Uso do Padrão AMQP para o Transporte de Mensagens entre Atores Remotos

Thadeu de Russo e Carmo

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

São Paulo, novembro de 2011

Uso do Padrão AMQP para o Transporte de Mensagens entre Atores Remotos

Esta dissertação trata-se da versão original
do aluno Thadeu de Russo e Carmo.

Agradecimentos

Agradecimentos ...

Resumo

Resumo em português ...

Palavras-chave: palavra-chave1, palavra-chave2, palavra-chave3.

Abstract

Abstract ...

Keywords: keyword1, keyword2, keyword3.

Sumário

Lista de Abreviaturas

AMQP	Protocolo Avançado para Enfileramento de Mensagens (*Advanced Message Queuing Protocol*)
API	Interface para Programação de Aplicativos (*Application Programming Interface*)
CLR	Linguagem Comum de Tempo de Execução (*Common Language Runtime*)
IANA	Autoridade de Atribuição de Números da Internet (*Internet Assigned Number Authority*)
JMS	Serviço de Mensagem Java (*Java Message Service*)
JVM	Máquina Virtual Java (*Java Virtual Machine*)
LGPL	Licença Pública Geral Menor (*Lesser General Public License*)
MOM	*Middleware* Orientado a Mensagens (*Message Oriented Middleware*)
MPL	Licença Pública Mozilla (*Mozilla Public License*)
MTA	Agente de Transferência de Correio (*Mail Transfer Agent*)
STM	Memória Transacional de *Software* (*Software Transactional Memory*)
TCP	Protocolo para Controle de Transmissão (*Transmission Control Protocol*)
UDP	Protocolo de Datagrama de Usuário (*User Datagram Protocol*)
XML	Linguagem Extendida de Marcação (*Extented Markup Language*)

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A proposta deste trabalho é explorar a potencial sinergia entre duas classes de sistemas de *software* baseados em troca de mensagens. A primeira classe, usada em ambientes corporativos, compreende os sistemas de *middleware* orientados a mensagens (MOMs) e os *message brokers*. A segunda, voltada para a criação de programas concorrentes, é composta pelas implementações do modelo de atores.

1.1 Troca de mensagens em ambientes corporativos

Sistemas de *middleware* orientados a mensagem trabalham com troca assíncrona de mensagens. As mensagens enviadas são armazenadas e mantidas em filas até que o destinatário esteja pronto para fazer o recebimento e processamento. Em todo envio de mensagem há uma entidade que desempenha o papel de produtor (remetente) e outra que desempenha o papel de consumidor (destinatário) da mensagem. Não existe vínculo entre esses papéis e os papéis de cliente (usuário de um serviço) e servidor (provedor de um serviço), tradicionalmente usados em sistemas baseados em *remote procedure call* (RPC). A diferença entre cliente e servidor é conceitual e somente pode ser definida por humanos que conheçam a semântica das trocas de mensagens.

MOMs formam uma base que simplifica o desenvolvimento de aplicações distribuídas, permite interoperabilidade com baixo acoplamento e provê suporte para o tratamento robusto de erros em caso de falhas. Eles são frequentemente apresentados como uma tecnologia que pode mudar a maneira com que sistemas distribuídos são construídos [ACKM04].

A garantia da entrega de mensagens é uma das características mais importantes dos MOMs. Filas transacionais são uma abstração usada para garantir que toda mensagem recebida pelo MOM seja salva, de modo persistente, e que a remoção da mensagem ocorra somente após a confirmação do recebimento pelo destinatário. Em caso de queda do sistema, quando ele for reiniciado, ocorrerá a entrega das mensagens que foram salvas de modo persistente e cujo recebimento não foi confirmado. A retirada de uma mensagem de uma fila transacional ocorre como parte de uma transação atômica que pode incluir também outras operações, como envios de mensagens e atualizações em bancos de dados, bem como outras retiradas de mensagens.

MOMs tradicionalmente estabelecem ligações ponto-a-ponto entre sistemas, sendo um tanto inflexíveis no que diz respeito ao roteamento e filtragem de mensagens. *Message brokers* são descendentes diretos de MOMs que eliminam essas limitações. Eles agem como intermediários e provêm maior flexibilidade para roteamento e filtragem, além de permitirem que se adicione lógica de negócios para o processamento de mensagens no nível do próprio *middleware*.

Os protocolos usados por *message brokers* variam de produto para produto. A especificação de Java *Messaging Service* (JMS) define uma API padrão para que programas Java possam interagir

com *message brokers*. Boa parte dos *message brokers* têm implementações do padrão JMS. Dentre os mais conhecidos, podemos destacar JBoss Messaging [JBoa], IBM Websphere MQ [IBM] (mais conhecido como MQ Series) e Apache Active MQ[Act].

O protocolo AMQP (*Advanced Message Queuing Protocol*) é uma proposta recente de padronização de um protocolo para *message brokers*. Foi criada por um conjunto de empresas (Red Hat, JPMorgan Chase, Cisco Systems, entre outras), com o objetivo de viabilizar tanto o desenvolvimento quanto a disseminação de um protocolo padrão para esse tipo de sistema.

1.2 Modelos não convencionais de programação concorrente

Nos últimos anos, o aumento da velocidade de *clock* passou a não acompanhar mais o aumento de transistores em processadores por questões físicas, como aquecimento e dissipação, alto consumo de energia e vazamento de corrente elétrica. Por conta dessas e de outras limitações, a busca por ganhos de capacidade de processamento levou à construção de processadores com múltiplos núcleos (*multicore*).

Um dos principais impactos que processadores *multicore* causam no desenvolvimento de programas está relacionado com o modo com que programas são escritos. Para usufruir de ganhos de desempenho com esses processadores, programas precisam ser escritos de forma concorrente [Sut05], uma tarefa que não é simples. A maioria das linguagens de programação e dos ambientes de desenvolvimento não são adequados para a criação de programas concorrentes [SL05].

A abordagem convencional ao desenvolvimento de programas concorrentes é baseada em travas e variáveis condicionais. Em linguagens orientadas a objetos, como Java e C#, cada instância implicitamente possui sua própria trava, e travamentos podem acontecer em blocos de código marcados como sincronizados. Essa abordagem não permite que travas sejam compostas de maneira segura, criando situações propensas a bloqueios e impasses (*deadlocks*). A composição de travas é necessária quando há mais de uma instância envolvida na ação a ser executada de modo exclusivo. Existem ainda outras dificuldades no uso de travas, como esquecimento de se obter a trava de alguma instância, obtenção excessiva de travas, obtenção de travas de instâncias erradas, obtenção de travas em ordem errada, manutenção da consistência do sistema na presença de erros, esquecimento de sinalização em variáveis de condição ou de se testar novamente uma condição após o despertar de um estado de espera [JOW07]. Essas dificuldades mostram que a abordagem convencional, baseada em travas, é inviável para uma programação concorrente modular, ou seja, para a criação de grandes programas concorrentes compostos por programas menores. Elas impulsionaram a pesquisa em abordagens alternativas à programação concorrente convencional.

Dois modelos de programação concorrente não convencionais vem ganhando espaço recentemente. O primeiro deles é a memória transacional implementada por *software* (*software transactional memory*, ou STM) [ST95], um mecanismo de controle de concorrência análogo às transações de bancos de dados. O controle de acesso à memória compartilhada é responsabilidade da STM. Cada transação é um trecho de código que executa uma série atômica de operações de leitura e escrita na memória compartilhada.

O segundo modelo não convencional de programação concorrente é o modelo de atores. Atores [Agh86] são definidos como agentes computacionais que possuem uma caixa de correio e um comportamento. Uma vez que o endereço da caixa de correio de um ator é conhecido, mensagens podem ser adicionadas à caixa para processamento assíncrono, ou seja, o envio de uma mensagem é desacoplado do processamento da mensagem pelo ator.

1.3 A linguagem Scala

Scala [OAC+06] é uma linguagem moderna, com tipagem estática, inferência de tipo e que unifica os paradigmas de programação funcional e orientado a objetos. Vem sendo desenvolvida desde 2001 no laboratório de métodos de programação da EPFL (*École Polytechnique Fédérale de Lausanne*). O código escrito em Scala pode ser compilado para execução tanto na JVM (*Java Virtual Machine*) quanto na CLR (*Common Language Runtime*). A criação da linguagem Scala foi impulsionada pela necessidade de um bom suporte para o desenvolvimento de sistemas componentizados e escaláveis.

A linguagem foi desenvolvida para interoperar bem tanto com Java quanto com C#, e adota parte da sintaxe dessas linguagens, além de compartilhar a maioria dos operadores básicos, tipos de dados e estruturas de controle. Contudo, para atingir seus objetivos, Scala abre mão de algumas convenções enquanto adiciona novos conceitos. Algumas de suas principais características são:

- Scala possui certa semelhança com Java e C#, de modo que tanto programas escritos em Scala podem utilizar bibliotecas escritas em Java e C#, quanto o inverso;
- Scala possui um modelo uniforme para objetos, onde todo valor é um objeto e toda operação é um método;
- Scala é uma linguagem funcional e todas as funções são valores de primeira classe. No contexto de linguagens de programação, valores de primeira classe são entidades que podem ser criadas em tempo de execução, utilizadas como parâmetro, retornadas por uma função, ou ainda atribuídas a variáveis;
- Scala permite construções via composição de classes e *traits*. *Traits* possuem definições de métodos e campos assim como uma classe abstrata, porém não definem construtores. *Traits* são importantes unidades para reuso de código em Scala já que classes ou objetos podem ser compostos (*mixin*) por diversas *traits*;
- Scala permite a decomposição de objetos via casamento de padrões;
- Scala possui suporte ao tratamento de XML (*Extended Markup Language*) na própria sintaxe da linguagem;
- Scala não possui o conceito de membros de classes estáticos. A linguagem possui o conceito de *singleton objects*, que representa uma instância única de uma classe. *Singleton objects* são definidos com a palavra-chave `object` ao invés de `class`. *Singleton objects* que possuem uma classe definida com o mesmo nome são chamados de objeto acompanhante (*companion object*);
- Scala possui suporte a *currying* que, junto com as funções de ordem superior, permite a criação de novas estruturas de controle.

Optamos por desenvolver este trabalho em Scala tanto pela linguagem ser executável na JVM e interoperar naturalmente com Java, como por suas características proverem facilidades para a implementação de atores.

1.4 Objetivos

Nosso propósito é criar uma implementação em Scala do modelo de atores que use o padrão AMQP para o transporte de mensagens entre atores remotos. Geraremos um protótipo baseado na implementação do modelo de atores feito projeto Akka. Iremos substituir o mecanismo de transporte atualmente usado pelo Akka por um que utiliza um *message broker* AMQP.

1.5 Contribuições

As principais contribuições deste trabalho são as seguintes:

- Item 1. Texto texto.
- Item 2. Texto texto.

1.6 Organização do Trabalho

No Capítulo 2, apresentamos o modelo de atores, sua semântica e algumas das implementações do modelo. No Capítulo 3 analisamos em maiores detalhes a implementação do modelo de atores feita no projeto Akka. No Capítulo 4 apresentamos o protocolo AMQP, suas camadas e algumas implementações de *message brokers* que implementam o protocolo. No capítulo 5 apresentamos uma estrutura para a troca de mensagens entre entidades remotas via *message broker* AMQP. No capítulo 6 apresentamos nossa implementação de atores remotos que utiliza a estrutura que definimos do capítulo 5.

Finalmente, no Capítulo **??** discutimos algumas conclusões obtidas neste trabalho. Analisamos as vantagens e desvantagens do método proposto ...

Capítulo 2

Atores

Apresentamos neste capítulo o modelo de atores. Na seção 2.1 apresentamos a semântica do modelo de atores. Na seção 2.2 apresentamos um breve histórico com alguns estudos relacionados ao modelo. Por fim na seção 2.3, apresentamos algumas das implementações do modelo.

2.1 Modelo

O modelo de atores é um modelo para programação concorrente em sistemas distribuídos, que foi apresentado como uma das possíveis alternativas ao uso de memória compartilhada e travas. Atores são agentes computacionais autônomos e concorrentes que possuem uma fila de mensagens e um comportamento [Agh86].

O modelo define que toda interação entre atores deve acontecer via trocas assíncronas de mensagens. Uma vez que o endereço da fila de mensagens de um ator é conhecido, mensagens podem ser enviadas ao ator. As mensagem enviadas são armazenadas para processamento assíncrono, desacoplando o envio de uma mensagem do seu processamento.

Toda computação em um sistema de atores é resultado do processamento de uma mensagem. Cada mensagem recebida por um ator é mapeada em uma 3-tupla que consiste de:

1. Um conjunto finito de envios de mensagens para atores cujas filas tenham seus endereços conhecidos (um desses atores pode ser o próprio ator destinatário da mensagem que está sendo processada);
2. Um novo comportamento, que será usado para processar a próxima mensagem recebida;
3. Um conjunto finito de criações de novos atores.

Um ator que recebe mensagens faz o processamento individual de cada mensagem em uma execução atômica. Cada execução atômica consiste no conjunto de todas as ações tomadas para processar a mensagem em questão, não permitindo intercalações no processamento de duas mensagens.

Em um sistema de atores, envios de mensagens (também chamados de comunicações) são encapsulados em tarefas. Uma tarefa é uma 3-tupla que consiste em um identificador único, o endereço da fila do ator destinatário e a mensagem. A configuração de um sistema de atores é definida pelos atores que o sistema contém e pelo conjunto de tarefas não processadas. É importante ressaltar que o endereço da fila de mensagens na tarefa deve ser válido, ou seja, ele deve ter sido previamente comunicado. Há três maneiras de um ator ao receber uma mensagem m, passar a conhecer um destinatário no qual o ator pode enviar mensagens:

1. O ator já conhecia o endereço da fila do destinatário antes do recebimento da mensagem m;
2. O endereço da fila estava presente na mensagem m;
3. Um novo ator foi criado como resultado do processamento da mensagem m.

Um aspecto importante que faz parte o modelo de atores é existência de igualdade durante a execução (*fairness*). Sem essa propriedade, um sistema de atores teria ambos a garantia de entrega e a componibilidade comprometida [AMST98]. No contexto de atores, garantia de entrega significa que uma mensagem que foi depositada na fila de mensagens de um ator, não deve ficar armazenada indefinidamente sem ser eventualmente repassada para o ator fazer o seu processamento. A ordem de chegada de mensagens obedece uma ordem linear, e fica a cargo da implementação do modelo arbitrar no caso de conflitos, caso duas mensagens sejam recebidas no mesmo momento. O modelo assume uma entrega eventual, pois podem ocorrer casos onde o ator esteja em um estado não propício a receber novas mensagens, como por exemplo, estar executando um laço infinito ou alguma operação ilegal. A garantia de entrega de mensagens não assume que as mensagens entregues tenham um processamento semanticamente significativo. O processamento de uma mensagem depende do comportamento definido no ator. Por exemplo, um tipo de atores poderia optar por descartar todas as mensagens recebidas.

Gul Agha, em sua definição da semântica do modelo de atores [Agh86], afirma que "em qualquer rede real de agentes computacionais, não é possível prever quando uma mensagem enviada por um agente será recebida por outro". A afirmação é enfatizada para redes dinâmicas, onde novos agentes podem ser criados ou destruídos, e reconfigurações como migrações de agentes para diferentes nós podem acontecer. Em sua conclusão, Agha afirma que um modelo realista deve assumir que a ordem de recebimento das mensagens não é determinística, mas sim arbitrária e desconhecida. O não determinismo na ordem de recebimento das mensagens pelo ator não interefe na garantia de entrega das mensagens.

O comportamento de um ator define como o ator irá processar a próxima mensagem recebida. Ao processar uma mensagem, o ator deve definir o comportamento substituto que será utilizado para processar a próxima mensagem que lhe for entregue. Mostramos na figura 2.1 a máquina de estados de um ator processando uma mensagem recebida e definindo seu novo comportamento. Nessa figura, o ator A com comportamento inicial B_1 retira a mensagem m_1 de sua fila e a processa. Para esse exemplo, o processamento da mensagem m_1 resultou na criação de um novo ator A', com comportamento inicial $B_{1'}$, e no envio da mensagem $m_{1'}$ do ator A para o ator A'. O ator A passa a ter o comportamento B_2, que será usado no processamento da mensagem m_2. Vale ressaltar que o comportamento substituto não precisa ser necessariamente diferente do comportamento anterior. Por exemplo, os comportamentos B_1 e B_2 podem ser comportamentos idênticos.

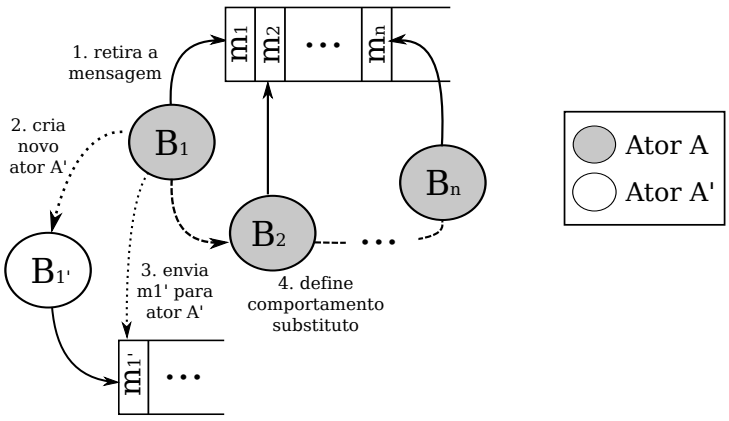

Figura 2.1: *Máquina de estados de um ator.*

No modelo de atores, a real localização de um ator não afeta a interação com outros atores. Os atores que um ator conhece podem estar distribuídos em diferentes núcleos de um processador, ou mesmo em diferentes nós de uma rede de computadores. A transparência da localidade abstrai a infra-estrutura e permite que programas sejam desenvolvidos de modo distribuído, sem que a real localização de seus atores seja conhecida. Duas consequências diretas da transparência da localidade são o encapsulamento do estado do ator e a capacidade de se mover atores entre os diferentes nós de uma rede de computadores.

Apesar de o modelo ser bem explícito e claro em relação a interação ser via troca de mensagens e dizer que o estado de um ator não deve ser compartilhado, implementações poderiam permitir que o estado interno de um ator fosse acessado diretamente por outro ator. Por exemplo, um ator em sua pilha de execução fazer a invocação de algum método de outro ator. Mesmo que a interface de um ator não permita o compartilhamento do seu estado interno que não seja via troca de mensagens, mensagens enviadas poderiam expor um compartilhamento indesejado de estado. O uso de mensagens imutáveis e passadas via cópia é uma abordagem mais apropriada para evitar um compartilhamento indesejado de memória.

Mobilidade é definida como a habilidade de se poder mover uma computação de um nó para outro, e pode ser classificada como mobilidade fraca ou mobilidade forte. Mobilidade fraca é a habilidade de se transferir código entre nós, enquanto que mobilidade forte é definida como a habilidade de se transferir ambos o código e o estado da execução [FPV98]. Em um sistema baseado em atores, mobilidade fraca permite que atores que não possuem mensagens em suas filas e não estão fazendo processamento algum sejam transferidos para outros nós.

Pelo fato de o modelo de atores prover transparência de localidade e encapsulamento, a mobilidade se torna natural. Mover atores entre diferentes nós em um sistema de atores é uma necessidade importante para se obter um melhor desempenho, tolerância a falhas e sistemas reconfiguráveis [PA94].

A semântica padrão do modelo de atores apresentada nesta seção objetiva proporcionar uma arquitetura modular e componível [AMST98], e um melhor desempenho para sistemas concorrentes e distribuídos, em particular para situações onde as aplicações precisem de escalabilidade [KA95].

2.2 Breve histórico

O modelo de atores foi originalmente proposto por Carl Hewitt em 1971 [Hew71]. O termo ator foi originalmente utilizado para descrever entidades ativas que analizavam padrões para iniciar atividades. O conceito de atores passou então a ser explorado pela comunidade científica e em 1973, a noção de atores se aproximou do conceito de agentes existente na área de inteligência artificial distribuída, por possuírem intenções, recursos, monitores de mensagens e um agendador [HBS73].

Em 1975, Irene Greif desenvolveu um modelo abstrato de atores orientado a eventos onde cada ator guardava um histórico dos seus eventos. O objetivo do estudo era analizar a relação causal entre os eventos. Baker e Hewitt formalizaram um conjunto de axiomas para computação concorrente em 1977. Desse estudo surgiu uma propriedade importante: a ordem em que eventos são gerados deve ser obedecida a fim de prever violações de causalidade. No mesmo ano, Hewitt apresentou um estudo sobre como o entendimento dos padrões de troca de mensagens entre atores pode ajudar na definição de estruturas de controle. Esse estudo demonstrou o uso do estilo de passagem de continuações no modelo de atores.

O conceito de guardião foi definido em 1978 por Attardi e Hewitt. Um guardião é uma entidade que regula o uso de recursos compartilhados. Guardiões incorporam explitamente a noção de estado e são responsáveis por agendar o acesso e fazer a proteção dos recursos. Hewitt e Atkinson definiram em 1979 um conceito relacionado ao de guardião denominado seriador (*serializer*). Um seriador age como um monitor, porém, ao invés de aguardar sinais explícitos vindos dos processos,

seriadores procuram ativamente por condições que permitam que processos em espera possam voltar a ser executados. Em 1987, Henry Lieberman implementou em Lisp a linguagem Act1 [Lie87], uma linguagem de atores com guardiões, seriadores e atores chamados de "trapaceiros" (*rock-bottom*). Atores trapaceiros são atores que podem burlar as regras do modelo de atores para, por exemplo, fazer uso dados primitivos e funções da linguagem usada em sua implementação.

Gul Agha definiu em 1986 um sistema simples de transição para atores. Em seu trabalho foi desenvolvido o conceito de configurações, recepcionistas e atores externos. Atores recepcionistas são atores que ocultam a existência de outros atores em um sistema de atores, agindo como intermediários no recebimento das mensagens. O uso de atores recepcionistas permite uma forma de encapsulamento, já que eles acabam agindo como interface para atores externos. Esse modelo foi implementado na linguagem Acore por Carl Manning em 1987, e na linguagem Rosette por Tomlinson e outros em 1989.

2.3 Implementações

O suporte ao modelo de atores pode estar disponível em uma linguagem de programação, seja fazendo parte da estrutura da linguagem, como uma biblioteca embutida em sua distribuição ou ainda como uma biblioteca separada da distribuição padrão. Apresentamos na sub-seção 2.3.1 algumas linguagens de programação baseadas no modelo de atores, e que possuem primitivas que facilitam a criação de sistemas baseados em atores. Apresentamos na sub-seção 2.3.2 algumas bibliotecas que adicionam o suporte ao modelo de atores a linguagens mais gerais.

2.3.1 Linguagens

Linguagens como Axum [Cor09], SALSA [VA01] e Erlang [Arm07] são exemplos de linguagens baseadas no modelo de atores. Nessas linguagens, o suporte é dado via primitivas na estrutura da linguagem.

Axum

Axum (anteriormente conhecida como Maestro) é uma linguagem experimental orientada a objetos[1] criada pela Microsoft Corporation para o desenvolvimento de sistemas concorrentes na plataforma .Net [Cor09]. A criação da linguagem teve como motivação permitir que sistemas modelados como componentes que interajem entre si, pudessem ser traduzidos naturalmente para código. Pelo fato de ser uma linguagem da plataforma .Net, Axum pode interagir com outras linguagens da plataforma, como VB.Net, C# e F#.

Em Axum, o termo "ator" é substituído pelo termo "agente" (*agent*). Agentes são executados como *threads* dentro da CLR (*Common Language Runtime*) e são definidos com o uso da palavra-chave `agent`. A troca de mensagens é feita via canais. Os canais são responsáveis por definir os tipos de dados que trafegam neles com o uso das palavras-chaves `input` e `output`. Canais possuem duas extremidades, a extremidade implementadora (*implementing end*) e a extremidade de uso (*using end*). Agentes que implementam o comportamento da extremidade implementadora de um canal, passam a agir como servidores de mensagens de seus respectivos canais. A extremidade de uso do canal é visível e deve ser utilizada por outros agentes para fazer o envio de mensagens.

A linguagem possui ainda o conceito de domínio. Domínios são definidos com a palavra-chave `domain` e permitem que agentes definidos dentro de um domínio compartilhem informações de um modo diferente. Uma instância de domínio pode ter atributos que são usados para compartilhamento seguro e controlado de informações.

[1] O projeto está em uma encubadora de projetos.

Apesar da linguagem ter seu desenvolvimento interrompido na versão 0.3 no início de 2011, segundo os seus autores os conceitos implementados na linguagem poderão ser portados para as linguagens C# e Visual Basic [Corb].

SALSA

SALSA (*Simple Actor Language System and Architecture*) é uma linguagem que foi criada em meados de 2001 no *Rensselaer Polytechnic Institute* para facilitar o desenvolvimento de sistemas abertos dinamicamente reconfiguráveis. SALSA é uma linguagem concorrente e orientada a objetos que implementa a semântica do modelo de atores. A linguagem possui um pré-processador que converte o código fonte escrito em SALSA para código fonte Java. Esse que por sua vez pode ser compilado para *bytecode* e ser executado sobre a JVM.

A linguagem possui algumas primitivas para a criação, uso e organização dos atores. A primitiva `behavior` é equivalente a palavra-chave `class` de Java, e é utilizada para definir os métodos que definem o comportamento do ator. Assim como na hierarquia de classes em Java, onde não existe herança múltipla e classes podem implementar zero ou muitas *interfaces*, *behaviors* seguem a mesma regra. O *behavior* `salsa.language.UniversalActor` é análogo a classe `java.lang.Object`, e todos os *behaviors* o extendem direta ou indiretamente. A primitiva `<-` é utilizada para enviar uma mensagem a um ator.

A implementação de atores da linguagem utiliza o objeto `salsa.language.Actor` como ator base que extende `java.lang.Thread`, criando uma relação onde cada ator executa em uma *thread*. Cada ator possui uma fila de mensagens onde as mensagens recebidas ficam armazenadas até que o método `run` as retire para que o método correspondente seja executado via reflexão.

Além de prover a implementação da semântica do modelo de atores, a linguagem introduz ainda três abstrações para facilitar a coordenação das interações assíncronas entre os atores:

1. Continuações com passagem de *token* (*Token-passing continuations*): Envios de mensagens para atores resultam em invocações de métodos. Esses métodos podem retornar valores que precisarão ser repassados a outros atores. Os valores de retorno são definidos como *tokens*. Essa abstração permite que uma mensagem enviada a um ator contenha uma referência para o ator que irá consumir o *token*, como mostrado no exemplo a seguir:

```
gerente <- aprovacao1(500)@ gerente2 <- aprovacao2(token)
           @ diretor <- aprovacaoFinal(token);
```

   Nesse exemplo, `gerente` processa a mensagem `aprovacao1(500)` e seu retorno, quando computado, é passado como `token` no envio da mensagem para o ator `gerente2`. A palavra *token* é uma palavra-chave e seu valor é associado no contexto do último *token* passado. O ator `diretor` recebe como argumento o resultado da computação do ator `gerente2`. Ainda nesse exemplo, utilizamos a primitiva `@` para indicar que desejamos utilizar a continuação com passagem de *token*;

2. Continuações de junção (*Join continuations*): Atores podem receber vetores com *tokens* recebidos de diversos atores. Essa abstração permite que os *tokens* sejam agrupados para que o ator que os está recebendo só execute mediante a todos os *tokens* aguardados terem sido recebidos. Por exemplo:

```
join(gerente <- aprovacao1(500), gerente2 <- aprovacao2(500))
     @ diretor <- aprovacaoFinal(500);
```

Nesse exemplo, os dois atores `gerente1` e `gerente2` fazem suas aprovações em paralelo e, somente quando os dois tiverem dado suas aprovações, o ator `diretor` irá receber a mensagem de `aprovacaoFinal`;

3. Continuações de primeira classe (*First-class continuations*): Essa abstração é análoga à funções de ordem superior, onde funções podem receber outras funções. No caso da linguagem SALSA, a abstração permite que continuações sejam passadas como parâmetro para outras continuações. O processamento da mensagem pode optar por delegar o restante do processamento à continuação recebida, por exemplo em chamadas recursivas. No processamento de uma mensagem, a continuação recebida como parâmetro é acessada pela palavra-chave `currentContinuation`.

Erlang

Erlang é uma linguagem funcional, com tipagem dinâmica e executada por uma máquina virtual Erlang. Voltada para o desenvolvimento de sistemas distribuídos de larga escala e tempo real, foi desenvolvida nos laboratórios da Ericsson no período de 1985 à 1997 [Arm97].

A linguagem em si, embora bem enxuta, possui características interessantes para simplificar o desenvolvimento de sistemas concorrentes. Exemplos de tais características são: variáveis de atribuição única, casamento de padrões e um conjunto de primitivas que inclui `spawn` para criação de atores, `send` e `!` para o envio de mensagens, `receive` para o recebimento de mensagens e `link` para a definição de adjacências entre atores. Ademais a linguagem dá suporte a hierarquias de supervisão entre atores e troca quente de código (*hotswap*).

Em Erlang, atores são processos ultra leves criados dentro de uma máquina virtual. Embora Erlang implemente o modelo de atores, sua literatura e suas bibliotecas não utilizam o termo "ator" (*actor*), mas sim o termo "processo" (*process*). A criação, destruição e troca de mensagens entre atores é extremamente rápida. Num teste feito em um computador com $512MB$ de memória, com um processador de $2.4GHz$ Intel Celeron rodando Ubuntu Linux, a criação de 20000 processos levou em média $3.5\mu s$ de tempo de CPU por ator e $9.2\mu s$ de tempo de relógio por ator [Arm07]. Esses números mostram que a criação dos atores é rápida e que com pouca memória, muitos atores podem ser criados.

Com a possibilidade de se criar uma quantidade considerável de atores, o uso de uma hierarquia de supervisão torna-se extremamente importante. No que diz respeito ao tratamento de erros em atores filhos, as primitivas existentes na linguagem dão suporte a três abordagens:

1. Não se tem interesse em saber se um ator filho foi terminado normalmente ou não;
2. Caso um ator filho não tenha terminado normalmente, o ator criador também é terminado;
3. Caso um ator filho não tenha terminado normalmente, o ator criador é notificado e pode fazer o controle de erros da maneira que julgar mais apropriada.

Em Erlang é possível criar atores em nós remotos (*remote spawn*). Vale ressaltar que o código do ator deve estar acessível na máquina virtual onde o ator irá ser executado, pois não há suporte para carga remota de código. Uma vez que alguns detalhes de infra-estrutura foram observados (as máquinas virtuais Erlang necessitam se autenticar umas com as outras), a troca de mensagens entre atores remotos acontece de maneira transparente. No processo de envio, as mensagens trafegam com o uso de *sockets* TCP e UDP.

2.3.2 Bibliotecas

Diversas linguagens de programação possuem suporte ao modelo de atores via bibliotecas. Essas bibliotecas disponibilizam arcabouços para desenvolvimento de sistemas concorrentes baseados no modelo de atores. Listamos a seguir algumas linguagens e referências para algumas de suas bibliotecas:

- C++: Act++ [KML93], Thal [Kim97] e Theron [Mas];
- Smalltalk: Actalk [Bri89];
- Python: Parley [Lee] e Stackless Python [Tis];
- Ruby: Stage [Sil08] e a biblioteca de atores presente na distribuição Rubinius[2] [Rub];
- .Net: Asynchronous Agent Library [Cora] e Retlang [Retb];
- Java: Akka [Akk], Kilim [SM08], Jetlang [Reta] e Actor Foundry [Ope];
- Scala: Scala Actors [HO09], Akka [Akk] e Scalaz [Sca].

Pelo fato de nosso trabalho ter sido desenvolvido sobre a JVM, mais especificamente na linguagem Scala, optamos por não comparar todas as bibliotecas listas anteriormente para não nos distanciarmos do escopo deste trabalho. Karmani e Agha fizeram uma análise comparativa entre alguns arcabouços de atores para a JVM e os apresentaram em [KSA09]. Nessa análise, eles fizeram comparações considerando a implementação da semântica de execução e das abstrações.

Apresentamos a seguir informações sobre as bibliotecas de atores que foram consideras como opções e analizadas para o desenvolvimento do nosso trabalho.

A biblioteca de atores de Scala

A distribuição de Scala inclui uma biblioteca de atores inspirada pelo suporte a atores de Erlang.

Nesta biblioteca os atores foram projetados como objetos baseados em *threads* Java e possuem métodos como `send`, `!`, `receive`, além de outros métodos como `act` e `react`. Cada ator possui uma caixa de correio para o recebimento e armazenamento temporário das mensagens. O processamento de uma mensagem é feito por um bloco `receive` declarado dentro do método `act`.

No método `receive` são definidos os padrões a serem casados com as mensagens que o ator processa e as ações associadas. A primeira mensagem que casar com qualquer dos padrões é removida da caixa de correio e a ação correspondente é executada. Caso não haja casamento com nenhum dos padrões, o ator é suspenso.

Atores são executados em um *thread pool*[3] que cresce conforme a demanda. É importante ressaltar que o uso do método `receive` fixa o ator à *thread* que o está executando, limitando superiormente a quantidade de atores pelo número de *threads* que podem ser criadas. É recomendado o uso do método `react` ao invés do método `receive` sempre que possível, já que um ator que não está em execução cede sua *thread* para que ela execute outro ator. Do ponto de vista de funcionalidade, o método `receive` é bloqueante e pode retornar valores, enquanto que o método `react`, além de não retornar valores, faz com que o ator passe a reagir aos eventos (recebimentos de mensagens). As implicações práticas no uso de `react` em relação ao `receive` da perspectiva de codificação são: o uso da estrutura de controle `loop` ao invés de laços tradicionais, para indicar que o ator deve

[2] Rubinius é uma implementação da linguagem Ruby que possui uma máquina virtual escrita em C++.
[3] Idealmente o tamanho do *thread pool* corresponde ao número de núcleos do processador.

continuar reagindo após o processamento de uma mensagem (o uso dos laços tradicionais bloquearia a *thread*); o ator ser responsável por invocar eventuais métodos após o processamento de uma mensagem [HO09].

Além de métodos para envio de mensagens equivalentes às primitivas de Erlang, a biblioteca de atores de Scala implementa métodos adicionais, que facilitam o tratamento de algumas necessidades específicas. Esses métodos são: `!?` faz envio síncrono e aguarda uma resposta dentro de um tempo limite especificado; `!!` faz o envio assíncrono da mensagem e recebe um resultado futuro correspondente a uma resposta.

O suporte a atores remotos faz parte da biblioteca, porém com algumas restrições em comparação com os atores de Erlang. Nesta biblioteca, atores remotos são atores remotamente acessíveis. já que não é possível a criação de um ator em um nó que não seja o local, ou seja, *remote spawns* não são possíveis. Os atores são acessíveis remotamente via *proxies*. Para obter uma referência a um ator remoto, um cliente faz uma busca em um determinado nó (uma JVM identificada por um par com endereço do hospedeiro e a porta), utilizando como chave o nome sob o qual o ator foi registrado. Esta abordagem, apesar de soar restritiva, evita um problema importante que é a necessidade de carga remota da classe do ator (*remote class loading*) e torna desnecessário o uso de interfaces remotas como em Java RMI. O tráfego das mensagens é feito via seriação padrão Java através de *sockets* TCP.

O projeto Akka

O projeto Akka é composto por um conjunto de módulos escritos em Scala[4], que implementam uma plataforma voltada para o desenvolvimento de aplicações escaláveis e tolerantes a falhas. Na versão 1.0, suas principais características são: uma nova biblioteca de atores locais e remotos, suporte a STM, hierarquias de supervisão e uma combinação entre atores e STM (*"transactors"*) que da suporte a fluxos de mensagens baseados em eventos transacionais, assíncronos e componíveis. Akka oferece ainda, uma série de módulos adicionais para integração com outras tecnologias.

A biblioteca de atores do projeto Akka é totalmente independente da que é parte da distribuição de Scala, apesar de também seguir as idéias de Erlang. O comportamento dos atores Akka no recebimento de mensagens inesperadas (que não casam com nenhum dos padrões especificados em um `receive`) é diferente dos atores de Erlang e Scala, onde o ator é suspenso. No caso de atores Akka, tais mensagens provocam o lançamento de exceções.

O suporte a atores remotos do projeto Akka é bem mais completo do que o oferecido pela biblioteca de atores de Scala e será discutido no capítulo 3. Assim como a biblioteca de atores de Scala, o Akka oferece métodos para diferentes tipos de envio de mensagens: `!!` semelhante ao método `!?` da biblioteca de atores de Scala, no qual o remetente fica bloqueado aguardando uma resposta durante um tempo limite; `!!!` semelhante ao método `!!` da biblioteca de atores de Scala, que devolve um resultado futuro ao remetente.

No que diz respeito à seriação das mensagens para um ator remoto, o Akka oferece as seguintes opções: JSON [Cro06], Protobuf [Goo], SBinary [Har] e seriação Java padrão. O transporte das mensagens é feito via *sockets* TCP, com o auxílio do JBoss Netty [JBob], um arcabouço para comunicação assíncrona dirigido a eventos e baseado em *sockets*. Esse arcabouço oferece facilidades para compressão de mensagens, que são utilizados pelo Akka.

Optamos por utilizar a implementação de atores do projeto Akka para o desenvolvimento deste trabalho. Tomamos como base a versão 1.0 do Akka por ser a última versão estável disponível quando iniciamos o desenvolvimento. Nossa escolha pela implementação de atores do projeto Akka foi impulsionada pelos fatores a seguir:

[4] O projeto disponibiliza também uma versão de suas APIs voltada para aplicações Java.

- Ter código aberto;
- Ser escrito em Scala;
- Possuir uma implementação mais completa de atores remotos em relação às demais implementações que observamos;
- A implementação de atores remotos possuir certa disposição para novas implementações de transportes;
- Ter grande volume de atividade na comunidade de usuários e de desenvolvedores;
- Estar com desenvolvimento ativo;
- Ser não apenas um projeto, mas sim de um produto que foi criado com objetivos comerciais[5].

Os principais detalhes da implementação da biblioteca de atores feita no projeto são apresentados no capítulo 3.

[5] A distribuição do Akka faz parte da *Typesafe Stack*, uma plataforma para desenvolvimento de sistemas concorrentes e escaláveis da empresa Typesage

Capítulo 3

Atores no projeto Akka

O projeto Akka disponibiliza tanto atores locais quanto remotos. O termo "ator local" é usado para denotar um ator que pode receber mensagens apenas de atores residentes na mesma máquina virtual. Por outro lado, um ator remoto pode receber mensagens de quaisquer outros atores, inclusive daqueles residentes em outras máquinas virtuais. Em outras palavras, o termo "ator remoto" é um sinônimo de "ator remotamente acessível".

Na seção 3.1 examinaremos a implementação de atores locais. Nosso objetivo é focar na criação desses atores, no envio e despachamento de mensagens e na hierarquia de supervisão. Na seção 3.2 examinaremos a implementação de atores remotos. Nosso objetivo é focar na estrutura definida para o suporte a atores remotos, seriação de mensagens e no protocolo que foi definido para o envio de mensagens para atores remotos.

3.1 Atores locais

A definição de atores locais acontece por meio de extensões da *trait* `akka.actor.Actor`, onde deve-se prover uma implementação para o método `receive` como mostrado na listagem 3.1. A definição de um ator descreve o comportamento inicial que o ator terá. O ator propriamente dito (aquele que possui as funcionalidades providas pelo arcabouço) é uma instância de `akka.actor.ActorRef`. `ActorRefs` são imutáveis, seriáveis, identificáveis, possuem um relacionamento único com a definição do ator e armazenam o endereço do nó onde foram criadas.

A *trait* `Actor` possui em sua declaração uma referência para o seu `ActorRef` definida como `self`. No caso de atores locais, `self` referencia uma instância de `akka.actor.LocalActorRef`. Com essa referência, a classe que define o comportamento do ator pode alterar as definições padrão providas pelo arcabouço e utilizar seus métodos para, por exemplo, responder ou encaminhar mensagens recebidas e acessar a referência de um ator supervisor.

```
1 class SampleActor(val name: String) extends akka.actor.Actor {
2   def this() = this("No name")
3
4   def receive = {
5     case "hello" => println("%s received hello".format(name))
6     case       _ => println("%s received unknown".format(name))
7   }
8 }
```

Listagem 3.1: *Classe SampleActor.*

Atores são criados pelo método `actorOf` do objeto `akka.actor.Actor` como mostrado na

listagem 3.2. Nessa listagem, a instância do ator é criada por reflexão com base no tipo da classe onde o construtor padrão é executado. O método `actorOf` é sobrecarregado para permitir que uma função sem argumentos e com tipo de retorno `Actor`, possa ser utilizada como alternativa ao construtor padrão na criação do ator, como mostrado na listagem 3.3.

Nas duas listagens tivemos que chamar explicitamente o método `start` para iniciar o ator. Os atores do projeto Akka possuem um conjunto de estados simples, bem definido e linear. Um ator logo após sua criação está no estado chamado de "novo" e ainda não pode receber mensagens. Após a invocação do método `start`, o ator passa ao estado "iniciado" e está apto a receber mensagens. Uma vez que o método `exit` ou `stop`[1] é invocado, o ator passa ao estado de "desligado" e não pode mais executar ação alguma.

```
1  val theActor = Actor.actorOf[SampleActor].start
2  theActor ! "hello"
```

Listagem 3.2: *Criação e inicialização de SampleActor via construtor padrão.*

```
1  val function = {
2    // outras ações
3    new SampleActor("John")
4  }
5  val theActor = Actor.actorOf(function).start
6  theActor ! "hello"
```

Listagem 3.3: *Criação e inicialização de SampleActor via função de inicialização.*

As mensagens que são enviadas para um ator são colocada sincronamente na fila de mensagens do ator, levando tempo $O(1)$. As mensagens enfileiradas são então despachadas assincronamente para a função parcial definida no bloco `receive`, como mostrado na figura 3.1.

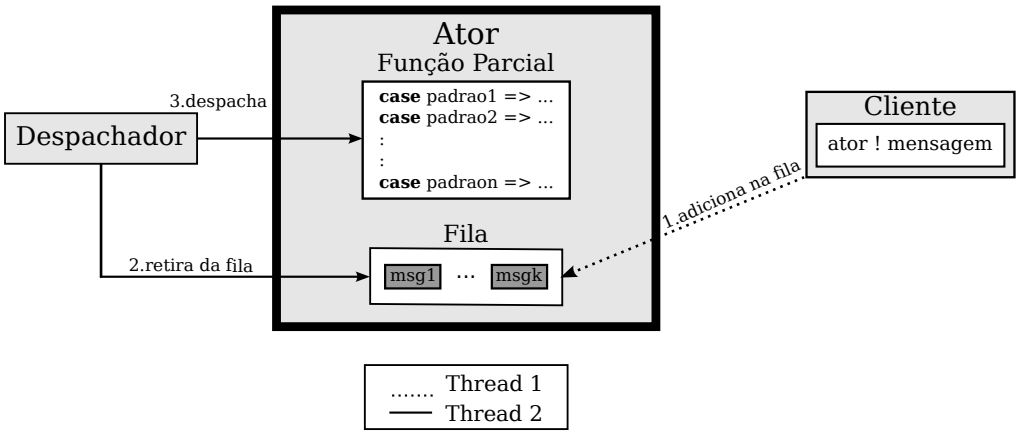

Figura 3.1: *Envio e despacho de mensagens para atores locais.*

3.1.1 Despachadores

O despachador de um ator é uma entidade que possui um papel importante. O despachador permite a configuração do tipo da fila do ator e a semântica do despachamento das mensagens. A fila de um ator pode ser durável ou transiente e ter seu tamanho limitado superiormente ou não. O Akka possui em sua distribuição os quatro despachadores listados a seguir:

[1] O método `exit` invoca internamente o método `stop` que é responsável por desligar o ator.

- Despachador impulsionado por eventos: O despachador é definido na classe `akka.dispatch.ExecutorBasedEventDrivenDispatcher` e é normalmente compartilhado por diversos atores de diferentes tipos, já que ele utiliza um *thread pool* para agendar as ações de despachamento. É o despachador mais flexível em termos de configurações, já que permite que parâmetros do *thread pool* e da fila de mensagens sejam configurados;

- Despachador impulsionado por eventos com balanceamento de carga: O despachador é definido na classe `akka.dispatch.ExecutorBasedEventDrivenWorkStealingDispatcher` e é semelhante ao despachador anterior, já que também é impulsionado por eventos. Esse despachador deve ser usado em atores do mesmo tipo já que permite que atores que não estejam processando mensagens possam "roubar" mensagens das filas dos atores que estão sobrecarregados, permitindo o balanceamento do processamento das mensagens entre os atores;

- Despachador quente: O despachador é definido na classe `akka.dispatch.HawtDispatcher` e é inspirado no *Grand Central Dispacher* do Mac OS X [App09]. Esse despachador define um *thread pool* cujo tamanho é ajustado automaticamente para minimizar a quantidade de *threads* concorrentes e inativas[2]. A grande diferença desse despachador é a capacidade de agrupar diversos eventos gerados pela aplicação, por exemplo diversas mensagens recebidas, gerando uma única tarefa assíncrona;

- Despachador impulsionado por *threads*: O despachador é definido na classe `akka.dispatch.ThreadBasedDispatcher` e é o mais simples de todos os dispachadores, já que associa uma *thread* para cada ator. O uso desse despachador implica no ator utilizar filas transientes. É importante ressaltar que internamente é utilizado uma fila que faz o bloqueio da *thread* que está tentando adicionar um elemento, caso o limite superior da fila tenha sido atingido (para o caso onde há um limite informado).

Por padrão atores são associados ao despachador global impulsionado por eventos. A configuração padrão do despachador define uma fila transiente sem limite máximo de mensagens. Caso seja necessário definir um outro despachador para um ator, a definição deve acontecer antes de o ator ser inciado via método `self.dispatcher`. O Akka permite ainda que novos dispachadores sejam criados.

3.1.2 Envios de respostas

Envios de mensagens podem gerar mensagens de resposta ao remetente. Os métodos para envios de mensagens capturam implicitamente uma cópia da referência da entidade que está fazendo o envio (*sender*). A assinatura do método `!!` é mostrada na listagem 3.4. Uma cópia da referência do *sender* é enviada junto com a mensagem para que o ator destinatário possa eventualmente enviar uma mensagem de resposta.

```
1 def !!(message: Any, timeout: Long = this.timeout)
2   (implicit sender: Option[ActorRef] = None): Option[Any] = {
3   ...
4 }
```

Listagem 3.4: *Assinatura do método !!.*

O ator destinatário pode responder a uma mensagem via método `self.reply`. A execução do método para envio de uma resposta nada mais é que um envio de mensagem como já apresentado.

[2]Idealmente a quantidade de *threads* corresponde ao número de núcleos disponíveis.

A implementação dos métodos `!!` e `!!!` é baseada em resultados futuros. Resultados futuros são representações de resultados de computações assíncronas. Em Java, a classe `java.util.concurrent.FutureTask` provê uma implementação básica de resultados futuros. A implementação feita no projeto Akka tem sua própria implementação para representar resultados futuros de computações assíncronas. Essa implementação provê algumas facilidades em relação a implementação de Java que são necessárias em seu uso pelo arcabouço. A *trait* `CompletableFuture` define métodos para que resultados futuros possam ser completados por outras entidades como mostrado na listagem 3.5. Essa implementação foi inspirada no projeto Actorom [Bos]. A implementação do Akka é construída sobre classes do pacote `java.util.concurrent`.

```
1 trait CompletableFuture[T] extends Future[T] {
2   def completeWithResult(result: T)
3   def completeWithException(exception: Throwable)
4   def completeWith(other: Future[T])
5 }
```

Listagem 3.5: *Trait CompletableFuture.*

Envios de mensagens feitos via métodos `!!` e `!!!` são considerados como feitos por "remetentes futuros". Uma vez que o ator enviou a mensagem de resposta, a mensagem não é coloca em uma fila de mensagens assim como acontece para atores, mas é utilizada para indicar no resultado futuro que a computação foi completada.

Na `ActorRef`, um envio de uma mensagem feito via método `!!!` é processado da seguinte maneira:

1. Uma instância de `CompletableFuture` é criada dentro do método;
2. A mensagem é colocada da fila do ator como se tivesse sido um envio feito via método `!`, porém é colocado junto a mensagem uma cópia da referência para o `CompletableFuture` criado no passo anterior;
3. É retornado para quem invocou o método uma outra cópia da referência para o `CompletableFuture` criado no passo 1;
4. Quando a mensagem terminou de ser processada, seu resultado é atualizado na referência que foi enviado junto à mensagem. Como a instância que foi teve o resultado "completado" (seja com um valor ou com uma exceção) é a mesma que é referenciada por quem fez a invocação do método, o resultado pode então ser utilizado.

Um envio feito via método `!!` possui quase os mesmos passos de processamento. A diferença está no retorno da referência. Como o envio é síncrono, a espera do resultado da computação assíncrona acontece de modo bloqueante dentro do próprio método `!!`. Quando a computação for completada, o resultado é retornado a quem fez a invocação do método.

Envios de mensagens não são limitados somente a atores. Qualquer objeto ou classe pode enviar uma mensagem a um ator. Podemos notar que o código da listagem 3.2 não está definido em um ator. O ator que está para enviar a mensagem de resposta pode verificar se há um remetente definido associado a mensagem recebida, porém existe um modo alternativo e mais simples: o método `self.reply_?` faz o envio ao remetente somente no caso em que há um remetente definido, devolvendo o valor boleano `true` para indicar se houve envio ou `false` caso contrário.

3.1.3 Hierarquias de supervisão

A biblioteca de atores do Akka permite que o tratamento de erros possa ser feito por atores definidos como supervisores. Sua implementação é totalmente baseada na abordagem feita na linguagem Erlang [Arm07] e é conhecida como "deixe que falhe" (*let it crash*).

Uma hierarquia de supervisão é criada com o uso de ligações entre atores. A ligação de dois atores pode acontecer via métodos `link`, `startLink` e `spawnLink`, definidos em `ActorRef`. O método `link` faz a ligação de dois atores $A1$ e $A2$. O método `startLink` também faz a ligação de dois atores $A1$ e $A2$, porém coloca o ator $A2$ em execução. O método `spawnLink` faz a criação do ator $A2$, sua ligação com $A1$ e ainda coloca $A2$ execução.

É necessário definir no ator que estará sob supervisão, qual ciclo de vida que esse ator deverá ter. O ciclo de vida de um ator indica ao ator supervisor como proceder no caso de erros. Existem dois tipos de ciclo de vida:

1. Permantente: O ciclo de vida é definido na classe `akka.config.Supervision.Permanent`. Indica que o ator possui um ciclo de vida permante e sempre deve ser reiniciado no caso de erros;
2. Temporário: O ciclo de vida é definido na classe `akka.config.Supervision.Temporary`. Indica que o ator possui um ciclo de vida temporário e não deve ser reiniciado no caso de erros. Contudo, esse ciclo de vida indica que o ator deve ser desligado pelo processo regular, ou seja, como se o método `exit` tivesse sido invocado. O processo regular ainda faz uma invocação ao método `postStop`.

A definição de um ator com ciclo de vida permanente é mostrada na listagem 3.6. A definição do ciclo de vida do ator acontece na linha 2.

O processo de reinicialização de um ator consiste na criação de uma nova instância da classe que define o ator. Os métodos `preRestart` e `postRestart` são invocados, respectivamente, antes de o ator ser desligado e logo após ele ter sido reiniciado. Alguns detalhes importantes a observarmos sobre a reinicialização de atores:

- O método `preRestart` é invocado na instância onde ocorreu o erro;
- O método `postRestart` é invocado na nova instância;
- A nova instância criada utiliza a mesma `ActorRef` do ator onde ocorreu o erro, logo, ambos os atores possuem o mesmo valor para `self`;
- A criação da nova instância é feita da mesma maneira que o ator foi criado originalmente. Por exemplo, na listagem 3.2 o ator foi originalmente criado via construtor padrão. Já na listagem 3.3, o ator foi originalmente criado com o uso de uma função específica.

No ator supervisor, por sua vez, deverá ser definida a estratégia de reinicialização para os atores que ficarão ligados a ele. Existem duas estratégias possíveis:

1. Um por um: É definida na classe `akka.config.Supervision.OneForOneStrategy`. Quando essa estratégia é utilizada, exceções que sejam lançadas em um ator sob supervisão que possua um ciclo de vida permanente. Faz com que somente o ator seja reiniciado;
2. Todos por um: É definida na classe `akka.config.Supervision.AllForOneStrategy`. Quando essa estratégia é utilizada, exceções que sejam lançadas em um ator sob supervisão, fazem com que todos os atores sob o mesmo supervisor sejam reiniciados.

```
1 class MySupervised extends akka.actor.Actor{
2  self.lifeCycle = akka.config.Supervision.Permanent
3
4  override def postRestart(reason: Throwable): Unit = {
5   // restaura o estado
6  }
7
8  override def preRestart(reason: Throwable): Unit = {
9   // salva o estado corrente
10 }
11
12 def receive = {
13  case msg => println("message received: %s".format(msg))
14 }
15 }
```

Listagem 3.6: *Ator com ciclo de vida permanente.*

Junto com a estratégida de reinicialização, é necessário definir quais são as exceções que o ator é responsável por fazer o tratamento, além de configurações relacionadas a quantas tentativas de reinicialização devem ser feitas dentro de um período de tempo. Mostramos na listagem 3.7 a definição de um ator supervisor. Na linha 2 dessa listagem definimos a estratégia um por um, supervisionando exceções do tipo `java.lang.Exception`, com duas tentativas de reinicialização em um período de cinco segundos. Caso o limite de tentativas de reinicialização tenha sido atingido, porém sem sucesso na reinicialização do ator supervisionado, uma mensagem específica é enviada para o ator supervisor. Essa mensagem pode estar definida no método `receive` para que as ações necessárias sejam tomadas.

```
1 class MySupervisor extends akka.actor.Actor {
2  self.faultHandler = akka.config.Supervision.OneForOneStrategy(List(
       classOf[Exception]), 2, 5000)
3
4  def receive = {
5   case _ => // ignora todas as mensagens
6  }
7 }
```

Listagem 3.7: *Ator supervisor.*

Apresentamos na listagem 3.8 a criação de uma hierarquia de supervisão entre os atores apresentados nas listagens 3.6 e 3.7.

```
1 object SampleSupervisorHierarchy extends Application {
2  val actorSupervisor = Actor.actorOf[MySupervisor].start
3  val supervised = Actor.actorOf[MySupervised]
4  actorSupervisor.startLink(supervised)
5 }
```

Listagem 3.8: *Criação da hierarquia de supervisão.*

Atores supervisores e supervisionados podem trocar mensagens. Quando criamos uma ligação de supervisão entre dois atores, como na linha 4 da listagem 3.8, a referência para o ator supervisor passa a estar definida no ator `supervised`. Essa referência é acessível via método `self.supervisor`. Um ator supervisor também pode ser supervisionado por outro ator, como mostrado na figura 3.2. O encadeamento de atores supervisores abre a possibilidade da criação de sub-hieraquias, cada uma com seu supervisor.

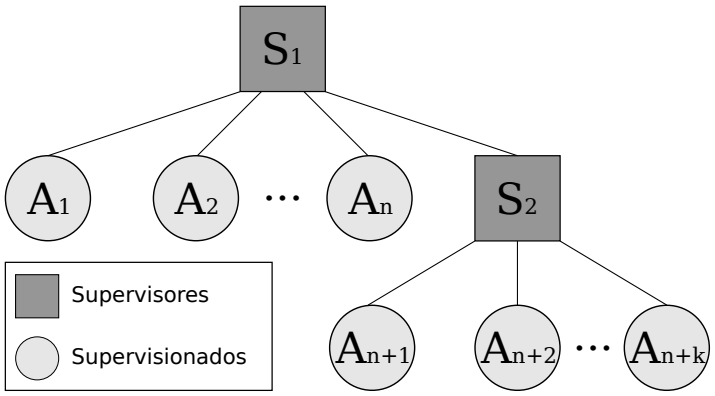

Figura 3.2: *Hierarquia de supervisão de atores.*

3.2 Atores remotos

No contexto de envios de mensagens para atores remotos, o termo cliente se refere à entidade que está enviando a mensagem. O termo servidor denota o processo (máquina virtual) no qual reside o ator remoto. Atores remotamente acessíveis possuem as mesmas características de atores locais no que diz respeito ao recebimento e despachamento de mensagens. Atores locais e remotos diferem basicamente na sua criação e no envio de mensagens. A infra-estrutura de atores remotos utiliza os seguintes elementos:

- `RemoteServerModule`: É uma *trait* que define as responsabilidades do componente usado no lado do servidor. Suas implementações têm como responsabilidade manter registrados os atores, bem como encaminhar a eles as mensagens recebidas de clientes remotos. Cada `RemoteServerModule` é associado a um endereço de hospedeiro (*host*) e a uma porta TCP. Um mesma máquina virtual pode conter múltiplos `RemoteServerModules`. A documentação do Akka utiliza para esse componente a terminologia "servidor remoto" (*remote server*);

- `RemoteClientModule`: É uma *trait* que define as responsabilidades do componente usado no lado do cliente. Suas implementações têm como responsabilidade visível oferecer uma interface para obtenção de referências a atores remotos. Oferece também suporte em tempo de execução para a infra-estrutura de atores do lado do cliente, provendo uma série de serviços não vísiveis para o usuário, tais como seriação de mensagens, envio de mensagens para atores remotos, conversão de ator local em ator remoto e intermediação de mensagens de resposta vindas do `RemoteServerModule`, no caso envios via `!!` ou `!!!`. A documentação do Akka utiliza para esse componente a terminologia "cliente remoto" (*remote client*);

- `RemoteSupport`: É acessível via `Actor.remote`. Essa classe abstrata é responsável por concentrar as responsabilidades definidas para os módulos remotos do cliente e do servidor e ser um ponto único de suporte remoto;

- `RemoteActorRef`: É uma classe equivalente a `LocalActorRef`, porém utiliza o suporte remoto para fazer envio das mensagens.

A implementação de atores remotos do Akka, implementa os componentes de suporte remoto com o auxílio do JBoss Netty. Apresentamos na figura 3.3 como os componentes se relacionam. Dividimos a figura em duas partes, sendo a camada de interface remota que é a camada que se interage diretamente, e a camada de implementação. A camada de implementação é associada em tempo de execução, permitindo que outras implementações criadas possam ser utilizadas.

Na versão 1.0 do Akka, a definição de atores remotos não possui diferença em relação a definição de atores locais. Podemos utilizar a classe apresentada na listagem 3.1 para criamos instâncias de atores remotos. A criação de atores remotos pode acontecer tanto no nó onde reside a aplicação servidora, quanto ser criado remotamente por uma aplicação cliente.

A criação de atores remotos gerenciados pelo servidor (*server managed actors*) é mostrada na listagem 3.9. Nessa listagem, um servidor remoto é inicializado na linha 2 para que os atores possam ser registrados. Na implementação padrão feita com o Netty, a inicialização de um servidor remoto implica na associação de um componente do Netty para monitorar a *socket* associada ao par *host* e porta. Na linha 3 registramos o ator sob o nome `hello-service`. O ator passou a estar remotamente acessível a aplicações que residam em outras máquinas virtuais. Um detalhe importante a ser destacado é a inicialização implícita do ator feita pelo método `register`. A implementação desse método fica a critério dos componentes da camada de implementação de suporte remoto, porém sua interface obriga que implementações compatíveis inicializem atores que ainda não estão em execução.

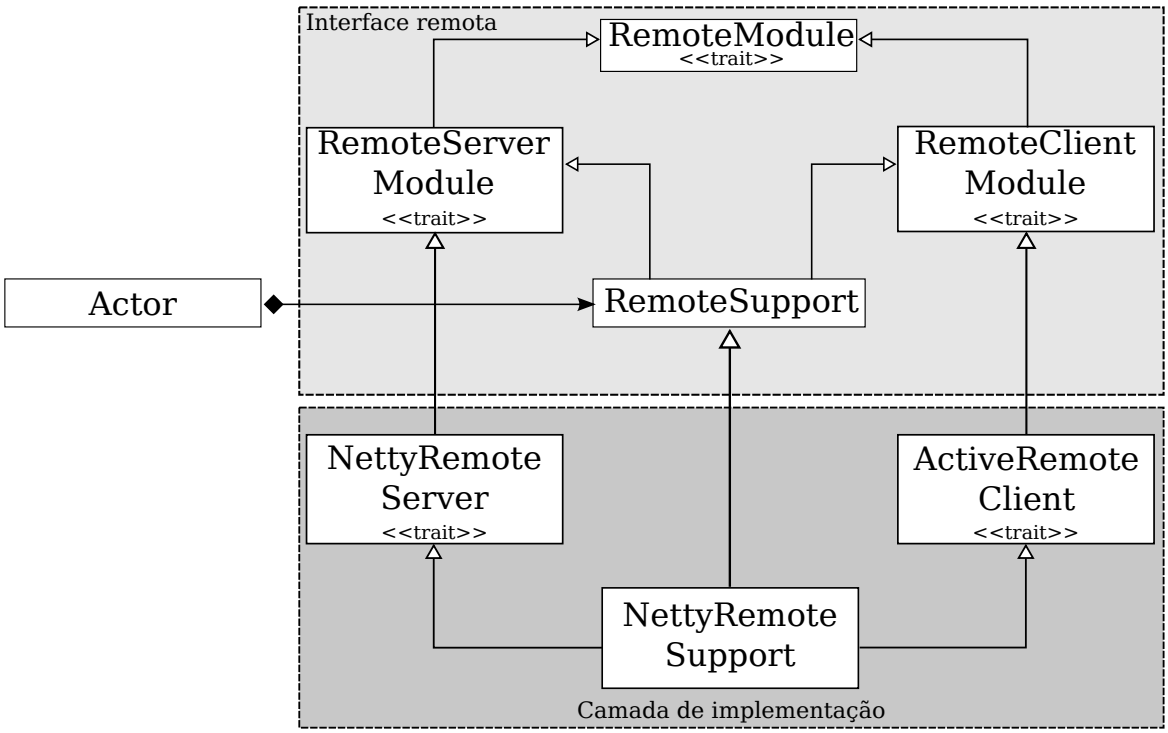

Figura 3.3: *Relacionamento entre os componentes remotos.*

Referências para atores remotos podem ser obtidas via método `actorFor`. Esse método possui diversas sobrecargas, sendo que a mais simples recebe como argumentos o nome do ator, o *host* e a porta onde foi feito o registro. A linha 2 da listagem 3.10 mostra como uma aplicação cliente obtém uma referência para o ator. Podemos notar que o uso do ator na linha 3 não possui nenhuma indicação explícita sobre um envio remoto. O método `actorFor` retorna na maioria das vezes instâncias de `RemoteActorRef`, que representam *proxies* locais para o ator remoto. A implementação possui algumas otimizações para que referências locais sejam utilizadas sempre que possível. Por exemplo, a entidade que está enviando a mensagem reside no mesmo nó que o ator. Em casos como esse, uma instância de `LocalActorRef` é retornada.

```
1 object SampleRemoteServer extends Application {
2   Actor.remote.start("localhost", 2552)
3   Actor.remote.register("hello-service", Actor.actorOf[SampleActor])
4 }
```

Listagem 3.9: *Aplicação SampleRemoteServer.*

```
1 object SampleRemoteClient extends Application{
2   val helloActor = Actor.remote.actorFor("hello-service", "localhost",
        2552)
3   helloActor ! "Hello"
4 }
```

Listagem 3.10: *Aplicação SampleRemoteClient.*

Aplicações cliente podem optar por criar e iniciar atores em nós que não sejam o corrente. Esses atores são chamados de atores gerenciados pelo cliente (*client managed actors*). Para tal, a biblioteca de atores fornece uma versão alternativa do método `actorOf` que recebe parâmetros adicionais para indicar o nó onde o ator criado será executado. O Akka não possui carregamento remoto de classes, obrigando que as classes com as definições dos atores estejam acessíveis para a máquina virtual onde ele será instanciado e executado. Uma outra observação importante a ser feita em relação ao servidor remoto que irá executar o ator sendo criado, é a necessidade de ele estar em execução antes de o método `actorOf` ser invocado. Os detalhes observados são alguns dos exemplos que motivam discussões entre a comunidade de desenvolvedores do Akka para que esse tipo de suporte seja depreciado e removido em versões futuras, já que os mesmos resultados podem ser obtidos com o uso de atores gerenciados pelo servidor.

3.2.1 Fluxo de envio das mensagens

O envio assícrono de uma mensagem a um ator remoto tem alguns passos adicionais em relação a um envio local de mensagens. O processo de um envio assíncrono de uma mensagem para um ator remoto é ilustrado na figura 3.4 e acontece em sete passos:

1. O processo começa com o *proxy* local embrulhando a mensagem e adicionando a ela informações de cabeçalho necessárias para o envio e processamento posterior. A mensagem embrulhada e com as informações de cabeçalho é entao repassada ao seriador;
2. O seriador é responsável por converter a informação recebida em um vetor de *bytes* para que o transporte possa ocorrer. Uma vez que a informação esteja no formato a ser transportado, o *proxy* usa uma implementação de `RemoteSupport` (que na figura 3.4 é uma instância de `NettyRemoteSupport`) para enviar a mensagem ao `RemoteSupport` que está no lado do servidor;
3. A classe `ClientBootstrap` transporta a mensagem para o `ServerBootstrap` que está conectado. O processo de envio, do ponto de vista do cliente leva tempo $O(1)$, já que o transporte da mensagem pelo JBoss Netty entre o `ClientBootstrap` e o `ServerBootstrap` é feito de modo assíncrono;
4. A classe `ServerBootstrap` repassa a mensagem ao *handler* que está associado a ele;
5. A implementação do *handler* feita no Akka repassa a mensagem para o seriador para que a deseriação seja feita;
6. O seriador desseria a mensagem para o formato que ela estava antes de ser seriada para ser enviada. Em seguida, o seriador repassa novamente a mensagem para o *handler*;
7. Por fim, a implementação do *handler* utiliza as informações definidas de cabeçalho da mensagem para identificar qual o ator destinatário no registro local de atores e como deve ser feito

o envio. No caso do envio assíncrono, o ator destinatário tem seu método `!` invocado com a mensagem e eventualmente a referência do ator que fez o envio (ou `None` caso contrário). O despachador associado ao ator tem o mesmo comportamento já mostrado na figura 3.1.

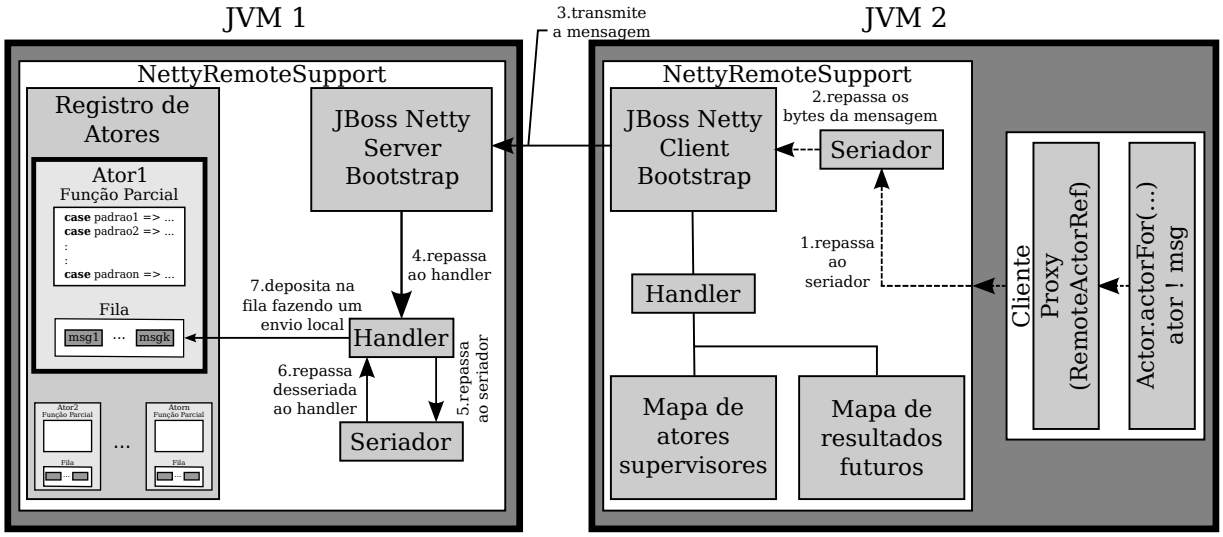

Figura 3.4: *Fluxo de envio de mensagens para atores remotos.*

Os envios feitos via métodos `!!` e `!!!` para atores remotos são iguais, e possuem um passo a mais do que os mostrados na figura 3.4. O passo adicional acontece entre os passos 2 e 3. Antes de a mensagem ser enviada uma cópia da referência para instância de resultado futuro que é retornada é colocada no mapa de resultados futuros. A chave para a instância é o identificador da mensagem.

Envios feitos via métodos `!!` e `!!!`, assim como mensagens para atores supervisores que estão remotos, geram envios de mensagens no caminho contrário ao mostrado na figura 3.4. Caso a mensagem que foi enviada pelo servidor seja o resultado de um envio com `!!` ou `!!!`, o conteúdo da mensagem (seja um resultado de sucesso ou uma exceção) é utilizado para completar a instância cujo a referência havia sido colocada no mapa de resultados futuros.

No caso de mensagens originadas no servidor que possuam um ator supervisor definido, o ator supervisor é localizado e é feito um envio local assíncrono da mensagem para ele.

O Akka possui uma configuração opcional de segurança onde cada JVM que esteja executando o arcabouço pode definir um *cookie*. O *cookie* nada mais é do que uma chave que o usuário pode gerar e que é parte das informações de cabeçalho das mensagens. Quando a verificação de segurança está ativada, o recebedor da mensagem verifica se a informação do *cookie* presente na mensagem confere com a informação esperada. Essa verificação ocorre entre os passos 6 e 7 da figura 3.4 e caso os valores sejam iguais, o passo 7 será executado. Caso contrário, uma exceção de segurança é lançada na JVM que recebeu a mensagem e o passo 7 não é executado.

O Netty oferece algumas opções específicas para o transporte de mensagens baseadas em *socket*s TCP. Exemplos dessas opções são a definição do tamanho máximo da janela utilizada para o envio das mensagens a atores remotos, o tempo limite de espera para leitura na *socket*, o intervalo de espera para tentativa de reconexão e o intervalo máximo em que o cliente deve tentar se conectar. Além dessas opções, o Netty fornece ainda a possibilidade de compactação das mensagens durante o envio. O Akka faz uso de todas as opções listadas, permitindo que os usuários da biblioteca definam os valores desejados. As configurações de todos os módulos do Akka são feitas no arquivo `akka.conf`.

O arquivo `akka.conf` possui é o ponto de entrada para todas as configurações que são parametrizadas de todos os módulos do projeto Akka (o formato do arquivo difere dos arquivos para

configurações de propriedades baseados em chaves e valores. O projeto Akka utiliza uma biblioteca chamada Configgy [Poi] que define um formato mais idiomático para arquivos de configurações). Existe uma seção específica para as configurações correspondentes aos atores remotos. Na distribuição padrão, boa parte das propriedades para a configuração de atores remotos são relacionadas ao JBoss Netty, como por exemplo o tipo nível de compressão, tamanho da janela da mensagem e tempo limite de espera pelo cliente para se conectar. Contudo, existem algumas propriedades que são mais gerais, como por exemplo se a comunicação deve ser autenticada via *cookies* pré-definidos, qual o *cookie* de segurança a ser utilizado e ainda qual a classe que deve ser utilizada como implementação da camada de suporte para atores remotos.

Os dados informados no arquivo `akka.conf`, uma vez que foram lidos ficam acessíveis em objetos de acordo com o contexto que fazem parte. Por exemplo, as propriedades relacionadas aos atores remotos estão organizadas em objetos dentro do módulo `RemoteShared`. Existe ainda uma classe auxíliar que é utilizada para acessar os módulos dos sub-projetos do Akka via reflexão chamada de `ReflectiveAccess`. A camada de implementação para o transporte de mensagens entre atores remotos é instanciada nessa classe. A seção para se informar as configurações do módulo de atores remotos é mostrada na listagem 3.11.

```
1 ...
2 remote {
3    secure-cookie = "050E0A0D0D0601AA00B0090B..."
4    compression-scheme = "zlib"
5    zlib-compression-level = 6
6    layer = "akka.remote.netty.NettyRemoteSupport"
7
8    server {
9      hostname = "localhost"
10     port = 2552
11     message-frame-size = 1048576
12     connection-timeout = 1
13     require-cookie = on
14     untrusted-mode = off
15   }
16
17   client {
18     reconnect-delay = 5
19     read-timeout = 10
20     message-frame-size = 1048576
21     reconnection-time-window = 600
22   }
23 }
```

Listagem 3.11: *Seção remote do arquivo akka.conf.*

3.2.2 Protocolo para envios de mensagens a atores remotos

O protocolo utilizado para envios de mensagens entre atores remotos é definido no Akka com o uso da biblioteca Protobuf.

Protobuf [Goo] é uma biblioteca que permite que dados estruturados sejam representados em um formato eficiente e extensível. A biblioteca possui um compilador que converte a definição de protocolos em classes Java, Python e C++, permitindo uma interação direta com o protocolo nas linguagens, eliminando a necessidade de bibliotecas adicionais para conversões. A bibliteca permite que protocolos sejam definidos com a palavra-chave `message`. Protocolos podem contér outros protocolos e tipos primitivos de dados, como por exemplo números, textos, boleanos e vetores de

dados. Ademais, pode-se definir a obrigatoriedade dos valores para alguns atributos com as palavras-chaves `optional` e `required`.

O protocolo das mensagens é mostrado na listagem 3.12. Uma mensagem é definida pelo tipo de seriação (linha 1) que foi utilizado na codificação dos dados da mensagem (listagem 3.13), pelos bytes da mensagem (linha 2) e por um atributo que é utilizado para informar o nome da classe da mensagem. As mensagens para atores remotos são definidas com base no tipo de seriação que foi definido na JVM de onde partiu o envio. Para que a JVM que esteja recebendo a mensagem possa fazer o processamento correto, a informação do tipo de seriação deve estar presente. O atributo de manifesto da mensagem (linha 3) é opcional, porém é utilizado para forçar o carregamento da classe no *class loader* onde a mensagem será desseriada.

```
1 message MessageProtocol {
2   required SerializationSchemeType serializationScheme = 1;
3   required bytes message = 2;
4   optional bytes messageManifest = 3;
5 }
```

Listagem 3.12: *Protocolo para mensagens.*

```
1 enum SerializationSchemeType {
2   JAVA = 1;
3   SBINARY = 2;
4   SCALA_JSON = 3;
5   JAVA_JSON = 4;
6   PROTOBUF = 5;
7 }
```

Listagem 3.13: *Enumeração com tipos de seriação suportados.*

Pelo fato de as referências remotas dos atores que estão enviando uma mensagem serem enviadas junto à mensagem foi necessário a definição de um protocolo para as `RemoteActorRefs`. O protocolo é mostrado na listagem 3.14. O protocolo define o nome que é utilizado para identificar o ator (linha 1), a classe que foi utilizada para a criação do ator (linha 2), o endereço de onde o ator remoto foi criado (linha 3) e por fim um valor opcional (linha 4), que indica o tempo máximo que o ator deve ficar bloqueado durante um envio síncrono. O protocolo `AddressProtocol` utilizado na linha 3 é definido pelos atributos endereço do hospedeiro e porta.

```
1 message RemoteActorRefProtocol {
2   required string classOrServiceName = 1;
3   required string actorClassname = 2;
4   required AddressProtocol homeAddress = 3;
5   optional uint64 timeout = 4;
6 }
```

Listagem 3.14: *Protocolo para referências de atores remotos.*

Por fim, o protocolo utilizado para as mensagens remotas que é composto, dentre outros protocolos menores, pelo protocolo para mensagens (linha 4) e pelo protocolo para referências remotas (linha 7), é apresentado na listagem 3.15. Na linha 1, o atributo `uuid` é um identificador único para as mensagens. A implementação do protocolo `UuidProtocol` utiliza a biblioteca UUID [Bur] que é uma implementação de identificadores globais únicos. Podemos notar na linha 4 um atributo boleano que é utilizado para indicar se é esperado uma mensagem de resposta para a mensagem que está sendo processada. Envios feitos via método `!` possuem valor `oneWay = true`, enquanto que os demais envios não.

```
1 message RemoteMessageProtocol {
2   required UuidProtocol uuid = 1;
3   required ActorInfoProtocol actorInfo = 2;
4   required bool oneWay = 3;
5   optional MessageProtocol message = 4;
6   optional ExceptionProtocol exception = 5;
7   optional UuidProtocol supervisorUuid = 6;
8   optional RemoteActorRefProtocol sender = 7;
9   repeated MetadataEntryProtocol metadata = 8;
10  optional string cookie = 9;
11 }
```

Listagem 3.15: *Protocolo para envios de mensagens remotas.*

Optamos por não detalhar os demais sub-protocolos utilizados para compor o protocolo para mensagens remotas pois, apesar de serem importantes para o suporte a atores remotos, eles não possuem grande relevância para o desenvolvimento deste trabalho.

3.2.3 Seriação de mensagens e de referências remotas

Os diferentes tipos de seriação suportados pelo projeto Akka são utilizados na seriação da mensagem (listagem 3.12) e opcionalmente na seriação da referência remota (listagem 3.14). O módulo `Serializable` define um conjunto de *traits* que podem ser utilizadas para trocar a seriação Java padrão por algum dos outros formatos suportados.

O módulo `Serializer` define um objeto chamado de `MessageSerializer` que é o responsável para seriação e desseriação das mensagens. A seriação das mensagens acontece com base no tipo da mensagem. Por exemplo, se a instância utilizada como argumento para os métodos de envio de mensagens implementar a *trait* `ScalaJSON`, o conteúdo binário da mensagem estará no formato JSON.

A seriação de `RemoteActorRefs` é suportada por padrão e é utilizada no envio implícito do ator remetente (quando existe um ator remetente) junto à mensagem. Contudo, a seriação do estado ou mesmo da definição do ator (chamada de *deep serialization*) deve, quando necessária, ser implementada pelo usuário da biblioteca. A seriação completa do ator é útil quando desejamos realocar o ator em um nó diferente do atual.

Capítulo 4

O Padrão AMQP

AMQP [Gro] (*Advanced Message Queuing Protocol*) é um protocolo aberto para sistemas corporativos de troca de mensagens. Especificado pelo AMQP *Working Group*, o protocolo permite completa interoperabilidade para *middleware* orientado a mensagens. A especificação [AMQ08] define não somente o protocolo de rede, mas também a semântica dos serviços da aplicação servidora. Também é parte do foco que capacidades providas por sistemas de *middleware* orientados a mensagem possam estar pervasivamente disponíveis nas redes das empresas, incentivando o desenvolvimento de aplicações interoperáveis baseadas em troca de mensagens.

AMQP é um protocolo binário, assíncrono, seguro, portável, neutro, eficiente e possui suporte a múltiplos canais. A especificação apresenta o protocolo dividido em três camadas, como mostrado na figura 4.1.

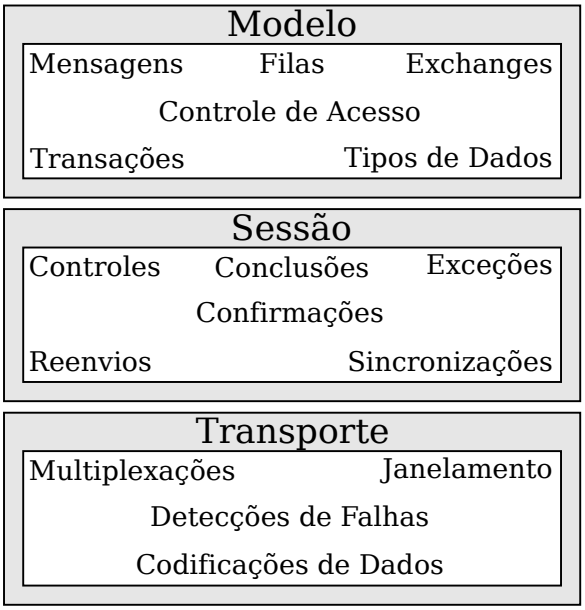

Figura 4.1: *Camadas do padrão AMQP.*

Na seção 4.1 descrevemos a camada de modelo, que é a camada onde é definido o conjunto de objetos que utilizamos em nosso trabalho. Na seção 4.2 descrevemos camada de sessão que age como intermediária entre as camadas de modelo e transporte. Comentamos brevemente na seção 4.3 sobre a camada de transporte. Os detalhes dessa camada não se enquadram no escopo deste trabalho. Por fim, apresentamos na seção 4.4 algumas das implementações disponíveis do padrão e qual implementação escolhemos para o desenvolvimento deste trabalho.

4.1 A camada de modelo

A camada de modelo é a camada onde está definido o conjunto de comandos e dos objetos que as aplicações podem utilizar. A especificação dos requisitos dessa camada inclui, entre outros itens: garantir a interoperabilidade das implementações, prover controle explícito sobre a qualidade do serviço, proporcionar um mapeamento fácil entre os comandos e as bibliotecas de nível de aplicação e ter clareza, de modo que cada comando seja responsável por uma única ação. Os componentes são:

- Servidor: É o processo, conhecido também como *broker*, que aceita conexões de clientes e implementa as funções de filas de mensagens e roteamento;
- Filas: São entidades internas do servidor que armazenam as mensagens, tanto em memória quanto em disco, até que elas sejam enviadas em sequência para as aplicações consumidoras. As filas são totalmente independentes umas das outras. Na criação de uma fila, várias propriedades podem ser especificadas: a fila de ser pública ou privada, armazenar mensagens de modo durável ou transiente, e ter existência permanente ou temporária (e.g.: a existência da fila é vinculada ao ciclo de vida de uma aplicação consumidora). A combinação de propriedades como essas viabiliza a criação de diversos tipos de fila, como por exemplo: fila armazena-e-encaminha (*store-and-forward*), que armazena as mensagens e as distribui para vários consumidores na forma *round-robin*, fila temporária para resposta, que armazena as mensagens e as encaminha para um único consumidor; e fila *pub-sub*, que armazena mensagens provenientes de vários produtores e as envia para um único consumidor;
- *Exchanges*: São entidades internas do servidor que recebem e roteiam as mensagens das aplicações produtoras para as filas, levando em conta critérios pré-definidos. Essas entidades inspecionam as mensagens, verificando, na maioria dos casos, a chave de roteamento presente no cabeçalho de cada mensagem. Com o auxílio da tabela de *bindings*, uma *exchange* decide como encaminhar as mensagens às respectivas filas, jamais armazenando mensagens. *Exchanges* podem ser: diretas (*direct*), onde o valor da chave de roteamento, que é parte do cabeçalho da mensagem, deve ser exatamente igual a uma entrada da tabela de *bindings* para que a mensagem seja roteada para a fila; tópicos (*topic*), onde é utilizado o casamento de padrões para determinar o roteamento às filas. Os caracteres coringa suportados são *, que indica uma única palavra, e #, que indica zero ou muitas palavras. A chave de roteamento deve ser formada por palavras e pontos. Por exemplo, o padrão *.stock.# casa com usd.stock e eur.stock.db, nas não com stock.nasdaq; e *fanout*, onde o roteamento acontece para todas as filas associadas a *exchange*, indepententemente da chave de roteamento;
- *Bindings*: São relacionamentos entre *exchanges* e filas. Esses relacionamentos definem como deverá ser feito o roteamento das mensagens;
- *Virtual hosts*: São coleções de *exchanges*, filas e objetos associados. *Virtual hosts* são domínios independentes no servidor e compartilham um ambiente comum para autenticação e segurança. As aplicações clientes escolhem um *virtual host* após se autenticarem no servidor.

A figura 4.2 mostra os componentes acima descritos e os relacionamentos entre esses componentes.

Em modelos pré-AMQP, as tarefas das *exchanges* e das filas eram feitas por blocos monolíticos que implementavam tipos específicos de roteamento e armazenamento. O padrão AMQP separa essas tarefas e as atribui a entidades distintas (*exchanges* e filas), que têm os seguintes papéis: *(i)* receber as mensagens e fazer o roteamento para as filas; *(ii)* armazenar as mensagens e fazer o encaminhamento para as aplicações consumidoras. Vale frisar que filas, *exchanges* e *bindings* podem ser criados tanto de modo programático como por meio de ferramentas administrativas.

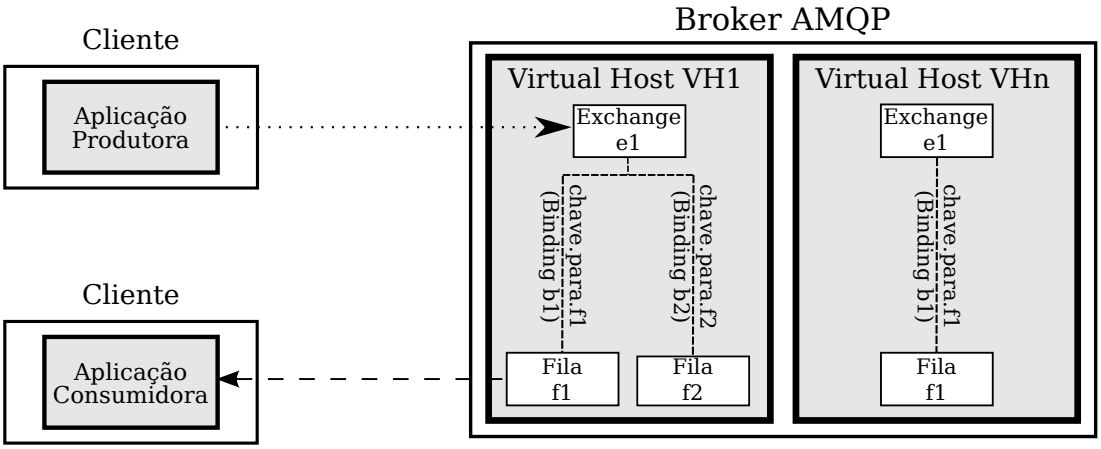

Figura 4.2: *Componentes da camada de modelo do padrão AMQP.*

Há uma analogia entre o modelo AMQP e sistemas de email:

1. Uma mensagem AMQP é análoga a uma mensagem de *email*.
2. Uma fila é análoga a uma caixa de mensagens.
3. Um consumidor corresponde a um cliente de *email* que carrega e apaga as mensagens.
4. Uma *exchange* corresponde a um *mail transfer agent* (MTA) que inspeciona as mensagens e, com base nas chaves de roteamento, verifica as tabelas de registro e decide como enviar as mensagens para uma ou mais caixas de mensagens. No caso do correio eletrônico as chaves de roteamento são os campos de destinatário e cópias (To, Cc e Bcc).
5. Um *binding* corresponde a uma entrada nas tabelas de roteamento do MTA.

4.1.1 Envios de mensagens

Para enviar uma mensagem, uma aplicação produtora especifica uma determinada *exchange* de um determinado *virtual host*, uma rotulação com informação de roteamento e eventualmente algumas propriedades adicionais, bem como os dados do corpo da mensagem. Uma vez que a mensagem tenha sido recebida no servidor AMQP, ocorre o roteamento para uma ou mais filas do conjunto de filas do *virtual host* especificado. No caso de não ser possível rotear a mensagem, seja qual for o motivo, as opções são: rejeitar a mensagem, descartá-la silenciosamente, ou ainda fazer o roteamento para uma *exchange* alternativa. A escolha depende do comportamento definido pelo produtor. Quando a mensagem é depositada em alguma fila (ou possivelmente em algumas filas), a fila tenta repassá-la imediatamente para a aplicação consumidora. Caso isso não seja possível, a fila mantém a mensagem armazenada para uma futura tentativa de entrega. Uma vez que a mensagem foi entregue com sucesso a um consumidor, ela é removida da fila. A figura 4.3 mostra o envio e recebimento de uma mensagem.

A aceitação ou confirmação de recebimento de uma mensagem fica a critério da aplicação consumidora, podendo acontecer imediatamente depois da retirada da mensagem da fila ou após a aplicação consumidora ter processado a mensagem.

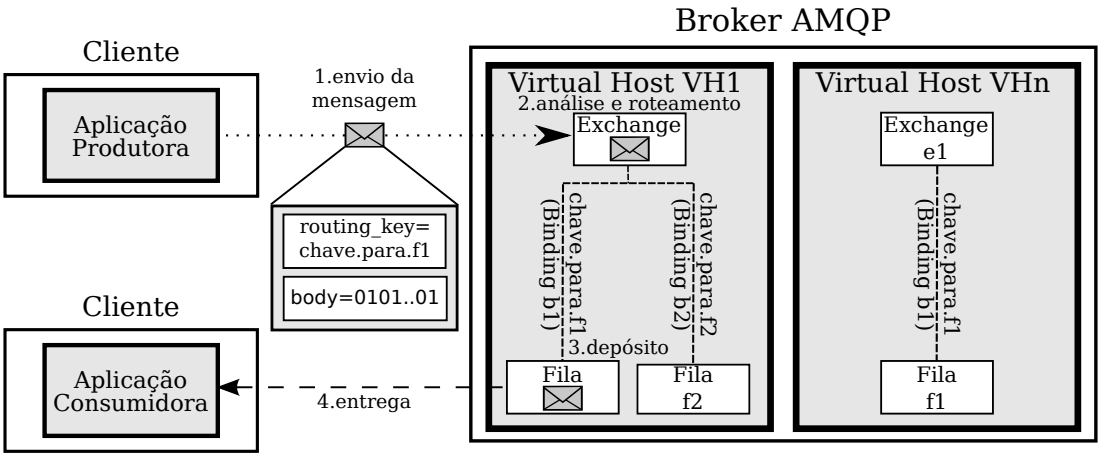

Figura 4.3: *Fluxo de uma mensagem no padrão AMQP.*

4.2 A camada de sessão

A camada de sessão age como uma intermediária entre as camadas de modelo e transporte, proporcionando confiabilidade para a transmissão de comandos ao *broker*. É parte das suas responsabilidades dar confiabilidade às interações entre as aplicações cliente e o *broker*.

Sessões são interações nomeadas entre um cliente e um servidor AMQP (também chamados de pares (*peers*)). Todos os comandos, como por exemplo envios de mensagens e criações de filas ou *exchanges*, devem acontecer no contexto de uma sessão. O ciclo de vida de alguns dos objetos da camada de modelo como filas, *exchanges* e *bindings* podem ser limitados ao escopo de uma sessão.

Os principais serviços providos pela camada de sessão para a camada de modelo são:

- Identificação sequencial dos comandos: Cada comando emitido pelos pares é identificado, única e individualmente, dentro da sessão para que o sistema seja capaz de garantir sua execução exatamente uma vez. Utiliza-se um esquema de numeração sequencial. A noção de identificação permite a correlação de comandos e o retorno de resultados assíncronos. O identificador do comando é disponibilizado para a camada de modelo e, quando um resultado é retornado para um comando, o identificador desse comando é utilizado para estabelecer a correlação entre o comando e o resultado;

- Confirmação que comandos serão executados: É utilizado para que o par solicitante possa, seguramente descartar o estado associado a um comando com a certeza de que ele será executado. A camada de sessão controla o envio e recebimento das confirmações permitindo o gerenciamento do estado a ser mantido na sessão corrente. O estado da sessão é importante para que o sistema possa se recuperar no caso de falhas temporárias em um dos pares. As confirmações podem ser entregues em lotes ou mesmo serem deferidas indefinidamente, no caso do par solicitante não requerer a confirmação de que o comando será executado;

- Notificação de comandos completados: Diferente do conceito de confirmação, este serviço notifica o par que solicitou a execução de um comando que o comando foi executado por completo. As notificações de comandos completados tem como motivação a sincronização e a garantia da ordem de execução entre diferentes sessões. Quando o par que solicitou a execução do comando não exige confirmação imediata, as confirmações podem ser acumuladas e enviadas em lotes, reduzindo o tráfego de rede;

- Reenvio e recuperação no caso de falhas de rede: Para que o sistema possa se recuperar no caso de falhas de rede, a sessão deve ser capaz de reenviar comandos cujos recebimentos pelo

outro par são duvidosos. A camada de sessão provê as ferramentas necessárias para identificar o conjunto com os comandos rotulados como duvidosos e reenviá-los, sem o risco de causar duplicidade.

4.3 A camada de transporte

A camada de transporte é responsável por tarefas como multiplexação de canais, detecção de falhas, representação de dados e janelamento (*framing*). A lista dos requisítos dessa camada inclui, entre outros itens: possuir uma representação de dados binária e compacta que seja rápida de se embrulhar e desembrulhar, trabalhar com mensagens sem um limite significante de tamanho, permitir que sessões não sejam perdidas no caso de falhas de rede ou de aplicação, possuir assincronidade e neutralidade em relação à linguagens de programação.

4.4 Implementações

Dentre as implementações de *message brokers* baseados no padrão AMQP disponíveis, destacamos Apache Qpid [Qpi], ZeroMQ [Zer] e RabbitMQ [RMQ].

O projeto Apache Qpid é uma implementação de código aberto com uma distribuição escrita em Java e uma outra escrita em C++. A implementação está disponível sob a licença Apache 2.0 [APL04] e possui bibliotecas para aplicações cliente em diversas linguagens, como Java, C++, Ruby, Python e C#.Net. A implementação da biblioteca cliente para Java é compatível com o a versão 1.1 do padrão Java *Message Service* [Pro03].

O projeto ZeroMQ também é uma implementação de código aberto feita em C++, com bibliotecas disponibilizadas para aplicações cliente em mais de 20 linguagens, incluindo Java, Python, C++ e C. A implementação está disponível sob a licença LGPL [LGP07]. A biblioteca para clientes Java depende de outras bibliotecas nativas específicas para sistema operacional suportado.

Assim como os outros dois projetos, o projeto RabbitMQ também é um projeto de código aberto, porém é implementado na linguagem Erlang e está disponível sob a licença MPL [MPL]. Possui bibliotecas para aplicações cliente em diversas linguagens como Java, Erlang e Python. A implementação da biblioteca para clientes Java é totalmente feita em Java e não possui dependências de código nativo, sendo totalmente independente de plataforma.

Como a especificação do padrão AMQP não define uma API padrão para as aplicações clientes, tivemos de optar por uma implementação para o desenvolvimento deste trabalho. Optamos por utilizar o projeto RabbitMQ. Tomamos como base para nossa decisão alguns fatores, a destacar:

- Grande quantidade de atividade na comunidade de usuários e desenvolvedores;
- Facilidade para se obter informações, seja via tutoriais ou em listas de discussões;
- Possuir código aberto;
- Biblioteca para clientes Java independente de plataforma;
- Se tratar não só de um projeto, mas sim de um produto[1];
- Ter sido escrito em Erlang, uma linguagem desenvolvida para o desenvolvimento de sistemas distribuídos e concorrentes [Arm97];
- Possuir ferramentas para administração e monitoramento.

[1] A empresa responsável pelo suporte é a Spring Source, uma divisão da VMware

Apresentamos a seguir uma visão geral de alguns dos principais componentes da biblioteca para clientes Java do projeto RabbitMQ e como eles se relacionam. Mostramos também uma aplicação produtor/consumidor de exemplo escrita na linguagem Scala.

4.4.1 RabbitMQ - Biblioteca para clientes Java

A biblioteca para clientes Java disponibilizada pelo projeto RabbitMQ [Jav] define diversas classes para que seja possível a interação com o *broker*. A conexão acontece com uma instância de `com.rabbitmq.client.Connection`, que pode ser obtida pela classe `com.rabbitmq.client.ConnectionFactory`, via método `newConnection`. É na classe `ConnectionFactory` que informamos os valores dos parâmetros necessários para conexão com o *broker* (listagem 4.1).

```
1 val factory = new ConnectionFactory()
2 factory.setHost(...)
3 factory.setPort(...)
4 factory.setUsername(...)
5 factory.setPassword(...)
6 factory.setVirtualHost(...)
7 val connection = factory.newConnection
```

Listagem 4.1: *Configuração e uso de uma ConnectionFactory.*

As conexões têm como papel principal estabelecer a comunicação da aplicação cliente com o *broker*. Conexões podem ser vistas como sendo a implementação de uma parte considerável da camada de transporte do padrão AMQP. Como descrito na seção 4.3, a camada de transporte deve ser capaz de prover a multiplexação de canais. A classe `Connection` provê, dentre outros métodos, o método `createChannel`. A invocação desse método resulta em uma instância de `com.rabbitmq.client.Channel` por onde podemos executar os comandos definidos na camada de sessão (listagem 4.2). Por exemplo a criação de filas (`queueDeclare`) ou o envio (`basicPublish`) e recebimento de mensagens (`basicConsume`).

```
1 val channel = connection.createChannel
2 channel.exchangeDeclare(...)
3 channel.queueDeclare(...)
4 channel.queueBind(...)
5 channel.basicPublish(...)
```

Listagem 4.2: *Criação e uso de canais.*

O recebimento das mensagens é feito com o uso de consumidores que são instâncias de `com.rabbitmq.client.Consumer`. Os consumidores são registrados nos canais e podem receber as mensagens assincronamente através do método `handleDelivery` (listagem 4.3).

A figura 4.4 mostra como uma fábrica de conexões, uma conexão, um canal e um consumidor se relacionam. Um exemplo completo de uma aplicação produtora e de uma aplicação consumidora com a biblioteca Java do RabbitMQ é apresentado no apêndice A.

Um detalhe importante sobre a biblioteca para clientes Java do RabbitMQ, é a verificação da existênca de um objeto antes da criação efetiva do objeto. Por exemplo, se tivéssemos criado uma fila durável F_1, e tentássemos criar uma nova fila durável F_1, a fila não seria recriada e a execução seguiria normalmente. Eventuais mensagens em F_1 não sofreriam alterações. Contudo, caso estivéssemos tentando recriar F_1 como não durável, uma exceção seria lançada e o canal utilizado para executar o comando de criação seria fechado. A regra também vale para *exchanges*. O procedimento correto é remover, seja via código ou via alguma aplicação administrativa do RabbitMQ, para depois fazer a criação.

```
1 class MyConsumer(channel: Channel) extends Consumer{
2 ...
3   def registerConsumer(queue: String, autoAck: Boolean): Unit = {
4      channel.basicConsume(queue, autoAck, this)
5   }
6   @Override
7   def handleDelivery(consumerTag: String, envelope: Envelope, properties:
        BasicProperties,
8                message: Array[Byte]): Unit = {
9     // processamento da mensagem
10  }
11 ...
12 }
```

Listagem 4.3: *Registro de um consumidor.*

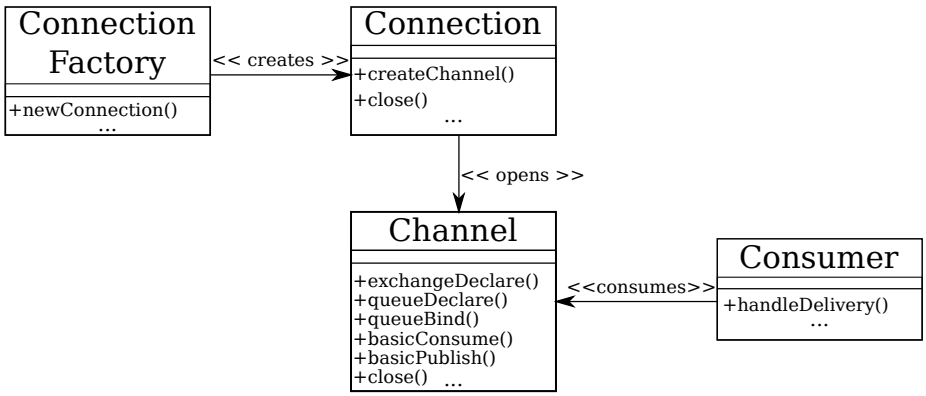

Figura 4.4: *RabbitMQ Java API – Relacionamento entre classes de transporte e sessão.*

Para finalizar este capítulo, é importante destacar que a implementação de canais [API] não é uma segura para acesso concorrente (*thread safe*), de modo que é de responsabilidade da aplicação fazer a devida proteção para evitar que diferentes *threads* façam uso dos canais concorrentemente. Canais permitem ainda que sejam registrados tratadores de erros (`setReturnListener`), como por exemplo para o caso em que mensagens não puderem ser enviadas ou entregues aos seus respectivos destinatários. Em situações como essas, o corpo da mensagem e algumas informações do envio, como o nome da *exchange*, a chave de roteamento e o código de rejeição, são repassados para o tratador de erros que foi registrado no canal. A aplicação produtora passa a ter autonomia para fazer o tratamento apropriado.

Capítulo 5

Troca de mensagens entre entidades remotas via broker AMQP

Apresentamos nesse capítulo a estrutura que definimos para a troca de mensagens entre entidades remotas via *message broker* AMQP. A estrutura apresentada neste capítulo foi utilizada como suporte para o desenvolvimento dos novos componentes da camada de implementação para transporte de mensagens entre atores remoto do projeto Akka (figura 3.3). Entretanto, antes de apresentar a estrutra que definimos, comentamos brevemente algumas características de entidades conectadas ponto-a-ponto via *sockets* que são impactadas ou deixam de fazer sentido com a nova abordagem.

5.1 Entidades conectadas ponto-a-ponto via *sockets*

Em uma transferência de mensagens entre duas entidades conectadas ponto-a-ponto por *sockets* TCP ou UDP, ambas as entidades devem conhecer detalhes sobre a conexão e sobre o protocolo das mensagens que são enviadas. O fato de não haver uma entidade intermediando a troca de mensagens, permite que uma das entidades identifique uma eventual desconexão da outra, ou ainda que uma entidade conheça o endereço do nó onde está a outra entidade. Nos casos onde as entidades estão em nós que não fazem parte de uma rede interna, a informação de localidade pode não indicar exatamente o nó corrente da entidade, mas o endereço de alguma porta de ligação (*gateway*).

Uma característica importante da transferência de mensagens entre entidades conectadas ponto-a-ponto via *sockets*, é que cada *socket* utiliza exclusivamente uma porta para aceitar as conexões remotas, criando uma relação de um para um entre portas e entidades.

Os protocolos TCP e UDP utilizam para a numeração de portas em seus cabeçalhos inteiros de 16 bits sem sinal, limitando o número de portas a 65536 (0 – 65535)[Ins81]. Ademais, algumas portas que são utilizadas para serviços comuns, como por exemplo servidores de correio eletrônico, possuem numeração fixa definida pela IANA (*Internet Assigned Numbers Authority*) e não podem ser abertas para uso geral.

Este tipo de abortagem não implica no uso de armazenamento intermediário das mensagens. Uma mensagem enviada de uma entidade para outra, até poderia ser armazenada pela entidade que recebe a mensagem, mas não pelo mecanismo de transporte. Fica a cargo da entidade que recebe a mensagem implementar o armazenamento temporário caso haja necessidade.

5.1.1 Atores remotos conectados ponto-a-ponto

Uma implementação de atores remotos que não imponha como limite à quantidade de atores remotos a serem criados em um nó o número de portas disponíveis no próprio nó deve, de alguma maneira permitir que conjuntos de atores sejam associados às portas TCP ou UDP.

Na implementação de atores remotos do Akka apresentada na seção 3.2, vimos que os atores são agrupados por endereço do nó hospedeiro (*host*) e porta e ficam armazenados no `RemoteServerModule` . Vimos também na seção 3.1 que atores são identificáveis, o que permite que eles sejam localizados no `RemoteServerModule` para que possam receber as mensagens.

Ainda que a implementação de atores remotos do Akka agrupe os atores, removendo a relação do limite do número de atores com o número de portas disponíveis, as entidades `RemoteServerModule` e `RemoteClientModule` ainda assim possuem o conhecimento do *host* e porta no qual se está conectado.

A implementação de atores remotos de Erlang não usa portas explicitamente. Cada máquina virtual Erlang possui um nome associado, e esse nome é utilizado junto com a informação do *host* durante a criação de atores remotos. O nome da máquina virtual é definido durante a inicialização da máquina virtual via parâmetro `-name`. Diversas máquinas virtuais podem estar em execução em um determinado *host* e são unicamente identificadas por `name@host`. Vale ressaltar que as máquinas virtuais Erlang que estão em uma mesma rede de computadores estão por padrão em *cluster*.

5.2 Entidades conectadas via *broker* AMQP

A substituição por um mecanismo baseado em troca de mensagens naturalmente introduz novas característas à comunicação. As entidades passam a não estarem mais conectadas diretamente, já que o *broker* passa a ser o componente central de conexão entre todas as entidades. Eventuais funcionalidades baseadas na comunicação ponto-a-ponto, como por exemplo uma das partes saber que a outra não está mais conectada, ainda que possíveis não são triviais. O *broker* passa a ser o responsável por abrir uma porta para que as demais entidades possam se conectar e é responsável por fazer o gerenciamento de múltiplas conexões na porta.

Tanto a rotulação das entidades por *host* e porta ou por nome e *host* deixam de fazer sentido pois são redundantes. As entidades passam a ser identificadas somente por seus nomes. Cada nova entidade que entrar no sistema é responsável por registrar no *broker* uma fila e associá-la a uma *exchange* para poder receber mensagens. Dependendo do papel que a entidade exerce, além de definir a fila e o *binding*, a entidade precisa definir antes uma *exchange*. Entidades que possuem o papel de servidora têm essa responsabilidade. Tarefas administrativas como a criação de *virtual hosts*, a criação de usuários e a definição das permissões dos usuários não são de responsabilidade das entidades, e devem ter sido executadas previamente.

O suporte ao desenvolvimento dos novos componentes da camada de implementação de transporte remoto do projeto Akka (figura 3.3) é dado com a ajuda de alguns componentes intermediários. Esses componentes intermediários formam uma ponte entre a nova camada de transporte remoto e a implementação do *broker* RabbitMQ [RMQ].

5.2.1 Pontes AMQP

O módulo `AMQPBridge` é responsável por fornecer classes que agem como pontes para o *message broker* utilizado neste trabalho. Juntas, suas classes definem uma interface com funcionalidades básicas para troca de mensagens como envios, recebimentos, tratamento de mensagens retornadas e suporte a múltiplas entidades cliente conectadas a uma entidade servidora. O módulo `AMQPBridge` contém as seguintes classes:

- `AMQPBridge`: É a classe abstrata que define o comportamento básico de uma entidade conectada a um *broker* AMQP. O construtor de `AMQPBridge` recebe como argumentos o nome que identifica a ponte e uma conexão para um *broker* AMQP. Essa classe define ainda o padrão dos nomes a serem utilizados na criação dos objetos no *virtual host* acessível pela conexão recebida no construtor, como mostrado na listagem 5.1. A classe possui também um objeto companheiro que define métodos para a criação de instâncias das sub-classes de `AMQPBridge`;
- `MessageHandler`: É a *trait* que define os métodos a serem invocados quando mensagens forem recebidas ou envios feitos pela entidade forem rejeitados pelo *broker*;
- `ServerAMQPBridge`: É uma das sub-classes de `AMQPBridge` e define o comportamento para entidades que atuam como servidoras de um serviço. Além dos métodos herdados, essa classe ainda define o método `setup`. Esse método recebe como argumento uma implementação de `MessageHandler` e os parâmetros com as configurações da fila e da *exchange* que serão criadas durante a execução do método, como mostrado na listagem 5.2;
- `ClientAMQPBridge`: É a outra sub-classe de `AMQPBridge` e define o comportamento para entidades que atuam como clientes de um serviço. Essa classe tem em seu construtor um parâmetro adicional para a identificação do cliente. A identificação é necessária para compôr a chave de roteamento que associa a fila que essa ponte irá criar no método `setup`, com a *exchange* correspondente. A definição dessa classe é mostrada na listagem 5.3.

```
1 abstract class AMQPBridge(val name: String,
2                 val connection: SupervisedConnectionWrapper) {
3   require(nodeName != null)
4   require(connection != null)
5
6   val id: String
7   lazy val inboundExchangeName = "actor.exchange.in.%s".format(name)
8   lazy val inboundQueueName = "actor.queue.in.%s".format(name)
9   lazy val outboundQueueName = "actor.queue.out.%s".format(name)
10  lazy val routingKeyToServer = "to.server.%s".format(name)
11
12  def sendMessageTo(message: Array[Byte], to: String): Unit
13  def shutdown: Unit = { ... }
14 }
```

Listagem 5.1: *Classe AMQPBridge do módulo AMQPBridge.scala.*

```
1 class ServerAMQPBridge(name: String,
2               connection: SupervisedConnectionWrapper)
3                   extends AMQPBridge(name, connection){
4   lazy val id = "server.%s".format(name)
5
6   def setup(handler: MessageHandler,
7         exchangeParams: ExchangeConfig.ExchangeParameters,
8           queueParams: QueueConfig.QueueParameters): ServerAMQPBridge = {
9     connection.serverSetup(
10      RemoteServerSetup(handler,
11        ServerSetupInfo(exchangeParams, queueParams,
12                  exchangeName = inboundExchangeName,
13                  queueName = inboundQueueName,
14                  routingKey = routingKeyToServer))
15    )
16    this
17  }
```

```
18  ...
19 }
```

Listagem 5.2: *Classe ServerAMQPBridge do módulo AMQPBridge.scala.*

```
1 class ClientAMQPBridge(name: String,
2               connection: SupervisedConnectionWrapper,
3                 idSuffix: String) extends AMQPBridge(name, connection) {
4  lazy val id = "client.%s.%s".format(name, idSuffix)
5
6  def setup(handler: MessageHandler, queueParams: QueueConfig.
       QueueParameters): ClientAMQPBridge = {
7    connection.clientSetup(
8      RemoteClientSetup(handler,
9        ClientSetupInfo(config = queueParams,
10              name = outboundQueueName + id,
11              exchangeToBind = inboundExchangeName,
12              routingKey = id))
13   )
14   this
15  }
16 ...
17 }
```

Listagem 5.3: *Classe ClientAMQPBridge do módulo AMQPBridge.scala.*

Definimos também um módulo auxiliar chamado `AMQP`. Esse módulo possui algumas classes com configurações pré-definidas para os objetos a serem criados no *broker*. A enumeração `StorageAndConsumptionPolicy` define um conjunto de valores que definem políticas de armazenamento e de consumo. A enumeração é composta pela combinação da enumeração `QueueConfig` e `ExchangeConfig`. Essas duas enumerações definem as configurações específicas e detalhadas para filas e *exchanges*, respectivamente. A combinação dos seus valores dão semântica às políticas de configuração.

Os três valores da enumeração `StorageAndConsumptionPolicy` são:

1. `DURABLE`: Define uma configuração onde os objetos criados são duráveis, ou seja, continuam a existir caso o *message broker* seja reiniciado;

2. `TRANSIENT`: Define uma configuração onde os objetos criados são transientes, ou seja, deixam de existir caso o *message broker* ser reiniciado;

3. `EXCLUSIVE_AUTODELETE`: Define uma configuração onde os objetos estão atrelados ao ciclo de vida da aplicação que os criou. Os objetos são removidos pelo *message broker* quando a conexão por onde os objetos foram criados é fechada. Ademais, as filas criadas por essa configuração possuem acesso exclusivo a um único consumidor. O consumidor deve estar atrelado ao canal por onde a criação da fila ocorreu.

5.2.2 Gerenciamento de conexões e canais

O módulo `ConnectionPoolSupervisor` é responsável por encapsular os detalhes da interação com o *broker*. Como mostrado na sub-seção 4.4.1, o envio de comandos e recebimentos de mensagens com o RabbitMQ acontece por meio de canais que são abertos em uma conexão. Esses canais não oferecem proteção para acesso concorrente, deixando a cargo da aplicação fazer a proteção.

A principal missão desse módulo é prover uma implementação segura para acesso concorrente aos canais e conexões abertos com o RabbitMQ. Além disso, o módulo ainda define uma interface simples e clara para as pontes AMQP, abstraindo toda a responsabilidade em relação à interação com o *broker*.

Para que os canais possam ser acessados concorrentemente de modo seguro, optamos por encapsulá-los em atores Akka. A idéia de encapsular o acesso aos canais em atores foi inspirada na implementação de um dos módulos adicionais do projeto Akka. O módulo AMQP do projeto Akka abstrai a criação de consumidores e produtores como atores, além de também abstrair a criação de objetos no *broker*.

A classe `SupervisedConnectionWrapper` define uma conexão supervisionada, expondo uma interface muito simples com apenas quatro métodos. Os métodos expõem funcionalidades básicas para envios de mensagens, fechamento da conexão e configuração, como mostrado na listagem 5.4. Nessa listagem podemos notar também que o construtor da classe recebe como argumento três atores, sendo o primeiro ator um ator referente a conexão, o segundo um ator referente ao canal utilizado para recebimento de mensagens (canal de leitura) e por último, o ator referente ao canal utilizado para o envio de mensagens (canal de escrita). Esses atores são os responsáveis por executar efetivamente as ações, já que a classe `SupervisedConnectionWrapper` delega aos atores todas as ações via envios síncronos e assíncronos de mensagens.

```
1 class SupervisedConnectionWrapper(connection: ActorRef,
2                     readChannel: ActorRef, writeChannel: ActorRef){
3  def close() { ... }
4
5  def clientSetup(setupInfo: RemoteClientSetup) { ... }
6
7  def serverSetup(setupInfo: RemoteServerSetup) { ... }
8
9  def publishTo(exchange: String, routingKey: String,
10           message: Array[Byte]) { ... }
11 }
```

Listagem 5.4: *Classe SupervisedConnectionWrapper.*

As principais classes do módulo `ConnectionPoolSupervisor` são:

- `BridgeConsumer`: É uma implementação de um consumidor padrão de mensagens. Essa implementação extende a classe `DefaultConsumer` do RabbitMQ, e utiliza a implementação do `MessageHandler` para a notificação de eventos como mensagens recebidas ou mensagens rejeitadas e retornadas;
- `ChannelActor`: É a *trait* que define o comportamento inicial de um ator responsável por um canal genérico;
- `ReadChannelActor`: É a classe que define o comportamento inicial de um ator responsável por um canal de leitura, utilizado para o recebimento de mensagens;
- `WriteChannelActor`: É a classe que define o comportamento inicial de um ator responsável por um canal de escrita, utilizado para o envio de mensagens;
- `ConnectionActor`: É a classe que define o comportamento inicial de um ator responsável pela conexão e abertura de canais com o *broker*. Uma outra responsabilidade desse ator é supervisionar os atores que lhe solicitaram a abertura de canais;
- `AMQPSupervisor`: É a *trait* que define o comportamento padrão a ser utilizado por um ator que será responsável por supervisionar os atores de conexão. Essa *trait* define ainda métodos que interagem com a fábrica de conexões do RabbitMQ para a criação de novas conexões;

- `AMQPConnectionFactory`: É a classe raíz da hierarquia de supervisão. Essa classe herda de `AMQPSupervisor` e define o despachador para todos os atores que estarão direta ou indiretamente sob supervisão. A classe possui ainda um objeto companheiro que define métodos para a criação dos atores de canais e conexões;

- `ConnectionSharePolicy`: É a enumeração que define a política de compartilhamento de uma conexão. A enumeração possui apenas dois valores que indicam se os canais de escrita e de leitura devem ser abertos na mesma conexão, ou se em conexões diferentes. O uso de uma conexão por canal permite que a alta atividade em um canal não impacte o outro.

O módulo possui ainda diversas classes menores (*case classes*) que são utilizadas para definir as mensagens que cada ator aceita em sua função `receive`, como por exemplo as mensagens `RemoteServerSetup` e `ServerSetupInfo` utilizadas na listagem 5.2. Um outro exemplo são as mensagens `RemoteClientSetup` e `ClientSetupInfo` utilizadas na listagem 5.3.

A figura 5.1 mostra o relacionamento do módulo que define as pontes AMQP, o módulo auxiliar para definição de configurações e o módulo para gerenciamento de conexões e canais descrito nesta sub-seção.

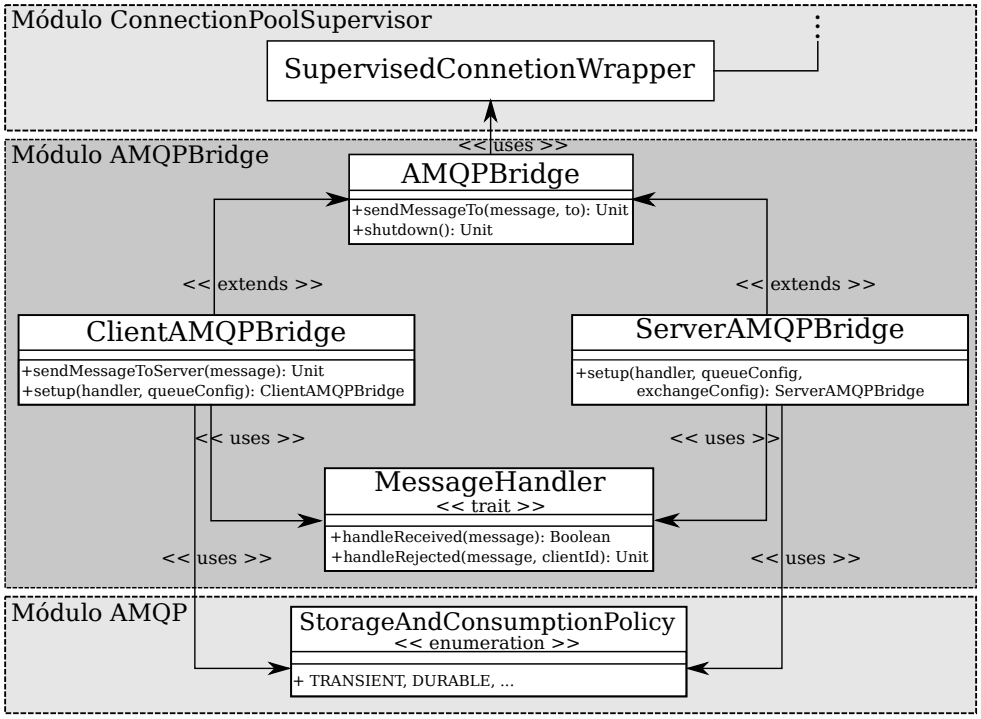

Figura 5.1: *Módulos de acesso ao broker AMQP.*

5.2.3 O processo de criação dos objetos no *message broker*

A criação de objetos como filas e *exchanges* no *broker* é iniciado nas pontes AMQP. O método `setup` das classes `ServerAMQPBridge` e `ClientAMQPBridge` serve como ponto de entrada para a criação e configuração dos objetos no *broker*. A criação de uma ponte AMQP é feita via métodos definidos no objeto `AMQPBridge` e depende da criação de uma conexão supervisionada.

A conexão supervisionada é criada pelo objeto `AMQPConnectionFactory`. Para que essa conexão seja criada, é necessário que os atores que encapsulam a conexão e os canais também sejam criados. A primeira instância a ser criada é a do ator responsável pela conexão. A criação do ator de conexão é ilustrada na figura 5.2 e acontece em quatro passos:

1. Uma instância de `ConnectionActor` é criada (`Actor.actorOf`) e iniciada (`start`);
2. O ator é ligado (`link`) ao ator supervisor[1];
3. A mensagem `Connect` é enviada sincronamente para o ator (`!!`);
4. A mensagem `Connect` é processada pelo ator. Seu processamento leva a solicitação de uma nova conexão para a fábrica de conexões do RabbitMQ.

Dependendo do tipo de política definida para o compartilhamento da conexão entre os canais de leitura e escrita, o passo 4 é executado duas vezes.

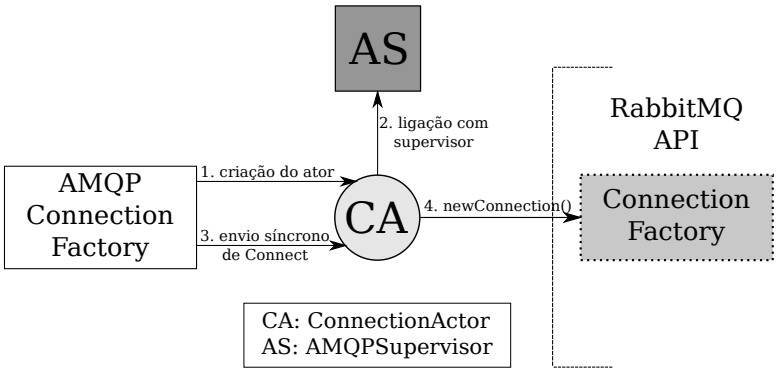

Figura 5.2: *Passos para a criação do ator de conexão.*

Uma vez que o ator de conexão foi criado, são criados os atores responsáveis pelo canal de leitura e pelo canal de escrita. A criação do ator responsável pelo canal de leitura acontece é ilustrada na figura 5.3 e acontece em cinco passos:

1. Uma instância de `ReadChannelActor` é criada (`Actor.actorOf`) e iniciada (`start`);
2. O ator é ligado (`link`) ao ator supervisor, que neste caso é o ator de conexão recém criado;
3. A mensagem `StartReadChannel` é enviada sincronamente para o ator (`!!`);
4. A mensagem `StartReadChannel` é processada pelo ator. Seu processamento leva ao envio síncrono da mensagem `ReadChannelRequest` para o ator de conexão que foi definido como supervisor;
5. O ator de conexão solicita um novo canal à sua conexão de leitura. O novo canal é enviado como resposta para `ReadChannelRequest`;

O processo para criação do ator responsável pelo canal de escrita é bem semelhante, com diferenças nas mensagens enviadas nos passos 3 (`StartWriteChannel`) e 4 (`WriteChannelRequest`). No passo 5, o novo canal é solicitado para a conexão de escrita. As conexões de escrita e leitura se referem à mesma instância no caso de uma política de compartilhamento de conexões.

É importante destacar que todos os parâmetros de configurações utilizados para a criação de uma conexão, como por exemplo a política de compartilhamento de conexões e o endereço do *broker*, são lidos de um arquivo de propriedades. Os detalhes sobre esse arquivo são apresentados no capítulo 6.

[1] A classe `AMQPConnectionFactory` é o ator supervisor de todos os atores de conexão.

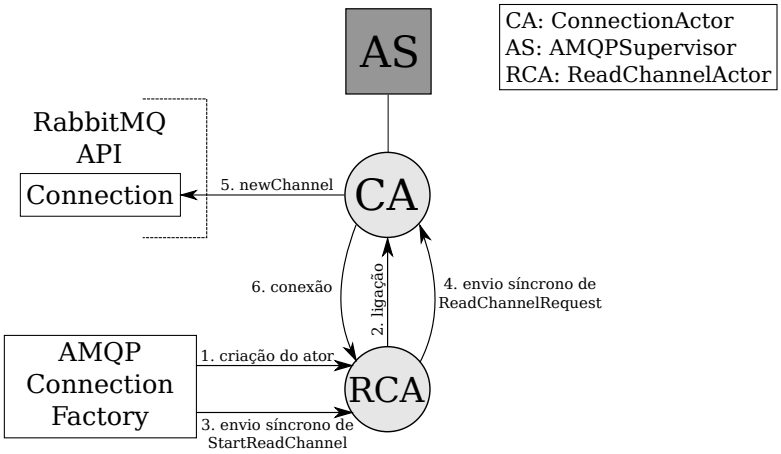

Figura 5.3: *Passos para a criação do ator do canal de leitura.*

Uma vez que os atores de conexão e de canais foram criados, a ponte que está sendo criada tem seu método `setup` executado. A execução do método `setup` tem diferentes passos para cada tipo de ponte.

A execução do método `setup` da classe `ServerAMQPBridge` é ilustrada na figura 5.4 e acontece em oito passos:

1. O método `setup` da classe `SupervisedConnection` é invocado;
2. A mensagem `RemoteServerSetup` é enviada para o canal de escrita com as configurações dos objetos a serem criados;
3. O processamento da mensagem `RemoteServerSetup` leva a criação de uma *exchange* direta com a configuração de durabilidade solicitada;
4. A implementação de `MessageHandler` informada é utilizada na associação do tratador de mensagens retornadas (*return listener*);
5. A mensagem `RemoteServerSetup` é enviada para o canal de leitura com as configurações dos objetos a serem criados;
6. O processamento da mensagem `RemoteServerSetup` leva a criação de uma fila com a configuração de durabilidade solicitada;
7. O *binding* entre a fila recém criada e a *exchange* criada no passo 3 é criado;
8. A implementação de `MessageHandler` informada é utilizada na associação do consumidor na fila criada no passo 6.

Vale lembrar que a criação de *exchanges* e filas pela biblioteca Java do RabbitMQ verifica a existência do objeto antes de sua criação. Caso já exista um objeto de mesmo nome e com as mesmas características, o objeto não é recriado e a execução prossegue normalmente.

Podemos notar que os passos são divididos entre os canais. O canal de leitura fica responsável por definir a fila e associar o consumidor padrão, algo que se torna mandatório por conta de podermos utilizar o consumidor como único e exclusivo da fila.

A execução do método `setup` da classe `ClientAMQPBridge` é ilustrada na figura 5.5 e acontece em sete passos:

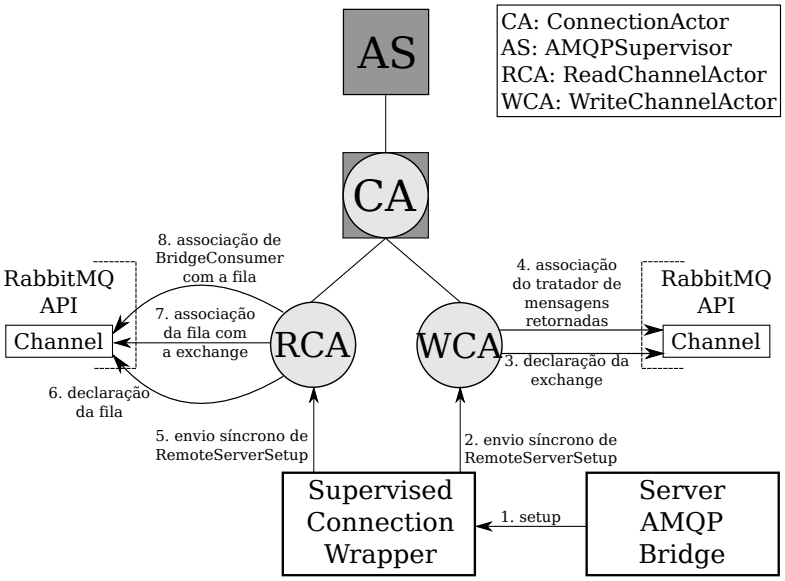

Figura 5.4: *Passos de configuração da classe ServerAMQPBridge.*

1. O método `setup` da classe `SupervisedConnection` é invocado;
2. A mensagem `RemoteClientSetup` é enviada para o canal de escrita com as configurações dos objetos a serem criados;
3. A implementação de `MessageHandler` informada é utilizada na associação do tratador de mensagens retornadas;
4. A mensagem `RemoteClientSetup` é enviada para o canal de leitura com as configurações dos objetos a serem criados;
5. O processamento da mensagem `RemoteClientSetup` leva a criação de uma fila que rotulamos como "fila de saída" que possui a configuração de durabilidade solicitada;
6. O *binding* entre a fila recém criada e a *exchage* definida pela ponte servidora correspondente é criado;
7. A implementação de `MessageHandler` informada é utilizada na associação do consumidor na fila criada no passo 5.

O Akka mantém um registro com todos os atores em execução. Nesse registro é possível localizar um ator por seu identificador e até mesmo interromper a execução de todos os atores do registro. Para evitar efeitos colaterais entre as interações do registro de atores do Akka com a nossa implementação, optamos por remover a hierarquia que definimos do registro do Akka. Isso acontece após a incialização de cada ator via método `Actor.registry.unregister(actor)`. Nossa hierarquia de atores, por sua vez, pode ter sua execução interrompida via método `AMQPConnectionFactory .shutdownAll`.

Segundo a específicação do padrão AMQP [Gro], as implementações de *brokers* devem suportar ao menos 256 filas por *virtual host*. Contudo, a recomendação é que não haja imposição de limites que não pela disponibilidade de recursos. A especificação também menciona que o número mínimo de *exchanges* por *virtual host* é de 16, e faz a mesma observação quanto a não imposição de limite que não seja pela disponiblidade de recursos. A implementação RabbitMQ utilizada neste trabalho deixa as limitações a cargo da disponibilidade de recursos do sistema hospedeiro.

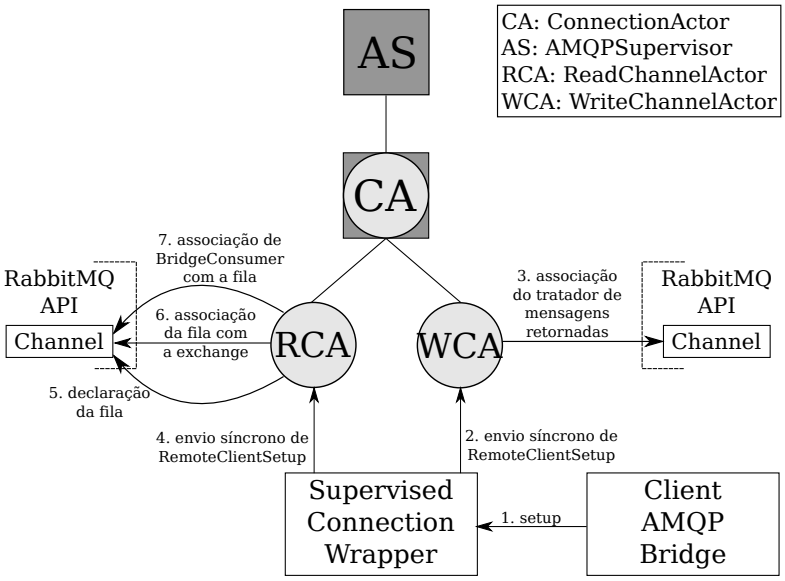

Figura 5.5: *Passos de configuração da classe ClientAMQPBridge.*

Mostramos na figura 5.6 como ficaria a criação de uma ponte servidora de nome `node1` com duas pontes clientes associadas a ela. A *exchange* e a fila que foram criadas pela ponte servidora estão marcadas em preto. Pela figura 5.6 podemos notar como a *exchange* definida pela ponte servidora tem o papel de ponto central para envio de mensagens para todas as filas criadas para o nó `node1`. Podemos notar também como os padrões definidos para os nomes na classe `AMQPBrigde` (listagem 5.1) são utilizados na criação dos objetos no *broker*. A ponte servidora define a fila de entrada de mensagens. As pontes cliente definem as filas de saída de mensagens. As mensagens que serão eventualmente roteadas para as filas de saída serão mensagens de resposta vindas da ponte servidora. Devemos observar que a ponte servidora precisa do identificador de uma ponte cliente para poder lhe enviar uma mensagem.

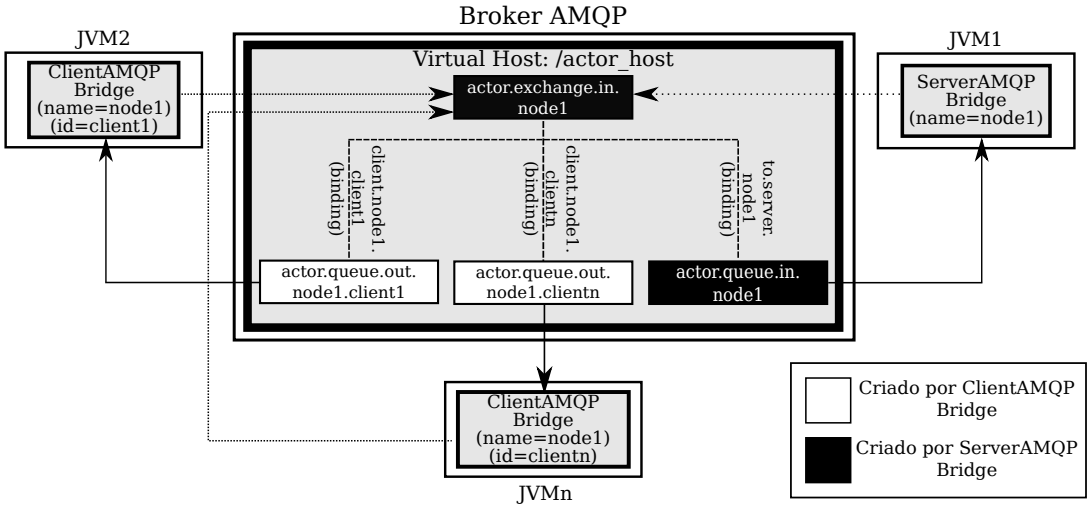

Figura 5.6: *Estrutura para troca de mensagens entre entidades remotas via broker AMQP.*

5.2.4 Envio e recebimento de mensagens via pontes AMQP

Iremos analisar agora os passos para o envio de mensagens de uma ponte cliente para a sua ponte servidora.

O envio de mensagens de uma ponte cliente para uma ponte servidora acontece na invocação do método `sendMessageToServer` da classe `ClientAMQPBridge`. A execução do método é ilustrada na figura 5.7 e acontece em quatro passos:

1. O método `sendMessageToServer` foi invocado por alguma classe que deseja enviar uma mensagem;
2. O método `publishTo` da classe `SupervisedConnectionWrapper` é invocado. A invocação do método utiliza como argumentos a mensagem a ser enviada, o nome da *exchange* que foi criada com base no nome da ponte e o nome do *binding* entre a *exchange* e a fila de entrada da ponte servidora;
3. O método `publishTo` faz um envio assíncrono da mensagem `BasicPublish` ao ator responsável pelo canal de escrita. A mensagem `BasicPublish` possui informações sobre detalhes do envio a ser executado como por exemplo, se o envio deve ser mandatório e o consumo imediato;
4. O método `basicPublish` é invocado na classe `Channel` do RabbitMQ como resultado do processamento da mensagem `BasicPublish`. Os argumentos utilizados para execução do método são os que foram passados junto à mensagem `BasicPublish`.

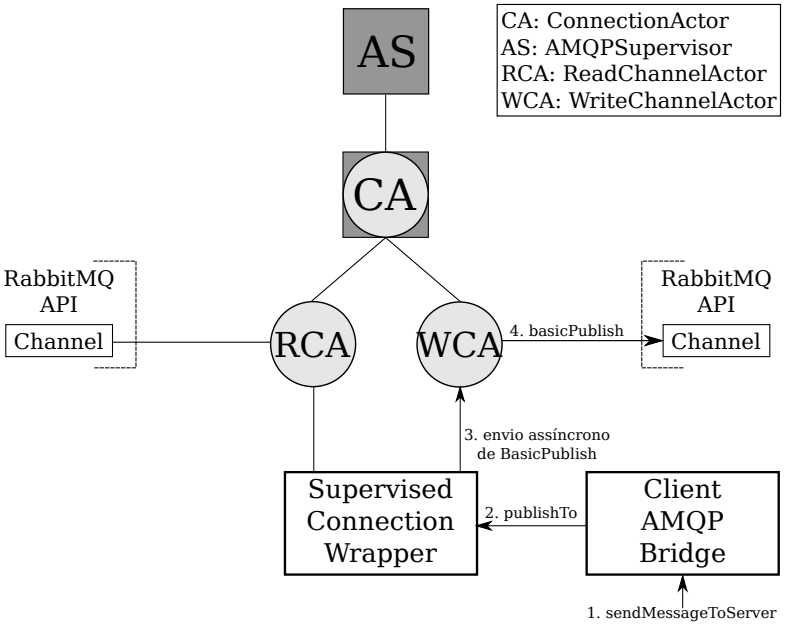

Figura 5.7: *Passos do envio de mensagens de um cliente via pontes AMQP.*

Optamos por fazer envios mandatórios porém com consumo não imediato. A fila de destino deve obrigatoriamente existir no momento do envio, mas a mensagem não precisa ser consumida no momento do depósito na fila. O trecho de código que executa o passo 4 é mostrado na listagem 5.5. Os valores para as variáveis `mandatory` e `immediate` são respectivamente `true` e `false`. Caso ocorra um envio massivo de mensagens, o consumo imediato das mensagens pode não ser possível, acarretando no retorno de mensagens para o remetente. Decidimos por não forçar um consumo imediato para que não sobrecarregar as implementações dos consumidores, minimizando a possibilidade de retornos de mensagens.

Ainda assim, caso uma mensagem seja retornada, o método `handleRejected` do `MessageHandler` é invocado para que alguma ação seja tomada com a mensagem rejeitada. Mensagens podem ser retornadas por diversos motivos: uma mensagem não pode ser roteada para uma fila pois a chave

de roteamento utilizada no envio não está associada à *exchange*; uma mensagem foi depositada em alguma fila, porém a fila foi removida. Nesse caso, todas as mensagens são retornadas aos seus respectivos remetentes.

O envio de mensagens de uma ponte servidora para uma ponte cliente é similar ao mostrado na figura 5.7. A diferença está no nome do *binding* que é utilizado. O nome do *binding* é definido com base no identificador do cliente.

```
...
case BasicPublish(exchange, routingKey, mandatory, immediate, message) =>
{
  channel.foreach {
    ch => ch.basicPublish(exchange, routingKey, mandatory, immediate,
        null, message)
  }
}
...
```

Listagem 5.5: *Envio de mensagem mandatória e não imediata.*

O processo de roteamento da mensagem acontece como explicado na sessão 4.1.1. A mensagem é enviada para a *exchange* relacionada a `ClientAMQPBridge`. O identificador do remetente (atributo `id` da listagem 5.3) deve ser enviado como parte da mensagem e esse envio é de responsabilidade do criador da mensagem. Essa identificação será utilizada como argumento do método `sendMessageTo` no caso de haver uma mensagem de resposta.

O recebimento de uma mensagem por uma ponte servidora é ilustrado na figura 5.8 e acontece em três passos:

1. O *broker* repassa a mensagem ao canal de leitura;
2. O método `handleDelivery` da classe `BridgeConsumer` é invocado;
3. O método `handleReceived` da implementação de `MessageHandler` (definida durante a configuração da ponte servidora) é executado pelo `BridgeConsumer` para que a mensagem possa ser processada.

Devemos recordar que a execução do método `handleReceived` deve retornar um valor boleano. Esse valor indica se o recebimento deve ser confirmado para então a mensagem ser removida da fila pelo *broker*. Os passos utilizados para o recebimento de uma mensagem por uma ponte cliente são idênticos aos ilustrados na figura 5.8.

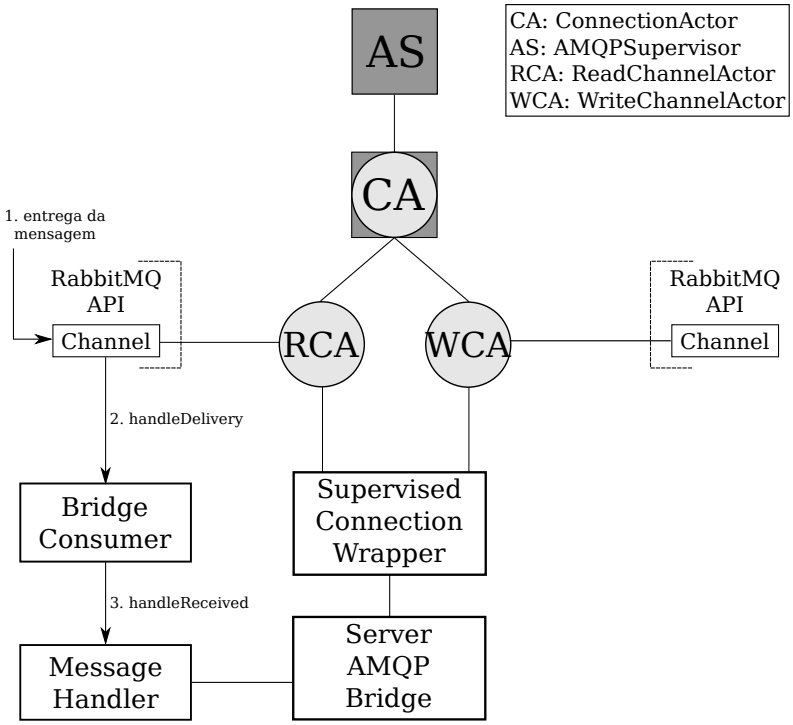

Figura 5.8: *Passor para recebimento de mensagens via pontes AMQP.*

Capítulo 6

Atores Remotos com o Padrão AMQP

Apresentamos neste capítulo nossa implementação de atores remotos que utilizam o padrão AMQP como meio de transporte para troca de mensagens. Nossa implementação utiliza a implementação de atores remotos do projeto Akka apresentada no capítulo 3. Na seção 6.1 apresentamos os novos componentes que criamos dentro da estrutura do Akka e na seção 6.2 como fizemos a integração dos novos componentes com a implementação existente.

6.1 Novos componentes

A implementação de atores remotos do projeto Akka define uma camada intermediária que desacopla a definição dos componentes para transporte das mensagens a atores remotos, como apresentado na seção 3.2.

Para viabilizar o envio de mensagens remotas via *broker* AMQP, tivemos que criar novos componentes para a camada de implementação com base nas classes e *traits* que definem a interface remota (figura 3.3). Utilizamos a estrutura apresentada no capítulo 5 como suporte para a implementação dos novos componentes do Akka. Os novos componentes são:

- `AMQPRemoteServer`: É uma classe utilizada no lado do servidor onde estão os atores remotamente acessíveis. Essa classe implementa os métodos abstrados de um `MessageHandler` e é responsável pelo recebimento, desseriação e encaminhamento das mensagens para os atores que estão em seu registro. A classe também é responsável por enviar eventuais mensagens de resposta para os remetentes, seja como consequência de envios feitos via `!!` e `!!!`, ou ainda mensagens para atores supervisores. Tanto no recebimento como no envio de mensagens a classe utiliza um módulo separado para seriação e desseriação de mensagens. Existe um relacionamento bidirecional um para um entre um `AMQPRemoteServer` e uma ponte servidora;

- `AMQPRemoteServerModule`: É uma *trait* que extende a *trait* `RemoteServerModule` e define um nó onde atores são registrados para ficarem acessíveis remotamente. Essa implementação mantém o registro desses atores e possui uma relação bidirecional um para um com um `AMQPRemoteServer`. As mensagens recebidas pelo `AMQPRemoteServer` são encaminhadas para os atores residentes nesse módulo;

- `AMQPRemoteClient`: É uma classe utilizada no lado do cliente onde ficam os *proxies* dos atores remotos. A classe mantém o registro de atores que são supervisores de atores remotos, além de também manter um registro para resultados futuros. Tal como a classe `AMQPRemoteServer`, essa classe também implementa os métodos abstratos de um `MessageHandler` e é também é responsável pelo recebimento, desseriação e encaminhamento das mensagens para os atores ou resultados futuros que estão em seu registro. O principal papel da classe é fazer o envio

das mensagens para o *broker* AMQP. Existe uma relação bidirecional um para um entre um `AMQPRemoteClient` e uma ponte cliente. A classe ainda é responsável por manter um identificador que deve ser único entre todos os clientes do mesmo servidor remoto. Esse identificador é utilizado na criação da instância da ponte cliente;

- `AMQPRemoteClientModule`: É uma *trait* que extende a *trait* `RemoteClientModule`. Essa é uma *trait* que tem um papel complementar a `AMQPRemoteServerModule` e seu papel principal é definir uma interface para envios de mensagens a atores remotos via `AMQPRemoteClient`. Ademais, a *trait* mantém ainda um registro de `AMQPRemoteClient`, que são utilizados para enviar as mensagens para seus respectivos nós remotos, já que uma aplicação cliente pode interagir com diversos atores em diferentes nós;
- `AMQPRemoteSupport`: É a implementação que tornamos acessível via `Actor.remote`. Essa classe é responsável por concentrar as responsabilidades definidas nos módulos acima listados para prover o suporte remoto via *broker* AMQP.

O relacionamento entre os novos componentes com as pontes AMQP e com as classes do Akka são mostrados na figura 6.1.

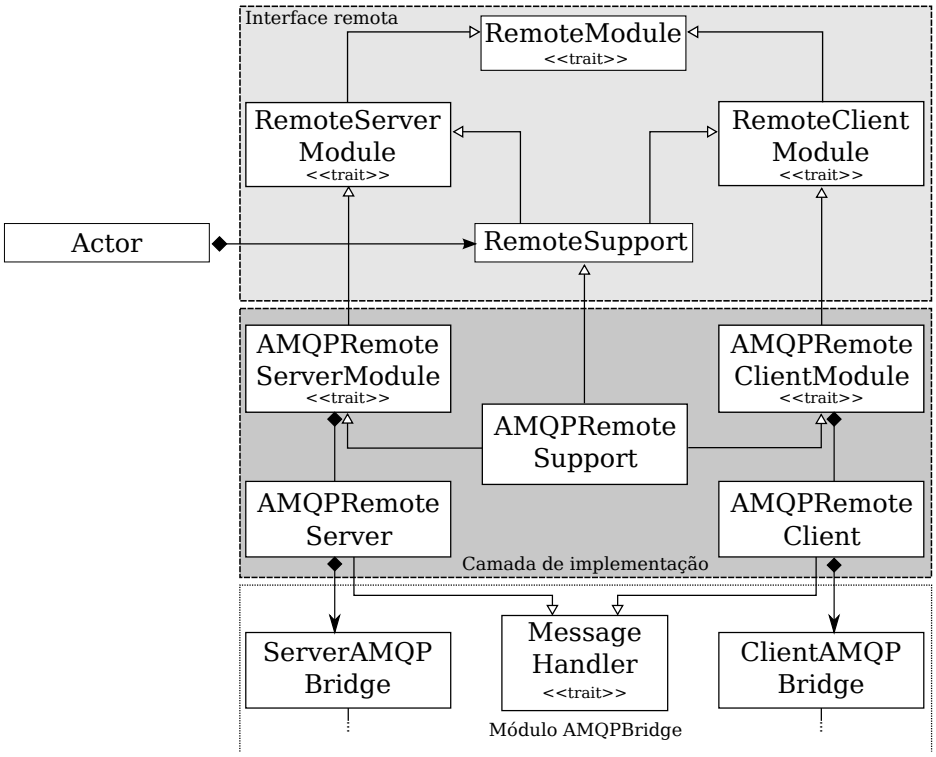

Figura 6.1: *Relacionamento entre os componentes para atores remotos com AMQP.*

A definição das *traits* para suporte a atores remotos do Akka define assinaturas de métodos baseadas em *host* e porta. Para não quebrar a compatibilidade em nossas implementações e nem alterar a interface dos métodos, optamos por criar internamente o nome do nó com base nos valores informados para *host* e porta. O padrão utilizado é `host@porta`.

Como mencionado seção 3.1, `ActorRefs` guardam o endereço do local onde o ator foi criado. Pelo fato de não haver associação entre o endereço definido e uma *socket*, e de os valores para *host* e porta serem recebidos no construtor da classe, optamos por manter a compatibilidade e não mudar nem o construtor e nem o atributo. Em nossa implementação, o valor do atributo `homeAddress` não é utilizado.

Pelo fato do projeto Akka possuir um desenvolvimento ativo, acreditamos que existe uma grande possibilidade de haver uma mudança na interface dos componentes do módulo de atores remotos,

de modo que haja um desacomplamento entre sua interface e as implementações da camada de transporte.

6.2 Integração com o Akka

Uma vez definidos os componentes dentro da estrutura do Akka, é necessário que a biblioteca do Akka faça uso deles, de modo que a instância referenciada por `Actor.remote` seja uma instância de `AMQPRemoteSupport`.

6.2.1 Alterações no arquivo akka.conf

Além de alterar o valor para a camada de suporte para atores remotos no arquivo `akka.conf` , optamos por adicionar novas propriedades para a configuração do suporte que implementamos, como os dados para conexão com o *broker*, a política de armazenamento e de compartilhamento de conexões entre canais. Uma outra propriedade importante que adicionamos foi a definição do identificador a ser utilizado na criação de diferentes `AMQPRemoteClients` (e consequentemente de `ClientAMQPBridges`). O novo valor para a camada de suporte para atores remotos e as novas propriedades são listadas a seguir:

```
remote {
   ...
   layer = "akka.remote.amqp.AMQPRemoteSupport"
   amqp {
     policy {
       storage {
         mode = "EXCLUSIVE_PERSISTENT"
         client_id {
           suffix = "MYPLACE"
         }
       }
       connection {
            server = "ONE_CONN_PER_NODE"
            client = "ONE_CONN_PER_NODE"
       }
     }
     broker {
       host = "192.168.0.121"
       virtualhost = "/actor_host"
       username = "actor_admin"
       password = "actor_admin"
     }
   }
   ...
}
```

Restrições

Pelo fato arquivo `akka.conf` ser único por JVM, a maneira pelo qual é feita a configuração das propriedades remotas impõe algumas restrições. Com o Akka em execução em uma JVM, podemos criar diversos servidores remotos, cada um com um nome de nó diferente. Também podemos criar clientes remotos, um para cada servidor remoto. Poderíamos ainda ter em uma JVM uma combinação dos dois cenários, com servidores remotos e clientes para servidores remotos.

Todos os clientes remotos que forem criados na JVM, independente do nome do servidor remoto utilizam o mesmo identificador de cliente. Contudo, essa restrição não impedimende a criação dos objetos no *broker*. Para exemplificar, podemos analizar o caso onde dois clientes remotos são criados em uma JVM para os servidores remotos cujos nomes são `node1` e `node2`. Os objetos criados pelos primeiro cliente remoto terão o sufixo `client.node1.<id>` e `client.node2.<id>`. Um outro detalhe importante é que, pelo fato de classe `AMQPRemoteClientModule` manter um registro dos clientes remotos, não há a possibilidade de na mesma JVM ser criado mais de uma instância de `AMQPRemoteClient` para um mesmo servidor remoto.

As propriedades para configuração da política de armazenamento são utilizadas para ambos os servidores e clientes remotos. Configurações divergentes entre JVMs podem levar aos seguintes cenários:

1. A JVM onde foi iniciado um servidor remoto utiliza uma configuração persistente e alguma das JVMs com um cliente remoto define uma configuração transiente: Esse não é um caso muito problemático, já que os objetos criados pelo servidor não dependem dos objetos criados pelos clientes. Como os objetos definidos como transientes só são removidos quando o *message broker* é desligado, o problema se limita a perda de eventuais mensagens que ainda não foram consumidas e estavam na fila;

2. Um cenário oposto ao anterior, onde o servidor remoto possui uma configuração transiente e algum de seus clientes uma configuração persistente: Esse é um cenário mais problemático, já que quando a *exchange* do servidor é removida todas as suas associações também são removidas, porém as filas persistentes não. Esse cenário deve ser analizado de duas ópticas diferentes: *(i)* Óptica da criação dos objetos: como a biblioteca Java do RabbitMQ faz uma verificação da existência de um objeto antes de tentar fazer a sua criação (5.2.2), logo não há problemas. A associação da fila persistente com a *exchange* recém criada acontecerá durante a execução da inicialização do módulo do cliente remoto; *(ii)* Óptica é a da aplicação: Eventuais mensagens não consumidas pelo servidor remoto são perdidas.

As configurações relacionadas ao compartilhamento da conexão entre os canais de leitura e escrita são definidas separadamente para os clientes e servidores remotos. A divergência nas configurações entre um cliente remoto em um nó e um servidor remoto em outro nó pode acontecer sem problemas, já que o impacto será no volume de dados trafegado na conexão.

6.2.2 Segurança

Com o intuito de previnir conexões em servidores remotos de clientes não autorizados, o Akka deixa como opção de configuração exigir que somente clientes autenticados possam interagir com os atores. A autenticação é feita via *secure cookie*. O *secure cookie* é definido no arquivo `akka.conf` na seção `remote`. O uso de autenticação é opcional e deve ser exigido, quando necessário, pelo servidor remoto. A propriedade utilizada para forçar o uso de *cookie* na comunicação é definida dentro de `remote { server { ... } }`.

O valor do *cookie* é parte do protocolo remoto definido no Akka, mostrado na listagem 3.15 no capítulo 3. Como o uso do *cookie* é definido no protocolo remoto, mantemos o suporte na nossa implementação. Mensagens oriundas de clientes que não se autenticarem nos servidores remotos, não são processadas e o cliente remoto remetente recebe uma mensagem de resposta com indicando a necessidade a autenticação.

Com o uso das filas definidas no *broker* como repositório intermediário das mensagens (salvo no caso de filas com acesso exclusivo), qualquer entidade que possua os dados necessários para acessar o *virtual host* onde estão criadas as filas pode registrar outros consumidores. Uma fila com mais de um consumidor pode ser interessante quando se deseja fazer um monitoramento do volume

ou até do conteúdo das mensagens. No caso de haver mais de um consumidor em uma fila, uma cópia da mensagem é entregue a cada consumidor. Não é possível, portanto, um consumidor mal intencionado "roubar" ou mesmo alterar a integridade das mensagens.

Com o uso de um *broker* AMQP, a segurança depende muito de quem possui as informações para acesso ao *virtual host* onde estão criadas as filas e *exchanges*.

6.2.3 Alterações no protocolo

Para que o servidor remoto pudesse identificar qual o remetente de uma mensagem recebida, foi necessária uma pequena alteração no protocolo remoto. A implementação de transporte feita com o JBoss Netty consegue identificar na *socket* qual o endereço do a entidade remota que está se conectada. O protocolo utilizado para definir uma mensagem remota passa a incluir um parâmetro não mandatório para que o identificador do cliente possa ser identificado. O protocolo com a alteração é mostrado na listagem 6.1. Essa alteração implicou em alterações no objeto `RemoteActorSerialization`. Esse objeto é o responsável por fabricar uma instância com as informações a serem seriadas. O método `createRemoteMessageProtocolBuilder` foi sobrecarregado e passou a receber o identificador do cliente como parâmetro.

A alteração apresentada na listagem 6.1 não seria necessária caso o protocolo definisse um identificador para o nó remoto responsável pelo envio da mensagem.

```
1 message RemoteMessageProtocol {
2   required UuidProtocol uuid = 1;
3   required ActorInfoProtocol actorInfo = 2;
4   required bool oneWay = 3;
5   optional MessageProtocol message = 4;
6   optional ExceptionProtocol exception = 5;
7   optional UuidProtocol supervisorUuid = 6;
8   optional RemoteActorRefProtocol sender = 7;
9   repeated MetadataEntryProtocol metadata = 8;
10  optional string cookie = 9;
11  optional string remoteClientId = 10; // identificador do cliente
12 }
```

Listagem 6.1: *Protocolo para mensagens remotas com identificador do cliente.*

Devemos nos lembrar que o protocolo que descreve as referências de atores remotos, mostrado na listagem 3.14, define um campo para armazenar o endereço de onde o ator remoto foi criado. Esse campo é composto pelo endereço do hospedeiro e porta do nó. Pelo fato de não haver uma associação entre o endereço do ator remoto e uma *socket* e do valor não ser utilizado no envio de mensagens e nem no envio de respostas, optamos por não fazer alterações na sua definição.

6.3 Fluxo de envio das mensagens

Parte do fluxo de um envio de uma mensagem a um ator remoto com a nova camada de transporte continua acontecendo em sete passos, como descrito na sub-seção 3.2.1. O processo de envio começa com o *proxy* local embrulhando a mensagem e adicionando a ela informações de cabeçalho necessárias para o envio e posterior processamento.

Antes da mensagem ser repassada para o seriador, o `AMQPRemoteSupport` adiciona o identificador do cliente na mensagem (campo na linha 11 da listagem 6.1).

O seriador continua sendo o responsável por converter a informação recebida em um vetor de *bytes* para que o transporte possa ocorrer. Uma vez que a informação esteja no formato a

ser transportado, o *proxy* usa uma implementação de `RemoteSupport` (que na figura 6.2 é uma instância de `AMQPRemoteSupport`) para enviar a mensagem ao `RemoteSupport` que está no lado do servidor.

Os passos 3 (transporte da mensagem do `ClientBootstrap` para o `ServerBootstrap`) e 4 (recebimento da mensagem pelo `ServerBootstrap` e repasse para o *handler*) acontecem de modo diferente em relação a implementação com JBoss Netty.

Pelo fato de utilizarmos as pontes AMQP definidas no capítulo 5, o passo 3 se resume em enviar uma mensagem à *exchange* associada a ponte utilizando o a chave de roteamento da ponte servidora. O *broker* roteia a mensagem para a fila criada pela ponte servidora. Apesar de o *broker* tentar entregar a mensagem para o consumidor da fila o quanto antes, a mensagem pode ficar armazenada caso o consumidor não esteja pronto para fazer o recebimento da mensagem. O passo 4 se resume no consumo da mensagem enviada e no seu repasse para a implementação de `MessageHandler` utilizada na ponte servidora.

Vale lembrar que após o recebimento de um mensagem remota, ocorre um envio exatamente igual a um envio local. O envio local de uma mensagem leva tempo $O(1)$, que essa é a ordem do tempo gasto para colocar uma mensagem na fila de um ator.

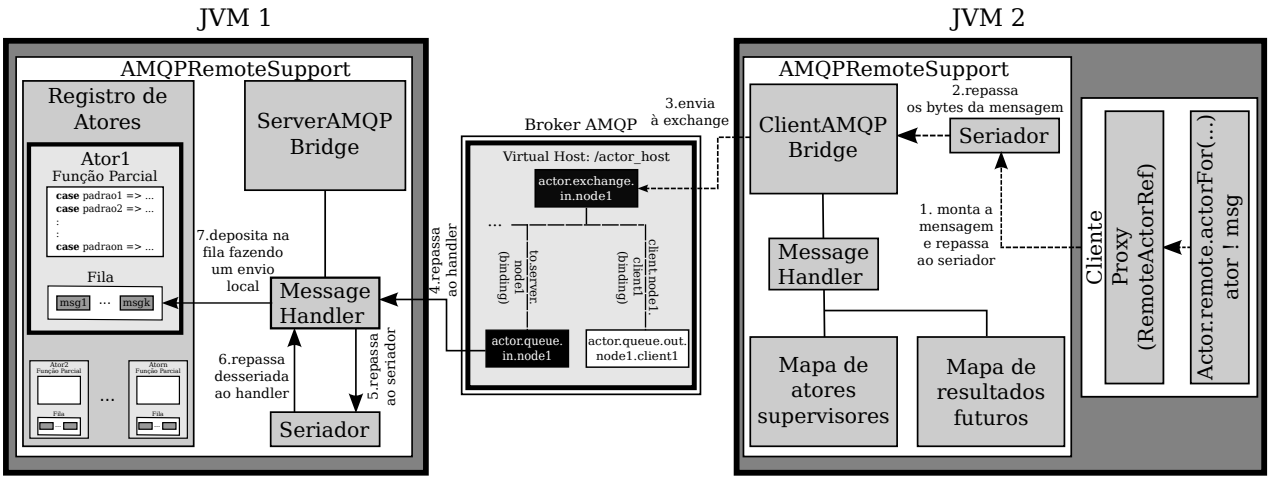

Figura 6.2: *Fluxo de um envio assíncrono de mensagem entre atores com AMQP.*

Assim como na implementação com JBoss Netty, os envios feitos via métodos `!!` e `!!!` possuem um passo a mais do que os mostrados na figura 6.2. Entre os passos 2 e 3, uma cópia da referência para instância de resultado futuro que é retornada é colocada no mapa de resultados futuros.

As mensagem de resposta para envios feitos via `!!` e `!!!`, ou para notificar atores supervisores sobre atores supervisionados fazer o caminho inverso. Essas mensagens são enviadas para a mesma *exchange* que a ponte cliente fez o envio, porém utilizando como chave de roteamento o valor que utiliza o identificador do cliente. Assim o *broker* fará o roteamento da mensagem para a fila do cliente.

Devemos lembrar que, caso a mensagem que foi enviada pelo servidor seja o resultado de um envio com `!!` ou `!!!`, o conteúdo da mensagem (seja um resultado de sucesso ou uma exceção) é utilizado para completar a instância cujo a referência havia sido colocada no mapa de resultados futuros.

Apêndice A

Exemplo de uma aplicação produtora e consumidora com RabbitMQ

Apresentamos neste apêndice um exemplo de uma aplicação produtora e de uma aplicação consumidora escrito em Scala com a biblioteca para clientes Java do RabbitMQ.

A listagem A.1 mostra a configuração da fábrica de conexões e alguns valores, como nome dos objetos e suas configurações, que são compartilhados entre as implementações de produtor e consumidor de mensagens.

```
1 trait CommonAMQP {
2
3   val EXCHANGE_NAME = "sample.exchange"
4   val QUEUE_NAME = "sample.queue"
5   val BINDING_KEY = "key.to.sample.queue"
6   val EXCHANGE_TYPE = "direct"
7   val AUTO_ACK  = true
8   val NOT_EXCLUSIVE = false
9   val NOT_DURABLE = false
10  val NOT_AUTODELETE = false
11  val QUEUE_ARGS = null
12  val BASIC_PROPS = null
13
14  private lazy val factory = {
15    val _factory = new ConnectionFactory()
16    _factory.setHost("localhost")
17    _factory.setPort(5672)
18    _factory.setUsername("anUser")
19    _factory.setPassword("t0psecr3t")
20    _factory.setVirtualHost("/amqp-sample")
21    _factory
22  }
23
24  def connect: Connection = {
25    factory.newConnection
26  }
27 }
```

Listagem A.1: *Trait CommonAMQP*

Para o nosso exemplo optamos por criar a classe `SampleProducer` para encapsular a criação dos objetos no *broker* e o envio das mensagens. A classe é apresentada na listagem A.2. O método `startProducer` se conecta ao servidor AMQP na linha 6, e abre um canal na linha 7 para interagir

com a camada de sessão. Nas linhas seguintes, o método declara uma *exhange*[1] direta e transiente, uma fila pública, transiente e temporária e faz o *binding* de ambos. O método `publish` por sua vez, verifica logo na sua primeira linha se o estado da instância é válido, e em seguida itera na sequência de mensagens fazendo o envio. Para esse exemplo em particular, omitimos as propriedades adicionais (`BASIC_PROPS`) que podem ser utilizadas no envio. Exemplos dessas propriedades são definir a prioridade da mensagem, especificar um remetente diferente para resposta (*replyTo*) e definir uma data para expiração da mensagem.

```
1 class SampleProducer extends CommonAMQP {
2   private var connection: Connection = _
3   private var channel: Channel = _
4
5   def startProducer: SampleProducer = {
6     connection = this.connect
7     channel = connection.createChannel
8     channel.exchangeDeclare(EXCHANGE_NAME, EXCHANGE_TYPE)
9     channel.queueDeclare(QUEUE_NAME, NOT_DURABLE, NOT_EXCLUSIVE,
          NOT_AUTODELETE, QUEUE_ARGS)
10    channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, BINDING_KEY)
11    this
12  }
13
14  def publish(messages: Seq[String]) = {
15    require(channel != null)
16    messages.foreach{
17      message => channel.basicPublish(EXCHANGE_NAME, BINDING_KEY,
            BASIC_PROPS, message.getBytes)
18    }
19  }
20
21  def stopProducer = {
22    connection.close
23  }
24 }
```

Listagem A.2: *Classe SampleProducer*

A listagem A.3 mostra a aplicação que faz uso da classe `SampleProducer`. Nessa listagem fazemos a instanciação e a inicialização de um `producer`, geramos uma sequência com 100 mensagens e as repassamos para o envio. Por fim, encerramos a conexão e implicitamente canais que tenham sido abertos junto a ela.

```
1 object SampleProducerApplication extends Application {
2   val producer = new SampleProducer
3   println("Inicializando SampleProducer")
4   producer.startProducer
5   val messages = for(i <- 1 to 100) yield "A string message #%d".format(i)
6   println("Enviando mensagens")
7   producer.publish {
8     messages
9   }
10  println("Mensagens enviadas com sucesso")
11  producer.stopProducer
12 }
```

Listagem A.3: *Aplicação SampleProducerApplication*

[1] Caso o objeto já exista, um novo não será criado. Caso o objeto existente possua uma configuração diferente, uma exceção é lançada.

A abordagem tomada para exemplificar o recebimento das mensagens foi a mesma tomada para o envio. Criamos a classe `SampleConsumer` para encapsular o recebimento, a confirmação e tratamento das mensagens, como mostrado na listagem A.4. O método `startConsumer`, assim como o método `startProducer` mostrado na listagem A.2, se conecta ao servidor AMQP e abre um canal para interagir com a camada de sessão. Em seguida, ele registra a instância corrente como um consumidor na fila previamente criada com confirmação implícita de recebimento. O método `handleDelivery` é o método invocado a cada mensagem recebida. Podemos notar que o método, além de receber o corpo da mensagem, ainda recebe algumas informações sobre o envio. Os demais métodos são herdados da interface `com.rabbitmq.client.Consumer` e servem como notificadores para outros eventos, como registro (`handleConsumerOk`) ou cancelamento (`handleCancelOk`) de um consumidor na fila. Mais detalhes sobre a interface `com.rabbitmq.client.Consumer` e de suas implementações podem ser encontradas em [Jav].

```
1 class SampleConsumer extends Consumer with CommonAMQP {
2
3   def startConsumer = {
4     val connection = this.connect
5     val channel = connection.createChannel
6     channel.basicConsume(QUEUE_NAME, AUTO_ACK, this)
7   }
8
9   def handleDelivery(consumerTag: String, envelope: Envelope, properties:
        BasicProperties,
10                     message: Array[Byte]): Unit = {
11     println("Mensagem recebida: %s".format(new String(message)))
12   }
13
14   def handleShutdownSignal(consumerTag: String, ex:
        ShutdownSignalException): Unit = {}
15   def handleRecoverOk: Unit = {}
16   def handleConsumeOk(consumerTag: String): Unit = {}
17   def handleCancelOk(consumerTag: String): Unit = {}
18 }
```

Listagem A.4: *Classe SampleConsumer*

A listagem A.5 mostra a aplicação que faz uso da classe `SampleConsumer`.

```
1 object SampleConsumerApplication extends Application {
2   val consumer = new SampleConsumer
3   println("Inicializando SampleConsumer e recebendo mensagens")
4   consumer.startConsumer
5 }
```

Listagem A.5: *Aplicação SampleConsumerApplication*

Referências Bibliográficas

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno e Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004. 1

[Act] Apache ActiveMQ. http://activemq.apache.org/. Último acesso em 30 de Outubro de 2010. 2

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986. 2, 5, 6

[Akk] Akka, Simpler Scalability, Fault-Tolerance, Concurreny & Remoting through Actors. http://www.akka.io. Último acesso em 15/2/2011. 11

[AMQ08] AMQP Working Group. *AMQP Specification v0.10*, 2008. 29

[AMST98] Gul Agha, Ian A. Mason, Scott F. Smith e Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1998. 6, 7

[API] RabbitMQ – API Guide. http://www.rabbitmq.com/api-guide.html. Último acesso em 12/5/2011. 35

[APL04] Apache License 2.0. http://www.apache.org/licenses/, 2004. Último acesso em 11/5/2011. 33

[App09] Apple Inc. *Technology Brief – Grand Central Dispatch*, 2009. 17

[Arm97] Joe Armstrong. The development of erlang. Em *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, páginas 196–203. ACM, Agosto 1997. 10, 33

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007. 8, 10, 19

[Bos] Sérgio Bossa. Actorom – actors concurrency in java, made simple. http://code.google.com/p/actorom. Último acesso em 22/7/2011. 18

[Bri89] Jean Pierre Briot. Actalk: a testbed for classifying and designing actor languages in the smalltalk-80 environment. Em *European Conference on Object-Oriented Programming (ECOOP'89)*, páginas 109–129. Cambridge University Press, 1989. 11

[Bur] Johann Burkard. Generate uuids (or guids) in java. http://johannburkard.de/software/uuid/. Último acesso em 24/7/2011. 26

[Cora] Microsoft Corporation. Asynchronous agents library. http://msdn.microsoft.com/en-us/library/dd492627.aspx. Último acesso em 27/5/2011. 11

[Corb] Microsoft Corporation. Axum programming language. http://msdn.microsoft.com/en-us/devlabs/dd795202. Último acesso em 24/5/2011. 9

[Cor09] Microsoft Corporation. *Axum – Language Overview*, Junho 2009. 8

[Cro06] Douglas Crockford. The application/json media type for javascript object notation (json). Relatório Técnico RFC 4627, The Internet Engineering Task Force (IETF), Jul 2006. 12

[FPV98] Alfonso Fuggetta, Gian Pietro Picco e Giovanni Vigna. Understanding code mobility. *IEEE Trans. Softw. Eng.*, 24:342–361, 1998. 7

[Goo] Protocol Buffers – Google's data interchange format. http://code.google.com/p/protobuf. Último acesso em 27/5/2011. 12, 25

[Gro] AMQP Working Group. Advanced message queuing protocol. http://www.amqp.org. Último acesso em 5/5/2011. 29, 45

[Har] Mark Harrah. Library for describing binary formats for Scala types. https://github.com/harrah/sbinary. Último acesso em 27/5/2011. 12

[HBS73] Carl Hewitt, Peter Bishop e Richard Steiger. A universal modular actor formalism for artificial intelligence. Em *Proceedings of the 3rd international joint conference on Artificial intelligence*, páginas 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. 7

[Hew71] Carl Hewitt. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A language for Manipulating models and proving theorems in a robot.* Tese de Doutorado, MIT, 1971. 7

[HO09] Philipp Haller e Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202 – 220, 2009. Distributed Computing Techniques. 11, 12

[IBM] IBM Websphere MQ - Software. http://www-01.ibm.com/software/integration/wmq/. Último acesso em 30 de Agosto de 2011. 2

[Ins81] Information Sciences Institute. Transmisson control protocol – protocol specification. Relatório Técnico RFC 793, University of Southern California, Set 1981. 37

[Jav] RabbitMQ Java API 2.4.1 Javadoc. http://www.rabbitmq.com/releases/rabbitmq-java-client/v2.4.1/rabbitmq-java-client-javadoc-2.4.1. Último acesso em 12/5/2011. 34, 59

[JBoa] JBoss Messaging. http://www.jboss.org/jbossmessaging. Último acesso em 30 de Agosto de 2011. 2

[JBob] Netty – the Java NIO Client Server Socket Framework. http://www.jboss.org/netty. Último acesso em 27/5/2011. 12

[JOW07] Simon Jones, Andy Oram e Greg Wilson. *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly))*. O'Reilly Media, Inc., 2007. 2

[KA95] WooYoung Kim e Gul Agha. Efficient support of location transparency in concurrent object-oriented programming languages. Em *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, página 39, New York, NY, USA, 1995. ACM. 7

[Kim97] Wooyoung Kim. *THAL: An Actor System For Efficient and Scalable Concurrent Computing.* Tese de Doutorado, University of Illinois at Urbana-Champaign, 1997. 11

[KML93] Dennis Kafura, Manibrata Mukherji e Greg Lavender. Act++ 2.0: A class library for concurrent programming in c++ using actors. *Journal of Object-Oriented Programing*, 6:47–55, 1993. 11

[KSA09] Rajesh K. Karmani, Amin Shali e Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. Em *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, páginas 11–20, New York, NY, USA, 2009. ACM. 11

[Lee] Jacob Lee. Parley. http://osl.cs.uiuc.edu/parley/. Último acesso em 26/5/2011. 11

[LGP07] GNU lesser general public license. http://www.gnu.org/licenses/lgpl.html, 2007. Último acesso em 11/5/2011. 33

[Lie87] Henry Lieberman. *Concurrent object-oriented programming in Act 1*, páginas 9–36. MIT Press, Cambridge, MA, USA, 1987. 8

[Mas] Ashton Mason. Theron multiprocessing library. http://theron.ashtonmason.net/. Último acesso em 26/5/2011. 11

[MPL] Mozilla Public License. http://www.mozilla.org/MPL/MPL-1.1.html. Último acesso em 11/5/2011. 33

[OAC+06] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stephane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoonn, Erik Stenman e Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Relatório Técnico LAMP 2006-001, EPFL, 2006. 3

[Ope] The Actor Foundry: A Java-Based Actor Programming Environment – Open Systems Laboratory, University of Illinois at Urbana-Champaign. http://osl.cs.uiuc.edu/af. Último acesso em 27/5/2011. 11

[PA94] Rajendra Panwar e Gul Agha. A methodology for programming scalable architectures. *J. Parallel Distrib. Comput.*, 22:479–487, 1994. 7

[Poi] Robey Pointer. Simple config and logging setup for Scala. http://github.com/robey/configgy. Último acesso em 5/7/2011. 25

[Pro03] Java Community Process. *JSR-000914 JavaTM Message Service (JMS) API*, Dezembro 2003. 33

[Qpi] Apache Qpid: Open Source AMQP Messaging. http://qpid.apache.org/. Último acesso em 11/5/2011. 33

[Reta] Mike Rettig. Jetlang – Message based concurrency for Java. http://code.google.com/p/jetlang. Último acesso em 27/5/2011. 11

[Retb] Mike Rettig. Retlang – Message based concurrency in .Net. http://code.google.com/p/retlang. Último acesso em 27/5/2011. 11

[RMQ] RabbitMQ - Messaging that just works. http://www.rabbitmq.com/. Último acesso em 11/5/2011. 33, 38

[Rub] Rubinius. http://rubini.us/. Último acesso em 26/5/2011. 11

[Sca] Scalaz: Type Classes and Pure Functional Data Structures for Scala. http://code.google.com/p/scalaz. Último acesso em 27/5/2011. 11

[Sil08] Jonathan Sillito. Stage: exploring erlang style concurrency in ruby. Em *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, páginas 33–40. ACM, 2008. 11

[SL05] Herb Sutter e James Larus. Software and the concurrency revolution. *Queue*, 3(7):54 – 62, 2005. 2

[SM08] Sriram Srinivasan e Alan Mycroft. Kilim: Isolation-typed actors for java. Em *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, páginas 104–128. Springer-Verlag, 2008. 11

[ST95] Nir Shavit e Dan Touitou. Software transactional memory, 1995. 2

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30:202 – 210, 2005. 2

[Tis] Christian Tismer. Stackless python. http://www.stackless.com/. Último acesso em 26/5/2011. 11

[VA01] Carlos A. Varela e Gul Agha. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36:20–34, 2001. 8

[Zer] Less is More - zeromq. http://www.zeromq.org/. Último acesso em 11/5/2011. 33

Índice Remissivo