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, fevereiro de 2012

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 Enfileiramento 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 troca de Mensagens Java (*Java Messaging 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*)
RPC	Chamada Remota de Procedimento (*Remote Procedure Call*)
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 mensagens 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), já que ambos enviam e recebem mensagens. No contexto de MOMs, os papéis de servidor ou cliente são definidos pela semântica da troca de mensagens. Uma entidade pode estar atuando ora como provedora de um serviço, ora como cliente de um serviço. Portanto, os termos "cliente" e "servidor", não são aplicáveis nesta classe de sistemas do mesmo modo como são aplicáveis em sistemas baseados em RPC.

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 utilizadas para garantir que mensagens recebidas pelo MOM sejam salvas de modo persistente. A remoção de uma mensagem de uma fila transacional ocorre somente após a confirmação do seu recebimento pelo destinatário. No caso de avarias no sistema, mensagens previamente salvas pelo MOM e que não tiveram seu recebimento confirmado não são perdidas. Uma vez que o sistema tenha sido restabelecido, essas mensagens podem ser entregues aos seus destinatários. 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 bro-*

kers 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 ao 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 [JBo], IBM Websphere MQ [IBM] (mais conhecido como MQ Series) e Apache Active MQ [Act].

O padrão AMQP [Gro] (*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 usufruirem 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, e ajudaram a impulsionar a pesquisa de 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. No modelo de atores, 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, que possui tipagem estática e inferência de tipos 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 por componentes 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 características são:

- Scala possui certa semelhança com Java e C#, de modo que tanto programas escritos em Scala podem utilizar bibliotecas escritas em Java ou C#, quanto o inverso.
- Scala possui um modelo uniforme para objetos, em que 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 parâmetros, devolvidas por uma função, ou ainda atribuídas a variáveis.
- Scala permite construções via extensão de classes e combinações de feições (*traits*). Feições possuem definições de métodos e campos assim como uma classe abstrata, porém não definem construtores. São importantes unidades para reuso de código em Scala, já que classes ou objetos podem ser combinados (*mixin*) com diversas feições.
- 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 object*, que representa uma instância única de uma classe. *Singleton objects* definidos com a construção `object` ao invés de `class`. Um *singleton object* que tenha o mesmo nome que uma classe é denominado objeto acompanhante (*companion object*) dessa classe.
- Scala possui suporte a *currying*, que em conjunto 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

Este trabalho teve como objetivo criar uma implementação em Scala do modelo de atores que use o padrão AMQP para o transporte de mensagens entre atores remotos. Geramos um protótipo baseado na implementação do modelo de atores do projeto Akka. Substituímos o mecanismo de transporte do Akka por um que utiliza um *message broker* AMQP.

1.5 Contribuições

As principais contribuições deste trabalho são as seguintes:

- uma implementação do modelo de atores que é escrita em Scala e utiliza um *message broker* AMQP como mecanismo de transporte de mensagens entre atores remotos;
- uma camada de abstração para acesso à biblioteca de conexão do *message broker* AMQP. Criada para dar suporte ao nosso protótipo, essa camada foi implementada com atores e sua principal motivação é prover acesso concorrente seguro aos serviços oferecidos pela biblioteca de conexão do *message broker*.

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 as camadas do padrão AMQP, mencionamos algumas das implementações desse protocolo e examinamos algumas das principais classes da biblioteca para clientes Java da implementação utilizada neste trabalho. No capítulo 5, apresentamos a camada de abstração criada para prover acesso concorrente seguro às classes de comunicação da biblioteca para clientes Java da implementação de AMQP que utilizamos. No capítulo 6, apresentamos nossa implementação de atores remotos que utiliza a camada de abstração descrita no capítulo 5.

Finalmente, no Capítulo 8, discutimos as conclusões obtidas neste trabalho. Apresentamos também uma comparação de desempenho entre a implementação original de atores remotos do projeto Akka e o protótipo desenvolvido neste trabalho.

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 mensagens 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 (mapeamento de mensagens em 3-tuplas), 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 ao ator remetente da mensagem. Há três maneiras de um ator, ao receber uma mensagem m, passar a conhecer um destinatário ao 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 do modelo de atores é a existência de imparcialidade (*fairness*) no processamento das mensagens: tanto a entrega das mensagens é garantida como o progresso no seu processamento [AMST98]. A garantia de entrega de mensagens é uma forma de imparcialidade e, no contexto de atores, significa que uma mensagem que foi depositada na fila de mensagens de um ator não deve ficar armazenada indefinidamente sem ser, mais cedo ou mais tarde, processada pelo ator. A ordem de chegada de mensagens obedece uma ordem linear e fica a cargo da implementação do modelo arbitrar no caso de conflitos. A garantia de entrega de mensagens não pressupõe 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, em que 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 pressupor 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 interfere 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. Assim, os comportamentos B_1 e B_2 podem ser comportamentos idênticos.

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 infraestrutura e permite que programas sejam desenvolvidos de modo distribuído, sem que a real localização de seus atores seja conhecida. Uma consequência direta da transparência da localidade é a capacidade de se mover atores entre os diferentes nós de uma rede de computadores.

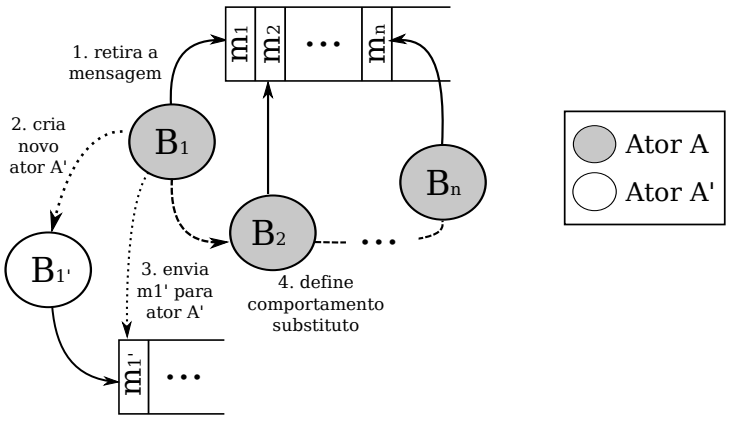

Figura 2.1: *Máquina de estados de um ator.*

Apesar do modelo ser bem explícito e claro em relação à interação ser via troca de mensagens e dizer que o estado de um ator não deve ser compartilhado, implementações poderiam permitir que um ator fizesse acesso ao estado interno de outro ator. Por exemplo, um ator poderia invocar algum método de outro ator. Mesmo que a interface de um ator não permita o compartilhamento do seu estado, mensagens enviadas poderiam expor um compartilhamento indesejado de estado com atores residentes no mesmo processo ("atores co-locados"). O uso de mensagens imutáveis ou passadas via "cópia funda" é uma abordagem mais apropriada para evitar um compartilhamento indesejado de memória entre atores co-locados.

Mobilidade é definida como a habilidade de se poder mover um programa 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 tanto o código quanto o estado da execução [FPV98]. Em um sistema baseado em atores, mobilidade fraca permite que atores que não possuam mensagens em suas filas e não estejam fazendo processamento algum sejam transferidos para outros nós.

Pelo fato do 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 reconfigurabilidade [PA94].

A semântica 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 em que 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 analisavam 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, pois tanto atores quanto agentes possuem intenções, recursos, monitores de mensagens e um agendador [HBS73].

Em 1975, Irene Greif desenvolveu um modelo abstrato de atores orientado a eventos em que cada ator guardava um histórico dos seus eventos [Gri75]. O objetivo do estudo era analisar a relação causal entre os eventos. Em 1977, Baker e Hewitt formalizaram um conjunto de axiomas para computação concorrente [HB77]. 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 1979 por Attardi e Hewitt [HAL79]. Um guardião é uma entidade que regula o uso de recursos compartilhados. Guardiões incorporam explicitamente a noção de estado e são responsáveis por agendar o acesso e fazer a proteção dos recursos. No mesmo ano, Hewitt e Atkinson definiram um conceito relacionado ao de guardião denominado seriador (*serializer*) [HA79]. 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 de dados primitivos e funções da linguagem usada em sua implementação.

Gul Agha definiu em 1986 um sistema simples de transição para atores [Agh86]. Em seu trabalho foram desenvolvidos os conceitos 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. Em 1987, esse modelo foi implementado na linguagem Acore, por Carl Manning [Man87], e em 1988, na linguagem Rosette, por Tomlinson e outros [TKS+88].

2.3 Implementações

O suporte ao modelo de atores pode estar disponível em uma linguagem de programação, seja fazendo parte da estrutura da linguagem, ou ainda como uma biblioteca. Apresentamos na subseção 2.3.1 algumas linguagens de programação baseadas no modelo de atores. Essas linguagens possuem primitivas que facilitam a criação de sistemas baseados em atores. Apresentamos na subseção 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 interajam entre si pudessem ser traduzidos naturalmente para código. Pelo fato de ser uma linguagem da plataforma .Net, Axum pode

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

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 que processam as mensagens recebidas por 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 estado. Uma instância de domínio pode ter atributos que são usados para compartilhamento seguro e controlado de informações.

Apesar de Axum 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, que por sua vez pode ser compilado para *bytecode* e ser executado sobre a JVM.

A linguagem possui primitivas para a criação, uso e organização dos atores. A construção `behavior` é semelhante à construção `class` de Java, e é utilizada para definir os métodos que definem o comportamento do ator. Há uma relação de herança em *behaviors*, similar à herança de classes Java. A hierarquia de *behaviors* segue regras análogas às da hierarquia de classes Java: não existe herança múltipla e um *behavior* podem implementar zero ou múltiplas *interfaces*. O *behavior* `salsa.language.UniversalActor` é análogo a classe `java.lang.Object`, e todos os *behaviors* o estendem 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 estende `java.lang.Thread`, de modo que cada ator é executado 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 devolver valores que precisarão ser repassados a outros atores. Os valores devolvidos 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 produz um valor que 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 prossiga depois de 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 às funções de ordem superior, na qual 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. Erlang possui o módulo `erl_prim_loader` que auxilia no carregamento de módulos em nós remotos. 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].

[2]Rubinius é uma implementação da linguagem Ruby que possui uma máquina virtual escrita em C++.

Pelo fato de nossa implementação ter sido desenvolvido sobre a JVM, mais especificamente na linguagem Scala, optamos por não descrever todas as bibliotecas listadas acima, pois isso nos distanciaria do escopo deste trabalho. Em [KSA09], Karmani e Agha apresentam uma análise comparativa de algumas implementações de atores para a JVM. Essa análise traz comparações entre as várias implementações da semântica de execução e das abstrações. Apresentamos a seguir informações sobre duas bibliotecas de atores Scala que foram consideradas como opções e analisadas 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. Nessa biblioteca os atores foram projetados como objetos baseados em *threads* Java. Elas 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`.

No bloco `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, pois dessa forma um ator que não estiver em execução cederá sua *thread* para que ela execute outro ator. Do ponto de vista de funcionalidade, o método `receive` é bloqueante e pode devolver valores, enquanto que o método `react`, além de não devolver valores, faz com que o ator passe a reagir aos eventos (recebimentos de mensagens). Da perspectiva do programador, uma implicação prática do uso de `react` em vez de `receive` é o uso da estrutura de controle `loop` ao invés de laços tradicionais, para indicar que o ator deve continuar reagindo após o processamento de uma mensagem (o emprego de laços tradicionais bloquearia a *thread*) [HO09].

Além de métodos para envio de mensagens equivalentes às primitivas de Erlang, a biblioteca de atores de Scala implementa dois métodos adicionais, que facilitam o tratamento de algumas necessidades específicas. Esses métodos são: `!?`, que faz envio síncrono e aguarda uma resposta dentro de um tempo limite especificado e `!!`, que 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. 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 o 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 e *sockets* TCP.

[3] Idealmente o tamanho do *thread pool* corresponde ao número de núcleos do processador.

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 dá 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, embora também siga 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 do comportamento dos atores de Erlang e Scala, em que 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, a de 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, e `!!!`, que é semelhante ao método `!!` da biblioteca de atores de Scala, o qual 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 [JNY], um arcabouço para comunicação assíncrona dirigido a eventos e baseado em *sockets*. Esse arcabouço oferece facilidades para compressão de mensagens, e tais facilidades são utilizadas 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 motivada pelos fatores a seguir:

- O projeto Akka tem código aberto.
- A implementação de atores é escrita em Scala.
- O Akka possui uma implementação mais completa de atores remotos em comparação com outras bibliotecas que examinamos.
- O grau de acomplamento entre a implementação de atores remotos e o mecanismo de transporte é relativamente baixo.
- O projeto tem grande volume de atividade em sua comunidade de usuários e de desenvolvedores.

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

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

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 examinamos 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 examinamos 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.

3.1 Atores locais

A definição de atores locais acontece se combinando a feição `akka.actor.Actor`, e provendo 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`. Essas referências são imutáveis, seriáveis, identificáveis, possuem um relacionamento único com a definição do ator e armazenam o endereço do nó onde foram criadas.

A feição `Actor` possui em sua declaração uma referência para o seu `ActorRef`, acessível via atributo `self`. 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. Vale mencionar que, caso de atores locais, `self` referencia uma instância de `akka.actor.LocalActorRef`.

```
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 mos-

trado na listagem 3.2. Nessa listagem, a instância do ator é criada por reflexão com base no tipo da classe (`SampleActor`). O construtor utilizado durante a reflexão é o construtor padrão. 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.

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

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

Nas listagens 3.2 e 3.3, 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 executa mais ação alguma.

```
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 colocadas 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.

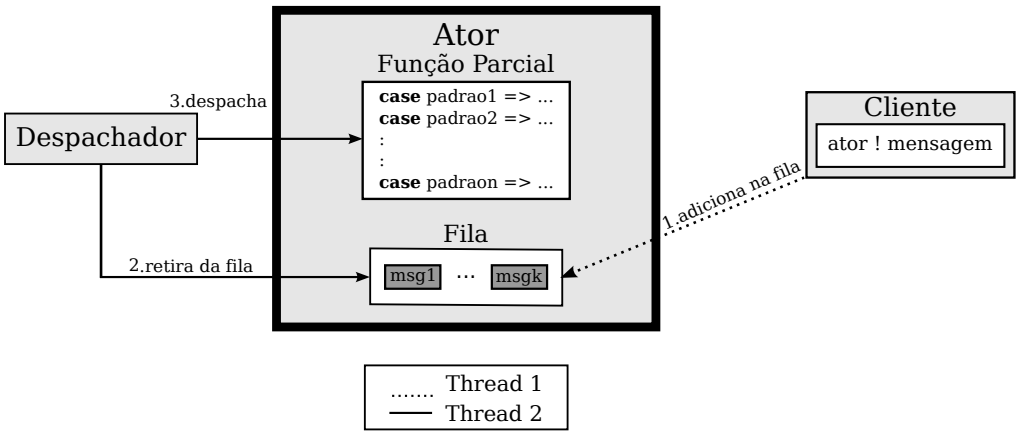

Figura 3.1: *Envio e despacho de mensagens para atores locais.*

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

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

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:

- Despachador impulsionado por eventos: É um despachador 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. É definido na classe `ExecutorBasedEventDrivenDispatcher`.

- Despachador impulsionado por eventos com balanceamento de carga: É um despachador semelhante ao despachador anterior, já que também é impulsionado por eventos. Definido na classe `ExecutorBasedEventDrivenWorkStealingDispatcher`, é um despachador para 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: É um despachador 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]. Sua grande vantagem, é a capacidade de agrupar diversos eventos gerados pela aplicação, por exemplo diversas mensagens recebidas, gerando uma única tarefa assíncrona. Este despachador é definido na classe `HawtDispatcher`.

- Despachador impulsionado por *threads*: É o despachador mais simples de todos os despachadores, 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 uma mensagem, caso o limite superior da fila tenha sido atingido (para o caso em que há um limite informado). Este despachador é definido na classe `ThreadBasedDispatcher`.

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 despachadores sejam criados. Por padrão, os despachadores são criados no pacote `akka.dispatch`.

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   ...
```

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

```
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.

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 funcionalidades básicas para resultados futuros. O 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 pelo arcabouço. A feição `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: *Feição 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. Assim como para um envio assíncrono, a mensagem é colocada da fila do ator. Contudo, uma referência para a instância criada no passo anterior é colocada junto da mensagem.
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 teve o resultado "completado" (seja com um valor ou com uma exceção) também 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 na devolução 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 devolvido 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 booleano `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. Permanente: Indica que o ator possui um ciclo de vida permanente e sempre deve ser reiniciado no caso de erros. Este ciclo é definido na classe `Supervision.Permanent`.

2. Temporário: 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 normalmente, ou seja, como se o método `exit` tivesse sido invocado. O processo normal ainda faz uma invocação ao método `postStop`. Este ciclo é definido na classe `Supervision.Temporary`.

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 em que ocorreu o erro.
- O método `postRestart` é invocado na nova instância.
- A nova instância criada utiliza a mesma `ActorRef` do ator em que ocorreu o erro, logo, as duas instâncias 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: Quando essa estratégia é utilizada, exceções lançadas por um ator sob supervisão que possua um ciclo de vida permanente, fazem com que somente o ator seja reiniciado. Esta estratégia é definida na classe `Supervision.OneForOneStrategy`.

2. Todos por um: Quando essa estratégia é utilizada, exceções lançadas por um ator sob supervisão que possua um ciclo de vida permanente, fazem com que todos os atores sob o mesmo supervisor sejam reiniciados. Esta estratégia é definida na classe `Supervision.AllForOneStrategy`.

```
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 estratégia 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.

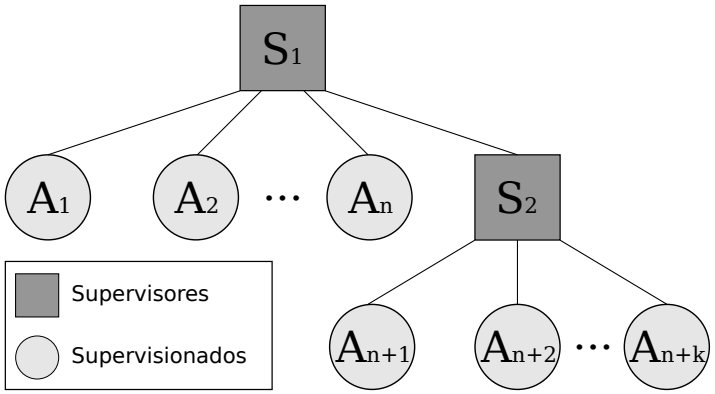

Figura 3.2: *Hierarquia de supervisão de atores.*

3.2 Atores remotos

No contexto de envios de mensagens para atores remotos, o termo cliente se refere ao processo (máquina virtual) onde residem as referências para os atores remotos e as demais entidades que, normalmente, iniciam o envio de mensagens. 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 infraestrutura de atores remotos utiliza os seguintes elementos:

- `RemoteServerModule`: É uma feição que define as responsabilidades do componente utilizado 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 feição 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. Ademais, oferece também suporte em tempo de execução para a infraestrutura de atores do lado do cliente, provendo uma série de serviços não visíveis 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`: 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. É acessível via `Actor.remote`.
- `RemoteActorRef`: É uma classe equivalente a `LocalActorRef`, porém utiliza o suporte remoto para fazer envio das mensagens.

A biblioteca de atores remotos do Akka implementa os componentes de suporte remoto com o auxílio do JBoss Netty [JNY]. Apresentamos na figura 3.3 como os componentes descritos acima estão relacionados. Dividimos a figura em duas camadas, sendo a primeira a camada de interface remota, e a segunda a camada de implementação. A camada de implementação é associada em tempo de execução, permitindo que demais implementações 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 hospedeiro 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.

As referências para os atores remotos são obtidas via método `actorFor`. Esse método possui diversas sobrecargas, sendo que a mais simples recebe como argumentos o nome do ator, o endereço do hospedeiro e a porta do *remote server* onde o ator foi registrado. A linha 2 da listagem 3.10 mostra como uma aplicação cliente obtém uma referência para o ator. Podemos notar que, no uso do ator na linha 3, não há nenhuma indicação que caracterize um envio remoto. O método `actorFor` devolve, 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, caso a entidade que está enviando a mensagem resida no mesmo nó que o ator, uma instância de `LocalActorRef` é devolvida.

```
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.*

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

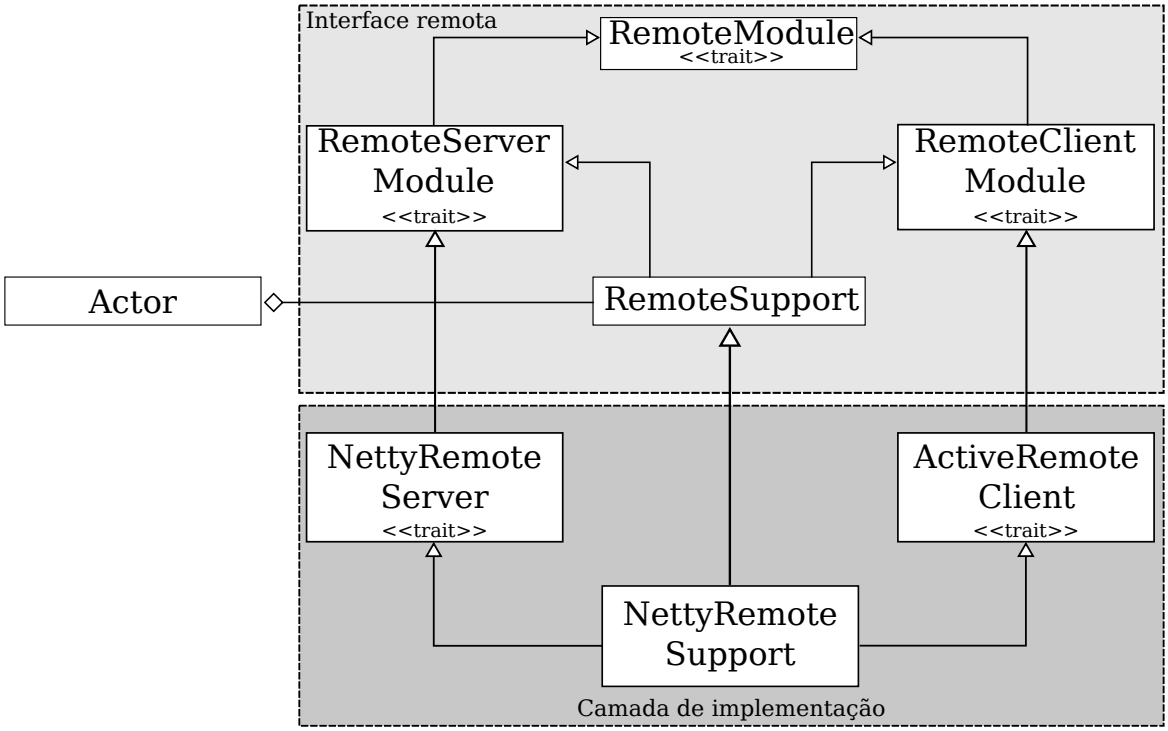

Figura 3.3: *Relacionamento entre os componentes remotos.*

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. Esses detalhes 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.

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

Listagem 3.10: *Aplicação SampleRemoteClient.*

3.2.1 Fluxo de envio das mensagens

O envio assíncrono 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 de envio 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 então 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* utiliza 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` repassa a mensagem para o `ServerBootstrap` ao qual se está conectada. 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 lhe foi associado.

5. A implementação do *handler* feita no Akka repassa a mensagem para o seriador para que a desseriação seja feita.

6. O seriador desseria a mensagem, deixando-a no formato original, e em seguida, repassa novamente a mensagem ao *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.

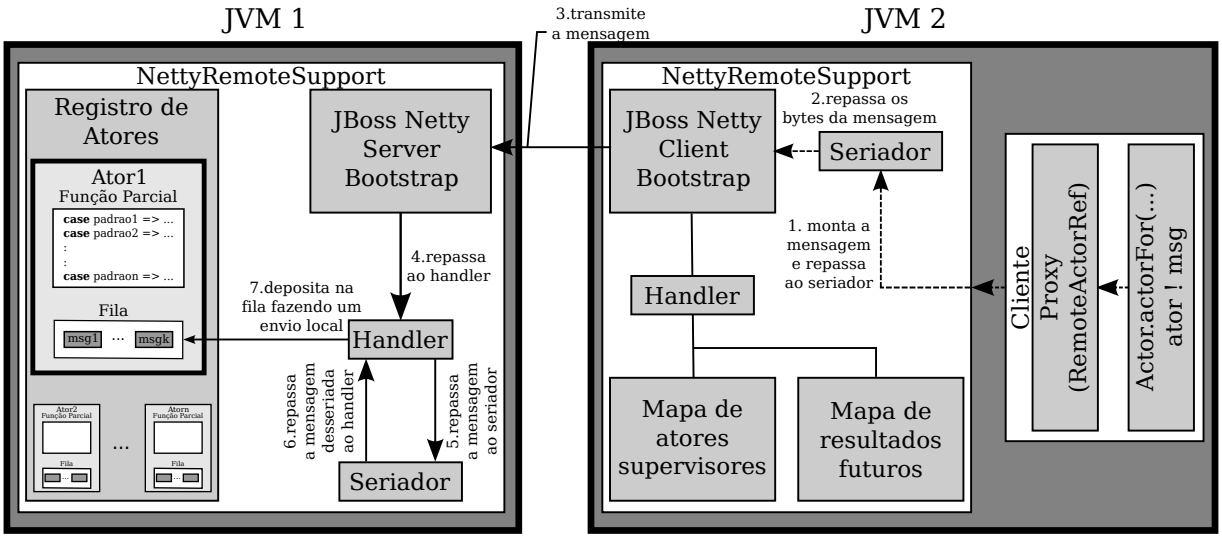

Figura 3.4: *Fluxo de envio de mensagens para atores remotos.*

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 da mensagem ser enviada, uma cópia da referência da instância do resultado futuro que é devolvida 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 feito via métodos `!!` ou `!!!`, o conteúdo da mensagem (seja um resultado de sucesso ou uma exceção) é utilizado para completar o resultado futuro. Já no caso de mensagens originadas

no servidor e que possuam um ator supervisor, o ator supervisor é localizado e é feito um envio local assíncrono da mensagem para ele.

O Akka possui uma configuração opcional de segurança em que cada JVM que esteja executando o arcabouço pode definir um *cookie*. O *cookie* nada mais é do que uma chave que o usuário pode definir e que faz parte das informações do 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. Caso os valores sejam iguais, o passo 7 será executado. Caso contrário, uma exceção de segurança será lançada na JVM onde a mensagem foi recebida e o passo 7 não será 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` é o ponto de entrada para todas as configurações que são parametrizáveis 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 configuração de atores remotos são relacionadas ao Netty. 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 se a comunicação deve ser autenticada via *cookies* pré-definidos, qual o *cookie* de segurança que deve 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. As propriedades relacionadas aos atores remotos estão organizadas em objetos dentro do módulo `RemoteShared`. Existe ainda uma classe auxiliar que é utilizada para acessar os módulos dos subprojetos do Akka via reflexão chamada de `ReflectiveAccess`. A camada de implementação para o transporte de mensagens entre atores remotos é instanciada nessa classe. A listagem 3.11 mostra a seção referente às configurações do módulo de atores remotos.

```
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 biblioteca permite que protocolos sejam definidos com a palavra-chave `message`. Protocolos podem conter outros protocolos e tipos primitivos de dados, como por exemplo números, textos, valores booleanos 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 onde se originou 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 da referência remota de um ator que faz um envio de mensagem ser enviada junto à mensagem, foi definido um protocolo para as `RemoteActorRefs`. Esse protocolo, mostrado na listagem 3.14, 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 que indica o tempo máximo que o ator

deve ficar bloqueado durante um envio síncrono (linha 4). O protocolo `AddressProtocol` utilizado na linha 3, possui atributos para o endereço do hospedeiro e a 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, é apresentado na listagem 3.15, o protocolo utilizado para as mensagens remotas. Esse protocolo é composto pelo protocolo para mensagens (linha 4) e pelo protocolo para referências remotas (linha 7), além de outros protocolos menores. 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 booleano 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 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 feições 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 combinar a feição `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.

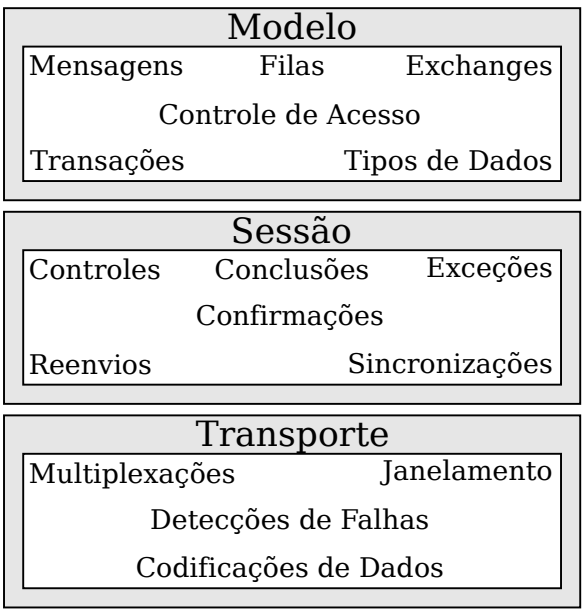

Figura 4.1: *Camadas do padrão AMQP.*

Na seção 4.1 descrevemos a camada de modelo, que é a camada em que 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, já que 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 a implementação que escolhemos para o desenvol-

vimento deste trabalho.

4.1 A camada de modelo

A camada de modelo é a camada em que 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 desta camada 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*), em que 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*), em que 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*, em que 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 como eles se relacionam.

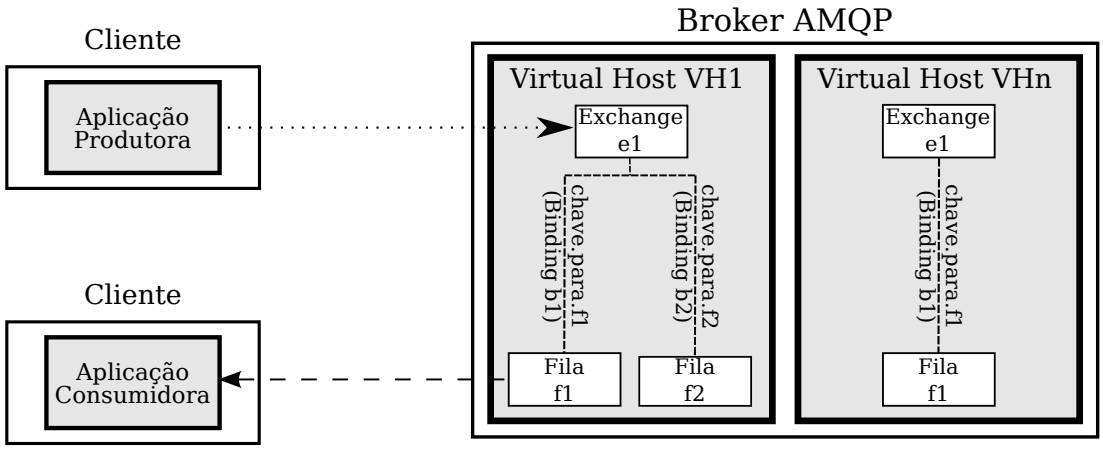

Figura 4.2: *Componentes da camada de modelo do padrão AMQP.*

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.

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 deve especificar a *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 os passos do envio e recebimento de uma mensagem.

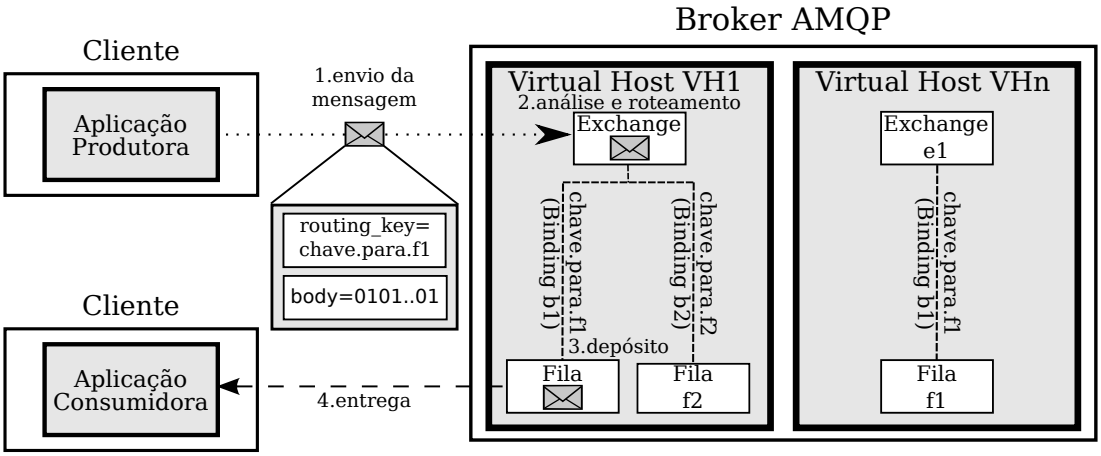

Figura 4.3: *Fluxo de uma mensagem no padrão AMQP.*

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.

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 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 a devolução de resultados assíncronos. O identificador do comando é disponibilizado para a camada de modelo e, quando um resultado é devolvido 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 descartar, seguramente, 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 que o gerenciamento do estado seja 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 na rede: Para que o sistema possa se recuperar no caso de falhas na 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 [ZMQ] 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#. 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 vinte 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 RabbitMQ. Tomamos como base para nossa decisão os seguintes fatores:

- 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 e sua biblioteca para clientes Java ser 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.

Apresentamos a seguir uma visão geral de alguns dos principais componentes da biblioteca para clientes Java do projeto RabbitMQ e como eles estão relacionados.

4.4.1 RabbitMQ - Biblioteca para clientes Java

A biblioteca para clientes Java disponibilizada pelo projeto RabbitMQ [API] 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` no qual 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.*

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

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 estão relacionados. 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. Para casos como este, 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.*

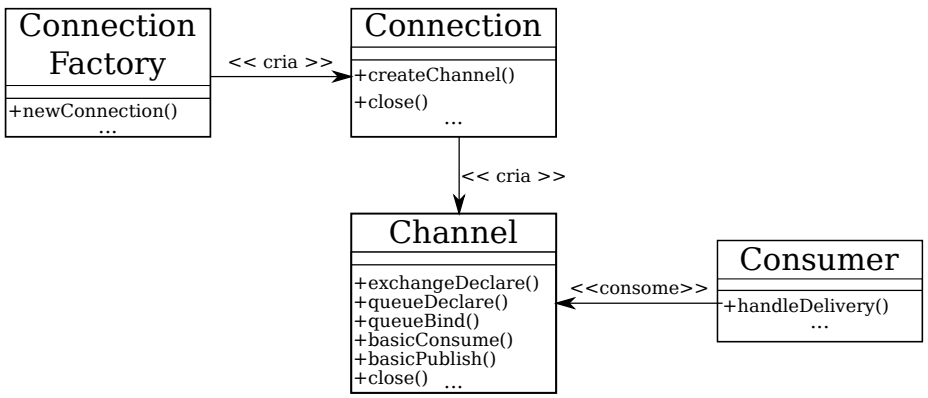

Figura 4.4: *RabbitMQ Java API – Relação entre classes de transporte e sessão.*

Para finalizar este capítulo, é importante destacar que a implementação de canais [RAG] 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 estrutura 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 hospedeiro onde está a outra entidade. Nos casos em que as entidades estão em nós que não fazem parte de uma rede interna, a informação da 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 abordagem 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 e da porta e ficam registrados 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 hospedeiro e da 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 hospedeiro 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 hospedeiro 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 *message broker* AMQP

A substituição por um mecanismo baseado em troca de mensagens naturalmente introduz novas características à comunicação. As entidades passam a não estarem mais conectadas diretamente, já que o *message 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 *message 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 nesta porta.

Tanto a rotulação das entidades por hospedeiro e porta ou por nome e hospedeiro 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 *message 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. Essas tarefas devem ter sido executadas previamente.

O suporte ao desenvolvimento dos novos componentes da camada de implementação de transporte remoto do projeto Akka 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 *message broker* RabbitMQ.

5.2.1 Pontes AMQP

O módulo `AMQPBridge` é responsável por fornecer classes que agem como pontes de comunicação para o *message broker*. Juntas, suas classes definem uma interface com funcionalidades básicas para troca de mensagens como envios, recebimentos, tratamento de mensagens rejeitadas 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 *message broker* AMQP. O construtor de `AMQPBridge` recebe como argumentos o nome que identifica a ponte e uma conexão para um *message broker* AMQP. Essa classe define ainda o padrão de nomeação utilizado na criação dos objetos, como mostra a listagem 5.1. A classe possui também um objeto companheiro que define métodos para a criação de instâncias das subclasses de `AMQPBridge`.

- `MessageHandler`: É a feição que define os métodos a serem invocados quando mensagens forem recebidas ou envios feitos pela entidade forem rejeitados pelo *message broker*.

- `ServerAMQPBridge`: É uma das duas subclasses de `AMQPBridge` e define o comportamento para entidades que atuam como servidoras. 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 subclasse de `AMQPBridge` e define o comportamento para entidades que atuam como clientes. Essa classe tem em seu construtor um parâmetro adicional em relação à classe `ServerAMQPBridge`. O parâmetro `id` é utilizado 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(name != 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 *message broker*. A enumeração `StorageAndConsumptionPolicy` define um conjunto de valores para 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 seguintes políticas:

1. `DURABLE`: Define uma configuração no qual os objetos criados são duráveis, ou seja, continuam a existir caso o *message broker* seja reiniciado.

2. `TRANSIENT`: Define uma configuração no qual os objetos criados são transientes, ou seja, deixam de existir caso o *message broker* ser reiniciado.

3. `EXCLUSIVE_AUTODELETE`: Define uma configuração no qual os objetos estão atrelados ao ciclo de vida da aplicação que os criou. Os objetos são removidos pelo *message broker* quando a conexão em que os objetos foram criados é fechada. Ademais, as filas criadas por essa configuração possuem acesso exclusivo a um único consumidor. O consumidor fica atrelado ao canal em que 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 *message broker*. Como mostrado na subseção 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 *message broker*.

Para que os canais possam ser acessados concorrentemente de modo seguro, optamos por encapsulá-los em atores Akka. A ideia 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 *message broker*.

A classe `SupervisedConnectionWrapper` define uma conexão supervisionada, expondo uma interface muito simples com apenas quatro métodos. Os métodos dão suporte a 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 ou 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 estende a classe `DefaultConsumer` do RabbitMQ, e utiliza a implementação do `MessageHandler` para a notificação de eventos como mensagens recebidas ou mensagens rejeitadas.
- `ChannelActor`: É a feição 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 *message broker*. Uma outra responsabilidade desse ator é supervisionar os atores que lhe solicitaram a abertura de canais.

- `AMQPSupervisor`: É a feição que define o comportamento padrão a ser utilizado por um ator que será responsável por supervisionar os atores de conexão. Essa feição define ainda métodos que interagem com a fábrica de conexões do RabbitMQ para a criação de novas conexões.

- `AMQPConnectionFactory`: É a classe raiz 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 como os módulos `AMQP`, `AMQPBrigde` e `ConnectionPoolSupervisor` estão relacionados.

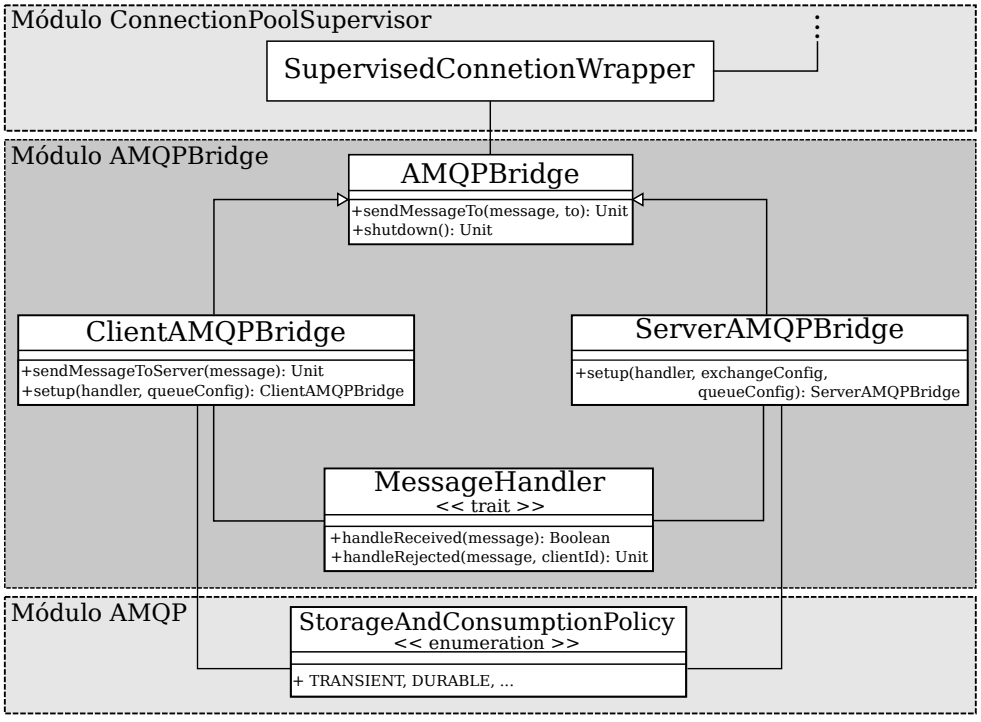

Figura 5.1: *Módulos de acesso ao broker AMQP.*

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

A criação de objetos como filas e *exchanges* no *message 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. 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 e inicializada;
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.

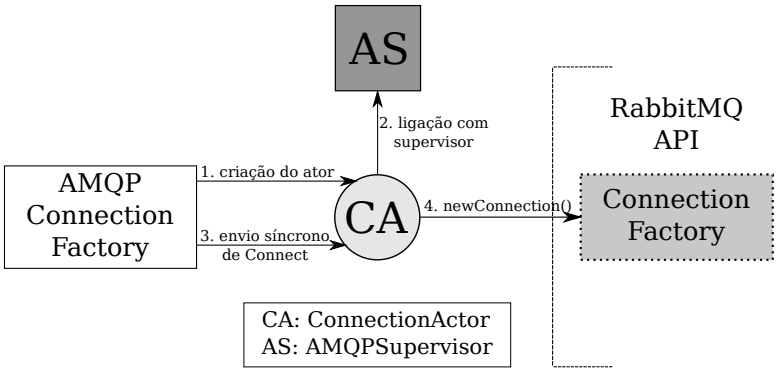

Figura 5.2: *Passos para a criação do ator de conexão.*

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 seis passos:

1. uma instância de `ReadChannelActor` é criada e inicializada;
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;

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

4. a mensagem `StartReadChannel` é processada pelo ator (seu processamento faz um 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;

6. o novo canal é enviado como resposta para `ReadChannelRequest`.

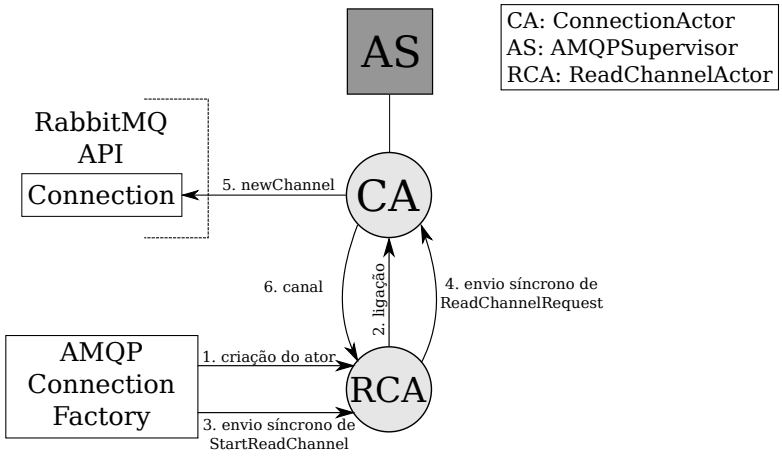

Figura 5.3: *Passos para a criação do ator do canal de leitura.*

O processo para criação do ator responsável pelo canal de escrita é bem semelhante, com diferenças nas mensagens enviadas nos passos 3 e 4. São utilizadas as mensagens `StartWriteChannel` e `WriteChannelRequest`, respectivamente. No passo 5, o novo canal é solicitado para a conexão de escrita. As conexões de escrita e de 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 *message broker*, são lidos de um arquivo de propriedades. Os detalhes sobre esse arquivo são apresentados no capítulo 6.

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 rejeitadas (*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 definido;

8. a implementação de `MessageHandler` informada é utilizada na associação do consumidor na fila criada no passo 6.

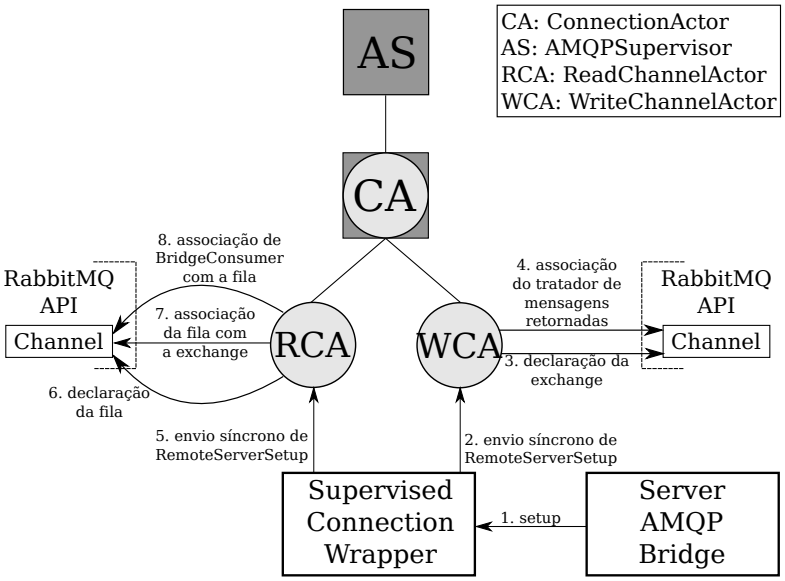

Figura 5.4: *Passos de configuração da classe ServerAMQPBridge.*

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:

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 rejeitadas;

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.

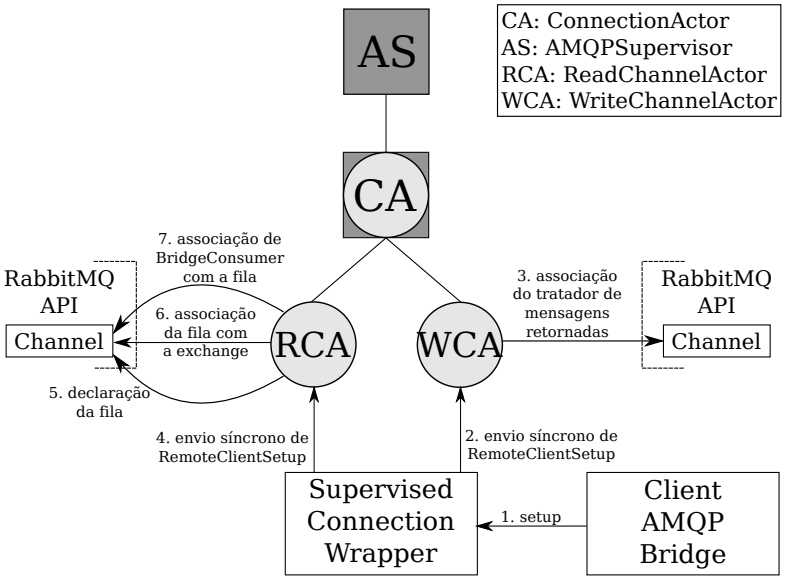

Figura 5.5: *Passos de configuração da classe ClientAMQPBridge.*

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 inicialização 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`.

Mostramos na figura 5.6 como ficaria, da perspectiva do *message broker*, 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. A ponte servidora define a fila de entrada de mensagens. As pontes clientes definem as filas de saída de mensagens. As mensagens roteadas para as filas de saída, são as 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.

Segundo a especificação 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 disponibilidade de recursos. A implementação RabbitMQ utilizada neste trabalho deixa as limitações a cargo da quantidade de recursos do sistema hospedeiro.

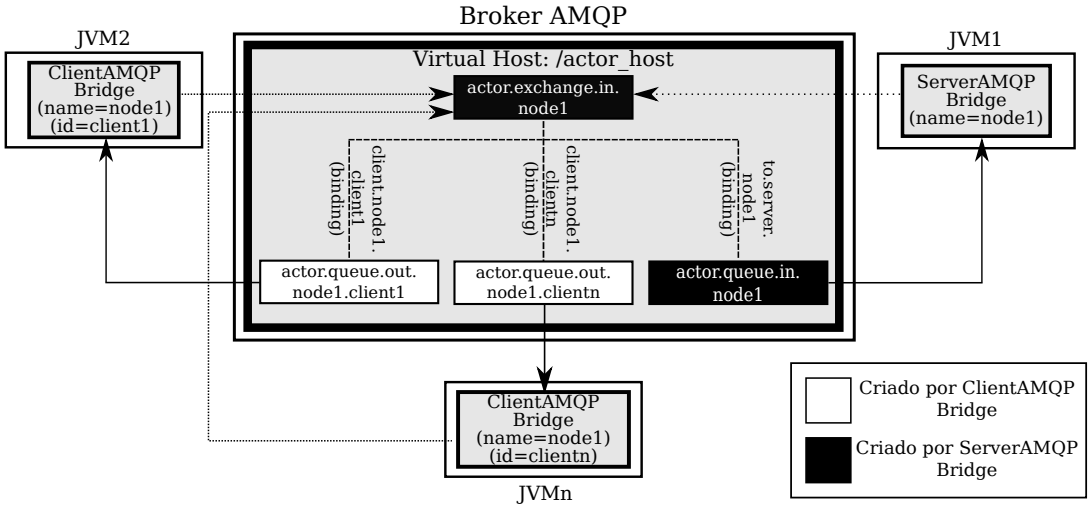

Figura 5.6: *Estrutura para troca de mensagens via message broker AMQP.*

5.2.4 Envio e recebimento de mensagens via pontes AMQP

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`).

Optamos por fazer envios mandatórios, porém sem consumo imediato. Isso significa que 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 na devolução de mensagens para o remetente. Decidimos por não forçar um consumo imediato para que não sobrecarregar as implementações dos consumidores, minimizando a possibilidade de rejeições de mensagens.

Ainda assim, caso uma mensagem seja rejeitada e devolvida, o método `handleRejected` do `MessageHandler` é invocado para que alguma ação seja tomada com a mensagem rejeitada. Mensagens podem ser rejeitadas por basicamente dois motivos: *(i)* uma mensagem

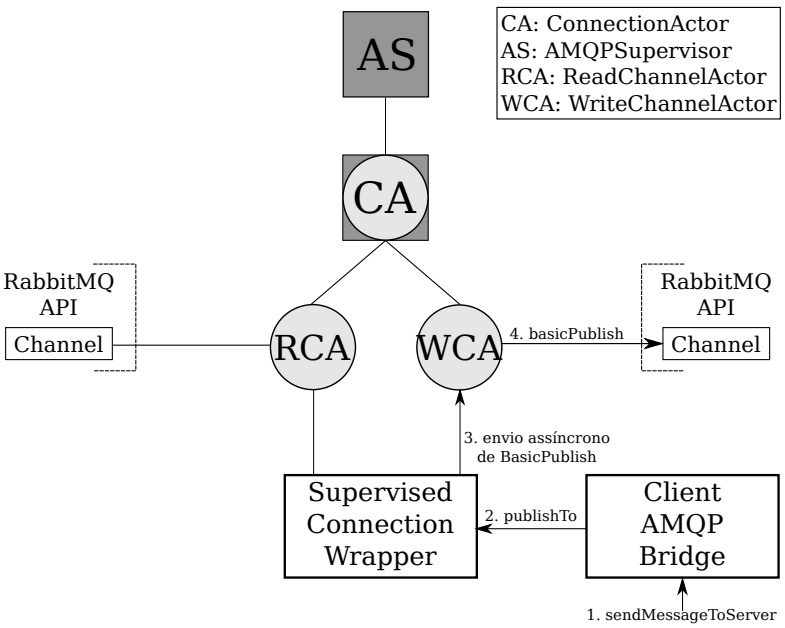

Figura 5.7: *Passos do envio de mensagens de um cliente via pontes AMQP.*

não pode ser roteada para uma fila pois a chave de roteamento utilizada no envio não está associada à *exchange*; *(ii)* uma mensagem foi depositada em alguma fila, porém a fila foi removida. Nesse caso, todas as mensagens são devolvidas 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* utilizado 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 *message 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.

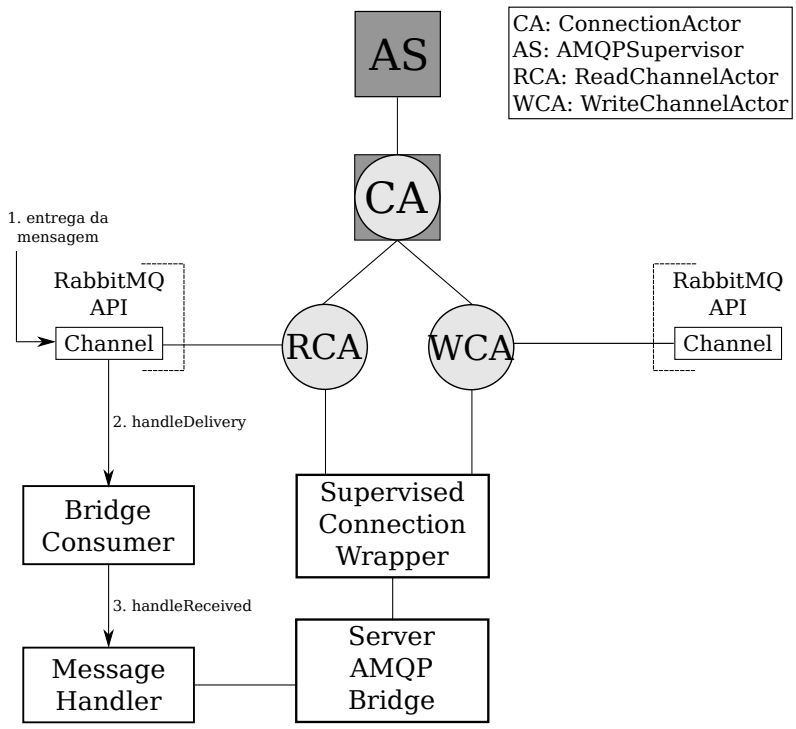

Figura 5.8: *Passos para recebimento de mensagens via pontes AMQP.*

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

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 *message broker* AMQP, tivemos que criar novos componentes para a camada de implementação com base nas classes e feições 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. Criamos os seguintes componentes:

- `AMQPRemoteServer`: É uma classe utilizada no lado do servidor onde estão os atores remotamente acessíveis. Essa classe implementa os métodos abstratos 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 feição que estende a feição `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 neste nó.

- `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 seu servidor remoto. 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 feição que estende a feição `RemoteClientModule`. Essa feição que tem um papel complementar a `AMQPRemoteServerModule`. Seu papel principal é definir uma interface para envios de mensagens a atores remotos via `AMQPRemoteClient`. Ademais, esta feição feição mantém ainda um registro dos `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 nas classes e feições acima listadas para prover o suporte remoto via *message broker* AMQP.

O modo como os novos componentes se relacionam com as pontes AMQP e com as classes do Akka são mostrados na figura 6.1.

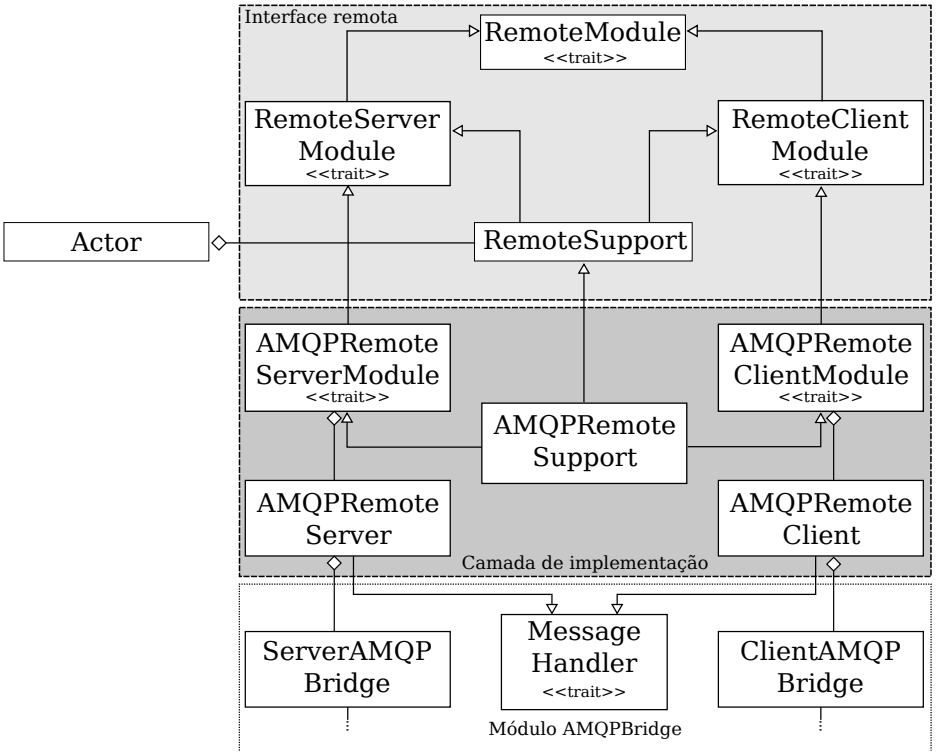

Figura 6.1: *Relacionamento entre os componentes para atores remotos com AMQP.*

A definição das feições para suporte a atores remotos do Akka, utiliza assinaturas de métodos baseadas em endereço de hospedeiro 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. 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 hospedeiro 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 desse atributo (`homeAddress`) não é utilizado. Como o projeto Akka possui desenvolvimento ativo, acreditamos que existe uma grande possibilidade de mudança na interface dos componentes do módulo de atores remotos, de modo que haja um desacoplamento entre sua interface e as novas 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`. Ademais, devemos informar as configurações que nosso suporte espera, como por exemplo, as políticas de armazenamento e compartilhamento de canais. As alterações necessárias para a integração do nosso novo suporte com a biblioteca de atores do Akka são apresentadas nas seções a seguir.

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 *message 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 impede a criação dos objetos no *message broker*. Para exemplificar, vamos considerar o caso em que dois clientes remotos seriam criados em uma JVM para servidores remotos cujos nomes são `node1` e `node2`. Os objetos que seriam criados pelos primeiro cliente remoto teriam o sufixo `client.node1.<id>` e `client.node2.<id>`. Um outro detalhe importante é que, pelo fato de classe `AMQPRemoteClientModule` manter um registro com os 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: Este 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 à devolução de eventuais mensagens que ainda estavam na fila.

2. Cenário oposto ao anterior, em que o servidor remoto possui uma configuração transiente e algum de seus clientes possui 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 analisado 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), 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 devolvidas.

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 prevenir 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 com *cookie* de segurança. O *cookie* de segurança 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 do *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, mantivemos o

suporte na nossa implementação. Mensagens oriundas de clientes que não se autenticarem nos servidores remotos, não são processadas e o remetente (cliente remoto) recebe uma mensagem de resposta indicando a necessidade a autenticação.

Com o uso das filas definidas no *message broker* como repositório intermediário das mensagens, qualquer entidade que possua os dados necessários para acessar o *virtual host* onde estão criadas as filas pode registrar outros consumidores (salvo no caso de filas com acesso exclusivo). 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 *message 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. O protocolo utilizado para definir uma mensagem remota passa a incluir um parâmetro não mandatório para que o identificador do cliente possa ser reconhecido. 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 implementação de transporte feita com o Netty consegue identificar na *socket* qual o endereço do a entidade remota que está se conectada.

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. 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, já que seu valor passou a ser irrelevante.

6.3 Fluxo de envio das mensagens

Parte do fluxo do envio de uma mensagem a um ator remoto com a nova camada de transporte continua acontecendo em sete passos, como descrito na 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 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 *message broker*, então, roteia a mensagem para a fila criada pela ponte servidora. Ainda que o *message broker* tente 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 ao consumo da mensagem enviada e no 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.

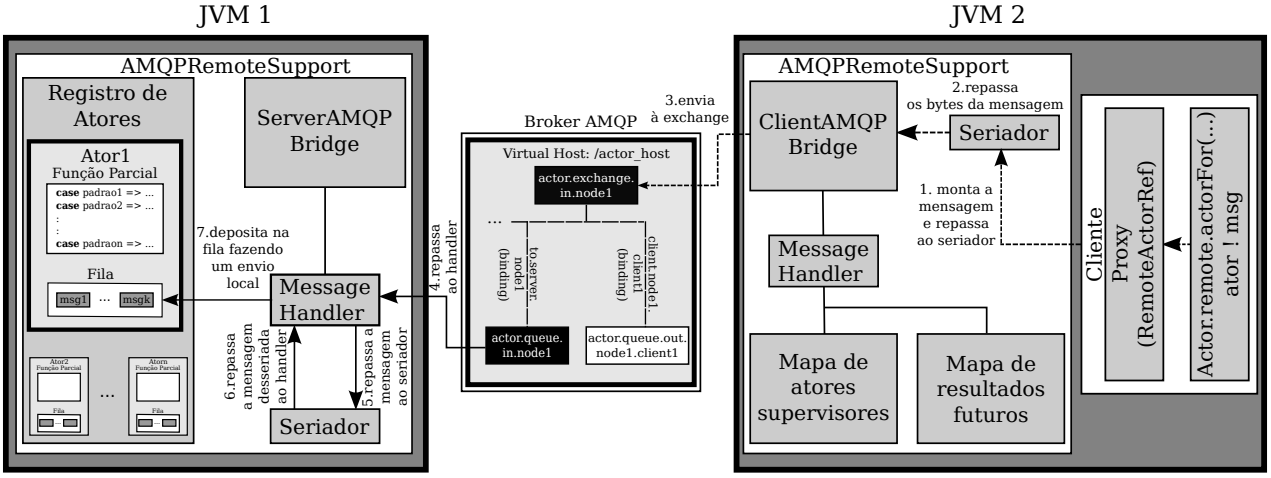

Figura 6.2: *Fluxo de um envio assíncrono de mensagem entre atores com AMQP.*

Assim como na implementação com Netty, os envios feitos via métodos `!!` e `!!!`, também 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 devolvida como resultado da invocação do método, é colocada no mapa de resultados futuros.

As mensagem de resposta para envios feitos via `!!` e `!!!`, ou para notificar atores supervisores sobre atores supervisionados fazem 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 *message broker* pode fazer 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.

Capítulo 7

Resultados

7.1 Trading System

Com o intuito de medir como novas alterações no código da biblioteca de atores melhoravam o desempenho, os desenvolvedores do projeto Akka criaram um sistema que simula a compra e venda de ações chamado de *Trading System*. Os desenvolvedores criaram também algumas classes que cuidam do armazenamento, comparação e geração de relatórios dos dados coletados nos experimentos de comparação de desempenho. A disposição dos componentes do sistema de cotações é mostrados na figura 7.1. Os componentes são descritos a seguir:

- *Order Book*: É uma entidade que possui informações sobre as ordens de compra e venda de uma determinada ação. Neste sistema, uma ação é composta por uma letra e por um número. Tem como responsabilidade fazer o casamento de uma ordem de venda com uma ordem de compra. São utilizados como nomes de ações as letras A, B e C, combinadas com números de 1 a 5, num total de quinze *order books*.
- *Matching Engine*: É um ator que age como intermediário no encaminhamento de mensagens aos *order books*. Existe um *matching engine* para cada letra de *order book*.
- *Order Receiver*: É um ator externo que tem o papel de receber mensagens de compra e venda de qualquer ação, encontrar o *matching engine* responsável e repassar a mensagem. A quantidade de *order receivers* não possui relação com *matching engines*, já que um *order receiver* recebe mensagens para qualquer tipo de ação. O sistema possui dez *order receivers*.
- *Trading System*: É o ator recepcionista do sistema. É responsável por fazer a criação e inicialização dos atores internos do sistema. Este ator ainda responde aos atores clientes com os endereços (referências) dos *order receivers* que foram criados.
- *Order*: É um componente que não é mostrado na figura, mas que representa uma ordem. É definida em uma classe abstrata e possui como informações o código da ação, o volume e o valor. Suas implementações são *Ask* para indicar uma ordem de venda, e *Bid* para indicar uma ordem de compra.

Existem ainda atores rotulados como "clientes". Um ator cliente é um ator externo ao sistema de cotações que possui uma lista de ordens a serem enviadas, um valor inteiro indicando a quantidade de repetições a serem executadas na lista e um *order receiver* para onde as mensagens devem ser enviadas. A quantidade de atores clientes varia de acordo com

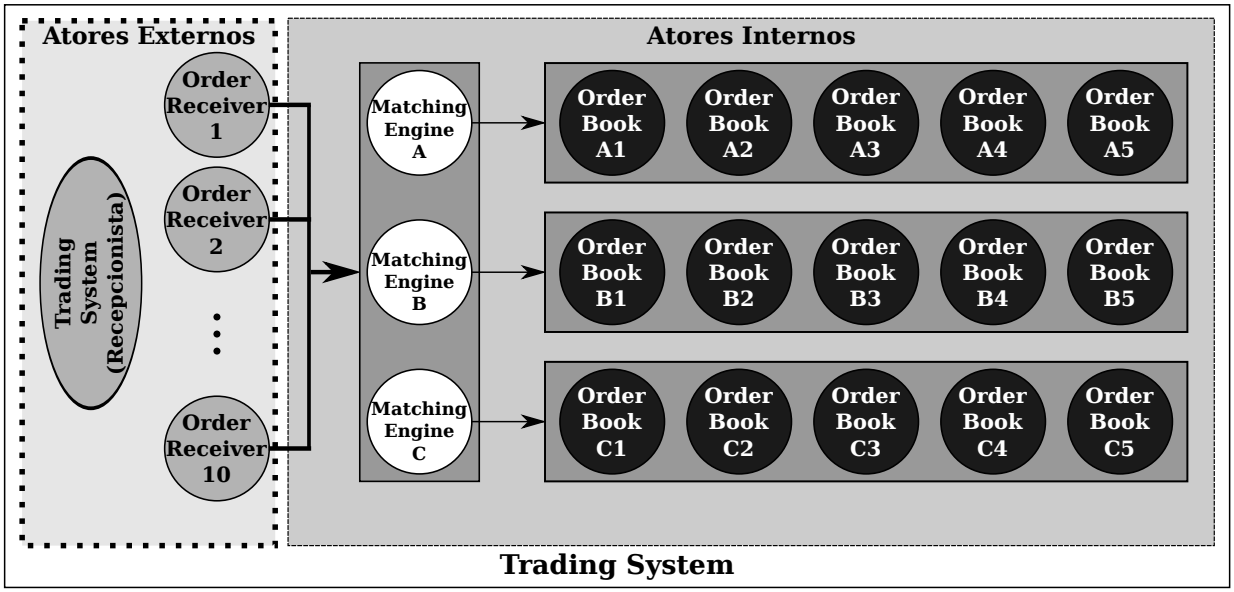

Figura 7.1: *Trading System – Sistema de compra e vendas de ações.*

a instância do experimento que se deseja fazer. A quantidade não será necessariamente igual à quantidade de *order receivers*. Por este motivo, a associação de um ator cliente a um *order receiver* é feita em sequência. Itera-se na lista de *order receivers* junto com a lista de atores clientes. Uma vez que se tenha alcançado o décimo *order receiver*, reinicia-se a iteração a partir do primeiro ator. Dependendo a quantidade de atores clientes, alguns *order receivers* podem recebem mensagens de mais, de menos ou até mesmo de nenhum ator cliente quando comparados com outros *order receivers*.

Cada ator cliente envia um total de trinta ordens (quinze ordens de compra e quinze ordens de venda) para o *order receiver* que lhe foi associado. Esse valor é multiplicado pela quantidade de repetições que o ator cliente deve executar. A medição é feita com base no tempo que uma mensagem enviada por um ator cliente levou para ser processada em um *order book*.

7.2 Trading System com atores remotos

Ainda que o *Trading System* não tenha sido criado para experimentos com atores remotos, consideramos ser de grande utilidade utilizá-lo para fazer uma comparação de desempenho entre o suporte padrão a atores remotos do Akka com o que desenvolvemos neste trabalho. Para tal, foram necessárias as alterações listadas a seguir:

1. Registramos todos os atores que estão posicionados na área rotulada como "Atores Externos" na figura 7.1, como atores remotos (seção 3.2). Essa modificação resultou em alterações na inicialização do *Trading System* e na obtenção dos *order receivers*. Na versão remota, apenas os nomes sob o qual os *order receivers* foram registrados no registro de atores remotos é devolvido. Na versão local, são devolvidas `ActorRefs`. Evitamos devolver `ActorRefs` como parte das mensagens pois, neste caso, os atores deveriam passar por uma seriação funda. Enviando os nomes sob os quais os atores foram registrados, permite que façamos buscas no registro de atores remotos, trabalhando apenas com as referências remotas.

2. Fizemos com que as classes *Ask* e *Bid* fossem seriáveis (seção 3.2.3). Optamos por

utilizar como formato de seriação, o formato definido na feição `ScalaJSON`.

3. Trocamos os envios assíncronos de mensagens dos atores clientes por envios síncronos. Assim, uma nova mensagem só é enviada uma vez que a mensagem anterior tenha sido processada e sua confirmação recebida no ator cliente. O comportamento de um ator cliente teve uma pequena alteração: para cada mensagem enviada, o ator cliente marca o momento de início do envio da mensagem, aguarda o valor devolvido pelo método `!!` (que é invocado no *order receiver*), marca o tempo de término do envio e salva a diferença de tempo em um repositório.

As configurações utilizadas no arquivo `akka.conf` para os experimentos com o suporte padrão a atores remotos do Akka são as recomendadas uso geral (compressão média de mensagens e tamanho do janelamento das mensagens de 1KB). Nos experimentos executados com o suporte desenvolvido neste trabalho, utilizamos uma conexão por canal além da configuração `DURABLE` para as pontes AMQP (seção 5.2.1).

7.3 Comparação de desempenho

Para que pudéssemos ter medidas mais reais, separamos os atores clientes e os atores do *Trading System* em JVMs residentes em nós fisicamente separados. A máquina virtual Erlang, onde o RabbitMQ esteve em execução, ficou separada em um terceiro nó. A máquina virtual Erlang, responsável por executar o RabbitMQ, ficou em execução em um *laptop* Dell Inspiron N4030 com processador Intel Core i3 de 2.4 GHz e 4 GB de RAM com Linux Ubuntu 10.10. A JVM onde implantamos os atores clientes, ficou em execução em um *laptop* Lenovo T410 com processador Intel Core i5 de 2.4 GHz e 4 GB de RAM com Linux Ubuntu 10.10. Por fim, a JVM onde o *Trading System* foi implantado, ficou em execução em um *laptop* MacBook Pro com processador Core 2 Duo de 2.4 GHz e 4 GB de RAM com Mac OS X 10.7. Os três *laptops* foram conectados por um roteador D-Link DI-524. Apesar de este ser um roteador para redes *wireless*, optamos por conectá-los com cabos de rede.

Definimos o número de mensagens que cada cliente enviou da seguinte maneira: seja n_c a quantidade de atores clientes utilizada em um experimento e seja r o número de repetições; cada ator cliente enviou $(30r)/n_c$ mensagens. O valor r de repetições utilizado foi 1000. Variamos os valores de n_c de modo que o resultado da divisão fosse sempre inteiro. O total de mensagens enviadas por todos os atores clientes em uma instância de experimento foi sempre 30000.

A estrutura criada para medição e comparação de desempenho de atores locais do Akka utiliza algumas bibliotecas para o armazenamento e geração de relatórios. Os valores das medidas registradas em uma instância de experimento são organizados em percentis. A biblioteca utilizada para o cálculo de percentis é a Commons Math da fundação Apache [CM1]. Os relatórios gerados apresentam também a quantidade de envios/respostas por segundo (ER/S), o tempo médio de cada envio/resposta e ainda o tempo total gasto para a execução de todos os envios/respostas.

A tabela 7.1 mostra as melhores medidas dentre todas as execuções feitas na execução do *Trading System* com atores remotos. O critério utilizado para a comparação entre as medidas de um mesmo conjunto de instâncias foi o valor que é apresentado na coluna média.

O fato de adicionarmos um terceiro nó com um *message broker* na comunicação entre os atores remotos, naturalmente adiciona mais passagens pela rede que conecta os nós. Um envio que utiliza o Netty faz com que duas mensagens trafeguem pela rede: a ordem é enviada pelo

Impl.	Clientes	ER/S	Média (μs)	25% (μs)	50% (μs)	75% (μs)	95% (μs)	Dur. (s)
Netty	1	501	1995	1364	1433	1968	3266	59.85
AMQP	1	263	3797	3283	3361	3537	4550	113.91
	Delta	-47.50%	90.33%	140.69%	134.54%	79.73%	39.31%	54.06 (90.33%)
Netty	2	894	2238	1770	1796	1836	3290	33.57
AMQP	2	528	3789	3234	3359	3552	4208	56.83
	Delta	-40.99%	69.30%	82.71%	87.03%	93.46%	27.90%	23.26 (69.33%)
Netty	4	1245	3212	2070	2517	2726	4214	24.09
AMQP	4	906	4416	3248	3614	4134	5088	33.12
	Delta	-27.31%	37.48%	56.91%	43.58%	51.65%	20.74%	9.03 (37.48%)
Netty	8	1494	5353	3274	3771	4745	6168	20.07
AMQP	8	1389	5760	3567	4149	5168	6425	21.60
	Delta	-7.10%	7.60%	8.95%	10.02%	8.91%	4.17%	1.52 (7.60%)
Netty	10	1542	6484	4005	4330	5704	7041	19.45
AMQP	10	1424	7023	4396	5107	6136	7610	21.06
	Delta	-7.72%	8.31%	9.76%	17.94%	7.57%	8.08%	1.61 (8.31%)
Netty	20	1568	12757	8206	9646	10146	12015	19.13
AMQP	20	1503	13309	8728	9964	10888	13430	19.96
	Delta	-4.15%	4.33%	6.36%	3.30%	7.31%	11.78%	0.82 (4.33%)
Netty	40	1589	25176	17616	18183	19436	21350	18.82
AMQP	40	1518	26345	18496	19525	20842	23360	19.75
	Delta	-4.41%	4.64%	5.00%	7.38%	7.23%	9.41%	0.87 (4.64%)
Netty	80	1615	49525	35403	36541	37863	40041	18.57
AMQP	80	1535	52122	37915	39399	40943	43496	19.54
	Delta	-5.02%	5.24%	7.10%	7.82%	8.13%	8.63%	0.97 (5.24%)

Tabela 7.1: *Medidas de tempo do Trading System com atores remotos.*

`RemoteClient` direto ao `RemoteServer` e a resposta faz o caminho inverso. Já quando utilizamos nossa implementação com AMQP, quatro mensagens são trafegadas: a ordem é enviada pelo `RemoteClient` ao *message broker*, o *message broker* envia a ordem para o `RemoteServer` e a resposta faz o caminho inverso. Por este motivo, esperávamos que o tempo gasto pela implementação com AMQP fosse, aproximadamente, duas vezes maior do que o tempo gasto pela implementação com Netty. Observando a comparação dos dados da tabela 7.1 para a instância com apenas 1 ator cliente, podemos notar uma diferença bem próxima da esperada, já que o número de envios/respostas por segundo é quase metade (-47.50%), o valor médio para cada envio/resposta leva quase o dobro do tempo (90.33%), fazendo com que a duração total também leve quase o dobro do tempo (90.33%). Quando observamos os percentis, podemos notar que a diferença de tempo das execuções com AMQP que estão abaixo da mediana, foram maior que duas vezes mais lentas em relação às execuções feitas com Netty (140.69% e 134.54%). Entretanto, quando comparamos os percentis acima da mediana, observamos uma boa redução na diferença (79.73% e 39.31%).

Com o aumento do número de atores clientes, a diferença de tempo apresenta uma boa diminuição. O gráfico da figura 7.2 mostra a evolução da quantidade de envios/respostas com o aumento da quantidade de atores clientes. Podemos observar que, conforme a quantidade de atores clientes foi aumentando, os resultados com nossa implementação apresentaram uma boa melhora e se mantiveram muito próximos dos resultados obtidos com a implementação original do Akka. A diferença que era de quase -47.50% envios/respostas cai até chegar em apenas -5.02%, o que no nosso experimento representa pouco menos de 1 segundo para completar todos os envios/respostas. De um modo geral, se observarmos a diferença do tempo de execução para os experimentos a partir de 8 clientes, temos uma diferença de tempo abaixo de 1.6 segundos.

Devemos relembrar que não adicionamos compressão de mensagens em nossa implementação com AMQP, e que os experimentos feitos com Netty utilizaram compressão média. Como as mensagens são menores que 1KB, entendemos que a implementação com AMQP possa ter levado ligeira vantagem. Ademais, o terceiro nó colabora para a distribuição do processamento. Na implementação feita com Netty, tanto as classes responsáveis pelo envio e recebimento de mensagens quanto as classes do arcabouço compartilham mesma JVM que

executa os atores do sistema de cotações. Já na implementação com AMQP, apenas as classes responsáveis pelo envio e recebimento de mensagens compartilham a mesma JVM. Durante a execução dos experimentos, mesmo diante de um cenário com alta concorrência em que vários atores clientes executaram diversos envios em paralelo para um `RemoteClient` compartilhando, não tivemos problemas comuns de concorrência como *deadlocks* e condições de corrida em nossa implementação.

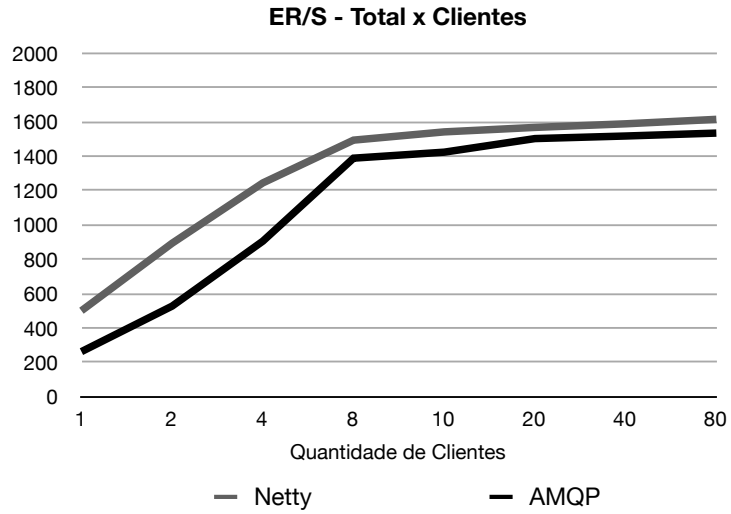

Figura 7.2: *Comparação da quantidade de envios/respostas.*

Pelo fato das mensagens serem trafegadas duas vezes a mais na rede, naturalmente, a latência da rede é um dos principais fatores que impactam os resultados. A tabela 7.2 apresenta as medidas que feitas na rede *wireless*, com maior latência. Utilizamos a mesma configuração descrita anteriormente, porém retiramos os cabos que ligavam os computadores e passamos a conectá-los via *wireless*. Os d ados da tabela 7.2 mostram que a diferença de tempo chega no pior caso, com 4 atores clientes, a pouco mais de 2.5 vezes o tempo gasto pela implementação com Netty. Quando observamos os percentis, podemos notar que a diferença no tempo das execuções com AMQP que estão abaixo da mediana foram, aproximadamente, 3 vezes mais lentas em relação às execuções feitas com Netty (209.28% e 197.17%). Quando comparamos os percentis acima da mediana, observamos uma redução razoável na diferença (179.11% e 145.07%). Assim como na tabela 7.1, conforme aumentamos a quantidade de atores clientes, a diferença de tempo cai chegando a 26.74%, pouco menos que 9 segundos para completar todos os envios/respostas.

O gráfico da figura 7.3 mostra que, com o aumento de atores clientes, o aumento da quantidade de envios/respostas feitos com nossa implementação não apresentou a mesma melhora de desempenho que a implementação com o Netty. Uma melhora mais significativa passa a acontecer quando a quantidade de clientes passa de 10. Se observamos novamente o gráfico da figura 7.2, notamos que o aumento no número de envios/respostas foi mais acentuado no início (de 1 a 8 atores clientes) do que no final. Com as mensagens levando mais tempo para chegar ao *message broker* e ao seu destino final, o volume de mensagens em um dado instante tende a ser menor. Com um volume menor de mensagens, a vantagem que levamos com a distribuição do processamento dos envios e recebimentos de mensagens só começa a contar positivamente com o aumento do volume de mensagens concorrentes.

Sobre as medidas, ainda que exista um processamento extra a cargo do *message broker* para recebimento, roteamento e entrega das mensagens, concluímos que existe uma vantagem

Impl.	Clientes	ER/S	Média (μs)	25% (μs)	50% (μs)	75% (μs)	95% (μs)	Dur. (s)
Netty	1	163	6129	3387	4279	6265	13403	183.87
AMQP	1	106	9464	5998	7223	9624	17360	283.92
	Delta	-34.97%	54.41%	77.09%	68.80%	53.62%	29.52%	100.05 (54.41%)
Netty	2	341	5857	3418	4113	5635	11481	87.855
AMQP	2	152	13121	8963	10784	14077	22309	196.815
	Delta	-55.43%	124.02%	162.23%	162.19%	149.81%	94.31%	108.96 (124.02%)
Netty	4	479	8337	5248	6334	8219	13892	62.5275
AMQP	4	183	21872	16231	18823	22940	34045	164.04
	Delta	-61.80%	162.35%	209.28%	197.17%	179.11%	145.07%	101.51 (162.35%)
Netty	8	491	16292	10528	12692	15780	23123	61.095
AMQP	8	198	40463	26918	33224	41392	58513	151.73625
	Delta	-59.67%	148.36%	155.68%	161.77%	162.31%	153.05%	90.64 (148.36%)
Netty	10	559	17883	12737	14918	18099	25705	53.649
AMQP	10	243	40891	29653	34970	41936	56755	122.673
	Delta	-56.27%	128.66%	132.81%	134.41%	131.70%	120.79%	69.02 (128.66%)
Netty	20	563	35486	22392	26879	32957	47837	53.23
AMQP	20	323	61951	47710	55291	64509	84108	92.92
	Delta	-42.63%	74.58%	113.07%	105.70%	95.74%	75.82%	39.69 (74.58%)
Netty	40	607	65782	41155	49748	60805	83042	49.33
AMQP	40	390	98951	62679	77785	98463	150803	74.21
	Delta	-35.75%	50.42%	52.30%	56.36%	61.93%	81.60%	24.87 (50.42%)
Netty	80	889	89089	36654	39031	50225	103695	33.41
AMQP	80	628	112913	66128	77563	94990	139215	42.34
	Delta	-29.36%	26.74%	80.41%	98.72%	89.13%	34.25%	8.93 (26.74%)

Tabela 7.2: *Medidas de tempo do Trading System com atores remotos (wireless).*

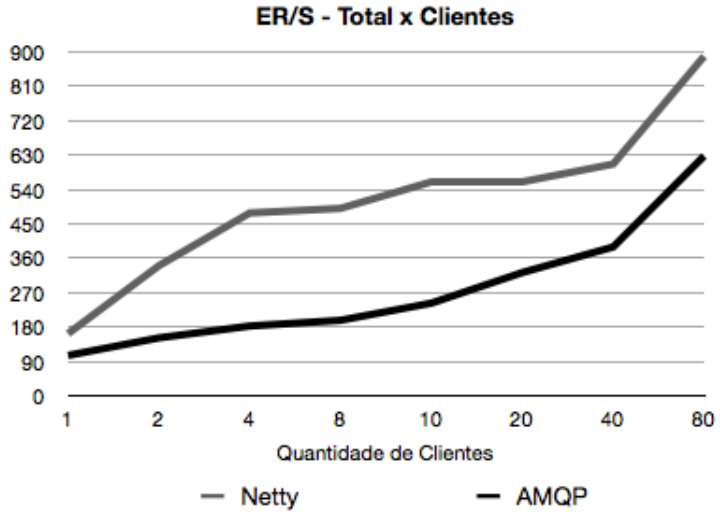

Figura 7.3: *Comparação da quantidade de envios/respostas na rede wireless.*

relativa em relação a implementação feita com o Netty. Tal vantagem deve-se ao fato do Netty também possuir tarefas semelhantes, como recebimento e entrega das mensagens, e de este processamento acontecer nos mesmos nós onde estão em execução as classes que fazem o processamento das mensagens, como os atores. Conforme a frequência das mensagens foi aumentando, houve uma redução considerável na diferença de tempo, como mostrado no gráfico da figura 7.2. Assim, entendemos que a carga extra de processamento adicionada pelo *message broker*, não invalida o seu uso como suporte para a troca de mensagens entre atores remotos. Enfatizamos que em um cenário realista de um sistema de atores, é esperado um número razoável de atores executando em paralelo.

Capítulo 8

Conclusões

Comentamos brevemente no capítulo 5 sobre algumas das características da trocas de mensagens entre entidades conectadas ponto-a-ponto via *sockets*, como saber o endereço do hospedeiro e porta da outra parte no qual se está conectado, a possibilidade de identificar se a outra parte continua conectada e o não armazenamento intermediário de mensagens. A biblioteca do Netty fornece meios para obter informações tanto sobre o endereço da outra parte, como ser notificado em caso de falhas na conexão. Esses meios são utilizados na implementação de atores remotos do Akka, tanto para notificar um `RemoteClient` que o `RemoteServer` correspondente não está mais conectado, quanto o contrário.

O fato de utilizarmos filas para fazer o armazenamento intermediário das mensagens (ainda que, na maior parte do tempo, por um período breve), permite que um `RemoteServer` ou `RemoteClient` possam receber mensagens mesmo após sua execução ter sido interrompida. Para tal, basta que eles tenham executado alguma vez com uma configuração durável[1]. As mensagens ficam armazenadas até que um novo consumidor seja registrado. Um detalhe importante a se destacar é em relação a *exchange* não existir no momento do envio. Em casos como esse, quem está fazendo o envio recebe uma exceção no mesmo momento que está tentando fazer o envio. Mensagens que foram armazenadas, por exemplo, porque o `RemoteServer` ou `RemoteClient` não estavam em execução podem deixar de ter a mesma semântica para o ator. O comportamento do ator no momento do recebimento da mensagem pode, eventualmente, ser diferente do comportamento que o ator possuía no momento do envio.

Assim como o modelo de atores não prevê a ordem de entrega das mensagens, deixando a cargo das aplicações que utilizam o modelo fazer o tratamento, o recebimento de mensagens que ficaram armazenadas é uma situação que leva a um cenário semelhante, também deixando a cargo da aplicação a responsabilidade sobre o processamento de tais mensagens. Mensagens enviadas de forma síncrona para atores remotos que não estão acessíveis, causam estouro no tempo limite de espera no resultado futuro. Uma eventual resposta do ator remoto após o tempo limite, não causa erros. Do modo como está a implementação, o mapa de resultados futuros (seção 3.1.2) mantém as referências para os resultados até que eles sejam completados.

O uso do *message broker* como ponto central de comunicação de todos os atores remotos, cria um ponto crítico para o caso de falhas. Caso o nó onde o *message broker* está em execução sofra alguma avaria, a comunicação de todos os atores remotos é afetada. O RabbitMQ pode ser configurado em *cluster* [RCL], no qual os nós compartilham as informações dos *virtual hosts*. O uso do RabbitMQ em *cluster* minimiza o problema de o *message broker* ser um

[1] O mesmo vale para configurações transientes, desde que o *message broker* não tenha sido reiniciado

ponto central de falhas.

8.1 Trabalhos Relacionados

8.1.1 Replicação de transações no Drizzle com RabbitMQ

Drizzle [Dri] é uma implementação de banco de dados voltado para computação em nuvem (*cloud computing*). A implementação desse banco de dados possui código aberto e foi desenvolvida tomando como base o código da implementação do banco de dados MySQL [MyS].

Uma das características do Drizzle é a replicação de transações. Transações como, inserções ou atualizações, que são executadas na instância principal (*master*), são replicadas para uma ou mais instâncias secundárias (*slaves*). Esse módulo utiliza uma combinação de tabelas transacionais (InnoDB) e mensagens no formato Protobuf **??**. O uso das tabelas transacionais tem como característica a criação de um arquivo para armazenamento das transações executadas.

O Drizzle oferece a possibilidade de replicação de transações via *message broker* AMQP. A implementação está disponível via módulo RabbitMQ Integration [DMQ]. Quando esse módulo é utilizado, deve-se informar as informações necessárias para conexão com o RabbitMQ, tais como endereço, porta, *virtual host*, usuário e senha. Deve-se informar também, o nome da *exchange* para onde as mensagens devem ser enviadas e a chave de roteamento. Quando o módulo de replicação RabbitMQ Integration está ativado, uma aplicação Java associada a instância principal coleta as informações do arquivo onde as transações foram armazenadas e faz o envio das mensagens com as transações. Aplicações Java associadas às instâncias secundárias são responsáveis por fazer o consumo das mensagens vindas do RabbitMQ, a conversão da mensagem e tomar alguma ação, como executar as operações da transação na instância do Drizzle no qual está associada. Vale destacar que as aplicações Java associadas às instâncias secundárias do Drizzle, são responsáveis por fazer a criação da fila onde esperam mensagens e definir seu *binding* com a *exchange* informada durante a ativação do módulo RabbitMQ Integration. As informações para a criação das filas, como nome, e da instância do Drizzle para onde as transações serão replicadas, são informadas em um arquivo de propriedades.

8.2 Considerações Finais

O uso do RabbitMQ como *message broker* AMQP em conjunto com a estrutura descrita no capítulo 5, se mostrou eficaz em nossos experimentos para transportar mensagens entre atores remotos. O tempo extra de processamento adicionado com uso do *message broker* se mostrou muito baixo para um cenário mais realista, em que exista uma quantidade razoável de atores executando em paralelo e baixa latência de rede, está bem abaixo do que esperávamos. Dependendo do tipo de aplicação, podemos afirmar que o tempo extra pode ser até considerado como desprezível. Nossa conclusão é de que as duas classes de sistemas estudas neste trabalho possuem boa sinergia.

O uso de um *message broker* ainda trás outros benefícios, como por exemplo:

- Ferramentas administrativas: A maioria dos *message brokers* possui ferramentas para administração e monitoramento. O uso deste tipo de ferramenta ajuda no monitoramento de uma parte importante da infraestrutura de um sistema distribuído. O RabbitMQ, por exemplo, possui um módulo para gerenciamento que é acessível de um

navegador *web* [RMG]. Com este módulo é possível criar, alterar e remover entidades, além de observar informações como, por exemplo, o volume de mensagens em determinada entidade, a quantidade de clientes conectados e a quantidade de mensagens que está em uma fila aguardando confirmação de recebimento.

- Alta disponibilidade: O RabbitMQ dá suporte, além das configurações de *cluster*, à criação de filas de alta disponibilidade [RHA]. A utilização do RabbitMQ em *cluster* dá suporte a *exchanges* e suas associações no caso de avarias no nó onde elas residem, porém as filas e suas mensagens não. O uso de filas de alta disponibilidade é uma solução complementar, baseada no espelhamento das filas, para o RabbitMQ prover alta disponibilidade dos seus serviços.

- Simplificação de acesso: Com o uso de nomes ao invés de IPs e número de portas para identificar os nós onde residem os atores remotos, conseguimos certa independência dos endereços IPs. O único endereço que deve ser conhecido é o do nó onde o *message broker* está em execução. Ainda que em nossa implementação tenhamos mantido as assinaturas dos métodos definidos no projeto Akka, nossa implementação não fica atrelada a esse detalhe. Os endereços IPs dos nós não possuem relação com os nomes utilizados para registrar ou localizar um ator remoto. Existe ainda uma outra vantagem que é a redução do número de portas que devem ser desbloqueadas em um *firewall*. Não é necessário desbloquear as portas que cada `RemoteServer` estará disponível. A única porta que precisa estar desbloqueada para aceitar conexões, é a porta que o *message broker* disponibilizou para os clientes e somente no nó onde ele está em execução.

8.3 Sugestões para Pesquisas Futuras

8.3.1 Melhoria no tratamento de erros nas pontes AMQP

O fato de utilizarmos atores para encapsular as principais classes da biblioteca para clientes Java do RabbitMQ, permite que a hierarquia de atores que foi criada venha ser utilizada para o tratamento de erros. Ainda que nossa implementação seja de uma hierarquia de supervisão, o atores supervisores não possuem comportamento para o tratamento de erros em atores filhos.

A implementação pode ser ampliada para que, em caso de erros nos atores filhos, como os responsáveis pelos canais ou conexões, os atores supervisores possam criar novos atores supervisionados no mesmo estado que estavam antes do erro acontecer. A restauração do estado nos atores filhos deve obedecer as configurações originais dos objetos que foram criados no *message broker*. É necessário ainda que, quando não for possível restabelecer o sistema, seja por não se conseguir reconectar ao *message broker* ou por algum outro motivo, notificar a ponte AMQP para que ela possa propagar o erro para o nível da aplicação.

8.3.2 Experimentos em um ambiente de computação em nuvem

Nos últimos anos, o conceito de computação em nuvem vem ganhando bastante espaço, não só na comunidade acadêmica, mas também na indústria de tecnologia. Computação em nuvem é um conceito que se refere tanto a aplicações disponibilizadas como serviços via internet quanto pelo *hardware* e sistemas de *software* que estão nos centros processamento de dados para prover estes serviços [AFG+10]. O termo "nuvem" é utilizado para denotar o *hardware* e o *software* que ficam no centro de processamento de dados. Um dos princi-

pais atrativos da computação em nuvem é a capacidade de elasticidade da nuvem. Caso a demanda por um serviço aumenta, recursos que estão disponíveis, como computadores e máquinas virtuais podem ser provisionados para aumentar a capacidade de processamento. Quando a demanda pelo serviço cai e a capacidade de processamento necessária passa a ser menor, parte recursos podem ser liberados para atenderem outros serviços.

Algumas plataformas de computação em nuvem existentes no mercado, como EC2 (Amazon) [AEC], Nebula (NASA) [NEB], Heroku [HER] e Cloud Foundry (VMWare) [CFY], ou possuem o RabbitMQ como *message broker*, ou permitem que o RabbitMQ seja instalado. Acreditamos que nossa implementação de atores remotos esteja muito próxima e necessite de poucas adaptações para poder ser implantada em uma infraestrutura de computação em nuvem. Acreditamos que a elasticidade da nuvem, alinhada com a capacidade do RabbitMQ de trabalhar com um número alto de requisições, permita uma alta escalabilidade no sistema que for desenvolvido. Deixamos como sugestão de experimento, fazer uma comparação do *Trading System* com a quantidade de clientes bem maior da utilizada em nossos experimentos.

Apêndice A

Exemplo de uma aplicação 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: *Feição 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

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

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.*

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 na documentação da biblioteca para clientes Java [API].

```
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

[AEC] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2. Último acesso em 6/12/2011. 68

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica e Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, Abril 2010. 67

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986. 3, 5, 6, 8

[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 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, 71

[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, 34

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007. 8, 11, 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. 27

[CFY] VMWare – Cloud Foundry. http://www.cloudfoundry.com. Último acesso em 6/12/2011. 68

[CM1] Math - Commons Math: The Apache Commons Mathematics Library. http://commons.apache.org/math/. Último acesso em 9/9/2011. 61

[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. 13

[DMQ] Dizzle – RabbitMQ Integration. http://docs.drizzle.org/plugins/rabbitmq/index.html. Último acesso em 10 de Janeiro de 2012. 66

[Dri] Drizzle – a database for the cloud. http://www.drizzle.org/. Último acesso em 10 de Janeiro de 2012. 66

[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. 13, 26

[Gri75] Irene Grief. *Semantics of communicating parallel process.* Tese de Doutorado, Department of Electrical Engineering and Computer Science, MIT, EUA, Aug 1975. 8

[Gro] AMQP Working Group. Advanced message queuing protocol. http://www.amqp.org. Último acesso em 5/5/2011. 2, 29, 46

[HA79] Carl Hewitt e Russell R. Atkinson. Specification and proof techniques for serializers. *IEEE Trans. Software Eng.*, 5(1):10–23, 1979. 8

[HAL79] Carl Hewitt, Giuseppe Attardi e Henry Lieberman. Specifying and proving properties of guardians for distributed systems. Em *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, páginas 316–336, London, UK, 1979. Springer-Verlag. 8

[Har] Mark Harrah. Library for describing binary formats for Scala types. https://github.com/harrah/sbinary. Último acesso em 27/5/2011. 13

[HB77] Carl Hewitt e Henry G. Baker. Laws for communicating parallel processes. Em *IFIP Congress*, páginas 987–992, 1977. 8

[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

[HER] Heroku Cloud Application Platform. http://www.heroku.com. Último acesso em 6/12/2011. 68

[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

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

[JNY] Netty – the Java NIO Client Server Socket Framework. http://www.jboss.org/netty. Último acesso em 27/5/2011. 13, 22

[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. 12

[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

[Man87] Carl Roger Manning. Acore – the design of a core actor language and its compiler. Dissertação de Mestrado, Dept. of Electrical Engineering and Computer Science, MIT, EUA, Maio 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

[MyS] MySQL – the world"s most popular open source database. http://www.mysql.com/. Último acesso em 10 de Janeiro de 2012. 66

[NEB] NASA – Nebula Cloud Computing Platform. http://nebula.nasa.gov. Último acesso em 6/12/2011. 68

[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

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

[RCL] RabbitMQ - Clustering Guide. http://www.rabbitmq.com/clustering.html. Último acesso em 28/11/2011. 65

[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

[RHA] RabbitMQ - High Available Queues. http://www.rabbitmq.com/ha.html. Último acesso em 30/11/2011. 67

[RMG] RabbitMQ - Management Plugin. http://www.rabbitmq.com/management.html. Último acesso em 30/11/2011. 67

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

[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

[TKS+88] C. Tomlinson, W. Kim, M. Scheevel, V. Singh, B. Will e G. Agha. Rosette: An object-oriented concurrent systems architecture. *SIGPLAN Not.*, 24:91–93, 1988. 8

[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

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

Índice Remissivo