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

Thadeu de Russo e Carmo

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

Programa: Ciência da Computação

Orientador: Prof. Dr. Francisco Carlos da Rocha Reverbel

São Paulo, novembro de 2011

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

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

Agradecimentos

Agradecimentos ...

Resumo

Resumo em português ...

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

Abstract

Abstract ...

Keywords: keyword1, keyword2, keyword3.

Sumário

Lista de Abreviaturas

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

Lista de Símbolos

Lista de Figuras

Lista de Tabelas

Capítulo 1

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 1.1.

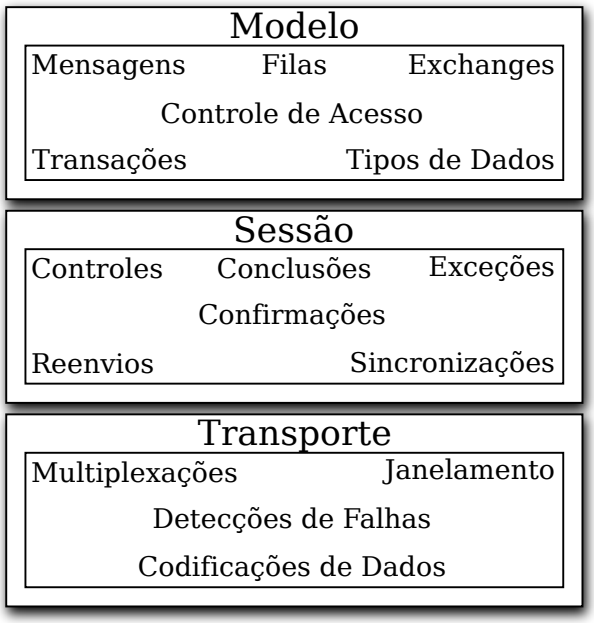

Figura 1.1: *Camadas do padrão AMQP.*

Nas seções seguintes descrevemos as camadas do padrão AMQP. A camada de modelo, que é a camada que define o conjunto de objetos que utilizamos em nosso trabalho, a camada de sessão que age como intermediária entre as camadas de modelo e transporte, e comentamos brevemente sobre a camada de transporte. Os detalhes dessa camada não se enquadram no escopo deste trabalho. Apresentamos também algumas das implementações disponíveis do padrão e qual implementação escolhemos para desenvolver este trabalho.

1.1 A camada de modelo

A camada de modelo é a camada responsável pela definição do conjunto de comandos e dos objetos que as aplicações podem utilizar, como por exemplo a criação de filas. A especificação dos requisítos para essa camada possui, entre outros itens: garantir a interoperabilidade das implementações, prover controle explícito sobre a qualidade do serviço, proporcionar um mapeamento fácil entre os comandos e as bibliotecas de nível de aplicação e ter clareza, de modo que cada comando seja responsável por uma única ação. Os componentes são:

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

A figura 1.2 mostra os componentes acima descritos e os relacionamentos entre esses componentes.

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

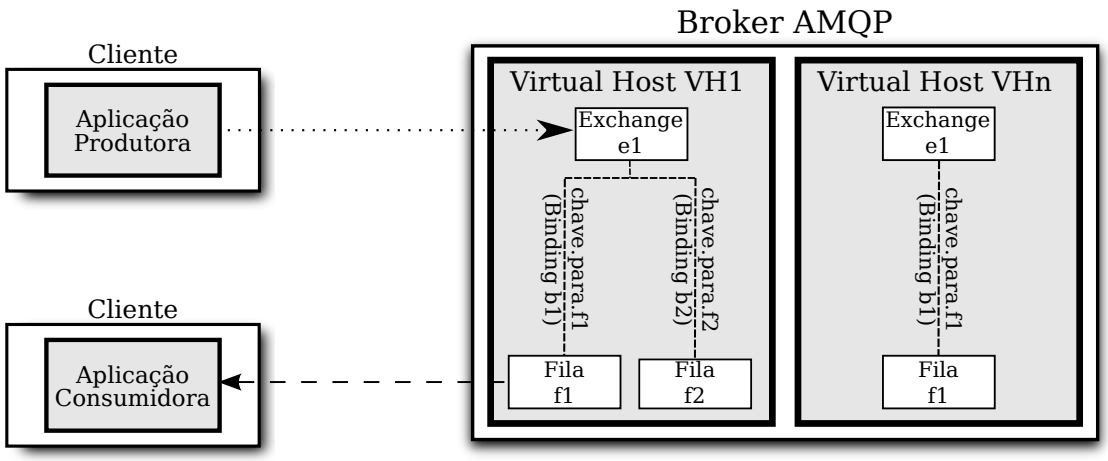

Figura 1.2: *Componentes da camada de modelo do padrão AMQP.*

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

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

1.1.1 Envios de mensagens

Para enviar uma mensagem, uma aplicação produtora especifica uma determinada *exchange* de um determinado *virtual host*, uma rotulação com informação de roteamento e eventualmente algumas propriedades adicionais, bem como os dados do corpo da mensagem. Uma vez que a mensagem tenha sido recebida no servidor AMQP, ocorre o roteamento para uma ou mais filas do conjunto de filas do *virtual host* especificado. No caso de não ser possível rotear a mensagem, seja qual for o motivo, as opções são: rejeitar a mensagem, descartá-la silenciosamente, ou ainda fazer o roteamento para uma *exchange* alternativa. A escolha depende do comportamento definido pelo produtor. Quando a mensagem é depositada em alguma fila (ou possivelmente em algumas filas), a fila tenta repassá-la imediatamente para a aplicação consumidora. Caso isso não seja possível, a fila mantém a mensagem armazenada para uma futura tentativa de entrega. Uma vez que a mensagem foi entregue com sucesso a um consumidor, ela é removida da fila. A figura 1.3 mostra o envio e recebimento de uma mensagem.

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

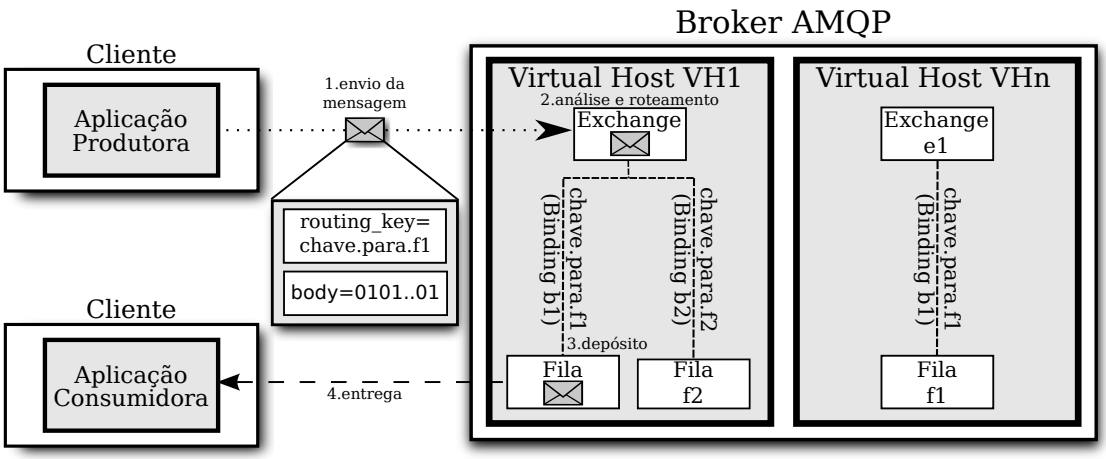

Figura 1.3: *Fluxo de uma mensagem no padrão AMQP.*

1.2 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 à aplicação servidora. É parte das suas responsabilidades dar confiabilidade às interações entre as aplicações cliente e a aplicação servidora.

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

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

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

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

- Notificação de comandos completados: Diferente do conceito de confirmação, esse serviço informa o par que solicitou a execução de um comando que esse foi executado por completo. As notificações de comandos completados tem como motivação a sincronização e a garantia de 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 em caso de falhas de rede: Para que o sistema possa se recuperar no caso de falhas de rede, a sessão deve ser capaz de reenviar comandos cujos recebimentos pelo

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

1.3 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.

1.4 Implementações

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

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

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

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

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

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

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

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

1.4.1 RabbitMQ - Biblioteca para clientes Java

A biblioteca [Jav] para clientes Java disponibilizada pelo projeto RabbitMQ 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 pelo objeto `com.rabbitmq.client.ConnectionFactory`, através do método `newConnection`. Na implementação, 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 1.3, 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` e através desta instância, podemos executar os comandos definidos na camada de sessão, como por exemplo a criação de filas (`queueDeclare`) ou o envio (`basicPublish`) e recebimento de mensagens (`basicConsume`). 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`.

A figura 1.4 mostra como uma fábrica de conexões, uma conexão, um canal e um consumidor se relacionam.

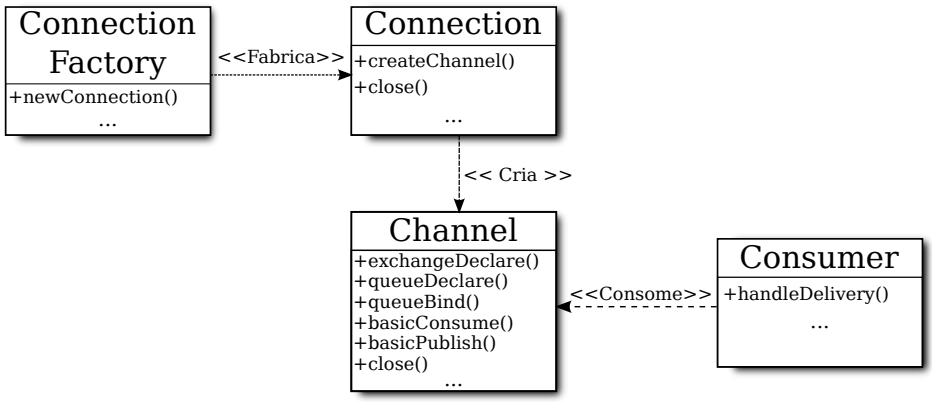

Figura 1.4: *RabbitMQ Java API – Relacionamento entre classes de transporte e sessão.*

A listagem 1.1 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 1.1: *Trait CommonAMQP*

Para o nosso exemplo optamos por criar a classe `SampleProducer` para encapsular a criação dos objetos no *broker* e o envio das mensagens. A classe é apresentada na listagem 1.2. 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*[2] direta e transiente, uma fila pública, transiente e temporária e faz o *binding* de ambos. O método `publish` por sua vez, verifica logo na sua primeira linha se o estado da instância é válido, e em seguida itera na sequência de mensagens fazendo o envio. Para esse exemplo em particular, omitimos as propriedades adicionais (`BASIC_PROPS`) que podem ser utilizadas no envio. Exemplos dessas propriedades são definir a prioridade da mensagem, especificar um remetente diferente para resposta (*replyTo*) e definir uma data para expiração da mensagem.

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

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

```
24 }
```

Listagem 1.2: *Classe SampleProducer*

A listagem 1.3 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 1.3: *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 1.4. O método `startConsumer`, assim como o método `startProducer` mostrado na listagem 1.2, se conecta ao servidor AMQP e abre um canal para interagir com a camada de sessão. Em seguida, ele registra a instância corrente como um consumidor na fila previamente criada com confirmação implícita de recebimento. O método `handleDelivery` é o método invocado a cada mensagem recebida. Podemos notar que o método, além de receber o corpo da mensagem, ainda recebe algumas informações sobre o envio. Os demais métodos são herdados da interface `com.rabbitmq.client.Consumer` e servem como notificadores para outros eventos, como registro (`handleConsumerOk`) ou cancelamento (`handleCancelOk`) de um consumidor na fila. Mais detalhes sobre a interface `com.rabbitmq.client.Consumer` e de suas implementações podem ser encontradas em [Jav].

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

Listagem 1.4: *Classe SampleConsumer*

A listagem 1.5 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 1.5: *Aplicação SampleConsumerApplication*

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

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, apresentamos na seção 2.3 algumas linguagens baseadas no modelo de atores e algumas bibliotecas desenvolvidas em linguagens mais gerais.

2.1 Modelo

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

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

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

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

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

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

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

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

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

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

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

Apesar de o modelo ser bem explícito e claro em relação a interação ser via troca de mensagens, e dizer que o estado de um ator não deve ser compartilhado, implementações poderiam permitir que o estado interno de um ator fosse acessado diretamente por outro ator. Por exemplo, um ator em sua pilha de execução fazer a invocação de algum método de outro ator. Mesmo que a interface de um ator não permita o compartilhamento do seu estado interno que não seja via troca de mensagens, as mensagens enviadas podem expor um compartilhamento indesejado caso não sejam imutáveis e

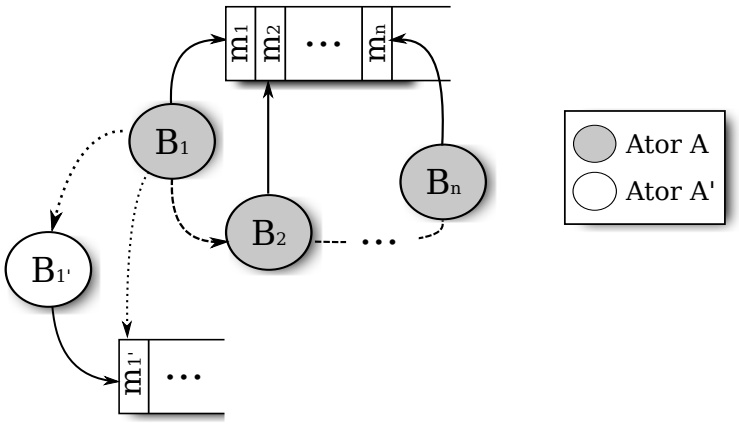

Figura 2.1: *Máquina de estados de um ator.*

passadas por cópia.

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

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

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

2.2 Breve histórico

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

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

O conceito de guardião foi definido em 1978 por Attardi e Hewitt. Um guardião é uma entidade que regula o uso de recursos compartilhados. Guardiões incorporam explitamente a noção de estado e são responsáveis por agendar o acesso e fazer a proteção dos recursos. Hewitt e Atkinson

definiram em 1979 um conceito relacionado ao de guardião denominado seriador (*serializer*). Um seriador age como um monitor, porém, ao invés de aguardar sinais explícitos vindos dos processos, seriadores procuram ativamente por condições que permitam que processos em espera possam voltar a ser executados. Em 1987, Henry Lieberman implementou em Lisp a linguagem Act1 [Lie87], uma linguagem de atores com guardiões, seriadores e atores chamados de "trapaceiros" (*rock-bottom*). Atores trapaceiros são atores que podem burlar as regras do modelo de atores para, por exemplo, usar dados primitivos e funções da linguagem usada em sua implementação.

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

2.3 Implementações

O suporte ao modelo de atores pode estar disponível em uma linguagem de programação, seja fazendo parte da estrutura da linguagem, como uma biblioteca embutida em sua distribuição ou ainda como uma biblioteca separada da distribuição padrão. Apresentamos a seguir algumas linguagens de programação baseadas no modelo de atores, e que possuem primitivas que facilitam a criação de sistemas baseados em atores. Apresentamos também 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 que fossem modelados como componentes que interajem entre si, pudessem ser traduzidos naturalmente para código. Pelo fato de ser uma linguagem da plataforma .Net, Axum pode interagir com outras linguagens da plataforma, como VB.Net, C# e F#.

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

A linguagem possui ainda o conceito de domínio. Domínios são definidos com a palavra-chave `domain` e permitem que agentes definidos dentro de um domínio compartilhem informações de um

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

modo diferente. Uma instância de domínio pode ter atributos que são usados para compartilhamento seguro e controlado de informações.

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

SALSA

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

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

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

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

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

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

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

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

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

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

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

Erlang

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

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

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

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

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

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

2.3.2 Bibliotecas

Diversas linguagens de programação possuem suporte ao modelo de atores via bibliotecas. Essas bibliotecas disponibilizam arcabouços para desenvolvimento de sistemas concorrentes baseados

no modelo de atores. Listamos a seguir algumas linguagens e referências para algumas de suas bibliotecas:

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

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

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

A biblioteca de atores de Scala

A distribuição de Scala inclui uma biblioteca de atores inspirada pelo suporte a atores em Erlang. A biblioteca Scala Actors oferece basicamente as funcionalidades já descritas na seção 2.3.1.

Atores foram projetados como objetos baseados em *threads* Java e possuem métodos como `send`, `!`, `receive`, além de outros métodos como `act` e `react`. Cada ator possui uma caixa de correio para o recebimento e armazenamento temporário das mensagens. O processamento de uma mensagem é feito por um bloco `receive` declarado dentro do método `act`. No método `receive` são definidos os padrões a serem casados com as mensagens que o ator processa e as ações associadas. A primeira mensagem que casar com qualquer dos padrões é removida da caixa de correio e a ação correspondente é executada. Caso não haja casamento com nenhum dos padrões, o ator é suspenso.

Atores são executados em um *thread pool*[3] que cresce conforme a demanda. É importante ressaltar que o uso do método `receive` fixa o ator à *thread* que o está executando, limitando superiormente a quantidade de atores pelo número de *threads* que podem ser criadas. É recomendado o uso do método `react` ao invés do método `receive` sempre que possível, já que um ator que não está em execução cede sua *thread* para que ela execute outro ator. Do ponto de vista de funcionalidade, o método `receive` é bloqueante e pode retornar valores, enquanto que o método `react`, além de não retornar valores, faz com que o ator passe a reagir aos eventos (recebimentos de mensagens). As implicações práticas do uso do `react` em relação ao `receive`, da perspectiva de codificação, são: o uso da estrutura de controle `loop` ao invés de laços tradicionais, para indicar que o ator deve continuar reagindo após o processamento de uma mensagem (o uso dos laços tradicionais bloquearia a *thread*); o ator ser responsável por invocar eventuais métodos após o processamento de uma mensagem [HO09].

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

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

O suporte a atores remotos faz parte da biblioteca, porém com algumas restrições em comparação com os atores de Erlang. Em Scala não é possível a criação de um ator em um nó que não seja o local, ou seja, *remote spawns* não são possíveis. 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 hospedeiro e porta), utilizando como chave o nome sob o qual o ator foi registrado. Esta abordagem, apesar de soar restritiva, evita um problema importante que é a necessidade de carga remota da classe do ator (*remote class loading*) e torna desnecessário o uso de interfaces remotas como em Java RMI. O tráfego das mensagens é feito via seriação padrão Java através de *sockets* TCP.

O projeto Akka

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

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

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

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

[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".

Nas próximas duas seções examinaremos a implementação de atores do projeto Akka. Mostraremos a criação e o uso de atores locais ou remotos, bem como o fluxo de uma mensagem (envio, tráfego e processamento da mensagem) tanto no caso local como no remoto.

3.1 Atores locais

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

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

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

Listagem 3.1: *Classe SampleActor.*

Atores são criados pelo método `actorOf` do objeto `akka.actor.Actor` como mostrado na listagem 3.2. Nessa listagem, a instância do ator é criada por reflexão com base no tipo da classe, onde o construtor padrão é invocado. O método `actorOf` é sobrecarregado para permitir que uma

função sem argumentos e com tipo de retorno `Actor`, possa ser utilizada como alternativa ao construtor padrão na criação do ator, como mostrado na listagem 3.3.

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

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

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

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

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

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

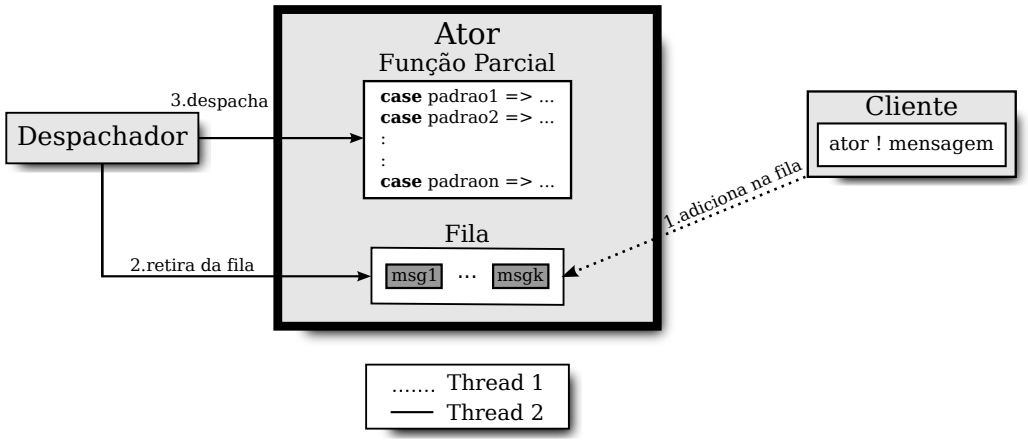

Figura 3.1: *Envio e despacho de mensagens para atores locais.*

3.1.1 Despachadores

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

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

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

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

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

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

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

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. Essa cópia é enviada junto com a mensagem para que o ator destinatário possa eventualmente enviar uma mensagem de resposta.

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

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

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

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 descrita na seção 2.3.1, conhecida como "deixe que falhe" (*let it crash*).

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

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

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

A definição de um ator com ciclo de vida permanente é mostrada na listagem 3.4. A definição do ciclo de vida do ator acontece na linha 2.

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

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

```
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 3.4: *Ator com ciclo de vida permanente.*

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

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

Junto com a estratégida de reinicialização, é necessário definir quais são as exceções que o ator é responsável por fazer o tratamento, além de configurações relacionadas a quantas tentativas de reinicialização devem ser feitas dentro de um período de tempo. Mostramos na listagem 3.5 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 3.5: *Ator supervisor.*

Apresentamos na listagem 3.6 a criação de uma hierarquia de supervisão entre os atores apresentados nas listagens 3.4 e 3.5.

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

Listagem 3.6: *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 3.6, a referência para o ator supervisor passa a estar definida no ator `supervised`. Essa referência é acessível via método `self.supervisor`. Um ator que supervisiona outros atores 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.

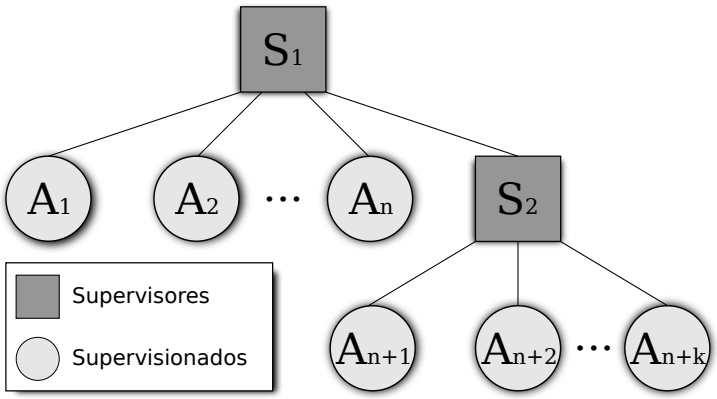

Figura 3.2: *Hierarquia de supervisão de atores.*

3.2 Atores remotos

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

- `RemoteServerModule`: É uma *trait* que define as responsabilidades do componente usado no lado do servidor. Suas implementações têm como responsabilidade manter registrados os atores, bem como encaminhar a eles as mensagens recebidas de clientes remotos. Cada `RemoteServerModule` é associado a um *host* e a uma porta TCP. Um mesma máquina virtual pode conter múltiplos `RemoteServerModule`s. A documentação do Akka utiliza para esse componente a terminologia "servidor remoto" (*remote server*);
- `RemoteClientModule`: É uma *trait* que define as responsabilidades do componente usado no lado do cliente. Suas implementações têm como responsabilidade visível oferecer uma interface para obtenção de referências a atores remotos. Oferece também suporte em tempo de execução para a infra-estrutura de atores do lado do cliente, provendo uma série de serviços não vísiveis para o usuário, tais como seriação de mensagens, envio de mensagens para atores remotos, conversão de ator local em ator remoto e intermediação de mensagens de resposta vindas do `RemoteServerModule`, no caso envios via `!!` ou `!!!`. A documentação do Akka utiliza para esse componente a terminologia "cliente remoto" (*remote client*);
- `RemoteSupport`: É acessível via `Actor.remote`. Essa classe abstrata é responsável por concentrar as responsabilidades definidas para os módulos remotos do cliente e do servidor e ser um ponto único de suporte remoto;
- `RemoteActorRef`: É uma classe equivalente a `LocalActorRef`, porém utiliza o suporte remoto para fazer envio das mensagens.

A implementação de atores remotos do Akka, implementa os componentes de suporte remoto com o auxílio do JBoss Netty. Apresentamos na figura 3.3 como os componentes se relacionam. Dividimos a figura em duas partes, sendo a camada de interface remota que é a camada que se

interage diretamente, e a camada de implementação. A camada de implementação é associada em tempo de execução, permitindo que outras implementações possam ser criadas e 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 3.7. Nessa listagem, um servidor remoto é inicializado na linha 2 para que os atores possam ser registrados. Na implementação padrão feita com o Netty, a inicialização de um servidor remoto implica na associação de um componente do Netty para monitorar a *socket* associada ao par *host* e porta. Na linha 3 registramos o ator sob o nome `hello-service`. O ator passou a estar remotamente acessível por aplicações que residam em outras máquinas virtuais. Um detalhe importante a ser destacado é a inicialização implícita do ator feita pelo método `register`. A implementação desse método fica a critério dos componentes da camada de implementação de suporte remoto, porém sua interface obriga que implementações compatíveis inicializem atores que ainda não estão em execução.

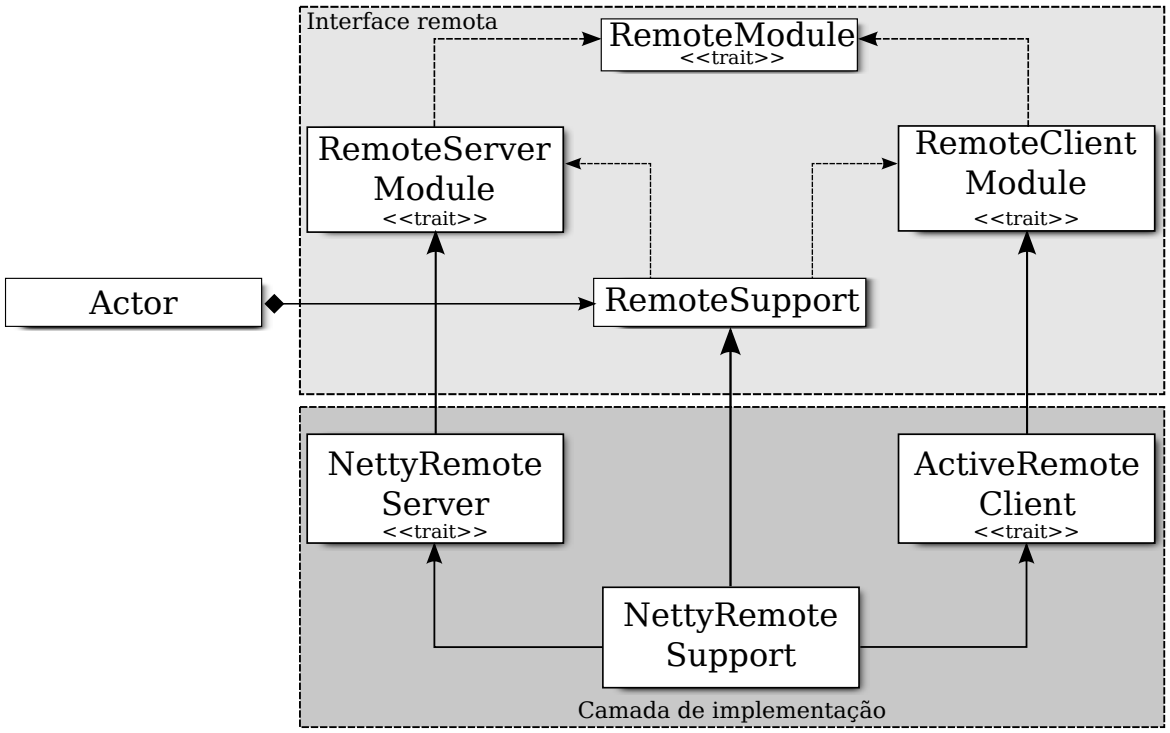

Figura 3.3: *Relacionamento entre os componentes remotos.*

Referências para atores remotos podem ser obtidas via método `actorFor`. Esse método possui diversas sobrecargas, porém a mais simples recebe como argumentos o nome do ator, o *host* e a porta onde foi feito o registro. A linha 2 da listagem 3.8 mostra como uma aplicação cliente obtém uma referência para o ator. Podemos notar que o uso do ator na linha 3 não possui nenhuma indicação explícita sobre um envio remoto. O método `actorFor` retorna na maioria das vezes instâncias de `RemoteActorRef`, que representam *proxies* locais para o ator remoto. A implementação possui algumas otimizações para que referências locais sempre que possível. Por exemplo, a entidade que está enviando a mensagem reside no mesmo nó que o ator. Em casos como esse, uma instância de `LocalActorRef` é retornada.

```
1 object SampleRemoteServer extends Application {
2   Actor.remote.start("localhost", 2552)
```

```
3   Actor.remote.register("hello-service", Actor.actorOf[SampleActor])
4 }
```

Listagem 3.7: *Aplicação SampleRemoteServer.*

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

Listagem 3.8: *Aplicação SampleRemoteClient.*

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

3.2.1 Fluxo de envio das mensagens

Quando um cliente envia uma mensagem a um ator remoto, alguns passos adicionais acontecem em relação a um envio local, como mostrado na figura 3.4. O processo começa com o *proxy* local embrulhando a mensagem e adicionando a ela informações de cabeçalho necessárias para o envio e posterior processamento. O seriador é responsável por converter a informação recebida em um vetor de *bytes* para que o transporte possa ocorrer. Uma vez que a informação esteja no formato a ser transportado, o *proxy* usa uma implementação de `RemoteSupport` (que na figura 3.4 é uma instância de `NettyRemoteSupport`) para enviar a mensagem ao `RemoteSupport` que está no lado do servidor. Este processo de envio, do ponto de vista do cliente leva tempo $O(1)$, já que o envio da mensagem é feita de modo assíncrono.

Uma vez que a mensagem tenha sido recebida pelo *handler* plugado ao JBoss Netty no lado servidor, o *handler* repassa a informação recebida para o seriador. O seriador faz o processo inverso do envio, repassando uma representação desseriada da informação recebida para o *handler*. O *handler* examina as informações de cabeçalho adicionadas na mensagem, localiza o ator destinatário no registro local de atores e encaminha a mensagem à caixa de mensagens do ator fazendo um envio local. Esse envio local respeita o remetente remoto original da mensagem. O despachador associado ao ator tem o mesmo comportamento já mostrado na figura 3.1.

Em alguns casos o processamento deve gerar uma mensagem de resposta, como por exemplo, um resultado futuro ou uma mensagem de erro para um ator supervisor. Em tais casos, a mensagem de resposta faz o caminho inverso. O *handler* ao receber a mensagem, utiliza o seriador e faz o repasse da resposta para quem a estiver aguardando.

O Akka possui uma configuração opcional de segurança onde cada JVM que esteja executando o arcabouço pode definir um *cookie*. O *cookie* nada mais é do que uma chave que o usuário pode gerar e que é parte das informações de cabeçalho das mensagens. Quando a verificação de segurança está ativada, o recebedor da mensagem verifica se a informação do *cookie* presente na mensagem

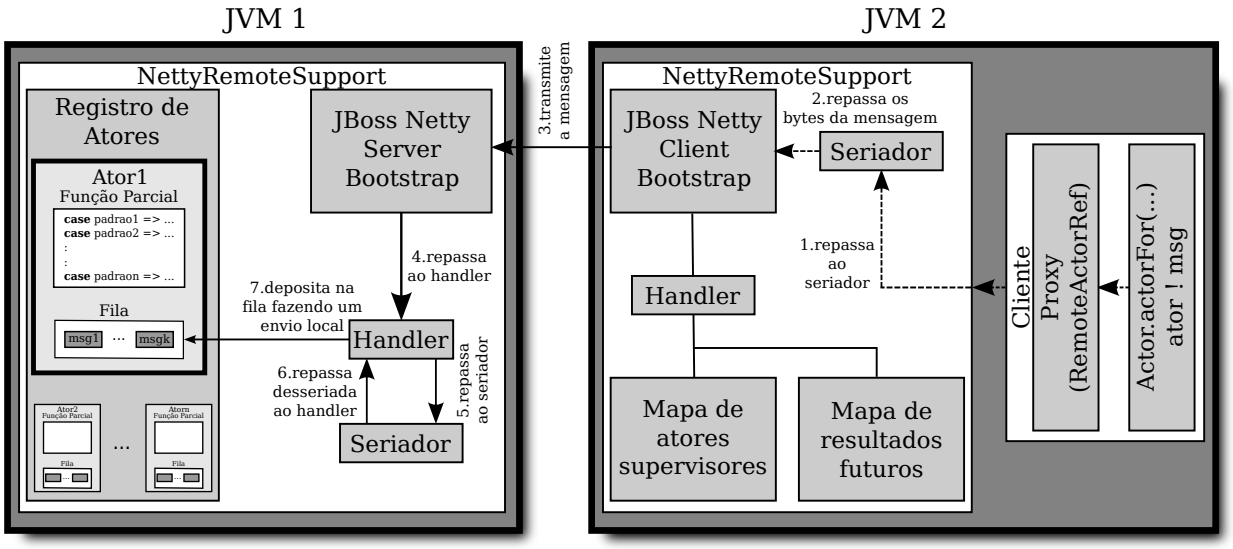

Figura 3.4: *Fluxo de envio de mensagens para atores remotos.*

confere com a informação esperada. Essa verificação ocorre entre os passos 6 e 7 da figura 3.4 e caso os valores sejam iguais, o passo 7 será executado. Caso contrário, uma excessão de segurança é lançada na JVM que recebeu a mensagem 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`.

3.2.2 Protocolo para envios de mensagens a atores remotos

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

Protobuf é uma biblioteca que permite que dados estruturados sejam representados em um formato eficiente e extensível. A biblioteca possui uma espécie de compilador que converte a definição de protocolos em classes Java, Python e C++, permitindo uma interação direta com o protocolo nas linguagens, sem a necessidade de bibliotecas adicionais para conversões. A bibliteca permite que protocolos sejam definidos com a palavra-chave `message`. Protocolos podem contér outros protocolos e tipos primitivos de dados, como por exemplo números, textos, boleanos e vetores de dados. Ademais, pode-se definir a obrigatoriedade dos valores para alguns atributos com as palavras-chaves `optional` e `required`.

O protocolo para endereços remotos é apresentado na listagem 3.9. Endereços remotos são utilizados por referências remotas para identificar, por exemplo, onde o ator remoto foi criado. O protocolo para referências de atores remotos é apresentado na listagem 3.10. As mensagens para atores remotos são definidas com base no tipo de seriação que foi definido na JVM que fez 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. Os protocolos para definição dos tipos de seriações e das mensagens são mostrados nas listagens 3.11 e 3.12, respectivamente. Por fim, o protocolo utilizado para as mensagens remotas que é composto, dentre outros protocolos menores, pelo protocolo para

mensagens e pelo protocolo para referências remotas, é apresentado na listagem 3.13.

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.

```
1 message AddressProtocol {
2   required string hostname = 1;
3   optional uint32 port = 2;
4 }
```

Listagem 3.9: *Protocolo para endereços remotos.*

```
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 3.10: *Protocolo para referências de atores remotos.*

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

Listagem 3.11: *Enumeração com tipos de seriação suportados.*

```
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 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 3.13: *Protocolo para mensagens remotas.*

Capítulo 4

Troca de mensagens entre entidades remotas via AMQP

Apresentamos nesse capítulo a estrutura que definimos para dar suporte a troca de mensagens entre atores remotos via *message broker* AMQP. Entretanto, antes de apresentar essa estrutura comentamos um pouco sobre a conexão de entidades remotas com a abordagem ponto-a-ponto, e algumas implicações dessa abordagem em uma implementação onde as entidades remotas são atores.

4.1 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 precisão saber detalhes de conexão e do protocolo das mensagens a serem 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 saiba o endereço do nó onde está a outra entidade. Em casos onde as entidades estejam em nós que não fazem parte de uma rede interna, a informação de localidade pode não indicar exatamente o nó corrente da entidade, mas o endereço de alguma porta de ligação (*gateway*).

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

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

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

4.1.1 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.

Na implementação de atores remotos do Akka apresentada na seção 3.2, vimos que os atores são agrupados por *host* e porta, sendo armazenados no `RemoteServerModule`. Vimos também na seção 3.1 que atores são identificáveis, o que permite que eles sejam localizados no `RemoteServerModule` para que possam fazer o recebimento e processamento das mensagens.

Ainda que a implementação de atores remotos do Akka agrupe os atores, removendo a relação do limite do número de atores com o número de portas disponíveis, as entidades `RemoteServerModule` e `RemoteClientModule` ainda assim possuem o conhecimento do *host* e porta aonde se está conectado e não fazem armazenamento intermediário das mensagens.

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

4.2 Entidades conectadas via *message broker* AMQP

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

Tanto a rotulação das entidades por *host* e porta ou por nome e *host* deixam de fazer sentido pois são redundantes. As entidades passam a ser identificadas somente por seus nomes. Cada nova entidade que entrar no sistema é responsável por registrar uma fila no *broker* e associá-la a uma *exchange* afim de poder receber mensagens. Dependendo do papel que a entidade exerce, além de definir a fila e a associação, a entidade precisa definir antes a sua *exchange*. Entidades que possuem o papel de servidora têm essa responsabilidade. Deve também haver algum tipo de mapeameamento entre o nome associado a entidade e os objeto que ela define. Tarefas administrativas como a criação de *virtual hosts*, a criação de usuários e a definição das permissões dos usuários não são de responsabilidade das entidades, e devem ser executadas anteriormente.

Para dar suporte ao desenvolvimento dos novos componentes da camada de implementação da figura 3.3, definimos alguns componentes intermediários. Esses componentes intermediários formam uma ponte entre a nova camada de implementação desenvolvida e a implementação RabbitMQ [RMQ], e são apresentados na figura 4.1.

O módulo central `AMQPBridge` é responsável por fornecer as funcionalidades básicas para troca de mensagens como envios, recebimentos, tratamento de mensagens retornadas e suporte a múltiplas entidades cliente em uma entidade servidora. O módulo e contém as seguintes classes:

- `AMQPBridge`: É a classe abstrata que define o comportamento básico de uma entidade conectada a um *broker* AMQP. O construtor de `AMQPBridge` recebe como argumentos o nome que identifica a ponte e uma conexão para um *broker* AMQP. Essa classe define ainda o padrão dos nomes a serem utilizados na criação dos objetos no *virtual host* acessível pela conexão recebida no construtor;
- `MessageHandler`: É a *trait* que define os métodos a serem invocados quando mensagens forem recebidas ou quando envios feitos pela entidade forem rejeitados pelo *broker*;

- `ServerAMQPBridge`: É uma das sub-classes de `AMQPBridge` e define o comportamento para entidades que atuam como servidoras. 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;

- `ClientAMQPBridge`: É uma das sub-classes de `AMQPBridge` e define o comportamento para entidades que atuam como clientes de um serviço. Essa classe tem em seu construtor padrão um parâmetro adicional para identificação do cliente. A identificação é necessária para compôr a chave de roteamento que faz a associação da fila que essa ponte irá criar no método `setup`, com a *exchange* correspondente definida anteriormente a criação da fila.

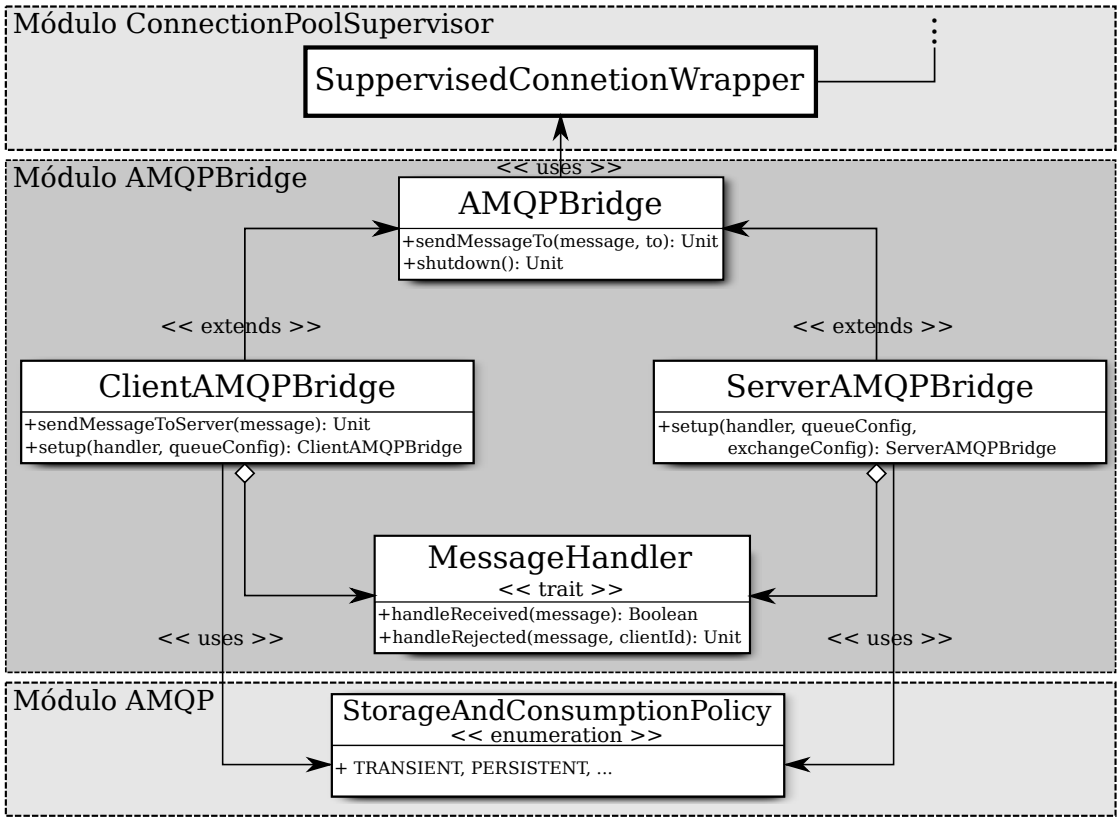

Figura 4.1: *Módulos de acesso ao broker AMQP.*

Definimos também o módulo `AMQP` que possui algumas classes com configurações pré-definidas para os objetos a serem criados no *broken*. A enumeração `StorageAndConsumptionPolicy` define um conjunto de valores que definem as políticas de armazenamento e consumo. A enumeração é composta pela combinação de outras duas enumerações: `QueueConfig` e `ExchangeConfig`. Essas duas enumerações definem as configurações específicas e detalhadas para filas e *exchanges*. A combinação dos seus valores dão semântica às políticas de configuração. Por exemplo, a configuração `EXCLUSIVE_PERSISTENT` presente em `StorageAndConsumptionPolicy` é uma combinação de `ExchangeConfig.exchangeDurable` e `QueueConfig.exclusiveQueueDurable`, e contém informações sufientes para a criação de uma *exchange* durável e uma fila também durável, com acesso exclusivo a um único consumidor.

A figura 4.2 mostra a disposição dos objetos criados pelas implementações de `AMQPBridge` na execução do método `setup` de cada classe. Nessa figura podemos ver que um `ServerAMQPBridge` cujo nome é `node1` define a *exchange* que é utilizada como ponto central de envios de mensagens para todas as entidades que possuam o nome `node1`, sejam elas com responsabilidades de cliente ou

de servidora. O *binding* para a fila de entrada da entidade servidora utiliza o padrão de nomeação `to.server.<name>`, onde `<name>` é substituído pelo nome da entidade durante a sua criação. Ambas a fila de entrada e a *exchange* criadas pela entidade servidora seguem um padrão de nomeação semelhante, já que o nome da ponte é utilizado como sufixo.

Cada entidade cliente cria uma fila de saída para que a entidade servidora possa enviar mensagens de resposta às entidades cliente. As filas de saídas utilizam como nomeação o padrão `actor.queue.out.<name>.<id>`, onde `<name>` corresponde ao nome da entidade e `<id>` ao identificador. Esses mesmos parâmetros são utilizados para compôr a chave de associação que liga a fila a *exchange* definida previamente.

O envio de mensagens de um `ClientAMQPBridge` para um `ServerAMQPBridge` acontece como já explicado na sessão 1.1.1. A mensagem é enviada para a *exchange* previamente definida relacionada a `ClientAMQPBridge` via método `sendToServer`. Esse método utiliza como chave de roteamento o valor definido com base no nome da ponte. É de responsabilidade da entidade que envia a mensagem adicionar, como parte da mensagem, a identificação do remetente. Essa identificação será utilizada, caso haja uma mensagem de resposta, como chave de roteamento.

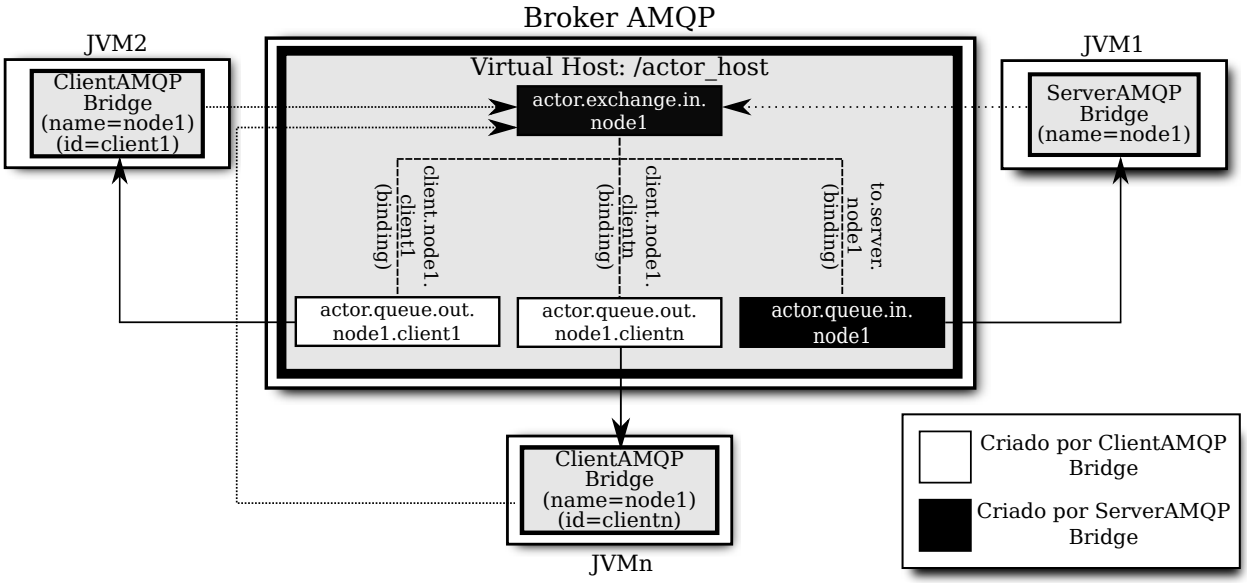

Figura 4.2: *Estrutura para troca de mensagens entre entidades remotas via broker AMQP.*

O módulo `ConnectionPoolSupervisor` que também é mostrado na figura 4.1 é responsável por encapsular os detalhes das conexões e dos canais utilizados para o envio e recebimento de mensagens com o *broker*. Esse módulo provê uma implementação segura para acesso concorrente aos canais. Como já comentado na seção 1.4.1, é de responsabilidade da aplicação controlar o acesso concorrente aos canais abertos junto à conexão. Optamos por encapsular tanto a conexão quanto os canais dentro de atores.

Referências Bibliográficas

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986. 11, 12

[Akk] Akka, Simpler Scalability, Fault-Tolerance, Concurreny & Remoting through Actors. http://www.akka.io. Último acesso em 15/2/2011. 17

[AMQ08] AMQP Working Group. *AMQP Specification v0.10*, 2008. 1

[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. 12, 13

[API] RabbitMQ – API Guide. http://www.rabbitmq.com/api-guide.html. Último acesso em 12/5/2011. 9

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

[App09] Apple Inc. *Technology Brief – Grand Central Dispatch*, 2009. 21

[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. 5, 16

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007. 14, 16

[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. 17

[Cora] Microsoft Corporation. Asynchronous agents library. http://msdn.microsoft.com/en-us/library/dd492627.aspx. Último acesso em 27/5/2011. 17

[Corb] Microsoft Corporation. Axum programming language. http://msdn.microsoft.com/en-us/devlabs/dd795202. Último acesso em 24/5/2011. 15

[Cor09] Microsoft Corporation. *Axum – Language Overview*, Junho 2009. 14

[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. 18

[FPV98] Alfonso Fuggetta, Gian Pietro Picco e Giovanni Vigna. Understanding code mobility. *IEEE Trans. Softw. Eng.*, 24:342–361, 1998. 13

[Goo] Protocol Buffers – Google's data interchange format. http://code.google.com/p/protobuf. Último acesso em 27/5/2011. 18, 27

[Gro] AMQP Working Group. Advanced message queuing protocol. http://www.amqp.org. Último acesso em 5/5/2011. 1

[Har] Mark Harrah. Library for describing binary formats for Scala types. https://github.com/harrah/sbinary. Último acesso em 27/5/2011. 18

[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. 13

[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. 13

[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. 17

[Ins81] Information Sciences Institute. Transmisson control protocol – protocol specification. Relatório Técnico RFC 793, University of Southern California, Set 1981. 29

[Jav] RabbitMQ Java API 2.4.1 Javadoc. http://www.rabbitmq.com/releases/rabbitmq-java-client/v2.4.1/rabbitmq-java-client-javadoc-2.4.1. Último acesso em 12/5/2011. 6, 8

[JBo] Netty – the Java NIO Client Server Socket Framework. http://www.jboss.org/netty. Último acesso em 27/5/2011. 18

[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. 13

[Kim97] Wooyoung Kim. *THAL: An Actor System For Efficient and Scalable Concurrent Computing.* Tese de Doutorado, University of Illinois at Urbana-Champaign, 1997. 17

[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. 17

[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. 17

[Lee] Jacob Lee. Parley. http://osl.cs.uiuc.edu/parley/. Último acesso em 26/5/2011. 17

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

[Lie87] Henry Lieberman. *Concurrent object-oriented programming in Act 1*, páginas 9–36. MIT Press, Cambridge, MA, USA, 1987. 14

[Mas] Ashton Mason. Theron multiprocessing library. http://theron.ashtonmason.net/. Último acesso em 26/5/2011. 17

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

[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. 17

[PA94] Rajendra Panwar e Gul Agha. A methodology for programming scalable architectures. *J. Parallel Distrib. Comput.*, 22:479–487, 1994. 13

[Pro03] Java Community Process. *JSR-000914 JavaTM Message Service (JMS) API*, Dezembro 2003. 5

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

[Reta] Mike Rettig. Jetlang – Message based concurrency for Java. http://code.google.com/p/jetlang. Último acesso em 27/5/2011. 17

[Retb] Mike Rettig. Retlang – Message based concurrency in .Net. http://code.google.com/p/retlang. Último acesso em 27/5/2011. 17

[RMQ] RabbitMQ - Messaging that just works. http://www.rabbitmq.com/. Último acesso em 11/5/2011. 5, 30

[Rub] Rubinius. http://rubini.us/. Último acesso em 26/5/2011. 17

[Sca] Scalaz: Type Classes and Pure Functional Data Structures for Scala. http://code.google.com/p/scalaz. Último acesso em 27/5/2011. 17

[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. 17

[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. 17

[Tis] Christian Tismer. Stackless python. http://www.stackless.com/. Último acesso em 26/5/2011. 17

[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. 14

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

Índice Remissivo