any)
case class Get(key: String)
case class Remove(key: String)

class DictionaryActor extends Actor {
  private val dict = new HashMap[String, Any]

  def receive = {
    case Put(key, value) => dict.put(key, value)
    case Get(key) => self.reply(dict.get(key))
    case Remove(key) => dict.remove(key)
    case _ => println("Mensagem desconhecida recebida. Descartando...")
  }
}
```

Listagem 5.2: *Definição de um ator do Akka.*

A abordagem do Akka apresenta vantagens e desvantagens. Por um lado, ela desobriga o desenvolvedor de cuidar do ciclo de vida de um ator, ou seja, de manter o ator num laço enquanto ele estiver em execução. A infraestrutura Akka cuida de manter um ator sempre ativo, até que alguém (possivelmente o próprio ator) invoque seu método `exit()`.

Além disso, a implementação do Akka está mais de acordo com o modelo de atores originalmente proposto, no qual o comportamento de um ator é definido exclusivamente pelo modo como ele trata as mensagens enviadas a ele. No caso dos atores de Scala, o programador deve implementar o método `act()` e chamar explicitamente os métodos de tratamento de mensagens (`receive` ou `react`). Isso permite que, no corpo do método `act()`, sejam executadas ações que não estejam associadas ao processamento de alguma mensagem (como é o caso nas linhas 19 e 20 da Listagem 3.2).

Por outro lado, uma desvantagem no caso dos atores do Akka é a impossibilidade do uso de `receives` aninhados, já que as chamadas a esse método não estão sob o controle do criador do ator. Para contornar essa limitação, em alguns casos pode ser necessário modelar o ator de uma maneira menos intuitiva.

Uma outra característica dos atores do Akka é que, caso uma mensagem enviada a um ator não case com nenhum dos padrões especificados no método `receive`, uma exceção será lançada. Esse comportamento é diferente do apresentado pelos atores de Erlang e de Scala, em que mensagens inesperadas são mantidas na caixa de mensagens para que, no futuro, elas possam ser aceitas e tratadas por algum novo comportamento do ator.

5.2.2 Instanciação e uso de referências

O modelo de instanciação e manipulação de atores existente em nossa infraestrutura, visto na Seção 4.3.2, é amplamente baseado no dos atores do Akka. Assim, os atores do Akka também não podem ser instanciados diretamente por meio da palavra chave **new**. Para instanciar um ator, deve-se usar o método `actorOf`, análogo ao método `launch` de nossa infraestrutura.

Também o método `actorOf` pode ser parametrizado com a classe do ator a ser instanciado ou receber um trecho de código que instancie o ator. A Listagem 5.3 mostra esses dois modos de uso. Como vimos numa chamada análoga ao método `launch`, o uso de **new** na segunda chamada a `actorOf` só é possível porque tal método recebe um parâmetro por nome (Seção 2.3.2).

```
val ref1 = Actor.actorOf[DictionaryActor]
val ref2 = Actor.actorOf(new DictionaryActor())

ref1.start()
ref1 ! Put("ref2", ref2)
```

Listagem 5.3: *Instanciando atores com o método* `actorOf`.

A manipulação de um ator ocorre exclusivamente por meio de sua referência. As referências `ref1` e `ref2`, devolvidas pelas chamadas a `actorOf` na Listagem 5.3, implementam a feição `ActorRef`. Essa feição oferece os métodos que fazem o envio de uma mensagem (dos quais `!` é o principal), além de uma variedade de outros métodos, como por exemplo o que inicia a execução do ator.

Com essa arquitetura, o Akka impõe total separação entre a implementação de um ator e as referências para esse ator, de forma semelhante aos PIDs de Erlang (Seção 3.2.1). Isso garante, em particular, que o estado de um ator fique encapsulado e não se tenha acesso direto a ele. A única maneira de manipular o estado de um ator é por meio do envio de mensagens ao ator, como exige o modelo de atores.

Além disso, o sistema de processamento de mensagens garante que duas mensagens enviadas a um mesmo ator nunca serão processadas concorrentemente. Em outras palavras, não precisamos nos preocupar com concorrência dentro do método `receive` do ator. No código da Listagem 5.2, por exemplo, o dicionário implementado pelo ator, ainda que muito simples, poderia ser usado num cenário de concorrência sem a necessidade de quaisquer mecanismos adicionais de sincronização.

Nos atores do Akka, todos os atributos essenciais de um ator (os atributos presentes em todos os atores, como por exemplo o `UUID` do ator) estão na feição `ActorRef`, de modo a ficarem acessíveis aos usuários. Por outro lado, muitas vezes pode ser necessário ter acesso a esses atributos de dentro da implementação do ator. Isso pode ser feito por meio da variável `self`, presente na feição `Actor`.

Além de atributos, a feição `ActorRef` também disponibiliza métodos que podem ser usados pela implementação do ator. Um desses é o método `reply()`, que envia uma resposta ao ator remetente da mensagem que está sendo processada. A Listagem 5.2 mostra um exemplo do uso de `self.reply()` durante o processamento de uma mensagem.

Referências locais e remotas

Na realidade, a feição `ActorRef` serve como ancestral comum para as duas classes que implementam de fato as referências para atores existentes no Akka:

`LocalActorRef` Implementa as referências locais, usadas para referenciar atores que estão sendo executados na mesma máquina virtual que a referência. Cada referência local encapsula um atributo do tipo `Actor`, que contém a referência (no sentido de um apontador para um objeto) para a implementação do ator.

`RemoteActorRef` Implementa as referências remotas, usadas para referenciar atores que estão sendo executados numa máquina virtual diferente daquela onde se encontra a referência. No caso distribuído, um ator e uma referência para esse ator estarão em computadores diferentes, interligados por uma rede. Cada uma dessas referências funciona como um representante

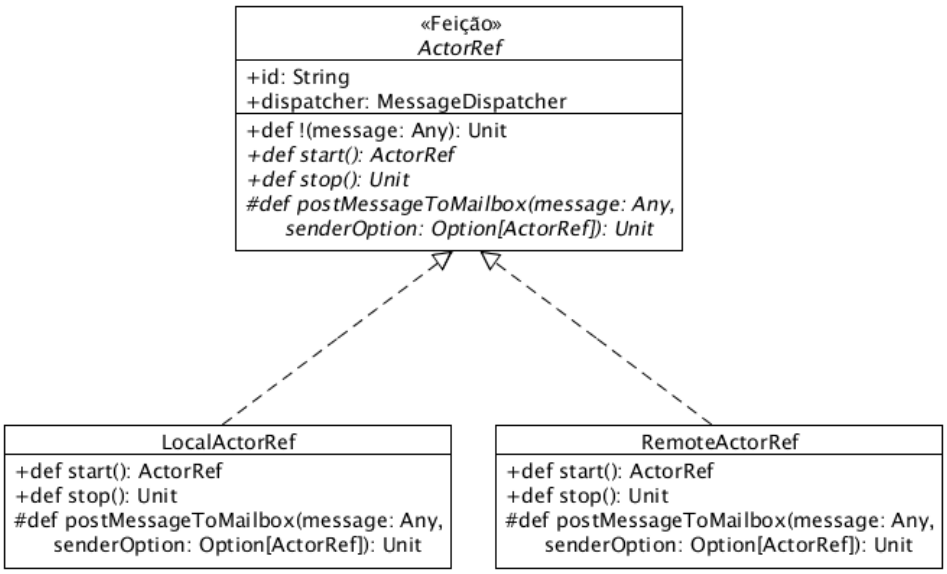


Figura 5.1: *Diagrama com as classes que implementam as referências a atores no Akka.*

(*proxy*) do ator propriamente dito. Internamente, uma referência remota encapsula informações como o identificador (`UUID`) do ator e a localização (nome da máquina e porta) do servidor remoto que o hospeda.

As chamadas a `actorOf()` vistas na Listagem 5.3 devolvem instâncias de `LocalActorRef`, ou seja, referências locais. Na Seção 5.2.5 veremos como instanciar atores em outros computadores, situação na qual é devolvida uma referência remota, instância de `RemoteActorRef`.

A Figura 5.1 mostra a relação entre a feição `ActorRef` e suas duas subclasses, `LocalActorRef` e `RemoteActorRef`. Os métodos com o nome em itálico em `ActorRef` representam métodos abstratos, ou seja, que precisam ser implementados nas subclasses.

Em particular, o método `!`, usado para fazer o envio de mensagens, é implementado concretamente na feição `ActorRef`. Porém, sua implementação depende do método abstrato `postMessageToMailbox()`. É nesse método que a mensagem será de fato processada. As implementações concretas de `postMessageToMailbox()` nas duas subclasses de `ActorRef` têm comportamentos totalmente distintos: enquanto as referências locais colocam a mensagem na caixa de mensagens do ator local, as referências remotas apenas repassam a mensagem para o nó remoto onde o ator está hospedado.

5.2.3 Processamento de mensagens (*threads* vs. eventos)

Como vimos na Seção 3.2.2, os atores de Scala possuem dois métodos para efetuar o recebimento e processamento de mensagens: `receive` e `react`. Enquanto o primeiro faz com que o ator fique associado a uma *thread*, o segundo permite que uma mesma *thread* (ou, em geral, um conjunto de *threads*) execute o código de diferentes atores. Esta última abordagem tem a vantagem de não limitar o número de atores pelo número de *threads* que podem ser criadas numa máquina virtual Java.

No Akka também há suporte para atores baseados em *threads* e para atores dirigidos por eventos, mas a implementação é diferente. Cada ator possui associado a si um *despachador de mensagens*

(*message dispatcher*), encarregado de processar as mensagens enviadas àquele ator. Um despachador é atribuído a um ator por meio do campo `dispatcher`, presente na feição `ActorRef`.

O código da Listagem 5.4 mostra como alterar o despachador de mensagens de um ator. Caso nada seja feito, o despachador padrão associado a um ator é uma instância da classe `ExecutorBasedEventDrivenDispatcher`, um processador de mensagens baseado em eventos. Isso fará com que o ator funcione de maneira semelhante a um ator de Scala que use o método `react`.

```
// Atores baseados em threads (receive)
class ThreadActor extends Actor {
  self.dispatcher = Dispatchers.newThreadBasedDispatcher(self)
  (...)
}

// Atores baseados em eventos (react)
class EventActor extends Actor {
  self.dispatcher = newExecutorBasedEventDrivenDispatcher(self)
  (...)
}
```

Listagem 5.4: *Definição do modo de processamento de mensagens em atores do Akka.*

Um detalhe importante é que atribuições ao campo `dispatcher` só podem ser feitas antes que o ator tenha sido iniciado. Com essa restrição, a abordagem do Akka não permite que, ao longo da execução de um ator, alternem-se períodos durante os quais o ator emprega um despachador baseado em *threads* e períodos durante os quais ele emprega um despachador baseado em eventos (tal alternância pode ocorrer no caso de um ator de Scala que utilize tanto `receive` quanto `react`).

Por outro lado, no Akka é possível definir, em tempo de execução, o tipo de despachador associado a um ator, desde que se faça isso antes que o ator tenha sido iniciado: basta atribuir um valor ao atributo `dispatcher` da referência local associada ao ator. Além disso, o despachador dirigido por eventos e o baseado em *threads* não são as únicas opções disponíveis no Akka. Existem ainda outros despachadores, que implementam diferentes maneiras de processar mensagens. Um desenvolvedor pode, inclusive, implementar seu próprio despachador, adequado às suas necessidades (isso de fato ocorreu em nosso trabalho).

Funcionamento do despachador de mensagens baseado em eventos

O despachador de mensagens padrão associado a atores no Akka, do tipo `ExecutorBasedEventDrivenDispatcher`, é o mais indicado para a maioria dos casos. Com ele, os atores respondem a eventos (o envio de mensagens) e o processamento das mensagens é feito por um conjunto (*pool*) de *threads*, permitindo a criação de um número bastante grande de atores (esse número é limitado essencialmente pela quantidade de memória disponível à aplicação).

Nesta seção daremos detalhes sobre como esse despachador processa as mensagens enviadas a um ator. Isso será importante para a Seção 6.1.3, onde descrevemos o despachador de mensagens que implementamos especificamente para atores móveis. Nosso despachador, também baseado em eventos, funciona de maneira muito semelhante ao original do Akka, porém com mudanças importantes que permitem o correto funcionamento do mecanismo de migração.

A Figura 5.2 exibe um diagrama com toda a sequência de passos que a infraestrutura executa

desde o envio de uma mensagem até o efetivo processamento da mensagem. Observe que existem barras mais escuras e mais claras. As barras escuras representam ações executadas pela *thread* do ator que enviou a mensagem. Como o envio de mensagens é assíncrono, essa ações devem ser executadas com a maior brevidade possível, para que o controle volte de imediato ao remetente da mensagem.

Já as barras mais claras representam ações executadas pela *thread* que vai de fato processar a mensagem enviada ao ator. No caso do despachador de mensagens que estamos estudando (`ExecutorBasedEventDrivenDispatcher`), essa *thread* faz parte de um conjunto de *threads* que se revezam no processamento de mensagens de diferentes atores.

O diagrama na Figura 5.2 ilustra a seguinte sequência de ações:

1. Um ator remetente (`sender`) envia a mensagem `Message` a um ator destinatário (`dest`), por meio do comando `dest ! Message`.

2. O ator `dest` cria uma instância da classe `MessageInvocation` com informações sobre a mensagem recebida, como o remetente e o conteúdo da mensagem. É essa instância que será colocada na caixa de mensagens do ator. A Listagem 5.5 mostra uma versão simplificada da classe `MessageInvocation` (alguns trechos foram omitidos por não serem essenciais para o entendimento do despachador baseado em eventos).

3. O ator `dest` chama o método `send()` na nova instância de `MessageInvocation`. Isso faz com que essa instância obtenha (por meio do atributo `dispatcher`) o despachador de mensagens associado àquele ator. Sobre esse despachador, que é uma instância de `MessageDispatcher` (de fato ele é um `ExecutorBasedEventDrivenDispatcher`), a instância de `MessageInvocation` chama o método `dispatch()`, passando como parâmetro uma referência a si mesma.

4. Dentro do despachador, a primeira providência é adicionar a mensagem à caixa de mensagens do ator `dest` (essa caixa é implementada como uma fila de `MessageInvocation`).

5. O despachador cria uma nova tarefa para processar as mensagens recebidas pelo ator. Essa tarefa é um objeto do tipo `Runnable`.

6. O despachador entrega a tarefa recém criada a um `ExecutorService`. Dessa forma ele solicita que aquela tarefa seja executada por alguma *thread* de um *pool* de *threads*.

7. O controle volta para o ator `sender`, o qual executará o código que sucede o comando `'dest ! Message'`. Como o envio de mensagens é assíncrono, o ator remetente não sabe quando a sua mensagem será processada pelo ator destinatário.

8. Em algum momento futuro, o `ExecutorService` iniciará o processamento da tarefa submetida pelo despachador de mensagens (passo 6). Note que isso ocorrerá em uma *thread* (representada no diagrama pelas barras de cor clara) diferente daquela em que as ações anteriores foram executadas.

9. Essa tarefa irá chamar o método `processMailbox()` do despachador de mensagens. Um detalhe importante é que somente uma *thread* por vez poderá processar as mensagens de um certo ator. Existe uma trava protegendo o método `processMailbox()` e evitando execuções

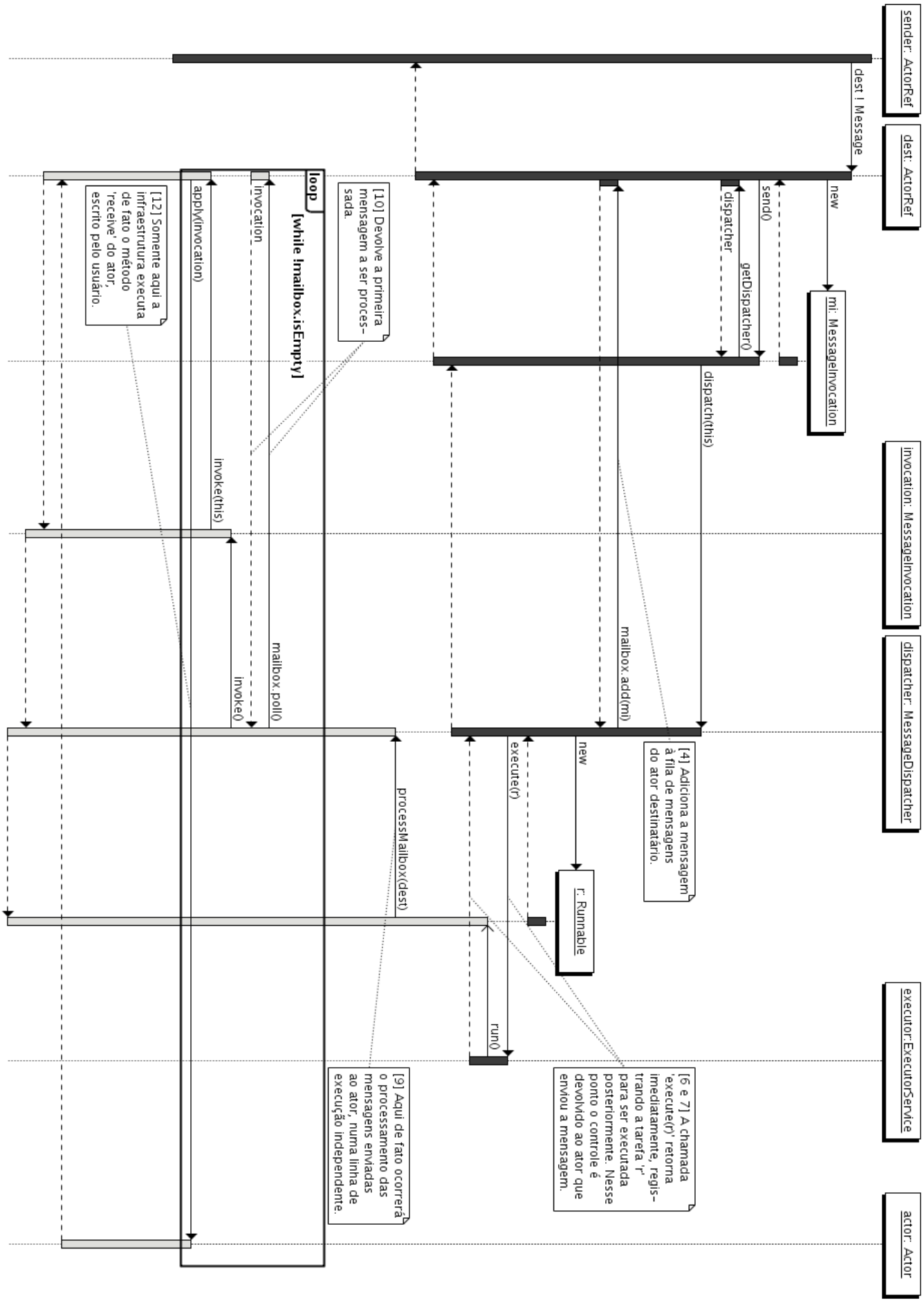


Figura 5.2: *Sequência de ações desencadeadas desde o envio de uma mensagem a um ator até o efetivo processamento da mensagem.*

concorrentes desse método que processem a caixa de mensagens de um mesmo ator. Caso a trava já esteja tomada, a tarefa simplesmente encerra sua execução sem fazer mais nada.

10. Dentro do método `processMailbox()`, existe um laço que retira o primeiro elemento da caixa de mensagens do ator e executa o método `invoke()` nesse elemento (que é do tipo `MessageInvocation`). Esse laço é executado até que a caixa de mensagens esteja vazia.

11. A chamada a `invoke()` na mensagem gera uma chamada ao método `invoke()` em `dest`, a referência local para o ator destinatário.

12. Somente nesse momento, a referência local, que é o único elemento a ter acesso à implementação do ator, chama nessa implementação o método `apply()`, passando como parâmetro a mensagem recebida. É nessa ocasião que será executado o código do método `receive` implementado pelo ator.

No comentários do diagrama na Figura 5.2, os números que aparecem entre colchetes indicam a correspondência com os itens da lista acima.

```
final class MessageInvocation(
   val receiver: ActorRef,
   val message: Any,
   val sender: Option[ActorRef], ...) {

def invoke = try {
  receiver.invoke(this)
} catch {
  (...)
}

def send = receiver.dispatcher.dispatch(this)

(...)
```

Listagem 5.5: *Alguns atributos e métodos da classe* `MessageInvocation`.

É importante enfatizar o interrelacionamento entre os itens 5, 9 e 10 na relação acima. O item 5 diz que, a cada nova mensagem enviada ao ator, uma nova tarefa responsável pelo processamento das mensagens destinadas a esse ator é criada e submetida para execução posterior. Porém, conforme o item 9, se alguma outra *thread* já estiver processando as mensagens desse ator quando a nova tarefa entrar em execução, essa tarefa simplesmente encerrará sua execução, sem processar mensagem alguma. Isso só faz sentido pelo comportamento descrito no item 10: a tarefa que está de fato processando as mensagens de um ator percorre toda a caixa de mensagens do ator. Ou seja, supondo que uma tarefa T_M é criada em consequência do envio de uma mensagem M, podemos afirmar que:

- Caso T_M obtenha a trava do item 9, essa tarefa processará não só a mensagem M, mas quaisquer outras mensagens que já estejam na caixa de mensagens do ator ou que sejam colocadas nessa caixa durante a execução de T_M;

- Caso T_M não obtenha a trava, ela encerrará sua própria execução sem processar mensagem alguma. Por outro lado, quando isso ocorrer a mensagem M já estará na caixa de mensagens

do ator (já que o passo 4 ocorre sempre antes do passo 9). Portanto, uma tarefa T_X, detentora da trava e criada em consequência do envio de alguma outra mensagem X, seguramente passará por M ao percorrer a caixa de mensagens do ator e processará essa mensagem.

5.2.4 Seriação

A plataforma Akka implementa um módulo que permite seriar completamente um ator. A versão seriada de um ator (um vetor de *bytes*) inclui a implementação do ator propriamente dita, sua caixa de mensagens e os atributos essenciais que fazem parte de sua referência local (como o `UUID` do ator). Esses *bytes* podem então ser enviados a um computador remoto ou armazenados em disco, e a partir deles será possível reconstituir posteriormente o ator e sua referência local para que voltem ao exato estado do momento da seriação.

Assim como na definição das mensagens que trafegam entre os nós (Seção 5.1.2), o Akka usa o Protobuf no mecanismo de seriação de atores. Uma mensagem[6] no formato Protobuf foi especificada para representar a versão seriada de um ator. Depois de construir uma mensagem desse tipo contendo a representação de um ator, podemos transformar essa representação num vetor de *bytes* por meio de uma chamada ao método `toByteArray` da mensagem.

A infraestrutura do Akka sabe como seriar as informações contidas na referência local do ator e as mensagens na caixa de mensagens do ator. Já o estado da implementação do ator em si (ou seja, da classe específica do ator definida pelo usuário, que estende a feição `Actor`) pode ser seriado de diferentes maneiras. Exigir que a classe do ator seja seriável pelo mecanismo de Java, por exemplo, seria uma abordagem possível. Todavia, a implementação do Akka dá maior liberdade aos usuários do sistema.

Assim, é o próprio usuário, criador do ator, quem define como tal ator deve ser seriado. Para isso, ele deve implementar uma classe que estenda a feição `Format`, exibida na Listagem 5.6. Como os nomes sugerem, os métodos `toBinary` e `fromBinary` devem implementar os mecanismos pelos quais o ator é transformado num vetor de *bytes* e reconstituído a partir desse mesmo vetor, respectivamente.

```
trait FromBinary[T <: Actor] {
  def fromBinary(bytes: Array[Byte], act: T): T
}

trait ToBinary[T <: Actor] {
  def toBinary(t: T): Array[Byte]
}

trait Format[T <: Actor] extends FromBinary[T] with ToBinary[T]
```

Listagem 5.6: *Definição da feição `Format`, usada na seriação de atores no Akka.*

Apesar de parecer complexa, essa abordagem dá bastante liberdade aos usuários do Akka. Uma solução para casos mais simples, por exemplo, seria implementar uma classe que estenda `Format` e use a seriação Java para seriar o estado dos atores, como no exemplo da Listagem 5.7. A feição

[6] Na terminologia do Protobuf, a palavra *mensagem* não se refere necessariamente a um objeto que trafegará por um canal de comunicação, mas sim a algum tipo de dados estruturado que é definido na linguagem do Protobuf e que pode ser seriado com o auxílio dessa caixa de ferramentas.

`SerializerBasedActorFormat` é definida pelo próprio Akka, de modo a facilitar esse tipo de solução. Bastaria então que os atores fossem feitos seriáveis, por meio do uso da anotação `@serializable` nas classes desses atores.

```
object DefaultActorFormat extends SerializerBasedActorFormat[Actor] {
  val serializer = Serializer.Java
}
```

Listagem 5.7: *Implementação de um `Format` que usa a seriação Java.*

A seriação/desseriação de atores pode ser feita então a partir do objeto *singleton* `ActorSerialization`, do Akka. A Listagem 5.8 exibe os dois métodos principais desse objeto. Como se pode observar, ambos recebem como parâmetro implícito uma instância do tipo `Format`. Ainda que esses parâmetros possam ser passados explicitamente, outro modo de uso seria tornar uma instância de um tipo como o definido na Listagem 5.7 acessível para ser empregada como parâmetro implícito, como visto na Seção 2.2.5.

```
object ActorSerialization {
  def fromBinary[T <: Actor](bytes: Array[Byte])
      (implicit format: Format[T]): ActorRef

  def toBinary[T <: Actor](a: ActorRef, serializeMailBox: Boolean = true)
      (implicit format: Format[T]): Array[Byte]
}
```

Listagem 5.8: *Interface da classe `ActorSerialization`, que permite a seriação e desseriação de atores no Akka.*

5.2.5 Atores distribuídos

Como o foco principal da plataforma Akka é o desenvolvimento e a execução de aplicações distribuídas, o suporte a atores remotos é bastante consistente e recebe atualizações constantes. Assim, esse suporte provavelmente está num estado mais maduro do que aquele oferecido pela biblioteca de atores de Scala.

Para criar aplicações distribuídas usando atores no Akka, o primeiro passo é iniciar um servidor remoto em cada um dos nós onde se deseja que os atores residam. A Listagem 5.1 mostra um exemplo de como iniciar um servidor remoto do Akka. Tendo iniciado um ou mais servidores remotos, a criação de atores remotos pode ser feita de duas formas:

- **Pelo cliente**, que solicita que um ator se torne remoto. Isso não implica em migração do ator, mas sim na instanciação de um outro ator no servidor e no uso do ator local como um representante do ator remoto. Um ator só pode ser feito remoto dessa maneira caso ainda não tenha sido iniciado. Exemplo:

```
class MyRemoteActor extends Actor {
  // Passados 'hostname' e porta
  self.makeRemote("192.168.18.321", 9999)
```

```
  def receive = {
    case msg => println("Mensagem recebida: " + msg)
  }
}

val remote = actorOf[MyRemoteActor]
remote.start
remote ! "Olá ator remoto!"
```

- **Pelo servidor**, que instancia o ator e o registra sob um nome. De posse do endereço do servidor e desse nome, clientes remotos podem obter referências (*proxies*) para enviar mensagens ao ator remoto. Exemplo:

```
class MyRemoteActor extends Actor {
  def receive = {
    case msg => self.reply("ACK")
  }
}

// Na máquina servidora:
val server = new RemoteServer
server.start("192.168.18.321", 9999)
server.register("remote-service", actorOf[MyRemoteActor])

// Na máquina cliente
val remote = RemoteClient.actorFor("remote-service",
  "192.168.18.321", 9999)
remote ! "Olá Servidor!"
```

Um fato importante a se notar é que, para o caso distribuído, a separação entre as implementações dos atores do Akka e as referências para esses atores apresenta uma grande vantagem. As referências contêm informações suficientes para o envio de mensagens aos atores, e elas podem ser seriadas e enviadas para diferentes nós.

Seria de se esperar, no entanto, que todas as referências a atores remotos fossem instâncias da classe `RemoteActorRef`. Entretanto, no caso de atores remotos instanciados pelo cliente, a referência é uma instância de `LocalActorRef`, que, após a chamada a `makeRemote()`, passa a se comportar como uma referência remota.

No Seção 6.1.2, veremos como as referências para atores móveis implementadas em nossa infraestrutura têm uma arquitetura mais consistente, que elimina esse tipo de incoerência. Em nosso caso, uma mesma referência pode, ao longo do tempo, alternar períodos em que ela desempenha o papel de referência local e períodos em que ela tem o papel de referência remota.

A classe `RemoteClient`

No exemplo anterior, que mostrou a criação de atores remotos por parte do servidor, a referência que servirá de representante para o ator remoto foi obtida com o uso do método `actorFor()` do objeto *singleton* `RemoteClient`. Esse método na realidade serve apenas para "fabricar" uma instância de `RemoteActorRef`, que será usada para referenciar o ator remoto.

A classe `RemoteClient`, no entanto, tem um papel importante na comunicação com atores remotos. Como vimos anteriormente, os servidores remotos do Akka usam o arcabouço Netty para implementar a comunicação entre nós remotos. A classe `RemoteClient` também usa o Netty, mas implementa a funcionalidade do lado do cliente que irá se comunicar com os servidores remotos.

Em um dado nó, existirá uma instância de `RemoteClient` para cada servidor remoto com o qual esse nó se comunica. Assim, sempre que uma referência for criada para um dado ator remoto, ela fará o envio de mensagens usando o `RemoteClient` conectado ao servidor que hospeda aquele ator. Caso no momento da criação da referência ainda não exista um `RemoteClient` conectado àquele servidor remoto, uma nova instância de `RemoteClient` será criada e abrirá uma conexão com o servidor usando os métodos apropriados do Netty.

Capítulo 6

Componentes da infraestrutura

A partir deste capítulo, descreveremos os detalhes de implementação de nossa infraestrutura. Neste capítulo em particular, estudaremos as "peças fundamentais" de nosso sistema. Os dois componentes principais da plataforma são, sem dúvida, os atores móveis e os teatros, estudados nas duas próximas seções. Posteriormente, analisaremos ainda três componentes adicionais, também de grande importância: o serviço de nomes e dois gerenciadores de referências.

Neste capítulo, estudaremos essencialmente a arquitetura desses componentes. No Capítulo 7 mostraremos como eles interagem entre si para realizar os serviços oferecidos pela infraestrutura.

6.1 Atores móveis

Os atores móveis de nossa infraestrutura foram implementados usando como base os atores do Akka[1]. A Tabela 6.1 lista as principais funcionalidades implementadas pelos atores móveis e, para cada uma, especifica como ela se relaciona com a implementação original dos atores do Akka.

Funcionalidade	**Já existia no Akka**		**Não existia no Akka**
	Modificações triviais	Modificações importantes	
Migração			✔
Referências		✔	
Seriação	✔		
Processamento de mensagens		✔	
Atores co-locados			✔

Tabela 6.1: *Relação das principais funcionalidades existentes nos atores móveis de nossa infraestrutura.*

Podemos observar que apenas a seriação de atores foi mantida basicamente sem alterações. Para dar suporte ao mecanismo de migração, sofreram importantes modificações as referências para atores e o processamento de mensagens por parte dos despachadores de mensagens. A migração propriamente dita, além dos atores co-locados, são conceitos totalmente inexistentes nos atores originais do Akka.

Nesta seção, descreveremos a arquitetura dos atores móveis, dando destaque para como ela

[1] Vale ressaltar que estamos nos referindo aos atores da versão 0.10 do Akka.

modifica o comportamento original dos atores do Akka. No Capítulo 7, explicaremos como essa arquitetura e a dos teatros funcionam em conjunto para realizar os serviços da infraestrutura, como o de migração.

6.1.1 A feição `MobileActor`

O usuário que deseja escrever sua aplicação usando atores móveis deve implementar seus atores como classes que estendam a feição `MobileActor`. Essa feição estende a feição `Actor`, do Akka, e uma de suas principais funções é servir de ancestral comum para todos os atores móveis da aplicação. Assim, em uma referência para um ator móvel, por exemplo, a implementação do ator é representada por uma instância de `MobileActor`.

Nessa feição aparecem também alguns atributos e métodos específicos do comportamento dos atores móveis. A principal definição dessa feição é a sobrescrita do método `apply()`, original de `Actor`. Como visto na Seção 5.2.3, é nesse método que a mensagem será de fato processada pelo código escrito no método `receive` do ator. A versão sobrescrita do método `apply()` em `MobileActor` permite que certas mensagens específicas, como `MoveTo`, sejam processadas internamente pela infraestrutura, e nunca cheguem ao método `receive`. Esse mecanismo será melhor detalhado na Seção 7.2.1.

Também na feição `MobileActor` aparecem dois métodos que servem como *callbacks* para serem implementados pelo desenvolvedor do ator, caso ele deseje. Os métodos `beforeMigration()` e `afterMigration()` serão chamados pela infraestrutura imediatamente antes e depois da migração, respectivamente. Isso dá, a quem cria um ator móvel, a oportunidade de especificar alguns procedimentos específicos para esses instantes, como a abertura e o fechamento de arquivos ou conexões com um banco de dados, por exemplo.

Por fim, nessa feição também aparece o atributo `groupId`, que identifica o grupo do qual aquele ator faz parte (ou tem valor `None` caso o ator não seja de nenhum grupo). Assim, ainda que `groupId` apareça para o cliente da infraestrutura como um atributo da referência do ator, `MobileActorRef`, tal campo está de fato na implementação do ator. A principal motivação para essa opção de projeto é fazer com que o atributo `groupId` seja automaticamente seriado junto com o ator no momento da migração, já que ele faz parte do estado de todo ator móvel. Caso ele estivesse apenas na referência, mudanças teriam que ser feitas no mecanismo de seriação de atores do Akka para permitir que o atributo `groupId` fosse transportado na migração junto com o ator.

6.1.2 Referências

Como visto na Seção 5.2.2, toda a manipulação de atores do Akka é feita por meio de suas referências, que implementam a feição `ActorRef`. A implementação de fato do ator, com seu estado e o comportamento de como ele tratará as mensagens recebidas, não fica acessível ao usuário, estando encapsulada pela referência.

Essa separação entre referência e implementação é muito importante para o mecanismo de migração de atores. Com ela, é possível dar aos atores *transparência de localização*: o código da aplicação pode manipular os atores normalmente, por meio de suas referências, sem precisar conhecer onde aquele ator está efetivamente hospedado.

Isso permite que alterações na localização de um ator sejam feitas sem grandes dificuldades, apenas cuidando para que a referência àquele determinado ator esteja sempre atualizada. Para

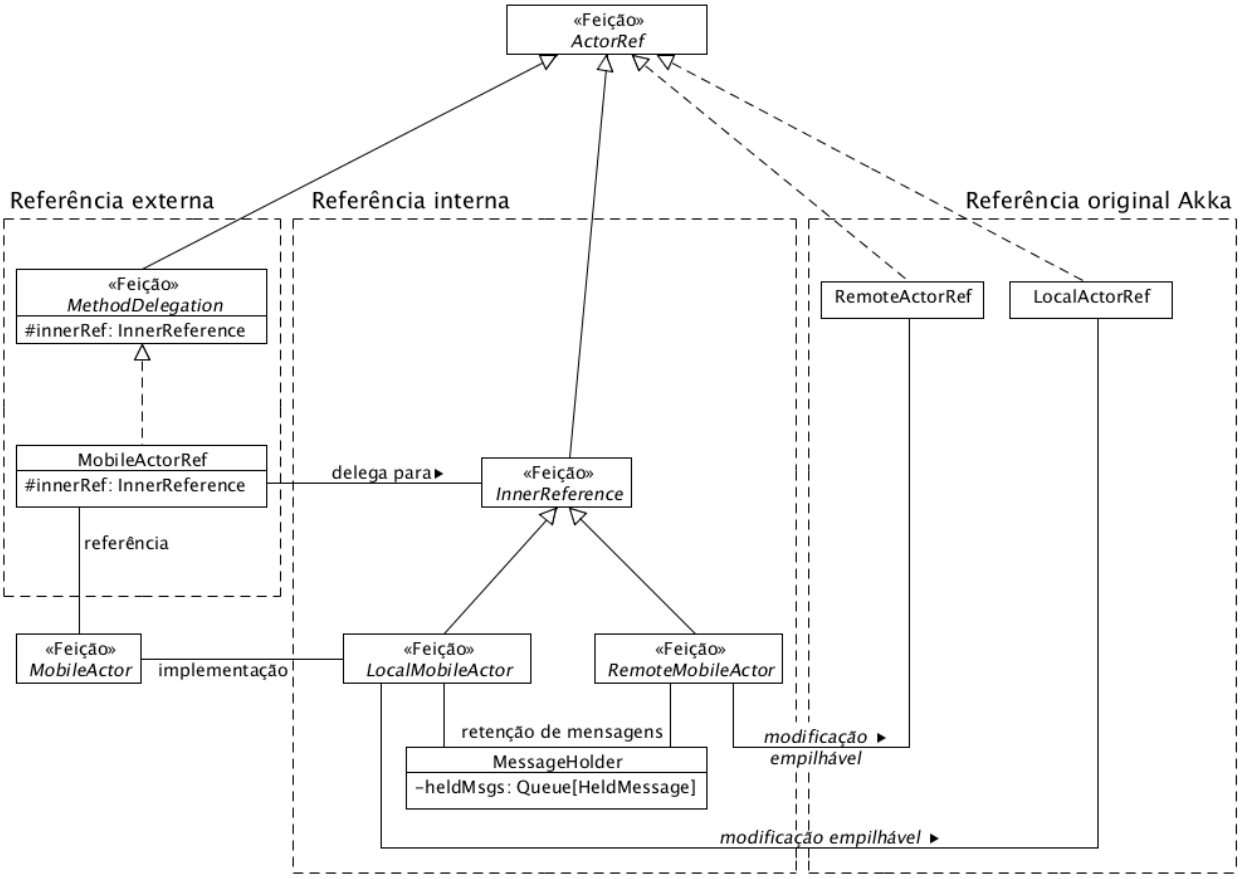


Figura 6.1: *Diagrama com as classes que implementam as referências a atores móveis da infraestrutura.*

os outros atores que se comunicam com o ator recém-migrado, toda a comunicação permanece exatamente igual, e eles sequer precisam ser notificados sobre a migração efetuada. O principal desafio é garantir que não seja perdida nenhuma mensagem enviada ao ator sendo migrado.

Portanto, o que desejamos é uma referência que: *(i)* possa se comportar tanto como local quanto como remota; e *(ii)* possa alternar entre esses dois comportamentos transparentemente. De modo a oferecer essas funcionalidades, as referências a atores móveis existentes em nossa infraestrutura são na realidade compostas por duas camadas: *referência externa* e *referência interna*.

A Figura 6.1 mostra as classes que implementam essas referências em duas camadas. Essencialmente, as referências externas são as devolvidas no momento da instanciação de um ator móvel, e servem para manipulá-lo. Porém, elas apenas delegam suas chamadas para as referências internas, que podem ser remotas ou locais. A seguir, explicaremos em maiores detalhes o funcionamento e a implementação das diferentes classes que compõem as referências a atores móveis.

Referências externas

Uma *referência externa* funciona, essencialmente, como uma *casca* que envolve uma referência interna e, com exceção de alguns métodos específicos, delega suas chamadas para a referência interna. A referência devolvida no momento da instanciação de um ator móvel é sempre uma referência externa. São as referências desse tipo que serão usadas para manipular atores.

As referências externas são instâncias da classe `MobileActorRef`, que implementa a feição `MethodDelegation`. Essa feição, por sua vez, estende a feição `ActorRef`, e portanto oferece

aos clientes a mesma interface das referências originais do Akka. Internamente, a feição `MethodDelegation` possui um atributo `innerRef`, cujo tipo `InnerReference` também estende a feição `ActorRef`. Esse atributo representa uma referência interna, e todas as chamadas de métodos feitas em `MethodDelegation` serão delegadas para essa referência interna. A principal motivação para a criação dessa feição foi permitir que o código da classe `MobileActorRef` ficasse mais limpo, colocando em uma classe separada todo o redirecionamento de métodos.

Com isso, na classe `MobileActorRef` aparecem apenas atributos e métodos específicos dos atores móveis. Dentre esses, o mais importante talvez seja o método `switchActorRef()`, que permite que a referência interna seja trocada. Isso acontece quando um ator local migra para uma máquina remota, ou quando um ator remoto migra para a mesma máquina da referência, passando a ser local. A Listagem 6.1 mostra a implementação desse método, que é bastante simples.

```
protected def switchActorRef(newRef: InnerReference): Unit = {
  val previousInnerRef = innerRef
  innerRef._lock.synchronized {
    innerRef = newRef
    innerRef.outerRef = this
  }
  previousInnerRef.stopLocal()
}
```

Listagem 6.1: *Método `switchActorRef()` da classe `MobileActorRef`.*

Perceba que esse método realiza uma sincronização explícita por meio de uma trava (`_lock`). Essa trava também é usada para fazer a sincronização do método `!`, que envia mensagens a um ator. Isso evita condições de corrida quando uma referência interna está sendo trocada e, concorrentemente, alguém está enviando uma mensagem ao ator referenciado. Além disso, o método `stopLocal()` essencialmente encerra a execução da implementação do ator rodando localmente. No entanto, esse método evita que o ator seja removido do serviço de nomes distribuído (um outro método, `stop()`, encerra o ator e o remove do serviço de nomes, ou seja, encerra *globalmente* a execução do ator).

Além de `switchActorRef()`, a classe `MobileActorRef` também implementa métodos que servem como intermediários entre a implementação do ator e o teatro local. Esses métodos são usados pelo ator quando ele recebe uma mensagem de migração (por exemplo `MoveTo` ou `MoveGroupTo`). O ator chama o método respectivo na referência externa, que por sua vez dá início ao processo de migração chamando os métodos apropriados no teatro local.

A classe `MobileActorRef` contém ainda o atributo *booleano* `isMigrating`, que indica se o ator está passando por um processo de migração. Assim que a migração é iniciada, esse atributo tem seu valor alterado para verdadeiro. Esse valor é consultado por diferentes componentes da implementação de atores móveis que, ao verificarem que uma migração está ocorrendo, mudam seu comportamento apropriadamente.

Nos próximos capítulos, ao descrever o processo de migração, veremos em mais detalhes a implementação da classe `MobileActorRef` e seus métodos mais importantes.

Referências internas

Uma *referência interna* pode ser vista como a "referência de verdade" de um ator, ainda que ela fique escondida e não seja manipulada diretamente pelos usuários da infraestrutura. Todavia, é essa referência que de fato se comunica com a implementação do ator. Assim como no caso das referências originais do Akka, as referências internas podem ser locais ou remotas.

Como mostrado na Figura 6.1, uma referência externa delega seus métodos para uma instância da feição `InnerReference`, que representa a referência interna. Essa feição tem como principal função servir de ancestral comum aos dois tipos de referência interna. Ali também aparecem alguns métodos abstratos, a serem implementados por suas subclasses, como `isLocal` e `node`. Nessa feição é definido também o atributo `outerRef`, do tipo `MobileActorRef`, que deve sempre apontar para a referência externa que "envolve" essa referência.

Já o comportamento das referências internas propriamente ditas é implementado de fato nas feições:

`LocalMobileActor`: Essa feição é usada como uma modificação empilhável sobre a classe `LocalActorRef`, que implementa as referências locais no Akka. Uma de suas principais responsabilidades é garantir que: *(i)* a implementação do ator associado à referência é uma instância de `MobileActor`; e *(ii)* o despachador de mensagens (Seção 5.2.3) usado pelo ator é apropriado para atores móveis. Os despachadores de mensagens para atores móveis serão descritos na próxima seção.

Além disso, a feição `LocalMobileActor` possui métodos para tratar da migração em si, além de outras funcionalidades da infraestrutura, como a implementação de atores co-locados. O funcionamento desses métodos será detalhado no Capítulo 7.

`RemoteMobileActor`: Essa feição é usada como uma modificação empilhável sobre a classe `RemoteActorRef`, que implementa as referências remotas no Akka. Essas referências já apresentam um comportamento muito próximo do desejado para as referências a atores móveis de nossa infraestrutura, já que funcionam como um representante para o ator remoto e abstraem sua localização, que não precisa ser conhecida por quem usa a referência.

As principais modificações efetuadas por essa feição foram no método `postMessageToMailbox()`, que envia uma mensagem ao nó remoto. Antes de enviá-la pela rede, a referência remota "empacota" a mensagem numa instância da classe `MobileActorMessage` (exibida na Listagem 6.2). Isso serve apenas para que a mensagem carregue consigo o endereço do teatro que a remeteu, de modo a possibilitar que a referência remota seja atualizada caso no futuro ocorra uma migração do ator referenciado[2].

Além disso, essa feição efetua ainda algumas modificações na mensagem remota propriamente dita, construída pela própria infraestrutura do Akka com o auxílio do Protobuf. Tais modificações, que serão discutidas mais adiante, permitem que a mensagem seja identificada apropriadamente no teatro de destino.

A feição `InnerReference` possui como atributo uma instância da classe `MessageHolder`, usada pelas referências internas quando for necessário reter, por algum tempo, as mensagens recebidas, antes de encaminhá-las à implementação do ator. A Listagem 6.3 mostra os principais

[2] O mecanismo de atualização de referências será explicado em detalhes na Seção 7.3.1.

```
case class MobileActorMessage(
  senderHostname: String, senderPort: Int, message: Any)
```

Listagem 6.2: *Classe que "empacota" as mensagens enviadas a atores móveis remotos, de maneira que elas levem consigo o endereço do teatro que as remeteu.*

atributos e métodos de `MessageHolder`, bem como a classe definida para representar as mensagens retidas.

Basicamente, a classe `MessageHolder` possui uma fila, para armazenar as mensagens, e métodos, para processá-las de alguma forma. O mecanismo de retenção temporária de mensagens é usado tanto nas referências locais quanto nas remotas. Ao longo do Capítulo 7 serão descritos os momentos em que esse mecanismo é ativado nas referências.

```
case class HeldMessage(message: Any, sender: Option[ActorRef])

class MessageHolder {
  private lazy val heldMessages = new SynchronizedQueue[HeldMessage]

  def holdMessage(message: Any, sender: Option[ActorRef]): Unit
  def processHeldMessages(p: HeldMessage => Unit): Unit
  def forwardHeldMessages(to: ActorRef): Unit
}
```

Listagem 6.3: *Métodos da feição `MessageHolder`, responsável por armazenar localmente mensagens que serão processadas no futuro.*

6.1.3 Despachador de mensagens

Para que o mecanismo de migração de atores funcione corretamente, a infraestrutura deve oferecer as seguintes garantias:

1. Durante o processo de migração de um ator, as mensagens na caixa de mensagens do ator não serão processadas. Essas mensagens serão seriadas junto com o ator e tratadas no nó de destino da migração. Portanto, é necessário que o processamento da caixa de mensagens cesse durante a migração, para assegurar que cada mensagem seja processada apenas uma vez.

2. As mensagens que chegarem após o processo de migração ter sido iniciado serão armazenadas localmente, para serem posteriormente encaminhadas ao ator em seu novo nó, após a migração ter sido concluída.

3. Após o ator receber uma mensagem do tipo `MoveTo`, o processo de migração começará tão cedo quanto possível. Isso significa que mensagens desse tipo devem ter uma prioridade maior do que mensagens normais.

A implementação dos itens 1 e 3 dependeu de modificações no componente responsável pelo processamento das mensagens enviadas a um ator: o despachador de mensagens dos atores do Akka, cujo funcionamento foi discutido na Seção 5.2.3. A seguir, discutiremos como esse funcionamento foi modificado de modo a permitir a mobilidade dos atores.

Suspensão do processamento de mensagens durante a migração

Como regra, após processar uma mensagem do tipo `MoveTo`, um ator não deve processar mais nenhuma mensagem no nó em que ele se encontra. Como a caixa de mensagens do ator será seriada e enviada juntamente com ele ao nó de destino, essa medida evita que uma mesma mensagem seja processada duas vezes, tanto no nó de origem como no de destino.

Para garantir esse comportamento, um novo despachador de mensagens teve de ser criado. A classe `MobileExecutorBasedEventDrivenDispatcher` funciona de modo muito semelhante ao despachador `ExecutorBasedEventDrivenDispatcher`, original do Akka. Porém, ele toma precauções adicionais para impedir que mensagens sejam processadas a partir do momento em que a migração tenha sido iniciada. Esse despachador de mensagens assegura a veracidade das seguintes afirmações:

1. Qualquer tarefa T_M encerrará sua execução imediatamente após processar uma mensagem do tipo `MoveTo`.

2. Qualquer tarefa T_X, ao iniciar sua execução, verificará o estado do ator ao qual está associada a caixa de mensagens que a tarefa deve processar. Se o ator estiver no meio de uma migração, a tarefa encerrará sua própria execução, ainda que tenha conseguido a trava que protege o método `processMailbox()`.

A Listagem 6.4 mostra como essas condições foram asseguradas no método `processMailbox()` da classe `MobileExecutorBasedEventDrivenDispatcher`. Em particular, o item 1 da lista anterior é implementado na linha 9 e o item 2 na linha 4 da referida listagem.

```
1  def processMailbox(receiver: LocalMobileActor): Boolean = {
2    val mailbox = getMailbox(receiver)
3
4    if (receiver.isMigrating) return false
5
6    var messageInvocation = mailbox.poll
7    while (messageInvocation != null) {
8      messageInvocation.invoke
9      if (receiver.isMigrating) return false
10
11     ...
12   }
```

Listagem 6.4: *Suporte (na classe `MobileExecutorBasedEventDrivenDispatcher`) à suspensão do processamento de mensagens durante a migração.*

Alguns detalhes da implementação do método `processMailbox()` foram omitidos em benefício da clareza. Em particular, pode-se supor que, ao retornar o valor `false`, o método sinaliza que o processamento de mensagens foi encerrado e, portanto, que a tarefa que o executa pode ser finalizada.

Como se pode observar na Listagem 6.4, a verificação do estado[3] de um ator é feita por meio do campo `isMigrating`, que tem seu valor alterado para verdadeiro durante o processamento da

[3] Aqui nos referimos tão somente ao estado de um ator no que diz respeito à migração: ou o ator está em migração ou não está.

mensagem `MoveTo`. Como a chamada `invoke` (linha 8 da Listagem 6.4) só retorna após o processamento da mensagem atual ter sido concluído, temos a garantia de que, logo após o tratamento de uma mensagem do tipo `MoveTo`, nenhuma outra mensagem será retirada da caixa de mensagens do ator.

Mensagens com prioridade

Quando alguém solicita a migração de um ator, muito provavelmente o solicitante deseja que tal migração ocorra o mais rapidamente possível. Em um sistema que procura fazer distribuição de carga, por exemplo, pode ser que o computador onde o ator está localizado esteja sobrecarregado, enquanto outras máquinas estão livres. Já em grades oportunistas, muitas vezes os usuários compartilham seus computadores para execução de aplicações da grade durante períodos de ociosidade. Nesse caso, quando um usuário deseja retomar sua máquina, as tarefas da grade devem ser migradas para outros nós com a maior brevidade possível, de modo a evitar uma degradação de desempenho para as aplicações do usuário.

Portanto, o comportamento desejado para os atores móveis é que a mensagem que dá início a uma migração tenha prioridade e seja processada o mais rapidamente possível. No modelo teórico de atores, não existe exigência quanto à ordem em que as mensagens devem ser processadas por um ator. No caso dos atores do Akka, o que ocorre é que as mensagens são colocadas numa fila conforme vão sendo enviadas ao ator. Elas são processadas em ordem de chegada, à medida que forem sendo retiradas da fila.

No caso de nosso despachador de mensagens para atores móveis, precisamos modificar esse comportamento, de modo a adicionar o suporte a mensagens com prioridade. Para isso, a classe `MobileExecutorBasedEventDrivenDispatcher` implementa, além do método `dispatch()`, o método `dispatchWithPriority()`. Essencialmente, a diferença entre os dois é que o novo método insere a mensagem no começo da fila.

No caso do despachador de mensagens padrão do Akka, a caixa de mensagens do ator é implementada como uma instância de `Queue[MessageInvocation]`, ou seja, uma fila do tipo FIFO (*first in first out*) que contém instâncias de `MessageInvocation`. Para dar suporte ao método `dispatchWithPriority()`, a caixa de mensagens do ator passou a ser implementada por uma instância de `Deque[MessageInvocation]`. A interface `Deque`, que na prática é *subinterface* de `Queue`, apresenta como principal característica diferenciadora a possibilidade de inserir elementos tanto no começo quanto no final da fila. Assim, nosso despachador insere mensagens com prioridade no começo e mensagens normais no final da fila, e retira sempre do começo da fila a próxima mensagem a ser processada. A Figura 6.2 exibe de forma esquemática o tratamento de mensagens com prioridade (como as do tipo `MoveTo`).

Caso no momento da chamada a `dispatchWithPriority()` a caixa de mensagens do ator esteja vazia, tudo acontece como no caso de uma chamada a `dispatch()`: a mensagem é colocada na fila (até então vazia), e uma tarefa é iniciada. Por outro lado, caso existam mensagens na fila, existirá necessariamente uma tarefa em execução ou agendada para ser executada. No caso de uma tarefa já em execução, assim que ela terminar o processamento da mensagem atual, a próxima mensagem a ser processada será aquela despachada com prioridade.

Esse funcionamento mostra um dos motivos dos atores serem candidatos tão bons à migração. O momento da migração de um ator é muito bem definido: entre o processamento de duas mensagens

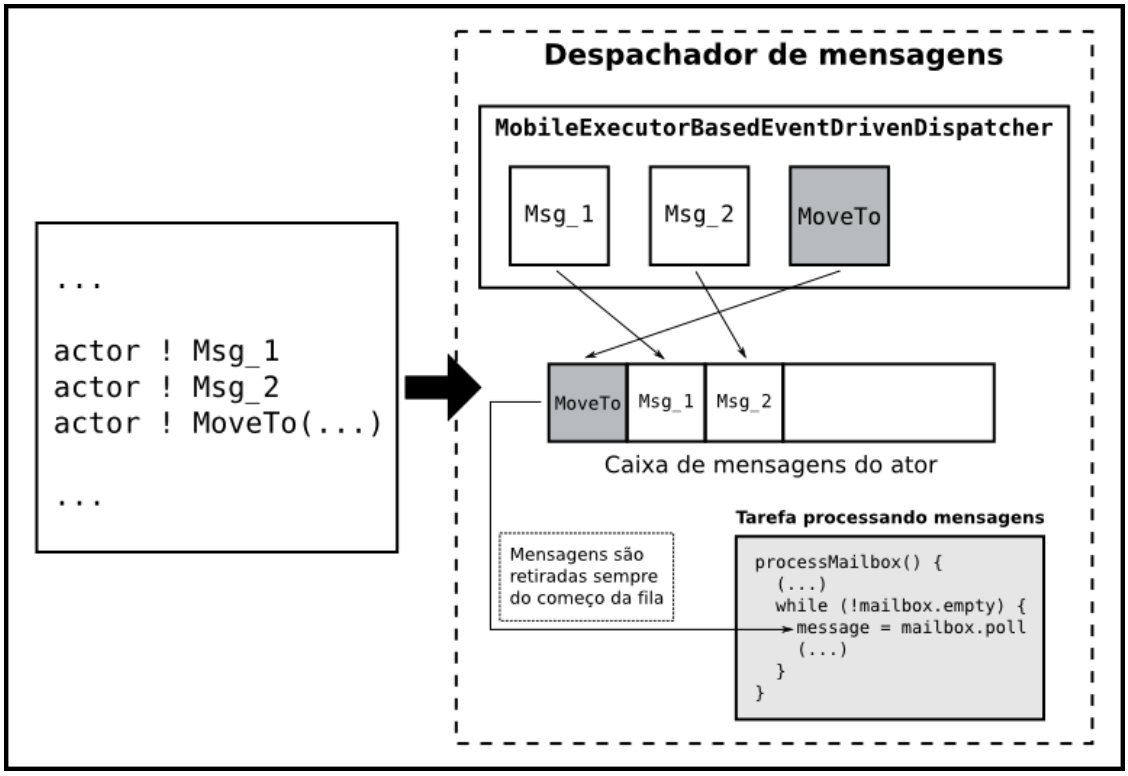


Figura 6.2: *Esquema de processamento de mensagens com prioridade (como as do tipo* `MoveTo`*).*

comuns (que não do tipo `MoveTo`). Após a migração, o ator, agora num novo nó, poderá continuar seu processamento de mensagens exatamente de onde parou, de forma bastante transparente.

Nossa implementação apresenta, porém, uma limitação: caso o ator já esteja processando uma mensagem no momento do envio da mensagem `MoveTo`, a migração será atrasada pelo tempo necessário para concluir esse processamento. Num caso extremo em que esse processamento nunca termine, devido a algum problema como *deadlock* ou laço perpétuo, a mensagem de migração nunca será processada, e a migração nunca ocorrerá.

6.2 Teatros

Os teatros são responsáveis por oferecer um ambiente de execução para os atores móveis da aplicação. Cada teatro encaminha as mensagens vindas de outros nós e destinadas a atores móveis nele hospedados. Além disso, os teatros trocam informações entre si para realizar serviços, como o de migração, para manter consistente o estado da infraestrutura e para manter atualizadas as referências para atores móveis.

As classes responsáveis pela implementação das funcionalidades dos teatros encontram-se dentro do pacote `theater`. A mais importante dessas classes é `Theater`, que contém os principais métodos associados a um teatro. A Figura 6.3 exibe uma representação interna de um teatro, mostrando seus principais componentes.

Nas próximos seções, daremos mais detalhes sobre os componentes de um teatro e como eles interagem entre si.

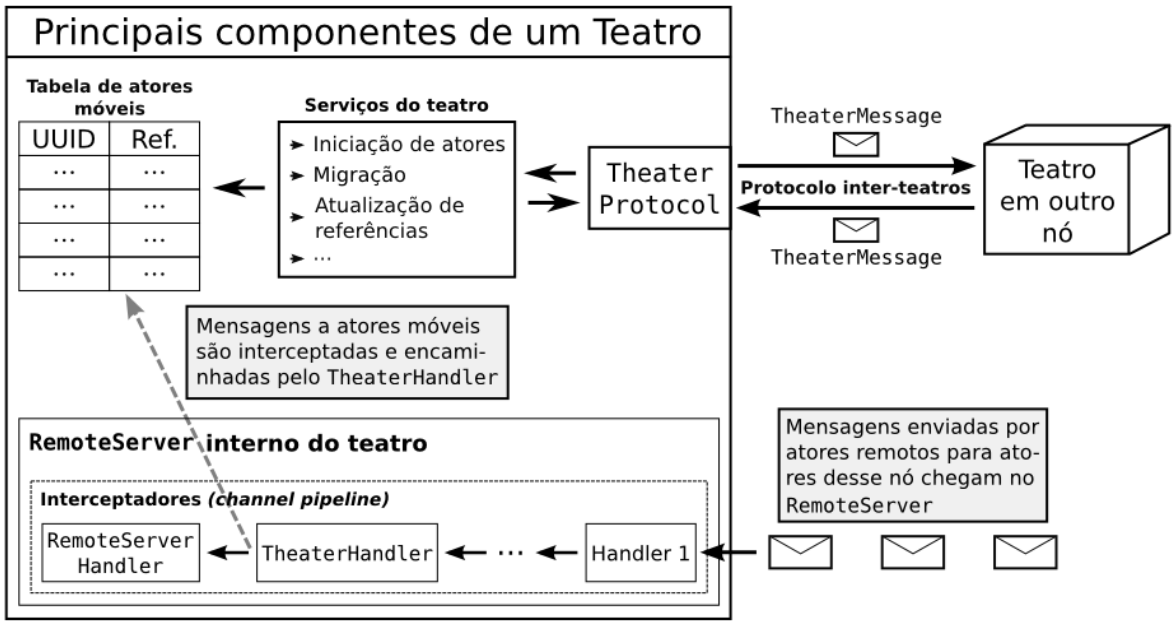


Figura 6.3: *Diagrama com as classes que implementam as referências a atores móveis da infraestrutura.*

6.2.1 Tabela de atores móveis

Os teatros possuem como atributo interno uma instância da classe `ConcurrentHashMap`, ou seja, um mapa associativo com suporte a acessos concorrentes. Esse mapa é parametrizado com os tipos `[String, MobileActorRef]`: cada entrada na tabela contém um par que associa um `UUID` (que é do tipo String e tem o papel de chave de busca) a uma referência para um ator móvel.

A maior parte dos serviços oferecidos por um teatro realiza consultas a essa tabela e/ou modificações nela. Em particular, os métodos de registro (`register()`) e remoção (`unregister()`) de atores modificam a tabela de modo a mantê-la atualizada.

6.2.2 `RemoteServer` interno

Um teatro funciona como uma extensão das funcionalidades já existentes nos servidores remotos do Akka. Dessa forma, internamente ele possui uma instância da classe `RemoteServer`, do Akka, sendo executada. Assim, toda a infraestrutura já existente para a troca de mensagens remotas é aproveitada. Todo teatro está associado a um *hostname* e porta específicos, e qualquer mensagem endereçada a esse teatro chega primeiramente em um `RemoteServer` vinculado a esse endereço.

O interceptador `TheaterHandler`

Como visto na Seção 5.1.1, um canal de comunicação entre um servidor remoto e um cliente (que, no nosso caso, será outro servidor remoto) possui o chamado *channel pipeline*. Esse *pipeline* é uma sequência de manipuladores (*channel handlers*) que atuam como interceptadores, efetuando alguma manipulação nas mensagens trafegando no canal e encaminhando-as para o próximo manipulador.

No caso dos teatros, foi preciso modificar o *pipeline* do servidor remoto interno, adicionando um novo manipulador, específico para a lógica interna do teatro. A classe `TheaterHandler` implementa esse interceptador, verificando se as mensagens chegando ao servidor remoto são destinadas a atores móveis.

As mensagens destinadas a atores chegando num servidor remoto são instâncias da classe `RemoteRequestProtocol`, definida pelo Akka usando o protocolo Protobuf. Na especificação dessa mensagem (num arquivo `.proto`), já aparecia originalmente um campo `ActorType`, identificando o tipo do ator destinatário da mensagem. Portanto, bastou adicionar um novo valor possível a esse campo para tornar as mensagens destinadas a atores móveis identificáveis. A feição `RemoteMobileActor`, que implementa as referências remotas a atores móveis, atribui o valor `MOBILE_ACTOR` ao atributo `ActorType` de todas as mensagens que são enviadas a atores móveis rodando em outro teatro.

Assim, o interceptador `TheaterHandler` verifica o valor desse atributo nas mensagens chegando ao teatro para decidir se elas se destinam a atores móveis. Em caso afirmativo, ele repassa essa mensagem diretamente à implementação do teatro, e não a encaminha ao próximo interceptador. A partir daí, o teatro procura o destinatário da mensagem em sua tabela de atores móveis e, ao encontrá-lo, envia a mensagem a ele. Caso a mensagem não seja destinada a um ator móvel, ela é encaminhada ao próximo interceptador, o manipulador `RemoteServerHandler`, definido pelo Akka (Seção 5.1.1). Nesse caso, o processamento se dará exatamente como ocorre originalmente no Akka (esse será o caso, por exemplo, de mensagens destinadas a atores estacionários, que serão descritos a seguir).

Os servidores do Netty, usados internamente nos servidores remotos do Akka, constroem um *pipeline* para um canal de comunicação aberto usando uma fábrica de *pipeline*, instância da classe `ChannelPipelineFactory`. Para nossa implementação, foi criado um novo tipo de fábrica de *pipeline*, `TheaterPipelineFactory`, que cria sequências de manipuladores idênticas às originais do Akka, porém adicionando uma instância de `TheaterHandler`. Assim, os servidores remotos internos dos teatros são praticamente idênticos aos originais do Akka, mas essa modificação em sua *pipeline* permite que eles se adaptem às necessidades de nossa infraestrutura.

Atores estacionários

Na Seção 5.2.5, vimos como se dá a instanciação de atores distribuídos no Akka. No caso de atores criados do lado do servidor, basta que eles sejam instanciados normalmente e então registrados no servidor remoto para tornarem-se acessíveis remotamente.

Atores registrados em um teatro também tornam-se acessíveis remotamente, já que mensagens destinadas a eles serão interceptadas no servidor remoto interno do teatro e encaminhadas corretamente. A infraestrutura implementa o conceito de *ator estacionário*, que nada mais é que um ator comum do Akka que é registrado junto a um teatro. Na prática, o teatro registra o ator diretamente em seu servidor remoto interno, exatamente como acontece originalmente no Akka. Os atores estacionários podem interagir normalmente com os atores móveis, mas estão sempre hospedados em seu teatro de origem, não podendo ser migrados.

A classe `Theater` implementa os métodos `registerStationaryActor()` e `unregisterStationaryActor()`, que permitem registrar e remover um ator estacionário de um teatro, respectivamente. Na próxima seção veremos o uso desse tipo de ator para implementar um protocolo de comunicação entre os teatros.

6.2.3 Protocolo inter-teatros

Os teatros rodando no aglomerado precisam se comunicar para coordenarem os diversos serviços oferecidos. O processo de migração, por exemplo, envolve a troca de mais de uma mensagem entre os teatros envolvidos: a primeira delas contém o ator seriado, e vai do nó de origem para o de destino, enquanto uma segunda mensagem, no sentido contrário, sinaliza que a migração foi finalizada e o ator já está rodando normalmente em seu novo nó. A partir daí, as referências para o ator podem ser atualizadas.

A comunicação entre teatros é realizada por um protocolo independente da implementação do teatro em si. Esse protocolo pode ser trocado pelo usuário, via arquivo de configuração. Assim, o usuário da infraestrutura pode escrever sua própria implementação de protocolo, caso deseje.

A classe `TheaterProtocol`

A classe abstrata `TheaterProtocol` representa o protocolo de comunicação entre teatros. Como podemos ver na Listagem 6.5, essa classe define um atributo e três métodos, sendo um deles abstrato.

```
abstract class TheaterProtocol {
  protected var theater: Theater = _

  def init(theater: Theater) {
    this.theater = theater
  }

  def sendTo(node: TheaterNode, message: TheaterMessage): Unit

  def stop() { }
}
```

Listagem 6.5: *Classe abstrata `TheaterProtocol`, cujas subclasses implementam o protocolo de comunicação entre os teatros.*

É provável que na maioria dos casos as subclasses de `TheaterProtocol` precisem estender o método `init()`, de modo a realizar as inicializações necessárias no protocolo. Todavia, é importante que essas subclasses chamem sempre o construtor de sua superclasse, para que o atributo `theater` seja iniciado corretamente. Já o método `stop()` deverá ser implementado caso o protocolo deseje realizar tarefas de encerramento quando o teatro não for mais usá-lo, como por exemplo fechar os canais de comunicação ou *sockets* com o teatro remoto.

O método `sendTo()`, por ser abstrato, deverá necessariamente ser implementado nas subclasses. Esse é o método mais importante do protocolo, pois é nele que a mensagem é de fato enviada via rede para o teatro remoto. Nesse método, cada protocolo definirá como codificar e enviar a mensagem pela rede.

Como se pode observar, o método `sendTo()` devolve `Unit`, ou seja, na prática não possui um valor de retorno. Isso significa que a comunicação entre os teatros, assim como entre atores, é assíncrona: um teatro envia uma requisição a um teatro remoto e não aguarda uma resposta.

Dessa forma, a comunicação entre os teatros é consistente com o restante da comunicação ocorrendo na aplicação, ou seja, a troca de mensagens assíncronas entre os atores. Além disso, isso

permitiu a construção de protocolos que usassem os próprios atores da infraestrutura para realizar a troca de mensagens entre teatros.

A feição `TheaterMessage` serve como ancestral comum a todas as mensagens possíveis de serem trocadas entre os teatros. Essa feição define um atributo chamado `sender`, de tipo `TheaterNode`, para que toda mensagem carregue consigo o endereço do teatro que a remeteu.

As mensagens propriamente ditas são implementadas como *case classes* que estendem a feição `TheaterMessage`. Ao longo do texto, serão mostrados exemplos dos principais tipos de mensagens trocadas pelos teatros.

Ao receber uma mensagem, o protocolo deve decodificá-la de forma apropriada, obtendo a partir daí a instância de `TheaterMessage` originalmente passada ao método `sendTo()` no teatro de origem. Após isso, o protocolo deve chamar o método `processMessage()` em `Theater`, passando como parâmetro essa instância de `TheaterMessage`. É nesse método que o teatro irá processar a mensagem e executar a ação requisitada por ela.

Nossa infraestrutura conta com três implementações de protocolos de comunicação entre teatros. Duas delas são bastante simples, baseadas em atores estacionários. Nesses protocolos, a troca de mensagens é realizada, na prática, por atores que "representam" os teatros. Já o último protocolo usa o arcabouço Netty para realizar a troca de mensagens.

Protocolo inter-teatros baseado em atores estacionários

A implementação mais simples de protocolo inter-teatros de nossa infraestrutura está na classe `AgentProtocol`. Nesse protocolo, a comunicação entre teatros é realizada por atores estacionários que rodam junto a cada um dos teatros do aglomerado.

Dentro do método `init()` desse protocolo, um ator comum (não móvel) é instanciado e registrado junto ao teatro, como um ator estacionário. Este ator é registrado com o nome `theaterAgent@HOSTNAME:PORTA`, onde `HOSTNAME` e `PORTA` são substituídos pelos valores correspondentes ao endereço daquele teatro.

Esse ator estacionário implementa um processamento de mensagens bastante simples: caso ele receba uma mensagem do tipo `TheaterMessage`, ele chama o método `processMessage()` da instância de `Theater` ao qual o protocolo está associado, passando a mensagem recebida como parâmetro.

No método `sendTo()`, basta ao protocolo enviar uma mensagem ao ator estacionário que implementa o protocolo no teatro de destino da mensagem. Para isso, ele precisa primeiro obter uma referência para esse ator, usando o método `RemoteClient.actorFor()`, como visto na Seção 5.2.5 para o caso de atores distribuídos instanciados pelo servidor. De posse da referência para o ator remoto que implementa o protocolo no teatro de destino, basta efetuar o envio da mensagem da maneira usual no Akka.

Refinamento do protocolo com o uso do Protobuf

Como visto na Seção 5.1.2, todas as mensagens trocadas entre diferentes máquinas no Akka são codificadas usando o Protobuf. Isso permite uma codificação mais eficiente das mensagens, em comparação ao uso do mecanismo de seriação de Java.

Assim, uma funcionalidade interessante para os protocolos inter-teatros é realizar a codificação

das mensagens também usando o Protobuf. Nossa infraestrutura possui uma implementação abstrata de protocolo para realizar essa função.

Primeiramente, um arquivo `.proto`, na linguagem do Protobuf, foi escrito contendo descrições de mensagens equivalentes a todas as mensagens possíveis de serem trocadas entre os teatros (ou seja, todas as classes que estendem `TheaterMessage`).

Além disso, a classe abstrata `ProtobufProtocol`, que estende a classe `TheaterProtocol`, realiza a conversão de mensagens do tipo `TheaterMessage` para mensagens no formato do Protobuf, e vice-versa. Essa classe também possui um método `sendTo()` abstrato, mas que no caso envia mensagens no formato do Protobuf. Classes que desejem enviar mensagens com esse tipo de codificação precisam apenas estender a classe `ProtobufProtocol` e implementar tal método.

Uma versão do protocolo baseado em atores estacionários, descrito anteriormente, foi implementada estendendo `ProtobufProtocol`. A classe `AgentProtobufProtocol` funciona de maneira idêntica à classe `AgentProtocol`, com a diferença de que as mensagens enviadas estão no formato do Protobuf. O principal ganho é no tamanho das mensagens enviadas pela rede, que diminui.

No entanto, vale notar que, nesse caso, uma mensagem do tipo `TheaterMessage` é codificada duas vezes. Primeiramente, ela é transformada numa mensagem do Protobuf específica para o protocolo inter-teatros. Porém, como a mensagem será enviada usando a infraestrutura do Akka para o envio de mensagens a atores remotos, ela será codificada novamente, em um outro tipo de mensagem (também do Protobuf) específico do Akka para mensagens a atores remotos. Essa mensagem contém campos adicionais necessários ao Akka, e portanto a mensagem poderia ser menor ainda caso tivesse sido enviada após a primeira codificação.

Protocolo inter-teatros baseado no Netty

Já vimos antes que a comunicação remota no Akka é feita usando o Netty, um arcabouço para a realização de comunicação assíncrona entre computadores remotos. Nossa infraestrutura implementa uma versão de protocolo inter-teatros também baseada no Netty, disponível na classe `NettyTheaterProtocol`. Essa é a implementação de protocolo mais robusta dentre as disponíveis na infraestrutura, e é a usada por padrão.

Nesse protocolo, cada teatro executa um servidor do Netty responsável por receber as mensagens de teatros remotos. Neste esquema cada máquina irá executar dois servidores do Netty, um relacionado ao servidor remoto do Akka rodando em cada teatro, e outro responsável pelo protocolo inter-teatros.

Esse servidor do Netty relativo ao protocolo possui como endereço o próprio *hostname* do teatro e uma porta especificada no arquivo de configuração. Cada vez que um teatro deseja se comunicar com outro, uma canal de comunicação do Netty é aberto entre as duas máquinas e a mensagem é enviada.

A classe `NettyTheaterProtocol` estende a classe `ProtobufProtocol`, e portanto envia mensagens codificadas usando o Protobuf. Nesse caso, como as mensagens são codificadas apenas uma vez e enviadas, o tamanho das mensagens enviadas é o menor dentre os três protocolos disponíveis. Além disso, o Netty possui um suporte nativo à mensagens codificadas com o Protobuf, facilitando bastante a implementação do protocolo.

6.3 Serviço de nomes

A capacidade de migração dos atores em nossa infraestrutura dá origem ao problema de manter as referências para atores sempre atualizadas. Ou seja, dada a migração de um determinado ator, como disseminar sua nova localização para todas as instâncias de `MobileActorRef` que representam esse ator nos diferentes nós do aglomerado.

Para resolver esse problema, nossa infraestrutura conta com um *serviço de nomes*. Esse serviço de nomes é essencialmente uma tabela que relaciona `UUIDs` de atores à sua localização (nome da máquina e porta do teatro que o hospeda). Sempre que ocorrer uma migração, essa tabela é atualizada. E, sempre que houver necessidade de consulta do endereço atual de um ator, o serviço de nomes é usado.

Nossa infraestrutura conta um com objeto *singleton* chamada `NameService`, que define a API do serviço de nomes. A Listagem 6.6 exibe os métodos dessa API. Os métodos de manipulação do serviço de nomes são todos de uso exclusivo da infraestrutura, com exceção de `get`. Assim, consultas ao serviço de nomes podem ser feitas também pelo código da aplicação do cliente, caso este deseje obter a localização atual de um dado ator.

```
private[mobile] def init(runServer: Boolean)

private[mobile] def stop()

private[mobile] def put(uuid: String, node: TheaterNode)

def get(uuid: String): Option[TheaterNode]

private[mobile] def remove(uuid: String)
```

Listagem 6.6: *API do serviço de nomes de nossa infraestrutura, que relaciona atores (por seu `UUID`) à sua localização atual.*

Implementação do serviço de nomes

O objeto `NameService` permite manipular o serviço de nomes da infraestrutura, mas não é ele que implementa tal serviço de fato. Ao invés disso, nosso sistema permite que essa implementação seja *plugável*, isto é, possa ser trocada pelo usuário caso deseje. A Seção 4.6.1 mostrou como esse tipo de configuração deve ser feita. Para uma classe ser usada como serviço de nomes, ela deve apenas implementar a feição `NameService`, que define exatamente a mesma API vista na Listagem 6.6.

Nossa infraestrutura já contém uma implementação de serviço de nomes, a classe `DistributedNameService`, que será usada por padrão caso o usuário não especifique uma diferente no arquivo de configuração. Essa é uma implementação de serviço de nomes distribuído, na qual a tabela que relaciona os atores a suas localizações é repartida entre diferentes nós. A execução desse tipo de serviço de nomes depende das seguintes configurações:

- **Nós participantes:** Define quais nós do aglomerado executarão um *servidor de nomes*, ou seja, irão manter parte da tabela do serviço de nomes. É o usuário quem especifica isso, via arquivo de configuração, e é obrigatório que ao menos um nó seja configurado para executar um servidor de nomes.

O parâmetro `runServer`, no método `init()`, indica se o nó local será um dos nós que participarão da execução do serviço de nomes. O código de iniciação da infraestrutura passará como parâmetro, nas chamadas ao método `init()` executadas nos vários nós, o valor da propriedade `cluster.NODE_NAME.name-server` especificado para cada nó no arquivo de configuração.

- **Função de espalhamento:** Define como mapear um `UUID` a um dos nós do aglomerado. Essa função deve devolver, para um dado ator, em qual máquina está a parte da tabela que contém (ou deverá conter) a localização desse ator.

  A função que será usada também pode ser trocada via arquivo de configuração. Por padrão, é usada uma função simples que obtém um inteiro a partir de uma cadeia de caracteres (nesse caso, o `UUID` do ator). Cada nó que participa do serviço de nomes é colocado então num vetor, de tal forma que cada nó receba um índice de 0 a N. É importante que o índice atribuído a cada nó seja o mesmo em todos os computadores do aglomerado, o que é garantido desde que o mesmo arquivo de configuração seja usado em todas as máquinas. Por fim, basta que a função de espalhamento seja configurada para devolver um inteiro entre 0 e N.

O serviço de nomes distribuído implementado em nossa infraestrutura é baseado em atores estacionários. Nas chamadas do método `init()` em que o parâmetro `runServer` é verdadeiro, é iniciado um ator no teatro local para desempenhar as funções do serviço de nomes. A Listagem 6.7 mostra a classe que implementa esses atores. Como é possível ver, algumas mensagens especiais foram definidas para indicar requisições relacionadas com o serviço de nomes.

```
class NameServiceAgent extends Actor {

  private val actors = new HashMap[String, TheaterNode]

  def receive = {
    // Registra um ator no serviço de nomes
    case ActorRegistrationRequest(actorUuid, hostname, port) =>
      actors += (actorUuid -> TheaterNode(hostname, port))

    // Remove um ator do serviço de nomes
    case ActorUnregistrationRequest(actorUuid) =>
      actors.remove(actorUuid)

    // Requisita a localização de um ator no aglomerado
    case ActorLocationRequest(actorUuid) =>
      actors.get(actorUuid) match {
        case Some(node) =>
          self.reply(ActorLocationResponse(node.hostname, node.port))

        case None =>
          self.reply(ActorNotFound)
      }
  }
}
```

Listagem 6.7: *Classe que define os atores que serão registrados como atores estacionários nos nós que executam uma parte do serviço de nomes distribuído.*

Cada um dos atores estacionários do serviço de nomes é registrado no teatro local com um nome padrão: `nameserver@HOSTNAME:PORTA`, onde `HOSTNAME` e `PORTA` correspondem ao endereço do teatro que hospeda o ator. Portanto, após executar a função de espalhamento e descobrir em qual nó do serviço de nomes está a informação pretendida, basta obter uma referência para o ator estacionário registrado, naquele nó, com o nome pré-definido. A manipulação do serviço de nomes pode ser feita então por meio do envio de mensagens a esse ator.

6.4 Gerenciadores de referências

A cada momento da execução de uma aplicação, uma máquina virtual contém um conjunto de referências a atores móveis. O teatro em execução nessa máquina virtual precisa conhecer todas essas referências, tanto as locais quanto as remotas. Esse conhecimento permite que a infraestrutura execute alguns de seus serviços e mantenha a consistência das referências.

Em cada nó existem dois objetos *singleton* responsáveis pelo gerenciamento de referências. Um deles gerencia as referências de fato, enquanto o outro é responsável pela administração dos grupos de atores co-locados.

6.4.1 `ReferenceManagement`

O objeto `ReferenceManagement` é responsável por diferentes tarefas relacionadas a manutenção da validade das referências a atores móveis. Internamente, ele possui duas tabelas, implementadas por mapas de espalhamento concorrentes (*concurrent hash map*):

Tabela de referências: Essa tabela associa `UUIDs` (cadeias de caracteres) a referências para atores móveis (`String` → `MobileActorRef`). Nela devem constar todas as referências atualmente em uso dentro de uma máquina virtual.

Tabela de ouvintes de clientes remotos: Essa tabela associa endereços de teatros a referências para atores normais do Akka (`TheaterNode` → `ActorRef`). Uma referência remota do Akka usa a classe `RemoteClient` para se comunicar com o nó que hospeda o ator que ela representa. A classe `RemoteClient` permite registrar ouvintes (*listeners*) que serão informados de eventos no canal de comunicação com o nó remoto, como por exemplo problemas que inviabilizem a troca de mensagens.

Um ouvinte nada mais é que um ator comum do Akka. Para cada teatro rodando no aglomerado, é instanciado um ator ouvinte, que é colocado nessa tabela. Toda referência remota a um ator móvel deve registrar-se junto ao ator ouvinte associado ao teatro em que se encontra o ator móvel referenciado. Para tanto, a referência usa métodos da classe `ReferenceManagement`. Esse arranjo permite que as referências sejam avisadas sobre problemas de comunicação com o teatro onde está o ator que elas representam e tomem as providências necessárias.

6.4.2 `GroupManagement`

O objeto `GroupManagement` mantém estruturas de dados e implementa métodos relacionados aos grupos de atores co-locados existentes numa aplicação. Quando um conjunto de atores é instanciado de forma co-locada (Seção 4.3.3), um novo registro é feito em `GroupManagement` associando

o identificador desse grupo com os `UUIDs` de todos os atores participantes. Essa classe possui dois mapas de espalhamento concorrentes (*concurrent hash map*):

Tabela de grupos: Essa tabela associa cadeias de caracteres a listas de referências a atores móveis (`String` → `List[MobileActorRef]`). A cada identificador de grupo de atores co-locados (que também é um identificador único universal, gerado com o mesmo algoritmo dos `UUIDs` dos atores móveis) está associada uma lista com as referências a todos os atores móveis que fazem parte desse grupo.

Tabela de tarefas de migração: A migração de um grupo de atores co-locados não é instantânea, pois é preciso aguardar até que todos os atores do grupo estejam prontos pra migrar. Por outro lado, existe um tempo máximo de espera (*timeout*) que, se atingido, faz com que a migração ocorra mesmo com o grupo incompleto. Esse mecanismo é implementado da seguinte maneira: para cada migração atualmente em curso há uma tarefa registrada para execução depois do tempo máximo de espera ou quando todos os atores estiverem prontos para migrar, o que ocorrer primeiro.

A implementação acima descrita usa uma tabela que guarda todas as tarefas de migração atualmente agendadas. Cada entrada nessa tabela associa um identificador de grupo de atores a uma tarefa (`String` → `GroupMigrationTask`). O mecanismo de migração de atores co-locados será melhor explicado na Seção 7.2.4.

Capítulo 7

Serviços da infraestrutura

Neste capítulo, mostraremos como os componentes estudados no Capítulo 6 são usados para oferecer os principais serviços da infraestrutura. Primeiramente descreveremos os serviços disponibilizados diretamente aos usuários: a distribuição inicial de atores (via `launch` ou `spawn`) e sua migração.

A seguir, serão descritos serviços não utilizados diretamente pelos usuários, mas importantes para a manutenção da consistência da infraestrutura, em particular após a migração de atores. Por fim, será estudado o módulo de monitoramento da troca de mensagens entre atores (*profiler*), que embora não esteja em uso atualmente, é um importante ponto de extensão para que algoritmos de reconfiguração dinâmica sejam incorporados à infraestrutura no futuro.

7.1 Distribuição inicial

Uma das principais funcionalidades oferecidas pela infraestrutura é a possibilidade dos usuários instanciarem atores sem especificar onde esses atores serão executados. Isso permite escrever o código da aplicação sem fazer nenhuma referência aos nós que compõem o aglomerado, que precisam apenas estar descritos no arquivo de configuração. Nesta seção explicaremos a implementação desse serviço de distribuição automática dos atores.

7.1.1 Algoritmos

Quando o código da aplicação chama o método `launch` (Seção 4.3.2), passando como parâmetro um ator móvel, a infraestrutura utiliza um algoritmo para decidir em qual nó da infraestrutura aquele ator será executado. A Listagem 7.1 mostra a feição `DistributionAlgorithm`, que define a interface que esses algoritmos devem implementar.

```
trait DistributionAlgorithm {
  def chooseTheater: TheaterNode
}
```

Listagem 7.1: *Interface que deve ser implementada pelos algoritmos de distribuição inicial.*

O método `launch` irá realizar chamadas ao método `chooseTheater` para cada ator instanciado, de maneira a decidir qual teatro irá hospedar aquele ator. As implementações desses algoritmos

devem fazer consultas ao objeto *singleton* `ClusterConfiguration` para obter a lista de todos os nós disponíveis no aglomerado.

Nosso sistema conta com duas implementações de algoritmos de distribuição inicial:

- **Aleatório:** Devolve um teatro escolhido aleatoriamente dentro da lista de nós do aglomerado.
- **Rodízio:** Implementa um algoritmo do tipo *round-robin*, que percorre a lista de nós do aglomerado de forma circular, devolvendo sempre o teatro que está há mais tempo sem ser selecionado.

O usuário tem total liberdade para implementar seus próprios algoritmos de distribuição inicial, bastando para isso estender a feição `DistributionAlgorithm`. A escolha do algoritmo que será usado pelo sistema é feita por arquivo de configuração, como visto na Seção 4.6. Caso nenhum algoritmo seja escolhido explicitamente, é usado como padrão o algoritmo de rodízio (*round-robin*).

7.1.2 Instanciação local de atores

Quando o algoritmo de distribuição inicial devolve o próprio nó local, basta à infraestrutura instanciar o ator móvel localmente. A instanciação de um ator móvel, assim como no caso dos atores do Akka, deve ser feita juntamente com a criação de sua referência. Como visto na Seção 6.1.2, as referências a atores móveis são compostas de duas camadas (interna e externa). Assim, o que ocorre de fato em nossa infraestrutura é:

1. A referência interna é instanciada juntamente com a implementação do ator. Essa referência é uma instância de `LocalActorRef`, do Akka, combinada com a feição `LocalMobileActor`.
2. A referência externa (`MobileActorRef`) é instanciada por meio de uma chamada que recebe como parâmetro a referência interna previamente criada. Assim, a referência externa irá "envolver" a referência interna e a ela delegar chamadas de métodos.

A classe `MobileActorRef` possui apenas um construtor, que recebe como parâmetro uma `InnerReference`. No entanto, esse construtor é privado, não podendo ser chamado diretamente. Ao invés disso, a instanciação de referências deve ser feita por meio de um conjunto de métodos de fábrica presentes no objeto acompanhante dessa classe. Como todos esses métodos têm o nome `apply`, eles podem ser chamados aplicando-se uma lista de argumentos ao objeto *singleton* `MobileActorRef` (como no exemplo da Listagem 2.2 na Seção 2.2.4).

A Listagem 7.2 mostra os cinco tipos diferentes de `apply()` do objeto `MobileActorRef`. Cada um deles tem uma utilização específica, a saber:

1. Método usado quando já se tem uma referência interna instanciada, mas ela ainda não possui uma referência externa correspondente. Esse caso só acontece após a migração, quando um ator é desseriado e uma nova referência interna para esse ator é construída.
2. Método usado quando o cliente da infraestrutura invoca `launch` ou `spawn` passando um trecho de código que instancie o ator, em geral uma chamada a um construtor com argumentos.
3. Método usado quando o cliente da infraestrutura invoca `launch` ou `spawn` passando apenas o nome da classe do ator que deve ser instanciado.

4. Método usado quando se deseja obter uma referência para um ator de `UUID` conhecido. Caso o ator exista em qualquer nó do aglomerado, ele será encontrado via serviço de nomes e a referência será devolvida. Caso o ator não exista, será devolvido o valor `None`.

5. Método usado quando se deseja obter uma referência externa que, na realidade, sirva como representante para um ator em execução num teatro remoto. Isso significa que tal referência irá envolver uma referência interna do tipo `RemoteMobileActor`.

```
1. private[mobile] def apply(reference: InnerReference): MobileActorRef

2. private[mobile] def apply(factory: => MobileActor): MobileActorRef

3. private[mobile] def apply(clazz: Class[_ <: MobileActor]):
   MobileActorRef

4. def apply(uuid: String): Option[MobileActorRef]

5. private[mobile] def apply(
     uuid: String,
     hostname: String,
     port: Int,
     timeout: Long = Actor.TIMEOUT): MobileActorRef = {
```

Listagem 7.2: *Métodos `apply()` usados para criar instâncias de `MobileActorRef`.*

O uso desses métodos dá a infraestrutura maior controle sobre as referências que são criadas. É garantido, por exemplo, que todas as referências locais instanciadas são registradas junto ao gerenciador de referências (`ReferenceManagement`) e ao teatro local. Também se garante que chamadas à forma 4 de `apply` que recebam como parâmetro o mesmo `UUID` devolvam sempre o mesmo valor, de modo a impedir que num dado teatro existam duas instâncias de `MobileActorRef` apontando para um mesmo ator.

Vale notar ainda que todas as versões do método `apply()`, exceto a número 4, possuem o modificador de acesso `private[mobile]` e portanto são de uso exclusivo da infraestrutura. As versões 1 e 5 do método são, de fato, usadas exclusivamente em procedimentos internos do sistema, e não devem ser acessíveis ao cliente. Já as versões 2 e 3 serão utilizadas após o usuário fazer chamadas aos diferentes tipos de `launch` e `spawn`. Esses dois últimos métodos são, portanto, a única maneira disponível aos clientes da infraestrutura para instanciar atores móveis. Por fim, a versão 4 do método `apply()` é de acesso público, já que ela não tenta instanciar um ator novo, mas sim obter uma referência para um ator já existente.

Um detalhe adicional é que todos os atores gerados com os métodos `launch` e `spawn` são automaticamente iniciados, ou seja, não há necessidade de chamar o método `start()` em uma referência devolvida por algum desses métodos.

7.1.3 Instanciação remota de atores

O comportamento esperado dos métodos `launch` e `spawn` é que eles devolvam imediatamente a referência ao ator instanciado, não importando se ele é local ou remoto. Isso possibilita uma

homogeneidade na utilização dos atores, de forma que o desenvolvedor da aplicação cria e usa os atores sempre da mesma forma, sem se importar com sua localização.

Para atender a essa exigência, o mecanismo de instanciação de atores remotos possui uma implementação mais complexa. Além disso, essa implementação é diferente para atores instanciados pelo nome da classe e para atores instanciados com um trecho de código, sendo este último o caso em que se deseja passar argumentos ao construtor do ator.

Instanciação de atores pelo nome da classe

Para o caso de atores instanciados pelo nome da classe, duas abordagens foram consideradas para iniciar o ator no nó remoto: *(i)* instanciar o ator localmente e depois usar o mecanismo de migração para enviá-lo ao nó correto; ou *(ii)* enviar ao teatro de destino uma solicitação, contendo o nome da classe, para que aquele teatro instancie um ator do tipo desejado.

A primeira possibilidade certamente seria mais simples, já que usaria ferramentas já existentes na infraestrutura. Por outro lado, a diferença na quantidade de informações que trafegam pela rede nos dois casos pode ser bastante grande: enquanto no segundo caso basta que seja enviada uma cadeia de caracteres (o nome da classe), no primeiro todo o estado do ator deve ser seriado e enviado.

Assim, procurando minimizar o tráfego de dados pela rede durante a execução das aplicações, decidimos implementar em nossa infraestrutura a segunda opção. Nesse caso, o teatro em que foi chamado o método de instanciação (`launch` ou `spawn`) envia uma requisição ao teatro onde o ator deverá ser executado, passando o nome da classe. Quando o ator tiver sido iniciado e estiver pronto para receber mensagens, o teatro remoto notifica o teatro de onde partiu a requisição. A partir daí, a referência para aquele ator pode ser ativada, passando a encaminhar as mensagens ao ator propriamente dito, no teatro remoto.

A maior dificuldade nessa abordagem resulta do fato de que todo esse processo de comunicação entre os teatros não é imediato, podendo levar um tempo razoável. Por outro lado, o método de instanciação deve devolver imediatamente uma referência utilizável para o ator.

A solução encontrada foi instanciar o ator localmente, porém imediatamente colocá-lo em *estado de migração*. Esse estado será explicado na Seção 7.2.1, porém resumidamente é o estado em que fica o ator **durante** a migração, ou seja, antes de ele estar totalmente estabelecido no nó de destino. Nesse estado, as mensagens enviadas ao ator são retidas localmente, para serem encaminhadas posteriormente ao ator já em seu nó de destino.

Assim, a referência devolvida comporta-se como se o ator estivesse migrando. Mas, diferentemente do processo de migração, que seria o estado do ator e o envia, apenas uma requisição é enviada ao nó de destino, solicitando a instanciação de um ator com base no nome de sua classe. Assim que a criação do ator tiver sido concluída com sucesso, o teatro de destino notifica o de origem, e tudo ocorre como durante uma migração normal: as mensagens retidas são encaminhadas ao novo teatro do ator e a referência torna-se remota. A partir daí, portanto, todas as mensagens enviadas a essa referência serão imediatamente encaminhadas ao teatro que hospeda o ator.

Como um ator foi instanciado localmente, a princípio ele aparece no serviço de nomes como estando hospedado no teatro onde a referência foi criada (ainda que, na prática, o ator nunca tenha estado hospedado de fato ali). Todavia isso não é um problema: as mensagens que vierem de nós remotos serão retidas na referência e processadas somente no teatro escolhido para hospedar

o ator. Portanto, ainda que a princípio o serviço de nomes contenha uma informação que pode ser considerada incorreta, esse mecanismo permite que, tão logo a referência seja devolvida pelo método `launch` ou `spawn`, ela seja válida, e o novo ator seja acessível a partir de qualquer nó do aglomerado.

Instanciação de atores usando um construtor com argumentos

Neste caso, o usuário especifica um trecho de código que, ao ser avaliado, deve resultar numa instância do tipo de ator desejado. Na prática, esse método é normalmente usado para instanciar atores com um construtor que receba argumentos.

Em tal situação, no entanto, não seria possível enviar ao teatro remoto uma solicitação com o trecho de código passado ao método de instanciação. Isso porque o envio de algum objeto pela rede requer que o objeto possa ser seriado, condição não satisfeita no caso de um conjunto de comandos arbitrário passado pelo usuário ao sistema em tempo de execução.

Portanto, a abordagem nesse caso é indireta, porém de implementação muito mais simples. O ator é instanciado localmente, exatamente como se seu teatro de destino fosse o teatro local. Porém, antes mesmo que sua referência seja devolvida a quem chamou o método de instanciação, a própria infraestrutura envia uma mensagem do tipo `MoveTo` ao ator, solicitando que o ator seja migrado para seu teatro de destino. Assim como no caso anterior, a referência é devolvida imediatamente, já que o processo de migração é assíncrono e permite que mensagens sejam enviadas ao ator enquanto a migração ocorre.

A principal desvantagem desse modo de instanciação é que a quantidade de informação que deverá trafegar pela rede pode ser razoavelmente maior do que no caso anterior. Como a primeira mensagem que o ator recebe é a de migração, sua caixa de mensagens certamente estará vazia no momento da seriação. Porém, todo o estado que o ator tenha no momento de sua instanciação será seriado e enviado ao nó remoto.

Portanto, os usuários da infraestrutura devem evitar usar esse tipo de instanciação sempre que possível. A menos, é claro, que seja usado o método `spawn` com o especificador `here`, solicitando que o ator seja instanciado localmente. Caso seja necessário usar o método `launch` para instanciar atores passando argumentos ao construtor, o ideal é cuidar para que o estado inicial do ator seja o menor possível.

7.1.4 Atores co-locados

A implementação da instanciação de um grupo de atores co-locados é bastante semelhante à de atores individuais. No caso da instanciação no teatro local, o processo efetuado para os métodos `launch` e `spawn` é essencialmente repetido para o número de atores que se deseja instanciar. No caso de instanciação remota, a mesma diferenciação discutida na seção anterior (nome da classe ou construtor com argumentos) é válida para o caso de atores co-locados.

O principal fator que distingue o caso de atores co-locados é o atributo `groupId`, presente nas referências a atores móveis. Todos os atores instanciados juntos, de maneira co-locada, devem possuir o mesmo valor de `groupId`. Ao atribuir um valor a esse campo, a infraestrutura automaticamente notifica o gerenciador de grupos `GroupManagement`, que mantém uma lista atualizada dos grupos existentes naquele teatro.

Uma observação importante é que o atributo `groupId` está, na realidade, na classe `MobileActor`. Por fazer parte do estado do ator, ele é automaticamente seriado no momento da migração, e portanto é mantido quando o grupo de atores muda de teatro. Por outro lado, esse campo só pode ser consultado para atores locais. Na realidade, o valor de `groupId` em uma referência para um ator remoto é sempre `None`, indicando que o ator não está em um grupo. Ainda assim, é possível que o ator propriamente dito esteja em um grupo no teatro que o hospeda.

Instanciação de atores próximos a algum ator

Como vimos na Seção 4.3.3, a instanciação de atores co-locados com o método `spawn` pode ser acompanhada do especificador `nextTo`, solicitando que os novos atores sejam instanciados juntamente com um determinado ator já existente na infraestrutura. Além de colocar todos os atores recém-instanciados no mesmo nó que o ator pré-existente, esse método ainda faz com que eles sejam colocados no mesmo grupo que aquele ator.

Na implementação desse tipo de instanciação, a principal diferença é quanto à atribuição do identificador para o grupo de atores. Enquanto nos outros casos esse identificador é criado aleatoriamente, com o mesmo algoritmo de geração de `UUIDs` de atores móveis, agora é preciso verificar se o ator passado como parâmetro já faz parte de um grupo e, em caso afirmativo, atribuir aos novos atores o identificador desse grupo.

Além disso, no caso de atores co-locados instanciados a partir de construtores com argumentos, o mesmo procedimento descrito anteriormente será adotado: os atores serão instanciados localmente e então uma mensagem de migração será enviada. Nesse caso, como precisamos da informação adicional sobre se tal grupo deve ser co-locado juntamente com um ator pré-existente, a mensagem que solicita a migração de atores co-locados deve possuir um campo adicional para indicar essa possibilidade.

Assim, a mensagem `MoveGroupTo` na verdade possui um campo adicional, além do nome da máquina e a porta que identificam o teatro de destino do ator. O terceiro campo, do tipo `Option[String]`, identifica, opcionalmente, o `UUID` do ator junto ao qual o grupo sendo migrado deve ser colocado.

No entanto, apenas a infraestrutura pode enviar solicitações de migração de grupos com o atributo `nextTo` diferente de `None`. Para os usuários, deve parecer que a mensagem `MoveGroupTo` possui apenas dois campos: nome da máquina e porta do teatro de destino. A Listagem 7.3 mostra como a definição da mensagem `MoveGroupTo` alcança esses objetivos.

```
case class MoveGroupTo private[mobile] (
  hostname: String,
  port: Int,
  nextTo: Option[String]) extends MigrationMessage

object MoveGroupTo {
  def apply(hostname: String, port: Int) =
    new MoveGroupTo(hostname, port, None)
}
```

Listagem 7.3: *Definição da mensagem `MoveGroupTo`, que solicita a migração de grupos de atores co-locados.*

É interessante notar como esse pequeno trecho de código exibe algumas das características específicas da linguagem Scala, apresentadas no Capítulo 2, e como tais características podem ser úteis no desenvolvimento de aplicações. Ainda que o detalhamento do uso dessas particularidades da linguagem não seja essencial para o entendimento da instanciação dos atores, a título de informação vale destacar os seguintes pontos:

- O uso de qualificadores no modificador de acesso `private`, aplicado ao construtor da classe, faz com que esse construtor só possa ser chamado pelos componentes da infraestrutura, pertencentes ao pacote `mobile`.
- A mensagem é uma *case class*, o que: *(i)* permite fazer casamento de padrões; *(ii)* facilita a criação de instâncias, pois dispensa o uso de **new**; e *(iii)* torna a mensagem imutável, uma característica essencial no modelo de atores (como observado na Seção 3.2.2). Vale ressaltar que essas vantagens não são exclusivas da mensagem `MoveGroupTo`, mas valem para todas as mensagens que serão trocadas entre atores. Por essa razão, todas as mensagens da infraestrutura são modeladas como *case classes*.
- O método `apply()` do objeto acompanhante de `MoveGroupTo` tem o papel de método d fábrica para a classe. Esse método é público e recebe apenas os parâmetros `hostname` e `port`, exatamente como desejado.
- O uso do tipo `Option` para representar o campo opcional `nextTo` deixa o código mais claro e menos suscetível a erros do que o uso de uma referência que pode assumir o valor `null`.

Açúcar sintático na instanciação de atores por construtores com argumentos

Na Seção 4.3.3, vimos os modos de se instanciar um grupo de atores co-locados. Como foi observado naquela seção, aparentemente há uma inconsistência entre a definição do método `launch` para atores co-locados (Listagem 4.7) e a utilização desse método (Listagem 4.8).

A aparente inconsistência surge porque os parâmetros do segundo tipo de `launch`, que recebe trechos de código, são do tipo literal funcional. Isso é evidenciado pela presença dos parênteses vazios `()` antes do tipo `MobileActor`. Enquanto o método `launch` para instanciação individual (Listagem 4.4) recebe um parâmetro por nome, isso não é possível para atores co-locados pois a linguagem Scala não permite parâmetros repetidos passados por nome. Ou seja, a seguinte definição de método, que seria intuitiva, causa um erro de compilação:

```
// ERRO DE COMPILAÇÃO
def launch(factories: (=> MobileActor)*): List[MobileActorRef]
```

Portanto, a única maneira de usar parâmetros repetidos nesse caso é com a passagem explícita de literais funcionais. Porém, apenas com a definição da Listagem 4.7, a instanciação de atores co-locados produzidos por fragmentos de código teria que ser feita da seguinte forma:

```
val clumsy = launch(
     () => new StatefulMobileActor(10),
     () => new StatefulMobileActor(100),
     () => new StatefulMobileActor(1000))
```

Obviamente, esse modo de uso é desagradável em comparação com a instanciação de atores individualmente. Portanto, para permitir uma usabilidade mais intuitiva como a vista na Listagem 4.8, implementamos uma conversão implícita (Seção 2.2.5), que será aplicada nos parâmetros de `launch` e `spawn`:

```
implicit def fromByNameToFunctionLiteral(byName: => MobileActor): () =>
    MobileActor = { () => byName }
```

Essa conversão transforma o parâmetro passado por nome em um literal funcional explícito. Quando ela é colocada no escopo da utilização de `launch` (por meio de `import Mobile._`), cada um dos construtores passados será convertido apropriadamente, obtendo-se o resultado esperado com uma interface mais agradável ao cliente.

7.2 Migração

A migração de um ator, iniciada por uma mensagem do tipo `MoveTo`, é efetuada por meio da execução de uma sequência de etapas coordenadas entre os teatros envolvidos. Ao longo do texto, chamaremos de T_O o teatro de origem do ator, aquele que o hospeda antes da migração, e de T_D o teatro de destino do ator, aquele para o qual ele será migrado. Além disso, chamaremos o ator sendo migrado (viajante, ou *traveler*) de A_T.

O processo de migração de um ator é composto por três etapas:

1. **Etapa 1:** Ocorre em T_O, onde se dá a iniciação da migração. Começa com o envio de uma mensagem do tipo `MoveTo` ao ator, e se estende até o envio do ator, em sua forma seriada, ao teatro T_D.

2. **Etapa 2:** Ocorre em T_D, correspondendo ao recebimento do ator pelo teatro que passará a hospedá-lo. Isso envolve essencialmente reconstituir o ator a partir de sua forma seriada e, após isso, notificar T_O sobre a conclusão do recebimento do ator.

3. **Etapa 3:** Ocorre em T_O, onde o processo de migração é finalizado. Ao receber a notificação de T_D, o teatro onde o ator originalmente era executado atualiza a referência para aquele ator, trocando sua referência interna de local para remota. Além disso, mensagens enviadas ao ator durante o processo de migração, que ficaram retidas, são encaminhadas ao teatro T_D para serem processadas.

Nas próximas seções, estudaremos em mais detalhes cada uma dessas etapas. Por fim, na última seção mostraremos as particularidades do mecanismo de migração conjunta de atores co-locados.

7.2.1 Etapa 1 – Iniciação da migração no teatro de origem

A Figura 7.1 exibe um diagrama com a sequência de chamadas de métodos envolvidas na implementação da primeira etapa da migração de atores. Nessa sequência, o ator sendo migrado A_T é representado pela referência `travelerRef` e a implementação `travelerImpl`. Observe ainda que, ao final da sequência, ocorre um envio de mensagem ao teatro de destino T_D, envio esse que dará início à etapa 2.

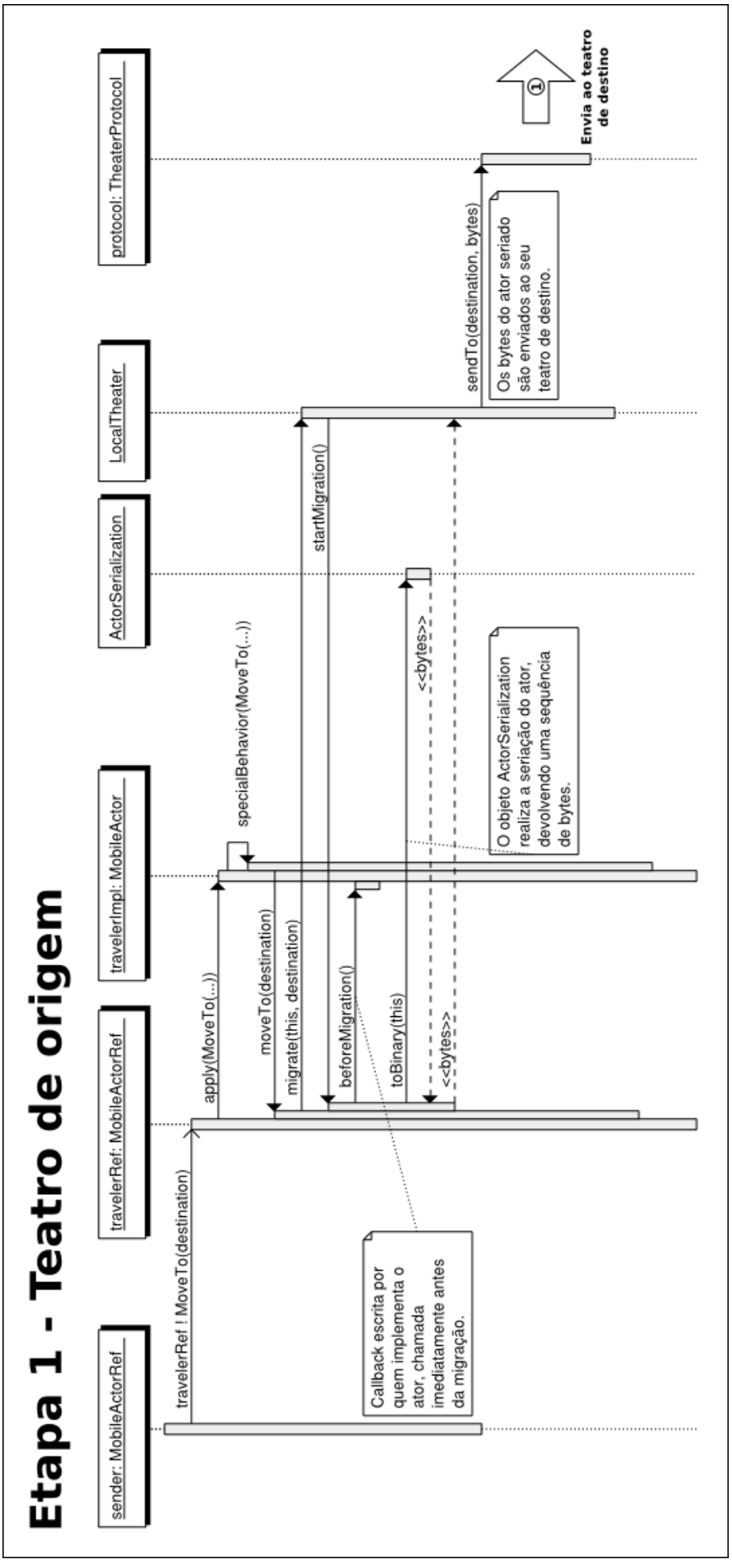


Figura 7.1: *Sequência de chamadas envolvidas na primeira etapa da migração de atores, que acontece no teatro de origem do ator.*

Solicitação de migração

Para dar início à migração de um ator, devemos enviar a ele uma mensagem `MoveTo(hostname, port)`, com o endereço do teatro de destino. Observe que, ainda que na Figura 7.1 essa mensagem tenha sido enviada por um outro ator (`sender`), uma migração pode ser iniciada de qualquer ponto da aplicação, não sendo necessário que essa mensagem parta de um outro ator. Além disso, a mensagem `MoveTo` pode ter sido enviada a um ator remoto, ou seja, a solicitação de migração de um ator pode partir de qualquer nó do aglomerado.

Ainda que o diagrama tenha sido simplificado para facilitar a visualização, sabemos, pela Seção 5.2.3, que o envio e o processamento de uma mensagem ocorrem em linhas de execução distintas, de modo a garantir que o envio seja assíncrono. Assim, após o envio da mensagem (com o método `!`), a referência a encaminha para o despachador de mensagens para que ela seja processada pela implementação do ator em algum momento futuro.

Um detalhe importante, no entanto, é que a referência verifica se a mensagem sendo enviada é de algum tipo especial (como é o caso de `MoveTo`) e, quando for o caso, a despacha com prioridade (Seção 6.1.3). Isso garante que essa mensagem seja a próxima a ser processada pelo ator.

Para efetuar o processamento da mensagem, o despachador de mensagens chama o método `apply()` da classe que implementa o ator. Como vimos na Seção 5.2.3, é nesse momento que será chamado o comportamento do ator, definido por meio das cláusulas *case* em seu método `receive`. No entanto, a feição `MobileActor` sobrescreve o método `apply()`, de maneira a interceptar certos tipos de mensagens que devem ser processadas pela própria infraestrutura e não pelo código do usuário.

Essa interceptação é feita pelo método `specialBehavior()`, parcialmente exibido na Listagem 7.4. Esse método funciona como um "pré-*receive*": para toda mensagem que chega ao ator, primeiro é testado se `specialBehavior()` é capaz de tratá-la, e somente em caso negativo ela é repassada ao comportamento do ator definido no método `receive`. Assim, mensagens do sistema nunca chegam ao `receive` escrito pelo usuário.

```
private val specialBehavior: Receive = {
  case MoveTo(hostname, port) =>
    outerRef.foreach(_.moveTo(hostname, port))

  case MoveGroupTo(hostname, port) =>
    outerRef.foreach(_.moveGroupTo(hostname, port))

  (...)
}
```

Listagem 7.4: *Método `specialBehavior`, da feição `MobileActor`, que intercepta e trata mensagens especiais da infraestrutura.*

Na Listagem 7.4, o atributo `outerRef` aponta para a referência externa relativa àquele ator. A sintaxe que usa esse atributo pode parecer um pouco confusa para aqueles com menor experiência em Scala, mas o que está ocorrendo é essencialmente uma chamada de método nessa referência. No tratamento de `MoveTo`, por exemplo, chama-se o método `moveTo()` na referência externa. O código adicional se deve ao fato de `outerRef` ser do tipo `Option`.

Ator em estado de migração

A execução do método `moveTo()` em `MobileActorRef` irá acionar o teatro local para dar início à migração. Quando finalmente o teatro chama o método `startMigration()` na referência de um ator móvel, a migração é de fato iniciada. A partir desse instante, o ator entra no chamado *estado de migração*. Esse estado é caracterizado pelo atributo `isMigrating` na referência do ator, que passa a ter valor `true`.

O ator em estado de migração tem seu processamento de mensagens suspenso. Isso envolve duas providências:

1. A referência deixa de encaminhar as mensagens ao despachador de mensagens do ator. Ao invés disso, ela retém essas mensagens localmente, numa fila de mensagens. Essa funcionalidade é implementada na classe `MessageHolder`, que é usada pela referência interna do ator para guardar temporariamente as mensagens.

2. O despachador de mensagens deixa de processar as mensagens que já estavam na caixa de mensagens do ator, conforme explicado na Seção 6.1.3.

Portanto, assim que o ator entrar em estado de migração, ele não processará mais nenhuma mensagem em seu teatro de origem. Todas as mensagens enviadas a um ator nesse estado serão processadas no teatro de destino, após a migração. Isso garante que nenhuma mensagem enviada a um ator móvel seja processada mais de uma vez.

Após entrar em estado de migração, mudando o valor do atributo `isMigrating`, a primeira providência tomada pela referência é chamar o método `beforeMigration()`, da implementação do ator. Esse é um método de *callback* escrito por quem implementa o ator, e é útil no caso em que se deseja efetuar ações imediatamente antes do ator ser de fato migrado. Assim que esse método retornar, o ator será seriado e enviado ao teatro de destino.

Seriação do ator

Quando um ator entra em estado de migração, sua execução está, na prática, temporariamente suspensa. Portanto, fazer a migração desse ator resume-se basicamente a seriar seu estado atual, enviá-lo a um outro nó, reconstruí-lo a partir do estado seriado e então continuar a execução (o processamento de mensagens) exatamente de onde ela foi interrompida.

A seriação de um ator em nossa infraestrutura usa o objeto *singleton* `ActorSerialization`, da implementação original do Akka (Seção 5.2.4). A seriação de todos os atores móveis é feita passando-se explicitamente aos métodos desse objeto uma instância de `Format` que usa a seriação Java. Na realidade, essa instância é precisamente o objeto definido na Listagem 5.7. Assim, uma exigência de nossa infraestrutura é que todas as classes que definem atores móveis possuam a anotação `@serializable`, que as torna seriáveis pelo mecanismo de Java.

Após obter a versão seriada do ator, uma sequência de *bytes*, o teatro de origem envia esses dados ao teatro de destino. Os *bytes* do ator são enviados "empacotados" numa mensagem do tipo `MovingActor(bytes)`, que estende a feição `TheaterMessage`, ancestral comum de todas as mensagens do protocolo inter-teatros.

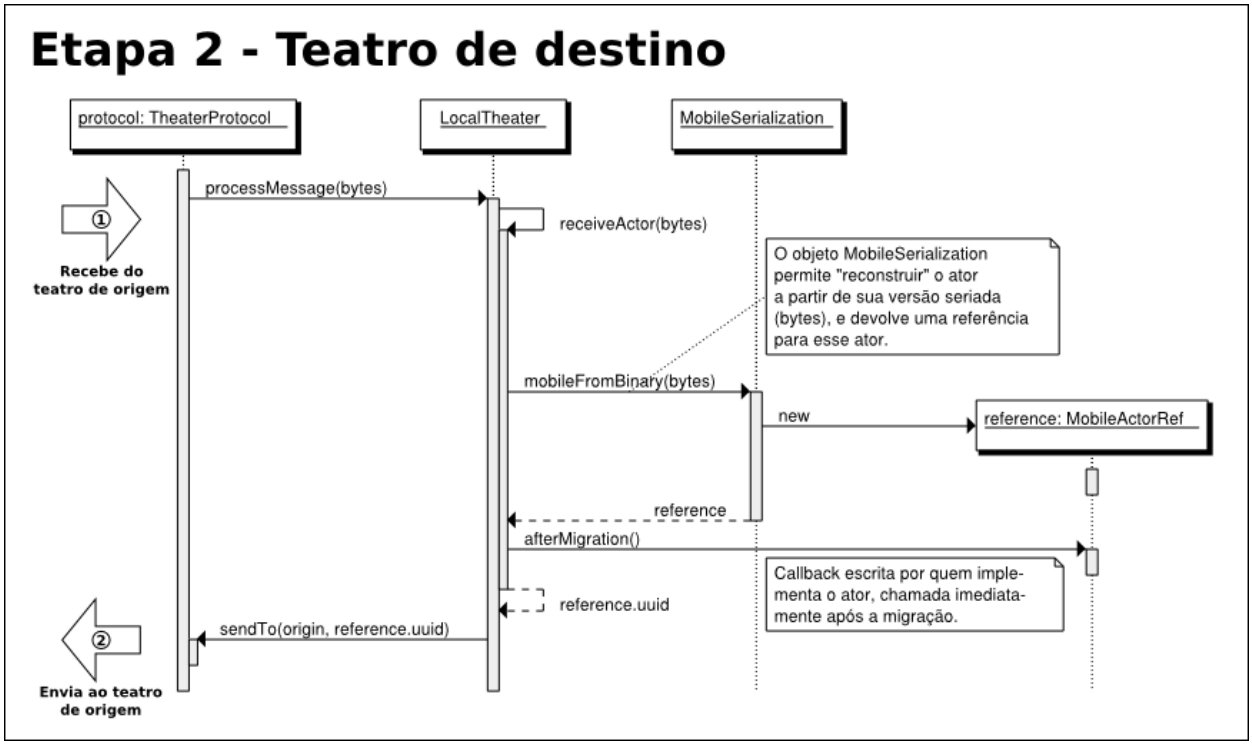


Figura 7.2: *Sequência de chamadas envolvidas na segunda etapa da migração de atores, que acontece no teatro de destino do ator.*

7.2.2 Etapa 2 – Recebimento do ator no teatro de destino

A Figura 7.2 exibe um diagrama com a sequência de chamadas de métodos envolvidas na implementação da segunda etapa da migração de atores. Essa sequência deve ser vista como uma continuação daquela exibida na Figura 7.1, sendo que a mensagem enviada ao fim daquele diagrama (`MovingActor`) é o que dá início ao processamento descrito neste. Além disso, o diagrama da etapa 2 termina com o envio de uma mensagem ao teatro de origem do ator, o que dará início à etapa 3 da migração.

Desseriação

Quando um teatro recebe uma mensagem do tipo `MovingActor(bytes)`, a primeira providência é efetuar a *desseriação* do ator, ou seja, reconstruir uma instância do ator a partir da sequência de *bytes* recebida.

Para isso, é usado o objeto *singleton* `MobileSerialization`. Esse objeto usa um mecanismo praticamente idêntico ao de seriação original do Akka, contido na classe `ActorSerialization`. No entanto, enquanto a seriação de atores móveis pôde usar a implementação original do Akka, o mecanismo de desseriação teve que ser sutilmente modificado. Essas modificações permitiram que a referência construída a partir do vetor de *bytes* fosse uma referência para atores móveis, instância de `MobileActorRef`.

Assim, a primeira modificação assegura que a instância de `LocalActorRef` que servirá de referência para o ator seja criada combinando-se a feição `LocalMobileActor`. Isso permite que tal referência seja usada como uma referência interna local para o ator móvel. Por fim, a segunda modificação é a construção de uma referência móvel externa, contendo a referência interna previa-

mente criada.

Exceto por essas modificações na instanciação da referência, todo o processo de desseriação propriamente dita do estado do ator foi aproveitado exatamente como na implementação pré-existente do Akka.

Logo após a criação da referência para o ator recém-migrado, a infraestrutura chama o método `afterMigration()` da implementação do ator. Esse é um método de *callback* escrito pelo desenvolvedor do ator, e é útil quando há ações que deverão ser executadas imediatamente após a migração do ator.

Envio de notificação ao teatro de origem

No momento da criação da referência externa do ator, após este ser desseriado, as seguintes providências são tomadas:

1. O ator é registrado no teatro local. Isso significa que ele é colocado na tabela de atores móveis interna desse teatro.
2. A localização do ator é atualizada no serviço de nomes distribuído. A partir desse instante, qualquer nó que tentar localizar o ator no serviço de nomes obterá seu novo endereço.
3. As mensagens que porventura estivessem na caixa de mensagens do ator antes da migração, e que portanto foram seriadas e enviadas juntamente com o estado do ator, são processadas.

Assim, nesse instante o ator está completamente estabelecido em seu novo nó, apto a receber mensagens (locais ou remotas) normalmente. Resta então ao teatro de destino notificar o teatro de origem do ator sobre a migração bem-sucedida.

Para isso, o teatro T_D envia a T_O uma mensagem do tipo `MobileActorsRegistered(uuids)`, contendo uma lista de cadeias de caracteres. Cada cadeia corresponde ao `UUID` de um ator recebido com êxito. Para o caso de migrações individuais, iniciadas com uma mensagem do tipo `MoveTo`, essa lista conterá apenas um `UUID`. Porém, a mensagem é usada também na migração de grupos de atores co-locados, situação em que múltiplos atores são recebidos e registrados por um teatro de uma só vez.

7.2.3 Etapa 3 – Conclusão da migração no teatro de origem

A Figura 7.3 exibe um diagrama com a sequência de chamadas de métodos envolvidas na implementação da terceira etapa da migração de atores. Essa sequência deve ser vista como uma continuação daquela exibida na Figura 7.2, sendo que a mensagem enviada ao fim daquele diagrama (`MobileActorsRegistered`) é o que dá início ao processamento descrito nesta seção. Esta é a última etapa da migração de um ator. Quando ela se encerrar, a migração terá sido concluída com êxito.

Como explicado na seção anterior, a mensagem `MobileActorsRegistered` é empregada tanto no caso da migração individual como no da migração conjunta de atores co-locados. Nesta seção, para fins de simplicidade, suporemos que tal mensagem contém apenas um `UUID`, ou seja, notifica o recebimento de apenas um ator. No caso geral, os procedimentos descritos a seguir serão executados para cada um dos `UUIDs` contidos na mensagem recebida.

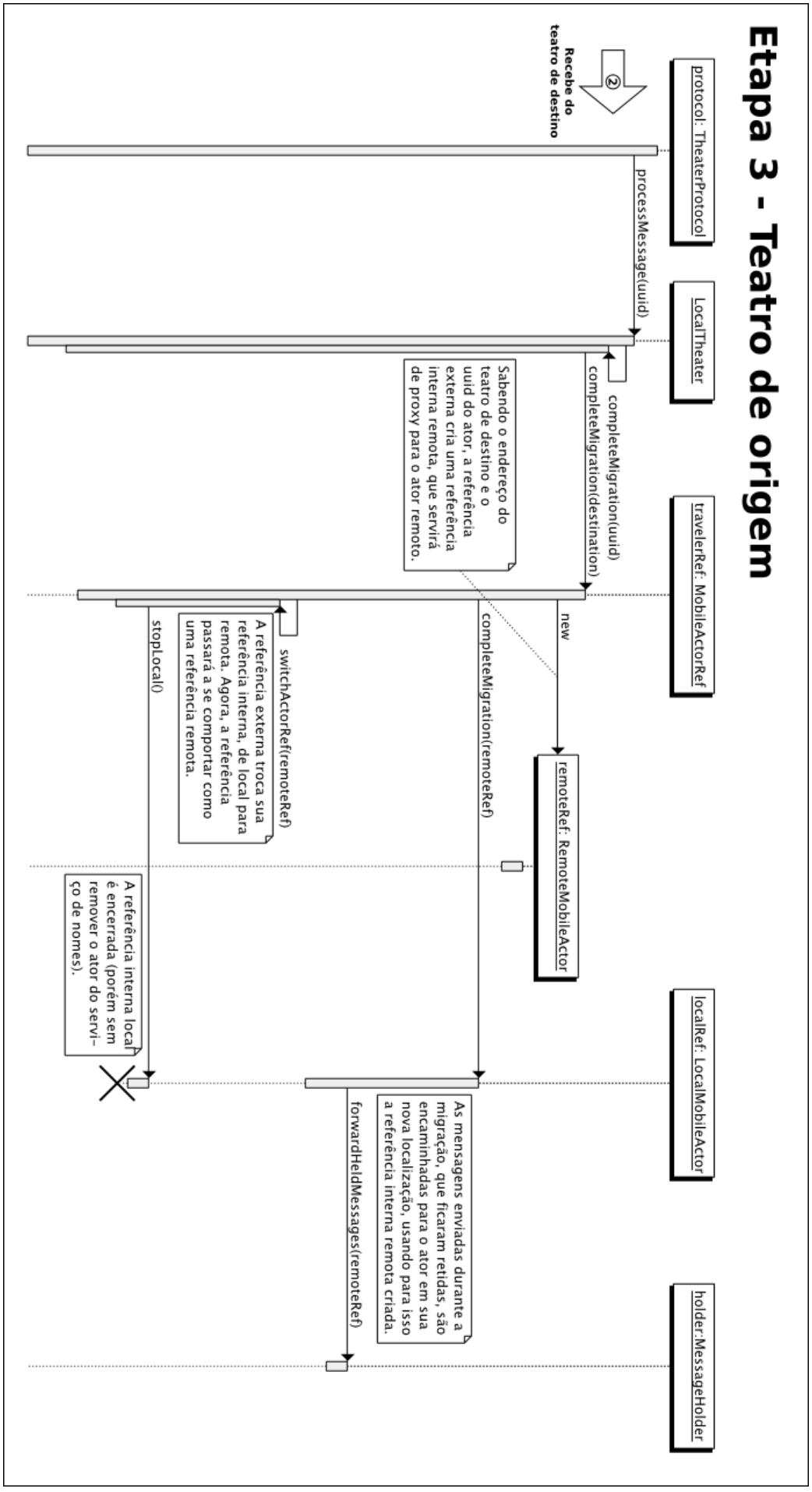


Figura 7.3: *Sequência de chamadas envolvidas na terceira e última etapa da migração de atores, que acontece no teatro de origem do ator.*

Troca da referência interna

Ao receber a notificação de que um ator foi migrado corretamente, a primeira providência que um teatro deve tomar é atualizar a referência desse ator. Nesse caso, a referência para tal ator que existe naquela máquina virtual é local, ou seja, possui como referência interna uma referência local. Naturalmente, essa referência interna deve ser trocada por uma referência remota que aponta para o novo endereço do ator. De fato, esse é o único momento em que a infraestrutura troca uma referência interna local por uma remota.

A infraestrutura cria então uma referência interna remota, instância de `RemoteMobileActor`, que servirá de representante para o ator recém-migrado. A criação dessa referência depende apenas do conhecimento do endereço do novo teatro do ator, já que o `UUID` permanece o mesmo durante toda existência do ator.

A infraestrutura usa então essa referência remota para encaminhar as mensagens que porventura tenham ficado retidas localmente. Como explicado anteriormente, ao entrar em *estado de migração* (etapa 1), a referência interna do ator passa a colocar as mensagens enviadas a ele numa fila de mensagens, ao invés de encaminhá-las ao despachador de mensagens.

Após encaminhar todas essas mensagens ao ator em seu novo nó, a referência interna local pode ser completamente desativada. Dentro do método `switchActorRef`, que troca a referência interna para a qual a referência externa delega suas chamadas, é chamado o método `stopLocal()` na referência sendo substituída. Portanto, ao realizar a troca de local para remota, a execução da referência interna local é interrompida. O método `stopLocal()` difere do também existente `stop()` no sentido que ele encerra o ator apenas localmente, removendo-o dos registros do teatro (local) mas não do serviço de nomes (global), já que o ator continua ativo, porém num nó diferente.

Vale ressaltar que todo esse processo é totalmente invisível para os usuários da referência. A menos que eles chamem os métodos `isLocal` ou `node` na referência externa, nenhuma mudança será notada na manipulação do ator. É claro, porém, que a manipulação de um ator remoto é suscetível a problemas na comunicação devido a falhas na rede.

7.2.4 Atores co-locados

A infraestrutura oferece a possibilidade de grupos de atores co-locados serem migrados em conjunto para um determinado teatro. Usa-se para isso a mensagem `MoveGroupTo`, de maneira análoga à mensagem `MoveTo`.

Como já foi dito, o mecanismo de migração funciona bem com o modelo de atores pelo fato de o momento em que a migração ocorrerá ser bem definido: entre o processamento de duas mensagens. Na prática, assim que o ator processar a mensagem especial que sinaliza a migração, o procedimento de mudança de nó é iniciado. No caso da migração de atores co-locados, essa abordagem torna-se mais complicada, já que é preciso aguardar o momento em que todos os atores do grupo estejam disponíveis para migração, ou seja, não estejam processando mensagens "normais" (que não sejam de migração).

Para lidar com essa dificuldade adicional, a migração de atores co-locados em nossa infraestrutura depende na realidade do envio de mensagens de dois tipos:

- **`MoveGroupTo`:** Com essa mensagem se dá início à migração de atores co-locados. Basta que ela seja enviada a apenas um dos atores do grupo, não importando qual deles. São os próprios

usuários da infraestrutura que enviam tal mensagem.

- **`PrepareToMigrate`:** Essa mensagem solicita que um ator entre em *estado de migração*, exatamente como se tivesse recebido uma mensagem do tipo `MoveTo`. Porém, o ator não é necessariamente migrado no exato instante em que recebe a mensagem. Ele aguarda até que todos os atores do grupo do qual ele faz parte estejam também preparados para migrar. Essa mensagem é de uso interno da infraestrutura e portanto o código da aplicação não pode enviá-la diretamente a um ator.

Dessa forma, os atores vão um a um entrando em *estado de migração*, situação na qual a migração pode ser feita sem grandes dificuldades. Quando o sistema verifica que todos os atores estão preparados para migrar, a migração de fato ocorre.

Gerenciador de grupos

O objeto *singleton* `GroupManagement` é encarregado de coordenar o processo de migração de atores co-locados. Assim, os atores que recebem a mensagem `PrepareToMigrate` notificam esse objeto de que estão prontos para migrar. Quando todos os atores de um determinado grupo estiverem prontos, o objeto invoca um método do teatro local para que a migração do grupo de atores de fato ocorra.

No entanto, na prática o que ocorre não é exatamente isso. O gerenciador de grupos não pode esperar indefinidamente para que todos os atores de um grupo estejam prontos para migrar. Um desses atores pode estar no meio do processamento de uma mensagem que levará muito tempo ou, por conta de algum erro de programação, pode até mesmo estar preso num *deadlock* ou laço perpétuo.

Portanto, a migração de atores co-locados tem um tempo máximo de espera (*timeout*). Esse tempo começa a ser contado a partir da solicitação de migração. Quando ele for atingido, a migração ocorrerá mesmo que nem todos os atores estejam preparados para migrar. Os atores que ficarem para trás serão migrados individualmente, tão logo respondam à mensagem `PrepareToMigrate`.

A migração de atores co-locados pode ser descrita pelas seguintes etapas:

1. Ao receber uma mensagem `MoveGroupTo`, um ator chama o método `startMigration()` do objeto `GroupManagement`.

2. Uma mensagem do tipo `PrepareToMigrate` é enviada a cada ator do grupo que está sendo migrado.

3. Uma nova tarefa é criada para gerenciar a migração, e é agendada para ser executada depois do tempo máximo de espera. Essa tarefa é uma instância da classe `GroupMigrationTask`, que guarda todas as informações sobre a migração, como o identificador do grupo e o endereço do teatro de destino.

4. Ao processar uma mensagem `PrepareToMigrate`, um ator chama o método `readyToMigrate()` do objeto `GroupManagement`. Nesse instante, o ator será seriado e seus *bytes* serão colocados num vetor juntamente com os dos outros atores já seriados, aguardando o momento da migração.

5. Ao se atingir o tempo limite, ou quando todos os atores do grupo tiverem chamado o método `readyToMigrate()`, a migração é efetuada. Para isso, o método `migrateGroup()` é invocado no teatro local, enviando o vetor de atores seriados ao teatro de destino.

6. Caso algum ator tenha ficado para trás, quando este chamar o método `readyToMigrate()` ele será migrado individualmente para o teatro que contém o grupo do qual ele faz parte.

7. Quando todos os atores do grupo tiverem sido migrados, a tarefa responsável pela migração desse grupo é removida da tabela de tarefas de migração do objeto `GroupManagement`. Tal migração é dada como encerrada.

Existe um cuidado especial que deve ser tomado na etapa 6, descrita acima. Quando o "ator atrasado" responder à mensagem `PrepareToMigrate`, não podemos simplesmente migrá-lo para o teatro de destino original da migração (contido na instância de `GroupMigrationTask`). Isso porque, nesse momento, o grupo já pode ter recebido outras mensagens de migração e pode estar num teatro diferente do que foi destino de sua migração originalmente.

A solução encontrada foi registrar a localização de todo grupo de atores co-locados no serviço de nomes, exatamente da mesma forma que é registrada a localização de atores. Como já vimos, um grupo possui um identificador, que aparece no atributo `groupId` de todos os atores pertencentes àquele grupo. Na prática, esse identificador é gerado com o mesmo algoritmo de geração de `UUIDs` dos atores, e portanto temos a garantia de que ele é único.

Assim, sempre que um novo grupo é criado ou migrado, sua localização atual é registrada no serviço de nomes. Essa informação é usada quando um ator que não migrou juntamente com seu grupo responde à mensagem `PrepareToMigrate`, e precisa saber onde o grupo ao qual pertence está localizado para juntar-se a ele. A Figura 7.4 exemplifica esse tipo de situação:

1. No instante $T1$, o grupo de atores 123 é migrado do $Teatro_A$ para o $Teatro_B$. Dois atores não respondem à mensagem `PrepareToMigrate` antes do tempo máximo de espera e, portanto, são deixados para trás.

2. No instante $T2$, o mesmo grupo de atores 123 recebe uma nova solicitação de migração, agora com destino ao $Teatro_C$.

3. No instante $T3$, o ator de `UUID` 777, que faz parte do grupo 123 mas tinha sido deixado para trás no $Teatro_A$, encerra o processamento de sua mensagem e responde à mensagem `PrepareToMigrate`. Nesse instante, a tarefa que cuida da migração do grupo 123 no $Teatro_A$ conhece, como destino do grupo, o $Teatro_B$. Porém, na realidade o grupo está localizado no $Teatro_C$, devido a uma segunda migração. Dessa forma, uma consulta é feita ao serviço de nomes pelo identificador do grupo, e o ator é migrado para o teatro que atualmente hospeda o grupo do qual ele faz parte.

Todo esse procedimento é gerenciado efetivamente pela classe `GroupMigrationTask`. Uma tarefa de migração assume três estados diferentes durante sua existência, e são esses estados que definem o que fazer com um ator que respondeu à mensagem `PrepareToMigrate`. Tais estados são:

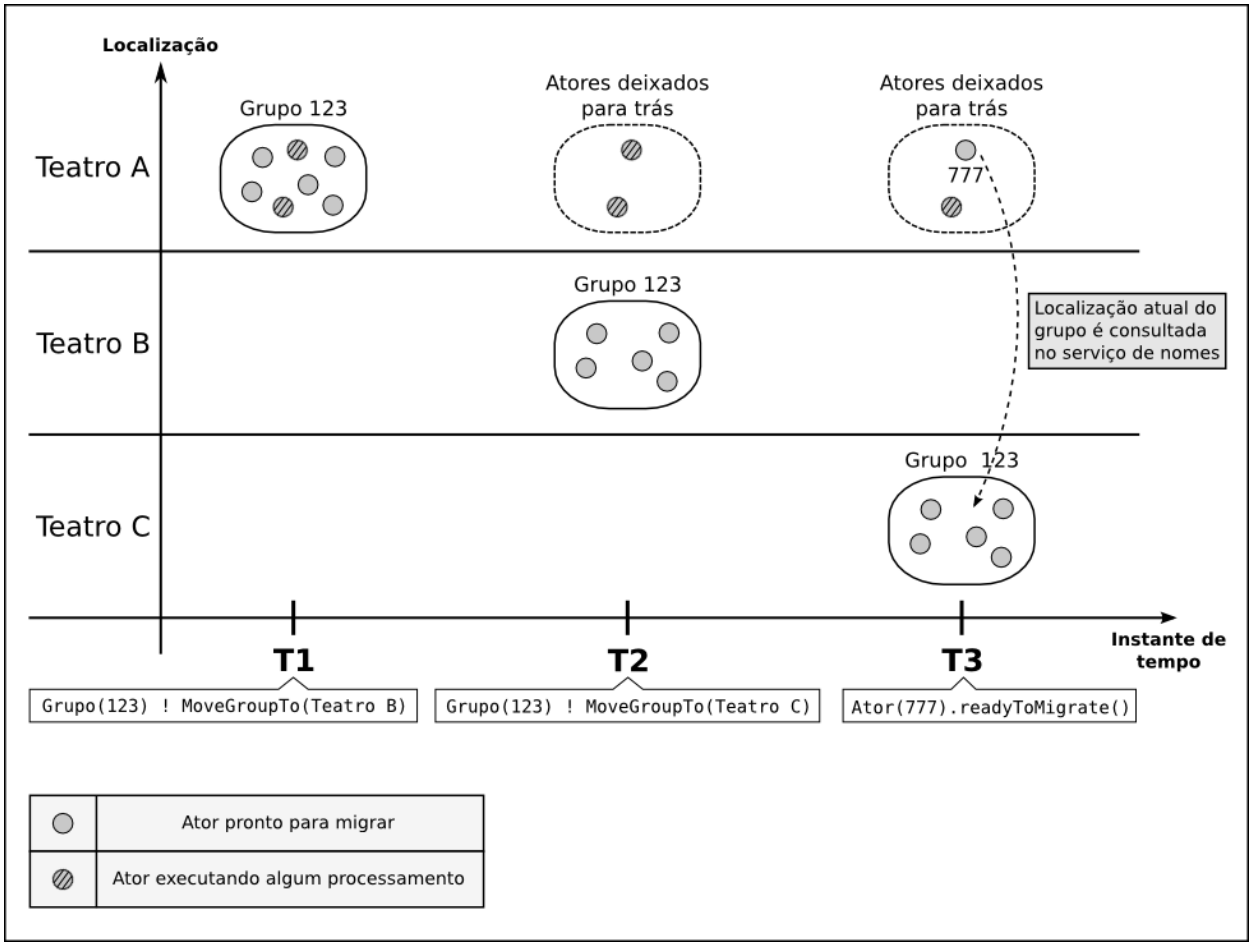


Figura 7.4: *Exemplo de situação de migração de um grupo de atores co-locados onde alguns atores ficam para trás no teatro de origem, e devem ser recolocados junto ao grupo assim que possível.*

- **Waiting:** A tarefa está esperando a migração ser de fato iniciada. Esse estado indica que o tempo de espera máxima ainda não foi atingido e que, até o momento, nem todos os atores do grupo estão prontos para migrar. Qualquer ator que chamar o método `readyToMigrate()` com a tarefa nesse estado será imediatamente seriado e colocado no vetor de *bytes* que será enviado ao teatro de destino da migração.

- **Migrating:** A migração foi iniciada, seja pelo esgotamento do tempo máximo de espera ou porque todos os atores responderam à mensagem `PrepareToMigrate`. No primeiro caso, se um ator chamar `readyToMigrate()` com a tarefa no estado *Migrating*, ele será migrado para o destino original da migração. Isso porque, nesse momento, a migração ainda não foi concluída e, portanto, o serviço de nomes ainda não foi atualizado para refletir a nova localização do grupo.

- **Migrated:** A migração foi concluída, e podemos supor que o serviço de nomes contém a localização atual do grupo. Assim, caso o ator chame `readyToMigrate()` com a tarefa no estado *Migrated*, uma consulta será feita ao serviço de nomes, passando o identificador do grupo, e o ator será migrado para o nó devolvido como resposta a essa consulta.

Recebimento dos atores e finalização da migração

Depois que o conjunto de atores sendo migrado é enviado, em sua forma seriada, ao teatro de destino, o processo de migração ocorre basicamente de forma igual à migração individual de um ator. O atributo `groupId`, disponível na referência externa, expõe na verdade um atributo de mesmo nome existente na classe `MobileActor`. Portanto, o identificador de grupo faz parte do estado do ator, e será seriado e desseriado junto com ele.

O teatro de destino portanto deverá apenas cuidar para que atores que possuam um identificador de grupo sejam registrados junto ao `GroupManagement` local. Além disso, o teatro também atualiza a informação sobre a localização do grupo recém-migrado no serviço de nomes, como vimos na seção anterior.

Por fim, o teatro de destino notifica o teatro de origem sobre a migração concluída. Como vimos anteriormente, o tipo de mensagem que informa sobre o término de uma migração é o mesmo para migrações individuais ou em grupo. Assim, o que será enviado na mensagem é na verdade uma lista de `UUIDs`, contendo o identificador de cada ator recebido. O teatro de origem dos atores realizará o processo de atualização das referências para cada um dos atores migrados.

7.3 Manutenção da consistência da infraestrutura

7.3.1 Atualização de referências ao ator migrado

Uma das grandes dificuldades a serem tratadas num ambiente que permita a migração de objetos é manter as referências a estes objetos atualizadas. Em nossa infraestrutura, isso não é diferente, em particular por partirmos do princípio da transparência da migração, segundo o qual o usuário que manipula o ator (por meio de sua referência) não precisa se preocupar se aquele ator foi ou está sendo migrado.

Portanto, tivemos que lidar com o problema de atualizar as referências para atores quando estes migram de um teatro para outro. Na Seção 6.1.2, vimos como a existência de duas camadas nas referências a atores móveis permite a atualização transparente dessas referências: enquanto o cliente manipula a referência externa, esta delega suas chamadas para uma referência interna que pode ser trocada sempre que ocorrer uma migração. Porém, ainda precisamos de um mecanismo que efetue de fato essa troca de referência interna no momento apropriado.

Unicidade de referências

Como explicado na Seção 7.1.2, as referências a atores móveis não são instanciadas explicitamente com o uso de **new**. Ao invés disso, o objeto acompanhante da classe `MobileActorRef` define algumas versões do método `apply()` que permitem fabricar tais referências.

Um desses métodos recebe apenas um `UUID` e devolve uma referência para o ator com esse identificador, desde que tal ator já exista na infraestrutura. A existência de um serviço de nomes distribuído permite encontrar qualquer ator em execução no aglomerado a partir de seu `UUID`, e portanto a tarefa de seriar uma referência para enviá-la remotamente torna-se tão simples quanto a de enviar o `UUID` do ator. O `UUID` permite identificar um ator e construir referências para ele em qualquer teatro da infraestrutura.

Porém, essa facilidade de criar referências poderia tornar um problema a manutenção das referências. Se cada vez que o método `apply()` fosse invocado com um determinado `UUID` ele devolvesse uma nova instância como referência para o ator correspondente, depois de algum tempo um mesmo ator poderia ter inúmeras referências diferentes apontando para ele de um mesmo nó. Isso dificultaria bastante o processo de atualização de referências.

Dessa forma, a implementação de nossa infraestrutura garante a *unicidade de referências em um mesmo nó*. Isso significa que, em um mesmo teatro, existirá no máximo uma referência para um dado ator. Assim, numa chamada ao método `apply()` passando um `UUID` como parâmetro, verifica-se junto ao `ReferenceManagement` se já existe uma referência instanciada para aquele ator. Em caso afirmativo, essa referência é devolvida.

Mecanismo de atualização de referências

A garantia de unicidade de referências facilita bastante sua atualização, já que, após uma migração, basta que cada teatro atualize a sua referência para o ator migrado, caso ela exista nesse teatro. Precisamos apenas de um mecanismo que, de alguma maneira, notifique cada teatro sobre as referências que precisam ser atualizadas. Para isso, consideramos adotar dois tipos de estratégias em nossa infraestrutura:

- **Atualização imediata:** Assim que a migração terminasse, um dos teatros envolvidos (origem ou destino) enviaria uma mensagem a todos os outros teatros, notificando a mudança de endereço do ator. Cada teatro, ao receber essa notificação, verificaria em `ReferenceManagement` se ele possui uma referência para aquele ator e, em caso afirmativo, a atualizaria com o novo endereço.

- **Atualização tardia:** Ao fim de uma migração, nenhuma referência seria atualizada (além, é claro, daquelas nos teatros de origem e destino). As referências seriam atualizadas posteriormente, no momento em que fosse feita uma tentativa de envio de mensagem para o ator em sua localização antiga. Seria responsabilidade do teatro antigo do ator (a origem da migração) notificar o teatro detentor da referência desatualizada sobre a mudança de endereço.

No caso da atualização imediata, uma vantagem é a garantia de que, a todo instante, a aplicação possui todas as suas referências a atores móveis atualizadas (descontando-se, é claro, o tempo transcorrido entre o fim da migração e a notificação a todos os teatros para que atualizem suas referências). Todavia, isso implicaria que todos os nós do aglomerado recebessem a notificação da migração, mesmo que eles não possuíssem uma referência para o ator em questão. Isso poderia representar um alto custo, em particular em aglomerados com muitos computadores. Uma alternativa seria manter uma lista, para cada ator, de todos os teatros que possuem uma referência para ele. Esse tipo de gerenciamento, no entanto, não é implementado em nossa infraestrutura, ainda que seja uma possibilidade interessante para trabalhos futuros.

Visando diminuir a quantidade de mensagens remotas trafegando pela rede, decidimos implementar o mecanismo de atualização tardia de referências. Assim, as referências vão sendo atualizadas uma a uma conforme são feitas tentativas de envio de mensagem para um endereço antigo do ator.

A Figura 7.5 ilustra como ocorre esse processo de atualização na infraestrutura, dividindo-o em quatro etapas:

1. A referência remota existente no $Teatro_A$ comunica-se normalmente com o ator A_{1234}, o qual ela representa, hospedado no $Teatro_B$. Essa referência remota possui como atributos o `UUID` do ator e o teatro onde ele está localizado.

2. O ator A_{1234} é migrado do $Teatro_B$ para o $Teatro_C$.

3. A referência em $Teatro_A$, agora desatualizada, tenta enviar uma mensagem a A_{1234}. Ao chegar em $Teatro_B$, essa mensagem dá início ao processo de atualização da referência incorreta. Esse procedimento será explicado em detalhes mais abaixo.

4. A referência em $Teatro_A$, agora com o atributo da localização do ator atualizado, volta a trocar mensagens normalmente com o ator A_{1234}.

Como visto, é na etapa 3 que ocorre de fato a atualização da referência. Essa atualização é composta dos seguintes passos:

1. As mensagens a atores remotos chegam primeiro no `RemoteServer` interno do teatro, como visto na Seção 6.2.2. A mensagem será interceptada pelo `TheaterHandler`, que irá procurar o destinatário da mensagem na tabela de atores móveis do teatro.

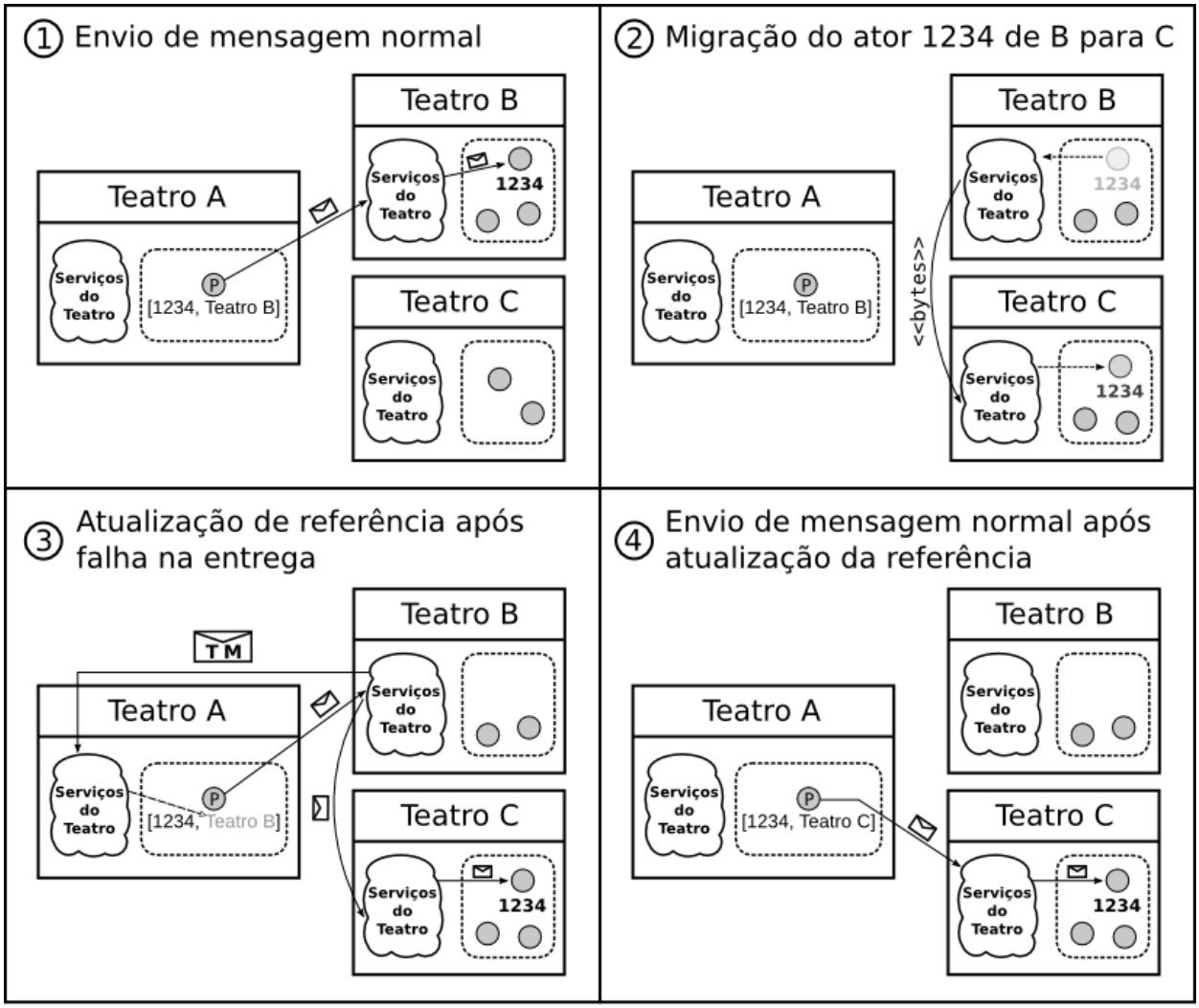
2. Como o ator foi migrado, o destinatário da mensagem não será encontrado nessa tabela. O teatro chama então o método `handleActorNotFound()`. Esse método irá usar o serviço de nomes para procurar a localização atual do ator em questão.

3. Caso o ator não seja encontrado no serviço de nomes, a mensagem é simplesmente descartada.

4. Caso o ator seja encontrado em algum nó, o próprio teatro se encarrega de encaminhar a mensagem para a localização correta do ator. Além disso, ele envia uma mensagem do tipo `ActorNewLocationNotification` ao teatro que remeteu a mensagem, usando o protocolo inter-teatros. Essa mensagem já contém o endereço atual do ator, evitando uma nova chamada ao serviço de nomes no teatro que atualizará a referência. Como vimos na Seção 6.1.2, as referências remotas a atores móveis "empacotam" as mensagens sendo enviadas numa instância de `MobileActorMessage`, contendo o endereço do teatro onde a referência está localizada. É a partir dessa instância que o teatro de destino da mensagem obtém o endereço para onde a mensagem `ActorNewLocationNotification` deverá ser enviada.

5. Ao receber essa notificação, o teatro que detém a referência incorreta a atualiza com o novo endereço do ator. Para encontrar tal referência, o teatro usa o objeto `ReferenceManagement`, o gerenciador de referências. Na prática, a referência interna não tem seu endereço atualizado. Ao invés disso, uma nova referência com o endereço correto é criada, e substituída como referência interna da referência externa que representa o ator.

Uma limitação desse procedimento é que ele depende da disponibilidade do teatro antigo do ator migrado. Caso o nó desse teatro sofra qualquer problema, a comunicação não será possível, ainda que o nó que de fato hospede o ator destinatário da mensagem esteja funcionando normalmente. Portanto, é importante que a infraestrutura implemente algum tipo de tratamento de problemas nas máquinas do aglomerado.



XXXX	Ator de UUID XXXX
[XXXX, TEATRO]	Representante (*proxy*) do ator XXXX, executando em TEATRO
	Mensagem para ator
TM	Mensagem do protocolo inter-teatros

Figura 7.5: *Sequência de etapas que ilustra o mecanismo de atualização de referências da infraestrutura.*

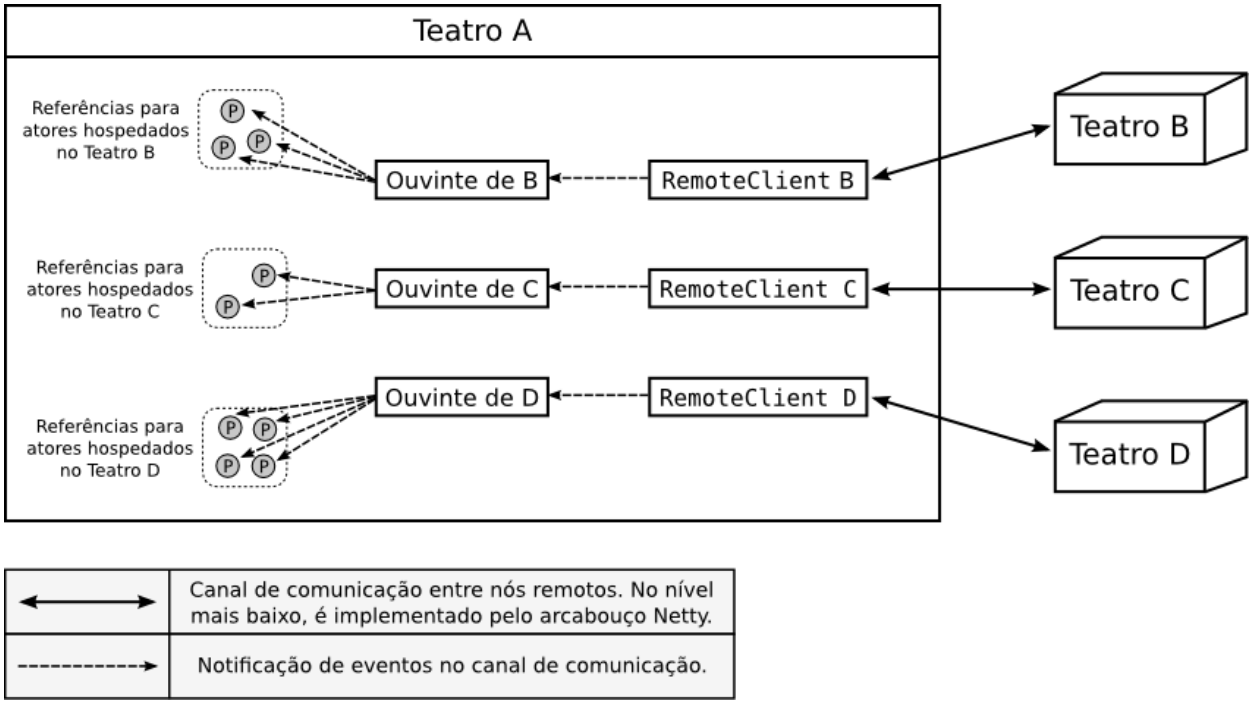


Figura 7.6: *Mecanismo de notificação às referências sobre eventos do canal de comunicação.*

7.3.2 Tratamento de quedas de nós

Nossa infraestrutura implementa um mecanismo básico de tolerância a falhas, para permitir que quedas em nós que hospedam teatros sejam detectadas e as referências remotas notificadas. Assim, as referências a atores em execução no computador problemático terão a chance de ser atualizadas, caso os atores tenham migrado antes do problema ocorrer.

Como vimos na Seção 5.2.5, as referências a atores remotos do Akka usam a classe `RemoteClient` para fazer a comunicação com o nó que hospeda o ator. Esse aspecto da comunicação funciona exatamente da mesma forma para os atores móveis.

A classe `RemoteClient` oferece a possibilidade de se registrar *atores ouvintes*, que receberão mensagens informando de eventos relacionados ao canal de comunicação entre o cliente e o servidor remoto. Assim, no caso por exemplo de uma queda no nó com o qual `RemoteClient` se comunica, um evento será gerado e repassado a todos os atores ouvintes registrados na classe.

A Figura 7.6 esquematiza o mecanismo de notificação sobre eventos no canal de comunicação. Em cada nó existe uma instância de `RemoteClient` para cada teatro com o qual os atores daquele nó se comunicam. A infraestrutura instancia, para cada um desses `RemoteClients`, um ator ouvinte, e o registra para receber eventos sobre o canal de comunicação.

A Listagem 7.5 mostra a definição da classe que implementa os atores ouvintes a serem registrados junto à classe `RemoteClient`. Essa classe possui internamente uma lista de referências remotas para atores móveis. Num dado ator ouvinte A_T, recebendo notificações sobre o canal de comunicação com um teatro T, por exemplo, deverão aparecer em sua lista todas as referências remotas representando atores que rodam em T.

A consistência desses registros é mantida usando o objeto `ReferenceManagement`. Como visto na Seção 6.4.1, esse objeto possui uma tabela de atores ouvintes de clientes remotos. Para cada teatro T da infraestrutura, existirá um (e apenas um) ator ouvinte A_T registrado no ob-

```
private class RemoteClientEventsListener extends Actor {
  private var references: List[RemoteMobileActor] = Nil

  def receive = {
    case event: RemoteClientLifeCycleEvent =>
      references.foreach(ref => ref.handleRemoteClientEvent(event))

    case AddReference(ref) =>
      references = ref :: references

    case RemoveReference(ref) =>
      references = references.filter(_ != ref)
  }
}
```

Listagem 7.5: *Classe que implementa os atores ouvintes que serão registrados para receber eventos relativos ao canal de comunicação com o teatro remoto.*

jeto `ReferenceManagement`, recebendo notificações sobre o canal de comunicação com T. Sempre que uma referência remota para um ator hospedado em T é iniciada, ela solicita ao objeto `ReferenceManagement` que seja colocada na lista de referências de A_T.

Assim, caso algum nó apresente problemas e a comunicação com ele seja interrompida, o ator ouvinte relativo àquele nó será notificado e repassará essa notificação a todas as referências apropriadas, chamando o método `handleRemoteClientEvent()` da feição `RemoteMobileActor`. Existem diferentes tipos de notificação, mas atualmente as referências remotas tratam apenas de avisos de desconexão com o nó remoto, representados pela classe `RemoteClientDisconnected`.

Ao receber uma notificação desse tipo, a referência remota fará uma busca no serviço de nomes pelo `UUID` do ator que ela representa. Caso o endereço retornado seja o endereço atual para o qual a referência aponta, isso significa que o ator não havia migrado antes de ocorrer a desconexão com o teatro remoto. Nesse caso, a referência não toma ação alguma, e ela permanecerá inválida até que, eventualmente, a conexão seja restabelecida.

Caso o serviço de nomes devolva um endereço diferente do atualmente apontado pela referência, no entanto, esta será atualizada para o novo endereço. Esse seria o caso em que o ator havia migrado, mas a referência ainda não tinha tentado se comunicar com ele e portanto, pelo mecanismo de atualização tardia de referências, ainda não havia sido atualizada.

7.4 Monitoramento da troca de mensagens

Como visto na Seção 4.5, a infraestrutura registra informações sobre as mensagens trocadas entre atores. Mais precisamente, um teatro pode contabilizar, para cada um dos atores móveis que hospeda, quantas mensagens esse ator recebeu de cada um dos outros teatros da infraestrutura. Para fins de comparação, o número de mensagens locais recebidas por tal ator também é contabilizado.

Esse tipo de informação pode ser importante para sistemas que desejem usar o mecanismo de migração para reconfigurar uma aplicação em tempo de execução. Um algoritmo de balanceamento de carga, por exemplo, poderia usar esses registros para juntar, num mesmo nó, atores que se comunicam intensivamente e, dessa forma, diminuir a quantidade de dados trafegando pela rede. Ainda que nossa implementação não inclua esse tipo de algoritmo, nossa infraestrutura foi construída

tendo em mente esse tipo de utilização e, portanto, a construção de uma plataforma desse tipo sobre a nossa já contará com algumas facilidades.

A classe `Profiler` é a responsável por gerenciar os dados sobre o monitoramento da troca de mensagens, além de oferecer uma API para que os clientes da infraestrutura tenham acesso aos resultados do monitoramento.

Todo teatro possui um atributo `profiler` de tipo `Option[Profiler]`. A razão desse atributo ser do tipo `Option` é que, como explicado no Capítulo 4, a execução do monitoramento é opcional em cada teatro. Por padrão, o monitoramento não é efetuado, de maneira a evitar custos adicionais na execução da infraestrutura. Nesse caso, o atributo `profiler` do teatro local terá como valor `None`.

No entanto, caso o usuário especifique, via arquivo de configuração, a execução do monitoramento para um determinado conjunto de nós, os teatros rodando nesses nós terão, em seu atributo `profiler`, uma instância válida de `Profiler`. Por meio dessa instância, os clientes consultarão os dados sobre a troca de mensagens entre os atores.

A notificação do recebimento de mensagens é efetuada pelos próprios atores. A feição `LocalMobileActor`, que adiciona às referências locais o comportamento desejado para atores móveis, modifica o método de envio de mensagens (`!`) para que mensagens remetidas a um ator sejam registradas junto ao `Profiler`. Naturalmente isso só ocorre caso o atributo `profiler` do teatro que hospeda tal ator seja diferente de `None`.

A Listagem 7.6 mostra os métodos usados para notificar o recebimento de uma mensagem. Os dois primeiros métodos deverão ser chamados pela referência, dependendo do remetente da mensagem ser um ator local ou remoto. A referência identifica mensagens remotas por estas virem "envelopadas" numa mensagem do tipo `MobileActorMessage`, que contém o endereço do teatro que hospeda o remetente da mensagem.

Deve ser notado que os dois primeiros métodos fazem uma chamada ao método que de fato irá realizar o registro de recebimento da mensagem. A principal diferença nos dois casos é o valor do parâmetro `usePriorityQueue`. O uso da fila de prioridade para armazenar os registros de mensagens recebidas será explicado na próxima seção. No entanto, deve ficar claro que registros relacionados ao envio de mensagens locais nunca são colocados na fila de prioridade.

7.4.1 Estruturas de dados da classe `Profiler`

A informação sobre quantas mensagens um ator específico recebeu de um determinado teatro é armazenada por instâncias da classe `IMRecord` (*Incoming Messages Record*). Essa classe já foi descrita brevemente no Capítulo 4, e seus principais atributos podem ser vistos na Listagem 4.11.

Em Scala, uma classe definida como *case class* implementa automaticamente o método `equals()` em função dos argumentos da classe. No caso de `IMRecord`, isso significa que duas instâncias da classe serão consideradas iguais caso seus atributos `uuid` e `from` sejam iguais, não importando para o teste de igualdade o valor do campo `count`.

A classe `Profiler` possui duas estruturas de dados para armazenar as instâncias de `IMRecord`, que representam os registros de trocas de mensagens entre os atores: uma tabela de espalhamento e uma fila de prioridade.

```
private def messageArrived(
   uuid: String,
   from: TheaterNode,
   usePriorityQueue: Boolean) {
  ...
}

private[mobile] def localMessageArrived(uuid: String) {
  messageArrived(uuid, localNode, false)
}

private[mobile] def remoteMessageArrived(
   uuid: String,
   message: MobileActorMessage) {
  messageArrived(
    uuid,
    TheaterNode(message.senderHostname, message.senderPort),
    true)
}
```

Listagem 7.6: *Métodos da classe `Profiler` que notificam o recebimento de uma mensagem por um ator.*

Tabela de espalhamento

Todos os registros sobre mensagens recebidas por atores em um determinado teatro ficam armazenados em uma tabela de espalhamento da classe `Profiler`. Essa é na verdade uma tabela de tabelas, ou uma tabela de espalhamento de dois níveis. Seu tipo exato é `HashMap[String, HashMap[TheaterNode, IMRecord]]`.

A tabela externa é indexada pelos `UUIDs` dos atores. Cada entrada nessa tabela associa um ator (representado por seu `UUID`) a uma outra tabela, que contém os registros das mensagens recebidas por esse ator. Essa tabela interna discrimina quantas mensagens esse ator recebeu de cada um dos teatros do aglomerado. Teatros de onde o ator nunca tenha recebido mensagens não aparecerão na tabela interna.

A informação sobre a quantidade de mensagens recebidas, apesar de na prática ser apenas um número inteiro, é representada por uma instância de `IMRecord`. Nesse caso, os atributos `uuid` e `from` dessa instância podem ser deduzidos pela localização dessa instância nas tabelas de espalhamento. Todavia, esse esquema é usado pois essa mesma instância poderá ser colocada numa outra estrutura de dados, como veremos na próxima seção.

As mensagens locais recebidas pelos atores também são registradas nessa tabela, de forma exatamente igual às mensagens remotas. No caso, a tabela interna de um determinado ator conterá uma entrada indexada pelo endereço do teatro local, que armazenará a contagem de mensagens que tal ator recebeu de outros atores localizados na mesma JVM que ele.

Fila de prioridade

Essa estrutura de dados é responsável por armazenar as instâncias de `IMRecord` ordenadas pelo seu atributo `count`. O registro que possuir o mais alto valor de `count` será o de maior prioridade e, portanto, o primeiro da fila. A classe `Profiler` disponibiliza, em sua API, um método para obter a instância de `IMRecord` com o maior valor de `count`, portanto o primeiro elemento da fila

de prioridade.

É importante ressaltar que nessa fila somente são colocados registros sobre mensagens recebidas de nós remotos. Assim, sua principal finalidade é dar destaque aos atores que mais recebem mensagens de atores em outros teatros e, portanto, possivelmente sejam bons candidatos à migração.

Além disso, como essa fila tem a finalidade de destacar bons candidatos à migração, pode fazer sentido colocar atores nessa estrutura apenas após eles receberem um número pré-determinado de mensagens. Sem esse controle, caso um teatro tenha recebido poucas mensagens remotas, mesmo o ator de maior prioridade nessa fila terá recebido, em termos absolutos, um número pequeno de mensagens, que muito possivelmente não justifique o custo de sua migração para outro nó.

Assim, a classe `Profiler` conta com um atributo para definir o limiar mínimo de mensagens que um registro deve atingir antes de ser colocado na fila de prioridades. Somente instâncias de `IMRecord` cujo campo `count` ultrapasse esse limiar serão inseridas na fila. O valor do limiar mínimo é definido pelo usuário, via arquivo de configuração.

Para implementar essa fila de prioridade foi usada a classe `PriorityBlockingQueue`, de Java. Caso vá armazenar instâncias de uma classe arbitrária, devemos fornecer à fila, no momento de sua criação, uma instância da interface `Comparator`, que irá definir como os elementos colocados na fila se comparam e, por consequência, quem tem prioridade sobre quem.

Um detalhe interessante da classe `Profiler` é que o critério de comparação a ser usado pela fila de prioridade diverge do critério de igualdade definido para a classe `IMRecord`: enquanto a comparação de igualdade entre duas instâncias dessa classe, pelo método `equals()`, não leva em conta o atributo `count`, o critério que define as prioridades dos elementos é justamente o valor de `count`.

Assim, a implementação de `Comparator` passada na instanciação da fila de prioridade ordena instâncias de `IMRecord` de maneira que as com o maior valor de `count` tenham maior prioridade. Instâncias com o valor de `count` iguais são dadas como iguais, já que sua prioridade é a mesma. A ordenação imposta por tal `Comparator` é dita ser *inconsistente com `equals`*, pois duas instâncias dadas como iguais pelo `Comparator` podem ser consideradas diferentes pelo método `equals()`. O uso de comparadores inconsistentes pode causar problemas em certas situações, mas em nossa infraestrutura esse tipo de utilização permite que as duas estruturas de dados de `Profiler` funcionem corretamente.

Por fim, vale notar que a classe `PriorityBlockingQueue` não dispõe de métodos para atualizar a posição de um elemento já inserido na fila. Assim, a cada atualização do atributo `count` de uma instância de `IMRecord`, é necessário removê-la da fila de prioridade e inseri-la novamente. Uma estrutura especializada, que apenas atualizasse a posição do elemento dada sua nova prioridade, seria mais eficiente. Usando um *heap* como implementação da fila de prioridade, por exemplo, essa operação de atualização poderia ser feita em tempo $O(lg\ n)$, onde n é o número de elementos na fila de prioridade.

7.4.2 Limpeza periódica dos dados registrados

O sistema de monitoramento de troca de mensagens de nossa infraestrutura permite que os registros de mensagens recebidas sejam apagados de tempos em tempos. Isso evita que informações antigas, que possivelmente já não refletem o estado atual da aplicação, influenciem a tomada de decisões sobre a migração de atores.

A classe `Profiler` implementa o método `reset()`, que é de acesso público. Sua execução apaga todas as instâncias de `IMRecord` tanto da tabela de espalhamento quanto da fila de prioridade. A partir desse momento, o monitoramento de mensagens recebidas passa a se comportar como se a aplicação tivesse acabado de ser iniciada.

Ao invés de ter que se preocupar em chamar o método `reset()` explicitamente, o cliente da infraestrutura pode configurar a classe `Profiler` para que esta invoque esse método periodicamente, em intervalos de tempo pré-definidos. Esse tipo de configuração pode ser feita diretamente na classe `Profiler`, ou via arquivo de configuração.

Caso o usuário opte pela limpeza automática dos registros, o sistema inicia uma *thread* que será responsável por esse serviço. Essa *thread* executa um laço enquanto a opção por remoção automática dos dados estiver ativada. Dentro desse laço, a *thread* "dorme" pelo período especificado e, ao acordar, chama o método `reset()`.

Capítulo 8

Trabalhos relacionados

8.1 *Worldwide Computing Middleware*

Um extenso trabalho foi desenvolvido pelo *Worldwide Computing Laboratory*[1] (WCL), sob a chefia do professor Carlos Varela, no sentido de desenvolver uma plataforma para a execução de aplicações distribuídas usando a infraestrutura da rede mundial de computadores, a Internet [5, 35].

O chamado *World-Wide Computer* (WWC) é construído tendo como base o modelo de atores. Ele permite a reconfiguração das aplicações rodando sobre ele por meio da migração de tais atores. Além disso, um serviço de nomes escalável foi implementado, de modo a permitir a localização dos atores na rede a qualquer momento após sua migração.

Juntamente com a implementação do middleware em si, a equipe do WCL desenvolveu ainda uma linguagem de programação baseada em atores naturalmente voltada para o desenvolvimento de aplicações distribuídas na Internet, e em particular no WWC. Essa linguagem, chamada SALSA, é pré-compilada para Java e permite a execução de operações vitais ao WWC, como registro e busca no serviço de nomes e migração de atores.

Além do desenvolvimento da linguagem SALSA e do WWC, uma linha de pesquisa que se desenvolveu bastante no grupo do WCL foi o estudo de diferentes algoritmos de balanceamento de carga em sistemas distribuídos. Diferentes estratégias quanto a decisão de migrar atores entre nós do WWC foram testadas e comparadas entre si, permitindo uma melhor tomada de decisão quanto a qual algoritmo se encaixa melhor em cada situação [14].

8.1.1 Arquitetura

A arquitetura da infraestrutura proposta para o WWC pode ser dividida em três camadas:

- A camada da aplicação, onde se destaca o suporte nativo da linguagem SALSA a medidas adaptativas, como a migração.
- O *World-Wide Computer*, uma infraestrutura que oferece suporte à execução de aplicações escritas em SALSA, permitindo a comunicação remota e a migração de atores. Essa camada é composta por teatros (*theaters*), que hospedam os atores, e por servidores de nomes.
- O middleware de reconfiguração dinâmica (IOS), que monitora a execução da aplicação e toma decisões quanto a sua reconfiguração.

[1] http://wcl.cs.rpi.edu

A linguagem SALSA

A linguagem SALSA (*Simple Actor Language, System and Architecture*) [36] implementa o modelo de atores. É uma linguagem que guarda semelhanças com linguagens orientadas a objetos: ao invés de classes, ela define comportamentos (*behaviors*), e todo ator herda de um comportamento comum, o `UniversalActor`. Um código-fonte em SALSA é primeiramente traduzido para Java para então ser compilado normalmente em *bytecodes* e executado na JVM.

Dentro de um comportamento, o programador define as mensagens que um determinado ator é capaz de processar. Essas mensagens são implementadas como métodos tradicionais de Java: possuem um nome, parâmetros e um tipo de retorno. Seu uso, no entanto, é bastante diferente, já que o envio de uma mensagem é assíncrono. A sintaxe também é um pouco diferente: ao invés de usar ponto (`.`) antes do nome do método, o símbolo `<-` é usado para indicar a passagem de mensagens.

Ainda que a definição de uma mensagem especifique um valor de retorno, como num método tradicional de Java, o assincronismo impede que esse valor seja obtido pelo remetente da mensagem no momento do envio. Para permitir a obtenção desse valor e o encadeamento de chamadas, a linguagem implementa três variações do conceito de continuações:

1. *Token-Passing Continuations:* Permitem que o valor devolvido pelo processamento de uma mensagem, quando obtido, seja enviado como parâmetro de uma outra mensagem para algum ator. Usam o símbolo `@`, como no exemplo:

   ```
   fractal <- computePixel()@ screen <- draw(token);
   ```

2. *Join Continuations:* Permitem sincronizar a execução da aplicação, criando uma barreira que aguarda até que todas as mensagens especificadas tenham sido processadas pelos atores respectivos.

3. *First-Class Continuations:* Parecidas com as continuações baseadas em *token*, as continuações de primeira classe são independentes do contexto da mensagem atual. Tais continuações podem ser continuamente passadas a uma função recursiva para, por exemplo, executarem alguma ação ao final da recursão.

Além desses mecanismos, a linguagem SALSA provê um suporte nativo para funcionalidades relativas ao WWC. O comportamento `UniversalActor` implementa mensagens que permitem registrar um ator no serviço de nomes e também solicitar a migração do ator. Todo *ator universal* possui associado a si dois identificadores:

- *Universal Actor Name* (UAN): Nome único que representa o ator durante toda a sua existência. Um UAN possui o endereço do serviço de nomes onde a localização atual do ator pode ser encontrada.

- *Universal Actor Locator* (UAL): Endereço atual do ator. É composto essencialmente pelo endereço do teatro que atualmente hospeda o ator, seguido por algum identificador particular do ator.

Por padrão, um ator SALSA não possui um nome. Quando se vincula um ator a um par (UAN, UAL), o ator é automaticamente registrado no serviço de nomes apropriado e migrado para o

endereço relativo ao UAL. É importante ressaltar que o UAN associado a um ator é fixo, não podendo ser modificado após a vinculação inicial. A linguagem permite a obtenção de referências a atores remotos a partir do UAN ou do UAL, e a partir daí a comunicação remota se dá de forma totalmente transparente.

Além disso, a migração de um ator pode ser efetuada enviando a ele uma mensagem do tipo `migrate`. Essa mensagem deve ter como parâmetro o UAL do teatro de destino do ator. O processamento dessa mensagem é responsabilidade de `UniversalActor`, e ocorre de forma totalmente transparente ao desenvolvedor da aplicação.

World-Wide Computer

O WWC é a infraestrutura que de fato dá suporte à execução das aplicações baseadas em atores SALSA em uma rede de larga escala como a Internet. Essa infraestrutura é formada basicamente por duas entidades: teatros e servidores de nomes.

Os teatros são os responsáveis pela hospedagem dos atores e participam do processo de migração. Toda a interação entre teatros, seja a troca de mensagens remotas entre atores ou a migração de algum ator, se dá por meio de um protocolo próprio, o *Remote Message Sending Protocol* (RMSP). Um teatro é composto por um servidor RMSP, que recebe mensagens nesse protocolo, e uma tabela dos atores ali hospedados, o que permite o encaminhamento das mensagens recebidas.

Ao receber um ator sendo migrado, o teatro verifica se esse ator possui como parte de seu estado referências a outros atores universais. Em caso afirmativo, o próprio teatro se encarrega de atualizar essas referências, trocando, se for o caso, referências locais por remotas e vice-versa. Os teatros podem ainda conter *atores do ambiente*, que representam recursos locais (como a saída local, por exemplo) e são, portanto, imóveis. As referências para tais atores também são atualizadas apropriadamente após a migração de um ator.

Outra tarefa que cabe a um teatro é manter um ator temporário após um ator ter deixado esse teatro via migração. Esse ator temporário recebe mensagens enviadas ao ator migrado e as encaminha para seu novo endereço. Após um tempo, o ator temporário deixa de existir e mensagens enviadas incorretamente geram uma consulta ao serviço de nomes, de modo a obter a localização atualizada do ator.

O serviço de nomes é composto por diversos servidores, que rodam de maneira independente dos teatros. Um servidor de nomes guarda uma tabela relacionando UANs a UALs. A comunicação com esses servidores se dá por um protocolo específico, o *Universal Actor Naming Protocol* (UANP), que permite manipular os registros daquela tabela. Além disso, a referência a um ator possui o UAN e o UAL do ator, sendo que uma consulta só será feita ao serviço de nomes caso o UAL ainda não seja conhecido ou esteja desatualizado.

Internet Operating System

O IOS foi um arcabouço desenvolvido sobre o WWC para adicionar a esta infraestrutura mecanismos de monitoramento e posterior reconfiguração da execução das aplicações [24]. A reconfiguração é efetuada usando duas características dos atores da aplicação: *mobilidade* e *maleabilidade*. O último é a capacidade dos atores modificarem sua granularidade dividindo-se em vários (*split*) ou combinando-se em um só (*merge*).

A arquitetura do IOS é composta por três módulos:

- **Monitoramento:** Armazena informações sobre a execução da aplicação, a serem usadas na tomada de decisões de reconfiguração. Da camada da aplicação, chegam ao módulo de monitoramento informações sobre os perfis de execução dos atores. Isso inclui, por exemplo, dados sobre quantas mensagens cada ator trocou com outros atores. Além disso, esse módulo recebe informações também sobre os recursos físicos do computador, como CPU, memória e rede de comunicação. A arquitetura do IOS permite que as informações sobre os recursos físicos sejam recebidas a partir de sistemas externos de monitoramento.

- **Protocolo:** Esse módulo efetua a comunicação entre as instâncias do IOS rodando nos diferentes nós. Toda a reconfiguração realizada no IOS é uma variação do modelo de *work-stealing*, onde nós com pouca carga tentam "roubar" tarefas (no caso, atores) de nós sobrecarregados. O módulo de protocolo se encarrega de tratar as requisições de roubo de trabalho propagadas na infraestrutura.

- **Decisão:** Ao receber uma requisição de roubo de trabalho, esse módulo decide se e como a reconfiguração será efetuada.

No que diz respeito à tomada de decisões quanto à migração, diferentes estratégias de balanceamento de carga foram testadas no IOS [13, 24]. Ao receber uma requisição de roubo de trabalho de algum nó, o módulo de decisão verifica se algum dos atores rodando localmente é um bom candidato para ser migrado. Esse ator pode ser escolhido aleatoriamente, por exemplo.

No entanto, algumas estratégias mais complexas também foram implementadas. Essas estratégias definem o melhor candidato para migração com base em cálculos que podem levar em conta informações como o nível de uso das máquinas envolvidas, os padrões de troca de mensagens entre os atores ou a expectativa de vida de um determinado ator.

Após a realização de experimentos, os desenvolvedores do IOS concluíram que as estratégias de balanceamento de carga que levavam em conta os padrões de troca de mensagens entre os atores foram as que apresentaram os melhores resultados [13, 14, 24]. Com tais estratégias, ao receber uma solicitação de roubo de trabalho, o IOS procura no nó local algum ator que troque um número grande de mensagens com o nó que originou a solicitação. Dessa forma, atores que realizam intensa comunicação tendem a ser colocados juntos, diminuindo o tráfego de dados na rede.

8.1.2 Relação com nosso trabalho

Nossa trabalho possui diversas semelhanças com o projeto do *World-Wide Computer*. Podemos dizer, inclusive, que o trabalho da equipe do WCL nos ajudou em alguns aspectos a direcionar o desenvolvimento de nossa infraestrutura. O desenvolvimento da classe `Profiler`, por exemplo, foi motivado pelos bons resultados demonstrados no WWC com o uso de estratégias de balanceamento de carga que levem em conta os padrões de comunicação entre os atores. A ideia é que, no futuro, esse `Profiler` seja usado por algoritmos de reconfiguração semelhantes aos implementados no IOS.

No entanto, ainda que em essência as infraestruturas sejam parecidas, existe uma diferença importante entre o WWC e nosso trabalho, relacionada ao ambiente de execução das aplicações. Enquanto o WWC ambiciona rodar sobre um ambiente tão amplo e aberto como a Internet, nosso trabalho foca na execução de aplicações distribuídas em aglomerados mais restritos, cujos computadores provavelmente estão sob uma mesma administração.

Dessa forma, podemos contar com a informação de que todos os nós serão conhecidos *a priori*, e portanto podemos oferecer distribuição automática dos atores para as aplicações. As aplicações em SALSA, por outro lado, devem conter no código-fonte referências explícitas à localização da máquina onde um ator deve ser colocado.

Também por conta disso, o esquema de serviço de nomes implementado no WWC é bastante diferente do nosso. Ali, o nome de um ator, que o identifica unicamente na aplicação, é um URI que contém o endereço da máquina que hospeda o servidor de nomes onde será encontrada a localização atual daquele ator. Assim, ao nomear um ator, o desenvolvedor deve conhecer o endereço do servidor de nomes onde tal ator será registrado. Além disso, o fato dos atores ficarem associados com um servidor específico pode tornar atores inacessíveis no caso de queda de seu servidor de nomes.

Acreditamos que a solução adotada em nosso trabalho é mais interessante, já que o identificador de um ator, seu `uuid`, é completamente independente da localização do servidor de nomes associado a esse ator. Além disso, nossa infraestrutura foi construída de maneira que o serviço de nomes seja configurável, podendo ser trocado por uma implementação que propicie maior desempenho, escalabilidade e tolerância a falhas.

Outra diferença importante entre nossa implementação e a do WWC é a linguagem de programação usada. Por um lado, o fato da linguagem SALSA oferecer suporte nativo a atores apresenta uma vantagem importante: a linguagem pode fazer cumprir certas exigências para que o modelo de atores funcione corretamente. No entanto, ainda que isso seja uma vantagem, com o uso de algumas boas práticas (como visto na Seção 3.2.2), os desenvolvedores podem programar com segurança usando atores em Scala.

Além disso, os atores de SALSA possuem uma limitação importante: ao serem executados na JVM, cada ator é associado a uma *thread* exclusiva. Isso faz com que possam ser criados no máximo alguns milhares de atores em um programa. Em Erlang, onde o modelo de atores encontrou sua implementação mais bem sucedida, os atores são tão leves que seu número pode passar de centenas de milhares, sem grandes problemas, em máquinas com configurações comuns. Nesse aspecto, os atores do Akka estão muito mais próximos dos atores de Erlang. No caso de atores baseados em eventos, o número de atores não está limitado ao número de *threads* da máquina virtual, permitindo uma escalabilidade muito maior à aplicação.

Vale mencionar que os atores do Akka também possuem métodos de coordenação entre atores, assim como SALSA. Enquanto esta última implementa esse coordenação por meio de continuações, os atores do Akka usam os chamados *futuros*. Um futuro pode ser visto como um recipiente para o valor de retorno da chamada, que é devolvido inicialmente vazio. Assim que o ator terminar seu processamento, ele preencherá esse futuro com o valor de retorno adequado. O ator que enviou a mensagem pode, caso deseje efetuar uma sincronização, ficar travado esperando que o futuro seja completado.

Em nossa implementação atual de atores móveis, o suporte ao uso de futuros não está completamente realizado. Caso o ator migre com mensagens associadas a futuros em sua caixa de mensagens, tais futuros nunca serão preenchidos. Ainda que esse suporte incompleto não viole o modelo de atores (atores de Erlang, por exemplo, não possuem nenhum mecanismo de sincronização semelhante), acreditamos que esse tipo de construção é importante para os desenvolvedores, e a extensão completa do suporte a futuros nos atores móveis não deve apresentar grandes dificuldades para futuras versões de nosso sistema.

A linguagem SALSA ainda está sob desenvolvimento: a versão 1.1.5 foi lançada em julho de 2011 e está disponível no sítio do projeto na Internet[2]. Já o middleware de reconfiguração dinâmica IOS parece ter sido descontinuado. Em sua página na Internet[3], consta como última versão lançada a 0.4, datando de 2006.

Quanto a linguagem SALSA, mesmo ainda estando sob desenvolvimento, sua aceitação parece ter ficado restrita a projetos acadêmicos. Ainda que não seja um indicador preciso, vale observar a diferença significativa no número de resultados de buscas no Google para os termos *"salsa programming language"* e *"scala programming language"*: em agosto de 2011, enquanto o primeiro retorna quase 10 mil resultados, o segundo possui mais de 790 mil ocorrências. Acreditamos, portanto, que nossa escolha foi acertada no que diz respeito à linguagem de programação, já que Scala parece estar ganhando cada vez mais popularidade entre os desenvolvedores.

8.2 Stage – Atores móveis em Python

Stage é uma linguagem de programação baseada em atores, com suporte nativo para distribuição e migração de atores [7, 8]. A linguagem, implementada usando a linguagem de programação Python, usa técnicas de metaprogramação para construir uma nova linguagem, com primitivas específicas do modelo de atores, sobre uma linguagem já existente.

Essa abordagem apresenta uma importante vantagem: Stage "herda" a sintaxe e a maioria das funcionalidades de Python. Isso facilita a sua disseminação, já que a aprendizagem é praticamente instantânea por parte daqueles que já conhecem Python. Esses programadores deverão, no entanto, se acostumar com uma nova forma de desenvolver os programas, em particular pela presença do assincronismo.

8.2.1 Arquitetura

A arquitetura de Stage é formada por dois componentes principais: atores móveis e teatros (*theatre*). Os teatros hospedam os atores e se comunicam entre si. Qualquer nó executando um teatro está apto a receber atores móveis.

Para implementar um ator, o desenvolvedor define uma classe que estenda a classe `MobileActor`, nativa da linguagem Stage. Os métodos que forem implementados nessa classe definem os tipos de mensagens que esse ator pode receber, e como ele irá tratá-las. Ou seja, em Stage não existe uma sintaxe especial para enviar mensagens a atores: o envio de uma mensagem corresponde a uma chamada de método na instância do ator. No entanto, essa chamada será assíncrona, sendo processada numa *thread* diferente daquela onde a chamada de método foi efetuada.

As mensagens que podem ser enviadas a um ator, que na realidade correspondem a chamadas de métodos, podem devolver valores, como um método comum. Nesse ponto, os atores de Stage são muito semelhantes aos atores universais do *World-Wide Computer*. Enquanto o WWC permitia o acesso a esses valores de retorno por meio de continuações, os atores de Stage usam o mecanismo de futuros, também existentes nos atores do Akka (porém ainda não totalmente contemplados em nossos atores móveis).

[2] http://wcl.cs.rpi.edu/salsa
[3] http://wcl.cs.rpi.edu/ios

Um ator que envia uma mensagem a outro ator possui três maneiras distintas de efetuar a sincronização com o futuro recebido como resposta:

1. Com o uso da palavra-chave `sync`, o ator remetente fica bloqueado até que um valor de retorno tenha sido computado pelo ator destinatário e associado ao futuro.

2. Com o uso da palavra-chave `ready`, é possível verificar se o futuro já foi preenchido com uma resposta pelo ator destinatário. Isso permite ao ator remetente fazer *polling*, ou seja, consultar de tempos em tempos se a resposta desejada já foi computada.

3. Com o uso da palavra-chave `handle`, é possível registrar uma *callback*, ou seja, um método que deverá ser chamado quando o futuro for computado. Esse método receberá como parâmetro o valor devolvido pelo ator destinatário e poderá efetuar o processamento desejado.

Caso nenhuma das três primitivas de sincronização explícita seja usada, a linguagem implementa um mecanismo de *sincronização preguiçosa* (*lazy synchronisation*). Essa funcionalidade implementa uma sincronização implícita com valores devolvidos por envios de mensagens a atores, sempre que esses valores são de fato necessários. Corresponde, portanto, ao uso explícito de `sync` no momento mais tardio possível.

Migração de atores

Um ator de Stage pode efetuar sua migração com o comando `migrate_to`. Esse comando está disponível apenas para que o próprio ator solicite sua migração. No entanto, é possível que um ator disponibilize um método em sua interface que chame a primitiva de migração. Isso corresponde a definir uma mensagem de migração que pode ser enviada por qualquer outro ator da aplicação. Essa mensagem, no entanto, não terá nenhuma prioridade sobre os outros tipos de mensagens que o ator pode receber.

Os atores móveis devem também implementar o método `arrive`, que será chamado assim que o ator chegar no novo teatro. Esse método define o ponto de onde a execução do ator deve ser retomada após a migração.

A linguagem Stage define ainda o conceito de *atores amigos*. Atores amigos são atores que trocam um grande número de mensagens. Um conjunto de atores amigos forma um *clique de atores*. Na implementação de Stage, quando um ator informa que um outro ator é um ator amigo, os dois atores são colocados no mesmo teatro.

Atores distribuídos

Para permitir a execução de atores num ambiente distribuído usando Stage, basta que teatros sejam iniciados em diferentes nós. O conjunto de nós que executa uma aplicação pode ser modificado dinamicamente, ou seja, nós podem entrar ou sair sem que a aplicação tenha que ser interrompida.

Para encontrar um determinado ator remoto dentro do conjunto de teatros executando a aplicação, a infraestrutura do Stage usa um esquema de "dicas":

1. Um ator A, desejando comunicar-se com um ator B, solicita uma dica de onde o ator B se encontra.

2. Ao receber essa informação, A envia uma mensagem para o teatro que acredita hospedar B.
3. Caso o ator B não esteja lá, o teatro irá recusar a mensagem, informando o ator A.
4. O ator A volta ao passo 1, obtendo uma nova dica sobre a localização de B.

Na implementação original de Stage, essas dicas podem ser obtidas pelos atores apenas de um local, chamado `StageManager`. No entanto, a implementação da linguagem oferece pontos de extensão que permitem adicionar outras fontes de dicas sobre a localização de atores.

Para lidar com a saída de nós durante a execução da aplicação, a linguagem implementa um esquema que tenta migrar todos os atores de um teatro em encerramento, mantendo assim todo o estado da aplicação. Os atores móveis podem implementar o método `theatre_closing`, que define o que fazer caso o teatro que hospeda o ator esteja sendo encerrado: o ator pode migrar para um nó conhecido, ou deixar que o teatro escolha seu destino.

Stage também implementa um mecanismo de atores supervisores, inspirado em Erlang, para lidar com falhas em atores. Caso um ator seja encerrado inesperadamente, seu supervisor será avisado para pode tomar alguma providência. Em Erlang, o ator supervisor pode decidir matar ou reiniciar todos os atores sob sua supervisão. Em Stage, além dessas duas possibilidades, um supervisor pode também migrar seus atores supervisionados.

A linguagem conta ainda com um esquema simples de balanceamento de carga, que é implementado como uma biblioteca. Cada teatro reporta suas medidas de uso de CPU para um nó balanceador, que classifica a carga desse teatro como sendo normal, baixa ou excessiva. Esse nó balanceador dispara então ordens de migração para atores que estão em nós com carga excessiva, solicitando que se mudem para nós com carga baixa.

8.2.2 Relação com nosso trabalho

A linguagem Stage implementa muitos conceitos semelhantes aos de nossa infraestrutura. Uma vantagem na implementação de Stage é a possibilidade de adição ou remoção de nós dinamicamente. Isso, juntamente com o mecanismo de migração, dá grandes possibilidades de reconfiguração para uma aplicação distribuída. No entanto, uma aplicação escrita em Stage deve explicitar no código os endereços dos teatros sendo usados, diferentemente de nossa infraestrutura.

Os *atores amigos* de Stage são muito parecido com os atores co-locados de nosso sistema, ainda que o uso dessa funcionalidade nas duas plataformas seja diferente. Em Stage, atores já existentes são marcados como amigos, de modo a permanecerem juntos. Já os grupos de atores co-locados devem ser definidos no momento da instanciação dos atores. Seria uma modificação interessante em nossa infraestrutura permitir que grupos de atores co-locados sejam formados e/ou modificados com atores já em execução. Com isso, até mesmo um sistema de criação automatizada de grupos poderia ser desenvolvido: as informações coletadas pelo `Profiler` poderiam ser analisadas para formar grupos co-locados com atores que se comunicam intensamente.

Em Stage, o fato de a primitiva de migração poder ser chamada apenas internamente pelo ator pode se tornar um inconveniente para o desenvolvedor. Ainda que os atores móveis possam expor essa primitiva através de uma mensagem, isso obriga que todas as classes que implementem atores móveis definam tal método. Além disso, essa mensagem não possui nenhuma prioridade sobre mensagens comuns, o que pode ser um problema para algoritmos de reconfiguração que desejem

usar o mecanismo de migração para efetuar uma mudança imediata na disposição dos atores dentro do aglomerado de computadores. Os atores de Stage compartilham ainda a mesma limitação dos atores do WWC: cada ator é associado a uma única *thread*. Como já discutido anteriormente, isso pode prejudicar a escalabilidade da aplicação.

A linguagem Stage possui um esquema de descoberta de nomes bastante simples, menos elaborado que o serviço de nomes de nossa infraestrutura. No caso de Stage, um ator pode demorar mais até encontrar a localização correta de um ator destinatário de uma mensagem, aumentando o tráfego da rede. No entanto, Zetter desenvolveu uma segunda versão da linguagem, chamada Stage# [40], onde as principais modificações envolvem justamente a funcionalidade de serviço de nomes. No Stage#, uma tabela de espalhamento distribuída (DHT) foi implementada para tornar essa funcionalidade mais eficiente, robusta e escalável. Seria muito interessante adicionar a nosso trabalho um suporte a DHT nos mesmos moldes do existente em STAGE#.

O esquema de supervisores presente em Stage, inspirado em Erlang, também existe no Akka. Esse mecanismo é bastante útil na construção de aplicações robustas e tolerantes a falhas. Em nossa implementação atual, os supervisores existentes no Akka não foram adaptados para funcionar com atores móveis. A existência de um comportamento específico para tratar o encerramento de teatros, presente em Stage, também é uma proposta interessante de tolerância a falhas, que poderia ser incorporada em futuras versões de nosso trabalho.

Por fim, o balanceamento de carga existente em Stage, ainda que bastante simples, já constitui uma funcionalidade muito importante e inexistente em nossa infraestrutura. Por outro lado, em Stage não existe nenhum suporte nativo para o monitoramento de mensagens trocadas entre atores. O próprio autor da linguagem enfatiza a importância de minimizar a quantidade de informação trafegando pela rede, sendo essa a justificativa para a existência dos *atores amigos*.

8.3 Migração de objetos CORBA

A arquitetura CORBA (*Common Object Request Broker Architecture*)[4] propõe uma maneira de se construir aplicações distribuídas e orientadas a objetos em ambientes heterogêneos. Em CORBA, o cliente e o servidor podem ser executados sobre plataformas diferentes, e até mesmo ser escritos em linguagens diferentes, e ainda assim realizarem a comunicação transparentemente, graças à padronização de interfaces e protocolos definida pela arquitetura.

Um objeto CORBA é um objeto remoto, que o cliente manipula localmente através de um representante local, conhecido como *stub* e gerado automaticamente pela infraestrutura. O objeto que de fato implementará os métodos é executado no servidor, e toda a comunicação remota que associa o *stub* à sua implementação concreta no servidor é feita transparentemente pelo ORB (*Object Request Broker*), um dos componentes principais da arquitetura CORBA.

Domingues [15] propôs uma infraestrutura para realizar a migração de objetos CORBA entre servidores. Ou seja, o que deve ser migrado é o objeto concreto, que responde às chamadas remotas e reside no servidor. Os *stubs*, que ficam no cliente e se comunicam com os objetos no servidor, devem de alguma forma tratar essa mudança de endereço do objeto.

[4] http://www.corba.org

8.3.1 Arquitetura

Apesar de CORBA permitir a utilização de diferentes linguagens, a implementação de Domingues tratou exclusivamente a migração de objetos escritos em Java. Além disso, foram utilizadas algumas funcionalidades oferecidas por CORBA para facilitar o tratamento da migração.

Basicamente, existem dois componentes principais: o objeto móvel e o servidor de mobilidade. Um servidor de mobilidade é capaz de hospedar um ou mais objetos móveis. Dentro de um servidor de mobilidade, o principal componente é o contexto de mobilidade, que é a entidade que administra os objetos móveis e realiza a migração de fato.

Um objeto móvel deve implementar uma interface com quatro métodos. Três deles são *callbacks* a serem chamadas pela infraestrutura durante o ciclo de vida do objeto: `onCreation`, `onDeparture` e `onArrival`. Já o quarto método chama-se `move` e será usado pelos clientes para solicitar a migração daquele objeto.

Já um contexto de mobilidade também deve implementar alguns métodos de uma interface pré-definida. Dentre estes, o método `createMovableObject` é a única maneira de um cliente obter uma referência para um objeto móvel. Já o método `delegateMove` é chamado pelo próprio objeto para solicitar, junto ao contexto de mobilidade onde reside, sua migração para um novo nó.

Contextos de mobilidade

O contexto de mobilidade implementa a parte principal da lógica de migração da infraestrutura. Num contexto de mobilidade, existem duas tabelas:

- **Tabela de Serventes Ativos (TSA):** Guarda todos os objetos móveis atualmente em execução naquele nó. Por meio dessa tabela um contexto de mobilidade consegue saber se, ao receber uma chamada para um determinado objeto, aquele objeto encontra-se de fato ali.
- **Tabela de Objetos Móveis (TOM):** Guarda uma referência para todos os objetos criados naquele contexto de mobilidade. Ou seja, uma entrada é criada nessa tabela apenas quando o cliente chama `createMovableObject` naquele contexto de mobilidade para criar a referência inicial para um objeto móvel. Nessa tabela está guardada, para cada objeto, seu endereço mais atualizado.

A utilização da TOM levanta a necessidade de que a informação sobre o nó onde o objeto foi criado seja sempre mantida. Com isso, surge o conceito de contexto de mobilidade *home*, o contexto de mobilidade inicial de um dado objeto. Essa informação será uma das armazenadas na TSA para cada objeto residente naquele nó.

Processo de migração

Atuam no processo de migração três contextos de mobilidade (CM): o de origem do objeto, o de destino do objeto e o *home*, onde o objeto foi criado inicialmente. De forma simplificada, a migração de um objeto CORBA ocorre da seguinte maneira:

1. O cliente chama o método `move` em um objeto móvel, passando como parâmetro o CM de destino daquela migração.

2. O objeto CORBA solicita, junto ao seu contexto de mobilidade, que seja efetuada a migração. Ele faz isso chamando o método `delegateMove`.

3. O CM de origem efetua a seriação do objeto, e então o envia para o CM de destino. Esse processo é feito chamando-se o método `receiveMovableObject` no CM de destino, e passando a versão seriada do objeto como parâmetro.

4. O CM de destino desseria o objeto, cria uma instância local para ele e adiciona suas informações em sua TSA. Cria então uma referência CORBA para que tal objeto seja acessível remotamente, e devolve essa referência para o CM de origem.

5. O CM de origem usa essa referência para atualizar o CM *home*, por meio do método `updateObjectLocation`. O CM *home* atualizará sua TOM com o novo endereço do objeto.

6. O CM de origem, por fim, retira o objeto recém-migrado de sua TSA.

Atualização de referências

Após a migração de um objeto, o *stub* do cliente continua com o endereço antigo. E, além disso, a associação entre o *stub* e o objeto concreto é feita automaticamente, pelo ORB. Para resolver esse problema, Domingues utilizou um recurso oferecido pela própria arquitetura CORBA.

Quando um cliente faz uma chamada remota de método ao servidor, o servidor pode devolver diferentes tipos de resposta. Dentre estas, uma mensagem do tipo `LOCATION_FORWARD` sinaliza que o cliente deve tentar reenviar sua requisição para um endereço diferente (também fornecido pelo servidor). Ou seja, é uma forma de redirecionar o cliente para o endereço (possivelmente) correto do objeto que ele procura.

O problema de manter as referências válidas após a migração foi resolvido utilizando esse mecanismo. Ao receber uma chamada para um objeto móvel, um contexto de mobilidade executa os seguintes passos:

1. Verifica se aquele objeto se encontra em sua TSA. Em caso positivo, basta chamar o método naquele objeto.

2. Caso o objeto não esteja no TSA, verifica se ele está na TOM. Isso ocorre nos casos em que o objeto foi criado naquele nó, porém já foi migrado para outro. Como a TOM sempre guarda o endereço mais atual do objeto, basta devolver uma mensagem de `LOCATION_FORWARD` passando esse endereço e o cliente será redirecionado corretamente.

3. Caso o objeto não esteja nem na TSA nem na TOM, isso significa que o objeto já passou por aquele nó (que não é seu nó inicial), porém já foi migrado. Nesse caso, o contexto de mobilidade extrai o endereço inicial daquele objeto, informação que todo objeto móvel carrega num campo chamado *object id*. Após isso, ele devolve ao cliente um `LOCATION_FORWARD` contendo esse endereço inicial. Assim, o cliente será redirecionado para o contexto de mobilidade *home* do objeto, caindo então no caso 2.

Um detalhe importante é que todo o mecanismo de redirecionamento via `LOCATION_FORWARD` é feito automaticamente, pelo próprio ORB. Ou seja, para o cliente todo esse procedimento ocorrerá transparentemente.

8.3.2 Relação com nosso trabalho

De maneira geral, os objetivos do trabalho apresentado são bem parecidos com o de nosso projeto. A principal diferença, é claro, está no contexto envolvido: enquanto nossa infraestrutura realiza a migração de atores, o trabalho exposto realiza a migração de objetos CORBA.

Por outro lado, muitos conceitos centrais para ambas as infraestruturas são os mesmos. Em ambos os casos, temos entidades (objetos ou atores) que são separadas em referência e implementação de fato. No caso dos objetos CORBA, as implementações tinham que migrar enquanto os *stubs* continuassem válidos. Já no caso dos atores, o que o cliente manipula também são referências, que possuem o endereço real do ator.

A principal diferença é que, nesse caso, a divisão cliente/servidor não é tão clara, já que essa referência pode ser para um ator local, e portanto a comunicação será feita diretamente. De toda forma, existirá sempre uma forma de "indireção" no acesso ao ator, assim como nos objetos CORBA.

Portanto, um problema central para ambas as infraestruturas é como realizar a atualização dessas referências após uma migração. Quando um objeto ou ator migra, as referências para ele espalhadas pelo sistema devem, de alguma forma, permanecer válidas. No caso dos objetos CORBA, esse problema foi resolvido utilizando um recurso da própria arquitetura, o mecanismo de `LOCATION_FORWARD`.

Já no caso de nosso sistema, tivemos que implementar completamente uma solução para a atualização das referências. Assim como no trabalho de Domingues, as referências a atores só são atualizadas quando o envio de uma mensagem falha. Nosso sistema possui uma vantagem importante: essa atualização é feita com base em um serviço de nomes distribuído, enquanto no caso dos objetos CORBA, a localização atual do objeto é buscada em seu contexto de mobilidade *home*. Caso essa máquina esteja inacessível, será impossível encontrar a localização atual do ator, ainda que ele esteja hospedado numa máquina acessível.

Ainda que nosso serviço de nomes não implemente tolerância a falhas ou redundância, é perfeitamente possível utilizar uma implementação mais robusta de serviço de nomes em nossa infraestrutura. Além disso, nosso mecanismo de atualização de referências exige que um número menor de mensagens sejam enviadas para ser efetuado. Por fim, vale ressaltar que o uso de um serviço de nomes já foi apontado como uma solução mais eficiente, em comparação com a abordagem existente em CORBA, para o problema de associar um representante a um objeto remoto [20].

8.4 Akka 2.0 (*Cloudy Akka*)

Quando começamos o desenvolvimento deste trabalho, estudamos algumas tecnologias que pudessem facilitar a implementação da infraestrutura que tínhamos em mente. Após algum tempo de pesquisa, verificamos que a plataforma Akka seria a base ideal para nosso projeto. O Akka já possuía uma implementação sólida do modelo de atores e era por natureza voltado ao desenvolvimento de aplicações distribuídas.

Por conta disso, acreditávamos que não só o Akka poderia ser de grande ajuda no desenvolvimento de nosso projeto, como também o resultado deste trabalho poderia se tornar uma contribuição à plataforma, talvez até mesmo se integrando à distribuição oficial do Akka. Assim, em meados de 2010, enviamos uma mensagem à lista de emails do projeto Akka, apresentando nossa ideia aos desenvolvedores do projeto.

A resposta recebida, do líder do projeto Akka, trazia um fato inesperado: ele estava desenvolvendo essencialmente as mesmas funcionalidades, porém para a versão paga e fechada do Akka (tal projeto viria a ser chamado *Cloudy Akka*). Isso nos deu, por um lado, a confiança de que havia utilidade real na infraestrutura que planejávamos, já que aparentemente havia interesse comercial em algo semelhante. Porém, por outro lado, isso impediria que nosso desenvolvimento se integrasse à distribuição do Akka, ao menos na versão de código aberto.

Resolvemos então desenvolver nossa infraestrutura de maneira totalmente isolada, sem interferência ou colaboração da comunidade Akka. Como dito no Capítulo 5, nossa infraestrutura baseou-se na versão 0.10 do projeto. Porém, tentamos manter nossas modificações o menos intrusivas possíveis, para que nosso módulo pudesse se integrar com versões futuras do Akka sem grandes dificuldades (ainda que, até a versão 1.0, o código do Akka ainda sofreu algumas mudanças razoavelmente profundas).

Em maio de 2011, no entanto, esse cenário mudou, quando foi anunciado que boa parte das funcionalidades do *Cloudy Akka* (em particular aquelas comuns à nossa infraestrutura) seriam integradas à versão aberta do Akka, a partir de sua versão 2.0, cujo lançamento está previsto para o final de 2011.

A seguir, daremos uma visão geral sobre as principais funcionalidades que estarão presentes nessa versão, e suas semelhanças e diferenças com aquelas presentes em nossa infraestrutura. Como a documentação das novas funcionalidades disponíveis na versão 2.0 do Akka ainda é escassa, nos basearemos em algumas descrições feitas pelos desenvolvedores do projeto sobre a nova versão, além do exame de seu código fonte, disponível no repositório oficial do Akka[5] e ainda sob ativo desenvolvimento.

8.4.1 Suporte a aglomerados

O que promete ser o principal diferencial na versão 2.0 do Akka é o chamado *suporte a aglomerados*, até então presente apenas na versão paga do sistema. Com esse suporte, a configuração de como os atores de uma aplicação se distribuem num aglomerado pode ser feita completamente via arquivo de configuração, sem a necessidade de alterações no código da aplicação.

A instanciação de atores no Akka 2.0 ainda é feita usando o método `actorOf`, assim como nas versões anteriores (como descrito na Seção 5.2.2). No entanto, para usar as novas funcionalidades do suporte a aglomerados, é necessário que a obtenção da referência ao ator seja feita especificando-se um *endereço virtual*, como no exemplo:

```
val actor = actorOf[MyActor]("my-service")
```

Nesse caso, o endereço virtual `my-service` pode ser usado para, no arquivo de configuração, definir como os atores associados a ele serão distribuídos no aglomerado, replicados, etc. Caso não exista uma configuração específica para tal endereço virtual, o ator simplesmente será instanciado localmente.

Para um dado endereço virtual, pode-se configurar, por exemplo, o número de réplicas de um determinado ator. Esse número define quantos atores de fato deverão ser instanciados, no aglomerado, para representar aquele serviço. A escolha de qual nó irá hospedar cada réplica do ator é

[5] https://github.com/jboner/akka/

feita aleatoriamente. Juntamente com essa configuração, é possível especificar qual tipo de balanceamento de carga deve ser feito entre as diferentes réplicas de um determinado ator. É possível definir que as mensagens sejam enviadas a cada réplica num esquema de rodízio (*round-robin*), por exemplo. Ou a escolha da réplica que deverá processar uma mensagem pode ser feita com base nos recursos físicos (CPU, memória) dos computadores que hospedam as réplicas.

Ao obter uma referência passando um endereço virtual que possua esse tipo de configuração, a referência devolvida será um representante (*proxy*) para o ator que saberá rotear as mensagens enviadas a ele de acordo com a política de balanceamento de carga definida no arquivo de configuração.

Além da replicação de atores para fins de balanceamento de carga, o suporte a aglomerados do Akka 2.0 também cuida da replicação de um ator no caso em que o nó que o hospeda sofra uma queda. O caso em que o ator não possui estado é mais simples: o ator apenas é reiniciado num novo nó, e as referências são corrigidas automaticamente. No caso de um ator configurado para ter N réplicas, caso o computador que executa alguma das réplicas caía, uma nova réplica será criada em outro nó, de maneira a manter o número de instâncias daquele ator.

O caso da replicação de atores com estado é mais complexo, já que o ator deve ser reiniciado mantendo o estado que ele tinha anteriormente. A infraestrutura do Akka provê dois mecanismos para resolver esse problema. No primeiro deles, cada mensagem que o ator recebe é registrada num *log* transacional. No caso de uma falha, o ator é recriado num nó diferente com o estado inicial, e todas as mensagens contidas no *log* são reenviadas para o ator. Para lidar com o problema do *log* crescer indefinidamente após longos períodos de execução, a infraestrutura permite que o estado do ator seja gravado de tempos em tempos, de forma que somente as mensagens registradas após a última gravação do estado do ator tenham que ser reprocessadas.

A outra maneira de preservar o estado de atores após uma falha é usando um dispositivo de armazenamento persistente externo. Nesse caso, a cada N mensagens recebidas o estado do ator é armazenado num dispositivo externo, como por exemplo um banco de dados. No caso de uma falha, o ator é recriado num novo nó a partir do último estado armazenado. O valor de N pode ser definido como 1, tendo-se assim a garantia que nenhuma mudança de estado do ator será perdida em caso de falhas no sistema.

A coordenação dos nós rodando no aglomerado é feita usando uma infraestrutura já existente, chamada ZooKeeper[6]. Vale destacar que a infraestrutura do Akka 2.0 permite a entrada e saída de nós do aglomerado em tempo de execução, além de contar com a possibilidade de registrar-se ouvintes para receberem atualizações sobre eventos como esses.

8.4.2 Relação com nosso trabalho

Uma das principais funcionalidades do suporte a aglomerados do Akka 2.0, senão a principal, é a separação entre a instanciação de atores e sua configuração de implantação (*deployment*). Com isso, a informação de onde cada ator será executado é retirada do código e colocada num arquivo de configuração. A aplicação torna-se então capaz de se adaptar automaticamente a mudanças no aglomerado onde é executada.

A ideia de facilitar o desenvolvimento de aplicações distribuídas delegando para a infraestrutura a escolha do nó que hospedará cada ator é exatamente a mesma em nosso trabalho. A implementação

[6] http://zookeeper.apache.org

difere, no entanto, já que em nossa infraestrutura cada ator possui exatamente uma implementação rodando em algum nó. Inexiste, portanto, o conceito de várias réplicas para um ator.

No que diz respeito à técnicas de balanceamento de carga, parte dela é feita no Akka 2.0 por meio de decisões de roteamento de mensagens para as diferentes réplicas de um ator. Esse comportamento não existe em nosso sistema, até porque não faria sentido sendo que as mensagens para um ator só podem ser encaminhadas para o único nó que o hospeda.

Já quanto ao balanceamento de carga feito com os atores propriamente ditos, movendo atores em busca de uma configuração que melhore o desempenho da aplicação, as descrições feitas até agora do Akka 2.0 não deixam claro se esse tipo de funcionalidade será contemplada. Além disso, não parece existir uma API para expor informações sobre a execução dos atores para que sistemas externos coordenem o rebalanceamento do aglomerado, de maneira semelhante à nossa implementação. Acreditamos que esse tipo de API é importante para permitir o desenvolvimento de diferentes mecanismos de balanceamento de carga, adequados a diferentes cenários.

A migração de atores no Akka 2.0 é voltada principalmente para a tolerância a falhas. Os atores administrados pela infraestrutura serão automaticamente migrados quando ocorrer um problema na máquina em que estão hospedados, como explicado na seção anterior. No entanto, a descrição do sistema prevê um método `cluster.migrate()`, que permitiria requisitar explicitamente a migração de um ator, especificando os nós de origem e destino (lembrando que um mesmo ator pode ter réplicas instanciadas em diferentes nós). Esse método ainda não aparece na versão do código disponibilizada no repositório do projeto, e portanto não pudemos avaliar as semelhanças de sua implementação com o mecanismo de migração em nossa infraestrutura.

Diferentemente do que existe em nosso sistema e em Stage, não parece existir um suporte no Akka 2.0 para algo como atores co-locados (ao menos nada do tipo foi descrito na documentação divulgada até então). Também não ficou claro se haverá a possibilidade de especificar em qual nó um ator deverá ser instanciado (ou em quais nós, no caso de mais de uma réplica). Ainda que a distribuição automática de atores em um aglomerado seja uma das bases da nossa infraestrutura, acreditamos ser de extrema importância dar a liberdade ao programador de especificar a localização exata para um determinado ator.

De maneira geral, o *Cloudy Akka*, agora sendo incorporado na versão 2.0 do Akka, contempla conceitos muito semelhantes aos de nossa infraestrutura. As duas implementações, no entanto, tomaram rumos razoavelmente diferentes. A ideia do Akka 2.0 de tratar atores como *serviços*, que podem ser replicados e terem as chamadas a eles balanceadas, não existe em nossa infraestrutura. Sem dúvida seria interessante estudar uma funcionalidade semelhante para versões futuras de nosso projeto.

Capítulo 9

Conclusões

Este trabalho teve como objetivo principal desenvolver uma plataforma que auxilie os desenvolvedores na construção de aplicações distribuídas usando duas ferramentas principais: atores e a linguagem Scala. Acreditamos que o produto final do trabalho cumpre essa tarefa. Ainda que nosso sistema ainda esteja numa fase inicial e comporte muitas melhorias, ele já representa um importante passo no sentido de desenvolver uma infraestrutura que é, ao nosso ver, extremamente útil.

Mais do que isso, cremos ser justo dizer que nosso trabalho é bastante contemporâneo no que diz respeito às tecnologias utilizadas. Ainda que o futuro das linguagens de programação seja desconhecido, a linguagem Scala cada vez mais ganha atenção por parte dos desenvolvedores. Já o modelo de atores, criado no ambiente acadêmico há mais de 30 anos, experimenta nos últimos anos um ressurgimento. Isso se deve principalmente à necessidade cada vez mais real de se enfrentar o desafio de desenvolver aplicações concorrentes.

O código de nossa infraestrutura pode ser obtido a partir do endereço https://github.com/tcoraini/akka-mobile-actors. A seguir, concluiremos este trabalho com algumas considerações finais e propostas de trabalhos futuros em nosso sistema.

9.1 Principais contribuições

Até onde sabemos, nossa infraestrutura é hoje o primeiro sistema disponível como software livre a implementar em conjunto as funcionalidades de migração e distribuição automática de atores. O conceito de atores móveis é mais comum, e inclusive uma das inspirações deste trabalho veio do WWC (Seção 8.1), que implementa o mecanismo de migração para seus atores. No que diz respeito à Scala, porém, não temos conhecimento de nenhuma outra implementação aberta de atores que contemple o mecanismo de migração.

Todas as infraestruturas que estudamos, ainda que possuíssem atores móveis, ainda dependendiam de referências explícitas no código a respeito da localização de cada ator no aglomerado. Portanto, acreditamos que nosso trabalho apresenta uma contribuição importante com a distribuição automática de atores, usando o conceito de transparência de localização ao máximo para promover a escalabilidade das aplicações.

Como vimos na Seção 8.4, hoje já existe uma infraestrutura que implementa as funcionalidades de migração e distribuição automática de atores: o *Cloudy Akka*. O início de seu desenvolvimento foi bastante próximo do começo deste trabalho, porém decidimos seguir o rumo previamente estabelecido também por acreditarmos que nossa infraestrutura tinha a vantagem de ser totalmente

livre. Isso mudará, no entanto, nos próximos meses, quando tais funcionalidades serão incorporadas à versão 2.0 do Akka, também livre.

Todavia, em nenhum momento encaramos nosso projeto como um concorrente do *Cloudy Akka*. Esse tipo de competição nem seria muito viável para nós, já que o que desenvolvemos foi um projeto puramente acadêmico, enquanto o *Cloudy Akka* conta com uma equipe de desenvolvedores dedicados em tempo integral que tinham no projeto sua fonte de renda.

Naturalmente que, ao ser a única opção de código aberta a implementar tais conceitos, nosso sistema tinha mais chances de receber atenção e contribuições de outros desenvolvedores ao redor do mundo. Por outro lado, a presença do *Cloudy Akka* (agora Akka 2.0) nos deu a confiança de que a solução que buscamos é bastante atual e desperta interesse real por parte da indústria.

Além disso, acreditamos também que este trabalho contribui para ajudar a trazer para o meio acadêmico certas tendências que parecem estar sendo consolidadas na indústria. A linguagem Scala, por exemplo, nasceu no ambiente acadêmico, na École Polytechnique Fédérale de Lausanne, na Suíça. No entanto, o que vemos atualmente é que tal linguagem está sendo mais difundida no mercado do que na universidade.

Ainda que a linguagem ainda seja nova e muitos ainda desconfiem de sua real capacidade de se firmar na indústria, são cada vez mais comuns os relatos de uso de Scala em aplicações reais. Por outro lado, até onde é de nosso conhecimento, poucos trabalhos acadêmicos foram realizados utilizando a linguagem Scala de alguma maneira, como é o caso do nosso.

O mesmo pode ser dito para o modelo de atores, que também surgiu a partir de pesquisas acadêmicas na década de 1970. Por muito tempo esse modelo não obteve amplo destaque, porém nos últimos anos ele ganhou novo fôlego, impulsionado pela crescente onda de interesse em torno da computação concorrente. Também nesse caso, o que vemos é que esse aumento na popularidade do modelo de atores partiu das necessidades impostas pelo mercado. Todavia, o meio acadêmico ainda não parece ter retomado o interesse e as pesquisas em torno desse modelo que tem potencial para se tornar uma importante ferramenta na construção de aplicações concorrentes nos próximos anos.

Assim, acreditamos que essa é também uma importante contribuição de nosso trabalho, ao buscar assuntos bastante contemporâneos e cada vez mais falados no mercado, e trazê-los para dentro da universidade. Não é incomum vermos tecnologias surgirem e em pouco tempo causarem grande impacto em áreas da computação por conta de demandas da indústria. Nesses casos, muitas vezes o meio acadêmico acaba ficando "atrasado", voltando seus olhares para tal tecnologia apenas algum tempo depois de ela ter sido estabelecida e estudada por profissionais do mercado. Acreditamos que é importantíssimo que exista grande sinergia entre a indústria e a universidade, e esperamos que este trabalho possa ter contribuído um pouco com isso.

9.2 Trabalhos futuros

A infraestrutura desenvolvida neste trabalho é plenamente funcional, implementando com sucesso todas as funcionalidades descritas ao longo deste texto. No entanto, certamente diversas melhorias e extensões ainda podem ser feitas no sistema. Ao longo do texto algumas dessas melhorias foram mencionadas. Nesta seção resumiremos aquelas que consideramos as mais importantes para a infraestrutura.

Dividiremos essa seção em duas partes. Inicialmente trataremos de algumas melhorias que seriam interessantes na infraestrutura já existente. Em seguida, discutiremos algumas ideias de outros sistemas que podem ser construídos sobre nossa plataforma.

9.2.1 Melhorias na infraestrutura existente

Modificações na topologia do aglomerado em tempo de execução

A versão atual da infraestrutura usa um arquivo de configuração para descrever os nós que fazem parte do aglomerado que irá executar a aplicação. Para o correto funcionamento, é necessário que exatamente o mesmo arquivo de configuração seja colocado em todas as máquinas que rodarão um teatro.

Desde o começo direcionamos o desenvolvimento de nossa plataforma tendo em mente sua execução num ambiente onde os computadores que formam o aglomerado são conhecidos de antemão. Assim, acreditamos que o uso de um arquivo de configuração da maneira que adotamos é condizente com essas condições.

Todavia, para um ambiente que procura promover a escalabilidade e a adaptabilidade das aplicações, certamente seria um grande avanço se permitíssemos que computadores entrassem e saíssem do aglomerado em tempo de execução, sem que a aplicação tivesse que ser interrompida. Na versão atual do sistema, uma alteração desse tipo demanda que a execução seja interrompida, os arquivos de configuração modificados (para refletir a nova topologia do aglomerado) e então a aplicação reiniciada.

Esse tipo de modificação, hoje, não é trivial. Em nosso serviço de nomes distribuído há uma função de espalhamento que usa a lista de nós obtida do arquivo de configuração para decidir onde será armazenado o endereço de um determinado ator. A hipótese é que essa lista não será modificada durante a execução da aplicação. Porém, tanto o serviço de nomes quanto essa função de espalhamento podem ser substituídas via arquivo de configuração, um ponto de extensão implementado justamente para facilitar a evolução da infraestrutura.

Uma mudança ainda mais profunda, mas que merece ser avaliada, é a de delegar todo o gerenciamento dos nós do aglomerado para algum sistema externo já existente. Como vimos na Seção 8.4, é justamente isso que faz o Akka 2.0. Esse sistema usa internamente o ZooKeeper, um sistema que implementa diversos mecanismos de coordenação de aplicações distribuídas, entre eles um serviço de nomes. Sem dúvida pode ser uma alternativa interessante usar algo semelhante em nossa infraestrutura, possivelmente o próprio ZooKeeper, para efetuar a coordenação dos nós do aglomerado, e consequentemente permitir a entrada e a saída de nós sem a necessidade de interromper a execução da aplicação.

Extensão do suporte às funcionalidades dos atores do Akka

No Capítulo 8, ao compararmos nosso trabalho ao WWC e ao Stage, mencionamos duas funcionalidades que existem nos atores do Akka, mas que não foram completamente implementadas nos atores móveis de nossa infraestrutura.

A primeira destas funcionalidades permite obter um valor de resposta a partir de um envio de mensagem a um ator, de maneira análoga ao valor de retorno recebido após a chamada de um

método. Naturalmente esse sincronismo precisa ser simulado de alguma maneira, já que o envio de mensagens a um ator é assíncrono e retorna o controle imediatamente ao remetente.

Na versão 0.10 do Akka (usada neste trabalho), existiam os métodos `!!` e `!!!` para realizar o envio de mensagens com a possibilidade de sincronização com a resposta. Ambos usam, internamente, o conceito de *futuro*, ou seja, um recipiente para o valor de retorno do processamento da mensagem. Esse futuro é devolvido ao remetente imediatamente, porém vazio, e será preenchido tão logo o ator destinatário processe a mensagem e compute seu resultado. No caso de `!!`, o ator remetente fica travado esperando o futuro ser preenchido com uma resposta, e então devolve esse valor. Trata-se, portanto, de uma espécie de envio síncrono de mensagem. Já o método `!!!` devolve explicitamente o futuro ao remetente, cabendo a ele aguardar seu preenchimento quando achar oportuno.

Nas versões mais atuais do Akka essa sintaxe mudou, porém o mecanismo de sincronização continua baseado em futuros. Os atores móveis de nossa infraestrutura, por serem extensões dos atores do Akka, possuem os métodos `!!` e `!!!` e contemplam esse tipo de envio de mensagem. Porém, o suporte não é completo: mensagens envolvendo futuros enviadas a um ator durante sua migração não são tratadas corretamente. Atualmente, o mecanismo de retenção e posterior encaminhamento de mensagens funciona apenas com o envio assíncrono de mensagens (método `!`).

Ainda que esse tipo de envio de mensagens vá, de certa forma, contra o modelo teórico de atores (que não previa nada desse tipo), acreditamos que essa é uma funcionalidade importante. Em particular, desenvolvedores acostumados a programar sequencialmente podem ter dificuldades para construir programas totalmente baseados no envio de mensagens assíncronas, ainda que esse paradigma favoreça a concorrência. Dessa forma, primitivas como `!!` e `!!!` são importantes para facilitar a transição de desenvolvedores vindos do "mundo de Java" para o "mundo de atores".

A segunda funcionalidade não totalmente implementada pelos atores móveis envolve a tolerância a falhas nas aplicações. Os atores do Akka implementam um esquema de supervisores bastante semelhante ao existente em Erlang, sendo este um dos mecanismos que conferem às aplicações distribuídas escritas nessa linguagem tamanha robustez e confiabilidade.

De maneira simplificada, o mecanismo de supervisores permite que sejam criadas ligações (*links*) entre pares de atores, sendo que em cada par um dos atores será o supervisor da ligação. Caso ocorra um problema com um ator supervisionado, seu supervisor será imediatamente notificado. Usando esse recurso é possível criar uma árvore de supervisores, em que atores podem desempenhar simultaneamente o papel de supervisor e o de supervisionado. Falhas em determinados pontos dessa árvore são detectados e tratados localmente.

Nossa implementação não contemplou o suporte ao mecanismo de supervisores. Assim, ligações feitas entre atores não são mantidas após uma migração. A extensão dos atores móveis para que eles ofereçam o suporte ao esquema de supervisão é, sem dúvida, um importante trabalho futuro.

Melhorias no componente de monitoramento

O componente de monitoramento (ou *profiler*) é aquele responsável por registrar informações sobre as trocas de mensagens entre os atores. Sua finalidade é detectar padrões de trocas de mensagens, identificando atores hospedados em computadores diferentes que realizam intensa troca de mensagens e, portanto, podem ser bons candidatos a migrarem para aumentar a eficiência da aplicação.

Durante o desenvolvimento desse módulo, identificamos alguns pontos de melhoria que poderiam

vir a ser contemplados em futuras versões da infraestrutura. Os principais deles são:

- **Limpeza contínua de dados antigos:** A versão atual do *profiler* permite realizar a limpeza periódica das informações obtidas com o monitoramento. Essa limpeza pode tanto ser feita explicitamente pelo usuário, chamando um método, como programada para acontecer periodicamente em intervalos de tempo pré-definidos.

  No entanto, sempre que tal limpeza ocorrer, todos os dados armazenados serão apagados. Uma estratégia mais inteligente apagaria continuamente as informações conforme elas ultrapassassem um "período de validade", porém mantendo sempre os dados obtidos mais recentemente. Ou seja, teríamos algo como uma "janela de tempo deslizante", que definiria a cada instante os dados válidos e apagaria aqueles fora da janela. Esse mecanismo permitiria retratar os padrões de comunicação de maneira muito mais fiel. No entanto, certamente que sua implementação seria bem mais complexa do que a utilizada atualmente. Justamente por isso tal funcionalidade foi deixada para uma versão futura, porém sua importância é inegável.

- **Monitoramento de atores co-locados:** Atualmente, o monitoramento não faz distinção entre atores comuns e atores fazendo parte de um grupo co-locado. No entanto, a decisão de migrar um grupo de atores co-locados não deve, em teoria, se basear no comportamento de um único ator do grupo.

  Portanto, seria interessante que o *profiler* usasse, de alguma maneira, a informação de que certos atores fazem parte de grupos de atores co-locados. Uma possibilidade seria contabilizar de maneira conjunta as mensagens enviadas a quaisquer atores de um grupo co-locado. Ou seja, para atores co-locados o módulo de monitoramento não guardaria quantas mensagens o ator, individualmente, recebeu. Ele guardaria, para um determinado grupo, quantas mensagens todo o grupo recebeu de atores de outros nós. Isso influenciaria na decisão de migrar todo o grupo. Talvez seja interessante, no entanto, continuar monitorando também de modo individual os atores integrantes de grupos, já que sempre existe a possibilidade de migrá-los sozinhos, removendo-os do grupo do qual fazem parte.

9.2.2 Aplicações baseadas na infraestrutura de atores móveis

Neste trabalho, não desenvolvemos uma aplicação final, a ser usada para resolver um problema específico. Ao invés disso, criamos uma infraestrutura genérica que deve servir de base para a implementação de uma série de aplicações dos mais diferentes tipos. O desenvolvimento dessas aplicações sobre nossa plataforma certamente serviria para engrandecer e validar nosso trabalho, colocando em uso tudo aquilo que desenvolvemos.

Podemos dividir essas aplicações em dois tipos. Primeiramente, existem outras infraestruturas, também genéricas, que estenderiam os recursos de nossa infraestrutura, oferecendo serviços adicionais a serem explorados por outras aplicações. O desenvolvimento deste trabalho manteve sempre em foco um tipo particular de extensão: o uso de algoritmos de reconfiguração dinâmica para aumentar a eficiência da aplicação.

O recurso de distribuição automática de atores já oferece uma oportunidade importante de balanceamento de carga: definir a localização inicial de cada ator, de maneira transparente, levando em conta dados relevantes sobre o estado atual do aglomerado, como por exemplo quais computadores possuem mais recursos livres.

Além disso, algoritmos de reconfiguração poderiam usar o mecanismo de migração de atores para mudar a distribuição destes durante a execução da aplicação, de maneira totalmente transparente e sem exigir qualquer tipo de medida por parte do desenvolvedor da aplicação. Nosso código já está preparado para oferecer dados sobre a troca de mensagens entre os atores, já que a comunicação remota via rede pode ser um dos maiores gargalos de uma aplicação. Portanto, algoritmos de reconfiguração dinâmica que procurem minimizar a troca remota de mensagens já possuem, por parte de nosso sistema, um suporte embutido para facilitar sua tarefa.

Porém, mesmo sem a extensão das funcionalidades de nossa infraestrutura para contemplar outros recursos como a reconfiguração dinâmica de aplicações, acreditamos que aplicações finais já podem tirar proveito dos mecanismos oferecidos na versão atual de nosso sistema. Em particular, a distribuição automática pode tornar uma aplicação escalável sem grandes dificuldades por parte do desenvolvedor, desde que ela seja baseada em atores.

A plataforma S4[1] oferece uma boa sugestão de aplicações que podem se beneficiar de infraestruturas como a nossa. Desenvolvida pela empresa Yahoo, essa plataforma é voltada ao desenvolvimento de aplicações que fazem o processamento de fluxos de dados contínuos e ilimitados, muito comum em aplicações de Internet de uso massivo. O S4 é inspirado no modelo MapReduce [12], desenvolvido pela Google. Todavia, enquanto o MapReduce é focado no processamento de lotes de dados em segundo plano, o S4 opera sobre um fluxo de eventos que entra no sistema de maneira contínua (a uma taxa variável), devendo emitir os resultados também continuamente e, de preferência, na mesma taxa de entrada dos dados.

A arquitetura do S4 é fortemente baseada no modelo de atores. Segundo os próprios autores, o encapsulamento dos dados e a transparência de localização permitem uma distribuição massiva. Na plataforma, o análogo aos atores são as *unidades de processamento*, que são espalhadas por um aglomerado e processam os *eventos* que trafegam pelo sistema, como um fluxo de dados. Ao processar um evento, uma unidade de processamento pode gerar novos eventos, que são automaticamente despachados para a unidade de processamento responsável por seu processamento.

Aplicações que realizem esse tipo de processamento podem ser facilmente implementadas em nossa infraestrutura. As unidades de processamento seriam implementadas por atores e, com isso, poderiam ser distribuídas automaticamente no aglomerado, promovendo a escalabilidade da aplicação. Os eventos fluiriam como trocas de mensagens normais e o serviço de nomes distribuído se encarregaria de encaminhá-los à unidade de processamento adequada.

Além disso, a migração poderia ser usada para levar tal plataforma a um nível ainda mais avançado, permitindo a redistribuição das unidades de processamento, buscando aumentar a eficiência da aplicação e, portanto, a sua vazão. Os atores co-locados também poderiam ser usados caso a semântica da aplicação permitisse identificar conjuntos de unidades de processamento que, em geral, participassem do mesmo fluxo de processamento de eventos.

Os autores do S4 afirmam: "o sistema atual carece de balanceamento de carga dinâmico e migração de unidades de processamento, mas pretendemos incorporar essas funcionalidades" [27]. Isso mostra que os autores estão de acordo com nossa opinião a respeito da importância desse tipo de mecanismo para infraestruturas voltadas à computação distribuída.

Dessa forma, cremos que seria bastante proveitoso o desenvolvimento de uma aplicação nos moldes das previstas para o S4, que façam o processamento de fluxos contínuos de dados. Como

[1] http://s4.io/

os próprios autores do S4 identificaram, o modelo de atores funciona muito bem para tal tipo de cenário. Além disso, tal aplicação poderia fazer uso de praticamente todas as funcionalidades de nossa infraestrutura. A implementação de uma aplicação desse tipo, para rodar sobre nosso sistema, é um próximo passo de extrema importância para dar continuidade a este trabalho.

Referências Bibliográficas

[1] Java generics. http://download.oracle.com/javase/tutorial/java/generics/index.html. Último acesso em 05/07/2011.

[2] The scala 2.8 actors api. http://www.scala-lang.org/docu/files/actors-api/actors_api_guide.html. Último acesso em 23/07/2011.

[3] G. Agha. An overview of actor languages. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 58–67. ACM, 1986.

[4] Gul A. Agha. Actors: a model of concurrent computation in distributed systems. *AITR-844*, 1985.

[5] Gul A. Agha and Carlos A. Varela. Worldwide computing middleware. In M. Singh, editor, *Practical Handbook on Internet Computing*. CRC Press, 2004. invited book chapter.

[6] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[7] John Ayres. Implementing stage: the actor based language. Master's thesis, Imperial College London, June 2007 .

[8] J.W. Ayres and Susan Eisenbach. Stage: Python with Actors. In *International Workshop on Multicore Software Engineering (IWMSE)*, May 2009. URL http://pubs.doc.ic.ac.uk/actors-in-python/.

[9] A. Carzaniga, G.P. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th international conference on Software engineering*, pages 22–32. ACM, 1997.

[10] A. Carzaniga, G.P. Picco, and G. Vigna. Is code still moving around? looking back at a decade of code mobility. 2007.

[11] David M. Chess, Colin G. Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea? In *Mobile Object Systems - Towards the Programmable Internet*, pages 25–45. Springer-Verlag, 1997. ISBN 3-540-62852-5.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Travis Desell. Autonomic grid computing using malleability and migration: An actor-oriented software framework. Master's thesis, Rensselaer Polytechnic Institute, May 2007.

[14] Travis Desell, Kaoutar El Maghraoui, and Carlos A. Varela. Load balancing of autonomous actors over dynamic networks. In *Proceedings of the Hawaii International Conference on System Sciences, HICSS-37 Software Technology Track*, pages 1–10, January 2004. URL http://doi.ieeecomputersociety.org/10.1109/HICSS.2004.10046.

[15] Helves H. Domingues. Uma infraestrutura para migração de objetos CORBA implementados em Java. Master's thesis, Universidade de São Paulo, December 2001.

[16] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding code mobility. *Software Engineering, IEEE Transactions on*, 24(5):342–361, 1998.

[17] Philipp Haller. An object-oriented programming model for event-based actors. Master's thesis, Karlsruhe University, May 2006.

[18] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In David E. Lightfoot and Clemens A. Szyperski, editors, *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006, Proceedings*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006. ISBN 3-540-40927-0. URL http://dx.doi.org/10.1007/11860990_2.

[19] Philipp Haller and Martin Odersky. Actors that unify threads and events. In Amy L. Murphy and Jan Vitek, editors, *Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume 4467 of *Lecture Notes in Computer Science*, pages 171–190. Springer, 2007. ISBN 978-3-540-72793-4. URL http://dx.doi.org/10.1007/978-3-540-72794-1_10.

[20] Michi Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8 (1):66–75, 2004.

[21] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[22] D. Kotz and R.S. Gray. Mobile agents and the future of the internet. *Operating Systems Review*, 33(3):7–13, 1999.

[23] D.B. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.

[24] Kaoutar El Maghraoui. *A Framework for the Dynamic Reconfiguration of Scientific Applications in Grid Environments*. PhD thesis, Rensselaer Polytechnic Institute, 2007.

[25] Kaoutar El Maghraoui, Travis Desell, Boleslaw K. Szymanski, and Carlos A. Varela. The Internet Operating System: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications (IJHPCA), Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms*, 20(4):467–480, 2006.

[26] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[27] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177. IEEE, 2010.

[28] Martin Odersky. The Scala language specification, Version 2.7, Draft, 2009.

[29] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, École Polytechnique Fédérale de Lausanne, 2006.

[30] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, Mountain View, CA, 2007.

[31] Scala Internals Mailing List. Time to take a long hard look at actor design and implementation. http://thread.gmane.org/gmane.comp.lang.scala.internals/453, . Último acesso em 20/07/2010.

[32] Scala Internals Mailing List. New lift actor code. http://thread.gmane.org/gmane.comp.lang.scala.internals/517, . Último acesso em 20/07/2010.

[33] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[34] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[35] Carlos A. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, U. of Illinois at Urbana-Champaign, 2001.

[36] Carlos A. Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36 (12):20–34, December 2001.

[37] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. *Mobile Object Systems Towards the Programmable Internet*, pages 49–64, 1997.

[38] Peter H. Welch. Java Threads in the Light of occam/CSP. In P. H. Welch and A. W. P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications*, volume 52 of *Concurrent Systems Engineering Series*, pages 259–284, Amsterdam, April 1998. WoTUG, IOS Press. ISBN 90 5199 391 9. URL http://www.cs.kent.ac.uk/pubs/1998/702.

[39] Peter H. Welch. Process Oriented Design for Java: Concurrency for All. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000. ISBN 1-892512-22-x. URL http://www.cs.kent.ac.uk/pubs/2000/1163.

[40] Chris Zetter. Stage: A truly distributed & scalable language. Master's thesis, Imperial College London, June 2009 .