|  |
| --- |
| Universidade do Estado do Rio de Janeiro  Instituto de Matemática e Estatística  Departamento de Informática e Ciência da Computação |
| Ecurrencies API |
| Uma abordagem assíncrona na implementação de um serviço de automação de pagamentos |
|  |
| **Rafael Brandão de Andrade** |
|  |

|  |
| --- |
| Rio de Janeiro  março de 2013 |

**Ecurrencies API**

Rafael Brandão de Andrade

Monografia submetida ao corpo docente do Instituto de Matemática e Estatística da Universidade do Estado do Rio de Janeiro - UERJ, como parte dos requisitos necessários à obtenção do grau de Bacharel em Informática e Tecnologia da Informação.

Banca Examinadora:

Profo : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Nome Completo do Orientador - Orientador

Professor(a) IME/UERJ

Profa: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Rosa Maria Costa

Professora IME/UERJ

Profa: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Maria Clicia Stelling de Castro

Professora IME/UERJ

Rio de Janeiro, 11 de março de 2013.

Resumo

Atualmente existem dezenas de serviços de pagamento realizados via Internet. Os mais conhecidos no Brasil são *PagSeguro[[1]](#endnote-1)*, o mais usado no país, e *PayPal[[2]](#endnote-2)*, o mais usado no mundo. Cada um destes serviços funciona como uma verdadeira conta bancária online, em que transferências de dinheiro ocorrem com a velocidade de um clique, sem as burocracias que existem nas tradicionais transferências bancárias. Muitos sites de e-commerce já aceitam pagamentos realizados através de algum serviço de pagamento digital.

Estes oferecem ferramentas de automação na forma de API’s, que permitem a realização de pagamentos em massa, consulta de saldo e extratos, que são úteis para se integrar a sistemas de pagamentos já existentes em uma empresa.

Um dos problemas enfrentados é o de não haver um padrão reconhecido internacionalmente para a construção dessas API’s, o que pode levar a dificuldades de integração com esse tipo de serviço.

O objetivo deste trabalho é apresentar uma arquitetura possa abstrair as implementações desses serviços de automação.

Sumário

[Introdução 1](#_Toc350760710)

[Conceitos Básicos 2](#_Toc350760711)

[Formas de interação com sites de pagamento digital 2](#_Toc350760714)

[Direta 2](#_Toc350760715)

[Shopping Cart Interface (SCI) 2](#_Toc350760716)

[Application Program Interface (API) 2](#_Toc350760717)

[Liberty Reserve API 3](#_Toc350760718)

[Operações 3](#_Toc350760719)

[Preparação 3](#_Toc350760720)

[Tecnologias suportadas 5](#_Toc350760721)

[Execução 5](#_Toc350760722)

[Arquitetura Proposta 7](#_Toc350760723)

[O agente MOM 7](#_Toc350760724)

[O protocolo *AMQP* 8](#_Toc350760725)

[Conexão com o broker 8](#_Toc350760726)

[Estrutura de uma mensagem AMQP 9](#_Toc350760727)

[Roteamento de Mensagens 10](#_Toc350760728)

[Exchanges 10](#_Toc350760729)

[Filas 12](#_Toc350760730)

[Bindings 12](#_Toc350760731)

[A estrutura do projeto 12](#_Toc350760732)

[Formato da mensagem 14](#_Toc350760733)

[Google Protocol Buffers 14](#_Toc350760734)

[Criação de um sistema concorrente 16](#_Toc350760735)

[O modelo de atores 16](#_Toc350760736)

[Detalhes de Implementação 19](#_Toc350760737)

[Preparação inicial 19](#_Toc350760738)

[Linguagem Scala 19](#_Toc350760739)

[Google Protocol Buffers 19](#_Toc350760740)

[Gradle 20](#_Toc350760741)

[Broker AMQP 20](#_Toc350760742)

[O toolkit Akka 21](#_Toc350760743)

[Hierarquia de atores 21](#_Toc350760744)

[Localidade transparente 23](#_Toc350760745)

[Ecurrency Domain 25](#_Toc350760746)

[Liberty Reserve Domain 25](#_Toc350760747)

[Provedor do Sistema 25](#_Toc350760748)

[AMQP Service 25](#_Toc350760749)

[Módulo API 28](#_Toc350760750)

[Módulo Akka Commons 29](#_Toc350760751)

[Ecurrency Service 29](#_Toc350760752)

[Módulo Liberty Reserve Service 30](#_Toc350760753)

[O módulo cliente 32](#_Toc350760754)

[Avaliação de Desempenho 34](#_Toc350760755)

[Conclusões 35](#_Toc350760756)

[Bibliografia 36](#_Toc350760757)

Lista de figuras

[Figura 1 - Formulário de criação e edição de uma API 4](#_Toc350719922)

[Figura 2 - Diagrama de casos de uso do sistema 5](#_Toc350719923)

[Figura 3 - Diagrama de atividades que define a execução de uma operação na API 6](#_Toc350719924)

[Figura 4 - Diagrama de atividades adequado à arquitetura proposta 8](#_Toc350719925)

[Figura 5 – Conexão AMQP e seus canais privados 9](file:///D:\rafael\workspace\ecurrencies\Monography.docx#_Toc350719926)

[Figura 6 - Modelo de roteamento de mensagens do protocolo AMQP versão 0.9.1 10](#_Toc350719927)

[Figura 7 – Exemplo de uso de um *topic exchange* 11](#_Toc350719928)

[Figura 8 – Diagrama que representa a iteração entre cliente, broker e provedor 13](file:///D:\rafael\workspace\ecurrencies\Monography.docx#_Toc350719929)

[Figura 9 - Parte do diagrama de classes que representa uma transferência 15](#_Toc350719930)

[Figura 10 – Representação de um ator e sua fila de mensagens 18](file:///D:\rafael\workspace\ecurrencies\Monography.docx#_Toc350719931)

[Figura 11 - Hierarquia e supervisão de atores 22](#_Toc350719932)

[Figura 12 – Hierarquia inicial do sistema de atores 23](file:///D:\rafael\workspace\ecurrencies\Monography.docx#_Toc350719933)

[Figura 13 - Localização de atores por meio de URL’s 24](#_Toc350719934)

[Figura 14 – Diagrama de atividades que representa o funcionamento do módulo *AMQP Service* 27](file:///D:\rafael\workspace\ecurrencies\Monography.docx#_Toc350719935)

[Figura 15 – Diagrama de atividades que representa o serviço de requisição do nome de uma conta Liberty Reserve 30](#_Toc350719936)

Lista de códigos

[Código 1 – Parte do arquivo *proto* criado para gerar as classes de domínio para a e-currency Liberty Reserve 15](#_Toc350719937)

[Código 2 – Método do ator *AmqpConsumer* que recebe e despacha uma mensagem *AMQP* 28](#_Toc350719938)

[Código 3 – Ator que converte um array de bytes em um objeto de domínio 29](#_Toc350719939)

[Código 4 – Classe HttpService, generalização da comunicação com o servidor da API Liberty Reserve 31](#_Toc350719940)

Lista de Siglas e Símbolos

AMQP *Advanced Message Queuing Protocol*  
API *Application Program Interface*   
HTML *HyperText Markup Language*HTTP *HyperText Transport Protocol*HTTPS *HyperText Transport Protocol Secure*IBAN *International Bank Account Number*IP *Internet Protocol*ISO *International Organization for Standardization*JMS *Java Messaging Service*  
JSON *JavaScript Object Notation*JVM *Java Virtual Machine*MOM *Message Oriented Middleware*NAPT *Network Address and Port Translation*NAT *Network Address Translation*NVP *Name Value Pair*OSI *Open Standards Interconnection*RPC *Remote Procedure Call*SASL *Simple Authentication and Security Layer*  
SCI *Shopping Cart Interface*SOA *Service Oriented Arquitecture*SOAP *Simple Object Access Protocol*STM *Software Transaction Memory*SWIFT *Society for Worldwide Interbank Financial Telecommunication*TCP *Transmission Control Protocol*TLS *Transport Layer Security*   
URI *Uniform Resource Identifier*  
URL *Uniform Resource Locator*  
XML *Extensible Markup Language*

Capítulo 1

Introdução

Ao longo dos anos 90 a Internet possibilitou uma rápida expansão para empresas de e-commerce, removendo fronteiras, permitindo qualquer um a realizar uma compra diretamente do conforto de sua casa. Bastava apenas um cartão de crédito na hora de pagar e um endereço para a entrega de algum produto. Porém, um dos problemas enfrentados nessa época era com a segurança de dados críticos na hora de se realizar esse tipo de transação, houve diversos casos de fraudes e roubos realizados por pessoas mal intencionadas, justamente porque não havia controles de segurança adequados, que permitissem uma integração segura entre estes sites de e-commerce às empresas de cartão de crédito.

Isso permitiu o surgimento de um novo tipo de serviço de pagamento especializado para que não ocorram fraudes, em que as transações não podem ser rastreadas, dando mais confiança a quem realiza pagamentos pela Internet. Cada cliente de um desses serviços recebe um número de uma conta, que pode ser operada online, através do seu respectivo site ou pode manipulada por meio de API’s.

Uma das cinco maiores empresas em número de clientes que fornecem esses serviços é a *Liberty Reserve[[3]](#endnote-3)*. Este trabalho irá implementar um cliente para a *API[[4]](#endnote-4)* que ela disponibiliza, de uma forma que componentes de software que já tenham sido criados ou utilizados nessa implementação possam ser reutilizados nas implementações de outros serviços semelhantes (por exemplo, ao se implementar a API do serviço PayPal).

Ele será organizado da seguinte forma. O capítulo 2 abordará os conceitos básicos sobre cada serviço oferecido por esta API. O capítulo 3 discutirá sobre a arquitetura proposta, bem como a organização de alguns componentes e porque ela é capaz de abstrair a implementação desse tipo de serviço de automação em um projeto que permitirá a adição de outros serviços similares, prestados por outras empresas. O capítulo 4 mostrará os detalhes de implementação do sistema que será o cliente, como as tecnologias utilizadas na sua construção. O capítulo 5 irá comparar esta implementação com as implementações já oferecidas em diversas tecnologias pela própria empresa *Liberty Reserve*. O capítulo 6 finalizará com as conclusões tiradas ao longo da elaboração deste trabalho.

Capítulo 2

Conceitos Básicos



Formas de interação com sites de pagamento digital

Direta

Acontece através do próprio site, como um serviço de internet banking tradicional, em que o cliente deve fazer um login seguro para poder realizar operações em sua conta.

Shopping Cart Interface (SCI)

Esta é a forma que sites de e-commerce usam para poder oferecer a seus clientes um meio seguro de pagamento através de serviços de pagamento digital. Estes serviços oferecem botões e formulários HTML personalizados para que os sites de e-commerce possam redirecionar seus clientes para pagarem por alguma compra efetuada.

Esta também é uma forma de interação direta, em que o cliente deve passar por controles de segurança adequados para provar sua identidade.

Application Program Interface (API)

API’s fornecem um meio de automação de acesso e operação de uma determinada conta, permitindo que o cliente não seja necessariamente uma pessoa física em frente a uma tela de computador.

As API’s normalmente são utilizadas na integração com um sistema de pagamentos já existente. Elas oferecem no mínimo três serviços:

* Serviço de transferência
* Serviço verificação de extrato (histórico de transações)
* Serviço de verificação se saldo

Liberty Reserve API

Operações

Além das operações fundamentais que uma API desse tipo de serviço deve realizar (transferência, verificação de extrato e verificação de saldo), a API da empresa Liberty Reserve possui mais dois:

* Serviço de verificação de nome de conta
* Serviço de verificação sobre transação efetuada

Preparação

De acordo com o manual (Haller, Procopec, Miller, Klang, Kuhn, & Jovanovic, 2012) que o site dispõe para consulta, há uma série de passos que devem ser seguidos para fazer um determinado sistema interagir com uma determinada conta:

* Criar uma API e habilitá-la. Por questões de segurança, o acesso realizado por API é desabilitado por padrão.
* Criar uma chave de segurança para esta API.
* Definir quais operações a API criada pode executar.

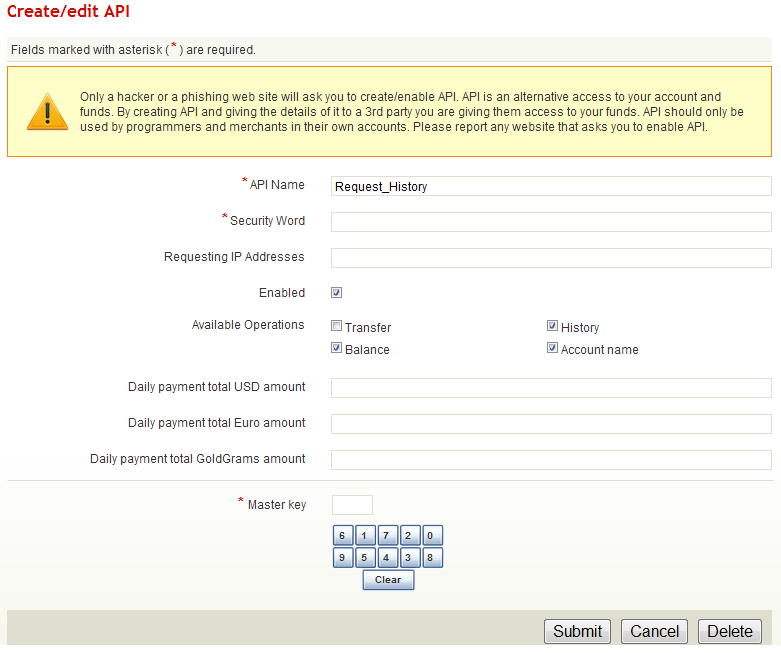


Figura - Formulário de criação e edição de uma API

De acordo com o site, quando o sistema estiver em um ambiente de produção é recomendado usar um servidor com IP fixo e fazer com que somente esse IP possa acessar a API criada, preenchendo o campo *“Requesting IP Adresses”* do formulário de criação de API’s. Mas através da técnica de *NAT / NAPT* é possível colocar vários servidores com endereços IP privados e utilizar somente um endereço IP público e fixo.

Das operações suportadas derivamos o seguinte diagrama de casos de uso:

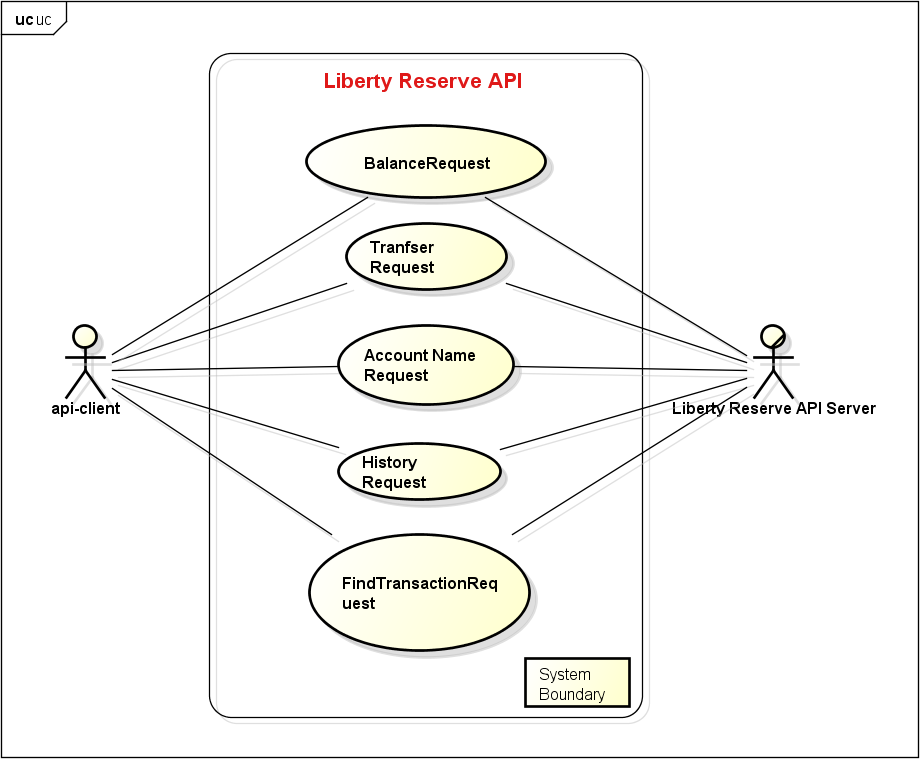


Figura - Diagrama de casos de uso do sistema

Tecnologias suportadas

Para cada uma das cinco operações disponibilizadas pela API, é necessária a elaboração de uma mensagem em um dos formatos suportados. A API oferece grande flexibilidade na integração, pois são suportados os seguintes formatos:

* XML
* JSON
* NVP
* SOAP

O manual disponível online no site da empresa mostra como deve ser feita a elaboração das mensagens para cada um dos formatos listados.

Execução

Para se acessar uma API devem-se seguir os seguintes passos:

* Elaborar uma mensagem no formato escolhido.
* Fazer uma requisição HTTPS (GET ou POST) ao servidor.
* Tratar a resposta do servidor.

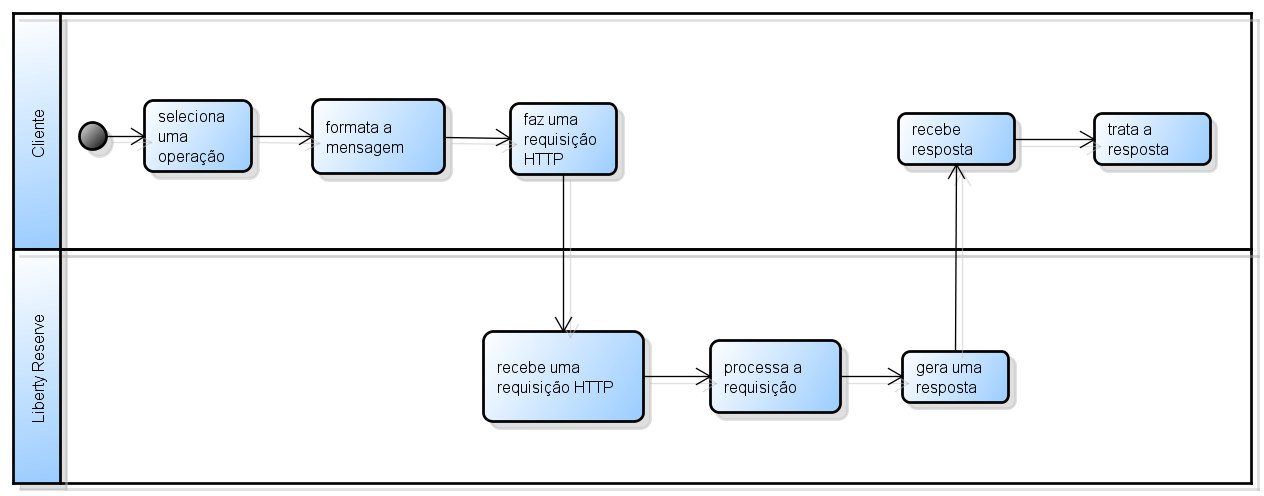


Figura - Diagrama de atividades que define a execução de uma operação na API

Capítulo 3

Arquitetura Proposta

A intenção deste projeto é criar um sistema que sirva de cliente da API criada pela empresa Liberty Reserve a fim de automatizar serviços fornecidos por ela, porém, de uma forma que permita a adição de novas *ecurrencies* [[5]](#endnote-5), sem aumentar a complexidade do projeto.

O sistema a ser desenvolvido deve ser um sistema modular, orientado a componentes, em que cada módulo possui uma função ou propósito bem definido, e que ao mesmo tempo seja capaz de atender múltiplas requisições de diferentes clientes.

A chave para isso está na criação de um sistema modular, orientado a mensagens. O cliente passa a ser o produtor de mensagens específicas ao domínio da *ecurrency*. Com isso ele só precisa se preocupar na formatação das mensagens e no envio a um sistema de filas (agente *MOM*).

O agente MOM

O agente orientado a mensagens é um sistema de filas. Como parte da arquitetura, optou-se pelo protocolo *AMQP*[[6]](#endnote-6) (protocolo avançado de orientação a mensagens). *AMQP* é baseado na API do *JMS*, mas em nível de protocolo, ou seja, em tese qualquer cliente *AMQP* pode se comunicar com qualquer broker *AMQP* (o sistema de filas), independente da plataforma ou linguagem usada.

O sistema de filas roteará as mensagens a consumidores específicos, e estes serão os responsáveis pela transformação dessas mensagens em um formato reconhecido pela API e também pela comunicação com esta API.

Isso denota um fraco acoplamento entre o produtor e o consumidor, permitindo o uso de tecnologias distintas no consumidor e no produtor, tornando mais fácil a integração do cliente com sistemas legados, visto que há apenas a necessidade de se preocupar com o formato da mensagem e com a comunicação com um sistema de filas.

No caso de adição de novas *ecurrencies* ao sistema, para cada *ecurrency* adicionada corresponderá um novo formato de mensagem, ou seja, um novo domínio.

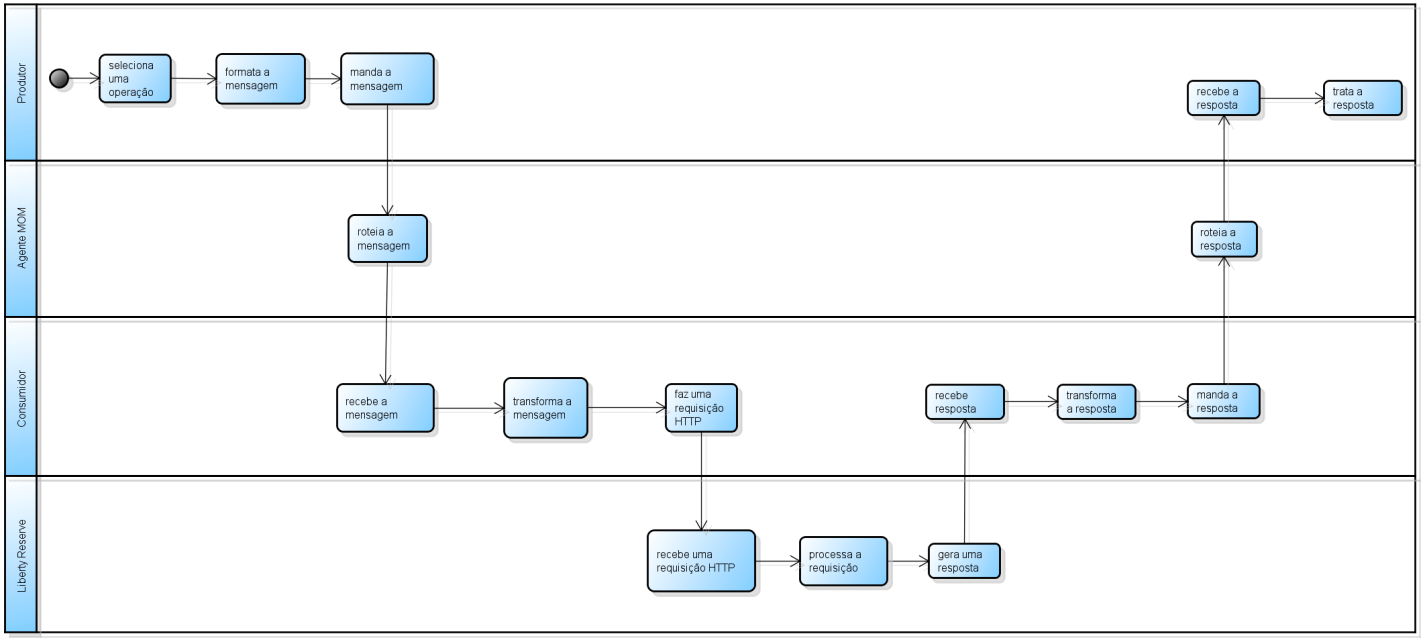


Figura - Diagrama de atividades adequado à arquitetura proposta

O protocolo *AMQP*

O protocolo *AMQP* define o formato da mensagem que deve ser codificada em um *stream* de bytes para ser mandado pela rede. Então qualquer cliente ou *broker* que possa criar ou interpretar mensagens que se adequem a esse formato pode se comunicar com qualquer outro cliente ou *broker* *AMQP*, independente da linguagem e/ou sistema operacional usado em sua implementação.

Ele é um protocolo binário, que atua na camada de aplicação do modelo *OSI*, elaborado para suportar de forma eficiente uma gama de aplicações orientadas a mensagens. Ele provê controle de fluxo, comunicação orientada a mensagens e garantias de entrega, como “*at least once”*, “*at most once”* e *“exactly once”*. Também provê mecanismos de autenticação e encriptação utilizando *SASL* e *TLS*.

Conexão com o broker

Antes de publicar ou consumir mensagens, um cliente *AMQP* deve conectar-se ao *broker* através de uma conexão TCP. Como *brokers* *AMQP* devem receber conexões de muitos clientes ao mesmo tempo, é de extrema importância criar apenas uma conexão por cliente. Para que cada cliente possa usar a mesma conexão em um ambiente com múltiplas *threads* foi criado o conceito de *channel*.

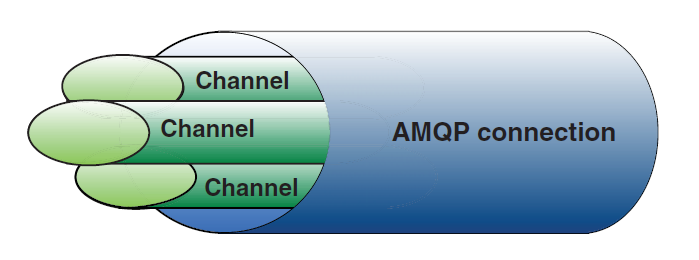
*Channels* são canais virtuais dentro de uma conexão TCP com o *broker*. Cada um recebe o seu próprio canal de comunicação com o *broker* e deve ser utilizado por somente uma thread.

Figura – Conexão AMQP e seus canais privados

Estrutura de uma mensagem *AMQP*

Mensagens no modelo *AMQP* possuem atributos (propriedades e cabeçalho) e *payload* (a mensagem em si, em um array de bytes). Propriedades são metadados já previstos pela especificação 0.9.1, bem como:

* Conteúdo da mensagem
* Codificação
* *Routing Key*
* Modo de Entrega
* Prioridade
* Momento da entrega (*timestamp*)
* Período de expiração da mensagem
* Identificação do produtor
* Cabeçalho

O cabeçalho oferece ao programador a criação de metainformações da mensagem não previstas no protocolo ou próprias da aplicação.

Roteamento de Mensagens

A imagem abaixo ilustra o protocolo na versão 0.9.1 em seu mais alto nível de abstração:

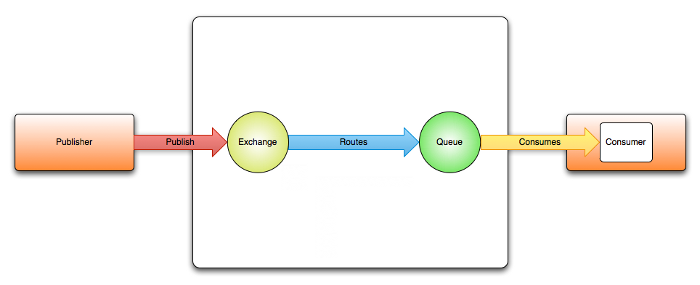


Figura - Modelo de roteamento de mensagens do protocolo AMQP versão 0.9.1

O agente responsável pelo sistema de filas é o *broker* *AMQP*. Ele oferece três entidades que devem ser manipuladas pelos clientes, tanto produtores quanto consumidores, para que estes possam enviar ou receber mensagens. São as filas, os *exchanges* e as *binding keys*.

Exchanges

São as entidades para as quais os produtores devem enviar as mensagens. Elas roteiam as mensagens para zero ou mais filas. O algoritmo de roteamento varia de acordo com o tipo de *exchange* criado e com um parâmetro também passado pelo produtor, chamado de *routing key*. Os tipos são:

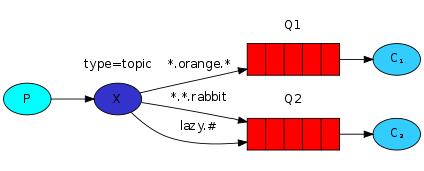
1. *Direto* – Redireciona as mensagens baseadas na *routing key* utilizada para a publicação.
   1. Uma fila se vincula a um *exchange* direto com uma *binding key* **K** ;
   2. Quando uma nova mensagem com uma *routing key* **R** chega a um *exchange* direto, ele entrega a mensagem à esta fila se **K = R**.
2. *Fanout* – Redireciona uma cópia de qualquer mensagem que chega a todas às filas vinculadas a este tipo de *exchange*, independente da *routing key* utilizada na publicação da mensagem.
3. *Topic* – Redireciona mensagens a uma ou mais filas baseado na *routing key* utilizada na publicação e no padrão utilizado para vincular a fila ao *exchange*. As *routing keys* utilizadas na publicação não podem ser arbitrárias, devem seguir um padrão de palavras separadas por pontos, como por exemplo *"stock.usd.nyse"*. Vejamos um exemplo:

Figura – Exemplo de uso de um *topic exchange*

Analisando a figura acima, vamos supor que os produtores usem como *routing keys* que consistem em três palavras seguidas por pontos: *velocidade*.*cor*.*animal*. A fila Q1 está vinculada ao *exchange* **X** pela *binding key “\*.orange,\*”*. A fila Q2 está vinculada ao exchange **X** com duas *binding keys*: *“\*.\*.rabbit”* e *“lazy.#”*.

O curinga *\** pode ser substituído por uma palavra e o curinga *#* pode ser substituído por zero ou mais palavras. A partir daí pode-se concluir que os consumidores da fila Q1 estão interessados em qualquer animal laranja e que os consumidores da fila Q2 estão interessados em qualquer animal que seja um coelho ou qualquer animal que seja preguiçoso.

1. *Cabeçalho* – As mensagens são roteadas a filas específicas de acordo com o conteúdo dos seus cabeçalhos. Este *exchange* é mais indicado para rotear mensagens que expressam melhor seus atributos em seus próprios cabeçalhos do que o uso de uma *routing key*.
2. *Padrão* – O *exchange* padrão é um *exchange* direto, representado por uma *string* vazia, previamente criado pelo *broker*. Qualquer fila declarada por algum cliente sempre estará vinculada ao *exchange* padrão pelo seu próprio nome como *binding key*. Então quando uma mensagem chega a este *exchange* com uma *routing key* **R**, ele entregará a mensagem à fila declarada com o nome **F** se **R = F**.

Filas

Assim como em qualquer outro sistema orientado a mensagens, as filas no modelo *AMQP* devem ser o ponto de armazenamento das mensagens para os consumidores da aplicação. Se mais de um consumidor estiver vinculado a uma fila, o *broker AMQP* automaticamente distribuirá as mensagens em um esquema *round robin*.

Bindings

*Bindings* são as regras que os *exchanges* usarão para rotear as mensagens às filas. Qualquer fila declarada já estará vinculada automaticamente ao *exchange* padrão pelo seu nome. Como já visto na seção sobre *exchanges*, as filas devem ser vinculadas por meio de *binding* keys a um ou mais *exchanges* para que possam receber mensagens.

A estrutura do projeto

Este projeto se dividirá em pelo menos dois sistemas distintos. Um será o provedor, a porção que consumirá mensagens da fila do broker *AMQP* e a delegará para a implementação apropriada. O outro será um cliente de exemplo para a aplicação, produtor das mensagens.

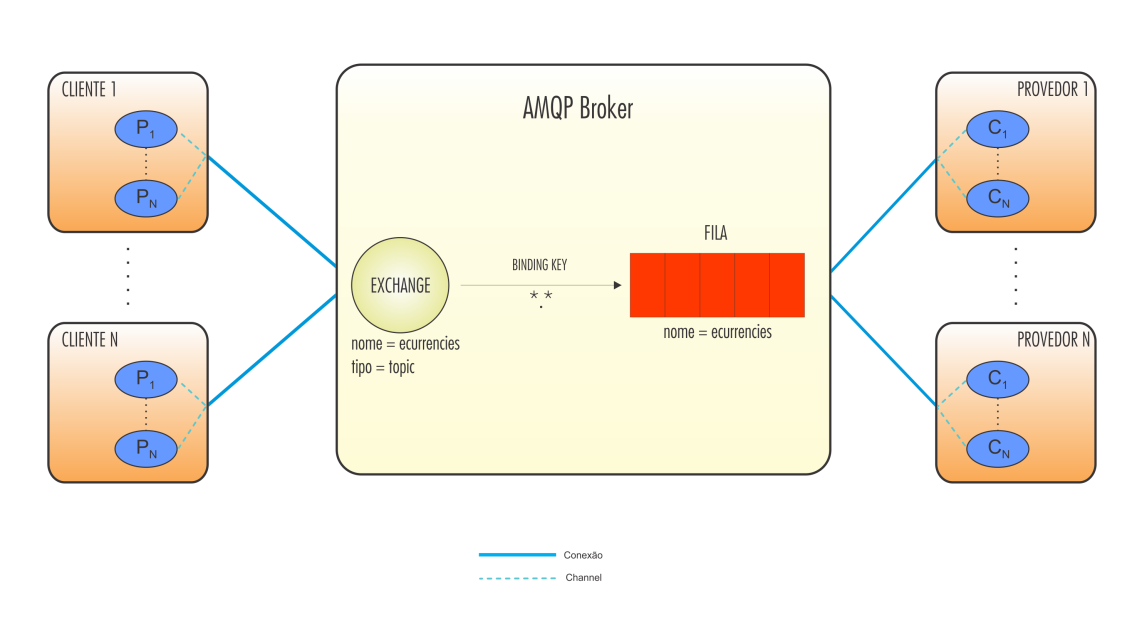


Figura – Diagrama que representa a iteração entre cliente, broker e provedor

1. Provedor é o responsável pela declaração da fila e do *exchange* no  *broker* *AMQP*, bem como realizar a ligação entre eles pela *binding key* *“\*.\*”;*
2. O cliente deve apenas saber para qual *exchange* deve mandar as mensagens;
3. Ao mandar uma mensagem para o *broker*, o cliente deve utilizar uma *routing key* que denote a identificação do provedor da *ecurrency* e a identificação do serviço requisitado: *“<ecurrency-id >.<service-id>”.*

Como a fila estará vinculada a este *exchange* com a *binding key* *“\*.\*”* ela receberá qualquer mensagem enviada com uma *routing key* que siga este padrão, como por exemplo *“liberty-reserve.transfer”*. Ao receber uma mensagem o consumidor redirecionará a mensagem à implementação do serviço que corresponde ao requisitado pela *routing key*, **permitindo que sejam disponibilizadas implementações de serviços de diversas *ecurrencies***.

No caso da API da empresa Liberty Reserve, todas as mensagens que se destinam ao processamento pela sua API deverão ser publicadas com uma *routing key* *“liberty-reserve.<service-id>”*, como por exemplo, *“liberty-reserve.transfer”*.

Formato da mensagem

Como o sistema possui flexibilidade para integrar diferentes plataformas e linguagens, é necessário adotar um formato serializável de mensagem neutro à linguagem ou à plataforma utilizada para a representação do domínio de cada *ecurrency* a ser implementada.

Google Protocol Buffers

Como não é possível utilizar a representação de mensagens com formatos de texto, como *XML* ou *JSON* sem serem serializados, optou-se pela utilização de *Protocol Buffers* [[7]](#endnote-7). Trata-se de um formato de dados estruturados, desenvolvido pelo Google para uso interno como protocolo *RPC*. Há algum tempo a empresa o disponibilizou como código aberto para a comunidade.

O diferencial é que esse formato gera automaticamente as classes que representam esses dados, assim como os métodos que serializam e desserializam os dados para um formato binário, mais enxuto que *XML* e *JSON*. Além disso, a serialização de formatos de texto como *XML* e *JSON*, exigida pelo protocolo *AMQP*, a acopla à linguagem utilizada no processo, fazendo com que produtor e consumidor tenham que ser implementados na mesma linguagem de programação.

Com *protocol buffers,* define-se como os dados devem ser estruturados em um arquivo de extensão *proto*, compila-se esse arquivo e automaticamente são geradas as classes e métodos de serialização das classes referentes ao domínio. A partir deste momento o termo *protocol buffers* será abreviado para *protobuffers*.

Oficialmente são suportados os compiladores para as linguagens Java, C++ e Python. Porém existem diversos projetos não oficiais que suportam compiladores para outras dezenas de linguagens.

Um diagrama de classes foi feito em cima de todo o domínio da API da *ecurrency* Liberty Reserve. Ele foi refinado para se adaptar ao uso de *protobuffers*.

Como o diagrama de classes para o domínio desta *ecurrency* é razoavelmente grande para ser exibido como uma imagem neste documento, destacou-se apenas um trecho dele para se mostrar como foi feito parte do arquivo *proto* que gerou estas classes na plataforma escolhida para a implementação.

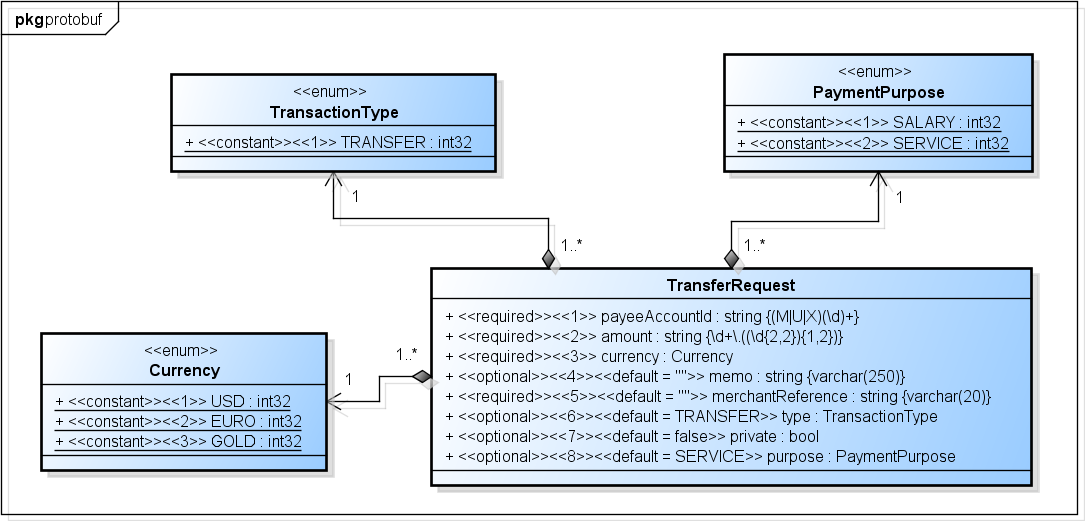


Figura - Parte do diagrama de classes que representa uma transferência

Do diagrama de classes, já refinado para o uso de *protobuffers,* foi criado o seguinte trecho do arquivo *proto*:

|  |
| --- |
| **enum** Currency {  **USD** = 1;  **EURO** = 2;  **GOLD** = 3;  }  **enum** PaymentPurpose {  **SALARY** = 1;  **SERVICE** = 2;  }  **enum** TransactionType {  **TRANSFER** = 1;  }  **message** TransferRequest {  **required string** payeeAccountId = 1;  **required string** amount = 2;  **required** Currency currency = 3;  **required string** memo = 4;  **required string** merchantReference = 5;  **optional** TransactionType type = 6 [**default** = TRANSFER];  **optional** **bool** private = 7 [**default** = false];  **optional** PaymentPurpose purpose = 8 [**default** = SERVICE];  } |

Código – Parte do arquivo *proto* criado para gerar as classes de domínio para a e-currency Liberty Reserve

Criação de um sistema concorrente

Como requisito, o sistema deve atender a múltiplas requisições de diferentes tipos de clientes. Indo mais além, nos dias de hoje qualquer computador vem equipado com processadores de múltiplos *cores*. É de extrema importância para um desenvolvedor de software saber lidar com hardware que verdadeiramente executa **mais do que uma tarefa ao mesmo tempo**. Para tal fim deve-se elaborar um sistema concorrente.

O modelo tradicional para a criação de sistemas concorrentes envolve o uso de memória compartilhada por múltiplas threads, no qual o programador deve lidar com a criação de semáforos e monitores para sincronizar este acesso compartilhado à memória e prevenir potenciais problemas derivados deste modelo como *deadlocks* e *race conditions*.

Lidar com o gerenciamento direto de threads é um processo de baixo nível e muito propenso a erros, que requer um nível de habilidade muito maior por parte do desenvolvedor, pois torna quase impossível a prevenção de todas as possibilidades de *deadlocks* e muitas vezes esses problemas se manifestam já com o sistema em produção.

Além disso, um problema inerente dos sistemas operacionais sobre threads que compartilham memória é a constante mudança de contexto para colocar uma nova thread em execução, o que gera mais *overhead* sobre o sistema.

O modelo de atores

Concebido em 1973 por Carl Hewitt, Peter Bishop e Richard Steiger ao publicarem *A Universal Modular Actor Formalism for Artificial Intelligence*, sendo posteriormente implementado na linguagem *Erlang[[8]](#endnote-8)*, uma linguagem criada em 1987 já com o propósito de criar sistemas concorrentes distribuídos, tolerantes a falhas, altamente escaláveis e de tempo real.

Munish K. Gupta o define como:

“The Actor Model takes a different approach to solving the problem of concurrency, by avoiding the issues caused by threads and locks. In the Actor Model, all objects are modeled as independent, computational entities that only respond to the messages received. There is no shared state between actors...”. (K. Gupta, 2012, p. 11)

“Actors change their state only when they receive a stimulus in the form of a message. So unlike the object-oriented world where the objects are executed sequentially, the actors execute concurrently”. (K. Gupta, 2012, p. 11)

Os objetos do modelo de atores (os atores) devem ser modelados como entidades computacionais independentes, que podem possuir estado interno e que respondem a estímulos na forma de mensagens. Não deve haver estado compartilhado entre essas entidades e as mensagens trocadas entre elas devem ser imutáveis.

Os seguintes princípios podem ser resumidos:

* A comunicação entre atores é feita através de mensagens imutáveis. Atores não compartilham estado (como variáveis globais ou referencies externas) e qualquer informação compartilhada é feita por troca de mensagens.
* Cada ator está vinculado a uma fila, para onde mensagens são enviadas. As mensagens são processadas pelo ator uma de cada vez. Um ator pode responder a uma mensagem mandando novas mensagens a outros atores, alterando seu estado interno (só ele deve fazer isso), criar novos atores e mudar de comportamento.
* Mensagens são passadas de forma assíncrona. O produtor não fica esperando uma resposta e pode retomar suas tarefas imediatamente.
* A comunicação entre o produtor e o consumidor é desacoplada. Isso permite que produtor e consumidor rodem em threads diferentes, provendo um modelo concorrente e escalável.

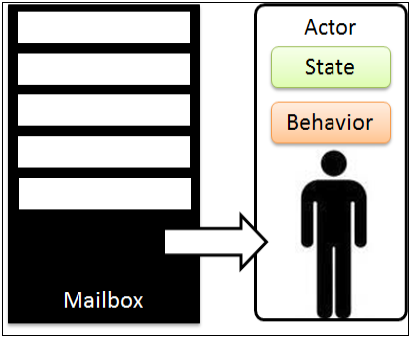


Figura – Representação de um ator e sua fila de mensagens

Capítulo 4

Detalhes de Implementação

Este capítulo irá expor os detalhes pertinentes às tecnologias escolhidas para a implementação do sistema de filas, o formato de mensagem adotado, assim como a disposição dos produtores e consumidores do sistema.

O repositório *git* que mantém todo o código fonte deste projeto e este documento encontra-se em <https://github.com/rafael-st-rj/ecurrencies>.

Preparação inicial

O ambiente de desenvolvimento deve ser preparado para a correta compilação e distribuição do projeto.

Linguagem Scala

Tanto provedor quanto cliente do projeto foi elaborado na linguagem de programação *Scala*, que roda sobre a plataforma *Java* e possui uma ótima integração com bibliotecas da linguagem *Java*. É necessário instalar e configurar corretamente o *Java Development Kit 6* ou superior, disponibilizado pela Oracle.

*Scala* é uma linguagem estática fortemente tipada. Sua inferência de tipos é bem superior ao de *Java*, permitindo um código mais limpo, sem construções desnecessárias. Também é uma linguagem híbrida, com suporte tanto à orientação a objetos e quanto à programação funcional. Valoriza a imutabilidade de todos os objetos criados, o que simplifica muito a programação concorrente, que será profundamente abordada neste projeto.

Google Protocol Buffers

O compilador *protobuffer* deve ser baixado do site do projeto e copiado para uma pasta na máquina. No caso de um sistema operacional \*Unix os seus fontes devem ser compilados, no caso de Windows ele já é disponibilizado como um arquivo executável.

Deve-se adicionar a pasta onde se encontra o arquivo executável do compilador à variável *path* do sistema operacional.

Gradle

Todo o projeto usa a ferramenta de automação de build de software *Gradle* para compilação, execução de testes, gerenciar dependências e release.

*Gradle* [[9]](#endnote-9)é uma ferramenta escrita na linguagem *Groovy* que gerencia builds automatizados de software e a busca de artefatos e bibliotecas. Seus plug-ins oficiais dão a ele uma gama de funções como compilação, realização testes de unidade, realização testes de integração, empacotamento e distribuição dos artefatos criados, conectar-se a repositórios de bibliotecas por diversos protocolos, relatórios de build e de testes detalhados, gerenciamento de releases, integração com ferramentas já conhecidas como *Ant* [[10]](#endnote-10) e *Maven* [[11]](#endnote-11), dentre muitas outras. Tudo isso é realizado em um pipeline de fases bem definidas. Além disso, existe a possibilidade de criação do seu próprio plug-in para se adequar a qualquer requisito de gerência de software.

Talvez o seu maior avanço na área de automação de build de software seja considerado o build incremental, em que ele só executa determinada fase do build se notar que algum arquivo relacionado tenha sido modificado. O compilador *Scala* usado no projeto também é capaz de realizar o mesmo, recompilando só o que foi modificado nos arquivos de código fonte, diminuindo bastante o tempo necessário para a execução da fase de compilação do sistema.

A sua instalação é simples. Além da plataforma Java JDK instalada, deve-se baixar a ferramenta do link <http://www.gradle.org/downloads>, descompactar para uma pasta e adicionar o endereço da pasta *bin* (dentro da pasta aonde a ferramenta foi descompactada) à variável *path* do sistema operacional.

Broker *AMQP*

A implementação de broker *AMQP* escolhida foi a RabbitMQ [[12]](#endnote-12), que talvez seja considerado o broker *AMQP* de código aberto mais usado do mundo, desenvolvido na linguagem *Erlang* e que implementa o protocolo *AMQP* versão 0.9.1. Ele disponibiliza clientes em diversas linguagens e plataformas, inclusive as que rodam diretamente em *front-ends*, como *JavaScript*.

O toolkit Akka

O toolkit *Akka*[[13]](#endnote-13) é uma implementação do modelo de atores na linguagem *Scala*. Oferece um ambiente para o desenvolvimento de aplicações concorrentes, orientadas a eventos, escaláveis tanto horizontal quanto verticalmente, distribuídas e tolerantes a falhas. Este é o ideal que qualquer aplicação ou sistema empresarial busca obter. *Akka* possui API’s para as linguagens *Java* e *Scala*.

*Akka* possui um modelo de atores que pode ser distribuído em diversas máquinas, de forma transparente para a aplicação, isto é, quando um ator manda uma mensagem para outro, para ele não faz diferença se este outro ator está ou não na mesma JVM.

Toda a aplicação desenvolvida neste projeto poderia ser feita usando-se somente *Akka*, mas limitaria as implementações dos sistemas às linguagens *Java* ou *Scala*, o que pode ser um fator não desejável nos requisitos da aplicação.

O sistema utilizará somente atores locais, tanto no cliente quanto no provedor. Qualquer interação externa será feita através do *broker AMQP*.

Serão apresentadas agora algumas características do *toolkit Akka*, que são de fundamental importância para o funcionamento do sistema proposto neste projeto.

Hierarquia de atores

Um conceito importante que *Akka* segue é o de hierarquia de atores. Um ator deve ser criado para executar apenas uma tarefa. Se esta tarefa for muito grande para ser executada por ele, ela deve ser dividida em partes menores, ao ponto que cada parte possa ser executada por um ator que esteja sob o seu controle. Cada ator tem exatamente um supervisor, o ator que o criou.

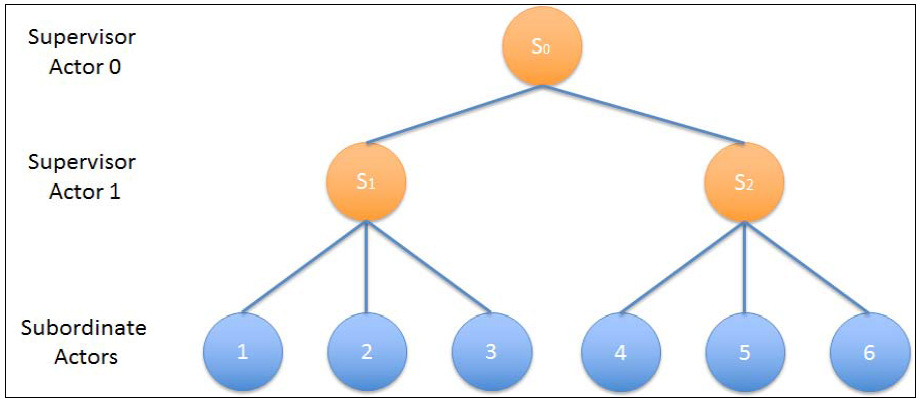


Figura - Hierarquia e supervisão de atores

A principal função do supervisor é tomar decisões sobre o que fazer com um ator sob o seu controle quando este apresenta alguma falha, se o mesmo deve ser terminado, reiniciado (com a possibilidade de manter seu estado interno ou não) ou se a falha deve ser repassada ao próximo supervisor, para que este saiba como lidar com a falha.

Para tal fim cada ator pode definir comportamentos de acordo com o seu estado, como executar uma tarefa antes de iniciar (*preStart*), depois de ser terminado (*postStop*), antes de reiniciar (*preRestart*) e depois de reiniciar (*postRestart*).

Outro conceito importante é a ordem de *shutdown* do sistema de atores. Um ator supervisor só pode ser terminado quando todos os atores sob o seu controle estiverem no estado terminado também, isto é, quando estes recebem um sinal de *shutdown* e não há mais nenhuma mensagem em suas *mailboxes* para ser processada.

Assim que é criado, o sistema de atores inicia pelo menos três atores.

* **/** – Este é o chamado *root guardian*. Não pode ser considerado um ator completo porque não possui um supervisor, mas é o supervisor de todos os outros atores que o próprio *toolkit Akka* cria ao iniciar a aplicação, que são os atores */system* e */user*.
* **/system** – Como o sistema de *logging* do *toolkit* foi implementado assincronamente com o usode atores, viu-se necessário a introdução deste supervisor para o controle de desligamento do mesmo, mantendo o sistema de logging ativo até que todos os outros atores tenham terminado suas tarefas adequadamente.
* **/user** – Este é o ator de maior interação por parte do programador. Ele será o pai de qualquer ator criado pela aplicação com o comando akka.actor.ActorSystem.actorOf() e é responsável pela estratégia de supervisão inicial de todos os atores.

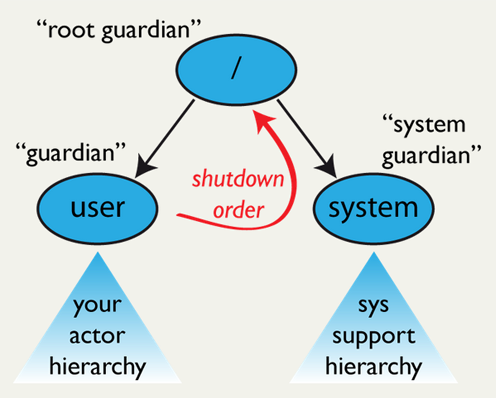


Figura – Hierarquia inicial do sistema de atores

Localidade transparente

*Akka* utiliza a mesma filosofia da *World Wide Web* para identificar e localizar recursos. URL’s são usadas com esquema *akka* ao invés de *http*. Os valores padrão são *akka://hostname/* e *akka://hostname:2552/* para o caso de haver atores remotos.

O conceito de hierarquia de atores visto na seção anterior permite que atores sejam unicamente identificados no sistema de atores. Adicionando-se a localização da máquina em que estiver rodando, tem-se um único endereço capaz de identificar um ator. Tanto atores locais quanto remotos são tratados da mesma forma, todos podem ser identificados por uma única URL.

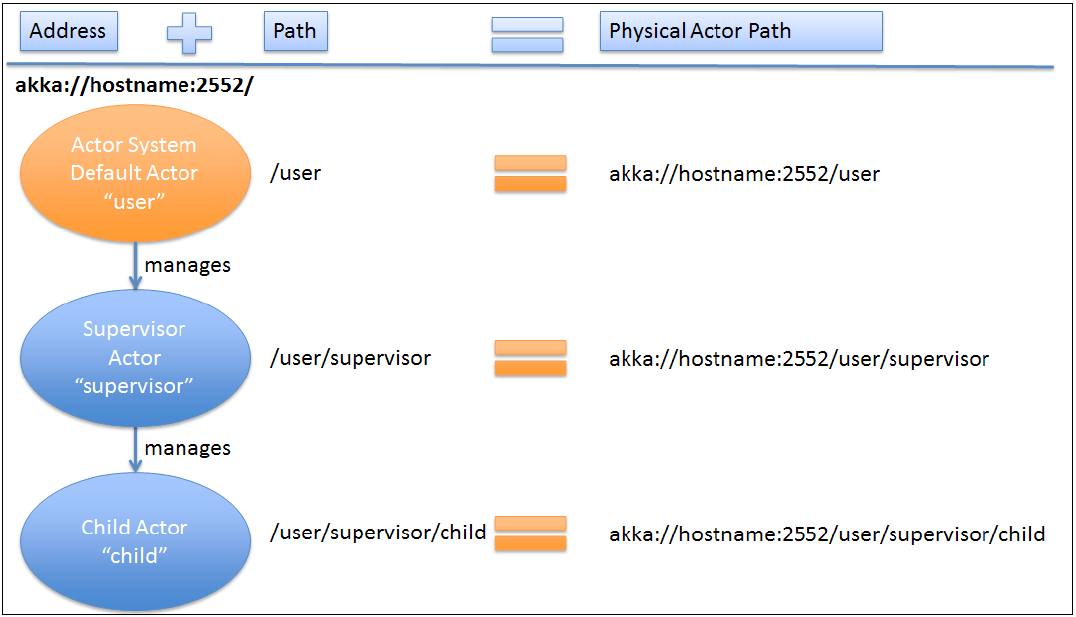


Figura - Localização de atores por meio de URL’s

Atores devem extender a interface (*trait* em *Scala*) akka.actor.Actor. Quando em ator é criado, o sistema de atores disponibiliza uma referência a ele através da classe akka.actor.ActorRef, que é baseada na URL que o identifica. Atores devem ser referenciados por esta classe e não pela referência direta a sua própria classe. Desta forma a localização de um ator torna-se transparente para a aplicação. Se for um ator remoto, o *toolkit Akka* cuida de todo o aparato de baixo nível necessário para a comunicação, como abertura e fechamento de *sockets*.

Todos os atores possuem acesso à sua própria referência e também ao ambiente que estão sendo executados através de um contexto. Atores podem obter referências de outros atores através do comando akka.actor.ActorSystem.actorFor(...).

*Akka* também possui suporte a transações combinando o modelo de atores com o conceito de STM (*Shared Transaction Memory*). STM utiliza o um mecanismo análogo ao de transações realizadas por bancos de dados para controlar o acesso à memória compartilhada por um sistema concorrente. Como está fora do escopo deste projeto, o suporte a transações não será mais abordado.

Ecurrency Domain

Como o sistema proposto é um sistema distribuído e orientado a mensagens, as mensagens serão o contrato entre cliente e provedor do sistema. Somando-se a isso o protocolo *AMQP* estabelece que mensagens devem ser passadas em um formato serializável.

Como visto anteriormente, cada *ecurrency* terá seu próprio domínio descrito por um arquivo *domain.proto*, que deverá ser compilado para gerar os objetos que representarão este domínio. Tudo isto é feito pelo compilador do Google Protocol Buffer. Cada *ecurrency* possuirá um módulo que contém este arquivo e as classes geradas pelo compilador.

Liberty Reserve Domain

Foi criado um módulo chamado *liberty-reserve-domain* que possui o arquivo *domain.proto*, que descreve todas as mensagens de domínio da API Liberty Reserve que serão trocadas pelo sistema.

A fase de compilação desse arquivo foi incorporada ao script que automatiza as tarefas de build do projeto, produzindo um arquivo *.jar* com as classes geradas na linguagem *Java*, relacionadas ao domínio desta *ecurrency*.

Provedor do Sistema

Esta seção mostrará a implementação do provedor do sistema, que é a parte consumidora do *broker AMQP* e que implementa os clientes das API’s disponibilizadas pelas *ecurrencies*. Será um sistema de atores *Akka*.

AMQP Service

Este módulo fará parte do sistema de atores com um ator supervisor, responsável pela criação de atores que implementarão a comunicação com o *broker RabbitMQ*. Cada um deles terá seu próprio *channel AMQP*.

A criação de um *channel* é passada como uma função ao ator para que este possa criá-lo no momento mais adequado possível, com uma simples chamada a esta função e sem conhecer os detalhes de sua criação. Com isso garante-se que cada *channel* criado seja acessado por somente uma thread.

O supervisor utilizará a estratégia de reiniciar o ator que apresentar alguma falha. Vale lembrar quando *Akka* reinicia um ator, ele destrói sua antiga referência e cria uma nova, mantendo sua *mailbox*, inclusive a mensagem que estava sendo processada no momento da falha.

Ao ser instanciado, o ator também criará uma simples implementação da classe com.rabbitmq.client.DefaultConsumer, que é o objeto que receberá as mensagens da fila do *broker RabbitMQ*. Sua única função será a de repassar as mensagens recebidas para a própria *mailbox* do ator. Esta classe está presente no cliente Java disponibilizado pelo *broker RabbitMQ*.

Ao receber uma mensagem de sua *mailbox*, a função do ator será a de determinar para qual implementação de *ecurrency* ele irá delegar a mensagem recebida. Para isso ele analisará a *routing key* da mensagem, que pelo contrato estabelecido com o broker deve seguir o padrão *<ecurrency-id>.<service-id>*.

Ao enviar uma mensagem de requisição, o cliente pode determinar a fila para onde deve ser mandada a resposta desta requisição. Toda mensagem *AMQP* (0.9.1) pode ter em seu cabeçalho os parâmetros *reply-queue* e *correlation-id*. Este último é usado em mensagens RPC, em que é necessário que o cliente permaneça bloqueado, esperando uma resposta para continuar seu processamento.

Ao detectar o parâmetro *reply-queue* na requisição, o ator manda a resposta para esta fila, usando o *exchange* padrão (em que a *routing key* possui o mesmo nome da fila). Se for detectado o parâmetro *correlation-id*, o ator o adicionará ao cabeçalho da resposta produzida.

Tudo será feito de forma assíncrona, usando a API de futures da linguagem Scala (*SIP-14 [[14]](#endnote-14)- Futures and Promises*), que permite que uma parte do código execute na forma de *callback*, logo que a resposta estiver disponível.

“The idea is simple, a *Future* is a sort of placeholder object that you can create for a result that doesn’t yet exist. Generally, the result of the *Future* is computed concurrently and can be later collected. Composing concurrent tasks in this way tends to result in faster, asynchronous, non-blocking parallel code.” (Haller, Procopec, Miller, Klang, Kuhn, & Jovanovic, 2012)

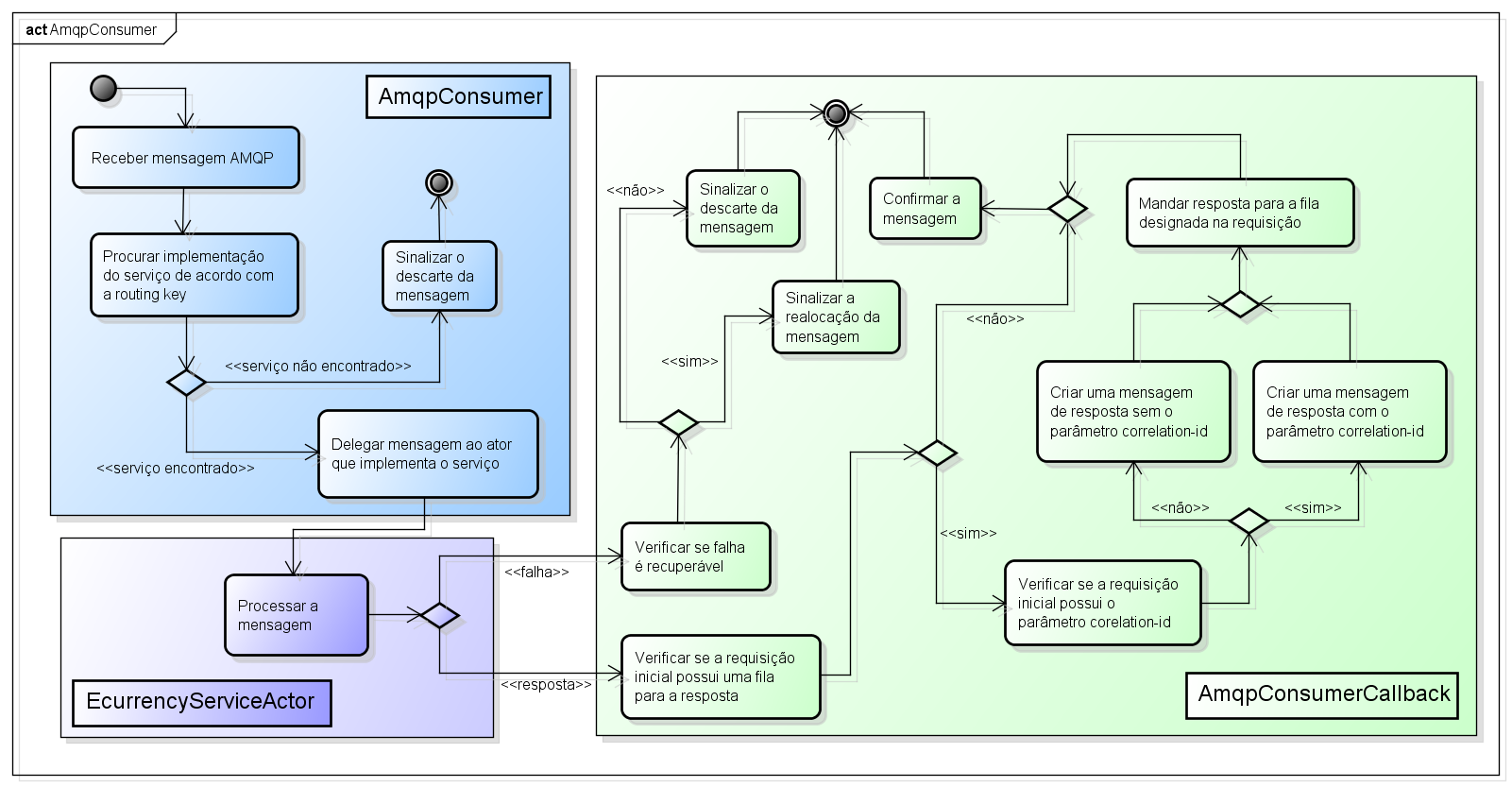
Uma das principais preocupações foi com o tratamento de falhas. A confirmação de mensagem recebida e processada só será enviada ao *broker AMQP* depois da requisição ter sido despachada e processada pela implementação adequada sem que haja nenhuma falha. Caso seja detectada uma falha irrecuperável, a mensagem será descartada e um sinal será enviado ao *broker*, informando que a mensagem não deve ser realocada. Caso a mensagem seja recuperável, o serviço deverá enviar um sinal de que a mensagem deve ser realocada na fila.

Figura – Diagrama de atividades que representa o funcionamento do módulo *AMQP Service*

Cada cor no diagrama de atividades acima representa um ator. A parte que de fato processa a mensagem (representada pela cor roxa) não é relevante ao serviço prestado por este módulo. Toda a execução do objeto de callback (*AmqpConsumerCallback*, em verde) fica a cargo do *framework* *Akka* como um ator temporário e caberá à classe *AmqpConsumer* definir esse comportamento. A listagem abaixo mostra como foi realizado este processo:



Código – Método do ator *AmqpConsumer* que recebe e despacha uma mensagem *AMQP*

Módulo API

Este é um módulo que contém classes em comum a todos os outros módulos que compõem o provedor. Porém sua única finalidade até o momento é sinalizar quando falhas que acontecem no processamento das mensagens são irreversíveis ou não.

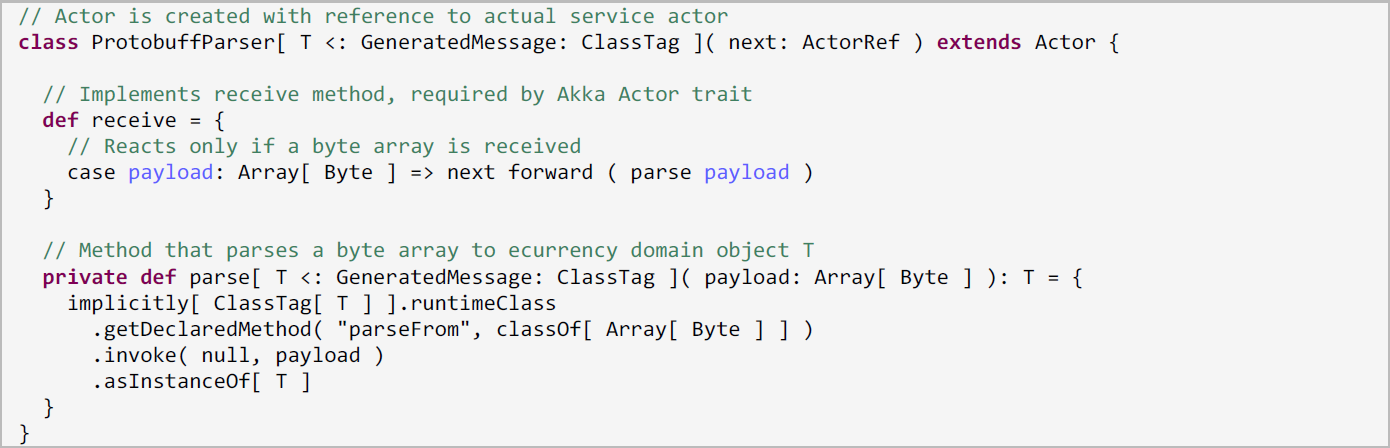
Seu uso pode ser visto na listagem do **Código 2**, em que a falha é sinalizada como sendo recuperável ou não, através da classe ecurrencies.api.EcurrencyServiceException, que possui o atributo *recoverable* do tipo booleano.

Módulo Akka Commons

Este módulo irá prover atores com funções convenientes a todos os módulos que implementam os serviços relacionados as *ecurrencies*.

Um caso que será comum a todos é a necessidade de se converter uma mensagem *protobuf* serializada (um array de bytes) em um objeto do domínio de uma *ecurrency*.

Para isto foi criada a classe ecurrencies.common.akka.ProtobufferParser. Este ator será executado de uma forma genérica, dependendo somente da biblioteca provida pelo Google Protocol Buffer. O módulo que for utilizar este serviço deverá prover em tempo de execução o módulo de domínio da *ecurrency*.



Código – Ator que converte um array de bytes em um objeto de domínio

O método parse usa a API de reflexão de Scala para dinamicamente executar o método parseFrom, que é um método estático presente em todas as classes geradas pelo compilador Protobuf.

Ao ser instanciado, este ator **recebe uma referência ao ator que de fato realizará a comunicação** com a *ecurrency*, para que este possa processar a mensagem já transformada em um objeto do domínio.

Ecurrency Service

Este serviço será composto de módulos em que cada módulo será o responsável pela implementação da API de uma *ecurrency*. Inicialmente foi disponibilizado o módulo que implementa o cliente da *ecurrency* Liberty Reserve. Todos farão parte do sistema de atores que irá compor o provedor.

Cada um destes módulos terá um ator supervisor com o nome que identifica a *ecurrency* que está sendo implementada, **o que é fundamental para a localização dos atores que realizam os serviços** relativos à API da *ecurrency*.

Todos os atores que implementam esses serviços deverão receber mensagens relativas ao domínio da *ecurrency* ao qual pertencem.

Módulo Liberty Reserve Service

Implementa todas as funções definidas pela API desta *ecurrency*. Algumas tarefas são necessárias, como converter objetos do domínio em mensagens JSON, realizar a comunicação. Cada um dos cinco serviços é realizado por um ator.

Todos os cinco casos de uso definidos no diagrama da Figura 2 foram generalizados de forma que uma classe genérica os implemente. O diagrama de atividades de um deles pode representar todos os outros:

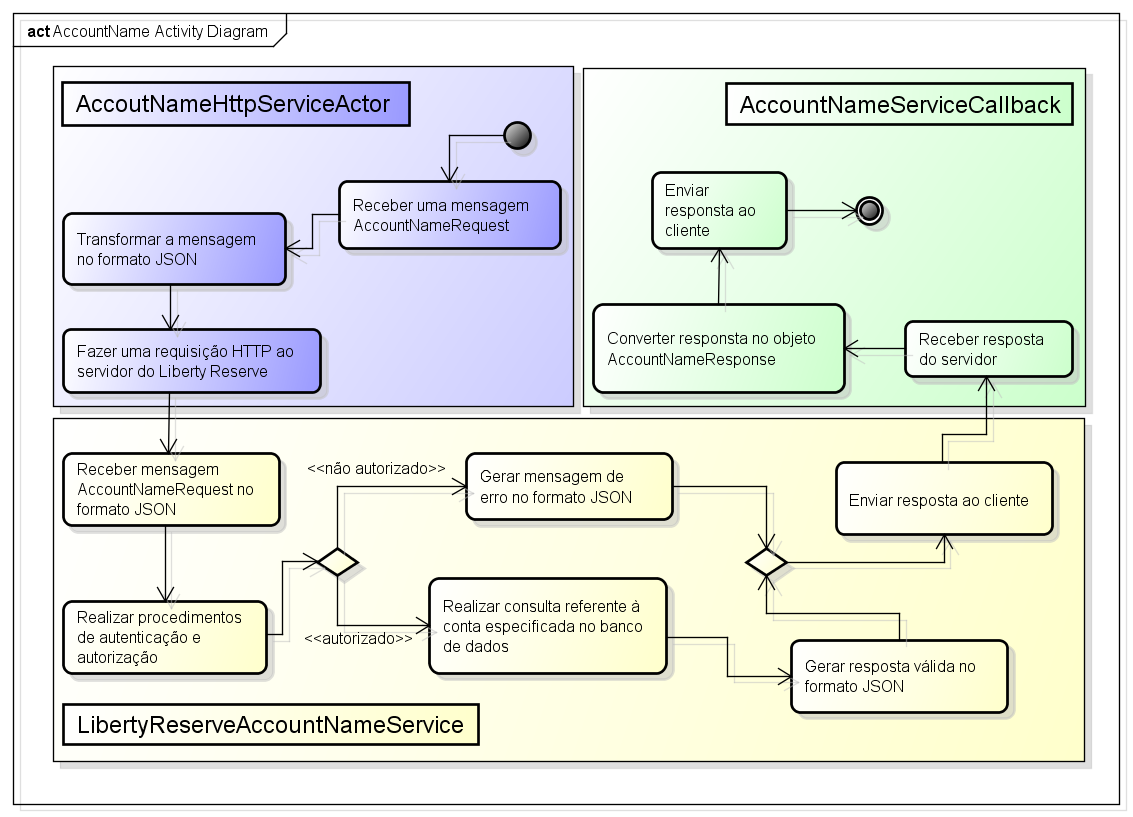
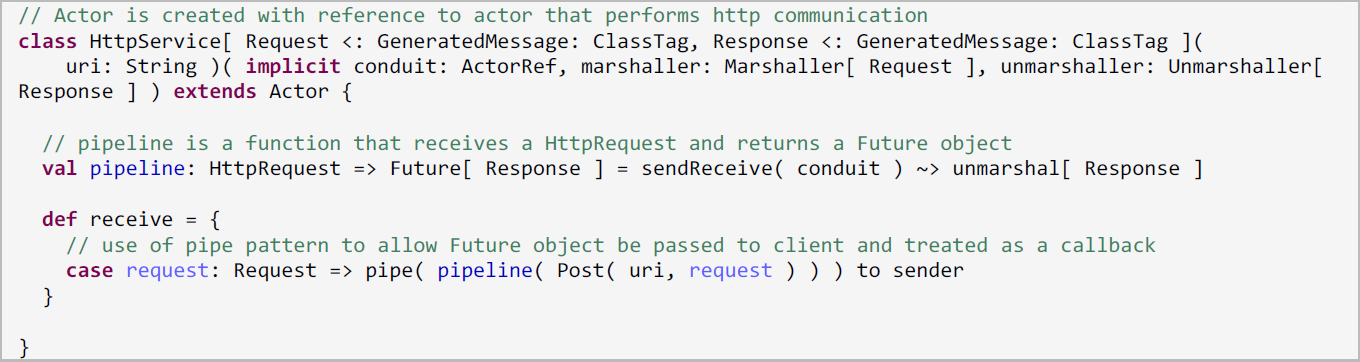


Figura – Diagrama de atividades que representa o serviço de requisição do nome de uma conta Liberty Reserve

O ator definido na classe ecurrencies.libertyreserve.service.HttpService é quem realiza este serviço de uma forma bem genérica. Precisa saber em tempo de compilação a classe do objeto que será a requisição e a classe do objeto que será a resposta. Em seu construtor devem estar presentes:

* A string que representa a URI para onde a mensagem deve ser enviada na requisição HTTP (*uri*)
* O objeto que realiza a conversão da requisição em JSON (*marshaller*), de acordo com as especificações da API Liberty Reserve
* O objeto que realiza a conversão da resposta em JSON na classe de domínio que representa a resposta (*unmarsahller*)
* O ator que de fato realizará a requisição HTTP (*conduit*)

A classe HttpService está ilustrada no código abaixo.



Código – Classe HttpService, generalização da comunicação com o servidor da API Liberty Reserve

Em oito linhas de código conseguiu-se realizar todas as iterações do diagrama de atividades representado na Figura 14. Para tal foram utilizadas as bibliotecas disponibilizadas no site [spray.io](http://spray.io/). Spray[[15]](#endnote-15) também é um toolkit construído na linguagem *Scala* e oferece bibliotecas de entrada e saída. Para este projeto foram utilizadas as bibliotecas spray-json, para a conversão de objetos criados pelo compilador protobuf no formato JSON e a biblioteca spray-client, um cliente HTTP assíncrono.

A classe ecurrencies.libertyreserve.service.LibertyReserveJsonProtocol realiza a conversão de todas as mensagens do módulo *liberty-reserve-domain* para JSON.

Como se pode notar existem atributos implícitos no construtor da classe HttpService. A biblioteca spray-client detecta a existência de um *marshaller* e um *unmarshaller* implicitamente e realiza a conversão da requisição para o formato JSON automaticamente na função *pipeline*.

Esta função recebe um objeto do tipo spray.http.HttpRequest e retorna um objeto *Future* com o tipo definido como a resposta na declaração da classe HttpService. Este objeto *Future* é repassado ao cliente para que ele possa executar sua tarefa sobre a resposta assim que ela estiver disponível. Na realidade o cliente cria um objeto de *callback* e o *toolkit* *Akka* faz com que ele execute somente no momento que a resposta estiver disponível, garantindo assim uma operação sempre assíncrona sobre as mensagens passadas.

O módulo cliente

O cliente desta aplicação será a parte produtora das mensagens *AMQP* e também um sistema de atores *Akka*. Trata-se de um sistema simples, que irá gerar aleatoriamente em um intervalo de tempo uma das cinco requisições da API Liberty Reserve para o *broker* *RabbitMQ*.

Serão dois tipos de produtores, ambos irão determinar uma fila para que haja o retorno de uma resposta. Os produtores serão implementados como atores.

O primeiro deles não bloqueará, sua função será somente preparar e mandar a mensagem com o parâmetro *reply-queue* definido. Haverá um consumidor, que também é um ator, para as respostas enviadas à fila definida na requisição. Caso este consumidor consiga determinar qual é o tipo de mensagem ele a imprimirá no console.

O segundo ator fará o mesmo, mas bloqueará para esperar por uma resposta. Para isto ele definirá, além do parâmetro *reply-queue*, o parâmetro *correlation-id*, que identificará a resposta como sendo relativa à requisição inicial.

O supervisor desses atores irá gerar em um determinado intervalo de tempo mensagens do domínio da *ecurrency* Liberty Reserve aleatórias para serem enviadas aos atores produtores, para que estes realizem a comunicação com o *broker RabbitMQ*.

Esse intervalo de tempo é especificado em um arquivo de configuração. Logo que executar a sua função, cada produtor irá suspender a *thread* na qual estiver sendo executado pelo período especificado. Logo que este período terminar, ele mandará uma mensagem ao seu supervisor relatando que está pronto para receber uma nova mensagem para ser enviada ao *broker*.

Este tipo de controle é necessário para prevenir o provedor da aplicação de inundar o servidor da API Liberty Reserve com requisições inválidas que foram geradas aleatoriamente para fins de teste.

Todos os atores produtores também recebem referências a atores que contabilizam mensagens produzidas, recebidas, que apresentaram algum erro ou que o tempo de resposta tenha expirado, no caso do produtor que bloqueia ao esperar pela resposta. Ao final da execução do sistema cliente, esses atores imprimirão um relatório comparando a quantidade de mensagens processadas usando-se os dois tipos de produtores.

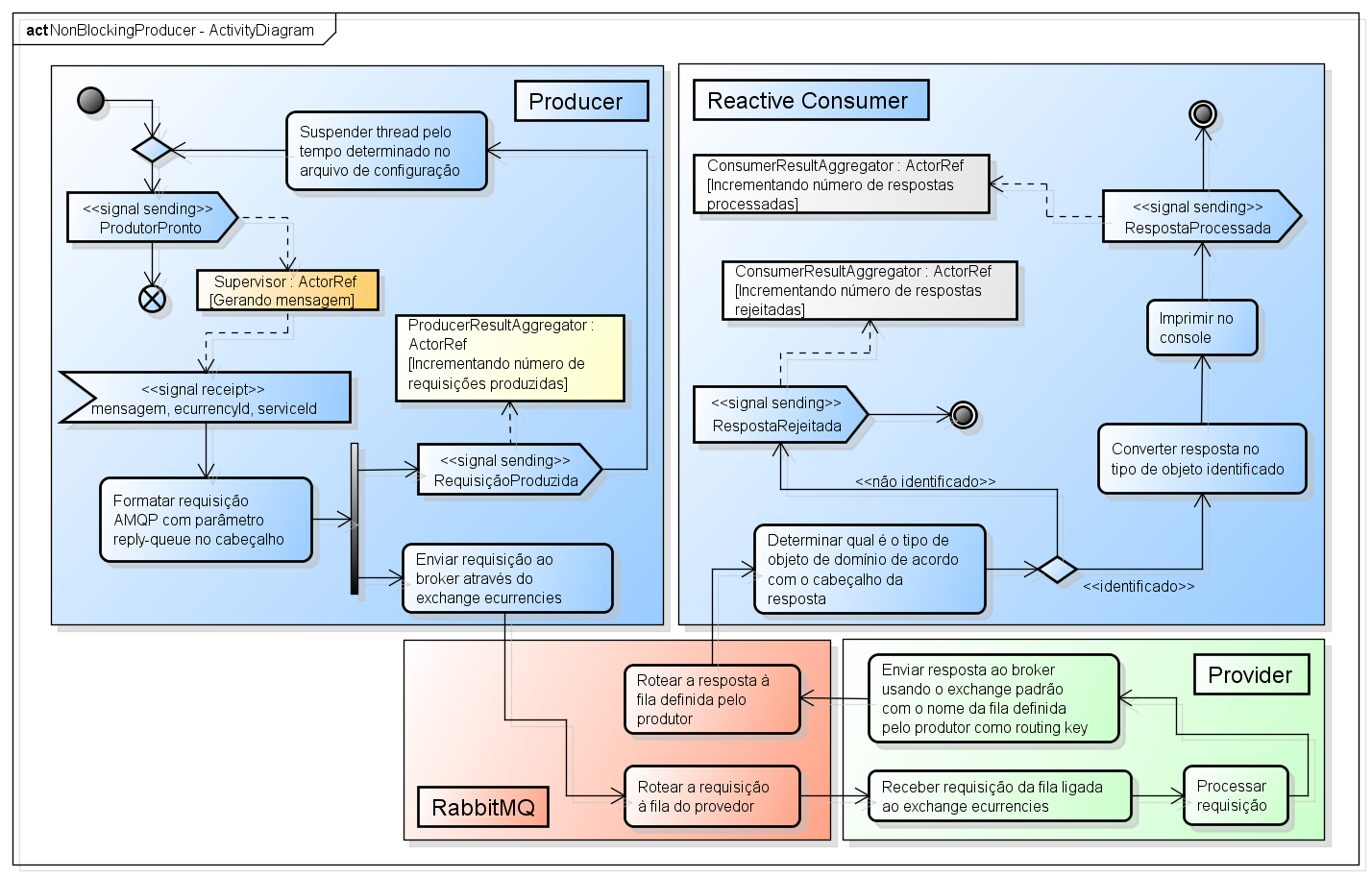
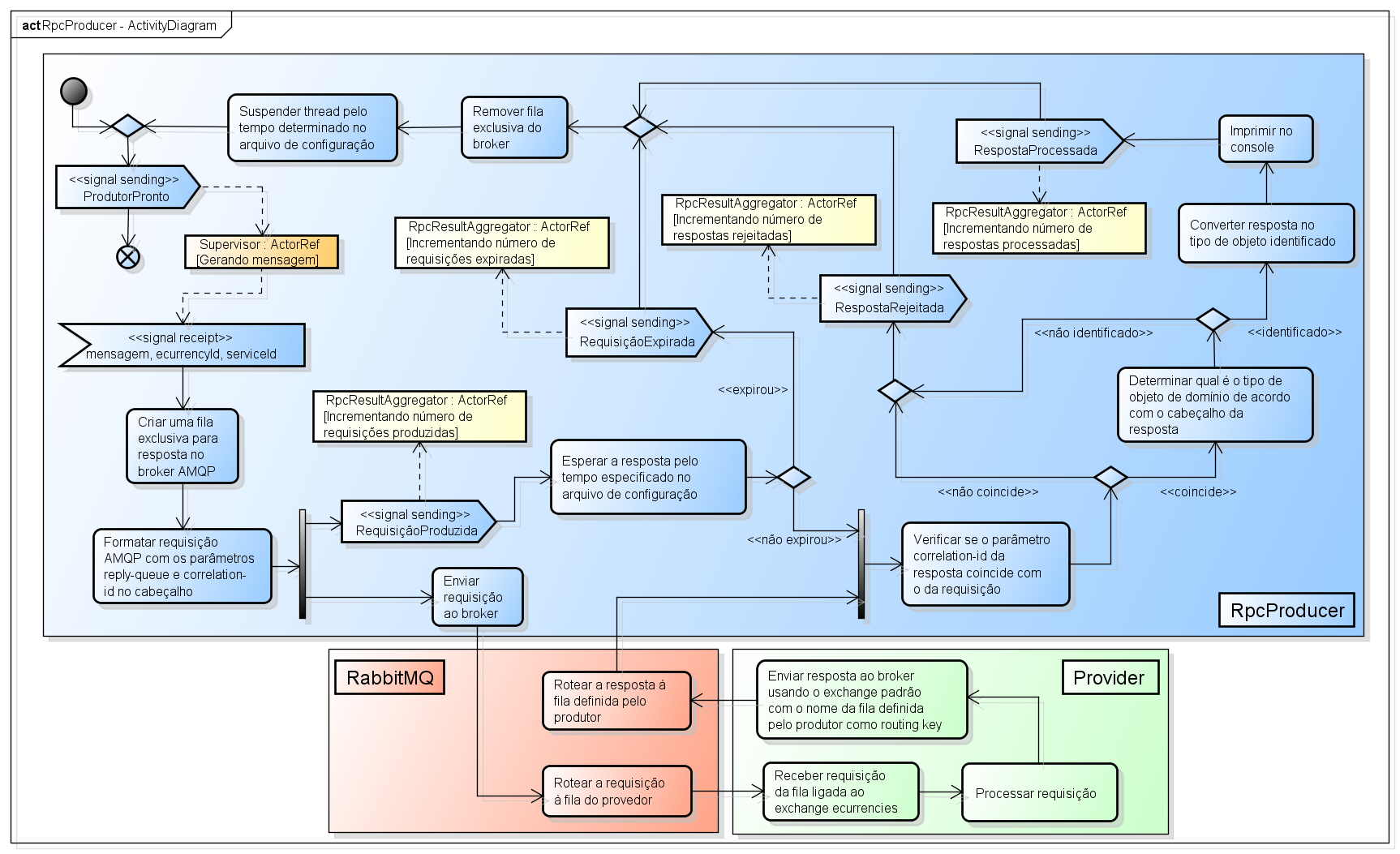


Figura – Diagrama de atividades do produtor que não bloqueia à espera de uma resposta

Todos os atores do módulo cliente estão representados na cor azul. O processamento por parte do provedor (Figura 14) não é relevante para o exemplo, por isso foi extremamente simplificado neste diagrama. Como se pode notar há apenas interação do cliente com o *broker AMQP*.

Vejamos agora como é o comportamento do segundo tipo de produtor:



capítulo 5

Avaliação de Desempenho

capítulo 6

Conclusões

# Bibliografia

Haller, P., Procopec, A., Miller, H., Klang, V., Kuhn, R., & Jovanovic, V. (2012). *SIP-14 - Futures and Promises.* Fonte: Documentação Scala: http://docs.scala-lang.org/sips/pending/futures-promises.html

K. Gupta, M. (2012). *Akka Essentials.* Packt Publishing.

Klein Ikkink, H. (2012). *Gradle Efective Implementation Guide.*

Ordesky, M., Spoon, L., & Venners, B. (2011). *Programming in Scala* (2ª ed.). Artima.

Videla, A., & J. W. Williams, J. (2012). *RabbitMQ in Action.* Manning Publications.

1. PagSeguro [Online] em <https://pagseguro.uol.com.br/> [↑](#endnote-ref-1)
2. PayPal [Online] em <https://www.paypal.com/> [↑](#endnote-ref-2)
3. Liberty Reserve [Online] em <https://www.libertyreserve.com/> [↑](#endnote-ref-3)
4. Liberty Reserve API [Online] em <https://www.libertyreserve.com/en/help/apiguide/> [↑](#endnote-ref-4)
5. Ecurrency [Online] em <http://en.wikipedia.org/wiki/Electronic_money> [↑](#endnote-ref-5)
6. AMQP [Online] em <http://www.amqp.org/> [↑](#endnote-ref-6)
7. Protocol Buffers [Online] em <https://developers.google.com/protocol-buffers/docs/overview> [↑](#endnote-ref-7)
8. Erlang [Online] em <http://www.erlang.org/> [↑](#endnote-ref-8)
9. Gradle [Online] em <http://www.gradle.org/> [↑](#endnote-ref-9)
10. Apache Ant [Online] em <http://ant.apache.org/> [↑](#endnote-ref-10)
11. Apache Maven [Online] em <http://maven.apache.org/> [↑](#endnote-ref-11)
12. RabbitMQ [Online] em <http://www.rabbitmq.com/> [↑](#endnote-ref-12)
13. Akka Toolkit [Online] em <http://akka.io/> [↑](#endnote-ref-13)
14. Scala Futures – SIP 14 [Online] em <http://docs.scala-lang.org/sips/pending/futures-promises.html> [↑](#endnote-ref-14)
15. Spray Toolkit [Online] em <http://spray.io/> [↑](#endnote-ref-15)