
Universidade de São Paulo
Instituto de Matemática e Estatística
Departamento de Ciência da Computação

Uso de AMQP para Transporte de Mensagens entre Atores Remotos

Exame de Qualificação de Mestrado

São Paulo, Novembro de 2010

Thadeu de Russo e Carmo

ORIENTADOR:

Francisco Carlos da Rocha Reverbel

Resumo

O modelo de atores tem sido visto como uma abordagem alternativa à programação concorrente convencional, baseada em travas e variáveis de condição. Atores são agentes computacionais que se comunicam por troca de mensagens e que possuem uma caixa de correio e um comportamento. As mensagens destinadas a um ator são armazenadas na caixa de correio do ator e processadas de maneira assíncrona.

Sistemas de *middleware* orientados a mensagens trabalham com troca assíncrona de mensagens e formam uma base que simplifica o desenvolvimento de aplicações distribuídas. Tais sistemas permitem interoperabilidade com baixo acoplamento e provêm suporte para o tratamento robusto de erros em caso de falhas. *Message brokers* são frequentemente apresentados como uma tecnologia que pode mudar a maneira com que sistemas distribuídos são construídos. A especificação AMQP é uma proposta recente de padronização de um protocolo para *message brokers*.

Este trabalho pretende explorar a potencial sinergia entre um *message broker* e uma implementação do modelo de atores. Criaremos uma versão modificada da implementação do modelo de atores do projeto Akka que utilize um *message broker* AMQP como mecanismo de transporte de mensagens para atores remotos.

Sumário

1	Introdução	1
1.1	Troca de mensagens em ambientes corporativos	1
1.2	Modelos não convencionais de programação concorrente	2
1.3	Objetivo	3
1.4	Organização do texto	3
2	Trabalhos relacionados	5
2.1	Atores em Erlang	5
2.2	A biblioteca de atores de Scala	6
2.3	O projeto Akka	7
3	Background	8
3.1	AMQP	8
3.2	Atores no projeto Akka	11
3.2.1	Atores locais	11
3.2.2	Atores remotos	12
4	O projeto	15
5	Próximas etapas	16
5.1	Atividades programadas	16
5.2	Cronograma	17

1 Introdução

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

1.1 Troca de mensagens em ambientes corporativos

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

MOMs formam uma base que simplifica o desenvolvimento de aplicações distribuídas, permite interoperabilidade com baixo acoplamento e provê suporte para o tratamento robusto de erros em caso de falhas. Eles são frequentemente apresentados como uma tecnologia que pode mudar a maneira com que sistemas distribuídos são construídos [5].

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

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

Os protocolos usados por *message brokers* variam de produto para produto. A especificação

de Java *Messaging Service* (JMS) define uma API padrão para que programas Java possam interagir com *message brokers*. Boa parte dos *message brokers* têm implementações do padrão JMS. Dentre os mais conhecidos, podemos destacar JBoss Messaging [13], IBM Websphere MQ [12] (mais conhecido como MQ Series) e Apache Active MQ [1].

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

1.2 Modelos não convencionais de programação concorrente

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

Um dos principais impactos que processadores *multicore* causam no desenvolvimento de programas está relacionado com o modo com que programas são escritos. Para usufruir ganhos de desempenho com esses processadores, programas precisam ser escritos de forma concorrente [21], uma tarefa que não é simples. A maioria das linguagens de programação e dos ambientes de desenvolvimento não são adequados para a criação de programas concorrentes [22].

A abordagem convencional ao desenvolvimento de programas concorrentes é baseada em travas e variáveis condicionais. Em linguagens orientadas a objetos, como Java e C#, cada instância implicitamente possui sua própria trava, e travamentos podem acontecer em blocos de código marcados como sincronizados. Essa abordagem não permite a componibilidade de travas de maneira segura, criando situações propensas a bloqueios e impasses (*deadlocks*). A componibilidade de travas seria necessária quando houver mais de uma instância envolvida na ação a ser executada de modo exclusivo. Existem ainda outras dificuldades no uso de travas, como esquecimento de se obter a trava de alguma instância, obtenção excessiva de travas, obtenção de travas de instâncias erradas, obtenção de travas em ordem errada, manutenção da consistência do sistema na presença de erros, esquecimento de sinalização em variáveis de condição ou de se testar novamente uma condição após o despertar de um estado de espera [15]. Essas dificuldades mostram que a abordagem convencional, baseada em travas, é inviável para uma programação concorrente modular, ou seja, para a criação de grandes programas concorrentes compostos por programas menores. Elas impulsionaram a pesquisa em abordagens alternativas à programação concorrente convencional.

Dois modelos de programação concorrente não convencionais vem ganhando espaço recente-

mente. O primeiro deles é a memória transacional implementada por software (*software transactional memory*, ou STM) [20], um mecanismo de controle de concorrência análogo às transações de bancos de dados. O controle de acesso à memória compartilhada é responsabilidade da STM. Cada transação é um trecho de código que executa uma série atômica de operações de leitura e escrita na memória compartilhada.

O segundo modelo não convencional de programação concorrente é o modelo de atores. Atores [2] são definidos como agentes computacionais que possuem uma caixa de correio e um comportamento. Uma vez que o endereço da caixa de correio de um ator é conhecido, mensagens podem ser adicionadas à caixa para processamento assíncrono, ou seja, o envio de uma mensagem é desacoplado do processamento da mensagem pelo ator. Cada mensagem recebida por um ator é mapeada em uma 3-tupla que consiste de:

- um conjunto finito de envios de mensagens para atores cujas caixas de correio têm endereços conhecidos (um desses atores pode ser o próprio ator destinatário da mensagem que está sendo processada);
- um novo comportamento, que será usado para processar a próxima mensagem recebida;
- um conjunto finito de criações de novos atores.

A implementação do modelo de atores da linguagem Erlang usa casamento de padrões para especificar o tratamento a ser dado aos diferentes tipos de mensagem. Essa combinação do modelo de atores com as facilidades de casamento de padrões da linguagem mostrou-se muito efetiva [8].

1.3 Objetivo

O objetivo principal deste trabalho é criar uma implementação do modelo de atores que use AMQP para o transporte de mensagens entre atores remotos. Geraremos um protótipo baseado na implementação do modelo de atores do projeto Akka¹. A idéia é substituir o mecanismo atualmente usado pelo Akka por um *message broker* AMQP. Tencionamos também fazer uma análise comparativa da implementação de atores remotos do Akka e o nossa, baseada em AMQP.

1.4 Organização do texto

O restante deste texto é organizado da seguinte forma: a seção 2 discute trabalhos relacionados, a seção 3 apresenta tópicos que são pré-requisitos para o trabalho aqui proposto, a seção 4

¹O trabalho será desenvolvido tendo como base a última versão estável disponível para *download* no site do projeto.

apresenta o projeto que desenvolveremos e, por fim, a seção 5 enumera as atividades previstas e traz um cronograma para a conclusão de tais atividades.

2 Trabalhos relacionados

Esta seção apresenta os trabalhos relacionados que foram estudados e contribuíram para o desenvolvimento do nosso trabalho. Dividiremos essa seção em três partes, sendo na primeira parte a apresentação da linguagem Erlang e seu modelo de atores; na segunda e na terceira parte apresentaremos a linguagem Scala e o projeto Akka, com suas respectivas implementações do modelo de atores.

2.1 Atores em Erlang

Erlang [9] 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. 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, atores² são processos ultra leves criados dentro de uma máquina virtual. A criação, destruição e troca de mensagens entre atores é extremamente rápida, e como não existe compartilhamento de memória entre os atores, a única interação possível é via troca de mensagens. 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 3.5μs de tempo de CPU por ator e 9.2μs de tempo de relógio por ator [10].

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: (i) não se tem interesse em saber se um ator filho foi terminado normalmente ou não; (ii) caso um ator filho não tenha terminado normalmente, o ator criador também é terminado; (iii) 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

²Embora Erlang implemente o modelo de atores, sua literatura e suas bibliotecas não utilizam o termo “ator” (*actor*). O termo utilizado é “processo” (*process*).

para carga remota de código. Uma vez que alguns detalhes de infra-estrutura foram observados³, 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.

2.2 A biblioteca de atores de Scala

Scala [17] é uma linguagem moderna, com tipagem estática e que unifica os paradigmas de programação funcional e orientado a objetos. Vem sendo desenvolvida desde 2001 no laboratório de métodos de programação da EPFL (*Ecole Polytechnique Fédérale de Lausanne*). O código escrito em Scala pode ser compilado para execução tanto na JVM (*Java Virtual Machine*) quanto na CLR (*Common Language Runtime*). Diferentemente de Erlang, a linguagem Scala é consideravelmente extensa e possui muitos conceitos não implementados nem em Java e nem em C#. Exemplos de tais conceitos são: funções de ordem superior, criação de tipos de dados algébricos (*case classes*), casamento de padrões e funções parciais.

A implementação do modelo de atores não é parte da linguagem Scala, mas pode ser oferecida por bibliotecas. A distribuição de Scala inclui uma biblioteca de atores inspirada pelo suporte a atores em Erlang. Essa biblioteca oferece basicamente as funcionalidades já descritas na seção anterior. 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*, que define os padrões a serem casados com a mensagem aguardada e a ações associadas a cada padrão. 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.

Além de métodos para envio de mensagem 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: *(i) !?* faz envio síncrono e aguarda uma resposta dentro de um tempo limite especificado; *(ii) !!* 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 *host* e porta), utilizando como chave o nome sob o qual o

³As máquinas virtuais Erlang necessitam se autenticar umas com as outras.

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 serialização padrão Java através de *sockets* TCP.

2.3 O projeto Akka

O projeto Akka [4] 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. Suas principais características são: uma nova biblioteca de atores locais e remotos, suporte a STM, hierarquias de supervisão e uma combinação entre atores e STM (*“transactors”*) que dá suporte a fluxos de mensagens baseados em eventos transacionais, assíncronos e componíveis. Akka oferece, ainda, uma série de módulos adicionais para integração com outras tecnologias.

A biblioteca de atores do projeto Akka é totalmente independente da que é parte da distribuição de Scala, 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 na seção 3.2.2. Assim como a biblioteca de atores de Scala, o Akka oferece métodos para diferentes tipos de envio de mensagens: (i) *!!* semelhante ao método *!?* da biblioteca de atores de Scala, no qual o remetente fica bloqueado aguardando uma resposta durante um tempo limite; (ii) *!!!* semelhante ao método *!!* da biblioteca de atores de Scala, que devolve um resultado futuro ao remetente.

No que diz respeito à serialização das mensagens para um ator remoto, o Akka oferece as seguintes opções: JSON [16], Protobuf [11], SBinary [19] e serialização Java padrão. O transporte das mensagens é feito via TCP/IP, com o auxílio do JBoss Netty [14], um arcabouço para comunicação assíncrona dirigido a eventos e baseado em *sockets*. Este arcabouço oferece facilidades para compressão de mensagens, que são utilizados pelo Akka.

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

3 Background

As duas subseções a seguir abordam tópicos que são pré-requisitos para nosso trabalho: AMQP e a implementação do modelo de atores do projeto Akka.

3.1 AMQP

AMQP (*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 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 [6].

Os componentes principais do modelo AMQP 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 (*bindings*). 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.

- *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 mostra os componentes acima descritos e os relacionamentos entre esses componentes.

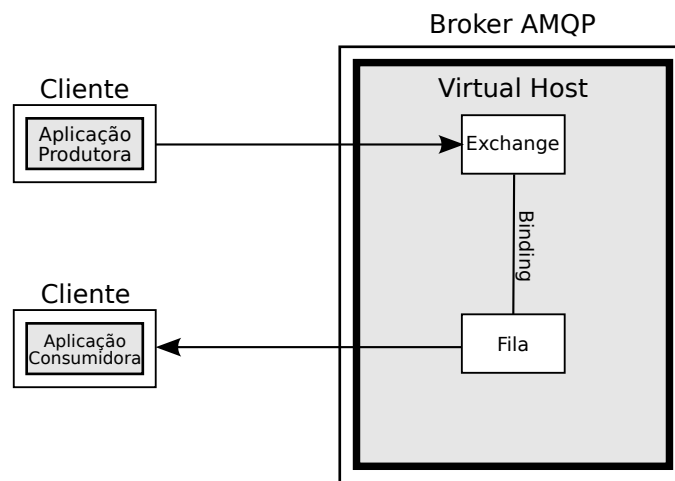


Figura 1: Visão geral dos componentes do modelo AMQP.

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

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

1. Uma mensagem AMQP é análoga a uma mensagem de *email*.
2. Uma fila é análoga a uma caixa de mensagens.
3. Um consumidor corresponde a um cliente de *email* que carrega e apaga as mensagens.

4. Uma *exchange* corresponde a um *mail transfer agent* (MTA) que inspeciona as mensagens e, com base nas chaves de roteamento, verifica as tabelas de registro e decide como enviar as mensagens para uma ou mais caixas de mensagens. No caso do correio eletrônico as chaves de roteamento são os campos de destinatário e cópias (To, Cc e Bcc).
5. Um *binding* corresponde a uma entrada nas tabelas de roteamento do MTA.

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 2 mostra o envio e recebimento de uma mensagem.

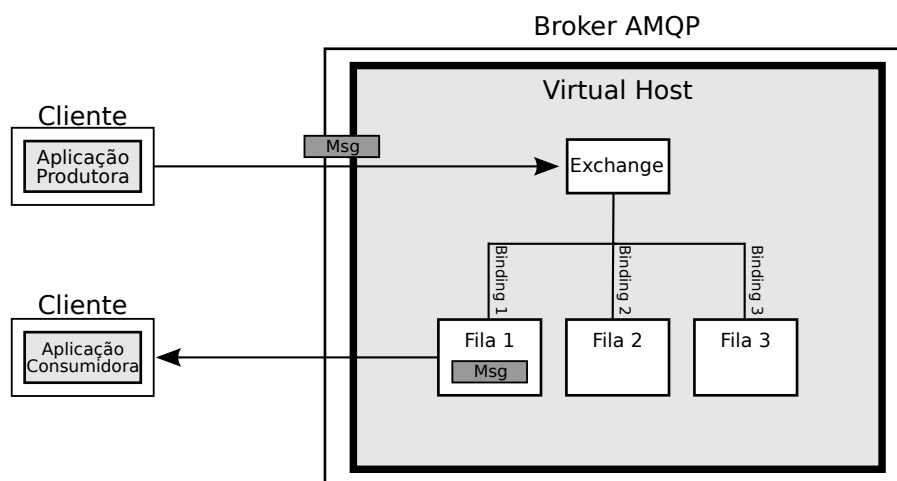


Figura 2: Visão geral do fluxo de uma mensagem desde o produtor até o seu consumidor.

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.

3.2 Atores no projeto Akka

Apesar do modelo de atores ser bem definido, não existe uma padronização para a sua implementação. De acordo com o modelo, o processamento das mensagens deve ser desacoplado do envio.

Os atores do Akka podem ser locais ou 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 sub-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.

3.2.1 Atores locais

A criação de atores locais pode acontecer tanto por meio uma sub-classe de `Actor` provendo uma implementação para o método `receive`, como mostrado na listagem 3.1, quanto via chamadas ao método `actor` da própria classe `Actor`, como mostrado na listagem 3.2. Note que no segundo caso o ator criado é uma instância de uma classe anônima, e que não é necessário chamar explicitamente o método `start` para inicializá-lo.

Listagem 3.1 Criação e uso da classe `MyActor`.

```
1      import se.scalablesolutions.akka.actor.Actor._
2
3      class MyActor extends Actor {
4          def receive = {
5              case "test" => println("received test")
6              case      _ => println("received unknown message")
7          }
8      }
9
10     val myActor = actorOf[MyActor].start
11     myActor ! "test"
```

Uma vez que uma mensagem foi enviada para um ator, ela é colocada sincronamente na fila de mensagens do ator, em tempo $O(1)$. As mensagens enfileiradas são então despachadas assincronamente para a função parcial definida no bloco `receive`, como mostrado na figura 3. Por padrão, atores são criados com um despachador que é impulsionado por eventos e que utiliza um *thread pool* para despachar as mensagens. Cada vez que uma mensagem é adicionada na fila de um ator, uma tarefa de despacho é criada e colocada na fila de processamento do *thread pool*. É

Listagem 3.2 Criação e uso de ator anônimo.

```

1      import se.scalablesolutions.akka.actor.Actor._
2
3      val myActor = actor {
4          case "test" => println("received_test")
5          case      _ => println("received_unknown_message")
6      }
7
8      myActor ! "test"

```

possível a definição e utilização de outros tipos de despachadores para se obter melhores resultados em casos específicos [3].

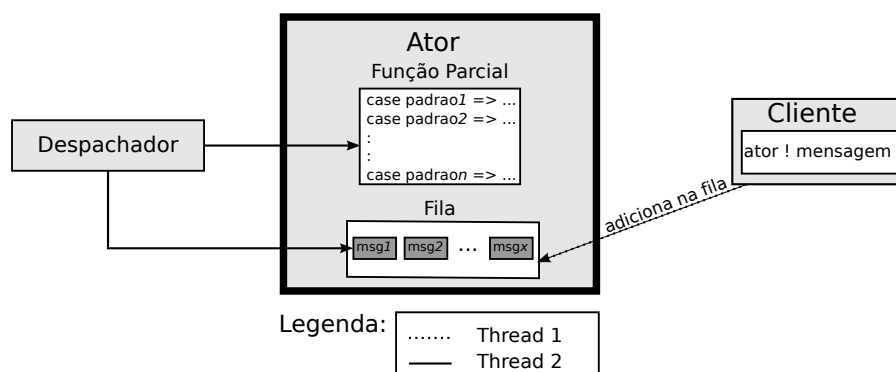


Figura 3: Envio e despacho de mensagens para atores locais.

3.2.2 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. A infra-estrutura de atores remotos utiliza os seguintes elementos:

- **RemoteActor**: Sub-classe de **Actor** que torna conveniente a criação de um ator remoto, ao invés de se criar explicitamente um ator local e torná-lo remoto.
- **RemoteServer**: É um componente usado no lado do servidor. Tem como responsabilidade manter registrados os atores, bem como encaminhar a eles as mensagens recebidas de clientes remotos. Cada **RemoteServer** é associado a um *host* e a uma porta TCP. Um mesma máquina virtual pode conter múltiplos **RemoteServers**.

- **RemoteNode**: Objeto *singleton* que é instância de **RemoteServer**. Não possui nenhum comportamento adicional e é usado quando se deseja ter um único **RemoteServer** numa dada máquina virtual.
- **Cluster**: Abstração para se agrupar instâncias de **RemoteServer**. Permite que um cliente interaja com múltiplos **RemoteServers**, por exemplo via *broadcasts* de mensagens;
- **RemoteClient**: É um componente usado pelo cliente. Tem 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 visíveis para o usuário, tais como: (i) serialização de mensagens; (ii) envio de mensagens para atores remotos; (iii) conversão de ator local em ator remoto; (iv) intermediação de mensagens de resposta vindas do **RemoteServer**, no caso envios via `!!` ou `!!!`.

A figura 4 mostra os elementos acima descritos e os relacionamentos entre esses elementos. A criação de atores remotos pelo cliente é análoga a de atores locais, salvo o fato de se informar o *host* e porta de um **RemoteServer** já em execução. A listagem 3.3 mostra a criação por meio de uma sub-classe de **RemoteActor**. Vale destacar que, nesta listagem, o valor de `myRemoteActor` (linha 12) é um *proxy* local para

o ator remoto. Outra possibilidade é criar um ator local e posteriormente convertê-lo em remoto, por meio de uma chamada ao método `makeRemote(host, port)` (listagem 3.4). Tal chamada deve ocorrer anteriormente à ativação do ator via chamada ao método `start`.

A criação de atores remotos pelo servidor, por sua vez, é exatamente igual a de atores locais, mostram as listagens 3.1 e 3.2. Depois de criar o ator localmente, basta registrá-lo em um **RemoteServer** ou **RemoteNode**, como mostra a listagem 3.5. A maneira com que o cliente remoto busca pelo ator e envia uma mensagem pode ser vista na listagem 3.6.

Quando um cliente envia uma mensagem a um ator remoto, o *proxy* local embrulha a mensagem, adicionando a ela informações de cabeçalho necessárias para o envio. O *proxy* usa um **RemoteClient** para enviar a mensagem ao **RemoteServer** correspondente. Este processo de envio, do ponto de vista do cliente leva tempo $O(1)$, já que a serialização da mensagem é feita de modo assíncrono.

A figura 5 mostra o caminho que a mensagem faz saindo da aplicação cliente via *proxy* e sendo repassada para um **RemoteClient**, que por sua vez faz a serialização assíncrona da mensagem e a envia para o **RemoteServer** correspondente. Uma vez que a mensagem tenha sido recebida pelo *handler* plugado ao JBoss Netty no lado servidor, o *handler* examina as informações de cabeçalho, localiza

Listagem 3.3 Criação e uso da classe `MyRemoteActor`.

```
1      import se.scalablesolutions.akka.actor._
2      import se.scalablesolutions.akka.actor.Actor._
3      import se.scalablesolutions.akka.remote._
4
5      class MyRemoteActor extends RemoteActor("localhost",15092) {
6          def receive = {
7              case "Hi" => println("hello")
8              case _ => println("received_unknown_message")
9          }
10     }
11
12     val myRemoteActor = actorOf[MyRemoteActor].start
13     myRemoteActor ! "Hi"
```

Listagem 3.4 Uso de `makeRemote` em atores locais.

```
1      import se.scalablesolutions.akka.actor.Actor._
2
3      class MyActor extends Actor {
4          def receive = {
5              case "test" => println("received_test")
6              case _ => println("received_unknown_message")
7          }
8     }
9
10     val myActor = actorOf[MyActor]
11     myActor.makeRemote("localhost", 15092)
12     myActor.start
13     myActor ! "test"
```

Listagem 3.5 Registro de um ator local em um nó.

```
1      RemoteNode.start("localhost", 15092)
2      RemoteNode.register("my-actor", myActor)
```

Listagem 3.6 Cliente acessando o ator remoto `myActor`.

```
1      val myActor = RemoteClient.actorFor("my-actor", "localhost", 15092)
2      myActor ! "Hi"
```

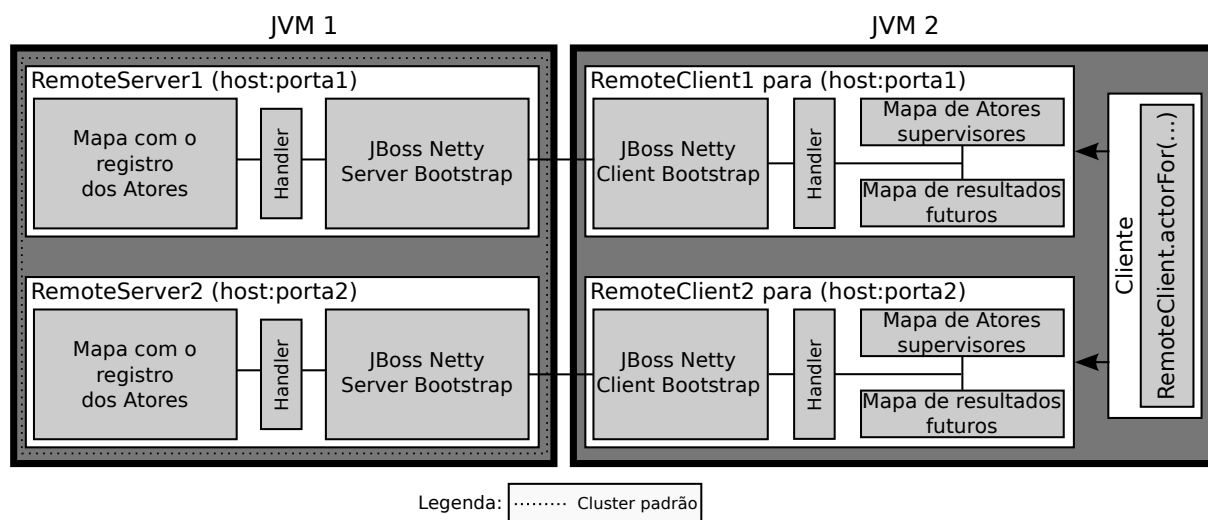


Figura 4: Disposição dos elementos da infra-estrutura de atores remotos.

o ator destinatário no registro local de atores e encaminha a mensagem a caixa de mensagens do ator, como se o envio fosse local. O despachador associado ao ator faz o tratamento mostrado na figura 3, já que do ponto de vista do despachador não há diferença entre um envio local e um remoto.

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 contrário, indo do `RemoteServer` para o `RemoteClient`, onde analogamente o *handler* faz o repasse da resposta para quem a estiver aguardando.

4 O projeto

Criaremos uma versão modificada do Akka, na qual o envio de mensagens para atores remotos seja feito via AMQP. O mecanismo de transporte atualmente utilizado pelo Akka é o JBoss Netty, baseado em *sockets*. Nossa proposta é substituir esse mecanismo de transporte por um *message broker* AMQP. Isso implica na substituição de todos os componentes associados ao JBoss Netty (*bootstraps* e *handlers*).

Optamos por basear nosso projeto na implementação de atores do Akka, ao invés da implementação presente na distribuição de Scala, pois a implementação do Akka é mais abrangente e, em nossa avaliação, mais robusta. Sobre a escolha do *broker* AMQP, consideramos como opções RabbitMQ [18] e Apache Qpid [7]. Como a especificação AMQP não define uma API para as

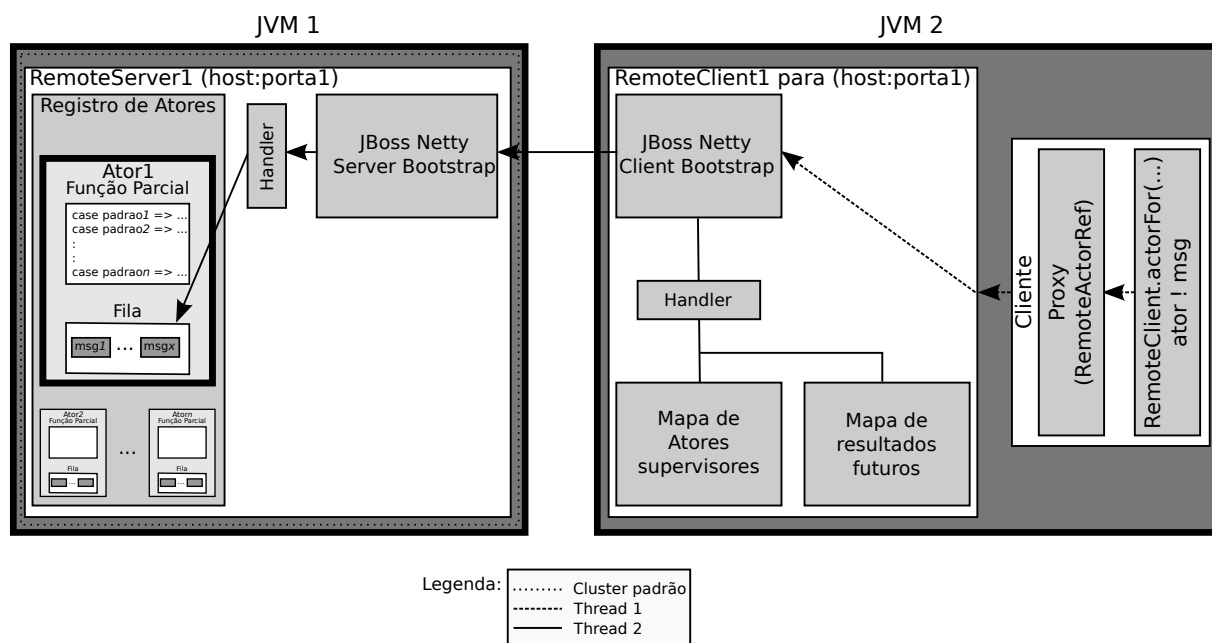


Figura 5: Fluxo de envio de mensagens para atores remotos.

aplicações clientes, o código a ser desenvolvido será específico para um desses *message brokers*.

5 Próximas etapas

5.1 Atividades programadas

Apresentamos a seguir uma lista com as próximas atividades:

1. análise dos *brokers* AMQP para definição do que será usado;
2. análise de desenvolvimento do código necessário para o transporte das mensagens para atores remotos utilizando AMQP;
3. desenvolvimento de testes, refatoração e aprimoramento da implementação;
4. elaboração e execução de cenários de experimento para comparação entre as implementações de atores remotos com JBoss Netty e AMQP;
5. redação da dissertação;
6. redação de artigo;

7. defesa.

5.2 Cronograma

O cronograma para as próximas atividades:

2010/2011									
	nov	dez	jan	fev	mar	abr	mai	jun	jul
1	•								
2		•	•	•					
3			•	•	•				
4					•	•			
5					•	•	•	•	
6								•	•
7									•

Referências

- [1] Apache ActiveMQ. <http://activemq.apache.org/>. Visitado em 30 de Outubro de 2010.
- [2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.
- [3] Akka, Dispatchers. <http://doc.akkasource.org/dispatchers>. Visitado em 19 de Maio de 2010.
- [4] Akka, Simpler Scalability, Fault-Tolerance, Concurrency & Remoting through Actors. <http://www.akkasource.org>. Visitado em 24 de Abril de 2010.
- [5] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.
- [6] Advanced Message Queueing Protocol. <http://www.amqp.org>. Visitado em 27 de Abril de 2010.
- [7] Apache QPid. <http://qp.id.apache.org/>. Visitado em 28 de Maio de 2010.
- [8] Joe Armstrong. Erlang - a survey of the language and its industrial applications. In *INAP'96 - The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, 1996.
- [9] Joe Armstrong. The development of erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, New York, NY, USA, 1997. ACM.
- [10] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.
- [11] Protocol Buffers – Google’s data interchange format. <http://code.google.com/p/protobuf>. Visitado em 14 de Setembro de 2010.
- [12] IBM Websphere MQ - Software. <http://www-01.ibm.com/software/integration/wmq/>. Visitado em 30 de Outubro de 2010.
- [13] JBoss Messaging. <http://www.jboss.org/jbossmessaging>. Visitado em 30 de Outubro de 2010.
- [14] Netty - the Java NIO Client Server Socket Framework. <http://www.jboss.org/netty>. Visitado em 27 de Abril de 2010.
- [15] Simon P. Jones. *Beautiful Concurrency*, chapter 24. O’Reilly Media, Inc., 2007.

- [16] Introducing JSON. <http://www.json.org/>. Visitado em 14 de Setembro de 2010.
- [17] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stephane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.
- [18] RabbitMQ - Messaging that just works. <http://www.rabbitmq.com/>. Visitado em 27 de Abril de 2010.
- [19] Library for describing binary formats for Scala types. <https://github.com/DRMacIver/sbinary>. Visitado em 14 de Setembro de 2010.
- [20] Nir Shavit and Dan Touitou. Software transactional memory, 1995.
- [21] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3):202 – 210, 2005.
- [22] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.