



Universidade do Minho

Escola de Engenharia

Departamento de Informática

João Paulo Oliveira de Andrade Marques

Middleware para integração com Sistemas de Queues



Universidade do Minho

Escola de Engenharia

Departamento de Informática

João Paulo Oliveira de Andrade Marques

**Middleware para integração com Sistemas
de Queues**

Dissertação de Mestrado

Mestrado Integrado em Engenharia Informática

Trabalho efetuado sobre a orientação do

Professor Doutor António Manuel Nestor Ribeiro

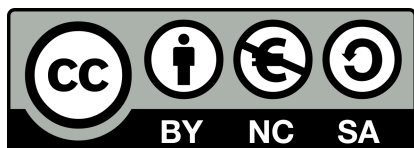
DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositoriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



**Creative Commons Atribuição-NãoComercial-Compartilhalgual 4.0 Internacional
CC BY-NC-SA 4.0**

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Agradecimentos

Antes de mais quero agradecer a toda a minha família por me terem apoiado sempre, mas com especial atenção à minha namorada, aos meus pais e irmã, e aos pais da minha namorada por me terem sempre incentivado e apoiado em todas as minhas decisões, tanto na escolha do curso, como na escolha do tema da dissertação. A eles, um muitíssimo obrigado por tudo o que me proporcionaram ao longo desta caminhada que, sem todos eles, talvez não teria sido possível.

A todos os meus professores, um enorme obrigado por todo o conhecimento que partilharam comigo, por tudo o que me ensinaram, mas em especial ao meu orientador, Professor António Manuel Nestor Ribeiro, que me apresentou este tema e que em todas as reuniões partilhou comigo as suas ideias e opiniões de forma a melhorar as funcionalidades do Middleware desenvolvido, bem como pelos momentos de conversa sobre vários outros assuntos, que de certa forma facilitaram bastante todo o trabalho.

Deixo também o agradecimento a todos os meus colegas que me acompanharam ao longo deste percurso e com os quais partilhei muitos projetos, principalmente ao José Pereira e ao Ricardo Petronilho, com os quais discuti algumas tecnologias a utilizar durante a realização desta dissertação, por toda a ajuda e momentos de convívio.

Assim me despeço desta que foi a minha segunda casa durante os últimos 5 anos. A todos um grande obrigado por me terem feito chegar até aqui.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

_____, _____
(Localização) (Data)

(João Paulo Oliveira de Andrade Marques)

Middleware para integração com Sistemas de Queues

Resumo

Com o crescente número de componentes para gestão de sistemas de *queues* e o crescente número de aplicações cliente a fazer uso desses mesmos componentes é necessário a criação de um *Middleware* para desacoplar as aplicações cliente dos sistemas de *queues*. Os sistemas de *queues* são também conhecidos com *Message-Oriented Middleware* (MOM).

O acoplamento das aplicações cliente a esses componentes torna-as muito dependentes destes, pelo que a introdução de um *Middleware* faz com que a aplicação cliente fique isolada das particularidades e das tarefas de manutenção. Posto isto, fica o *Middleware* dependente dessas particularidades e das tarefas de manutenção.

O RabbitMQ, o ActiveMQ e o Kafka são exemplos de sistemas de *queues* onde existe um sistema intermediário externo entre as aplicações que estão a comunicar, e o ZeroMQ que é um sistema de *queues* onde a própria aplicação fica como um nodo do sistema de *queues*, isto é, o ZeroMQ é um sistema intermediário interno. Todos estes são implementados de diferentes formas, pelo que a troca de um sistema para outro leva a uma reestruturação das aplicações que o estejam a usar, por isso estes serão estudados durante esta dissertação de forma a avaliar as suas características, vantagens e desvantagens para realizar a sua integração no *Middleware* a desenvolver.

O *Middleware* desenvolvido desacopla as aplicações dos sistemas de *queues*, permitindo assim a troca de um sistema para outro sem ser necessária uma reestruturação da aplicação. Este integrou o RabbitMQ, o ActiveMQ e o Kafka por forma a ser possível realizar as operações básicas de envio e leitura de mensagens. Além destas operações é também possível reler mensagens quando seja necessário.

Por forma a demonstrar e testar o *Middleware* ir-se-á recorrer a um caso de estudo.

Palavras-chave: Comunicação Assíncrona, *Middleware*, Sistemas de *Queues*

Middleware for integration with Queue Systems

Abstract

With the growing number of components for managing queue systems and the growing number of client applications making use of those same components, it is necessary to create Middleware to decouple client applications from queue systems. Queue systems are also known as Message-Oriented Middleware (MOM).

The coupling of client applications to these components makes them very dependent on them, so the introduction of Middleware makes the client application isolated from particularities and maintenance tasks. That said, Middleware is dependent on these particularities and maintenance tasks.

RabbitMQ, ActiveMQ and Kafka are examples of queue systems where there is an external intermediary service between the applications that are communicating, and ZeroMQ is a queue system where the application itself is a node of the queue system, that is, ZeroMQ is an internal intermediation service. All of these are implemented in different ways, so switching from one system to another leads to a restructuring of the applications that are using it, so these will be studied during this dissertation in order to evaluate its characteristics, advantages and disadvantages to perform its integration in the Middleware to be developed.

The developed Middleware decouples the applications from the queue systems, thus allowing the exchange from one system to another without needing to restructure the application. This integrated RabbitMQ, ActiveMQ and Kafka in order to be able to carry out the basic operations of sending and reading messages. In addition to these operations, it is also possible to reread messages when necessary.

In order to test the Middleware, a case study will be used.

Keywords: Asynchronous Communication, Middleware, Queue Systems

Lista de Figuras	x
Lista de Tabelas	xiv
Acrónimos	xvi
1 Introdução	1
1.1 Objetivos	3
1.2 Estrutura do Documento	3
2 Estado da Arte	5
2.1 Protocolos e APIs	5
2.1.1 Advanced Message Queue Protocol	5
2.1.2 Java Message Service	7
2.1.3 Message Queue Telemetry Transport	8
2.1.4 Simple (Streaming) Text Orientated Messaging Protocol	10
2.1.5 Extensible Messaging and Presence Protocol	12
2.2 Tecnologias	13
2.2.1 RabbitMQ	13
2.2.2 ActiveMQ	15
2.2.3 Kafka	17
2.2.4 ZeroMQ	18
2.3 Estudos Comparativos	21
2.3.1 RabbitMQ vs. ActiveMQ	21
2.3.2 RabbitMQ vs. ZeroMQ	22
2.3.3 ActiveMQ vs. OpenMQ	24
2.3.4 Kafka vs. RabbitMQ	27
2.4 Trabalhos Relacionados	31
2.4.1 Multi-MOM	31
3 Descrição do Problema	33
4 Conceção	37
4.1 Modelo de Domínio	37
4.2 Levantamento de Requisitos	38
4.2.1 Priorização	39
4.3 Modelação	40
4.3.1 Diagrama de Casos de Uso	40

4.3.2	Diagrama de Componentes	41
4.3.3	Decisões Arquiteturais	45
4.3.4	Diagrama de Classes	47
5	Implementação	71
5.1	Tecnologias	71
5.1.1	Desenvolvimento	71
5.1.2	Teste	73
5.1.3	Deployment	74
5.2	Especificações	75
5.2.1	Sistemas de Queues e Estratégias	75
5.2.2	API Rest	76
5.3	Decisões de Implementação	80
5.3.1	RabbitMQ	80
5.3.2	Tempo de Persistência das Mensagens	80
5.3.3	Domain Name System (DNS)	81
6	Caso de Estudo	82
6.1	Demonstração	83
7	Teste de Desempenho	90
8	Conclusão	95
8.1	Trabalho Futuro	95
	Bibliografia	97
	Anexos	101
A	Diagramas de Classe	101
A.1	Registration	101
A.2	Sender	102
A.3	Receiver	103
A.4	Discovery	104
B	Especificações API	105
B.1	Registration	105
B.1.1	Rotas	105
B.1.2	Respostas	107
B.1.3	Erros	108
B.2	Middleware	109

B.2.1	Rotas	109
B.2.2	Respostas	116
B.2.3	Erros	118

Lista de Figuras

Figura 1:	MOM broker-based.	2
Figura 2:	MOM brokerless.	2
Figura 3:	Arquitetura AMQP 0.9.1.	6
Figura 4:	API JMS 1.1.	7
Figura 5:	API JMS 2.0.	7
Figura 6:	PTP e Pub/Sub no JMS 1.1.	8
Figura 7:	PTP e Pub/Sub no JMS 2.0.	8
Figura 8:	Mensagens enviadas nos diferentes níveis do QoS do MQTT.	9
Figura 9:	Estrutura mensagem STOMP.	11
Figura 10:	Fluxo de comunicação entre nodos XMPP.	12
Figura 11:	Topologias RabbitMQ.	14
Figura 12:	Número de mensagens por segundo (msg/s) processadas pelo sistema RabbitMQ em função do número de utilizadores.	15
Figura 13:	Arquitetura ActiveMQ Broker.	16
Figura 14:	Topologia Kafka com apenas um nodo e múltiplos sistemas.	17
Figura 15:	Topologia Kafka com múltiplos nodos com e múltiplos sistemas.	17
Figura 16:	Padrão Pipeline.	19
Figura 17:	Padrão Extended Req/Res.	20
Figura 18:	Topologias do cenário de teste.	23
Figura 19:	Número de msg/s processadas com 3 consumidores.	24
Figura 20:	Fluxo de comunicação do OpenMQ em cluster utilizada.	25
Figura 21:	Fluxo de comunicação do ActiveMQ em cluster utilizada.	25
Figura 22:	Cenário de teste 1.	25
Figura 23:	Cenário de teste 2.	25
Figura 24:	Entrega fiável de mensagens.	28
Figura 25:	Resultados teste com QoS nível 0.	31
Figura 26:	Impacto no desempenho consoante o número de tópicos no sistema Kafka.	31
Figura 27:	Aplicação dependente do sistema de queues.	33
Figura 28:	Biblioteca entre as aplicações e os sistemas de queues.	34
Figura 29:	Middleware interno ao projeto.	35
Figura 30:	Middleware externo ao projeto.	35
Figura 31:	Modelo de Domínio.	37
Figura 32:	Diagrama de Casos de Uso.	40

Figura 33: Diagrama de componentes de alto nível de abstração.	42
Figura 34: Diagrama de componentes do Componente Registration.	42
Figura 35: Diagrama de componentes do Componente Middleware.	43
Figura 36: Padrão Hexagonal.	45
Figura 37: Padrão Adapter.	45
Figura 38: Padrão Facade.	46
Figura 39: Padrão Singleton.	46
Figura 40: Arquitetura e classes inbounds do componente Registry.	48
Figura 41: Arquitetura e classes business do componente Registry.	49
Figura 42: Arquitetura e classes outbounds do componente Registry.	49
Figura 43: Classe RegistryController.	50
Figura 44: Classe RegistryService.	50
Figura 45: Classe ProjectDAO.	51
Figura 46: Classe ProjectData.	51
Figura 47: Classe BrokerDAO.	51
Figura 48: Classe BrokerData.	52
Figura 49: Arquitetura e classes inbounds do componente Sender.	52
Figura 50: Arquitetura e classes business do componente Sender.	53
Figura 51: Arquitetura e classes outbounds do componente Sender.	54
Figura 52: Classe ProducerController.	55
Figura 53: Classe SenderService.	55
Figura 54: Classes KafkaProducer.	56
Figura 55: Classe ActiveMQ5Producer.	56
Figura 56: Classe RabbitMQProducer.	56
Figura 57: Classe ZeroMQProducer.	57
Figura 58: Classe ProducerDataDAO.	57
Figura 59: Classe ProducerData.	57
Figura 60: Classe MessageReceived.	58
Figura 61: Classe MessageToSend.	58
Figura 62: Arquitetura e classes inbounds do componente Receiver.	59
Figura 63: Arquitetura e classes business do componente Receiver.	59
Figura 64: Arquitetura e classes outbounds do componente Receiver.	60
Figura 65: Classe ConsumerController.	61
Figura 66: Classe ConsumerService.	61

Figura 67: Classe RabbitMQConsumer.	62
Figura 68: Classe ActiveMQ5Consumer.	62
Figura 69: Classe KafkaConsumer.	63
Figura 70: Classe ZeroMQConsumer.	63
Figura 71: Classe MessageDAO.	64
Figura 72: Classe MessageData.	64
Figura 73: Classe ConsumerDAO.	64
Figura 74: Classe ConsumerData.	65
Figura 75: Classe EmitterDAO.	65
Figura 76: Classe Discovery.	66
Figura 77: Classe ConsumerInstance.	66
Figura 78: Arquitetura e classes inbounds do componente Discovery.	66
Figura 79: Arquitetura e classes business do componente Discovery.	67
Figura 80: Arquitetura e classes outbounds do componente Discovery.	67
Figura 81: Classe ReceiverController.	68
Figura 82: Classe RabbitMQ.	68
Figura 83: Classe DiscoveryService.	69
Figura 84: Classe ReceiverDAO.	69
Figura 85: Classe ReceiverData.	69
Figura 86: Classe ReceiverService.	70
Figura 87: Injeção de dependência no Spring Boot.	72
Figura 88: Anotações biblioteca spring-boot-starter-validation.	72
Figura 89: Envio de mensagens e leituras das mesmas.	84
Figura 90: Alteração do Sistema de queues no producer1.	85
Figura 91: Envio e leitura de mensagens após alteração do Sistema de queues do producer1.	85
Figura 92: Alteração do Sistema de queues no consumer2.	86
Figura 93: Envio e leitura de mensagens após alteração do Sistema de queues do consumer2.	87
Figura 94: Alteração do Sistema de queues no consumer2 com estratégia incompatível.	87
Figura 95: Alteração do Sistema de queues no consumer2 para configuração idêntica ao consumer4.	88
Figura 96: Envio e leitura de mensagens após alteração do Sistema de queues do consumer2.	89
Figura 97: Arquitetura com Serviço de teste e Sistema de queues em máquinas distintas máquina.	90
Figura 98: Arquitetura com Serviço de teste e Middleware desenvolvido.	90
Figura 99: Cálculo do Tempo Médio de envio ($\bar{T}$).	91
Figura 100: Cálculo da Perda de Desempenho (Pd).	91

Figura 101: Cálculo da Média dos Tempos Médios ($\overline{T}_{med}$).	92
Figura 102: Diagrama de Classes do serviço Registration.	101
Figura 103: Diagrama de Classes do serviço Sender.	102
Figura 104: Diagrama de Classes do serviço Receiver.	103
Figura 105: Diagrama de Classes do serviço Discovery.. . . .	104

Lista de Tabelas

Tabela 1:	Tempo em milissegundos do envio de mensagens da aplicação cliente para o sistema.	21
Tabela 2:	Tempo em milissegundos do envio de mensagens do sistema para a aplicação cliente.	22
Tabela 3:	Tempo em milissegundos do envio de mensagens da aplicação cliente para o sistema com diferente número de clientes.	22
Tabela 4:	Mensagens enviadas/recebidas por segundo para o teste 1.	26
Tabela 5:	Payload enviado/recebido por segundo para o teste 1.	26
Tabela 6:	Melhoria de desempenho em percentagem do ActiveMQ em relação ao OpenMQ no envio/recepção para o teste 1.	26
Tabela 7:	Mensagens enviadas/recebidas por segundo para o teste 2.	27
Tabela 8:	Payload enviado/recebido por segundo para o teste 2.	27
Tabela 9:	Melhoria de desempenho em percentagem do ActiveMQ em relação ao OpenMQ no envio/recepção para o teste 2.	27
Tabela 10:	Latência em milissegundos com sistemas não replicados.	30
Tabela 11:	Latência em milissegundos com sistemas replicados.	30
Tabela 12:	Requisitos.	39
Tabela 13:	Mapeamento de especificações entre o Middleware e os sistemas de queues.	75
Tabela 14:	Formato Respostas e Exceções personalizadas.	76
Tabela 15:	Formato exceções APIGateWay (Kong).	77
Tabela 16:	Formato exceções framework Spring Boot.	77
Tabela 17:	API Rest do serviço Registration.	78
Tabela 18:	API Rest do serviço Middleware.	79
Tabela 19:	Especificações dos consumidores criados.	82
Tabela 20:	Especificações dos produtores criados.	83
Tabela 21:	Resultados dos testes realizados.	93
Tabela 22:	Campos da rota número 1.	105
Tabela 23:	Campos das rotas número 2 e 3.	106
Tabela 24:	Campos da rota número 4.	106
Tabela 25:	Campos da rota número 6.	106
Tabela 26:	Campos da rota número 7.	107
Tabela 27:	Respostas aos pontos de entrada do serviço Registration.	107
Tabela 28:	Códigos de erro do serviço Registration.	108
Tabela 29:	Erros por rota do serviço Registration.	109
Tabela 30:	Campos da rota número 1.	110

Tabela 31: Combinação de campos semi obrigatórios para a rota 1, 5 e 7 a 9.	110
Tabela 32: Campos das rotas número 2 e 3.	111
Tabela 33: Campos da rota número 4.	111
Tabela 34: Campos da rota número 5.	111
Tabela 35: Campos da rota número 6.	112
Tabela 36: Campos da rota número 7.	112
Tabela 37: Campos da rota número 8.	112
Tabela 38: Campos da rota número 9.	113
Tabela 39: Campos da rota número 10.	113
Tabela 40: Combinação de campos semi obrigatórios para a rota 10 e 16 a 18.	114
Tabela 41: Campos das rotas número 11 e 12.	114
Tabela 42: Campos da rota número 13.	114
Tabela 43: Campos da rota número 14.	114
Tabela 44: Campos da rota número 15.	115
Tabela 45: Campos da rota número 16.	115
Tabela 46: Campos da rota número 17.	115
Tabela 47: Campos da rota número 18.	116
Tabela 48: Campos da rota número 19.	116
Tabela 49: Campos das rotas número 20 e 21.	116
Tabela 50: Respostas aos pontos de entrada do serviço Middleware.	118
Tabela 51: Códigos de erros do serviço Middleware.	120
Tabela 52: Erros por rota do serviço Middleware.	121

Acrónimos

AMQP	Advanced Message Queue Protocol
API	Application Program Interface
ASF	Apache Software Foundation
BD	Base de Datos
CRUD	Create - Read - Update - Delete
DAO	Data Access Object
DI	Dependency Injection
DNS	Domain Name System
DTO	Data Transfer Object
EOL	End-of-Line
GCP	Google Cloud Platform
HTTP	Hyper Text Transport Protocol
HTTPS	Hyper Text Transport Protocol Secure
IETF	Internet Engineering Task Force
IM	Instant Messaging
IoT	Internet of Things
IP	Internet Protocol
IPSec	Internet Protocol Security
ISO	International Organization Standardization
JAAS	Java Authentication and Authorization Service
JID	Jabber ID
JMS	Java Message Service
JSON	JavaScript Object Notation
JWT	JSON Web Token
M2M	Machine-to-Machine
MOM	Message-Oriented Middleware
MQTT	Message Queue Telemetry Transport
MQTT-SN	Message Queue Telemetry Transport for Sensor Network
MVC	Model - View - Controller
MVP	Minimum Viable Product
OASIS	Organization for the Advanced Structured Information Standards
OpenPGM	Open Pragmatic General Multicast
P2P	Peer-to-Peer

PGM	Pragmatic General Multicast
POJO	Pure Old Java Object
PTP	Point-to-Point
Pub/Sub	Publisher/Subscriber
QoS	Quality of Service
Req/Res	Request/Response
RFC	Request for Comment
SASL	Simple Authentication and Security Layer
SSE	Server Sent-Event
SSL	Secure Sockets Layer
STOMP	Simple (Streaming) Text Orientated Messaging Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTL	Time to Live
UDP	User Datagram Protocol
UUID	Universally Unique Identifier
VM	Virtual Machine
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

1 Introdução

Com o atual crescimento do desenvolvimento de software foram aparecendo muitas aplicações, porém estas seguiam o modelo monolítico, em que uma aplicação é um componente único que faz tudo. Com o decorrer dos anos deparou-se com um problema de escalabilidade e de manutenção destas, pois sempre que era necessário realizar uma alteração era necessário reiniciar toda a aplicação.

Por forma a resolver este problema foi introduzido o modelo microsserviços, sendo que este consiste em dividir um componente complexo, como uma aplicação monolítica, em vários componentes mais simples por forma a obter os mesmos resultados pretendidos. Neste modelo cada componente é responsável por realizar apenas um conjunto específico de tarefas e sempre que necessário comunicar com outro componente para a realização de uma tarefa mais complexa.

Esta divisão de um componente complexo em vários componentes mais simples e a necessidade de comunicar entre eles fez surgir um novo problema. Este problema consiste na falha de comunicação entre dois destes componentes, muitas vezes devido a uma falha de rede ou então a uma falha do componente a ser contactado.

Algumas destas falhas podem levar ao bloqueio de alguns componentes, porque no caso de a informação a ser enviada entre os dois componentes ser de elevada importância o componente produtor da informação vai querer garantir que consegue entregar esta e que não é perdida. Este bloqueio ocorre porque a comunicação entre os componentes é síncrona.

Por forma a resolver estas falhas foram desenvolvidos sistemas de queues (filas). Estes resolvem os problemas relativos às falhas dos componentes recetores da informação, pois uma falha na rede pode sempre ocorrer, sendo que atualmente as falhas de rede são bastante reduzidas. Estes sistemas são comumente designados por sistemas intermediários ou Message-Oriented Middlewares (MOMs).

Sendo os sistemas de queues um componente estes também podem falhar, porém estes sistemas foram desenvolvidos de forma a serem altamente disponíveis e resilientes.

A introdução destes sistemas veio tornar a comunicação assíncrona entre os componentes que anteriormente necessitavam de comunicar, pois estes em vez comunicarem diretamente agora comunicam entre si através dos sistemas de queues.

Devido à necessidade destes componentes para realizar a comunicação assíncrona foram desenvolvidos diversos. Uns com mais funcionalidades do que outros, uns seguiram protocolos, outros seguiram Application Program Interfaces (APIs) de comunicação já existentes e alguns criaram ainda o seu próprio protocolo de comunicação.

Os sistemas intermediários dividem-se em dois grandes grupos, os intermediários externos ou *broker-based*, tal como o RabbitMQ [Dos14], ActiveMQ 5 [SBD11] e o Kafka [Gar15], e os intermediários internos ou *brokerless*, como o ZeroMQ [Akg13]. Os sistemas intermediários internos não possuem um componente externo para realizar a comunicação assíncrona entre os dois componentes que necessitam de comunicar, sendo os próprios componentes a comunicar diretamente, mas de forma assíncrona não levando ao bloqueio do componente produtor.

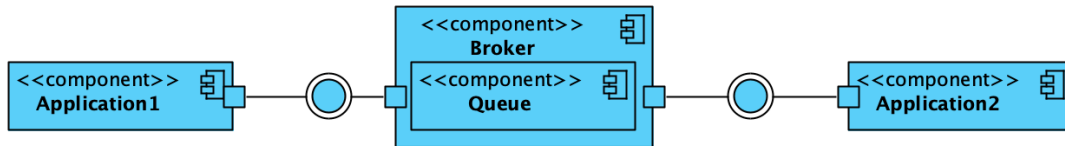


Figura 1: MOM broker-based.

A Figura 1 representa a utilização de um sistema intermediário externo, tal como pode observar-se o componente Broker está externo às aplicações. Por outro lado na Figura 2 está representada a utilização de um sistema intermediário interno, isto é, cada componente que esteja a utilizar este sistema torna-se um nodo do mesmo, não existindo fisicamente um componente externo a estas.

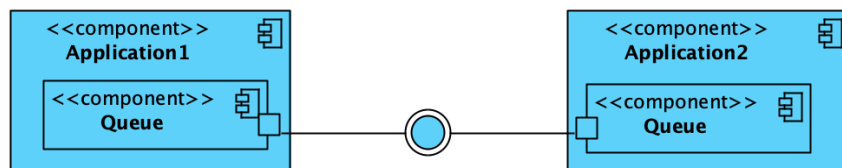


Figura 2: MOM brokerless.

Cada um destes sistemas apresenta as suas vantagens e desvantagens, pelo que existem estudos comparativos entre estes por forma a agilizar o processo de escolha no momento de decisão sobre qual destes sistemas utilizar. São exemplos destes estudos:

- N. Estrada e H. Astudillo em [EA15]: estudo de escalabilidade e desempenho entre o RabbitMQ e o ZeroMQ, um sistema intermediário externo e um sistema intermediário interno, respetivamente.
- V. M. Ionescu em [lon15]: estudo de desempenho entre o ActiveMQ e o RabbitMQ, dois sistemas intermediários externos.
- A. F. Klein et al. [Kle+15]: estudo de escalabilidade, desempenho e persistência de mensagens entre o ActiveMQ e o OpenMQ¹, dois sistemas intermediários externos que seguem a mesma API.
- Philippe Dobbelaere e Kyumars Sheykh Esmaili em [DE17]: estudo qualitativo e quantitativo entre o RabbitMQ e o Kafka, dois sistemas intermediários externos.

O objetivo desta dissertação passa pela conceção e implementação de um *Middleware* que integre vários destes sistemas de queues por forma a criar um nível de abstração entre os componentes que necessitam de comunicar e os sistemas de queues, acrescentado também características de valor acrescentado para uma maior experiência de utilização. Pois com a introdução dos sistemas de queues foi criado um novo problema que é a dependência que os componentes passam a ter sobre estes sistemas.

Este aspeto tem sido alvo de estudo pela comunidade, tal como o trabalho de Yuri Bezerra [Bez10], em que é criada uma biblioteca para abstração das estratégias de comunicação síncrona e assíncrona para aplicações

¹OpenMQ: <https://javaee.github.io/openmq/Documentation.html>

mobile.

1.1 Objetivos

Nesta secção serão ilustrados os objetivos para a dissertação a realizar. De seguida encontram-se enumerados os principais objetivos delineados.

1. Efetuar um estudo sobre alguns protocolos e APIs existentes na área da comunicação e utilizados ou suportados pelos sistemas intermediários a estudar, demonstrando a sua arquitetura e principais características.
2. Efetuar um estudo sobre os sistemas de queues já existentes, como os já acima referidos (RabbitMQ, ActiveMQ, Kafka e ZeroMQ), apontando as suas principais características, vantagens e desvantagens, demonstrando algumas comparações realizadas entre estes.
3. Fazer uma avaliação qualitativa dos sistemas de queues a utilizar.
4. Realizar o levantamento, análise e priorização de requisitos a ter em conta na conceção e implementação do Middleware.
5. Conceber e desenvolver um Middleware com características de valor acrescentado para a integração dos vários sistemas de queues por forma a tornar as aplicações que necessitam destes menos dependentes. Este Middleware deve ainda ser escalável e permitir redundância.
6. Por forma a manter o Middleware sempre atualizado com novos sistemas de queues este deve permitir a sua extensibilidade, mantendo o Middleware atualizado sem grandes complicações. Isto sem afetar os sistemas já adicionados anteriormente e sem afetar as aplicações cliente que estejam dependentes deste Middleware.
7. Criação de um caso de estudo que faça uso deste Middleware por forma a poder demonstrar as capacidades do mesmo.
8. Realizar um teste de desempenho por forma a avaliar o desempenho do mesmo.

1.2 Estrutura do Documento

A dissertação a realizar passará por um estudo sobre o estado da arte, englobando alguns protocolos de comunicação utilizados por MOMs, alguns dos sistemas anteriormente referidos, estudos comparativos sobre os mesmos e ainda trabalhos relacionados com este, sendo que este estudo servirá como base teórica para o Middleware a desenvolver.

Na fase sobre o estado da arte serão abordados protocolos como Advanced Message Queue Protocol (AMQP), Message Queue Telemetry Transport (MQTT), Simple (Streaming) Text Orientated Messaging Protocol (STOMP) e Extensible Messaging and Presence Protocol (XMPP), e a API Java Message Service (JMS), de seguida serão apresentados os sistemas de queues já anteriormente referidos (RabbitMQ, ActiveMQ 5, Kafka e ZeroMQ).

Após o estudo sobre o estado da arte será feita uma descrição do problema atual, sendo que neste serão

descritos os problemas que existem atualmente com a utilização em específico de um sistema de queues.

De seguida será elaborada a conceção do sistema a desenvolver e posteriormente a sua implementação, sendo descritas as decisões tomadas e respetivas consequências das mesmas.

Terminada a implementação será realizado um caso de estudo que demonstre as capacidades e funcionalidades do Middleware desenvolvido, sendo também realizados testes de desempenho sobre este.

Por último serão apresentadas as conclusões obtidas durante o desenvolvimento e teste do Middleware desenvolvido, bem como o respetivo trabalho futuro a realizar por forma a melhorar o Middleware.

2 Estado da Arte

No presente capítulo serão recolhidos e estudados vários trabalhos e projetos relevantes para esta dissertação, essencialmente trabalhos relacionados com MOMs. Estes são os sistemas intermediários já referidos anteriormente e são responsáveis pela entrega assíncrona de mensagens entre aplicações, daí o nome *Message-Oriented Middleware*, pois estes são orientados a mensagens.

Os diversos trabalhos e projetos aqui detalhados serviram como base para a conceção e desenvolvimento do Middleware.

Inicialmente serão abordados vários tipos de protocolos e APIs existentes na área de comunicação. De seguida serão apresentadas algumas tecnologias que serão utilizadas nesta dissertação. Após uma análise individual de cada sistema de queues serão apresentados alguns resultados de algumas comparações entre alguns destes sistemas na Secção 2.3, sendo que por último será descrito um trabalho relacionado com o Middleware a desenvolver.

2.1 Protocolos e APIs

A presente secção aborda alguns protocolos e APIs que são utilizados para comunicação entre aplicações, serviços, dispositivos, entre outros. Serão abordados os protocolos AMQP, MQTT, STOMP e XMPP, e a API JMS.

2.1.1 Advanced Message Queue Protocol

O AMQP surgiu para ser uma norma no que toca à conceção e implementação de sistemas de queues, permitindo diferentes plataformas em diferentes linguagens comunicar. Este foi desenvolvido pelas necessidades de um protocolo de alta fiabilidade e escalabilidade. Existem duas versões do AMQP que tiveram uma maior relevância, são estas a versão 0.9.1 (versão que o RabbitMQ implementa) e a versão 1.0.0 pertencente à Organization for the Advanced Structured Information Standards (OASIS)².

AMQP 0.9.1 apresenta uma arquitetura Client/Broker ou Client/Server permitindo assim uma abstração Publisher/Subscriber (Pub/Sub) ou Request/Response (Req/Res). O tamanho das mensagens enviadas sobre este protocolo tem um cabeçalho fixo de 8 bytes e um tamanho de *payload* indefinido ou negociável [Nai17].

Este protocolo apresenta três níveis de Quality of Service (QoS), QoS 0 em que não existe confirmação de receção da mensagem, QoS 1 em que é garantida a entrega da mensagem e o QoS 2 em que é garantida a entrega da mensagem sem duplicados.

Como protocolo de transporte este utiliza Transmission Control Protocol (TCP)³. Apresenta também vários protocolos para segurança como Transport Layer Security (TLS)⁴/Secure Sockets Layer (SSL)⁵, Internet Protocol

²OASIS: <https://www.oasis-open.org/>

³TCP: <https://tools.ietf.org/html/rfc793>

⁴TLS: <https://tools.ietf.org/html/rfc8446>

⁵SSL: <https://tools.ietf.org/html/rfc6101>

Security (IPSec)⁶ e Simple Authentication and Security Layer (SASL)⁷ [Nai17].

Dentro dos brokers AMQP existe pelo menos um *Exchange* e uma ou mais queues, tal como ilustrado na Figura 3, baseada em [Dos14].

As queues são responsáveis por armazenar as mensagens recebidas até serem entregues ao destinatário e o *Exchange* é responsável por receber as mensagens enviadas e decidir para quais queues enviar a mensagem.

A decisão sobre a qual queue entregar uma mensagem passa por analisar os vários campos da mesma, sendo que normalmente o campo que decide a rota que a mensagem deve seguir é o *routing key* que no caso de a comunicação ser Req/Res pode ser o nome da queue (caso este não tenha sido definido) e no caso de a comunicação ser Pub/Sub é o nome do tópico. Quando um consumidor liga a sua queue com um *Exchange* do tipo Pub/Sub este tem de especificar um tópico que será a sua subscrição, ou seja, sempre que o *Exchange* receber uma mensagem cujo o *routing key* coincida com a sua subscrição (tópico) a mensagem é enviada para a queue.

O tipo de comunicação Pub/Sub nem sempre necessita de um tópico, sendo que isto acontece porque existe um tipo de configuração para o *Exchange* em que este entrega uma mensagem a todas queues que estejam ligadas a ele e este tipo de *Exchange* designa-se por Fanout.

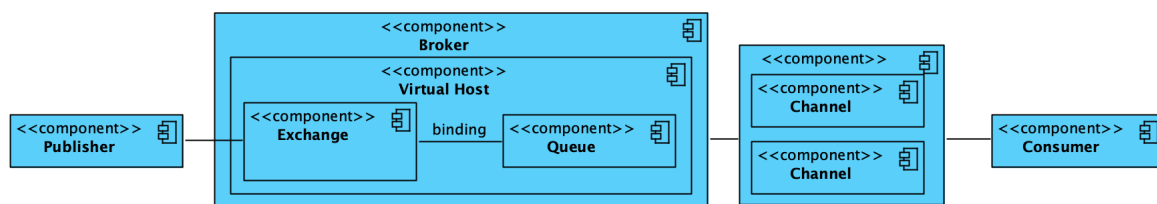


Figura 3: Arquitetura AMQP 0.9.1.

O AMQP 1.0.0 é definido como um protocolo de comunicação Peer-to-Peer (P2P), sendo que nesta versão cada entidade (*Consumer*, *Producer*, outros) está contida num *Container* e comunica através de *Links* unidirecionais, porém é possível criar dois *Links* com direções opostas de forma a gerar um *Link* bidirecional.

Como protocolo de transporte este, tal como a versão anterior, utiliza TCP e apresenta dois protocolos para segurança, o TLS e o SASL [Wal19].

Na Figura 3 é possível verificar a arquitetura do protocolo na versão 0.9.1 e uma das diferenças na arquitetura da versão 1.0.0 é que o *Exchange* deixou de existir [Dos14].

Apesar das diferenças e de a versão 1.0.0 ser quase uma década mais recente, a versão 0.9.1 continua a ser a mais utilizada [Wal19], pois as diferenças são de tal ordem que a versão 1.0.0 é considerada um protocolo diferente da versão 0.9.1 [Dos14].

⁶IPSec: <https://tools.ietf.org/html/rfc6071>

⁷SASL: <https://tools.ietf.org/html/rfc4422>

2.1.2 Java Message Service

Ao contrário do AMQP o JMS é uma API e não um protocolo, isto é, qualquer MOM que seja implementado através deste terá de utilizar a mesma interface, pois tem de seguir a API fornecida pelo JMS.

O JMS possui duas versões principais, a versão 1.1 [RMC09] e a versão 2.0 [Dea13]. As duas versões do JMS apresentam diferentes APIs, sendo estas ilustradas nas Figuras 4 e 5, correspondendo a versão 1.1 à Figura 4 e a versão 2.0 à Figura 5, baseadas em [Dev17].

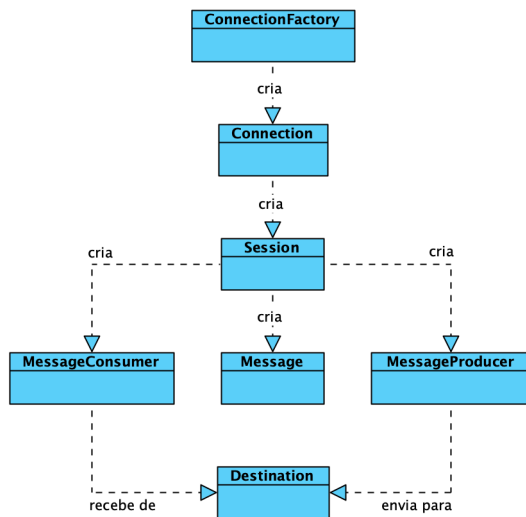


Figura 4: API JMS 1.1.

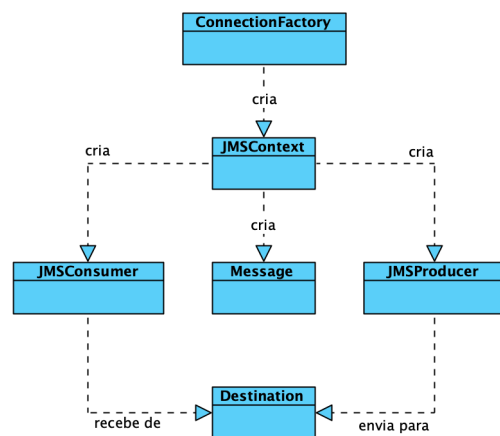


Figura 5: API JMS 2.0.

As principais diferenças que se podem observar da versão 1.1 para a versão 2.0 são a renomeação das classes *MessageConsumer* e *MessageProducer* para *JMSConsumer* e *JMSProducer*, respetivamente, e a junção das classes *Connection* e *Session* numa só classe denominada por *JMSContext*. Estas diferenças não só simplificaram o código, já que este fica menos verboso, como permitem uma melhor interpretação das classes através da nova nomenclatura.

Esta API funciona tanto para comunicação Point-to-Point (PTP) como Pub/Sub e as Figuras 4 e 5, baseadas em [Dev17], apenas ilustram as interfaces com que se interage diretamente. Nas Figuras 6 e 7 são ilustradas as classes que implementam as interfaces de forma a ser possível utilizar os dois tipos de comunicação suportados.

As classes *Topic*, *TopicConnectionFactory*, *TopicSession*, *TopicConnection*, *TopicPublisher* e *TopicSubscriber* são utilizadas para o Pub/Sub na versão 1.1, sendo que na versão 2.0 com a junção das interfaces *Session* e *Connection* a classe *TopicSession* deixou de ser utilizada externamente.

As classes *Queue*, *QueueConnectionFactory*, *QueueSession*, *QueueReceiver*, *QueueSender* e *QueueConnection* são utilizadas para o PTP na versão 1.1. Na versão 2.0 com a junção das interfaces *Session* e *Connection* a classe *QueueSession* deixou de ser utilizada externamente.

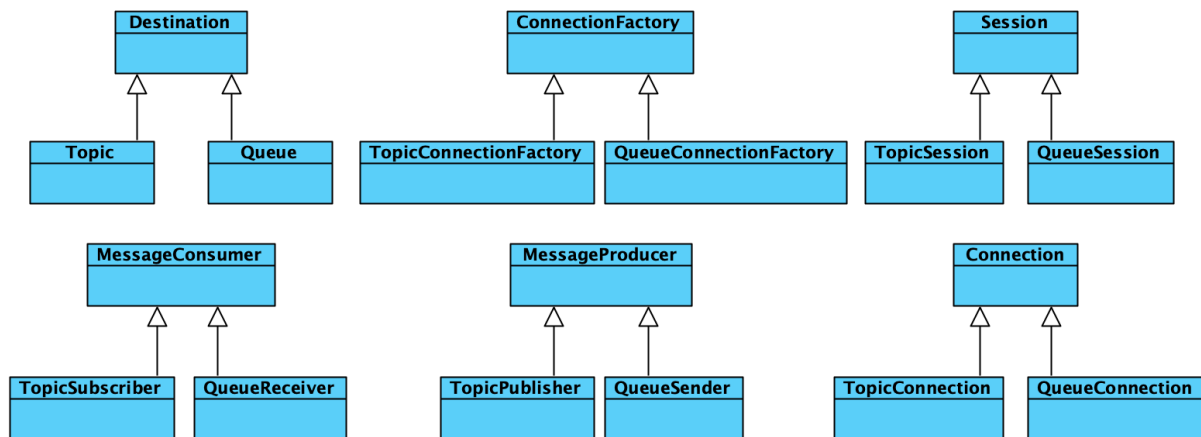


Figura 6: PTP e Pub/Sub no JMS 1.1.

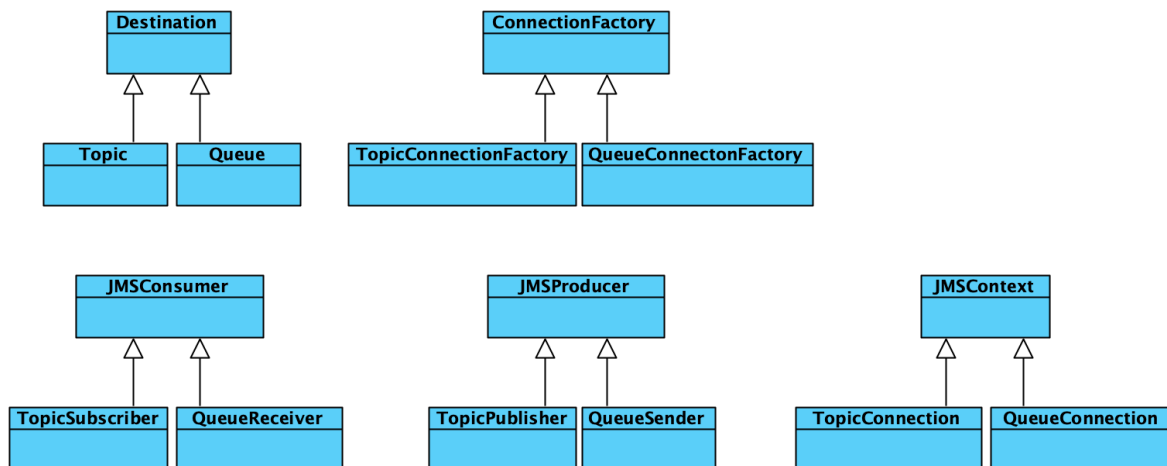


Figura 7: PTP e Pub/Sub no JMS 2.0.

Para além das alterações arquiteturais foi adicionado o *Delivery Delay* que consiste no cliente (produtor) que cria uma mensagem poder especificar um tempo para a mensagem ficar guardada antes de ser entregue a um outro cliente (consumidor).

2.1.3 Message Queue Telemetry Transport

O MQTT é um protocolo de comunicação maioritariamente utilizado na área de Internet of Things (IoT). Este é utilizado para a comunicação Pub/Sub e é um protocolo leve, pois este tem um cabeçalho fixo de 2 bytes o que ocupa muito pouca largura de banda [KS16], sendo que isto se traduz num *overhead* muito reduzido.

Este funciona sobre o protocolo TCP e o *payload* máximo das mensagens enviadas através do MQTT é de 256MB [KS16].

Além de um protocolo leve o MQTT é também um protocolo muito simples, apresentando apenas 14 métodos [KS16], referidos de seguida:

- CONNECT – pedido de conexão do cliente ao servidor.

- CONNACK – resposta do servidor ao pedido de conexão do cliente.
- PUBLISH – publicar uma mensagem.
- PUBACK – resposta do servidor à receção da mensagem (apenas quando QoS 1).
- PUBREC – resposta do servidor à receção da mensagem (apenas quando QoS 2).
- PUBREL – resposta do cliente ao PUBREC (apenas quando QoS 2).
- PUBCOMP – resposta do servidor ao PUBREL (apenas quando QoS 2).
- SUBSCRIBE – subscrição num tópico.
- SUBACK – resposta do servidor ao pedido de SUBSCRIBE do cliente.
- UNSUBSCRIBE – cancelar subscrição num tópico.
- UNSUBACK – resposta do servidor ao pedido de UNSUBSCRIBE do cliente.
- PINGREQ – PING enviado pelo cliente ao servidor para manter a conexão aberta.
- PINGRESP – resposta do servidor ao PING do cliente.
- DISCONNECT – fechar a conexão, enviado do cliente para o servidor.

Este protocolo oferece três níveis de QoS e na Figura 8, retirada de [KS16], pode observar-se as diferenças destes vários níveis, sendo o nível 0 o menos robusto em que o broker que implementa este protocolo não informa o cliente sobre a receção da sua mensagem e o nível 2 o mais robusto, em que é realizado um *4-way handshaking* [SM17]. Devido aos diferentes níveis de QoS o protocolo oferece retransmissão de mensagens, por forma a garantir que esta foi entregue (QoS 1 e QoS 2). Com o nível 1 é possível a entrega de mensagens replicadas por causa de retransmissões, enquanto que com o nível 2 o mesmo já não acontece, porque o protocolo garante que apenas uma é entregue ao recetor.

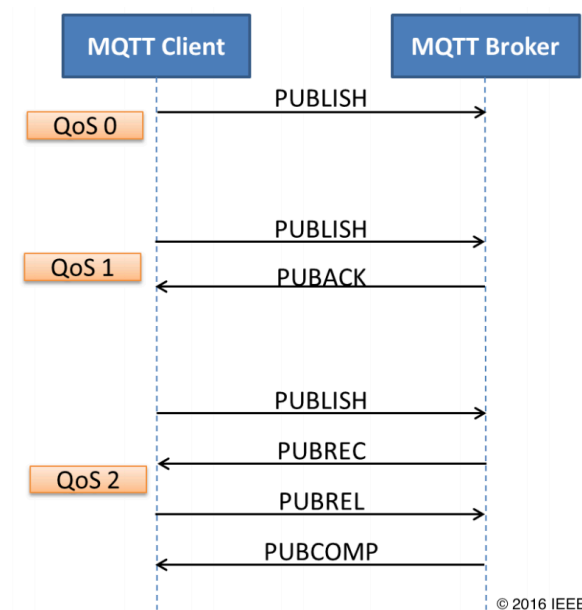


Figura 8: Mensagens enviadas nos diferentes níveis do QoS do MQTT.

Este protocolo oferece autenticação e autorização através de *username* e *password*, mas alguns brokers que implementam o MQTT acrescentam uma camada de segurança baseada no TLS/SSL.

O MQTT, disponibilizado pela IBM⁸ em 1999, possui várias versões, mas as mais relevantes atualmente são a versão mais recente do protocolo, 5.0 lançada em 2019 e pertencente à OASIS, e a versão 3.1.1 que atualmente é uma norma da International Organization Standardization (ISO)⁹, lançada em 2015.

Existe ainda uma outra versão do MQTT que é a Message Queue Telemetry Transport for Sensor Network (MQTT-SN)¹⁰ que funciona sobre o protocolo User Datagram Protocol (UDP)¹¹.

São exemplos de MOMs que utilizam este protocolo para comunicação o HiveMQ¹² e CloudMQTT¹³.

2.1.4 Simple (Streaming) Text Orientated Messaging Protocol

O STOMP é um protocolo de comunicação criado para linguagens de programação *scripting* tais como Perl, Ruby, Python, entre outras, é também um protocolo a pensar na interoperabilidade [EB15] e tal como o MQTT é um protocolo muito leve [See13].

Ao contrário do AMQP e MQTT este é um protocolo baseado em texto (text-based), sendo que isto faz com que seja muito mais verboso, mas por outro lado torna mais fácil a sua leitura e *debugging* [See13]. O STOMP é comparado ao Hyper Text Transport Protocol (HTTP)¹⁴ pela sua simplicidade e pela sua estrutura nas mensagens.

Este protocolo funciona sobre WebSockets¹⁵ [EB15], que por sua vez funciona sobre TCP.

Tal como o MQTT este é um protocolo bastante simples apresentando apenas 14 métodos, 10 dos quais são enviados pelos clientes STOMP para o servidor e os restantes 4 métodos são enviados pelo servidor STOMP aos clientes [EB15; ST012].

De seguida são ilustrados os métodos dos clientes deste protocolo:

- ABORT – efetua roll back à transação que estava em progresso.
- ACK – confirmação ao servidor de que a mensagem foi consumida com sucesso.
- BEGIN – inicia uma transação, utilizado para o cliente produtor receber a informação de que a mensagem foi consumida.
- COMMIT – confirma a transação, terminando a mesma.
- CONNECT – enviada ao servidor para iniciar a conexão.
- DISCONNECT – termina a conexão com o servidor.
- NACK – contrário do ACK, diz ao servidor que a mensagem não foi consumida.

⁸IBM: <https://www.ibm.com/>

⁹ISO: <https://www.iso.org/home.html>

¹⁰MQTT-SN: https://www.oasis-open.org/committees/document.php?document_id=66091&wg_abbrev=mqtt

¹¹UDP: <https://tools.ietf.org/html/rfc768>

¹²HiveMQ: <https://www.hivemq.com/>

¹³CloudMQTT: <https://www.cloudmqtt.com/>

¹⁴HTTP: <https://tools.ietf.org/html/rfc7540>

¹⁵WebSocket: <https://tools.ietf.org/html/rfc6455>

- SEND – envia uma mensagem.
- SUBSCRIBE – regista o cliente para receber mensagens de um certo destino.
- UNSUBSCRIBE – remove o registo do cliente para receber as mensagens de um certo destino.

Os métodos do servidor são os seguintes:

- CONNECTED – enviada em resposta ao CONNECT.
- ERROR – enviada ao cliente quando algo corre mal.
- MESSAGE – enviado aos clientes quando recebe uma mensagem na qual estão subscritos.
- RECEIPT – enviada ao cliente que produziu a mensagem quando esta é entregue ao destino, apenas é enviada caso o cliente produtor tenha pedido.

Além de um protocolo simples este suporta a troca de mensagens bilateral, isto é, um cliente produtor pode também ser consumidor. Suporta também Pub/Sub de 1 para N [CMS16].

Sendo este um protocolo baseado em texto o formato das mensagens é o apresentado na Figura 9, retirada de [STO12]¹⁶.

```
COMMAND
header1:value1
header2:value2

Body^@
```

Figura 9: Estrutura mensagem STOMP.

Na Figura 9 é possível observar o formato das mensagens enviadas pelo STOMP, sendo que esta é constituída por um comando (COMMAND), cabeçalhos (headers) e corpo (Body). Nas mensagens deste protocolo é utilizado o End-of-Line (EOL) para determinar o final de uma linha da mesma.

O protocolo STOMP possui atualmente 3 versões, a 1.0, 1.1 e 1.2. Quando o protocolo evoluiu para a versão 1.1 foi adicionado os *heartbits* para detetar uma desconexão fiável por parte do cliente e servidor, foi também adicionado a negociação para interoperabilidade para clientes e servidores com diferentes versões do protocolo conseguirem comunicar e por último adicionado o método NACK apresentado anteriormente.

Da versão 1.1 para a versão 1.2 não foram adicionadas novas funcionalidades, mas sim feita uma melhoria por forma a clarificar alguns pontos da versão anterior. Porém, com esta atualização foram inseridas duas incompatibilidades entre as versões, pois as mensagens de ACK tem agora um header específico, não reconhecido pelas versões anteriores, e é possível terminar as linhas com o 13 octet (0x0D) mais o EOL, sendo o 13 octet opcional, mas apenas compreendido pela versão 1.2 [STO12].

¹⁶Sob licença CC BY (<https://creativecommons.org/licenses/by/3.0/>).

São exemplos de MOMs que implementam este protocolo o CoIlMQ¹⁷ e o Xtomp¹⁸.

2.1.5 Extensible Messaging and Presence Protocol

O XMPP é um protocolo de comunicação baseado em texto, tal como o STOMP pelo que também tem um grande *overhead* de mensagem, sendo que este é baseado no protocolo Extensible Markup Language (XML)¹⁹. Este utiliza o TCP como protocolo de transporte e o TLS como protocolo de segurança.

Este protocolo tal como o seu nome indica é extensível, isto é, foi desenhado e concebido por forma a que possam ser criadas várias extensões para adicionar ao protocolo base, como por exemplo a extensão para Pub/Sub (XEP-0060)²⁰, pois o protocolo base é um protocolo PTP.

Por forma a identificar todos os clientes e respetivos dispositivos, cada um tem o seu Jabber ID (JID). Porém os formatos para identificação de clientes e de dispositivos é diferente, sendo o JID dos clientes denominado de Bare JID, que consiste no seguinte formato <xmpp_user>@<xmpp_server>, e o dos dispositivos denominado de Full JID, que consiste no seguinte formato <xmpp_user>@<xmpp_server>/ <resource>, em que o <xmpp_user> é nome do cliente, o <xmpp_server> é o nome do servidor e o <resource> é o dispositivo do cliente [Sch+14].

Na Figura 10, retirada de [Alv+18]²¹, pode observar-se alguns componentes do XMPP. Nesta é possível observar que através do XMPP Gateway é possível comunicar com outras tecnologias que não utilizem o protocolo XMPP, isto é uma vantagem relativamente ao protocolo Instant Messaging (IM), que ao contrário do XMPP não consegue comunicar com outras tecnologias [LLZ12].

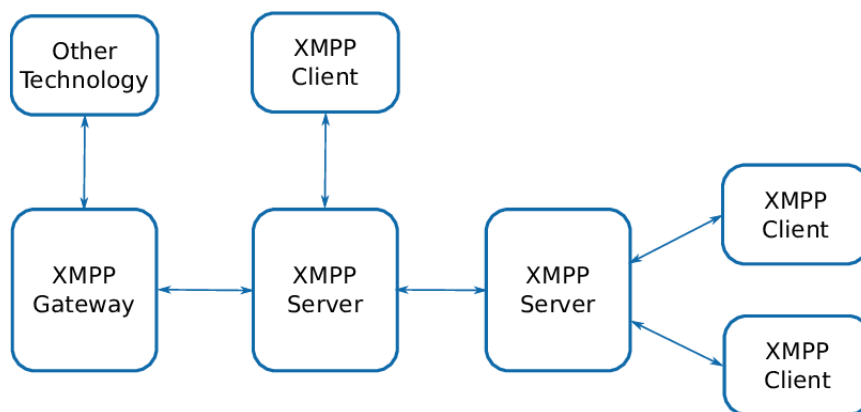


Figura 10: Fluxo de comunicação entre nodos XMPP.

O protocolo *Core* baseia-se em 3 tipos de mensagens, *XML Stream*, *XML Stanza* e *Stream Authentication* [LLZ12]. As mensagens do tipo *Stream Authentication*, tal como o próprio nome refere servem para autenticação. As mensagens *XML Stream* são utilizadas para comunicação entre quaisquer dois pontos utilizando as tags <stream>

¹⁷CoIlMQ: <https://pypi.org/project/CoIlMQ/>

¹⁸Xtomp: <http://xtomp.tp23.org/>

¹⁹XML: <https://tools.ietf.org/html/rfc3076>

²⁰XEP-0060: <https://xmpp.org/extensions/xep-0060.html>

²¹Sob licença CC BY (<https://creativecommons.org/licenses/by/4.0/>).

e `</stream>` para iniciar e terminar uma mensagem respetivamente, sendo o primeiro nível de hierarquia destas mensagens são as *XML Stanzas*, que podem ser do tipo `<message>`, `<presence>` ou `<iq>` [LLZ12].

As XML Stanzas tem alguns atributos muito comuns, o *to* que identifica quem será o recetor da Stanza, o *from* que identifica quem enviou, o *id* para facilitar a identificação de Req/Res, isto é, o nodo produtor do Request saberá exatamente qual é o Response ao seu Request, o *type* identifica o conteúdo e o contexto da Stanza e o *xml:lang* que serve para informar qual a linguagem a utilizar de forma a ser perceptível para seres humanos. Estes valores tem de seguir o Request for Comment (RFC) sobre *Tags for Identifying Languages* (RFC 5646)²².

O Stanza `<iq>` é normalmente utilizado para Req/Res de forma a ser trocada a informação relativa aos utilizadores [LLZ12], o Stanza `<message>` é utilizado para enviar informação de uma entidade para outra e o Stanza `<presence>` é utilizado para o Pub/Sub, sendo que estes não possuem o atributo *to* pois são entregues ao servidor XMPP e este saberá a quem entregar a mensagem.

O XMPP ou *Jabber*, como era denominado inicialmente quando foi introduzido pela Jabber Software Foundation, mais tarde XMPP Standards Foundation, foi inicializado em 1998 e tornado público em 2000, sendo que este só se passou a denominar de XMPP em 2004 quando este foi aceite pela Internet Engineering Task Force (IETF)²³ através dos RFCs 3920²⁴, 3921²⁵, 3922²⁶ e 3923²⁷.

Atualmente os RFCs 3920 e 3921 já se tornaram obsoletos, tendo sido substituídos pelos RFCs 6120²⁸ e 6121²⁹, respetivamente, e foi ainda acrescentado um novo (RFC 6122)³⁰ que especifica o formato do endereço do XMPP.

2.2 Tecnologias

De seguida iremos rever os sistemas de queues já anteriormente referidos. Serão abordadas as principais funcionalidades/caraterísticas, vantagens, desvantagens e arquitetura. Todos estes pontos serão relevantes para a forma como o Middleware será desenvolvido, pois este irá aproveitar os pontos fortes destes e tentar melhorar os pontos fracos.

2.2.1 RabbitMQ

O RabbitMQ é um sistema de queues que segue o protocolo AMQP 0.9.1, ou seja, foi desenvolvido segundo este protocolo, e sendo este um sistema intermediário externo existe um middleware entre os componentes que estão a comunicar através deste.

²²RFC 5646: <https://tools.ietf.org/html/rfc5646>

²³IETF: <https://www.ietf.org/>

²⁴RFC 3920: <https://tools.ietf.org/html/rfc3920>

²⁵RFC 3921: <https://tools.ietf.org/html/rfc3921>

²⁶RFC 3922: <https://tools.ietf.org/html/rfc3922>

²⁷RFC 3923: <https://tools.ietf.org/html/rfc3923>

²⁸RFC 6120: <https://tools.ietf.org/html/rfc6120>

²⁹RFC 6121: <https://tools.ietf.org/html/rfc6121>

³⁰RFC 6122: <https://tools.ietf.org/html/rfc6122>

Apesar de este ter sido desenvolvido segundo o protocolo AMQP este apresenta suporte para protocolos como MQTT e STOMP referidos anteriormente [DE17; Rab20].

Este sistema foi desenvolvido na linguagem de programação Erlang e neste momento é também disponibilizado para Java e .NET Framework. Atualmente já existem outros programadores que disponibilizam este sistema noutras linguagens, como por exemplo o Pika³¹ para Python.

Uma das principais características/funcionalidades do RabbitMQ é a forma como é possível criar topologias com vários brokers. Na Figura 11, baseada em [Dos14], são ilustradas algumas topologias de brokers RabbitMQ.

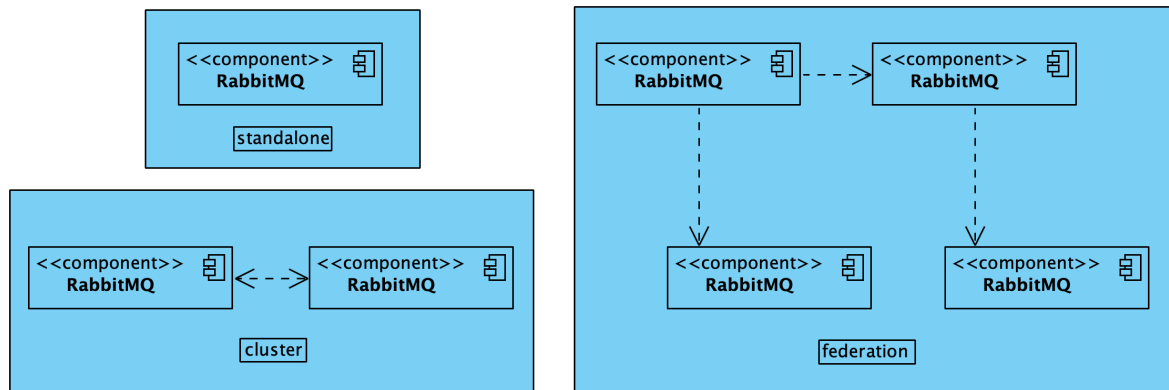


Figura 11: Topologias RabbitMQ.

A criação de topologias complexas com vários brokers interligados permite balanceamento de carga e tolerância a falhas.

Uma vantagem na utilização deste sistema é a persistência das mensagens, enquanto estas estão no sistema são persistidas pela base de dados embutida do Erlang, denominada de Mnesia³². Com a persistência destas mesmo que o sistema reinicie as mensagens já persistidas não se perdem. Existe a possibilidade de configurar o RabbitMQ de forma a não utilizar o disco para persistir as mensagens, ficando estas em memória [DE17].

Este sistema permite também a criação de um limite de mensagens por queue no caso de mensagens poderem ser apagadas de forma a não sobrecarregar o sistema se estas não estiverem a ser entregues. Outra forma de apagar mensagens das queues sem colocar um limite de mensagens é atribuir um Time to Live (TTL) a cada mensagem, sendo que se este TTL for atingido e a mensagem ainda estiver na queue esta será apagada [DE17].

No caso de o sistema estar a ser utilizado para Pub/Sub este mantém registo de todos os consumidores que consumiram uma mensagem e isto permite que uma mensagem seja apagada quando todos os consumidores a leram [DE17].

Um problema que este sistema apresenta é a pouca escalabilidade quando vários utilizadores utilizam o

³¹Pika: <https://pypi.org/project/pika/>

³²Mnesia: <https://erlang.org/doc/man/mnesia.html>

mesmo [Dos14]. Na Figura 12, retirada de [EA15], é apresentado um gráfico que ilustra o número de mensagens por segundo processadas pelo sistema RabbitMQ (linha azul do gráfico) em função do número de utilizadores. Apesar de este se encontrar em escala logarítmica é possível verificar que o número de mensagens por segundo processadas com 2 utilizadores é praticamente o dobro de quando utilizado apenas 1 utilizador, porém a diferença no número de mensagens processadas entre 2 utilizadores e 10 é praticamente constante.

O número de utilizadores também é apresentado em escala logarítmica, pelo que o 0 no eixo horizontal corresponde a 1 utilizador e o 1 corresponde a 10 utilizadores.

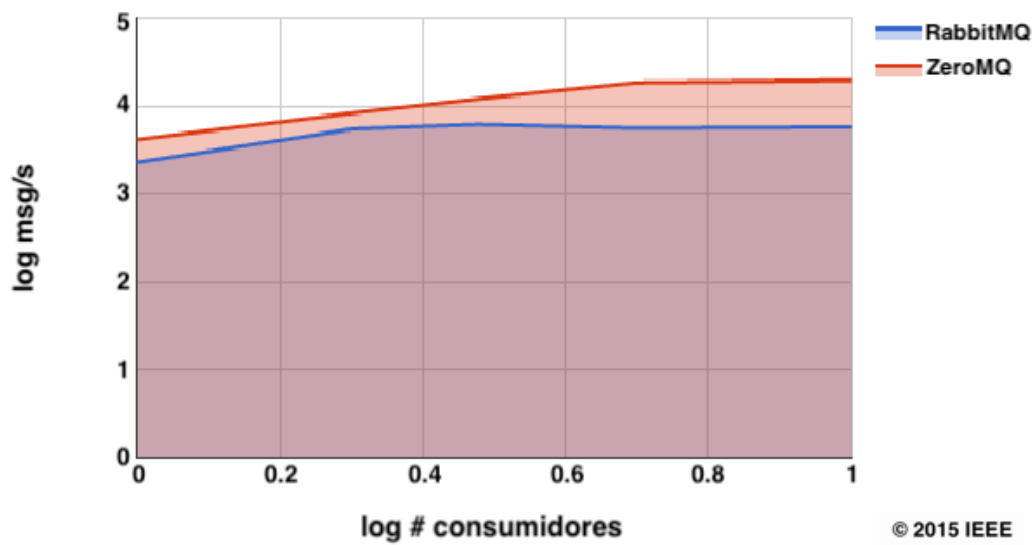


Figura 12: Número de mensagens por segundo (msg/s) processadas pelo sistema RabbitMQ em função do número de utilizadores.

O RabbitMQ foi desenvolvido inicialmente pela Rabbit Technologies Ltd. em 2007, sendo a versão 1.7.2 a última versão lançada pela empresa antes de ser adquirida pela SpringSource em 2010. Atualmente a versão mais recente deste sistema é 3.9.9.

Tal como referido o RabbitMQ implementa o AMQP 0.9.1, porém este pode suportar o AMQP 1.0.0 através de um plugin [DE17; Rab20].

2.2.2 ActiveMQ

O ActiveMQ é um sistema intermediário externo tal como o RabbitMQ e surgiu como uma necessidade para o Apache Geronimo³³, pois este necessitava de um sistema de queues baseado na API JMS e os existentes no mercado estavam todos sob licenças pagas pelo que os programadores do Apache Geronimo começaram a desenvolver o ActiveMQ.

Este sistema possui duas versões, sendo a principal diferença as versões da API JMS, o ActiveMQ 5 implementa o JMS 1.1, enquanto que o ActiveMQ Artemis implementa o JMS 2.0 e apresenta compatibilidade para o JMS

³³Apache Geronimo: <https://geronimo.apache.org/>

1.1. Parte do código do ActiveMQ Artemis pertencia ao HornetQ³⁴ da Red Hat, pelo que esta tem controlo sobre parte deste projeto.

Sendo este um sistema intermediário externo tem algumas funcionalidades idênticas ao RabbitMQ, tais como a persistência de mensagens (estas são persistidas pelo KahaDB³⁵) e as topologias complexas de vários nodos do mesmo por forma a atingir uma maior escalabilidade [SBD11; lon15].

Na Figura 13, baseada em [Ahu21], está representada a separação entre queues e topics realizada pelo ActiveMQ 5, sendo que uma vantagem desta separação é que permite a criação de topics e de queues com o mesmo nome sem gerar conflitos. Uma mensagem persistida numa queue é consumida apenas uma vez enquanto que mensagens persistidas em topics podem ser consumidas por vários consumidores.

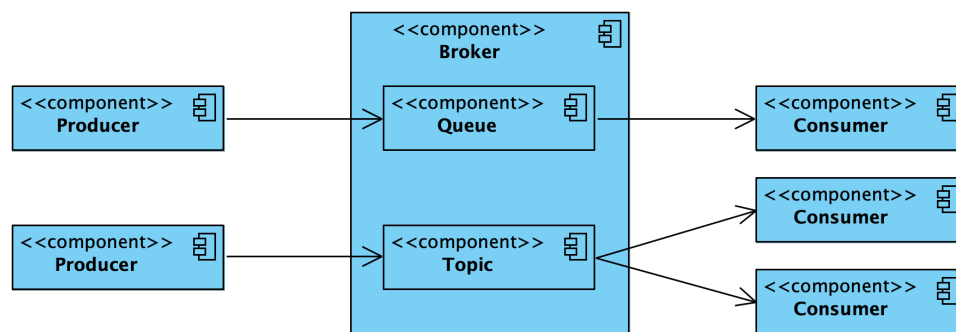


Figura 13: Arquitetura ActiveMQ Broker.

O ActiveMQ foi desenvolvido em Java, mas apresenta também soluções para clientes em C/C++, .NET, Python, entre outros, sendo que o sistema (servidor) funciona sempre na Java VM³⁶. Este foi desenvolvido segundo a API JMS, mas apresenta suporte para muitos outros protocolos tais como o AMQP 1.0.0 [Act19b], MQTT, XMPP, STOMP, entre outros [SBD11], sendo que no STOMP é apenas suportada a versão 1.0 e 1.1 para o JMS 1.1 [Act19c], o suporte para a versão 1.2 do STOMP apenas existe no JMS 2.0 [Act].

Uma funcionalidade deste sistema é o *delivery delay* (atraso na entrega) de mensagens, em que o produtor especifica o tempo que a mensagem tem de ficar armazenada antes de ser reencaminhada para o consumidor, sendo que este atraso (*delay*) pode ser especificado de diversas formas. Ora esta funcionalidade tem 4 propriedades, o atraso (*delay*), o período entre repetições (*period*), o número de repetições (*repeat*) e o cronómetro (*cron*) que consiste ficar indefinidamente a produzir o mesmo efeito com o tempo especificado, todas estas podem ser utilizadas em conjunto [Act19a].

O ActiveMQ tem incorporado autenticação e autorização através de ficheiros de configuração ou do Java Authentication and Authorization Service (JAAS)³⁷ [SBD11].

³⁴HornetQ: https://docs.jboss.org/hornetq/2.4.0.Final/docs/quickstart-guide/html_single/index.html

³⁵KahaDB: <https://activemq.apache.org/kahadb>

³⁶Java VM: <https://docs.oracle.com/javase/specs/jvms/se8/html/>

³⁷JAAS: <https://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html>

O ActiveMQ a abordar nesta dissertação é o ActiveMQ 5, a versão 5.16.3 lançada em agosto de 2021 é a mais recente. Atualmente já existe o ActiveMQ Artemis que se passará a denominar por ActiveMQ 6 quando a paridade de funcionalidades entre as duas versões estiver terminada ou a um nível superior.

2.2.3 Kafka

O Kafka ou Apache Kafka é um sistema intermediário externo para Pub/Sub altamente escalável. Tal como os outros sistemas este permite a existência de um único sistema ou então topologias complexas como clusters de um único nodo ou de múltiplos nodos. Estes encontram-se ilustrados nas Figuras 14 e 15, baseadas em [Gar15].

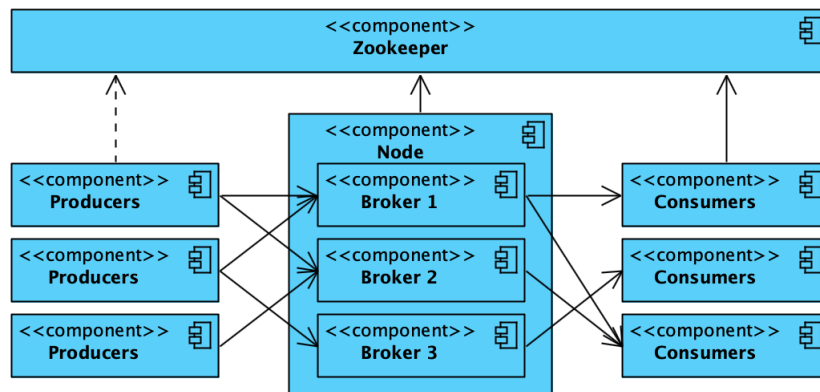


Figura 14: Topologia Kafka com apenas um nodo e múltiplos sistemas.

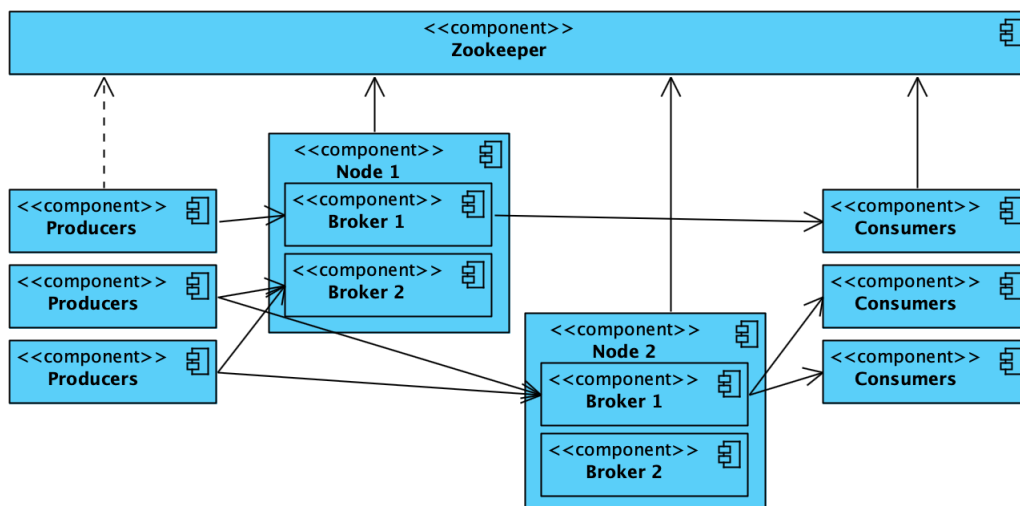


Figura 15: Topologia Kafka com múltiplos nodos com e múltiplos sistemas.

Nas Figuras 14 e 15, pode observar-se que tanto os nodos como os produtores e consumidores estão ligados ao ZooKeeper³⁸. Este é o sistema de gestão de ficheiros que o Kafka utiliza para persistir as mensagens que recebe [Gar15]. O ZooKeeper inicialmente era um subprojeto do Hadoop³⁹, mas atualmente é um projeto da Apache

³⁸ZooKeeper: <https://zookeeper.apache.org/>

³⁹Hadoop: <https://hadoop.apache.org/>

Software Foundation (ASF)⁴⁰.

As mensagens persistidas no ZooKeeper ficam armazenadas durante uma certa quantidade de tempo especificada na configuração dos tópicos, sendo que isto pode ser visto como uma vantagem ou desvantagem.

Uma vez que o Kafka não guarda informação sobre os consumidores que já leram uma mensagem este não sabe se todos os consumidores a leram, pelo que tem de ser única e exclusivamente eliminada pelo ZooKeeper quando esta atinge o tempo especificado para a persistência no tópico, a desvantagem é a mensagem ter de ficar persistida o tempo especificado em vez de ser eliminada mais cedo, a vantagem é um consumidor poder ler uma mensagem novamente se necessitar [DE17].

Este sistema foi concebido para alto débito e para ser distribuído. O sistema foi desenvolvido em Java e para as aplicações cliente existe suporte para várias linguagens como Java, .NET, Python e outros.

Uma característica deste sistema é o *real time*, isto é, a latência introduzida pelo sistema é tão baixa entre o consumir e produtor que é quase como se estivessem a comunicar diretamente, isto é uma funcionalidade crítica para os sistemas baseados em eventos [Gar15].

O Kafka utiliza um protocolo binário próprio sobre TCP e como protocolo de segurança utiliza o TLS. Em termos arquiteturais este sistema assemelha-se ao RabbitMQ e ao ActiveMQ, mas do ponto de vista do processamento de dados é comparado com o Scribe⁴¹ e o Flume⁴² [Gar15].

O Kafka foi inicialmente desenvolvido pela LinkedIn e disponibilizado em 2011. Em 2014 foi criada a empresa Confluent⁴³ para continuar o seu desenvolvimento e apesar de esta desenvolver o Kafka este é fornecido pela ASF.

A versão mais recente deste sistema é a 3.0.0 lançada em setembro de 2021, porém a versão a ser utilizada nesta dissertação é a versão 2.8.1.

2.2.4 ZeroMQ

O ZeroMQ é o único sistema de queues apresentado nesta dissertação que é um sistema intermediário interno. Este sistema funciona como uma biblioteca de sistema e tal como já foi referido anteriormente é um sistema onde as aplicações/componentes cliente ficam como um nodo do sistema de queues. Uma vez que este sistema funciona como uma biblioteca permite criar outros sistemas de queues em cima deste.

O sistema ZeroMQ suporta vários tipos de padrões para comunicação, como por exemplo o Req/Res denominado de Request/Reply (REQ/REP), o Pub/Sub, o Pipeline e o Exclusive Pair [Hin].

O padrão Exclusive Pair é igual ao Req/Res, mas com a restrição de que apenas podem estar dois nodos a comunicar, enquanto que no Req/Res podem estar dois ou mais nodos a produzir mensagens para um único nodo e vice-versa.

O Pipeline consiste na utilização de nodos PUSH e PULL, sendo que este é utilizado para a distribuição de

⁴⁰ASF: <https://www.apache.org/>

⁴¹Scribe: <https://engineering.fb.com/2019/10/07/data-infrastructure/scribe/>

⁴²Flume: <https://flume.apache.org/>

⁴³Confluent: <https://www.confluent.io/>

trabalho e recolha do mesmo. Na Figura 17, retirada de [Hin]⁴⁴, é possível observar o padrão Pipeline juntamente com o padrão Pub/Sub. Nesta as setas de traço contínuo são relativas ao padrão Pipeline, sendo as setas de traço descontínuo relativas ao padrão Pub/Sub.

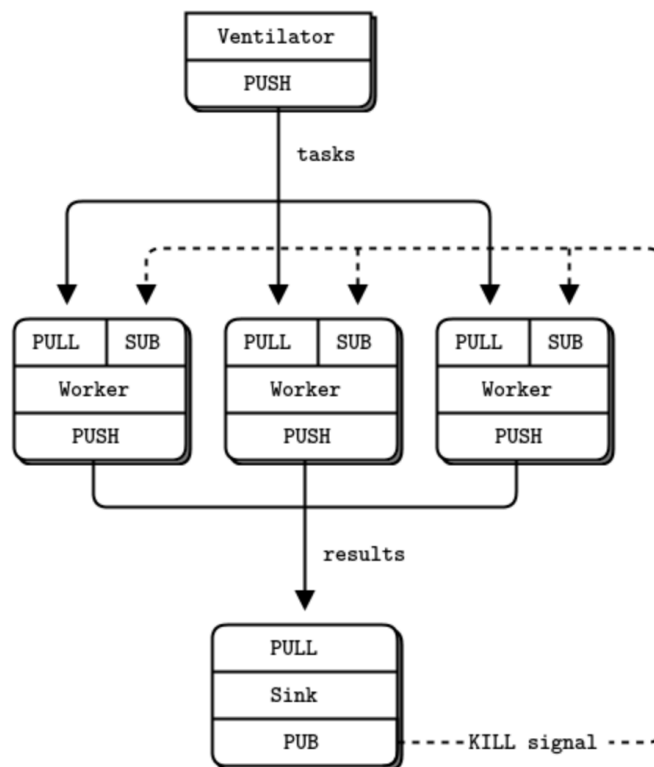


Figura 16: Padrão Pipeline.

Este é um sistema que implementa os seus próprios sockets sobre os sockets TCP, enquanto os sockets TCP são síncronos esta implementação dos sockets ZeroMQ faz com que sejam assíncronos, pelo que se uma aplicação tentar enviar uma mensagem a outra através dos sockets TCP só o consegue fazer se o destino estiver ativo. No caso da utilização dos sockets ZeroMQ não é necessária essa preocupação, pois no caso de o destino não estar disponível o socket ZeroMQ irá guardar a mensagem numa queue.

Outra funcionalidade que os sockets ZeroMQ implementam na aplicação destino é caso esta não consiga processar uma nova mensagem quando esta é recebida o socket guarda-a numa queue até a aplicação estar disponível para a processar. Atualmente estas queues geradas pelos sockets ZeroMQ tem um limite de 1000 mensagens por omissão, pelo que se este valor for atingido as mensagens que vierem a seguir podem ser descartadas dependendo do tipo de padrão no qual o ZeroMQ está a ser utilizado.

Os sockets ZeroMQ utilizam o Open Pragmatic General Multicast (OpenPGM)⁴⁵, que é uma implementação do Pragmatic General Multicast (PGM)⁴⁶, e este permite aos sockets funcionarem em *multicast*. Esta funcionalidade

⁴⁴Sob licença CC BY-SA (<https://creativecommons.org/licenses/by-sa/4.0/>).

⁴⁵OpenPGM: <https://code.google.com/archive/p/openpgm/>

⁴⁶PMG: <https://tools.ietf.org/html/rfc3208>

não existe nos sistemas de queues que utilizam o protocolo AMQP. Estes sockets permitem ainda ligações de M para N, ao contrário dos sockets TCP que apenas permitem ligações 1 para 1 [Akg13].

Além destes padrões existem o Extended Req/Res que utiliza o ROUTER e o DEALER, e o Extended Pub/Sub que utiliza o XPUB e XSUB. O ROUTER é uma adaptação do REP que torna a comunicação não bloqueante entre o REQ e o REP, além de reduzir o número de conexões do cliente REQ. O DEALER é idêntico ao REQ, sendo que este reencaminha as mensagens recebidas pelo ROUTER para os respetivos REP, reduzindo também o número de conexões dos REP, pois estes apenas recebem comunicações do DEALER.

O XPUB e o XSUB tem um comportamento idêntico ao ROUTER e DEALER, pois a função destes é reduzir o número de conexões dos PUB e SUB, fazendo com que os PUB apenas necessitem de conhecer o XSUB e com que os SUB apenas necessitem de conhecer o XPUB.

O ROUTER, DEALER, XPUB e XSUB são utilizados com a finalidade de criar um nodo que funcione como um sistema intermediário externo, em que todas as mensagens são enviadas para este nodo e depois este distribui-as pelos respetivos nodos consumidores.

A Figura 16, retirada de [Hin], ilustra o padrão Extended Req/Res.

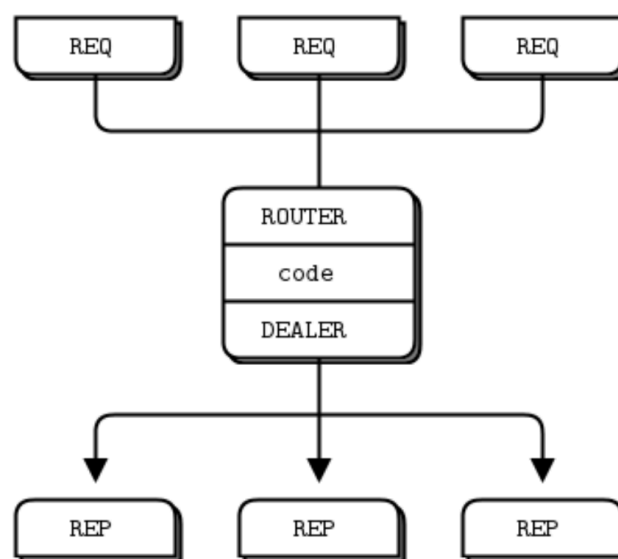


Figura 17: Padrão Extended Req/Res.

Apesar dos sockets ZeroMQ trazerem várias vantagens face aos sockets normais, o ZeroMQ tem pontos negativos também, como por exemplo a não persistência das mensagens por omissão, isto é, se um nodo (aplicação) se desligar ou reiniciar todas as mensagens armazenadas na queue desse nodo são perdidas [Akg13].

Uma particularidade deste sistema de queues em relação aos demais é a utilização de diferentes portas para comunicação TCP em vez da utilização explícita de queues, isto é, enquanto os sistemas de queues anteriores se especifica uma queue e é utilizada sempre a mesma porta de comunicação com o sistema de queues, no ZeroMQ com a falta de um sistema intermediário externo são utilizadas portas para realizar a comunicação ponto-a-ponto.

Este caso de utilização de diferentes portas é apenas para o Req/Res, pois no caso de Pub/Sub em que os consumidores se inscrevem num tópico já é possível reutilizar uma única porta para vários tópicos.

O sistema apresenta suporte para inúmeras linguagens, tais como Java, C, C++, Erlang, Python, entre outras.

O ZeroMQ apesar de ser um sistema intermediário interno permite a construção de um sistema intermediário externo, tal como o construído para o estudo realizado por N. Estrada e H. Astudillo em [EA15].

Este começou a ser desenvolvido em 2007 por Pieter Hintjens, quando este registou o domínio *zeromq.org*, juntamente com Martin Sustrik e a versão mais recente deste sistema é a 0.5.1.

2.3 Estudos Comparativos

Tal como já referido anteriormente existem vários estudos comparativos sobre os sistemas de queues referidos, sendo que estes permitem uma escolha ponderada sobre que sistema(s) utilizar dependendo das necessidades do projeto que se pretenda desenvolver. De seguida serão ilustrados alguns desses estudos realizados e os respetivos resultados.

2.3.1 RabbitMQ vs. ActiveMQ

V. M. Ionescu em [lon15] realizou um estudo comparativo entre dois sistemas intermediários externos, o ActiveMQ baseado na API JMS 1.1 e o RabbitMQ baseado no protocolo AMQP 0.9.1, ambos os sistemas foram testados nas mesmas condições.

Com este estudo foi concluído que o sistema ActiveMQ era mais rápido a processar as mensagens recebidas dos clientes do que o sistema RabbitMQ, tal como se pode observar na Tabela 1.

No que toca a entrega de mensagens a clientes o sistema RabbitMQ teve um melhor desempenho, tal como é ilustrado na Tabela 2.

Nas Tabelas 1 e 2, baseadas em [lon15], são ilustrados os valores de envio e receção de mensagens dos dois sistemas, os valores de envio do sistema ActiveMQ para a aplicação cliente na Tabela 2 são aproximações obtidas através do gráfico ilustrado no estudo, todos os outros estão em formato tabular.

Tamanho Ficheiro	58KB	148KB	450KB	1.44MB	3.97MB
RabbitMQ	0.694991	1.706642	3.663944	8.802655	27.794664
ActiveMQ	0.246679	0.715804	1.323073	5.370515	18.150352

Tabela 1: Tempo em milissegundos do envio de mensagens da aplicação cliente para o sistema.

Tamanho Ficheiro	58KB	148KB	450KB	1.44MB	3.97MB
RabbitMQ	0.003713	0.004573	0.00662	0.008245	0.012111
ActiveMQ	≈0.15	≈0.17	≈0.22	≈0.39	≈0.49

Tabela 2: Tempo em milissegundos do envio de mensagens do sistema para a aplicação cliente.

Apesar do desempenho no envio de mensagens por parte do RabbitMQ para os clientes ser de uma ordem muito superior à do ActiveMQ, não é o suficiente para compensar o desempenho na entrega de mensagens ao broker, pelo que no geral o desempenho do ActiveMQ é superior.

Neste mesmo estudo foi feita uma comparação da escalabilidade das duas tecnologias relativa ao número de clientes em simultâneo a fazer pedidos a estas. Neste foi possível verificar que para muitos clientes o RabbitMQ obtém melhores tempos de entrega de mensagens ao sistema. Estes resultados são ilustrados na Tabela 3, baseada em [lon15], para o envio de uma mensagem de 148KB.

Número de Clientes	1	3	10	20
RabbitMQ	2.24500	2.63496	5.03370	15.69899
ActiveMQ	1.00589	2.69662	11.23365	33.36614

Tabela 3: Tempo em milissegundos do envio de mensagens da aplicação cliente para o sistema com diferente número de clientes.

Posto isto será necessária especial atenção aos dois sistemas durante o desenvolvimento do Middleware para não piorar o seu desempenho geral de nenhum dos dois.

2.3.2 RabbitMQ vs. ZeroMQ

N. Estrada e H. Astudillo [EA15] realizaram um estudo de escalabilidade entre dois sistemas de queues que seguem duas abordagens diferentes, um deles o sistema RabbitMQ que é um sistema intermediário externo, o outro é o ZeroMQ, um sistema intermediário interno, sendo que nesta comparação foi criado um middleware para a utilização do ZeroMQ, fazendo assim com que a comparação passasse a ser entre dois sistemas intermediários externos.

De modo a comparação ser genuína ambos os sistemas foram testados na mesma máquina. Na Figura 18, retirada de [EA15], é possível observar-se os sistemas anteriormente descritos, sendo utilizados para o teste os sistemas que se listam ao centro (ZeroMQ broker) e à direita (RabbitMQ).

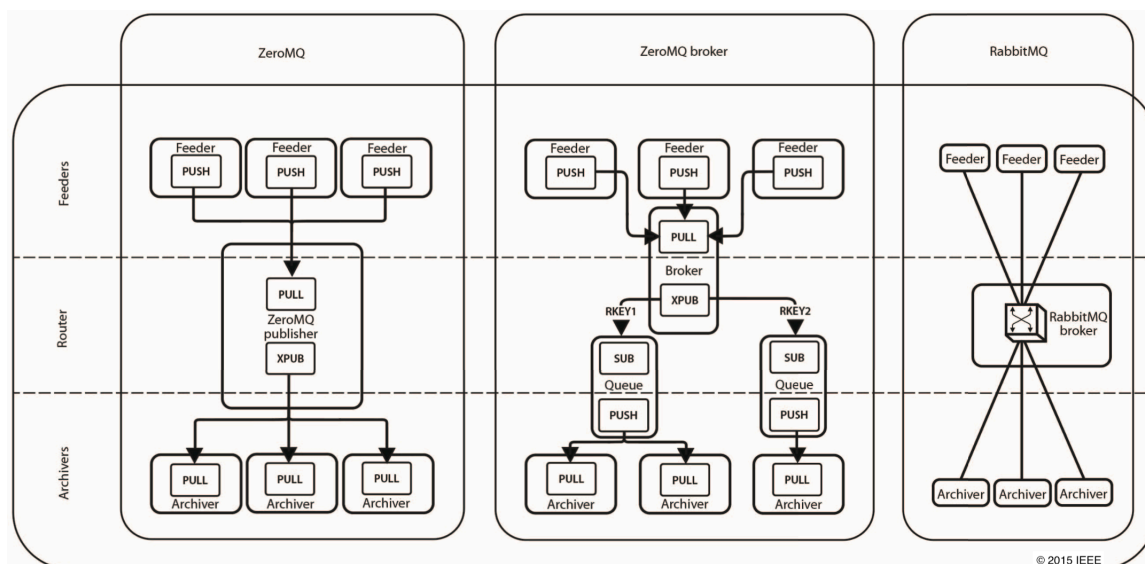


Figura 18: Topologias do cenário de teste.

Nesta comparação foram realizados os seguintes testes:

1. Determinar o número limite de operações que uma aplicação consumidora (mensagens do sistema para a aplicação) consegue fazer em disco, variando o número de mensagens por segundo.
2. Determinar o melhor rendimento para várias quantidades de clientes.
3. Determinar o melhor desempenho de cada sistema para um número variado de mensagens, usando a quantidade de clientes que obteve melhor resultado no ponto anterior.

O Teste 1 é apenas realizado para obter qual o número de operações por segundo onde são obtidos os melhores resultados para operações sobre disco, ambos os sistemas têm registos muito idênticos, cerca de 10000 operações por segundo, mas ligeiramente superior para o ZeroMQ.

Depois de determinado este valor passou-se ao Teste 2 onde se utilizou uma quantidade variada de clientes. Neste teste o ZeroMQ destacou-se do RabbitMQ obtendo uma maior quantidade de mensagens processadas por segundo para qualquer número de clientes entre 1 e 10. Sendo que para o ZeroMQ não foram obtidas melhorias no número de mensagens processadas a partir de 5 clientes para cima, o mesmo ocorreu para o RabbitMQ, mas a partir de 3 clientes, Figura 12 ilustrada na Secção 2.2.1.

Para o teste final (Teste 3) o número de clientes foi fixado em 1 produtor e 3 consumidores (3 corresponde ao valor para o qual o RabbitMQ atingiu o limite de mensagens por segundo) e o número de mensagens por segundo foi variando entre os seguintes valores 100 e 100000. No final do teste foi observado que o ZeroMQ conseguia processar cerca de 11000 mensagens por segundo, enquanto que o RabbitMQ apenas conseguiu cerca de 4500. Os resultados descritos podem ser observados na Figura 19.

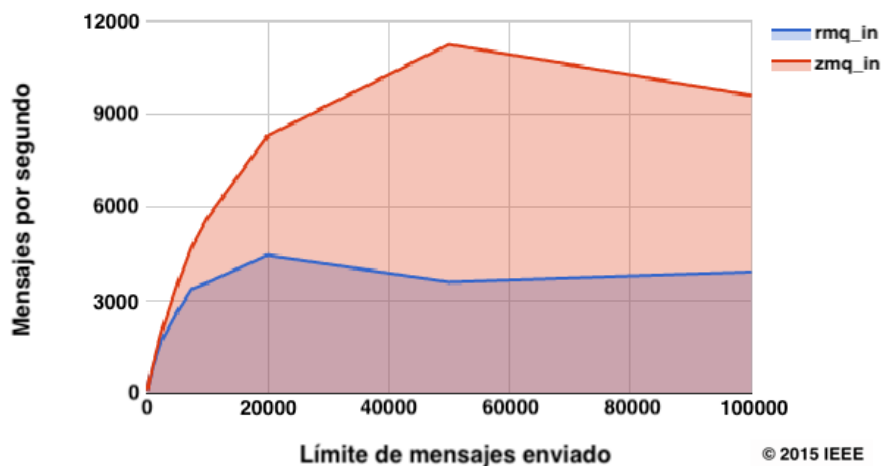


Figura 19: Número de msg/s processadas com 3 consumidores.

No final de todos os testes realizados o ZeroMQ obteve melhores resultados em todos eles, pelo que este demonstrou maior escalabilidade e desempenho. Posto isto, será dada maior atenção ao RabbitMQ para não degradar o seu desempenho e escalabilidade.

2.3.3 ActiveMQ vs. OpenMQ

A. F. Klein et al. [Kle+15] realizaram um estudo entre dois sistemas que seguem a API JMS 1.1, são estes o ActiveMQ e o OpenMQ. Este último não será abordado nesta dissertação, porém para um maior leque de comparações e para uma comparação de dois sistemas baseados na mesma API este será aqui referido. O OpenMQ referido neste trabalho é a versão que implementa o JMS 1.1, pois atualmente já existe OpenMQ 5 que implementa a versão 2.0 do JMS. Tal como nas comparações anteriores, para obter valores genuínos ambos os sistemas foram testados em condições idênticas na *cloud*.

Ambos os sistemas foram utilizados em cluster, tal como ilustrado nas Figuras 20 e 21, retiradas de [Kle+15]. Nestes as aplicações cliente comunicam com o cluster que por sua vez comunica com o sistema de ficheiros ou base de dados de forma a persistir as mensagens recebidas do cliente.

Por forma a testar o desempenho destes dois sistemas foram realizados 2 testes. No primeiro teste a aplicação envia mensagens para o cluster e ela própria lê do cluster, tal como ilustrado na Figura 22, retirada de [Kle+15]. No segundo teste a aplicação envia mensagens para o cluster e outra aplicação retira as mensagens do cluster para as colocar novamente neste, mas numa queue diferente de forma a que a primeira aplicação retire as mesmas do cluster através da nova queue, tal como ilustrado na Figura 23, retirada de [Kle+15].

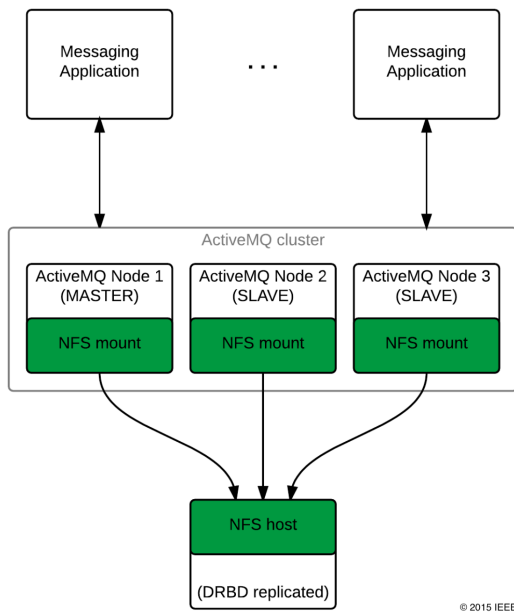


Figura 20: Fluxo de comunicação do OpenMQ em cluster utilizada.

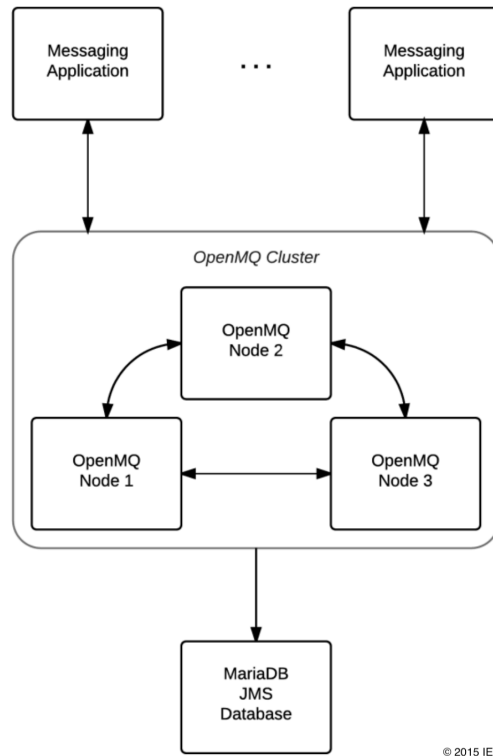


Figura 21: Fluxo de comunicação do ActiveMQ em cluster utilizada.

Nas Figuras 22 e 23 as setas a vermelho representam as mensagens enviadas das aplicações para o cluster e a verde as mensagens enviadas pelo cluster às aplicações.

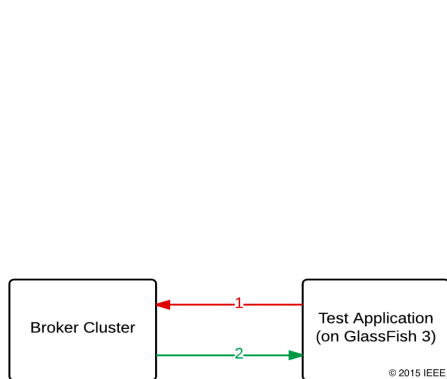


Figura 22: Cenário de teste 1.

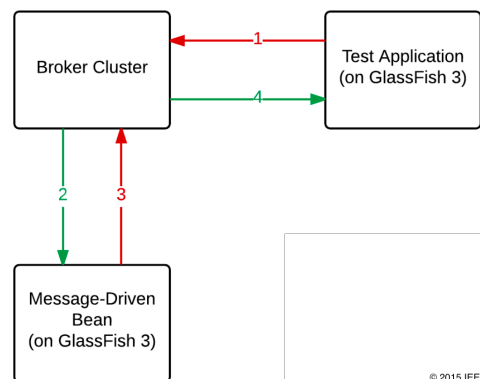


Figura 23: Cenário de teste 2.

Por forma a realizar um teste mais controlado as aplicações enviaram exatamente 2.096GB de dados em mensagens de vários tamanhos, desde 256B até 1.024MB. Nesta dissertação serão apenas ilustrados os valores obtidos para as mensagens de 256B, 2.048KB, 16.384KB, 131.072KB e 1.024MB. Ora estes parâmetros vão fazer com que existam muitas mensagens quando o tamanho de mensagem é pequeno e que existam menos mensagens quando o tamanho da mensagem for maior.

Nas Tabelas 4 e 5, baseadas em [Kle+15], são ilustrados os resultados obtidos para o teste 1 relativos ao número de mensagens por segundo (msg/s) e à quantidade de KB por segundo (KB/s), respetivamente, para os

quais não foi possível obter qualquer resultado para as mensagens de 256B no cluster do OpenMQ, pois este é muito mais lento a enviar as mensagens para os consumidores do que a processar as recebidas dos produtores, o que levou o cluster a ficar sem memória.

No estudo realizado foi calculada a quantidade de KiB por segundo (KiB/s), sendo aqui convertidos para KB/s, 1 KiB corresponde a 1.024 KB.

Tamanho da Mensagem	256B	2.048KB	16.384KB	131.072KB	1.024MB
ActiveMQ	2032.5/2032.5	1962.0/1962.1	1111.4/1111.4	235.1/235.1	30.6/30.6
OpenMQ	#N/A	313.2/118.8	245.5/114.5	99.6/77.8	19.5/19.5

Tabela 4: Mensagens enviadas/recebidas por segundo para o teste 1.

Tamanho da Mensagem	256B	2.048KB	16.384KB	131.072KB	1.024MB
ActiveMQ	520.3/520.3	4018.3/4018.3	18208.9/18207.8	30816.4/30815.0	32096.9/32096.9
OpenMQ	#N/A	641.3/243.3	4022.2/1876.8	13056.8/10191.7	20495.0/20487.9

Tabela 5: Payload enviado/recebido por segundo para o teste 1.

A Tabela 6, baseada em [Kle+15], ilustra a melhoria de desempenho do ActiveMQ em relação OpenMQ, como foi referido anteriormente o sistema OpenMQ não conseguiu processar todas as mensagens quando o tamanho destas era de 256B pelo que não é possível calcular a melhoria para este valor. Nesta é possível observar que o desempenho do ActiveMQ foi superior à do OpenMQ para qualquer tamanho de mensagem.

Tamanho da Mensagem	256B	2.048KB	16.384KB	131.072KB	1.024MB
ActiveMQ	#N/A	626.50%/1651.82%	452.70%/970.22%	236.01%/302.37%	156.61%/156.66%

Tabela 6: Melhoria de desempenho em percentagem do ActiveMQ em relação ao OpenMQ no envio/receção para o teste 1.

Tal como no primeiro teste os valores a utilizar para o segundo foram os mesmos, mudando apenas os intervenientes tal como ilustrado na Figura 23. Também como no teste anterior para mensagem de tamanho 256B o sistema OpenMQ voltou a ficar sem memória, sendo que neste teste também para mensagens de 512B o sistema ficou sem memória, as Tabelas 7 e 8, baseadas em [Kle+15], ilustram os resultados obtidos para este segundo teste.

Tamanho da Mensagem	256B	2.048KB	16.384KB	131.072KB	1.024MB
ActiveMQ	398.9/242.4	356.9/226.3	282.3/190.6	161.2/120.1	26.2/26.2
OpenMQ	#N/A	252.4/45.1	217.0/40.7	89.3/32.7	13.9/13.9

Tabela 7: Mensagens enviadas/recebidas por segundo para o teste 2.

Tamanho da Mensagem	256B	2.048KB	16.384KB	131.072KB	1.024MB
ActiveMQ	102.1/62.1	730.9/463.5	4625.2/3123.0	21123.6/15744.4	27493.7/27472.7
OpenMQ	#N/A	516.9/92.5	3555.9/666.9	11706.6/4180.8	14602.9/14589.3

Tabela 8: Payload enviado/recebido por segundo para o teste 2.

Mais uma vez tal como no teste anterior o desempenho do ActiveMQ foi superior ao do OpenMQ, sendo que isto pode observar-se na Tabela 9, baseada em [Kle+15], onde é ilustrada a melhoria de desempenho do ActiveMQ em relação ao OpenMQ.

Tamanho da Mensagem	256B	2.048KB	16.384KB	131.072KB	1.024MB
ActiveMQ	#N/A	141.42%/501.39%	130.07%/468.24%	180.44%/367.76%	188.28%/188.31%

Tabela 9: Melhoria de desempenho em percentagem do ActiveMQ em relação ao OpenMQ no envio/receção para o teste 2.

Após os 2 testes realizados é possível observar que o desempenho em cloud utilizando clusters é substancialmente superior no ActiveMQ, pelo que se no futuro o sistema de queues OpenMQ for adicionado ao Middleware a realizar terá de ser dada uma atenção superior para não degradar o desempenho deste.

2.3.4 Kafka vs. RabbitMQ

Philippe Dobblaere e Kyumars Sheykh Esmaili [DE17] realizaram um estudo qualitativo e quantitativo entre o sistema Kafka e o sistema RabbitMQ.

Em termos qualitativos foi avaliado o tempo de desacoplamento (*time decoupling*), lógica de roteamento (*routing logic*), garantia de entrega, garantia de ordem, disponibilidade, transações, *multicast* e escalabilidade dinâmica.

No estudo quantitativo foi comparado a latência e taxa de transferência.

Nas seguintes subsecções serão abordados os temas descritos acima, começando pelo estudo qualitativo.

2.3.4.1 Análise Qualitativa

Tempo de Desacoplamento

O RabbitMQ para além de persistir as mensagens guarda as mesmas na memória para um acesso mais rápido, mas quando a memória se esgota apenas persiste as mensagens tendo depois a leitura das mesmas um grande impacto no desempenho.

Por outro lado, o Kafka foi desenhado para suportar várias taxas de consumo pelo que está mais preparado para lidar com o *time decoupling*.

Lógica de Roteamento

O RabbitMQ implementa a lógica do protocolo AMQP para o roteamento, mas acrescenta algumas funcionalidades em cima do protocolo para permitir balanceamento de carga. Outra funcionalidade é o *Alternate Exchange* que permite os clientes retirarem mensagens do Exchange que estavam lá bloqueadas.

O Kafka por outro lado é bastante mais limitado, apenas suporta uma forma básica de roteamento baseada em tópicos.

Garantia de Entrega

O protocolo AMQP garante a entrega de mensagens através de transações, mas como estas tem um grande impacto no desempenho o RabbitMQ implementa um mecanismo de confirmação para não degradar o desempenho do mesmo. Para garantir que as mensagens não se perdem existe a noção de proprietário de uma mensagem.

Na Figura 24, baseada em [DE17], o *Producer* é proprietário da mensagem até receber o *ACK* em t3, após o *ACK* o proprietário da mensagem passa a ser o sistema de queues, sendo que este irá ser proprietário até existir um *ACK* coordenado com o *Consumer* em t5. No caso de o sistema de queues estar a ser utilizado como Pub/Sub este só deixará de ser proprietário da mensagem quando todos os *Consumers* a lerem. Isto é possível porque o RabbitMQ guarda estado de todos os *Consumers* que leram a mensagem. Quando um nodo do sistema deixa de ser proprietário de uma mensagem este elimina-a.

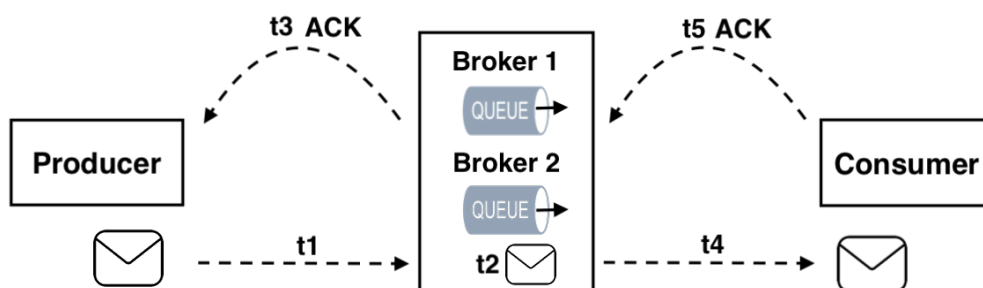


Figura 24: Entrega fiável de mensagens.

O Kafka não tem a noção de proprietário nem guarda informação de quais *Consumers* lerem uma mensagem, mas o trajeto de uma mensagem e dos *ACKs* são os mesmos. Não tendo a noção de proprietário e não guardando estado de quem já leu a mensagem o Kafka guarda a mensagem durante um certo período, normalmente durante vários dias.

Garantia de Ordem

O RabbitMQ garante a ordem das mensagens mesmo que seja necessária a retransmissão, mas no caso de existir balanceamento de carga mensagens que sejam enviadas de canais diferentes não é possível garantir a ordem.

O Kafka conserva a ordem dentro de uma partição, mas dentro de uma partição ou todas as mensagens são enviadas com sucesso ou todas falham.

Disponibilidade

O RabbitMQ pode ser configurado para replicar todos os *Exchanges* e ligações, mas para o espalhamento de queues é necessário explicitá-lo no momento de criação das mesmas.

No Kafka basta iniciar o sistema com um alto fator de replicação.

Transações

As transações do AMQP apenas se aplicam a publicações (envio de mensagens do produtor para o broker) e a confirmações (acks). O RabbitMQ adicionou a rejeição às transações. O AMQP garante a atomicidade quando uma transação envolve apenas uma queue, mas o RabbitMQ não garante, pelo que uma falta pode gerar inconsistência que é posteriormente resolvida através de comunicação entre o broker e o consumidor ou produtor.

À data do estudo realizado o Kafka não possuía transações, atualmente já existem transações e servem para tratar de alguns problemas de duplicações que podiam ocorrer na falha de alguns nodos, entre outros [MG17].

Multicast

O RabbitMQ fornece multicast replicando as queues, isto é, se existirem 5 consumidores existirão 5 queues, para cada consumidor e todas com as mesmas mensagens replicadas. No caso de *fan-out* é mantido na mesma uma queue por consumidor, mas em vez de estas conterem a mensagem contêm apenas certos dados, estando o corpo da mensagem guardada numa única queue.

O Kafka fornece multicast mantendo apenas uma cópia de cada mensagem no caso de o sistema ser não replicado. Como o Kafka não tem noção de quando todos os consumidores lerem uma mensagem, sendo que esta é mantida durante um certo tempo que é configurável.

Escalabilidade Dinâmica

O RabbitMQ suporta a adição e remoção de nodos de um cluster, sendo estes processos transparentes para os utilizadores. No caso da adição estes nodos poderão passar a ser nodos principais de queues que sejam, entretanto, criadas.

No Kafka a adição e remoção de nodos não é transparente para o utilizador, pois existe um mapeamento entre as partições e os consumidores de um grupo, pelo que é necessário atualizar sempre que são alteradas.

2.3.4.2 Análise Quantitativa

Latência

No teste de latência foram realizados dois testes para se poder observar qual o impacto de passar de um QoS de nível 0 para um QoS de nível 1 nos dois sistemas. Após os testes realizados notou-se que a alteração do nível do QoS praticamente não afetou a latência, tanto para o RabbitMQ como para o Kafka. De seguida na Tabela 10 e 11, baseadas em [DE17], são apresentados os resultados para os dois sistemas em condições normais.

	Média	Máximo
RabbitMQ	1-4	2-17
Kafka	1	15

Tabela 10: Latência em milissegundos com sistemas não replicados.

	Média	Máximo
RabbitMQ	1-4	2-17
Kafka	1	30

Tabela 11: Latência em milissegundos com sistemas replicados.

Taxa de Transferência

Tal como no teste de latência foram realizados dois testes para avaliar a taxa de transferência, baseando-se estas diferenças novamente ao nível do QoS (nível 0 e nível 1).

Na Figura 25, retirada de [DE17], é possível observar os resultados obtidos para o teste do QoS nível 0, tanto para o RabbitMQ como para o Kafka, é ainda possível verificar que para pacotes da mesma dimensão o RabbitMQ obteve uma maior taxa de transferência dos mesmos. Esta comparação é baseada na linha a roxo do RabbitMQ (*confirm=1, no replication, Mbps*) com a linha a azul do Kafka (*bytes in Mbps*). Esta comparação é possível de ser feita pois os gráficos estão ambos na mesma escala.

É também possível observar na Figura 25 que o RabbitMQ sem replicação e sem confirmações (acks) obtém um maior número de pacotes transferidos por segundos, sendo que esta comparação é baseada na linha azul do RabbitMQ (*confirm=1, no replication, pps*) com a linha vermelha do Kafka (*packets in pps*). Sobre os 50000 pacotes por segundo obtidos pelo Kafka quando o tamanho dos mesmos é muito reduzido nada se pode inferir uma vez que não existem resultados para esse tamanho de pacotes no RabbitMQ.

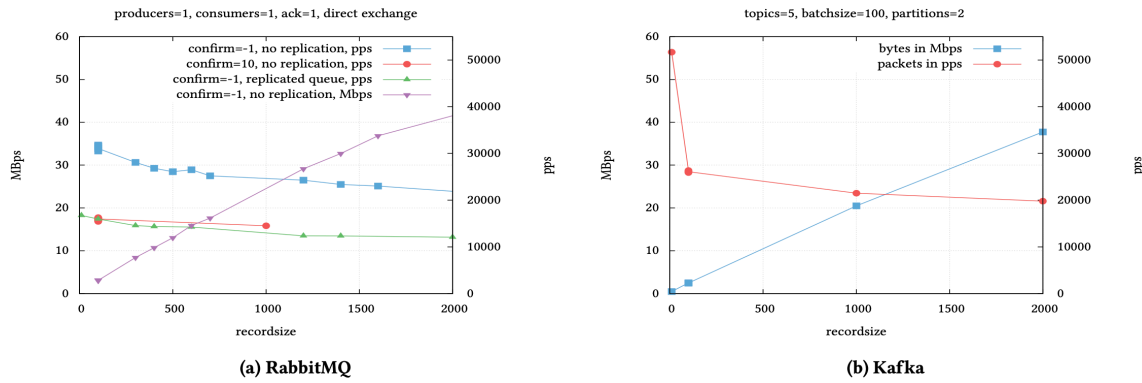


Figura 25: Resultados teste com QoS nível 0.

Na Figura 26, retirada de [DE17], é possível observar o impacto no desempenho do Kafka quando o número de tópicos varia. Nesta é possível observar que o Kafka é escalável quando o número de tópicos aumenta, uma vez que quando estes aumentam o número de pacotes por segundo aumenta também. É também possível observar um decréscimo significativo quando é introduzida replicação, uma vez que é necessária a coordenação entre as duas réplicas.

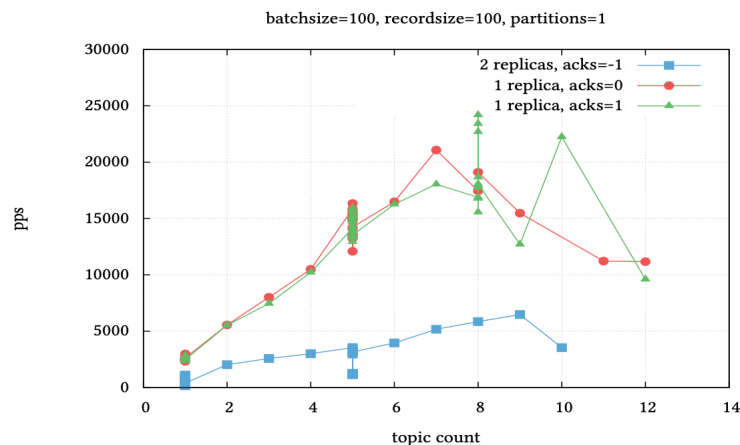


Figura 26: Impacto no desempenho consoante o número de tópicos no sistema Kafka.

2.4 Trabalhos Relacionados

De seguida será apresentado um trabalho relacionado com o Middleware a desenvolver e este servirá como base teórica para funcionalidades e decisões a tomar.

2.4.1 Multi-MOM

Multi-MOM é uma Dissertação de Mestrado apresentada e defendida por Yuri Morais Bezerra em [Bez10]. Nesta dissertação é abordado um problema semelhante ao da dissertação a desenvolver.

Yuri Bezerra dissertou sobre os vários padrões de comunicação síncrona e assíncrona existentes para o desenvolvimento de aplicações mobile. Existindo uma panóplia de soluções este criou uma biblioteca de forma a

dar suporte aos principais paradigmas de comunicação síncrona e assíncrona, permitindo assim a um programador estar apenas dependente da sua camada e obter acesso a vários paradigmas de comunicação.

Esta biblioteca foi desenvolvida de forma a ser extensível permitindo facilmente a adição e remoção de paradigmas de comunicação. Além disso a sua biblioteca é configurável, isto é, um programador que queira utilizar a sua biblioteca pode especificar quais os paradigmas de comunicação que pretende utilizar fazendo assim com que a biblioteca seja mais leve quando não são necessários todos os paradigmas de comunicação.

A funcionalidade de configuração de quais paradigmas de comunicação utilizar era muito útil na altura em que o Multi-MOM foi lançado porque os *smartphones* eram muito mais limitados em termos de recursos do que na atualidade. Porém esta funcionalidade continua a ser útil atualmente porque quanto menos recursos se necessitar mais recursos restam para outras funcionalidades.

Apesar de na altura existirem inúmeras soluções que implementavam vários paradigmas de comunicação, não era conhecida uma que implementasse tantos paradigmas como a solução proposta por Yuri Bezerra. Além disto nenhuma das soluções conhecidas eram facilmente extensíveis a novos paradigmas de comunicação, sendo que Yuri Bezerra teve este pormenor em atenção por forma a o tornar numa funcionalidade do Multi-MOM.

Um problema desta solução é que sendo uma biblioteca esta apenas funciona para um conjunto de *smartphones*, sendo que neste caso *smartphones* Android, os restantes que utilizam sistemas operativos como iOS ou Windows Phone não poderiam utilizar esta biblioteca.

A presente dissertação tem como objetivo a integração de diversos MOMs, pelo que a funcionalidade de extensibilidade é de elevada relevância uma vez que numa primeira fase serão apenas implementados alguns sistemas de queues e a funcionalidade de configuração é também extremamente importante uma vez que o cliente terá de escolher apenas os sistemas de queues que pretende utilizar para não sobrecarregar o Middleware com sistemas de queues que não lhe sejam relevantes.

3 Descrição do Problema

Com a existência de diversos sistemas de queues e o constante aparecimento de novos sistemas torna a escolha cada vez mais difícil, mesmo existindo vários estudos comparativos para tentar agilizar este processo.

Além do crescente número destes sistemas a inexistência de uma norma no que toca às suas APIs as aplicações ou componentes de uma aplicação que utilizem estes sistemas ficam bastante dependentes destes. No caso de surgir a necessidade de trocar de um sistema de queues para outro pode ser um processo muito moroso dependendo do grau de modularização da aplicação, pois a troca de um sistema para outro pode levar a uma reestruturação e modificação dos componentes da aplicação.

Existem alguns motivos que podem levar uma aplicação a trocar de sistemas de queues, sendo que de seguida são apresentados alguns dos motivos pelos quais uma aplicação pode necessitar de o fazer.

- Qualidade de serviço do sistema, isto é, as decisões que o sistema toma em cada situação de forma a evitar ao máximo o constrangimento da aplicação.
- Sistema que está a ser utilizado foi descontinuado, isto é, o sistema deixou de ter manutenção ou novas atualizações, pelo que surgem novos sistemas que ultrapassam este.
- A aplicação adicionou algumas funcionalidades para as quais precisa de um sistema de queues diferente do qual estava a utilizar.
- Sistema que está a ser utilizado não segue boas práticas no lançamento de novas atualizações, ou seja, as novas atualizações podem criar conflitos na aplicação que estava a utilizar o sistema, levando esta a ter de utilizar versões anteriores do sistema ou a ser reformulada para suportar a nova versão. Um exemplo destas atualizações que não seguiram boas práticas foi o protocolo AMQP, o sistema RabbitMQ utiliza a versão 0.9.1 deste protocolo e não foi criada a versão para suportar a 1.0.0 do mesmo por serem muito distintas, as diferenças foram abordadas na Secção 2.1.1.

A Figura 27 ilustra as Application1 e Application2 dependentes do Broker1, se estas agora necessitarem de trocar para o Broker2, por algum dos motivos anteriores ou outro não referido, tem de ser modificadas ou adaptadas por forma a conseguirem comunicar com este novo sistema de queues. Isto acontece porque as aplicações estão dependentes da API fornecida pela versão cliente dos sistemas de queues.

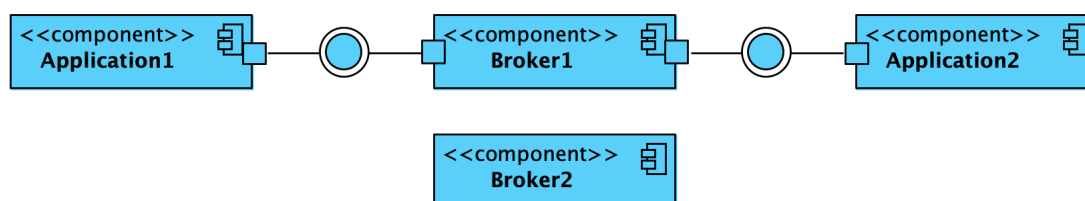


Figura 27: Aplicação dependente do sistema de queues.

Apesar de na Figura 27 as aplicações estarem dependentes de sistemas intermediários externos, sendo que

esta dependência também ocorre para os sistemas intermediários internos.

Além da dependência ao nível do código existe um outro problema que consiste na conjugação da linguagem de programação com as linguagens suportadas para cada um dos sistemas de queues. Por exemplo existe a versão cliente do ActiveMQ para a linguagem de programação Golo⁴⁷, mas para esta linguagem de programação não existe a versão cliente do RabbitMQ nem do ZeroMQ pelo que no caso de escolha desta linguagem de programação estes sistemas seriam logo excluídos. No caso de ser uma restrição da aplicação utilizar o RabbitMQ como sistema de queues então a linguagem de programação Golo já não poderia ser utilizada.

Por forma a resolver estes problemas existem algumas soluções, como a criação de uma biblioteca a nível de código, tal como a solução apresentada por Yuri Bezerra em [Bez10]. Porém esta solução necessita que para cada linguagem de programação seja criada uma biblioteca. A Figura 28 representa a introdução desta biblioteca entre a aplicação e os sistemas de queues.

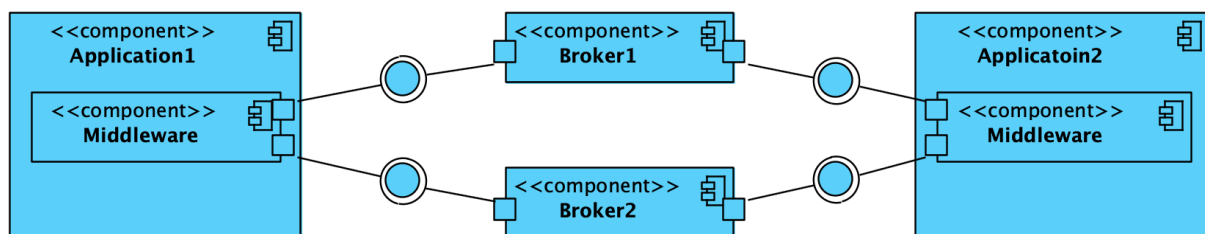


Figura 28: Biblioteca entre as aplicações e os sistemas de queues.

Em termos de desempenho esta solução tem um impacto quase nulo, uma vez que são apenas executados alguns métodos extra entre o momento em que a aplicação quer enviar a mensagem até ser efetivamente enviada para o sistema de queues.

Outra solução passa pela criação de um componente intermediário entre a aplicação e o sistema de queues, sendo que esta resolveria a dependência a nível de código e também o problema de falta de suporte das versões clientes dos sistemas de queues para algumas linguagens de programação, porém esta solução força à configuração deste componente para este poder comunicar com os sistemas de queues. Na Figura 29 é possível observar como ocorreriam as comunicações entre a aplicação e o sistema de queues através do novo componente.

Esta solução apresenta um maior impacto no desempenho em relação à solução anterior uma vez que é necessária a passagem por um componente extra, o Middleware, antes da mensagem ser entregue e lida do sistema de queues.

⁴⁷Golo: <https://golo-lang.org/>

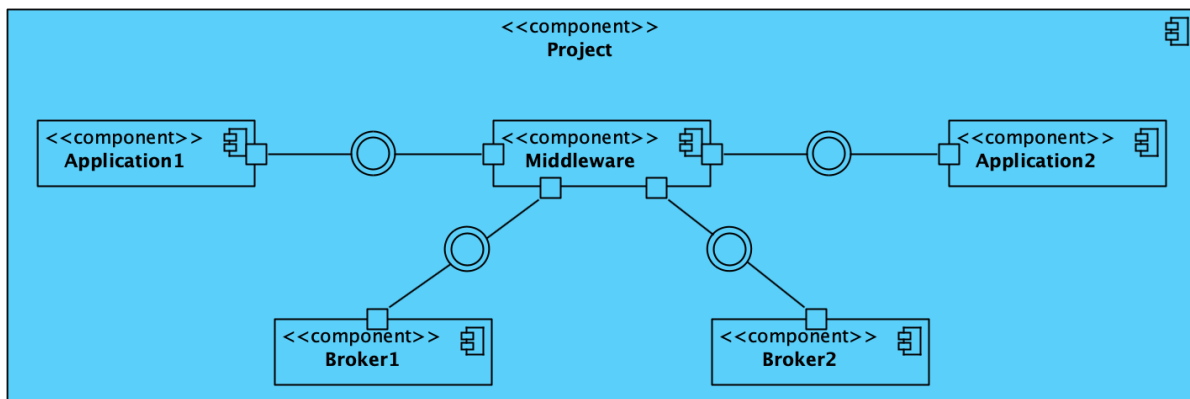


Figura 29: Middleware interno ao projeto.

Uma terceira solução é a criação de um serviço externo ao projeto. Com esta solução já não é necessária a configuração dos sistemas de queues, pois o serviço externo resolve ele próprio esse processo de configuração, removendo assim qualquer sistema de queues da arquitetura interna do projeto. Na Figura 30 é possível observar que o componente Middleware se encontra externo ao projeto e que é este que possui os sistemas de queues, abstraindo assim a sua configuração do projeto.

Ao contrário da solução anterior esta terá um grande impacto no desempenho, uma vez que é um sistema externo ao projeto existirá uma maior latência na comunicação entre as aplicações e os sistemas de queues através do Middleware e além disto terá de existir uma camada de segurança para garantir que apenas as aplicações do projeto conseguem comunicar através do Middleware, todas as medidas de segurança terão também impacto no desempenho.

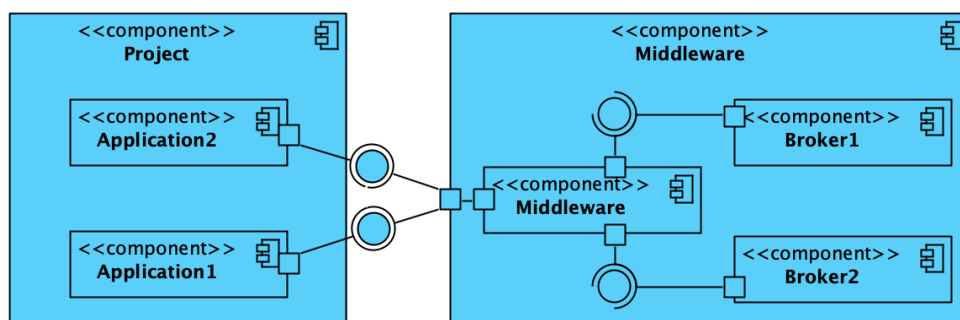


Figura 30: Middleware externo ao projeto.

Ponderadas as possíveis soluções para a resolução dos problemas referidos será adotada a terceira opção.

Postos estes problemas e as respetivas possíveis soluções o objetivo passa agora pelo desenvolvimento de um Middleware que englobe as duas soluções referidas acima por forma a ser um serviço que suporte vários sistemas de queues existentes e que possa ser acedido através de qualquer linguagem de programação.

De modo a que este Middleware não seja apenas um adaptador entre os vários sistemas de queues e as aplicações que necessitam deles para comunicar assincronamente este terá características de valor acrescentado

de forma a adicionar novas funcionalidades, permitindo assim às aplicações tirarem um maior partido na utilização deste sistema.

4 Conceção

No presente capítulo é apresentada a conceção do Middleware a desenvolver. Inicialmente será ilustrado o domínio do problema na Secção 4.1, de seguida serão elaborados os requisitos na Secção 4.2, após o levantamento, a análise e a priorização de requisitos será realizada a modelação necessária ao desenvolvimento do Middleware e apresentada na Secção 4.3, respeitando os requisitos e restrições levantados numa fase inicial.

4.1 Modelo de Domínio

O modelo de domínio representa as entidades do problema em mãos e de como estas interagem entre si. Apesar deste ser um modelo de alto nível de abstracção todas as decisões tomadas sobre ele tem impacto na elaboração de requisitos e subsequentes diagramas de modelação.

Posto isto este é fulcral para o desenvolvimento de qualquer aplicação, uma vez que será a base para a elaboração dos requisitos e subsequentes diagramas de modelação de mais baixo nível, como diagramas de classes. Na Figura 31 é ilustrado o modelo de domínio elaborado.

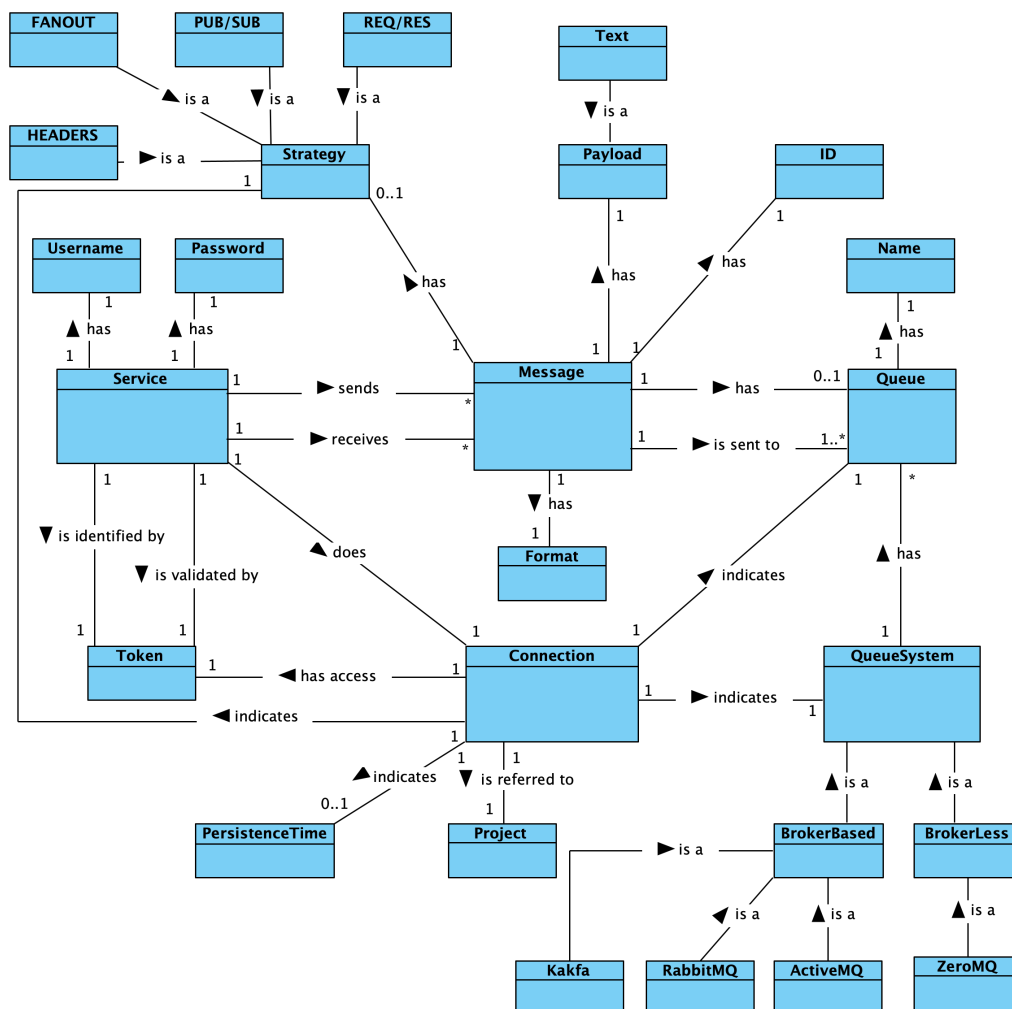


Figura 31: Modelo de Domínio.

As principais entidades do modelo de domínio elaborado são o serviço (Service), a conexão (Connection), o sistema de queues (QueueSystem) e a mensagem (Message).

Tal como se pode observar a entidade serviço tem de realizar uma conexão, sendo que nesta é indicado qual o sistema de queues, a estratégia e a Queue que se pretende utilizar, podendo ainda especificar um tempo pelo qual as mensagens ficarão persistidas. No momento em que é realizada a conexão é criado um Token único de forma a identificar e validar esse serviço.

O sistema de queues pode ser um sistema intermediário externo (BrokerBased) ou intermediário interno (BrokerLess), sendo que os intermediários externos são o RabbitMQ, ActiveMQ e Kafka, e o intermediário interno é o ZeroMQ.

Para iniciar o envio ou receção de mensagens o serviço tem de realizar a conexão anteriormente referida por forma a obter o Token de identificação.

A entidade mensagem tem de seguir um formato (Format) e o seu conteúdo (Payload) é texto (Text). Estas tem ainda de ser identificadas por um ID único e podem ou não ter uma estratégia associada, diferente daquela que o serviço indicou no momento da conexão. Além da estratégia pode também conter uma Queue permitindo assim o envio da mensagem para uma Queue diferente da indicada no momento da conexão.

A estratégia pode ser do tipo REQ/RES, PUB/SUB, FANOUT ou HEADERS, sendo que estas representam as estratégias suportadas pelos sistemas de queues, ressaltando que algumas podem ser incompatíveis, como por exemplo a Strategy FANOUT com o QueueSystem Kafka.

Os sistemas de queues são constituídos por Queues onde são armazenadas as mensagens que são enviadas por serviços. Estas são armazenadas nas Queues até um outro serviço as ler, dependendo da estratégia que esteja a ser utilizada.

4.2 Levantamento de Requisitos

Nesta secção serão listados todos os requisitos levantados numa fase inicial a ter em conta no desenvolvimento do Middleware.

Depois destes estarem elaborados será realizada a priorização dos requisitos, implementando assim os requisitos mais importantes para o correto funcionamento do Middleware.

De seguida são apresentados na Tabela 12 os requisitos de utilizador. Estes servem como base para a criação do Middleware, pois este terá de ter em conta todos os requisitos especificados, mesmo que apenas sejam implementados parte destes.

# Requisito	Descrição
1	Os utilizadores tem de ter a capacidade de se registar no Middleware.
2	No momento do registo tem de ser especificado o sistema de queues principal a utilizar.
3	No momento do registo tem de ser especificada a estratégia de comunicação principal a utilizar com o sistema de queues, bem como todos os componentes necessários para a utilização dessa estratégia.
4	Os utilizadores tem de efetuar o login para ficarem ligados ao sistema.
5	No momento de envio de uma mensagem o utilizador pode especificar uma queue ou estratégia diferente da principal.
6	No momento de envio de uma mensagem pode especificar um atraso na mesma para esta ser entregue mais tarde.
7	O utilizador pode ativar a persistência de mensagens, permitindo assim a releitura das mensagens.
8	Uma mensagem pode ser obtida novamente pelo seu utilizador enquanto estiver persistida.
9	O utilizador pode eliminar uma mensagem que esteja persistida.

Tabela 12: Requisitos.

4.2.1 Priorização

Terminada a especificação dos requisitos estes serão agora priorizados por forma a implementar os requisitos indispensáveis para o Middleware, ou seja, que ofereçam o Minimum Viable Product (MVP). O MVP é o produto mínimo a oferecer de forma a ser possível utilizar o sistema com funcionalidades básicas.

Por forma a atingir este MVP será então dada prioridade aos requisitos de 1 a 4 presentes na Tabela 12 da Secção 4.2.

Os requisitos 5 a 9 serão implementados assim que os requisitos prioritários estiverem finalizados ou praticamente finalizados. Apesar de estes serem apenas implementados posteriormente serão tidos em consideração na fase de modelação (Secção 4.3) por forma a não criar incompatibilidades que levassem a uma reestruturação da implementação dos requisitos prioritários.

Os requisitos não prioritários que não sejam implementados durante o desenvolvimento serão adicionados ao trabalho futuro a efetuar sobre este Middleware.

4.3 Modelação

Por forma a melhor entender as entidades envolvidas na criação deste Middleware serão nesta secção ilustrados o diagrama de casos de uso, os diagramas de componentes e os diagramas de classes.

Durante a conceção do Middleware serão tidos em conta todos os requisitos, prioritários e não prioritários, ilustrados na Secção 4.2, pois se estes forem tidos em conta nesta fase de conceção serão evitadas incompatibilidades aquando da implementação dos requisitos não prioritários no futuro.

Inicialmente será ilustrado na Secção 4.3.1 o diagrama de casos de uso que representa as funcionalidades que poderão ser efetuadas sobre o Middleware.

De seguida serão ilustrados os diagramas de componentes na Secção 4.3.2. Estes representam os componentes externos (já desenvolvidos por terceiros) e os componentes a desenvolver, bem como estes comunicam uns com os outros.

Após os diagramas de componentes será ilustrada na Secção 4.3.3 as decisões arquiteturais levadas a cabo para a conceção dos diagramas de classes que serão ilustrados na Secção 4.3.4. Os diagramas de classes são modelos de baixo nível que representam os métodos e variáveis das classes a implementar em cada componente, bem como as suas interações.

4.3.1 Diagrama de Casos de Uso

O diagrama de Casos de Uso tem como objetivo ilustrar as funcionalidades que os utilizadores, ou as várias categorias de utilizadores, podem realizar. Este é um mapeamento quase direto dos requisitos para funcionalidades do sistema.

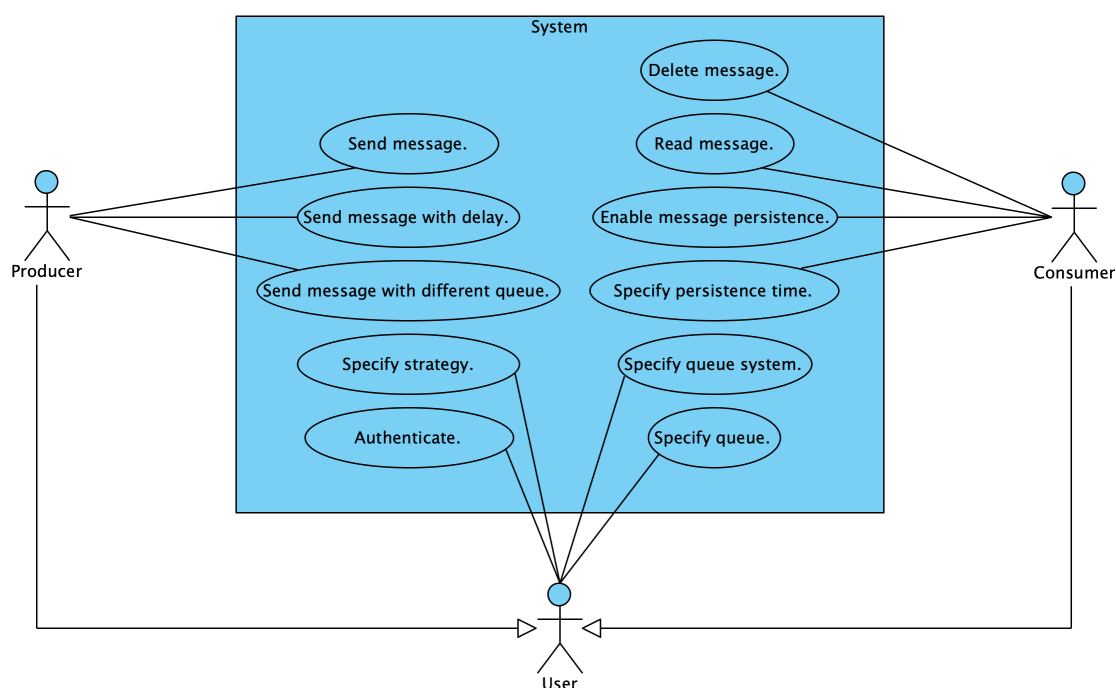


Figura 32: Diagrama de Casos de Uso.

A Figura 32 representa o diagrama de Casos de Uso elaborado através dos requisitos levantados. Neste é possível observar 3 atores, o User, o Producer e o Consumer. O User representa simultaneamente o Consumer e o Producer para algumas funcionalidades.

Tal como é ilustrado na Figura 32 as funcionalidades comuns ao Producer e ao Consumer são o registo, conexão ao sistema e atualização de dados.

O Producer além das funcionalidades comuns pode ainda enviar mensagens com ou sem atraso, especificando ou não uma queue ou estratégia diferente no envio da mesma.

O Consumer tal como o Producer tem funcionalidades extra para além das funcionalidades comuns, sendo que estas são a leitura mensagens sendo possível a releitura das mesmas se este tiver a persistência de mensagens ativa. O Consumer pode ainda eliminar mensagens que estejam persistidas e ativar ou desativar a persistência das mesmas.

Observando com atenção a Figura 32 é possível verificar efetivamente que os casos de uso fazem um mapeamento praticamente direto com os requisitos levantados e ilustrados na Tabela 12. Ora as funcionalidades do ator User presente no diagrama de casos de uso correspondem aos requisitos 1 a 4, mais especificamente o registo com o requisito 1, 2 e 3, e o login com o requisito 4..

A funcionalidade registo engloba 3 requisitos porque no momento em que o mesmo é efetuado tem de se especificar o sistema de queues e a estratégia juntamente com os restantes componentes necessários.

As funcionalidades do ator Producer coincidem com os requisitos 5 e 6, o envio de mensagens corresponde ao requisito 6, o envio de mensagens para uma estratégia ou queue diferente e o envio com atraso correspondem ao requisito 8 e 9, respetivamente.

No ator Consumer as funcionalidades coincidem com os requisitos 7 e 10 a 12, sendo a receção de mensagens correspondente ao requisito 7, a ativação da persistência das mensagens mapeia o requisito 10 e a releitura e a eliminação de mensagens corresponde aos requisitos 11 e 12, respetivamente.

4.3.2 Diagrama de Componentes

Na presente secção serão ilustrados três diagramas de componentes elaborados, sendo que estes representam diferentes níveis de abstração da arquitetura a utilizar. A Figura 33 representa um nível mais alto de abstração relativamente aos componentes Middleware e Registration. Estes encontram-se mais detalhados nas Figuras 34 e 35.

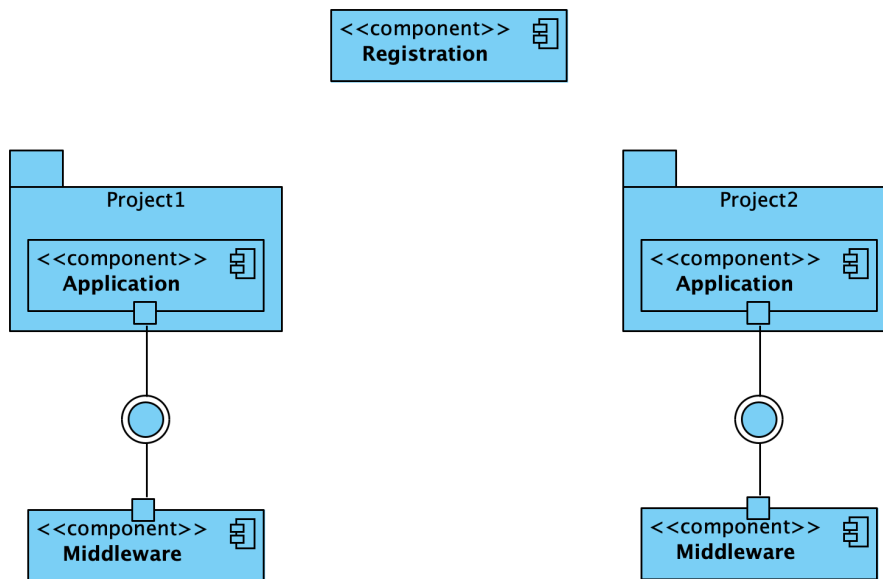


Figura 33: Diagrama de componentes de alto nível de abstração.

Tal como é possível observar na Figura 33 o componente Middleware vai ser um componente replicável, sendo que neste caso será um componente Middleware por cada conjunto de Applications que estejam num Project. O componente Registration será responsável pelo registo, criação e gestão de componentes Middleware.

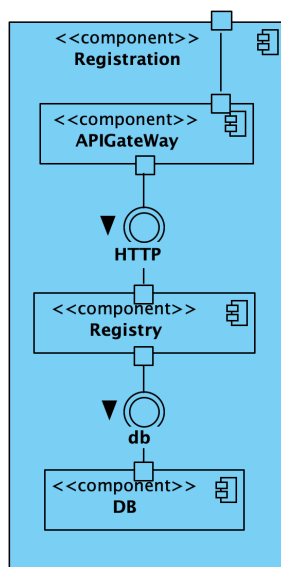


Figura 34: Diagrama de componentes do Componente Registration.

Na Figura 34 é possível observar a configuração interna do componente Registration. Ora tal como é possível observar o componente Registration é constituído internamente por um APIGateway, que irá ter a responsabilidade de filtrar todos os pedidos, deixando apenas passar aqueles que estejam autenticados quando necessário e aqueles que correspondam exatamente às rotas posteriormente definidas.

O componente Registry é o componente a desenvolver, sendo que este tem a responsabilidade de gerir as

diversas instâncias de Middleware. Este é auxiliado por uma Base de Dados (BD) onde são persistidas todas as informações pertinentes para o correto funcionamento deste componente.

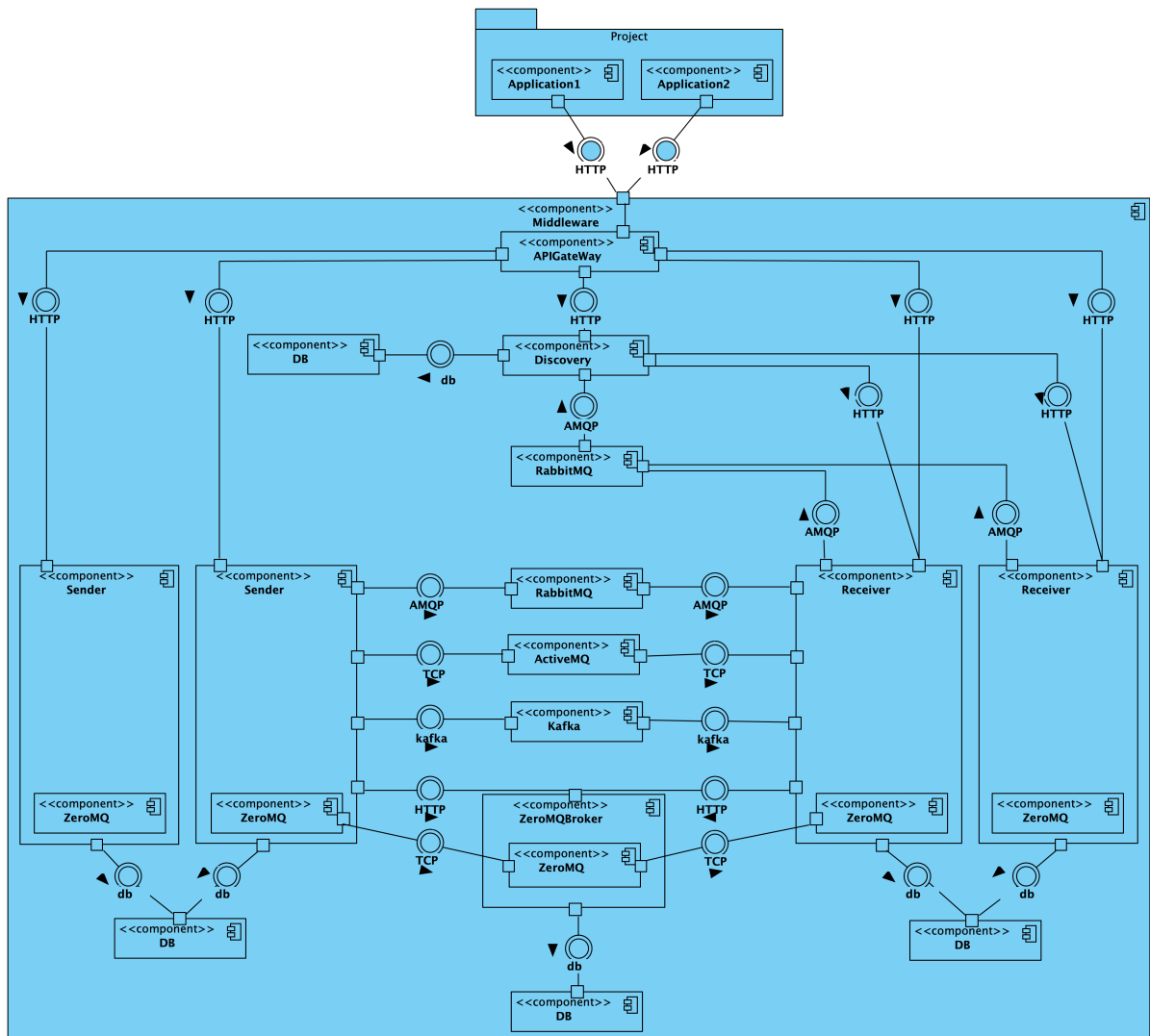


Figura 35: Diagrama de componentes do Componente Middleware.

O componente Middleware, representado na Figura 35, será o componente responsável pela abstração dos sistemas de queues, sendo que neste caso RabbitMQ, ActiveMQ, Kafka e ZeroMQ. Neste componente existem quatro componentes a desenvolver que são o Sender, o Receiver, o Discovery e o ZeroMQBroker, tal como os nomes indicam o Sender será responsável por enviar mensagens para os sistemas de queues, o Receiver irá receber as mesmas, o Discovery está responsável por saber em qual das réplicas Receiver um certo receiver se encontra ligado e o ZeroMQBroker é uma implementação do ZeroMQ de forma a tornar este num sistema intermediário externo.

Um dos motivos que levou à separação dos produtores dos consumidores (Sender e Receiver) foi o balanceamento de carga. Esta decisão ocorreu porque num projeto podem existir mais produtores do que consumidores e vice-versa, e com esta divisão é possível que existam mais réplicas de Sender do que Receiver e vice-versa.

Outro motivo para esta separação é a diferença de recursos utilizados entre um produtor e um consumidor.

Para um consumidor poder receber uma mensagem com o menor atraso possível este tem de manter uma conexão aberta com o componente Receiver para que quando esta chegue ao componente ser logo entregue ao consumidor, por outro lado, os produtores apenas possuem conexões enquanto fazem pedidos de envio de mensagens ou de atualização de dados, não mantendo uma conexão aberta por tempo indeterminado.

O ZeroMQ sendo um sistema intermediário interno que no caso de Req/Res funciona por portas e não por queues, isto é, a comunicação é feita ponto a ponto através de TCP sendo utilizada uma porta disponível, por exemplo se fossem necessárias duas queues diferentes neste sistema teriam de ser utilizadas duas portas diferentes. No caso de Pub/Sub é possível subscrever uma queue funcionando normalmente como os outros sistemas.

Por forma a resolver o problema do Req/Res para o sistema ZeroMQ será necessária a criação do ZeroMQ-Broker anteriormente referido. Este componente tem como responsabilidade a gestão de portas para cada queue diferente, permitindo assim aos Users utilizarem o ZeroMQ da mesma forma que os restantes são utilizados, isto é, um cliente é abstraído sobre a utilização das portas para o envio e receção de mensagens. Este sistema necessita de uma BD para manter os dados sobre que portas estão a ser utilizadas e queues associadas.

Tanto o APIGateWay do Registration como do Middleware apesar de não serem um serviço a desenvolver tem de ser configurados conforme as necessidades de cada um dos sistemas.

Cada um dos componentes a desenvolver irá ter uma BD privada, pois um padrão sobre microsserviços é que cada um tem a sua BD privada, isto é, um microsserviço não acede aos dados de outro. Ora na Figura 35 é possível observar que estão duas réplicas de Receiver a aceder à mesma BD. Isto não é contra o padrão anteriormente referido uma vez que em caso de microsserviços replicados as réplicas podem ficar todas a aceder à mesma BD.

Era também possível cada réplica ter a sua BD privada, mas isto levaria a uma maior complexidade uma vez que as várias réplicas das BDs teriam de se manter sincronizadas.

Como é possível observar na Figura 35 a comunicação entre o componente Discovery e o componente Receiver é efetuado de duas formas distintas. A comunicação do Discovery para o Receiver é realizada através de pedidos HTTP enquanto que do Receiver para o Discovery é feita através do RabbitMQ.

Esta decisão foi tomada pelo facto dos dados transmitidos do Receiver para o Discovery serem indispensáveis. Atualmente e como é possível observar na Figura 35 o componente Discovery não é replicável, mas este está preparado para o ser. Sendo estes dados indispensáveis no caso do Discovery não estar em funcionamento faria com que estes fossem perdidos, então é utilizado o RabbitMQ para a realização da comunicação assíncrona. No sentido inverso, do Discovery para o Receiver, a comunicação é realizada a uma réplica específica, porque o Discovery está a funcionar como um Router, encaminhando todos os pedidos que recebe para a réplica específica.

Relativamente ao diagrama representado na Figura 35 para uma maior simplicidade na leitura do mesmo foram apenas representados dois Senders e dois Receivers, pois estes poderão ser mais. Também para uma melhor legibilidade apenas foi representada a ligação de um Sender e de um Receiver aos sistemas de queues, pois todas

as réplicas terão exatamente as mesmas ligações.

4.3.3 Decisões Arquiteturais

Por forma a melhor conceber os vários componentes do Middleware a desenvolver é necessário tomar decisões relativas à arquitetura dos mesmos.

Como é possível observar na Secção 4.3.2 o Middleware a desenvolver será orientado a microserviços, sendo que isto levou à utilização da arquitetura hexagonal [Kon]. Este é um padrão arquitetural bastante utilizado na conceção e implementação de microserviços.

O padrão hexagonal consiste na utilização de adapters (adaptadores) e ports (portas). Existem adaptadores e portas de entrada e de saída, sendo que as portas fazem a ligação entre os adaptadores e a lógica de negócio.

A Figura 36, baseada em [Kon], ilustra a arquitetura hexagonal de um componente.

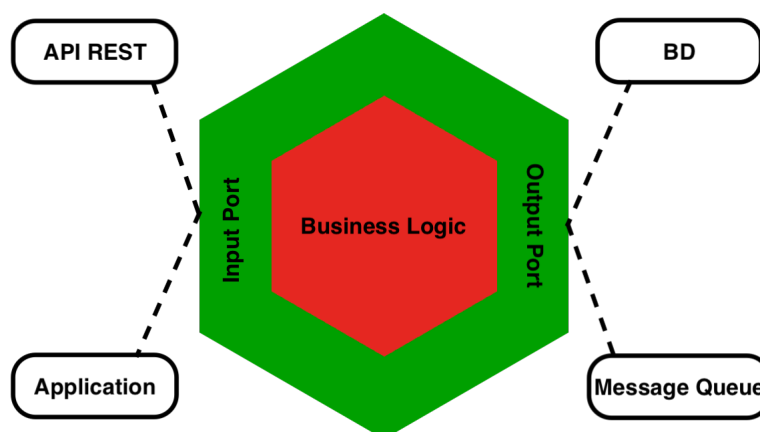


Figura 36: Padrão Hexagonal.

Com a utilização deste padrão a lógica de negócio não fica dependente dos adaptadores, uma vez que a lógica de negócio implementa as portas de entrada e comunica com as portas de saída.

Um padrão arquitetural inerente à utilização da arquitetura hexagonal é o padrão Adapter [Gura], sendo os intervenientes deste padrão a lógica de negócio, portas e adaptadores de saída, sendo que este padrão é de elevada relevância pois permite a mudança de um adaptador sem a lógica de negócio ser readaptada ao novo adaptador.

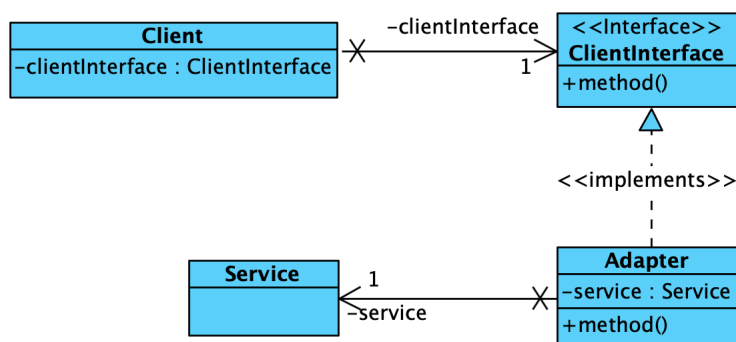


Figura 37: Padrão Adapter.

A Figura 37, baseada em [Gura], ilustra o padrão Adapter. Nesta o objeto Client representa a lógica de negócio, enquanto que a interface ClientInterface representa a porta de saída e o objeto Adapter representa o adaptador de saída.

Outra decisão tomada foi a utilização do padrão arquitetural Facade [Gurb] e Singleton [Gurc] na lógica de negócio. O padrão Facade foi utilizado para garantir que existia um ponto único de entrada para a lógica de negócio. O padrão Singleton será utilizado na mesma classe que implementa o padrão Facade garantindo assim que existe apenas uma instância do único ponto de entrada para a lógica de negócio.

Na Figura 38, baseada em [Gurb], encontra-se ilustrado o padrão Facade, sendo que nesta o objeto Client são os adaptadores de entrada e o objeto Facade o ponto único de entrada para a lógica de negócio. Este por sua vez comunica com todos os subsistemas que necessitar.

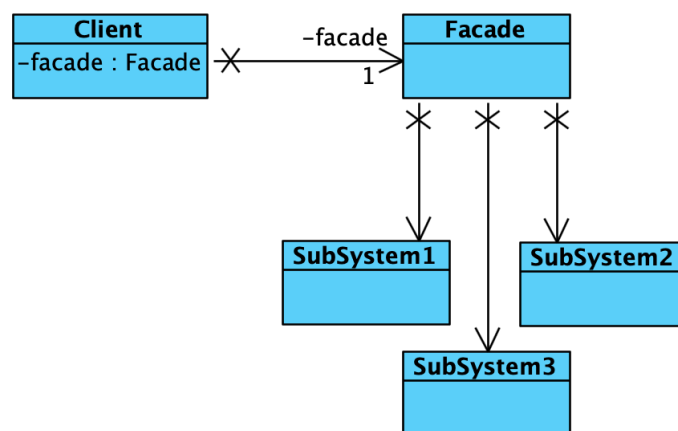


Figura 38: Padrão Facade.

O padrão Singleton encontra-se ilustrado na Figura 39, baseada em [Gurc]. Nesta o objeto Client representa os adaptadores de entrada e o objeto Singleton representa o ponto de entrada único da lógica de negócio, garantindo que todos os objetos Clients (adaptadores de entrada) utilizam a mesma instância.

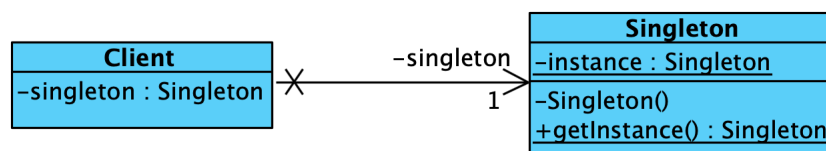


Figura 39: Padrão Singleton.

Por último, uma das decisões tomadas foi a utilização de Data Transfer Objects (DTOs), Pure Old Java Objects (POJOs) e entidades, isto é, o mesmo objeto tem 3 representações diferentes. Na lógica de negócio é utilizado o POJO, nos adaptadores de saída para ligação a BD é utilizada uma entidade que represente o POJO e nos adaptadores de entrada quando é necessário retornar o POJO este é convertido para um DTO. Esta decisão foi

tomada uma vez que os dados a persistir ou a retornar podem não corresponder exatamente a todas as variáveis do POJO, garantindo assim um maior controlo sobre os dados que são persistidos ou retornados.

4.3.4 Diagrama de Classes

Os diagramas de classes representam a estrutura interna de cada componente a desenvolver, bem como todos os métodos e parâmetros necessários para todas as funcionalidades serem concretizadas. Todos os diagramas de classes que irão ser representados a seguir seguem a mesma arquitetura interna, isto é, são maioritariamente compostos pelos mesmos packages.

Por forma a seguir o padrão hexagonal, apresentado na Secção 4.3.3, todos os microserviços a desenvolver são constituídos por três packages principais, o *inboundadapters*, o *businesslogic* e o *outboundadapters*.

O package *inboundadapters* contém as classes necessárias para as entradas do microserviço, o *outboundadapters* contém as classes necessárias para o envio de dados/informação para outros serviços (serviços externos ao microserviço). O *businesslogic* divide-se em três packages, o *inboundports*, o *business* e o *outboundports*.

No *inboundports* encontram-se apenas interfaces que são os pontos de entrada para o package *business*. Isto porque as classes do package *business* implementam as interfaces presentes no *inboundports* e as classes do package *inboundadapters* apenas conhecem as interfaces do *inboundports*.

O package *business* contém toda a lógica de negócio do microserviço em questão, sendo que este encontra-se dividido em mais packages, como o *resources* onde se encontram os POJOs, *security* onde ficam as classes para validação de tokens e o package *exception* onde estão colocadas todas as exceções personalizadas a cada microserviço.

Por último o package *outboundports* contém os pontos de saída do *business* do microserviço. Este é apenas constituído por interfaces tal como o package *inboundports*. As classes do package *outboundadapters* implementam as interfaces presentes no *outboundports* e as classes do package *business* apenas conhecem estas interfaces.

Por forma a facilitar a compreensão de qual classe vai implementar a respetiva interface, foram criados vários packages dentro do *outboundadapters* e do *outboundports* exatamente com os mesmos nomes, como por exemplo existe o package *data* no *outboundadapters*, então foi criado o package *data* no *outboundports*, sendo que a classe que se encontra no package *outboundadapters.data* implementa a interface que se encontra no package *outboundports.data*.

Como os diagramas de classes completos são de grandes dimensões estes serão apresentados em várias figuras em cada microserviço por forma a ilustrar a estrutura interna de cada microserviço e as classes mais importantes. Os diagramas completos encontram-se no anexo A.

Primeiramente será apresentado na Secção 4.3.4.1 o diagrama de classes do componente Registry, sendo de seguida apresentado o componente Sender na Secção 4.3.4.2, depois o componente Receiver na Secção 4.3.4.3 e por último o componente Discovery e ZeroMQBroker nas Secções 4.3.4.4 e 4.3.4.5, respetivamente.

4.3.4.1 Componente Registry

O componente Registry será o serviço responsável pela gestão de instâncias Middleware. Além disso este componente irá conter a informação sobre todos os sistemas de queues suportados para a gestão das instâncias. Inicialmente será ilustrada a arquitetura interna deste componente e de seguida serão ilustradas várias classes deste mesmo componente, começando pelas classes contidas no inboundadapters, passando pelas classes dos package businesslogic e por último as classes do package outboundadapters.

As Figura 40 a 42 representam a arquitetura interna do componente Registry, sendo que nestas é possível observar a arquitetura hexagonal descrita na Secção 4.3.4. Os packages inboundports e outboundports encontram-se repetidos propositadamente por forma a ser possível verificar as dependências que existem sobre as classes presentes nestes packages.

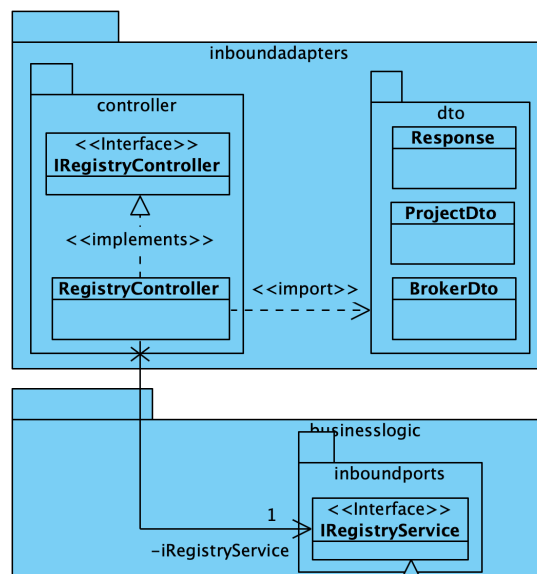


Figura 40: Arquitetura e classes inbounds do componente Registry.

Como é possível observar na junção da Figura 40 com a Figura 41 as classes do package inboundadapters apenas conhecem a interface presente no package inboundports.

Tal como acontece na junção das Figuras 40 e 41, na junção das Figuras 41 e 42 é possível observar que as classes do package business apenas conhecem as interfaces do package outboundports, sendo depois estas implementadas pelas classes do package outboundadapters.

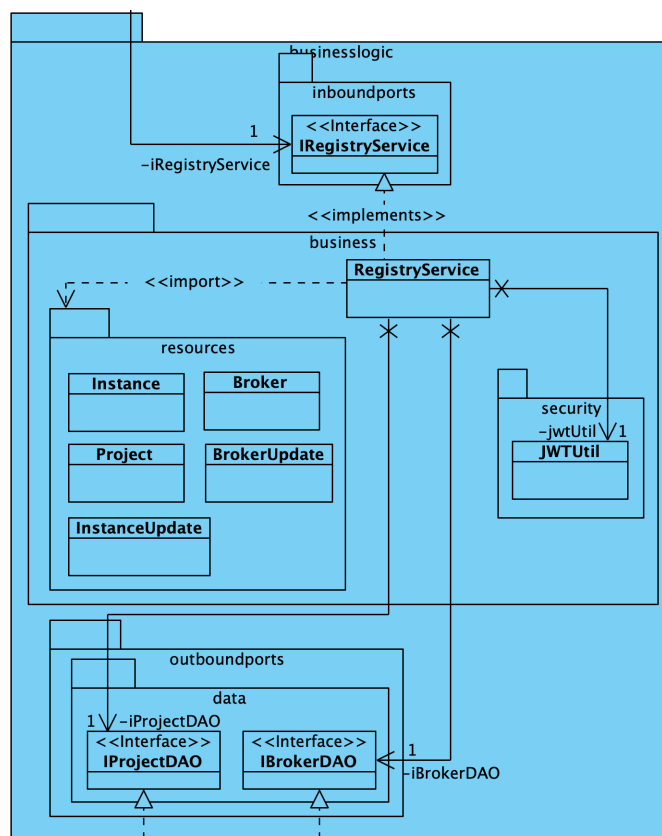


Figura 41: Arquitetura e classes business do componente Registry.

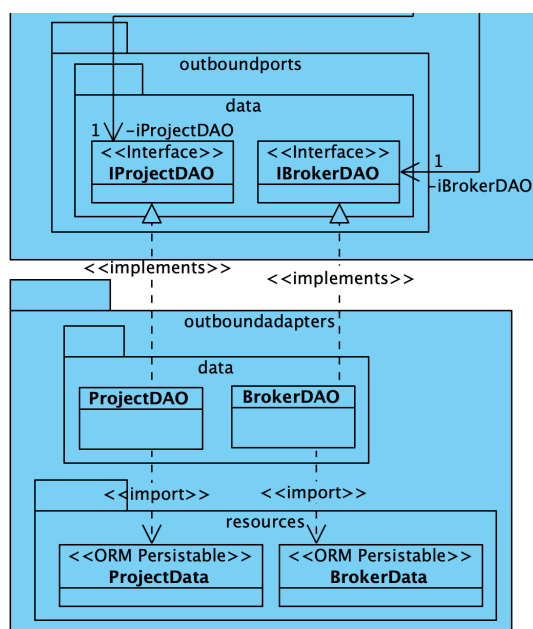


Figura 42: Arquitetura e classes outbounds do componente Registry.

A única forma de aceder a este serviço é através da sua API RESTful denominada de IRegistryController que é posteriormente implementada pela classe RegistryController. Este possui métodos para gestão de instâncias, que

são o registerProject, deleteProject, getProject e updateProject. O método getBrokers serve para informar todos os sistemas de queues suportados pelo Middleware. Por último os métodos registerBroker e updateBroker permitem a gestão dos sistemas de queues suportados pelo Middleware, permitindo assim a gestão desta informação sem a necessidade de parar o serviço. Todos estes métodos encontram-se representados na Figura 43.

RegistryController
-iRegistryService : IRegistryService
+registerProject(request : HttpServletRequest, response : HttpServletResponse) : void
+deleteProject(request : HttpServletRequest, response : HttpServletResponse) : void
+getProject(request : HttpServletRequest, response : HttpServletResponse) : void
+updateProject(request : HttpServletRequest, response : HttpServletResponse) : void
+getBrokers(request : HttpServletRequest, response : HttpServletResponse) : void
+registerBroker(request : HttpServletRequest, response : HttpServletResponse) : void
+updateBroker(response : HttpServletResponse, request : HttpServletRequest) : void

Figura 43: Classe RegistryController.

Sempre que um método da Figura 43 é invocado pela receção de um pedido HTTP este invocará um método da interface IRegistryService, sendo que esta invocação ocorre do seguinte modo, o método registerProject da classe RegistryController vai invocar o método registerProject da interface IRegistryService. Esta interface não é aqui demonstrada em detalhe pois a Figura 44 representa a classe RegistryService que implementa a interface IRegistryService, pelo que os métodos públicos da classe correspondem aos métodos da interface.

RegistryService
-iProjectDAO : IProjectDAO
-iBrokerDAO : IBrokerDAO
-jwtUtil : JWTUtil
-regions : Set<String>
+registerProject(project : Instance) : Project
+getProject(projectId : String, token : String) : Project
+deleteProject(projectId : String, token : String) : Project
+updateProject(instanceUpdate : InstanceUpdate, projectId : String, token : String) : Project
+getBrokers() : List<Broker>
+registerBroker(broker : Broker, token : String) : Broker
+updateBroker(brokerUpdate : BrokerUpdate, brokerId : String, token : String) : Broker

Figura 44: Classe RegistryService.

A classe RegistryService possui ainda uma coleção de regiões, sendo que isto permite validar que a região passada no momento de um registo de um projeto é válida.

Este componente tem a necessidade de persistir dois tipos de dados distintos, o Project e o Broker. Para isso foi criada uma interface para cada um no outboundports, denominadas de IProjectDAO e IBrokerDAO. Estas não estão representadas porque as classes ProjectDAO e BrokerDAO, representadas nas Figuras 45 e 47, respetivamente, contêm os mesmos métodos porque implementam estas interfaces.

Por forma a manipular os dados persistidos a interface IProjectDAO, representada pela classe ProjectDAO

na Figura 45, foram criadas as operações Create - Read - Update - Delete (CRUD) e além destas duas procuras para verificar se um certo elemento existe, o método existsByName é utilizado para verificar se existe algum projeto com o mesmo nome no momento da criação de um novo e o método existsByUuid verifica se existe o projeto com o Universally Unique Identifier (UUID)⁴⁸ fornecido. O UUID é o identificador único gerado no momento do registo do projeto.

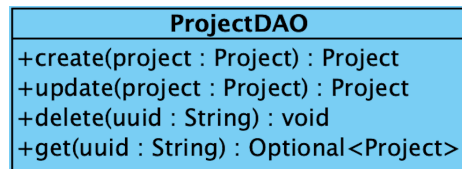


Figura 45: Classe ProjectDAO.

Na Figura 46 é possível observar todos os campos necessários persistir sobre a entidade ProjectData, o name é o nome definido para o projeto, o UUID e o atributo location será o Domain Name System (DNS) gerado para o projeto e este será baseado no nome do mesmo.

Region é um atributo que indica a preferência de região para o deployment do Middleware, permitindo assim uma menor latência na comunicação com o mesmo. O último atributo denominado de brokers é a lista de brokers suportados para a instância.

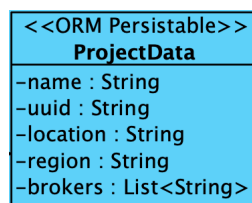


Figura 46: Classe ProjectData.

Para manipular os dados persistidos da classe Broker recorre-se à interface IBrokerDAO que tal como a anterior não é aqui representada, sendo ilustrada a classe BrokerDAO na Figura 47. Esta possui as operações CRUD sobre as instâncias de Broker e ainda a obtenção de todos os objetos e verificação de existência de um em específico.

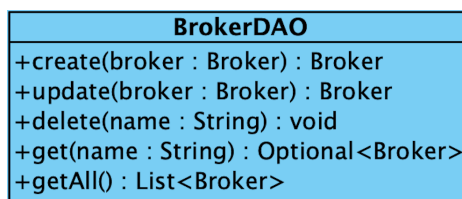


Figura 47: Classe BrokerDAO.

⁴⁸UUID: <https://tools.ietf.org/html/rfc4122>

No que toca a dados a persistir relativamente à entidade BrokerData representada na Figura 48 são o nome que é o nome do broker, o brokerVersion indica qual a versão suportada para este sistema de queues no Middleware e o brokerId é o identificador único gerado para o sistema de queues no momento do seu registo.

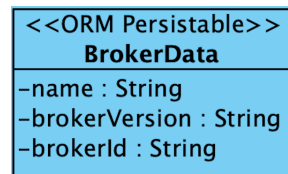


Figura 48: Classe BrokerData.

4.3.4.2 Componente Sender

O componente Sender será o serviço responsável pela gestão de produtores de mensagens, sendo este serviço também responsável por enviar as mensagens para os sistemas de queues. Inicialmente será ilustrada a arquitetura interna deste componente e de seguida serão ilustradas várias classes deste mesmo componente.

O conjunto de Figuras 49 a 51 representam a arquitetura interna do componente Sender, sendo que nestas é possível observar a arquitetura descrita na Secção 4.3.4. As Figuras 49 e 50 tem ambas o package inboundports representado permitindo assim visualizar as ligações entre o inboundadapters com o inboundports e o inboundports com o business, respetivamente.

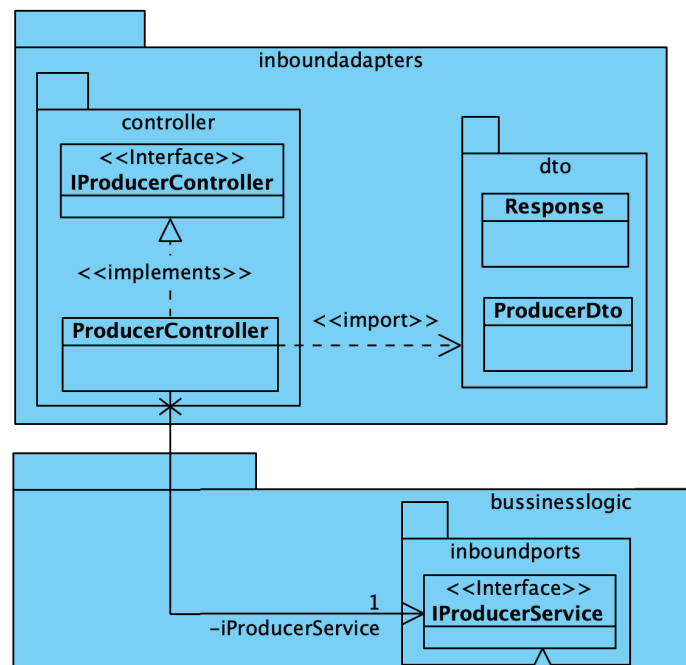


Figura 49: Arquitetura e classes inbounds do componente Sender.

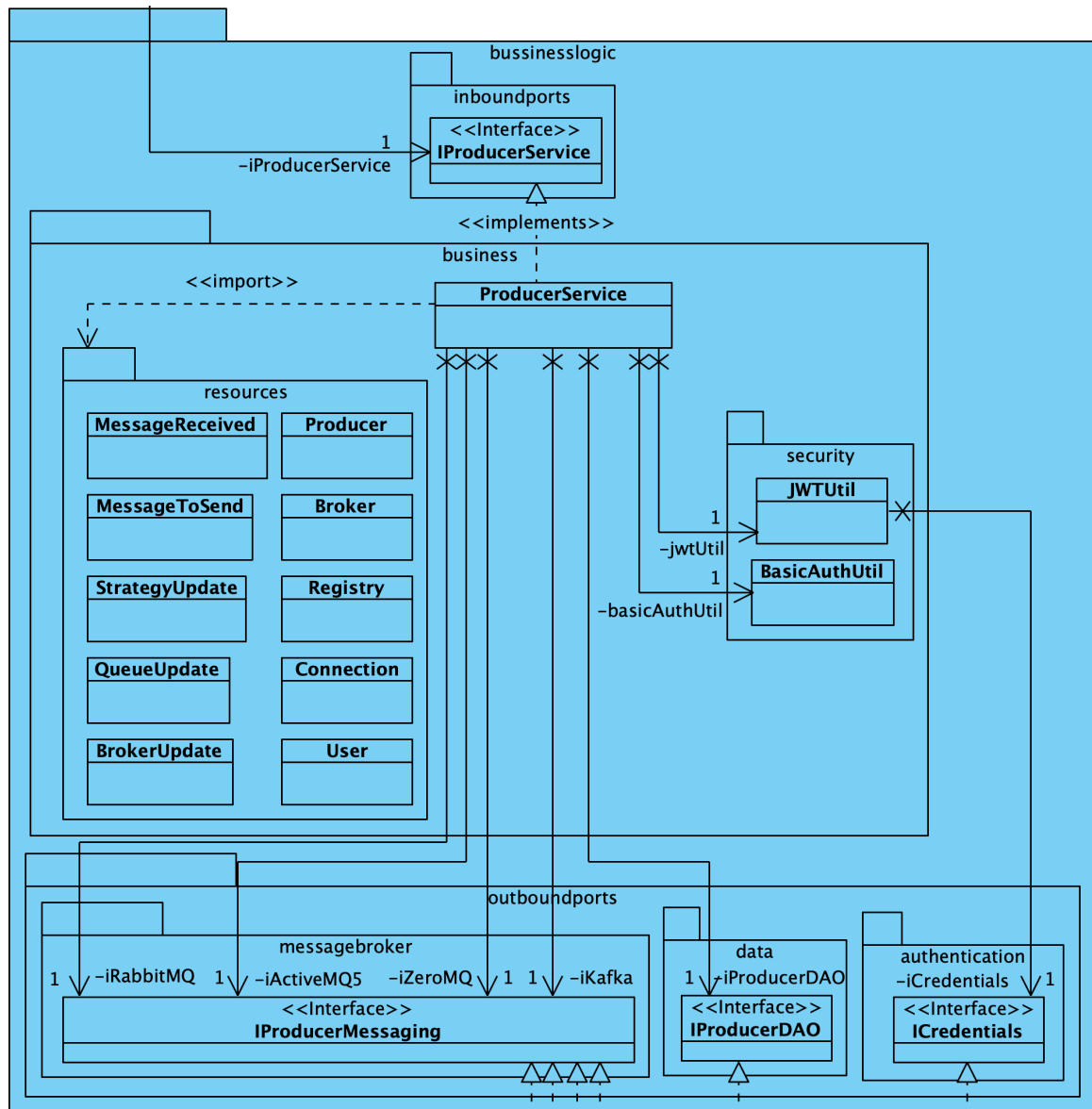


Figura 50: Arquitetura e classes business do componente Sender.

O package outboundports tal como o inboundports encontra-se representado nas Figuras 50 e 51, permitindo assim visualizar as ligações entre o outboundports com o business e o outboundadapters.

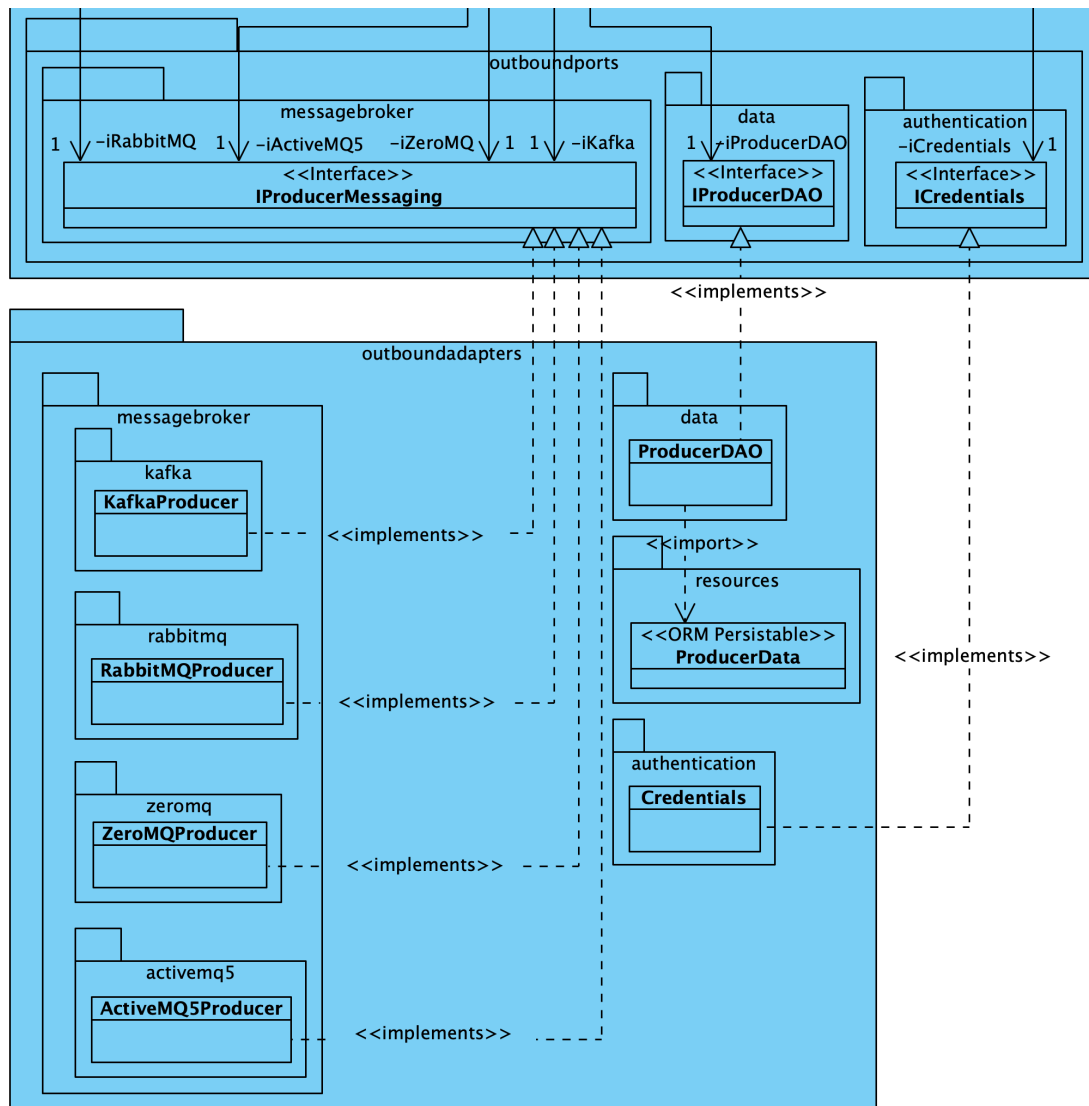


Figura 51: Arquitetura e classes outbounds do componente Sender.

Este serviço tal como o serviço Registry (Secção 4.3.4.1) apenas possui pontos de entrada através da sua API RESTful denominada de `IProducerController`. Esta interface não se encontra aqui representada em detalhe, mas a classe `ProducerController` como implementa a interface `IProducerController` apresenta os mesmos métodos.

Por forma a ser possível gerir instâncias de `Producer` (produtor) existem os métodos `registerProducer`, `getProducer`, `deleteProducer`, `setBroker`, `setStrategy` e `setQueue`. O método `registerProducer` cria novos `Producers`, `getProducer` retorna um `Producer` e `deleteProducer` elimina permanentemente um `Producer`. Em termos de atualizações de dados `setBroker`, `setStrategy` e `setQueue` atualizam grande parte dos dados do `Producer`.

Além destes métodos existem ainda o `connect` e o `close`, que conectam e desconectam os produtores do serviço, respetivamente, e o `send` que é o método utilizado para o envio de mensagens.

ProducerController
-iProducerService : IProducerService
+registerProducer(request : HttpServletRequest, response : HttpServletResponse) : void +getProducer(request : HttpServletRequest, response : HttpServletResponse) : void +deleteProducer(request : HttpServletRequest, response : HttpServletResponse) : void +connect(request : HttpServletRequest, response : HttpServletResponse) : void +send(response : HttpServletResponse, request : HttpServletRequest) : void +close(request : HttpServletRequest, response : HttpServletResponse) : void +setBroker(response : HttpServletResponse, request : HttpServletRequest) : void +setStrategy(request : HttpServletRequest, response : HttpServletResponse) : void +setQueue(response : HttpServletResponse, request : HttpServletRequest) : void

Figura 52: Classe ProducerController.

Na classe ProducerController sempre que um dos seus métodos é invocado este irá invocar um método da interface IProducerService. Esta interface não está aqui representada, mas as assinaturas dos seus métodos correspondem aos métodos públicos da classe ProducerService ilustrada na Figura 53.

Existem vários métodos para gerar instâncias do POJO Broker, sendo que este foi criado com o intuito de evitar que os métodos da interface IProducerMessaging possuissem muitos parâmetros, assim caso se tenha de adicionar um novo parâmetro apenas se modifica o POJO Broker e as assinaturas dos métodos mantêm-se iguais. Esta opção foi também tomada pelo facto de nem todos os sistemas de queues necessitarem de todos os parâmetros existentes, por exemplo o Kafka não necessita do atributo headers que é apenas necessário para o RabbitMQ.

ProducerService
-iProducerActiveMQ5 : IProducerMessaging -iProducerKafka : IProducerMessaging -iProducerRabbitMQ : IProducerMessaging -iProducerZeroMQ : IProducerMessaging -iProducerDAO : IProducerDAO -jwtUtil : JWTUtil -basicAuthUtil : BasicAuthUtil
+registerProducer(registry : Registry, token : String) : Producer +getProducer(userId : String, token : String) : Producer +deleteProducer(userId : String, token : String) : Producer +connect(connection : Connection, token : String) : Producer +send(messageReceived : MessageReceived, userId : String, token : String) : void +close(userId : User, token : String) : void +setBroker(brokerUpdate : BrokerUpdate, userId : String, token : String) : Producer +setStrategy(strategyUpdate : StrategyUpdate, userId : String, token : String) : Producer +setQueue(queueUpdate : QueueUpdate, userId : String, token : String) : Producer

Figura 53: Classe SenderService.

A interface IProducerMessaging não está aqui representada pois as classes KafkaProducer (Figura 54), ActiveMQ5Producer (Figura 55), RabbitMQProducer (Figura 56) e ZeroMQProducer (Figura 57) implementam esta, pelo que os seus métodos públicos correspondem à assinatura da interface.

Além dos métodos públicos as classes concretas KafkaProducer, ActiveMQ5Producer, RabbitMQProducer e

ZeroMQProducer possuem o método privado validateStrategy que verifica se a estratégia pretendida é suportada pelo respetivo sistema de queues e se para a estratégia pretendida foram fornecidos todos os campos necessários.

A classe RabbitMQProducer, representada na Figura 56, apresenta ainda o método privado createExchange, porque neste sistema as mensagens são enviadas para Exchanges e não diretamente para as queues. Devido a este pormenor de o RabbitMQ enviar mensagens para Exchanges é mantido um mapeamento de quais os Exchanges já criados e o respetivo tipo do mesmo.

Este mapeamento é realizado por forma a evitar a criação de Exchanges com o mesmo nome, mas tipos diferentes, uma vez que a criação de Exchanges com nomes iguais e tipos diferentes leva ao lançamento de uma exceção.

KafkaProducer
-kafkaTemplate : KafkaTemplate -gson : Gson
+connect(broker : Broker) : void +send(messageToSend : MessageToSend, broker : Broker) : void +close(producerId : String) : void -validateStrategy(broker : Broker) : void

Figura 54: Classes KafkaProducer.

ActiveMQ5Producer
-jmsQueueTemplate : JmsTemplate -jmsTopicTemplate : JmsTemplate -gson : Gson
+connect(broker : Broker) : void +send(messageToSend : MessageToSend, broker : Broker) : void +close(producerId : String) : void -validateStrategy(broker : Broker) : void

Figura 55: Classe ActiveMQ5Producer.

RabbitMQProducer
-rabbitAdmin : AmqpAdmin -rabbitTemplate : AmqpTemplate -gson : Gson -exchanges : Map<String, String>
+connect(broker : Broker) : void +send(messageToSend : MessageToSend, broker : Broker) : void +close(producerId : String) : void -createExchange(exchangeName : String, exchangeType : String) : Exchange -validateStrategy(broker : Broker) : void

Figura 56: Classe RabbitMQProducer.

ZeroMQProducer
-sockets : Map<String, ZMQSocket> -producers : Map<String, List<String>> -gson : Gson
+connect(broker : Broker) : void +send(messageToSend : MessageToSend, broker : Broker) : void +close(producerId : String) : void -validateStrategy(broker : Broker) : void

Figura 57: Classe ZeroMQProducer.

Por forma a manipular os dados persistidos sobre os produtores a interface IProducerDataDAO, representada pela classe ProducerDataDAO na Figura 58, possui as operações CRUD e além destas duas procuras para verificar se um certo elemento existe, o método existsByUsername é utilizado para verificar se existe algum Producer com o mesmo nome no momento da criação de um novo e o método existsByProducerId verifica se existe um Producer com o uuid fornecido.

ProducerDAO
+create(producerData : Producer) : Producer +delete(producerId : String) : void +update(producerData : Producer) : Producer +getByUsername(username : String) : Optional<Producer> +getByProducerId(producerId : String) : Optional<Producer>

Figura 58: Classe ProducerDataDAO.

Neste componente os únicos dados persistidos são os dados relativos ao ProducerData, ilustrados na Figura 59. Os atributos username e password são as credenciais do produtor, o producerId é um identificador único gerado no momento do registo do produtor e os restantes atributos são os necessários para o correto envio de mensagens para o sistema de queues especificado pelo atributo broker.

<<ORM Persistable>> ProducerData
-username : String -password : String -producerId : String -broker : String -strategy : String -exchange : String -queue : String -headers : String

Figura 59: Classe ProducerData.

Dois POJOs muito relevantes para este componente são o MessageReceived e o MessageToSend representados nas Figuras 60 e 61, respetivamente.

A classe `MessageReceived` contém os dados enviados pelo produtor para o envio de uma mensagem. Nesta o atributo `data` corresponde à mensagem em si para enviar para o sistema de queues, o atributo `timeToWait` é aqui utilizado para ir de encontro ao requisito sobre o envio de mensagens com atraso e os restantes atributos servem para ir de encontro aos requisitos de envio de uma mensagem para uma queue ou estratégia diferente.

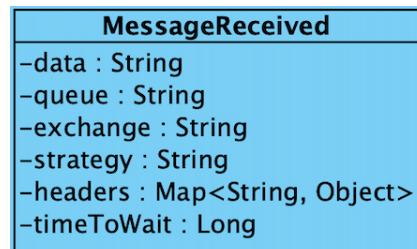


Figura 60: Classe `MessageReceived`.

A classe `MessageToSend` contém os dados a serem enviados para o sistema de queues. Esta é constituída apenas por dois atributos, `data` que corresponde ao corpo da mensagem, valor enviado pelo produtor no atributo `data` do `MessageReceived`, e `messageId` que é um identificador único gerado para a mensagem.

Apesar de as mensagens apenas serem persistidas no componente Receiver, apresentado na Secção 4.3.4.3, os identificadores das mensagens são gerados do lado do componente Sender. Esta decisão foi tomada pelo facto de algumas estratégias enviarem a mesma mensagem para vários consumidores, pelo que se fossem estes a gerar os identificadores a mesma mensagem teria vários identificadores.

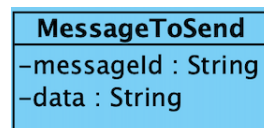


Figura 61: Classe `MessageToSend`.

4.3.4.3 Componente Receiver

O componente Receiver será o serviço responsável pela gestão de consumidores de mensagens, sendo este serviço também responsável por obter as mensagens dos sistemas de queues e enviar para os consumidores. Inicialmente será ilustrada a arquitetura interna deste componente e de seguida serão ilustradas várias classes deste mesmo componente.

Na Figura 62 é possível observar os `inboundadapters` que são os pontos de entrada deste serviço e também os `inboundports` que são os pontos de entrada para o business deste serviço. É possível observar que existe apenas uma interface nos `inboundports` e que a classe `ReceiverController` conhece a mesma.

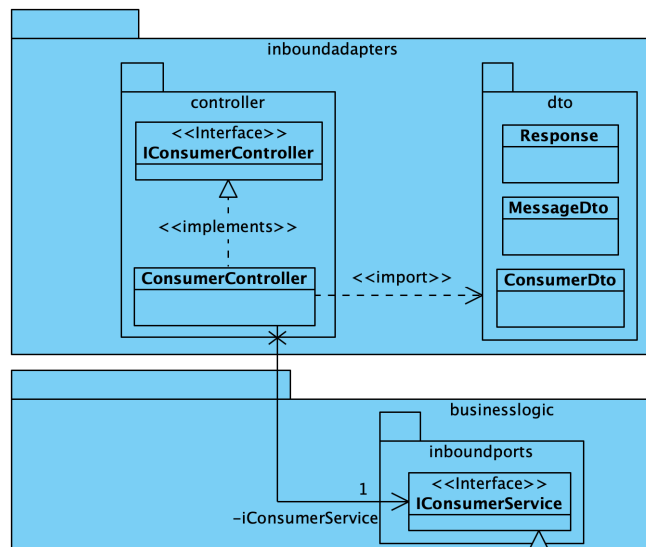


Figura 62: Arquitetura e classes inbounds do componente Receiver.

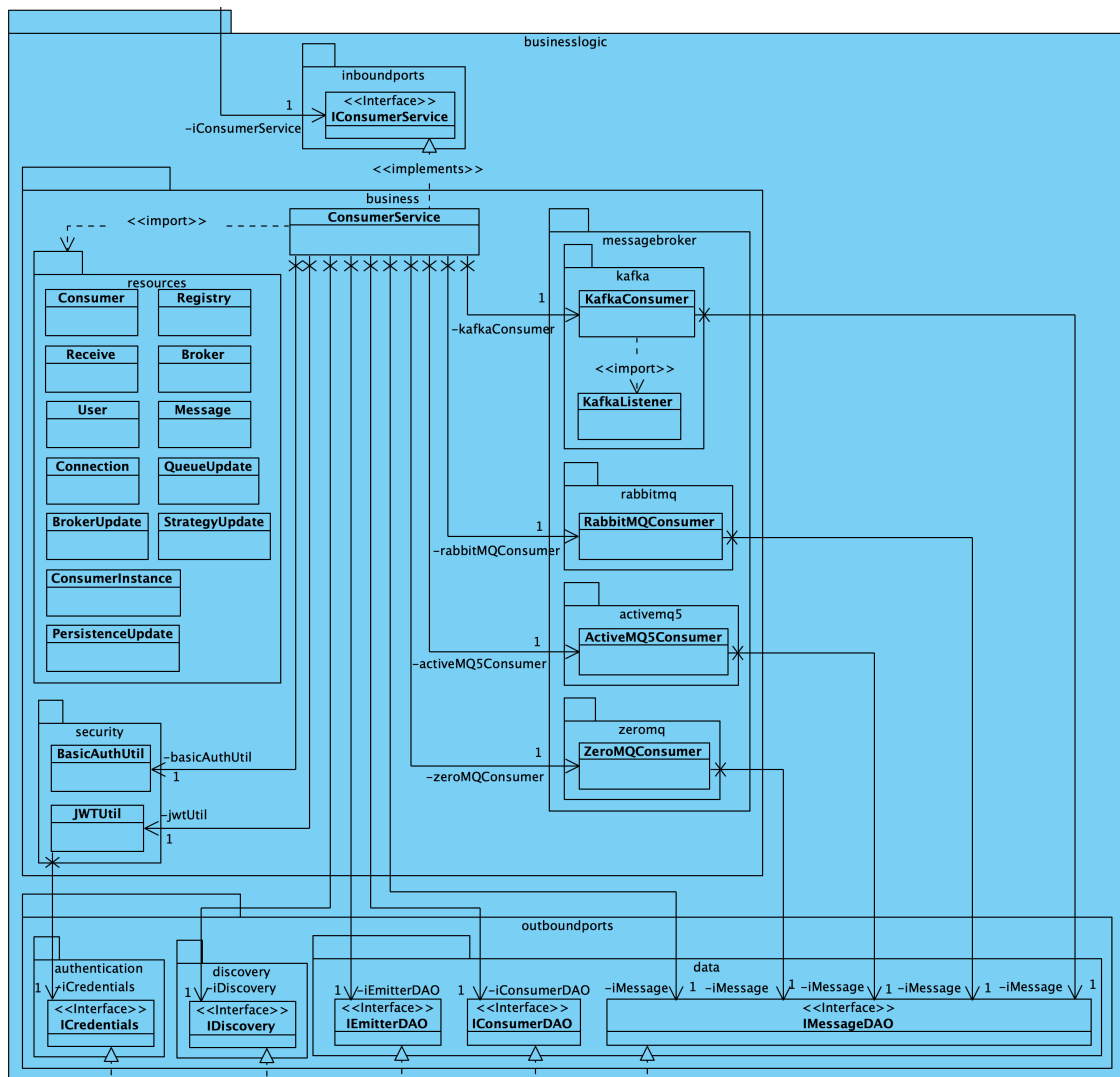


Figura 63: Arquitetura e classes business do componente Receiver.

O package businesslogic, representado na Figura 63, possui toda a lógica de negócio. Como se pode observar a ligação aos sistemas de queues encontra-se neste package e não no package inboundadapters, uma vez que estes são um ponto de entrada do serviço. Esta opção foi tomada pela necessidade de os listeners serem criados dinamicamente de forma a cumprir os requisitos estipulados anteriormente.

O motivo pelo qual os sistemas de queues não se podiam inserir nos inboundadapters deve-se ao facto de que a lógica de negócio necessita de ter controlo sobre os sistemas de queues, então se estes estivessem nos inboundadapters seria quebrada a arquitetura hexagonal em que a comunicação é realizada dos inboundadapters para o businesslogic e do businesslogic para o outboundadapters.

Nas Figuras 62 e 63 o package inboundports está repetido, permitindo assim a visualização das dependências entre os inbounds e o business.

Tal como o inboundports, o outboundports encontra-se replicado nas Figuras 63 e 64 permitindo assim visualizar as dependências entre o business e os outbounds.

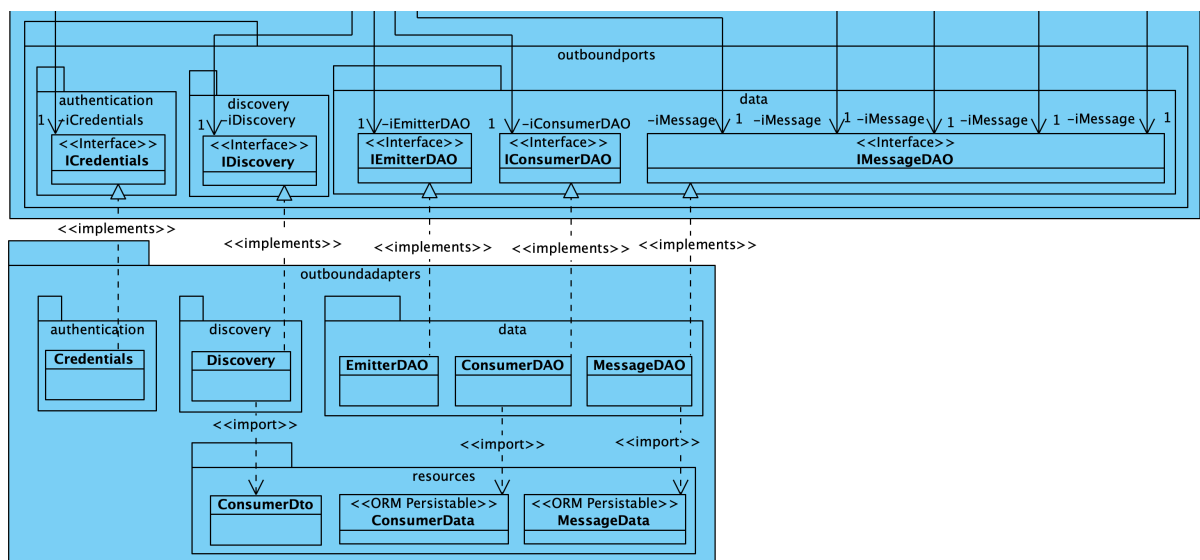


Figura 64: Arquitetura e classes outbounds do componente Receiver.

Este componente apesar de ter dois pontos de entrada apenas apresenta a API Rest no seu inbounds, sendo esta denominada por IConsumerController e aqui representada pela classe ConsumerController na Figura 65.

Ora esta expõe várias rotas sendo o registerConsumer responsável por registar um novo consumidor, getConsumer e deleteConsumer para obter e eliminar os dados de um consumidor respetivamente. Os métodos connect, receive e close são responsáveis pelo consumidor se ligar ao Middleware, abrir a conexão Server Sent-Event (SSE) para obter as suas mensagens e fechar a sua conexão com o Middleware.

Os métodos para atualização de dados relativos ao consumidor são o setBroker, setStrategy, setQueue e setPersistenceTime.

ConsumerController
-iConsumerService : IConsumerService
+registerConsumer(request : HttpServletRequest, response : HttpServletResponse) : void
+getConsumer(request : HttpServletRequest, response : HttpServletResponse) : void
+deleteConsumer(request : HttpServletRequest, response : HttpServletResponse) : void
+connect(request : HttpServletRequest, response : HttpServletResponse) : void
+receive(request : HttpServletRequest, response : HttpServletResponse) : SseEmitter
+close(request : HttpServletRequest, response : HttpServletResponse) : void
+setBroker(request : HttpServletRequest, response : HttpServletResponse) : void
+setStrategy(request : HttpServletRequest, response : HttpServletResponse) : void
+setQueue(request : HttpServletRequest, response : HttpServletResponse) : void
+setPersistenceTime(request : HttpServletRequest, response : HttpServletResponse) : void
+getMessage(request : HttpServletRequest, response : HttpServletResponse) : void
+deleteMessage(request : HttpServletRequest, response : HttpServletResponse) : void

Figura 65: Classe ConsumerController.

A classe ConsumerController e a respetiva interface IConsumerController possuem ainda mais dois métodos que são o getMessage e o deleteMessage. Apesar de estes métodos serem relativos a mensagens encontram-se na mesma classe porque são recursos relativos a consumidores, isto é, cada consumidor tem as suas mensagens e estas apenas existem se o consumidor existir. O método getMessage devolve a mensagem pretendida e o deleteMessage apaga a mesma.

Na Figura 66 está representada a classe ConsumerService que estende a interface IConsumerService, atualmente esta interface é apenas invocada pela classe ConsumerController e cada método desta classe invoca o respetivo método da interface IConsumerService, isto é, o método registerConsumer da classe ConsumerController invoca o método registerConsumer da interface IConsumerService.

ConsumerService
-jwtUtil : JWTUtil
-iDiscovery : IDiscovery
-iEmitterDAO : IEmitterDAO
-kafkaConsumer : KafkaConsumer
-activeMQ5Consumer : ActiveMQ5Consumer
-rabbitMQConsumer : RabbitMQConsumer
-zeroMQConsumer : ZeroMQConsumer
-iConsumerDAO : IConsumerDAO
-iMessageDAO : IMessageDAO
-basicAuthUtil : BasicAuthUtil
+registerConsumer(registry : Registry, token : String) : Consumer
+getConsumer(userId : String, token : String) : Consumer
+deleteConsumer(userId : String, token : String) : Consumer
+connect(connection : Connection, token : String) : Consumer
+receive(userId : String, token : String) : SseEmitter
+close(userId : User, token : String) : String
+setBroker(brokerUpdate : BrokerUpdate, userId : String, token : String) : Consumer
+setStrategy(strategyUpdate : StrategyUpdate, userId : String, token : String) : Consumer
+setQueue(queueUpdate : QueueUpdate, userId : String, token : String) : Consumer
+setPersistenceTime(persistenceUpdate : PersistenceUpdate, userId : String, token : String) : Consumer
+getMessage(messageId : String, userId : String, token : String) : Message
+deleteMessage(messageId : String, userId : String, token : String) : Message
-receive(sseEmitter : SseEmitter, consumer : Consumer) : void

Figura 66: Classe ConsumerService.

No que toca aos sistemas de queues cada um tem uma classe, sendo que neste caso a classe RabbitMQConsumer é relativa ao RabbitMQ, a classe ActiveMQ5Consumer ao ActiveMQ 5, a classe ZeroMQConsumer ao ZeroMQ e a classe KafkaConsumer ao Kafka. Este último necessita ainda de uma classe auxiliar KafkaListener para receber as mensagens enviadas pelo respetivo sistema de queues.

A classe RabbitMQConsumer tal como a RabbitMQProducer presente na Secção 4.3.4.2 possui um mapeamento de Exchanges criados permitindo assim evitar pedidos desnecessários ao sistema de queues quando estes já existem com um tipo diferente.

Além do mapeamento de Exchanges existe um mapeamento dos listeners que estão atualmente à escuta no sistema de queues. Este mapeamento é mantido para quando é feita uma atualização ao consumidor ao dados do consumidor, dados do consumidor relativos aos sistemas de queues, ser possível obter o listener atual para o terminar e criar um novo com os novos dados.

Este mapeamento dos listeners também existe nas restantes classes dos consumers pelo mesmo motivo, para ser possível terminar os listeners atuais e criar novos quando os dados dos consumidores são atualizados.

RabbitMQConsumer
-rabbitTemplate : AmqpTemplate -rabbitAdmin : AmqpAdmin -connectionFactory : ConnectionFactory -iMessageDAO : IMessageDAO -consumers : ConcurrentMap<String, SimpleMessageListenerContainer> -gson : Gson -exchanges : ConcurrentMap<String, String>
+connect(broker : Broker) : void +receive(receive : Receive) : void +close(consumerId : String) : void -createQueue(queue : String) : Queue -createBinding(queue : Queue, exchange : Exchange, routingKey : String) : Binding -createExchange(exchangeName : String, exchangeType : String) : Exchange -validateStrategy(broker : Broker) : void

Figura 67: Classe RabbitMQConsumer.

ActiveMQ5Consumer
-iMessageDAO : IMessageDAO -connectionFactory : ConnectionFactory -gson : Gson -consumers : ConcurrentMap<String, DefaultMessageListenerContainer>
+connect(broker : Broker) : void +receive(receive : Receive) : void +close(consumerId : String) : void -validateStrategy(broker : Broker) : void

Figura 68: Classe ActiveMQ5Consumer.

Tal como foi referido anteriormente para o KafkaConsumer é necessário a classe auxiliar KafkaListener, sendo que esta possui quatro variáveis de classe relativas a um consumidor. Isto porque será criada uma instância desta

classe por cada consumidor que seja criado.

As variáveis mínimas para esta classe seriam o SseEmitter para enviar os dados ao consumidor e o consumerId para ser possível persistir a mensagem como relativa a este consumidor. Porém foram adicionadas mais duas, a queue e o persistenceTime. Estes permitem evitar acessos a BD para obter dados sobre o consumidor, porque o persistenceTime é necessário para saber se a mensagem tem de ser persistida ou não e a queue é necessária para saber se este consumidor já recebeu esta mensagem por essa mesma queue.

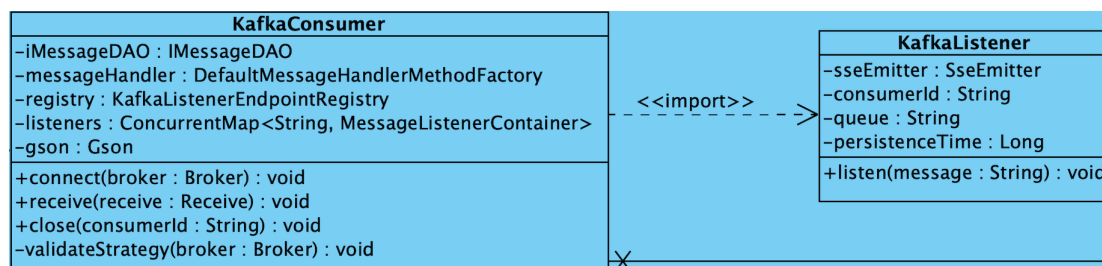


Figura 69: Classe KafkaConsumer.

A classe ZeroMQConsumer possui dois mapeamentos, um para os vários ZMQSockets que são necessários ter abertos por causa das conexões TCP que este utiliza e outro que mapeia os consumidores, a chave de ambos os mapeamentos são a port que está a ser utilizada, ou seja, para a port 6000 existe um ZMQSocket e uma lista de consumidores que o está a utilizar.

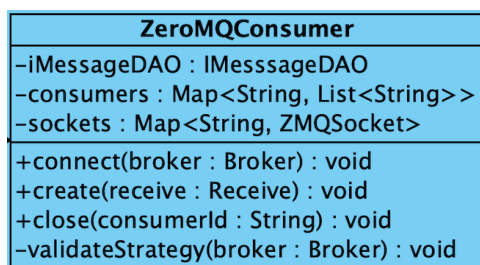


Figura 70: Classe ZeroMQConsumer.

O componente Receiver persiste dois tipos de dados os Consumers e as Messages, na Figura 71 é possível observar a classe MessageDAO responsável por persistir os dados relativos a mensagens. Neste caso a classe MessageData. A classe MessageDAO implementa a interface IMessageDAO presente no package outboundports e esta não é aqui ilustrada porque todos os métodos presentes na classe MessageDAO são as suas assinaturas. Esta permite fazer leituras de mensagens fornecendo uma queue ou um consumerId, permite persistir novas e apagar uma ou mais mensagens.

O método deleteAllByConsumer será apenas utilizado no momento em que um consumidor é eliminado, sendo por isso eliminadas todas as suas mensagens, porque estas apenas existem no caso do consumidor existir.

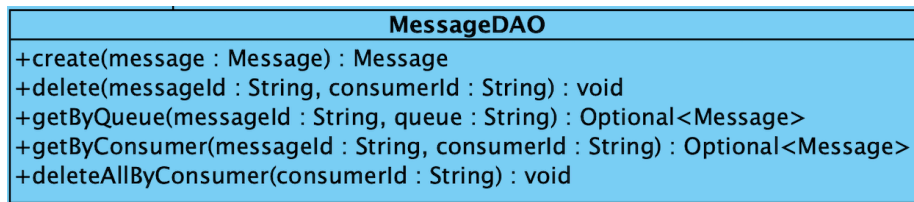


Figura 71: Classe MessageDAO.

Os dados relativos à MessageData que são necessários persistir são os presentes na Figura 72, o messageId é o identificador gerado pelo produtor no momento do envio de uma mensagem para os sistemas de queues, o consumerId identifica a qual consumidor a mensagem pertence, o atributo data contém os dados da mensagem, a queue indica por qual queue a mensagem foi lida e o expireAt é um valor calculado no momento da recepção da mensagem e baseado no persistenceTime fornecido pelo consumidor.

O atributo expireAt permite que uma mensagem seja eliminada automaticamente quando o seu valor passar a ser inferior à data atual.

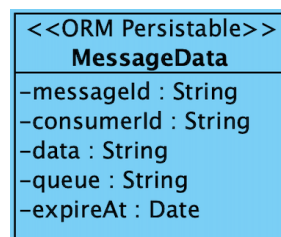


Figura 72: Classe MessageData.

Para persistir e gerir os dados relativos a consumidores a classe ConsumerDAO fornece todos os métodos necessários para o efeito, Figura 73, sendo que esta estende a classe IConsumerDAO que não é aqui ilustrada por a classe ConsumerDAO possuir apenas os métodos desta interface.

Os métodos fornecidos são os básicos para as operações CRUD, sendo que para leitura existem dois métodos distintos, o getByUsername e o getByConsumerId que permitem obter um consumidor pelo seu username e id, respetivamente.

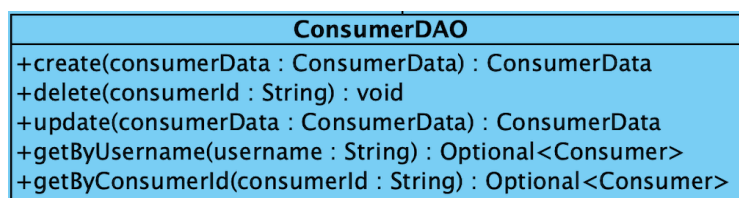


Figura 73: Classe ConsumerDAO.

Os dados necessários persistir relativamente ao consumidor são os presentes na Figura 74 que representa a classe ConsumerData. Os atributos username e password são as credenciais do consumidor e o consumerId é um

identificador único gerado para o consumidor no momento do seu registo. O persistenceTime é o tempo definido pelo consumidor para as suas mensagens ficarem persistidas, no caso deste valor ser 0 as mensagens deste consumidor não são persistidas. Os restantes atributos são os atributos mínimos para o correto funcionamento com qualquer sistema de queues atualmente suportado.

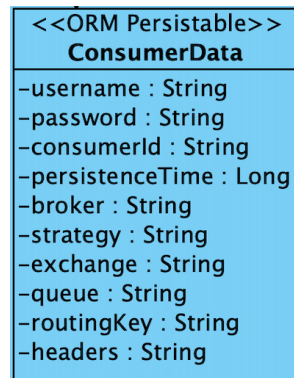


Figura 74: Classe ConsumerData.

Enquanto o ConsumerDAO e o MessageDAO fazem a ponte entre a BD e a lógica de negócio, a classe EmitterDAO representada na Figura 75 e que estende a interface IEmitterDAO, faz o mesmo trabalho com a nuance que os seus dados não são persistidos para uma BD.

Os dados deste Data Access Object (DAO) não são persistidos para BD porque são relativos às conexões SSE que estão abertas e estas só existem se o componente estiver ativo, pelo que se o componente for reiniciado por alguma falha todas as conexões se fechariam automaticamente e não faria sentido iniciá-las novamente quando o componente voltasse a ficar ativo. Isto porque do lado dos recetores a conexão também seria fechada no momento em que o componente fosse reiniciado.

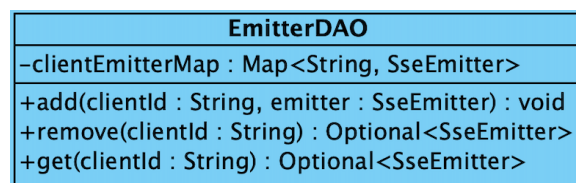


Figura 75: Classe EmitterDAO.

O último outbound a abordar sobre este componente é a classe Discovery, Figura 76, que estende a interface IDiscovery. Esta classe faz a ligação com o componente Discovery, que será apresentado na Secção 4.3.4.4, através de um RabbitMQ.

Apenas é necessário um método que serve para registar no componente Discovery que um consumidor abriu uma conexão com uma réplica em específico. O objeto consumer do tipo ConsumerInstance, representado na Figura 77, é utilizado para passar todos os dados necessários para o componente Discovery.

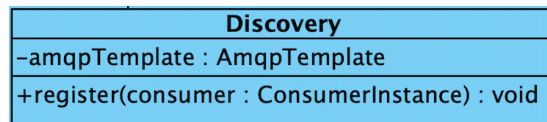


Figura 76: Classe Discovery.

O POJO ConsumerInstance é constituído apenas por três atributos, o consumerId que identifica qual o consumidor a registar, a replica que indica qual o Internet Protocol (IP) da réplica e o token que é o token de autenticação do utilizador.

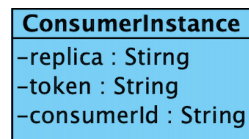


Figura 77: Classe ConsumerInstance.

4.3.4.4 Componente Discovery

O componente Discovery será responsável por encaminhar alguns dos pedidos HTTP para as réplicas corretas do componente Receiver. Uma vez que o componente Receiver possui um pedido que retorna um SSE isso faz com que um consumidor fique ligado a uma réplica específica, pelo que alterações feitas aos dados deste consumidor tem de ser efetuados diretamente a essa réplica para surtir efeitos imediatos.

As Figuras 78 a 80 representam a arquitetura interna do componente Discovery.

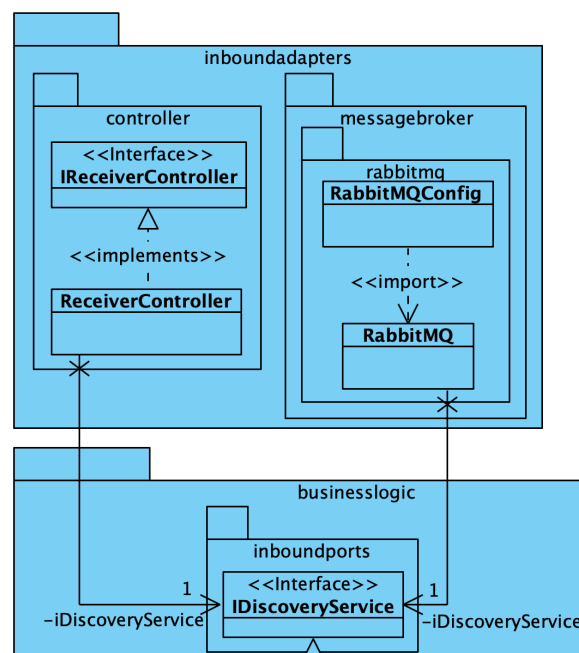


Figura 78: Arquitetura e classes inbounds do componente Discovery.

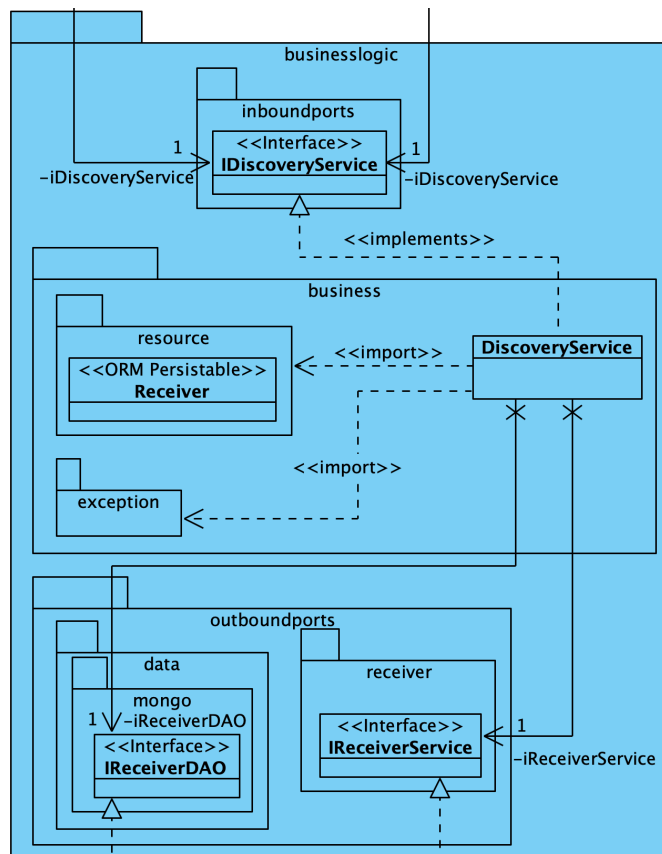


Figura 79: Arquitetura e classes business do componente Discovery.

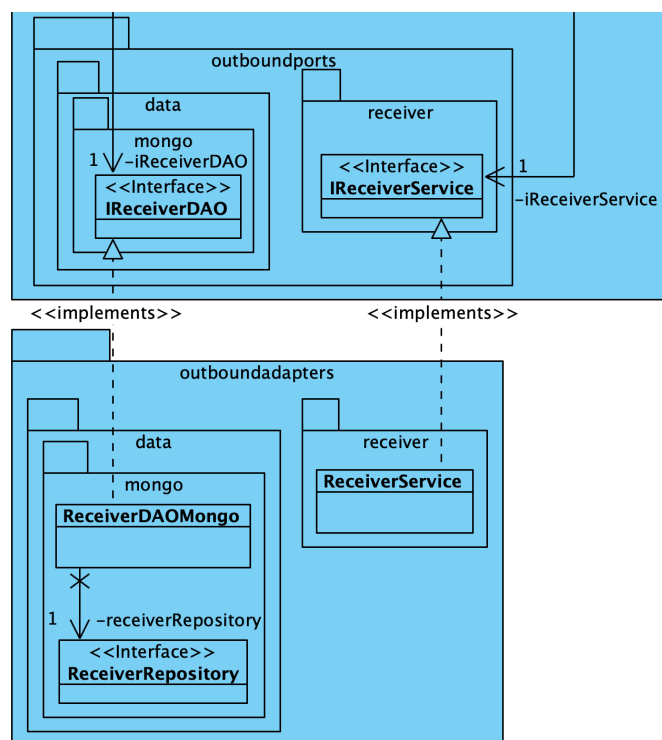


Figura 80: Arquitetura e classes outbounds do componente Discovery.

Ao invés dos outros componentes que apenas possuíam pontos de entrada através da sua API RESTful, sendo que este componente tem dois pontos de entrada distintos, a sua API RESTful representada pela interface IReceiverController que não é aqui ilustrada em detalhe, mas sim a classe ReceiverController que implementa esta e se encontra representada na Figura 81. O outro ponto de entrada é através de um sistema de queues, devido à necessidade de este componente receber comunicação assíncrona tal como foi referido na Secção 4.3.2.

Este segundo ponto de entrada encontra-se representado na Figura 82.

ReceiverController
-iDiscoveryService : IDiscoveryService
+closeReceiver(request : HttpServletRequest, response : HttpServletResponse) : void
+deleteReceiver(request : HttpServletRequest, response : HttpServletResponse) : void
+setBroker(request : HttpServletRequest, response : HttpServletResponse) : void
+setStrategy(request : HttpServletRequest, response : HttpServletResponse) : void
+setQueue(request : HttpServletRequest, response : HttpServletResponse) : void
+setPersistenceTime(request : HttpServletRequest, response : HttpServletResponse) : void

Figura 81: Classe ReceiverController.

Os pontos de entrada da API RESTful deste serviço são o closeReceiver, deleteReceiver, setBroker, setStrategy, setQueue e setPersistenceTime, todos estes pedidos tem de ser enviados à réplica específica de Receiver no caso de o consumidor possuir um SSE aberto. No caso de não existir SSE aberto o pedido é encaminhado para qualquer umas das réplicas Receiver que esteja ativa.

Quando qualquer um destes métodos é invocado será invocado o respetivo método da interface IDiscoveryService aqui representada através da classe DiscoveryService na Figura 83.

A classe ReceiverController possui este nome uma vez que todos os pontos nela presente são relativos ao Receiver, sendo que se no futuro for necessário reaproveitar este componente para reencaminhar pedidos para réplicas específicas de Sender será criada a interface ISenderController e a classe SenderController.

É possível observar através das Figuras 81 e 83 que a classe DiscoveryService possui um método público extra em relação à classe DiscoveryController e isto deve-se a que esse método extra, createReceiver, ser invocado através da classe RabbitMQ, presente na Figura 82.

RabbitMQ
-iDiscoveryService : IDiscoveryService
-rabbitAdmin : AmqpAdmin
-connectionFactory : ConnectionFactory
-gson : Gson
+container(listenerAdapter : MessageListener) : SimpleMessageListenerContainer
+listenerAdapter() : MessageListener

Figura 82: Classe RabbitMQ.

A classe DiscoveryService, Figura 83, será responsável por toda a lógica de negócio verificando se existem dados relativos ao consumidor que invocou o pedido e no caso de existirem encaminhar este pedido para a interface

IReceiverService com a rota para a réplica específica. Caso não exista informação sobre esse consumidor a rota será dirigida a qualquer umas das réplicas do componente Receiver.

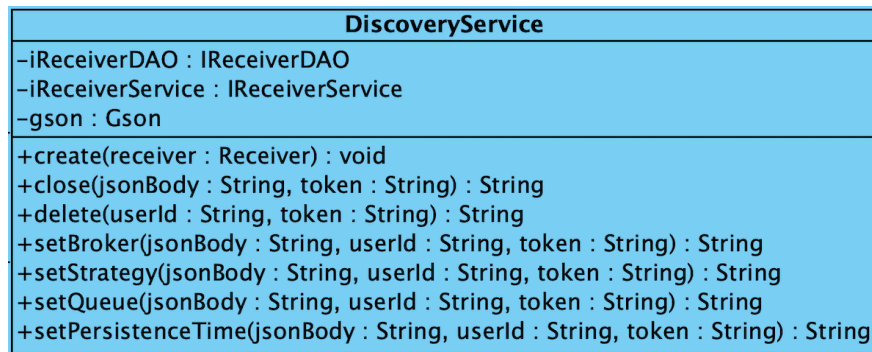


Figura 83: Classe DiscoveryService.

Tal como nos outros componentes a classe DiscoveryService está envolvida no padrão arquitetural Adapter ao apenas conhecer as interfaces do package outboundports, sendo depois estas implementadas pelas classes do package outboundadapters que são os adaptadores neste padrão.

A gestão dos dados relativos à localização dos consumidores é efetuada através da classe ReceiverDAO, representada na Figura 84, que implementa a interface IReceiverDAO. Esta não se encontra aqui ilustrada porque as assinaturas dos métodos correspondem aos métodos da classe ReceiverDAO. Em termos de gestão é possível efetuar as operações CRUD e existe ainda o método *exists* que permite verificar se o consumidor em questão existe neste componente.

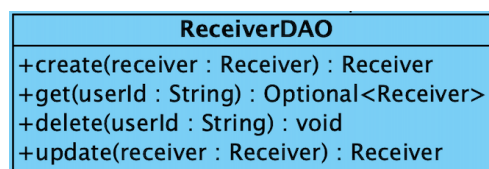


Figura 84: Classe ReceiverDAO.

Os dados persistidos pelo componente Discovery são os apresentados na Figura 85. São apenas persistidos estes dois valores porque são os necessários para identificar um receiver, userId, e a réplica onde se encontra a sua conexão SSE aberta, replica.

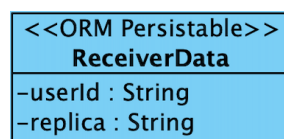


Figura 85: Classe ReceiverData.

Por forma a ser possível realizar a comunicação deste componente com o componente Receiver, a classe ReceiverService, representada na Figura 86, faz a comunicação síncrona através de pedidos HTTP. Para remoção de código duplicado existe o método privado makeRequest que realiza o pedido em si, sendo que os restantes métodos preparam o pedido HTTP para depois ser realizado no método makeRequest.

Esta classe implementa a interface IReceiverService.

ReceiverService
+close(token : String, replica : String, jsonBody : String) : String +delete(token : String, replica : String, userId : String) : String +broker(token : String, replica : String, jsonBody : String, userId : String) : String +strategy(token : String, replica : String, jsonBody : String, userId : String) : String +queue(token : String, replica : String, jsonBody : String, userId : String) : String +persistenceTime(token : String, replica : String, jsonBody : String, userId : String) : String

Figura 86: Classe ReceiverService.

4.3.4.5 Componente ZeroMQBroker

O diagrama de classes do componente ZeroMQBroker não está aqui representado pois este não será implementado, pelo que não chegou a ser realizado. Este não será implementado porque a complexidade de criar este componente não iria trazer qualquer valor ao Middleware a desenvolver, uma vez que este componente iria ser apenas mais um sistema de queues suportado pelo Middleware.

Apesar do componente ZeroMQBroker não ser implementado este foi tido em conta nos diagramas de classes dos componentes Sender e Receiver.

5 Implementação

No presente capítulo será ilustrada a implementação do Middleware, para tal primeiro serão abordadas as tecnologias utilizadas para a implementação, bem como as vantagens e desvantagens que cada uma fornece.

5.1 Tecnologias

A presente secção encontra-se dividida em três subsecções, na primeira subsecção 5.1.1 são descritas as tecnologias utilizadas para o desenvolvimento dos componentes do Middleware a desenvolver, na segunda subsecção 5.1.2 são apresentadas as tecnologias utilizadas para testar os componentes durante o processo de desenvolvimento e na terceira subsecção 5.1.3 são descritas as tecnologias utilizadas para o deployment do projeto como um todo.

As tecnologias utilizadas são iguais para o serviço Registration e para o serviço Middleware.

5.1.1 Desenvolvimento

Durante o desenvolvimento do Middleware foram utilizadas várias tecnologias por forma a ser possível atingir o objetivo pretendido.

Primeiramente escolheu-se qual a linguagem de programação a utilizar, sendo que esta escolha recaiu sobre Java, pelo maior à vontade nesta linguagem e pelo seu poder no que toca a abstrações, que permite a reutilização de código.

Após a escolha da linguagem de programação recorreu-se ao Visual Paradigm⁴⁹ para a construção dos diagramas apresentados no Capítulo 4, sendo que este foi crucial para as decisões arquiteturais do Middleware a desenvolver. Por forma a tirar o máximo partido da linguagem de programação escolhida foi utilizada a *framework Spring Boot*⁵⁰. Esta *framework* tem já desenvolvido código para 3 dos sistemas de queues a implementar (RabbitMQ, ActiveMQ e Kafka) e tem também o Web MVC que permite expor um serviço HTTP seguindo o modelo Model - View - Controller (MVC). Além destes também traz já integração com várias BDs, tais como MongoDB, MySQL, PostgreSQL, entre outras, e fornece também uma biblioteca de validação tirando alguma complexidade do código.

No Middleware desenvolvido os motores de BDs utilizados foram o PostgreSQL, sendo que este foi utilizado pelo à vontade e conhecimento já adquirido sobre o mesmo.

A biblioteca de validação da *framework* denomina-se de *spring-boot-starter-validation* e permite a adição de anotações antes das variáveis da classe. Quando é invocado um método HTTP com Body a *framework* consegue construir um POJO diretamente através dos dados contidos no Body e no caso de algum dos parâmetros não corresponder às anotações colocadas será lançada uma exceção, sendo que isto faz com que durante o código não seja preciso verificar se os campos tem o formato correto porque a *framework* tratou disso automaticamente quando recebeu o pedido HTTP.

⁴⁹Visual Paradigm: <https://www.visual-paradigm.com/>

⁵⁰Spring Boot: <https://spring.io>

Com a utilização da framework Spring Boot é também utilizado o padrão Dependency Injection (DI), em que os objetos recebem outros dois quais dependem. A utilização deste padrão no desenvolvimento do Middleware trás vantagens ao nível da manutenção e extensibilidade deste, tal como referido por Tiago Fernandes em [Fer].

Na Figura 87 é possível observar o objeto IReceiverDAO ser injetado na classe DiscoveryService. Na realidade o objeto injetado é o ReceiverDAO porque é o único que implementa a interface IReceiverDAO.

Neste caso o Spring consegue inferir automaticamente qual é a classe que implementa a interface IReceiverDAO, porém no componente Sender o Facade ProducerService possui 4 injeções do objeto IProducerMessaging, uma para cada sistema de queues, sendo que neste caso é preciso adicionar a anotação Qualifier, na qual se especifica o nome atribuído ao objeto concreto para o Spring conseguir realizar a injeção.

```
@Service
public class DiscoveryService implements IDiscoveryService {

    @Autowired
    private IReceiverDAO iReceiverDAO;
```

Figura 87: Injeção de dependência no Spring Boot.

A framework Spring Boot permite mais facilmente criar vários objetos como Singletons, como por exemplo o ProducerService, apesar de na Secção 4.3.4 ser ilustrado o método getInstance e o construtor privado estes não chegaram a ser implementados explicitamente, uma vez que com a utilização da anotação Service nesta classe faz com que esta seja criada quando o serviço inicia e possa ser injetada nas classes que dependam dela, e esta anotação garante que existe apenas uma instância deste objeto.

Na Figura 88 é possível observar algumas das anotações oferecidas pela biblioteca de validação, a anotação NotBlank força que o texto fornecido não seja constituído por texto vazio, como por exemplo caracteres de indentação. O Pattern força que o texto fornecido siga exatamente o padrão fornecido a esta anotação, por último a anotação NotEmpty força que variáveis que sejam listas ou coleções de chave-valor contenham pelo menos um objeto. Além destas existem outras anotações como Min e Max que para números define um número mínimo e máximo, respetivamente.

```
@NotBlank
@Pattern(regexp = "[a-zA-Z0-9]{8,32}")
private String projectName;

@NotEmpty
private List<String> brokers;

@NotBlank
@Pattern(regexp = "[A-Z]{2}")
private String region;
```

Figura 88: Anotações biblioteca spring-boot-starter-validation.

Como um projeto Spring Boot utiliza o Maven⁵¹ ou o Gradle⁵² para a criação automática do projeto, sendo que neste caso foi utilizado o Maven pela experiência de utilização do mesmo.

Além das dependências já descritas acima foram também inseridos JSON Web Token (JWT), Lombok⁵³ e o OkHttp⁵⁴.

O JWT é utilizado para a autenticação dos utilizadores, no momento em que se ligam ao Middleware cada utilizador recebe um JWT único.

Lombok é um projeto para simplificação de código, sendo que este é utilizado ao nível dos POJOs, permitindo que apenas sejam especificadas as variáveis de classe e através de algumas anotações sejam gerados setters, getters, construtores, equals, entre outros métodos. Sem esta dependência todos os métodos anteriores teriam de ser implementados manualmente. Além disto este projeto permite a colocação da anotação `NonNull` antes das variáveis de classe, fazendo com que seja lançada uma exceção no caso de se tentar passar um valor null para essa mesma variável.

O OkHttp foi utilizado para comunicação síncrona entre microserviços, sendo que neste caso do microserviço Discovery para o Receiver. Esta biblioteca permite a execução dos vários pedidos HTTP existentes.

O projeto foi desenvolvido através do IntelliJ IDEA Ultimate e este permitiu a instalação do plugin SonarLint⁵⁵. Este alerta para *code smells* permitindo assim corrigir algumas partes do código antes de seguir para teste e produção.

Com a utilização destas ferramentas e a conceção previamente realizada na Secção 4 o desenvolvimento deste Middleware foi facilitado.

5.1.2 Teste

Durante a fase de testes das funcionalidades do Middleware recorreu-se ao Docker Compose⁵⁶ para o deployment local de todos os microserviços sem replicação.

Por forma a interagir com os microserviços foi utilizado o Postman⁵⁷. Este permite realizar todo o tipo de pedidos HTTP, tornando assim possível verificar que o resultado de todas as rotas expostas se encontravam corretas.

Com estas duas tecnologias foi possível verificar e corrigir erros de rotas, comportamentos inesperados por parte dos microserviços a desenvolver, entre outros.

No deployment através do Docker Compose foram acrescentados alguns serviços como o Adminer que permitiram observar os dados persistidos, fazendo assim com que fosse possível corrigir erros de dados mal persistidos.

⁵¹ Maven: <https://maven.apache.org/>

⁵² Gradle: <https://gradle.org/>

⁵³ Lombok: <https://projectlombok.org>

⁵⁴ OkHttp: <https://square.github.io/okhttp/>

⁵⁵ SonarLint: <https://www.sonarlint.org/>

⁵⁶ Docker Compose: <https://docs.docker.com/compose/>

⁵⁷ Postman: <https://www.postman.com/>

Foram realizados três tipos de testes, os testes unitários, os testes de integração e os testes de qualidade (carga).

Nos testes unitários foi utilizado apenas o Postman enquanto o microserviço corria dentro do IntelliJ, durante estes testes foram realizadas as operações básicas corrigindo assim qualquer erro que pudesse existir no microserviço testado, garantindo que no final todas as operações básicas tinham o comportamento esperado.

Para os testes de integração já foi utilizado o DockerCompose realizando assim as operações mais complexas que exigiam a interação entre vários componentes.

Uma vez que não existia replicação algumas das operações do microserviço Discovery não eram possíveis de testar, pelo que no Docker Compose criou-se duas instâncias distintas do Receiver. Cada instância tem um nome e IP único sendo na abertura de um SSE enviado ao Discovery o Receiver e o IP, posto isto ao realizar os testes o Discovery conseguiu encaminhar os pedidos para as instâncias Receiver esperadas, testando assim esta funcionalidade que necessitava de replicação.

Os testes de qualidade foram realizados durante o deployment, isto é, quando o Middleware foi colocado no ambiente final (ambiente de deployment) foram realizados testes de carga para avaliar o desempenho do mesmo, sendo que estes serão abordados mais aprofundadamente no Capítulo 7.

5.1.3 Deployment

Atualmente para o deployment do Middleware está-se a recorrer ao Docker Swarm⁵⁸ e o Google Cloud Platform (GCP)⁵⁹.

O GCP divide-se em vários componentes, tendo sido utilizado o Compute Engine⁶⁰. Este permite a criação de várias Virtual Machines (VMs) possibilitando a criação de um Docker Swarm com várias VMs.

Por forma a configurar as várias VMs foi utilizado o Cloud SDK⁶¹ que é uma ferramenta de linha de comandos para gerir todos os projetos e instâncias de projetos.

Uma alternativa ao Docker Swarm passaria pela utilização de Kubernetes⁶². Enquanto que no Docker Swarm a replicação dos serviços é limitada pelo número de instâncias que estão no Swarm e pelo número máximo de instâncias especificadas no ficheiro docker-compose.yml sobre esse serviço, no Kubernetes o número de instâncias é variável sendo o próprio Kubernetes a realizar a gestão de quantas réplicas são necessárias em cada momento, baseando-se para isso na memória, CPU, conexões, entre outros parâmetros para realizar essa mesma gestão.

Esta alternativa é também suportada pelo GCP através do componente Google Kubernetes Engine⁶³.

⁵⁸Docker Swarm: <https://docs.docker.com/engine/swarm/>

⁵⁹Google Cloud Platform: <https://cloud.google.com>

⁶⁰Compute Engine: <https://cloud.google.com/compute/>

⁶¹Cloud SDK: <https://cloud.google.com/sdk>

⁶²Kubernetes: <https://kubernetes.io>

⁶³Google Kubernetes Engine: <https://cloud.google.com/kubernetes-engine/>

5.2 Especificações

Na presente secção serão apresentadas as especificações do Middleware desenvolvido. Inicialmente serão ilustrados quais os sistemas de queues suportados e as respetivas estratégias suportadas por cada um, de seguida na Secção 5.2.2 são apresentados os pontos de acesso para este Middleware.

5.2.1 Sistemas de Queues e Estratégias

Atualmente são suportados três sistemas de queues no Middleware desenvolvido que se encontram representados na Tabela 13 juntamente com as estratégias suportadas por cada um destes sistemas. Nesta é ainda representado o sistema de queues ZeroMQ que apesar de não implementado no Middleware foi abordado durante esta dissertação e fornecendo quais serão as estratégias suportadas para este.

O Middleware desenvolvido aceita apenas, atualmente, como estratégia os nomes: direct, topic, fanout e headers, pelo que qualquer outro valor será rejeitado.

Sistema de Queues	Especificação Middleware	Especificação Sistema de Queues
RabbitMQ	direct	direct
	topic	topic
	fanout	fanout
	headers	headers
ActiveMQ5	direct	queue
	topic	topic
	fanout	—
	headers	—
Kafka	direct	—
	topic	topic
	fanout	—
	headers	—
ZeroMQ	direct	REQ e REP (a desenvolver)
	topic	PUB e SUB (a desenvolver)
	fanout	—
	headers	—

Tabela 13: Mapeamento de especificações entre o Middleware e os sistemas de queues.

As estratégias suportadas para cada um dos sistemas de queues são as mesmas que o próprio sistema de

queues suporta, isto é, o Kafka suporta apenas topic então o Middleware suporta o topic, o ActiveMQ5 suporta o queue e o topic pelo que o Middleware suporta o direct e topic.

Os nomes adotados para as estratégias de comunicação coincidem com os nomes das estratégias do sistema de queues RabbitMQ, sendo que esta decisão foi tomada por o RabbitMQ ser o primeiro a ser implementado durante o desenvolvimento e também porque é o sistema que apresenta um maior número de estratégias suportadas atualmente. No caso de ser adicionado um novo sistema de queues com uma estratégia ainda não suportada pelo Middleware o nome da mesma será igual ou idêntica ao nome da estratégia desse novo sistema.

5.2.2 API Rest

Na presente subsecção serão ilustrados todos os pontos de entrada tanto para o serviço Registration como para o serviço Middleware. Todas as rotas que necessitem de um body este terá de ser texto no formato JavaScript Object Notation (JSON).

Atributo	Tipo de dados
appCode	int32
status	string
message	string
data	object

Tabela 14: Formato Respostas e Exceções personalizadas.

O appCode é utilizado porque existem exceções que utilizam o mesmo código de erro HTTP e então o appCode é uma forma de refinar qual o erro que ocorreu.

O atributo status tem apenas três valores possíveis: SUCCESS, WARNING e ERROR. O valor SUCCESS indica sucesso na execução do pedido realizado, o valor WARNING significa que foi possível realizar o pedido, mas algo não estava totalmente correto e o valor ERROR indica que não foi possível realizar o pedido.

O atributo data pode estar vazio, sendo que isto acontece porque apenas é utilizado na resposta de algumas rotas. O tipo de dados presente neste atributo será um array de valores ou uma estrutura de dados no formato JSON.

A message é o atributo menos relevante já que este é uma simples nota sobre o sucesso ou o insucesso na execução do método, sendo que no caso de insucesso tem uma descrição do que levou ao erro.

Além das exceções que seguem o formato representado na Tabela 14 existem outras, também em formato JSON. Umas são exceções lançadas pelo APIGateway (Kong) que tem o formato presente na Tabela 15 e outras pela framework Spring Boot que tem o formato descrito na Tabela 16.

Atributo	Tipo de dados
message	string

Tabela 15: Formato exceções APIGateway (Kong).

Atributo	Tipo de dados
timestamp	string
status	int32
error	string
message	string
path	string

Tabela 16: Formato exceções framework Spring Boot.

Como existem dois serviços distintos inicialmente será ilustrado o serviço Registration (5.2.2.1), serviço responsável pela gestão de projetos, e de seguida o serviço Middleware (5.2.2.2), serviço desenvolvido para abstrair a utilização dos sistemas de queues.

Variáveis assinaladas com o tipo de dados object indica que estas contem uma estrutura de dados aninhada no formato JSON. No caso de o atributo ser uma lista, o tipo de dados será representado pelo tipo de dados seguido de abrir e fechar parênteses retos ([]).

O tipo de dados string representa uma sequência de caracteres alfanuméricos, juntamente com caracteres de pontuação entre outros caracteres especiais.

Tipo de dados int32⁶⁴ e int64⁶⁵ representam números de 32 e 64 bits, respetivamente.

5.2.2.1 Registration

De seguida serão ilustrados os pontos de entrada do serviço Registration, sendo que os dados necessários, respostas e erros de cada ponto de entrada se encontram nos Anexos B.1.1, B.1.2 e B.1.3, respetivamente.

sendo de seguida especificados todos os parâmetros necessários para cada uma das rotas, bem como as suas restrições em termos de valores. Após estes serão demonstradas as respostas para cada rota e por último as exceções personalizadas e em quais rotas estas são utilizadas.

Na Tabela 17 é possível observar todos os pontos de entrada no serviço Registration através da sua API RESTful. Rotas cujo parte do caminho esteja entre chavetas ({}) indicam que essa mesma parte é variável.

⁶⁴Int32 contem um valor decimal entre -2.147.483.648 e +2.147.483.647.

⁶⁵Int64 contem um valor decimal entre -9.223.372.036.854.775.808 e +9.223.372.036.854.775.807.

Das rotas presentes na Tabela 17 a rota número 1 e 5 não necessitam de qualquer tipo de autenticação, podendo assim ser acedidas livremente. As rotas 2 a 4 necessitam de autenticação através de um JWT gerado na rota número 1. As restantes rotas, 6 e 7, são rotas de administração, as quais são autenticadas por um JWT com uma chave secreta incorporada.

#	Método	Rota
1	POST	/api/projects
2	GET	/api/projects/{projectId}
3	DELETE	/api/projects/{projectId}
4	PUT	/api/projects/{projectId}
5	GET	/api/brokers
6	POST	/api/brokers
7	PUT	/api/brokers/{brokerId}

Tabela 17: API Rest do serviço Registration.

As rotas 1 a 4 são utilizadas para a gestão dos projetos, a rota 1 é utilizada para a criação de novos projetos, depois de criados pode obter-se, apagar-se ou editar-se estes através das rotas 2 a 4, respetivamente. A edição atualmente permite adicionar ou remover sistemas de queues para o projeto em questão.

Com a rota 5 é possível saber quais os sistemas de queues atualmente suportados pelo Middleware e qual a versão que está atualmente em vigor para esse sistema⁶⁶.

Por forma a não ser necessário parar este serviço as rotas 6 e 7 permitem a adição e edição de sistemas de queues, respetivamente. Assim, este componente está sempre atualizado em relação aos sistemas de queues suportados sem a necessidade de reiniciar o mesmo, sendo apenas necessário reiniciá-lo para a adição de novas funcionalidades ou correção de erros.

5.2.2.2 Middleware

De seguida serão ilustrados os pontos de entrada do serviço Middleware, sendo que os dados necessários, respostas e erros de cada ponto de entrada se encontram nos Anexos B.2.1, B.2.2 e B.2.3, respetivamente.

Na Tabela 18 é possível observar todos os pontos de entrada no serviço Registration através da sua API RESTful. Rotas cujo parte do caminho esteja entre chavetas ({}) indicam que essa mesma parte é variável.

⁶⁶As versões atualmente disponíveis são a 3.9.9 para o RabbitMQ, a 2.8.1 para o Kafka e a 5.16.3 para o ActiveMQ5.

#	Método	Rota
1	POST	/api/senders
2	GET	/api/senders/{userId}
3	DELETE	/api/senders/{userId}
4	POST	/api/senders/connect
5	POST	/api/senders/{userId}/send
6	POST	/api/senders/close
7	PUT	/api/senders/{userId}/queue
8	PUT	/api/senders/{userId}/strategy
9	PUT	/api/senders/{userId}/broker
10	POST	/api/receivers
11	GET	/api/receivers/{userId}
12	DELETE	/api/receivers/{userId}
13	POST	/api/receivers/connect
14	GET	/api/receivers/{userId}/receive
15	POST	/api/receivers/close
16	PUT	/api/receivers/{userId}/queue
17	PUT	/api/receivers/{userId}/strategy
18	PUT	/api/receivers/{userId}/broker
19	PUT	/api/receivers/{userId}/persistence
20	GET	/api/receivers/{userId}/message/{messageId}
21	DELETE	/api/receivers/{userId}/message/{messageId}

Tabela 18: API Rest do serviço Middleware.

As rotas 1 a 3 e 7 a 9 são as rotas para as operações CRUD sobre utilizadores do tipo produtor, correspondendo a criação à rota 1, leitura à rota 2, eliminação à rota 3 e atualização às rotas 7 a 9.

As rotas 10 a 12 e 16 a 19 são as rotas para as operações CRUD sobre utilizadores do tipo consumidor, correspondendo a criação à rota 10, leitura à rota 11, eliminação à rota 12 e atualização às rotas 16 a 19.

Para a gestão das mensagens recebidas a rota 20 permite que o utilizador obtenha novamente uma mensagem e a rota 21 permite a eliminação destas.

5.3 Decisões de Implementação

Na presente secção serão ilustradas decisões e assunções realizadas ao nível do desenvolvimento do Middleware em função do conhecimento que foi adquirido.

De seguida na Secção 5.3.1 são ilustradas as decisões tomadas em relação ao RabbitMQ sobre problemas já conhecidos sobre este, na Secção 5.3.2 é relatada a decisão efetuada sobre os tempos de persistência das mensagens e na Secção 5.3.3 é apresentada a assunção realizada sobre o DNS.

5.3.1 RabbitMQ

O RabbitMQ é um sistema de queues bastante diferente dos demais, pois um produtor em vez de publicar uma mensagem para uma Queue publica esta para um Exchange, sendo depois responsabilidade do Exchange entregar a mensagem a uma ou mais Queues dependendo do tipo de Exchange (direct, topic, fanout ou headers).

Atualmente existe um problema com os Exchanges já documentado na própria documentação do RabbitMQ (<https://www.rabbitmq.com/ae.html>), sendo que este problema consiste em que quando uma mensagem é entregue a um Exchange e este não consegue entregar a mensagem a nenhuma Queue. Esta acaba por ser descartada sem conhecimento do produtor que a produziu.

A forma de resolver este problema é configurar um Alternate Exchange (AE) sendo este passado aos restantes Exchanges no momento da sua criação.

Um outro problema com o RabbitMQ é relativo às Queues e este pode ocorrer devido a 3 ocasiões (<https://www.rabbitmq.com/dlx.html>). As ocasiões são as seguintes:

- Mensagem expirada por TTL.
- Mensagem descartada pela Queue por esta ter atingido o seu limite de mensagens persistidas simultaneamente.
- Consumidor confirmou negativamente a receção da mensagem, sendo que isto é diferente de ocorrer um erro na receção da mensagem.

Este problema é resolvido de forma idêntica ao anterior configurando um Dead Letter Exchange (DLX) e passando este aos restantes Exchanges no momento da sua criação.

Tanto a solução de implementar um Alternate Exchange, como um Dead Letter Exchange não foram atualmente implementadas por não acrescentar qualquer valor ao Middleware que se pretendia desenvolver, pois este é um problema relativo apenas a um dos sistemas de queues suportados.

5.3.2 Tempo de Persistência das Mensagens

Um consumidor pode ter um persistenceTime, ver Tabela 38, máximo correspondente ao valor máximo de um long⁶⁷ (int 64) em Java, porém a data limite utilizada é calculada através do tempo especificado pelo consumidor mais o tempo atual em milissegundos, sendo que isto pode levar a um overflow do long (int64) em Java.

⁶⁷Valor máximo do long é 9.223.372.036.854.775.807.

Quando um overflow ocorre em Java o valor torna-se negativo, por exemplo, somar o valor máximo do long mais 1 é igual ao valor mínimo do long⁶⁸ em Java.

Para a realização da eliminação por tempo da mensagem esta tem um atributo do tipo Date e este aceita como parâmetro um long que irá corresponder ao tempo em milissegundos a partir da meia-noite do dia 1 de janeiro de 1970. Ora se o long passado no momento da criação de um Date for negativo este terá uma data anterior a 1970 o que levará a mensagem a não ser persistida porque essa data já passou.

Por forma a resolver o problema dos overflows é verificada se a soma do tempo de persistência especificado pelo consumidor e o tempo atual em milissegundos dá um valor negativo. No caso da soma dar um valor positivo utiliza-se o valor obtido na soma, caso contrário utiliza-se o valor máximo para a persistência da mensagem.

No caso de a soma dar positiva é garantido que não existiu um overflow porque a soma do valor máximo permitido para um long com ele próprio é igual a -1. Isto é garantido até ao ano 292.278.994, pois a partir desse ano a leitura em milissegundos do tempo atual dará um valor negativo, caso não seja alterado, pelo que a soma do valor lido com o tempo especificado pelo utilizador poderá dar um valor positivo apesar de ter ocorrido um overflow, fazendo com que a mensagem não seja persistida.

No entanto esta decisão no caso de não se poder somar os dois valores por causa de overflow leva a que a mensagem não fique persistida durante o tempo especificado pelo utilizador, porém esta fica persistida até ao ano 292.278.994.

Apesar desta decisão sobre o tempo de persistência da mensagem devido ao sistema de BD utilizado as mensagens não são eliminadas quando esse tempo passa. Este problema pode ser resolvido utilizando Jobs da framework Spring Boot, mas estes não foram implementados uma vez que em nada prejudicam os consumidores.

5.3.3 Domain Name System (DNS)

No momento do registo de um projeto na resposta é retornado o DNS de forma a não retornar o IPv4 para o utilizador, rota 1 da Tabela 27.

Porém este DNS retornado não é atualmente válido por 2 motivos, não está a ser verificado pelo que pode existir um DNS igual e no caso de não existir não está a ser registado.

Optou-se por retornar assim o DNS assumindo que existe um algoritmo para verificação e registo de DNS, sendo o DNS retornado o resultado final desse tal algoritmo.

Apesar disto ser um problema, como o deployment do Middleware é realizado no GCP este permite a colocação de IP externos estáticos nas máquinas virtuais, pelo que os Middlewares possuem atualmente IP externos estáticos já que não existe um DNS verdadeiro para a resolução do IP da máquina virtual.

Este IP estático podia ser retornado no momento do registo do projeto se o deployment do Middleware estivesse automático, mas como atualmente este é manual é então retornado o possível DNS a utilizar para a instância do Middleware.

⁶⁸Valor mínimo do long é -9.223.372.036.854.775.808.

6 Caso de Estudo

Com o intuito de demonstrar as funcionalidades e a utilização deste Middleware foi criado um caso de estudo. Este caso de estudo além de demonstrar as funcionalidades do sistema foi utilizado para a realização dos testes de desempenho presentes no Capítulo 7.

Durante a concepção e implementação do Middleware salientou-se que existiam dois tipos de utilizadores, os produtores (Producers/Senders) e os consumidores (Consumers/Receivers), e que existem aplicações ou micro-serviços que apenas produzem ou consomem mensagens, mas existem também alguns que produzem e consomem ao mesmo tempo, pelo que o caso de estudo é um serviço que consome e produz mensagens.

Optou-se por este serviço do caso de estudo coincidir com as aplicações ou microserviços que são simultaneamente produtores e consumidores de mensagens para não existir a necessidade de criar dois serviços distintos para demonstrar as funcionalidades do Middleware.

O caso de estudo criado gera inicialmente 4 consumidores e 1 produtor. Foram criados vários consumidores de forma a ser possível representar várias das funcionalidades desenvolvidas.

Os consumidores são criados com as especificações presentes na Tabela 19, apesar da password ser um campo obrigatório não foi aqui representado porque sendo um caso de estudo foi utilizado o valor password para todos os consumidores.

Campos	Consumidor 1	Consumidor 2	Consumidor 3	Consumidor 4
username	consumer1	consumer2	consumer3	consumer4
broker	rabbitmq	rabbitmq	rabbitmq	kafka
strategy	direct	direct	topic	topic
exchange	teste	teste	temporary	—
queue	studycase	xpto	example	studycase
routingKey	—	teste	example	—
persistenceTime	30.000	0	0	15.000

Tabela 19: Especificações dos consumidores criados.

No caso do consumidor 1 como o sistema de queues (broker) é o rabbitmq e a estratégia é direct, o campo routingKey não é obrigatório ficando depois com o valor do campo queue.

Os consumidores criados poderiam ter mais atributos, porém com os sistemas de queues e estratégias seleccionadas estes não seriam utilizados pelo que não foram adicionados.

O valor 30.000 e 15.000 para o atributo persistenceTime indica que as mensagens recebidas por estes produtores ficaram persistidas apenas por 30 e 15 segundos, respetivamente.

Na Tabela 20 estão especificadas os valores dos campos para os produtores criados, tal como nos consumidores o campo password não é especificado porque o valor é igual para todos.

Campos	Produtor 1
username	producer1
broker	rabbitmq
strategy	direct
exchange	teste
queue	studycase

Tabela 20: Especificações dos produtores criados.

Probabilidades dos eventos:

1. Enviar mensagem utilizando a queue por omissão registada no Middleware: 30%
2. Enviar mensagem para uma queue diferente da que está registada no Middleware: 30%
3. Enviar mensagem com uma estratégia diferente da que está registada no Middleware: 30%
4. Reler uma mensagem que esteja persistida: 10%

Estes eventos ocorrem em intervalos de 3 segundos.

O evento de enviar uma mensagem para queue diferente força o produtor a enviar esta para a queue "teste" e o evento de enviar uma mensagem para uma estratégia diferente força o produtor a enviar esta para a estratégia "topic" com um campo extra, sendo que este campo é o exchange com o valor "temporary". O campo exchange tem de ser adicionado uma vez que o rabbitmq necessita deste campo para enviar para uma estratégia diferente.

Para o evento de reler uma mensagem é escolhido aleatoriamente um consumidor e de seguida algum identificador de uma mensagem que já tenha lido.

6.1 Demonstração

Por forma a ilustrar o caso de estudo a funcionar este sempre que efetua um dos 4 eventos anteriormente descritos imprime para o terminal do IntelliJ IDEA Ultimate os dados relevantes a esse evento. Além destes eventos os consumidores sempre que recebem uma mensagem imprimem a mesma para o terminal.

Para demonstrar as restantes funcionalidades do Middleware, como a alteração de sistemas de queues, estratégia e queue, recorreu-se ao Postman para efetuar os pedidos de alteração de dados, tanto sobre os produtores como os consumidores.

Após 15 segundos ocorreram os eventos presentes na Figura 89, sendo que nesta é possível observar que quando o producer1 envia mensagens para a queue por omissão estas são recebidas pelo consumer1, porém quando este envia para uma queue diferente quem recebe as suas mensagens é o consumer2 e quando envia para

uma estratégia diferente quem recebe é o consumer3.

Na Figura 89 os retângulos a amarelo representam o envio de mensagens por parte do producer1. Os dois primeiros envios ocorreram para a queue por omissão, tendo depois o terceiro e quinto envios feitos para uma queue diferente e o quarto envio para uma estratégia diferente. As restantes cores representam diferentes consumidores.

```
Producer producer1 send message to queue studycase
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.,"data":null}

Consumer consumer1 received: {"messageId":"5c194648-7be0-4781-8964-cf6582510ed8","data":"Hello from producer1!"}

Producer producer1 send message to queue studycase
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.,"data":null}

Consumer consumer1 received: {"messageId":"7a9517c3-9b83-4dbb-a516-c666188560d8","data":"Hello from producer1!"}

Producer producer1 send message to queue teste
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.,"data":null}

Consumer consumer2 received: {"messageId":"4db058a5-4a1a-4a9e-af6b-431264f72f5c","data":"Hello from producer1 to"}

Producer producer1 send message with strategy topic
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.,"data":null}

Consumer consumer3 received: {"messageId":"4c8f53e9-62c6-4e16-bcd3-2dfb4a695800","data":"Hello from producer1 to"}

Producer producer1 send message to queue teste
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.,"data":null}

Consumer consumer2 received: {"messageId":"96726455-ee89-45c4-a64d-e2a76e8d8f64","data":"Hello from producer1 to"}
```

Figura 89: Envio de mensagens e leituras das mesmas.

Na Figura 90 é ilustrado o pedido efetuado ao Middleware desenvolvido de forma a alterar o sistema de queues do producer1 e na Figura 91 encontram-se as impressões dos eventos realizados após esta alteração, com a exceção da primeira impressão que corresponde à última da Figura 89.

Como se pode observar na Figura 91 o producer1 continuou a enviar mensagens para a queue "studycase", mas a para o sistema de queues Kafka em vez do RabbitMQ tal como estava inicialmente. Isto é comprovado com a mensagem a ser recebida pelo consumer4, que está a utilizar o Kafka, em vez do consumer1 que as estava a receber inicialmente no RabbitMQ.

Na Figura 91 a vermelho encontra-se o último evento da Figura 89, a amarelo o envio do producer1 para a queue por omissão, após a sua atualização e azul claro a receção da mesma por parte do consumer4. Nesta encontra-se ainda a azul escuro o evento de ler uma mensagem por parte do consumer3, que resultou numa mensagem de erro de que a mensagem pretendida já não existia, sendo que isto ocorreu porque este consumidor está com a persistência de mensagens desativada.

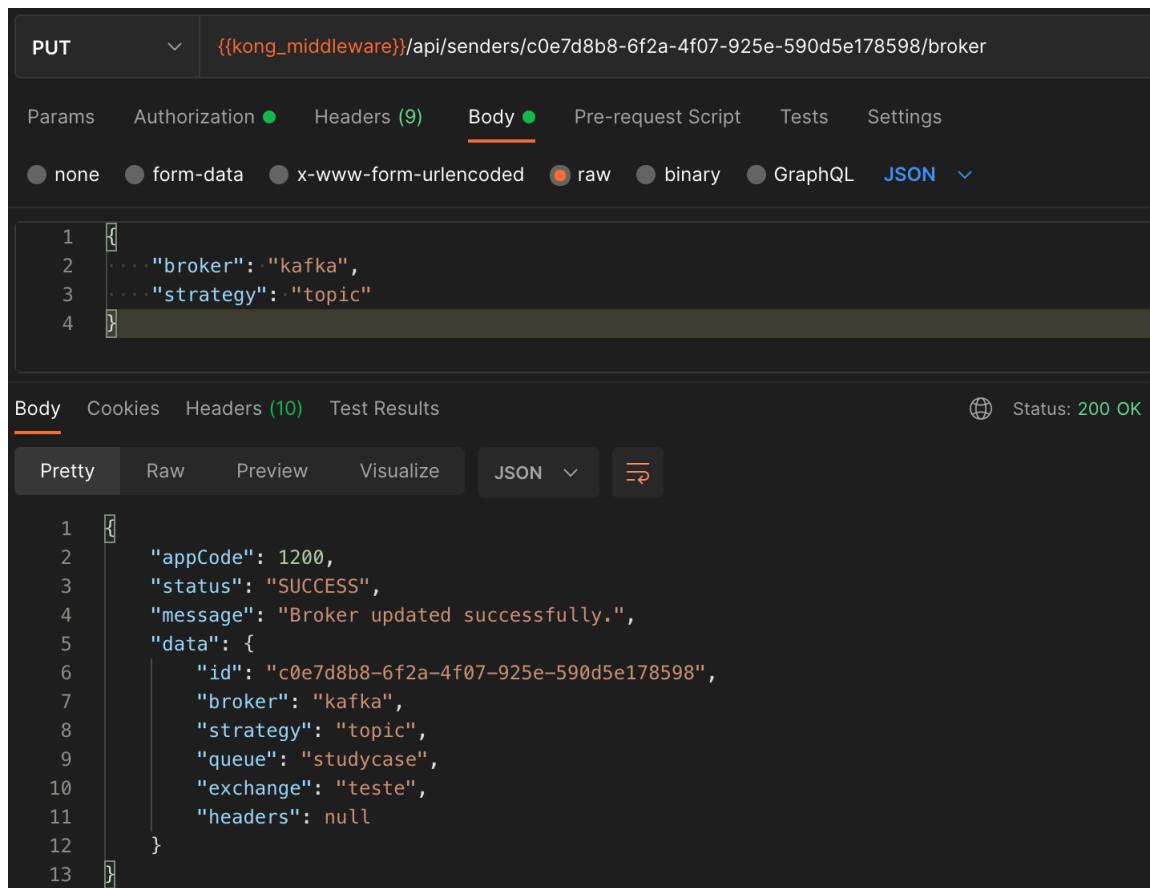


Figura 90: Alteração do Sistema de queues no producer1.

```

Producer producer1 send message to queue teste
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.,"data":null}

Consumer consumer2 received: {"messageId":"96726455-ee89-45c4-a64d-e2a76e8d8f64","data":"Hello from producer1 to

Producer producer1 send message to queue studycase
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.,"data":null}

Consumer consumer4 received: {"messageId":"d0197ab8-fd9c-4564-9c36-34dc098c7240","data":"Hello from producer1!"}

Consumer consumer3 read message 4c8f53e9-62c6-4e16-bcd3-2dfb4a695800 again
Response: {"appCode":1407,"status":"ERROR","message":"Message with ID 4c8f53e9-62c6-4e16-bcd3-2dfb4a695800 don't

```

Figura 91: Envio e leitura de mensagens após alteração do Sistema de queues do producer1.

Com as alterações anteriores o evento de enviar mensagens para uma estratégia deixou de fazer sentido, pois este faz com que o producer1 envie mensagens com a estratégia topic, mas essa já é a sua estratégia por omissão após a alteração efetuada na Figura 90. Porém o evento de enviar mensagens para uma queue diferente continua a ocorrer, mas estas não estão a ser impressas pelo caso de estudo porque não existe atualmente nenhum consumidor à escuta nessa queue.

Após as alterações é possível observar que o producer1 continuou a funcionar sem ter sido necessário reiniciá-lo.

Na Figura 92 é possível observar o pedido realizado ao Middleware desenvolvido de forma a alterar o sistema de queues do consumer2, além do sistema de queues foi também a estratégia pois o sistema de queues Kafka apenas funciona com a estratégia topic e ainda a queue, de forma a que este consumidor capte as mensagens enviadas pelo producer1 quando as mesmas são enviadas para a queue diferente da queue por omissão.

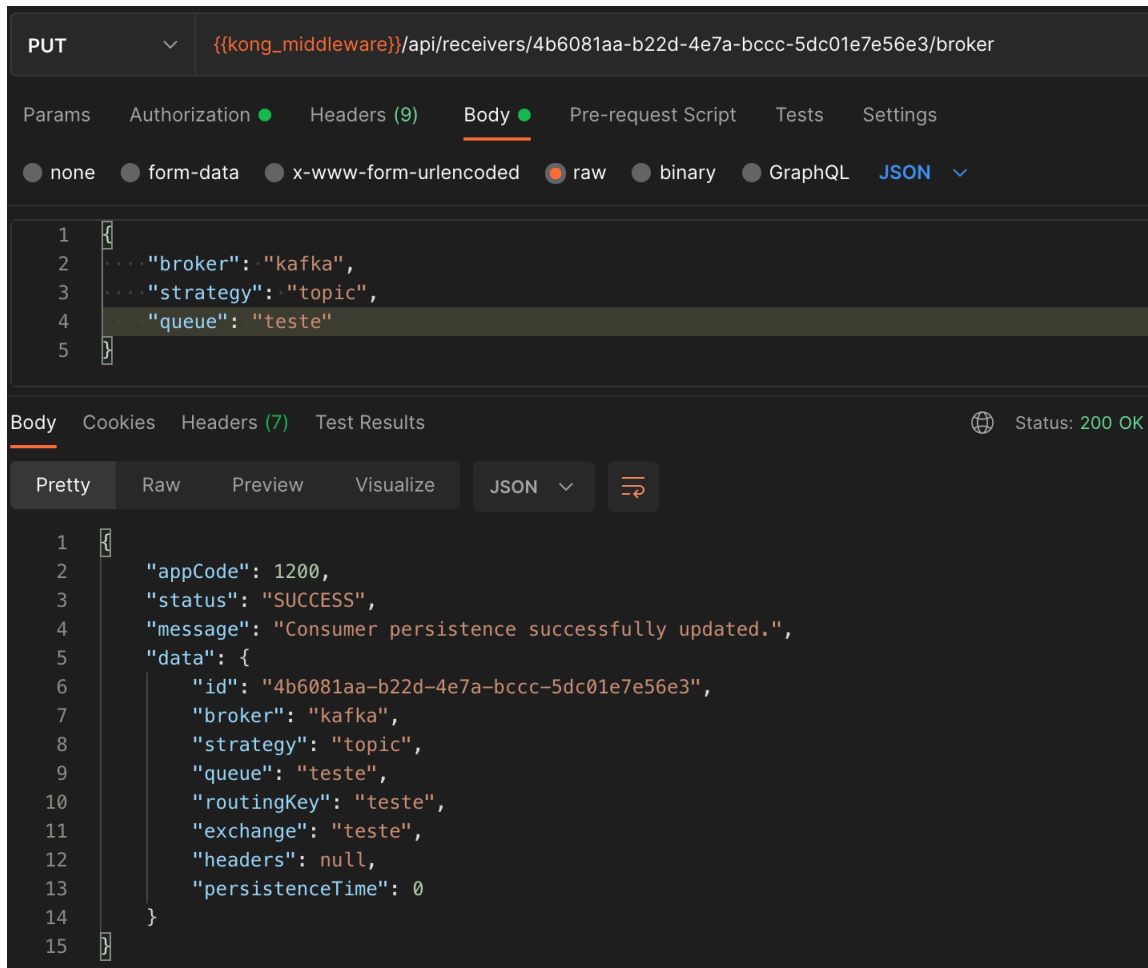


Figura 92: Alteração do Sistema de queues no consumer2.

Após estas alterações os resultados do envio e receção de mensagens no caso de estudo são os ilustrados na Figura 93. Nesta é possível observar que quando o producer1 envia mensagens para uma queue diferente da por omissão o consumer2 passa a receber essas mensagens.

Na Figura 93 é possível observar que o producer1 enviou 2 mensagens seguidas para a queue teste, porém o consumer2 apenas leu a segunda. Isto deveu-se ao facto de entre as duas mensagens enviadas ter sido efetuado o pedido de alteração de sistema de queues ao consumer2, pedido presente na Figura 92. Nesta figura o producer1 ainda enviou mais uma mensagem mas foi para a queue por omissão tendo sido capturada pelo consumer4.

```

Producer producer1 send message to queue teste
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.,"data":null}

Producer producer1 send message to queue teste
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.,"data":null}

Consumer consumer2 received: {"messageId":"c8b5415d-901a-47a5-a808-35adcbf08ece","data":"Hello from producer1 to

Producer producer1 send message to queue studycase
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.,"data":null}

Consumer consumer4 received: {"messageId":"e406147b-7b02-45f2-b64c-aa7d3defb432","data":"Hello from producer1!"}

```

Figura 93: Envio e leitura de mensagens após alteração do Sistema de queues do consumer2.

Foi ainda utilizado o Postman para mais um pedido de alteração de dados do consumer2, sendo que este pedido levou dados de um dos sistema de queues suportado, mas com a estratégia incompatível. Este pedido está visível na Figura 94, bem como a resposta obtida do Middleware desenvolvido.

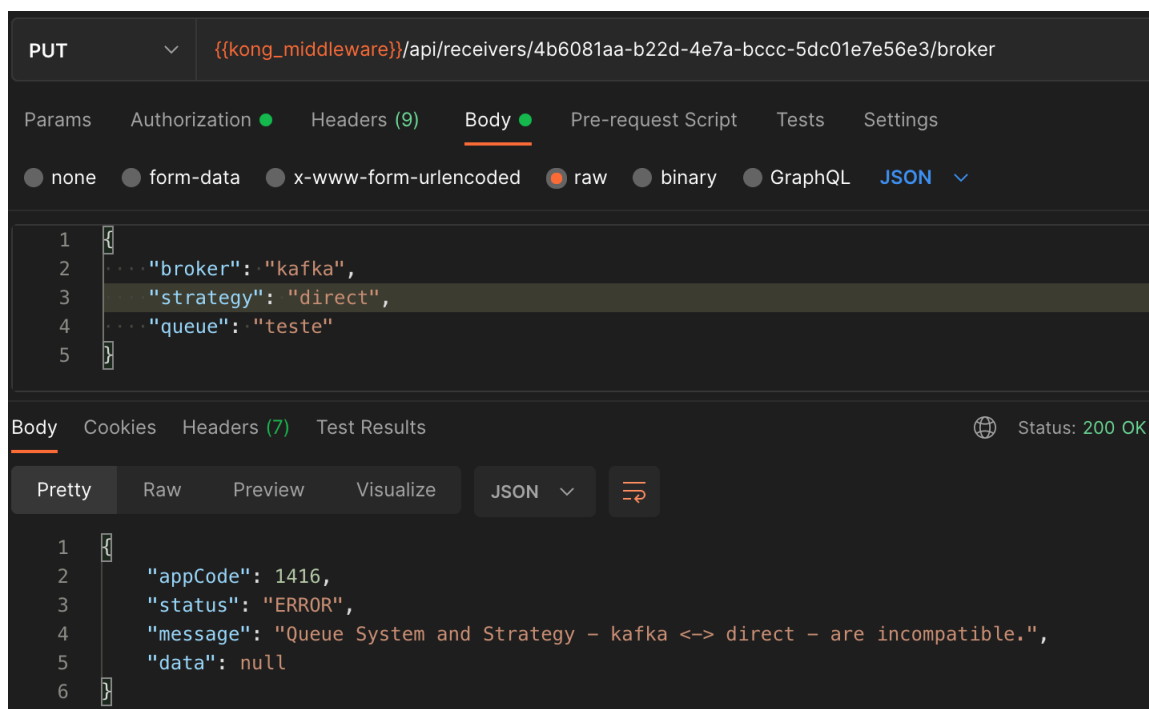


Figura 94: Alteração do Sistema de queues no consumer2 com estratégia incompatível.

Apesar de o pedido da Figura 94 ter falhado este consumidor continuou a funcionar e com a sua conexão SSE aberta, para demonstrar que este não foi abaixo foi realizado um novo pedido através do Postman, Figura 95, em que é colocado o consumer2 com a mesma configuração que o consumer4. Visto que estes estão a utilizar o Kafka com a estratégia topic, qualquer mensagem enviada pelo producer1 para a queue por omissão levará a que ambos recebam a mensagem.

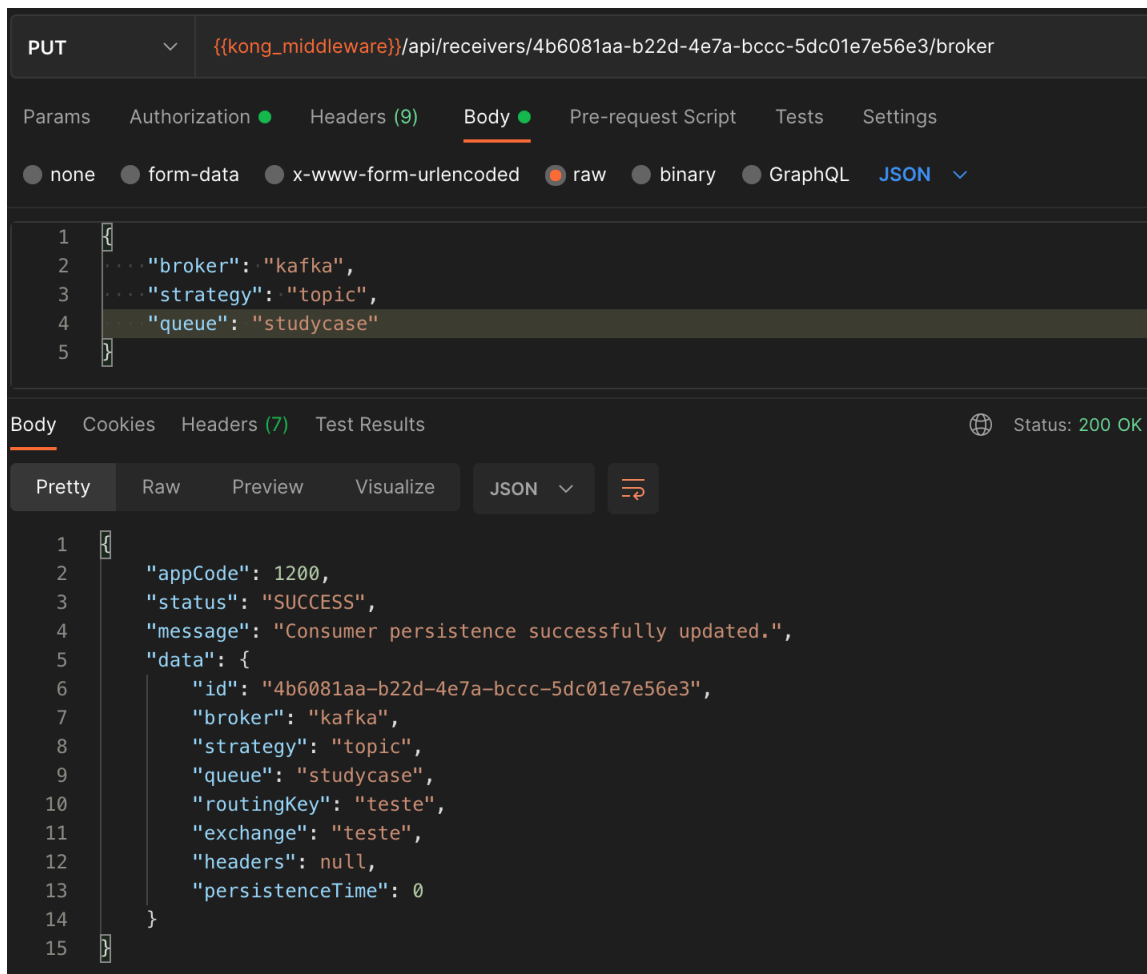


Figura 95: Alteração do Sistema de queues no consumer2 para configuração idêntica ao consumer4.

Ora após esta alteração é possível observar na Figura 96 o consumer2 e consumer4 a receber a mesma mensagem, sendo que isto apenas foi possível porque apesar da tentativa falhada de alteração de sistema de queues o Middleware desenvolvido manteve a conexão SSE do consumer2 aberta.

Na Figura 96 a primeira mensagem enviada pelo producer1 é apenas recebida pelo consumer4 porque o consumer2 ainda não foi alterado, entretanto foi realizado o pedido com o sistema de queues e estratégia incompatíveis e de seguida o pedido presente na Figura 95. Enquanto este último não era realizado o consumer2 ainda recebeu uma mensagem do producer1 quando este enviou a segunda mensagem da Figura 96, entretanto após a realização do pedido representado na Figura 95 o producer1 enviou uma mensagem que não foi recebida por nenhum consumidor. Isto deveu-se ao facto que o consumer2 já tinha sido alterado e não se encontrava à escuta naquela queue.

Porém na última mensagem enviada pelo producer1 é possível verificar que tanto o consumer2 como o consumer4 receberam a mesma mensagem, provando assim que o consumidor apesar do pedido falhado manteve a sua conexão aberta.

```
Producer producer1 send message to queue studycase
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.","data":null}

Consumer consumer4 received: {"messageId":"e91f71b2-f36a-42bf-8032-85eb414f2ec0","data":"Hello from producer1!"}

Producer producer1 send message to queue teste
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.","data":null}

Consumer consumer2 received: {"messageId":"c6cfec01-f4ec-441d-9ebd-d65dfb24bd8c","data":"Hello from producer1 to

Producer producer1 send message with strategy topic
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.","data":null}

Producer producer1 send message to queue studycase
Response: {"appCode":1200,"status":"SUCCESS","message":"Message successfully sent.","data":null}

Consumer consumer2 received: {"messageId":"791d3ab2-3166-4424-9cb0-5775488f6bc6","data":"Hello from producer1!"}
Consumer consumer4 received: {"messageId":"791d3ab2-3166-4424-9cb0-5775488f6bc6","data":"Hello from producer1!"}
```

Figura 96: Envio e leitura de mensagens após alteração do Sistema de queues do consumer2.

7 Teste de Desempenho

Por forma a verificar o desempenho do sistema desenvolvido foram realizados testes de desempenho. Estes testes servem para verificar a latência do Middleware desenvolvido bem como a perda de desempenho em relação à utilização direta de sistema de queues.

Para a realização do teste foi desenvolvido um serviço em Java. Este tanto realiza o teste utilizando o Middleware desenvolvido como utilizando diretamente um sistema de queues. Por forma a não repetir o mesmo teste para diferentes sistemas de queues optou-se por utilizar apenas o RabbitMQ para a realização dos mesmos, tanto para a comunicação direta com o sistema de queues como na utilização do Middleware desenvolvido.

O teste em que a comunicação é direta com o sistema de queues é denominada de teste 1, sendo a estrutura deste teste ilustrada na Figura 97. Este teste será realizado com duas máquinas uma vez que em contexto micro-serviços existem, com alta probabilidade, várias máquinas em funcionamento pelo que os vários componentes em comunicação podem não estar na mesma máquina, sendo assim representada a estrutura mais provável.

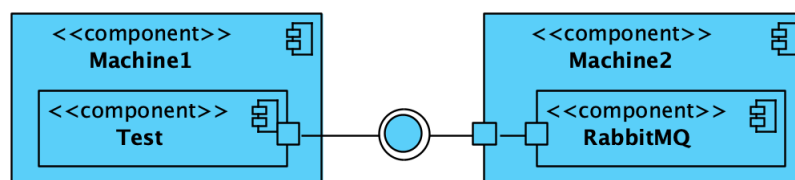


Figura 97: Arquitetura com Serviço de teste e Sistema de queues em máquinas distintas máquina.

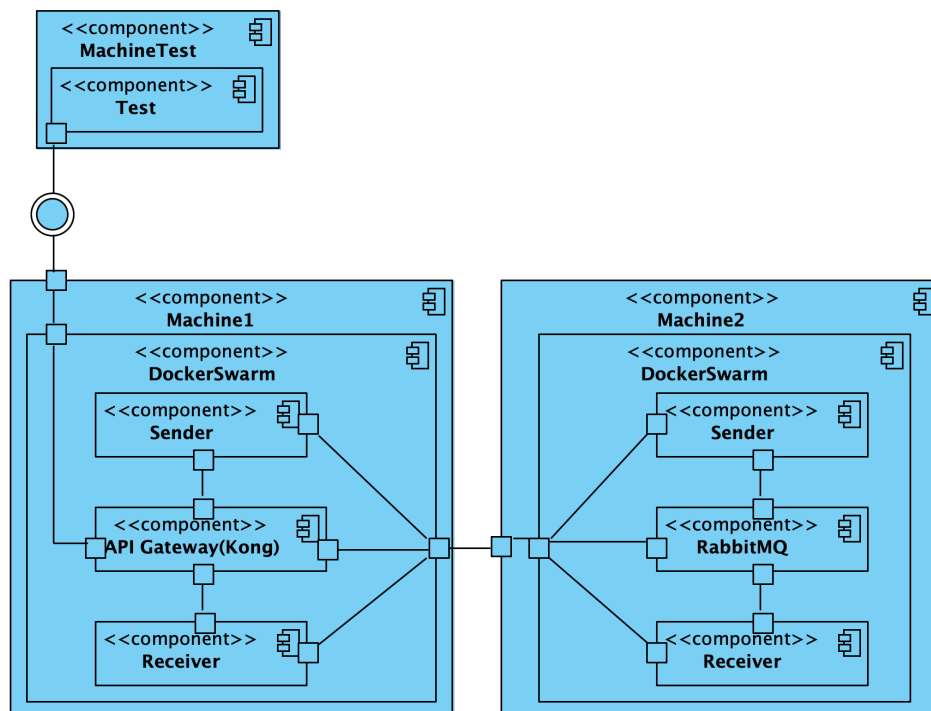


Figura 98: Arquitetura com Serviço de teste e Middleware desenvolvido.

No teste com o Middleware desenvolvido a arquitetura utilizada é a ilustrada na Figura 98, sendo que este é o caso de teste número 2. Apesar de na Figura 98 o RabbitMQ aparecer na Machine2 isto não é garantido podendo o mesmo estar na Machine1. Isto depende das decisões efetuadas pelo Docker Swarm por forma a dividir os microsserviços pelos recursos disponíveis, o mesmo acontece com o APIGateway (Kong). Os produtores e consumidores podem estar distribuídos pelas duas máquinas uma vez que o componente Sender e Receiver desenvolvidos estão replicados e por isso presentes nas duas máquinas.

A Figura 98 encontra-se simplificada não mostrando os protocolos de comunicação entre os vários componentes nem a direção em que estes ocorrem, porém estes são iguais aos presentes na Secção 4.3.2. Para uma maior simplicidade o componente Discovery bem como as diversas BDs não foram aqui demonstradas.

Os testes consistem na criação de 3 produtores que publicam todos para a mesma queue e 3 consumidores que lêem todos dessa mesma queue.

Cada produtor irá produzir 50.000 mensagens dando assim um total de 150.000 de mensagens enviadas, mas como o arranque costuma ser mais lento e o final também já não possui tanto trabalho serão apenas contabilizadas as mensagens a partir da 10.000 até à 140.000 fazendo assim um total de 130.000 mensagens.

O tempo de envio de cada mensagem é calculada através do tempo em milissegundos a partir do momento em que é criada e enviada até ser recebida. Para realizar este cálculo é feita a leitura do tempo em milissegundos antes da mensagem ser enviada e este valor é colocado como os dados da mensagem, quando a mensagem é recebida é feita uma nova leitura do tempo em milissegundos, subtraindo o tempo dos dados da mensagem a este último tempo obtido. Este cálculo representa o tempo de envio da mensagem.

O cálculo do tempo médio média ($\overline{T}$) é representado pela formula presente na Figura 99, em que n representa o número da mensagem e T_n representa o tempo de envio da mensagem n . O valor de n varia entre 100.000 e 9.900.000 porque são os valores anteriormente descritos como sendo o intervalo de mensagens que são contabilizadas para o cálculo do tempo de envio.

$$\overline{T} = \frac{\sum_{n=10000}^{140000} T_n}{130000}$$

Figura 99: Cálculo do Tempo Médio de envio ($\overline{T}$).

Além do cálculo do tempo médio de envio ($\overline{T}$) vai ser calculada a perda de desempenho (Pd) do teste em que se utilizou o sistema desenvolvido em relação ao caso de comunicação direta com o sistema de queues. Este cálculo é dado pela formula apresentada na Figura 100, em que $\overline{T}_{middleware}$ é o tempo médio na utilização do Middleware desenvolvido e o $\overline{T}_{queues}$ é o tempo médio na comunicação direta com o sistema de queues.

$$Pd = \frac{\overline{T}_{middleware}}{\overline{T}_{queues}}$$

Figura 100: Cálculo da Perda de Desempenho (Pd).

Por forma a obter um resultado mais preciso para o cálculo do tempo médio ($\bar{T}$) e posteriormente da perda de desempenho (Pd) cada caso de teste será realizado 10 vezes, sendo depois excluído o pior e o melhor resultado e por fim utilizada a média aritmética dos restantes resultados para o cálculo da perda de desempenho (Pd). Ora a média aritmética é calculado através a fórmula ilustrada na Figura 101. O pior e o melhor teste foram removidos uma vez que estes podiam ser *outliers* para a média.

$$\bar{T}_{med} = \frac{(\sum_{i=1}^{10} \bar{T}_i) - \bar{T}_{min} - \bar{T}_{max}}{8}$$

Figura 101: Cálculo da Média dos Tempos Médios ($\bar{T}_{med}$).

Uma vez que a linguagem de programação utilizada foi Java para garantir que o valor de n e de $\sum_{n=10000}^{140000} T_n$ não tinha problemas de escritas concorrentes que levaria a perda de valores e por sua vez incorreções nos valores registados, as variáveis responsáveis por estes valores são do tipo AtomicLong, garantindo assim que tentativas de modificações concorrentes sejam tratadas atomicamente, pelo que nunca existem duas ou mais Threads a modificar estes valores em simultâneo.

O AtomicLong possui ainda o método incrementAndGet que permite incrementar e obter o valor incrementado garantido que desde que foi incrementado mais nenhuma Thread modificou o valor antes de ser devolvido. Este método é utilizado para o parâmetro n por forma a ser possível verificar quando os tempos das mensagens passam a ser contabilizados.

Durante a realização dos testes o serviço de teste esteve em funcionamento numa máquina de núcleo duplo com 8GB de memória RAM, sendo que na comunicação direta com o sistema de queues foi criada uma outra máquina com os mesmos requisitos de forma a alojar o sistema de queues.

No teste com o Middleware desenvolvido este esteve em funcionamento num Docker Swarm com duas máquinas de núcleo duplo e 8GB de memória RAM cada uma.

Todas máquinas utilizadas durante a realização dos testes encontram-se na zona "europa-west2-c" do GCP. Esta escolha foi realizada com base no funcionamento do sistema desenvolvido, pois o deployment do sistema vai ser realizado o mais próximo possível da região pretendida de forma a reduzir a latência do serviço. A zona escolhida podia ter sido outra desde que todas as máquinas ficassem nessa mesma zona.

Na Tabela 21 são ilustrados os 10 resultados obtidos para cada um dos casos do teste realizado, são ainda ilustradas as médias dos tempos ($\bar{T}_{med}$) que serão utilizadas para o cálculo da perda de desempenho. As células coloridas a vermelho representam o melhor e o pior resultado de cada teste. Estas foram descartadas para o cálculo da Média dos Tempos.

Tempo médio	Teste 1	Teste 2
$\bar{T}_1$	1 ms	31 ms
$\bar{T}_2$	1 ms	32 ms
$\bar{T}_3$	1 ms	30 ms
$\bar{T}_4$	1 ms	32 ms
$\bar{T}_5$	1 ms	32 ms
$\bar{T}_6$	1 ms	32 ms
$\bar{T}_7$	1 ms	33 ms
$\bar{T}_8$	1 ms	31 ms
$\bar{T}_9$	1 ms	32 ms
$\bar{T}_{10}$	1 ms	34 ms
$\bar{T}_{med}$	1 ms	32 ms

Tabela 21: Resultados dos testes realizados.

Após os resultados obtidos na Tabela 21 é possível então calcular a perda de desempenho, sendo esta de 32 vezes.

A perda de desempenho registada pode dever-se em parte da distribuição dos produtores e consumidores pelas duas máquinas do Docker Swarm e também da localização do RabbitMQ dentro do Docker Swarm. Pois toda a comunicação é realizada à Machine1 do Docker Swarm, pelo que se o APIGateway se encontrar na Machine2 terá de ser realizado um pedido entre as duas máquinas para ser entregue a este.

De seguida o APIGateway terá de decidir a qual das réplicas de Sender entregar o pedido que recebeu, podendo a levar a mais um pedido entre máquinas. No momento da entrega de uma mensagem por parte do Sender ao RabbitMQ pode novamente ser realizado mais um pedido entre máquinas no caso destes se encontrarem em máquinas diferentes, podendo depois ocorrer novamente um pedido entre máquinas na entrega da mensagem ao Receiver e por último do Receiver ao APIGateway para a entrega da mensagem ao consumidor.

Além da perda de desempenho que em parte se deve ao problema anteriormente descrito, parte desta perda é também devido à lógica de negócio introduzida, em que existe manipulação de dados e verificação de condições para uma mensagem ser aceite. Estas validações são também aplicadas ao nível do produtor por forma a tentar garantir que este não se faz passar por outro e toda esta lógica leva a uma deterioração do desempenho.

Um outro ponto que pode levar à degradação observada nos resultados são as tecnologias utilizadas para o desenvolvimento e deployment do Middleware.

Apesar de o teste ter sido realizado apenas com o RabbitMQ, é expectável que para os restantes sistemas de queues os valores dos testes de desempenho se mantenham, uma vez que na Secção 2.3 estes sistemas obtinham

sensivelmente os mesmos valores nos testes realizados.

Com os resultados obtidos pode concluir-se que com a arquitetura concebida e a lógica adicionada, por forma a acrescentar funcionalidades de valor acrescentado, degradam o desempenho do Middleware em relação aos sistemas de queues, sendo os resultados deste em parte justificados pelos pontos acima descritos.

8 Conclusão

Com o crescente número de sistemas de queues no mercado e as dependências inseridas nas aplicações que utilizam estes fazem com que muitas destas aplicações tenham de ser bastante reestruturadas quando necessitam de trocar de um sistema de queues para outro.

O Middleware desenvolvido durante esta dissertação alcançou os objetivos traçados, fazendo a integração de 3 sistemas de queues (RabbitMQ, ActiveMQ5 e Kafka) e permitindo assim às aplicações ficarem apenas dependentes do Middleware desenvolvido em vez dos sistemas de queues. Com este as aplicações podem trocar de um sistema de queues para outro sem a necessidade de serem reestruturadas e podem ainda usufruir de algumas características de valor acrescentado que foram adicionadas ao mesmo.

Este Middleware respeita os requisitos identificados e com o estudo prévio sobre os sistemas de queues já existentes foi possível tirar o máximo partido dos mesmos. Este estudo foi fulcral para o desenho da arquitetura do Middleware, pois sem este estudo prévio muitos problemas seriam apenas detetados na fase de desenvolvimento e teste.

Após a conclusão deste Middleware foi realizado um caso de estudo que permitiu demonstrar as funcionalidades do Middleware desenvolvido e além deste foi desenvolvido um caso de teste que permitiu verificar e calcular a degradação do desempenho na utilização dos sistemas de queues através deste.

Com o resultado do teste de desempenho foi possível concluir que a arquitetura utilizada e a lógica introduzida degradaram bastante o tempo que uma mensagem demora a ser entregue ao consumidor após ter sido criada. Porém existem características de valor acrescentado como a troca de um sistema de queues para outro sem ser necessário reiniciar a aplicação e a possibilidade de um consumidor persistir as suas mensagens do lado do Middleware para as reler quando necessitar.

Apesar da degradação do tempo que uma mensagem demora a ser entregue a um consumidor as características de valor acrescentado são uma mais valia para algumas aplicações, sendo que isto irá depender da área de negócio de cada uma das aplicações que necessite de comunicar de forma assíncrona.

Um ponto a melhorar no Middleware desenvolvido é a divisão dos microsserviços, uma vez que o produtor e consumidor tem bastantes dados em comum e como uma aplicação pode ser produtora e consumidora ao simultaneamente faria sentido existir um componente extra para gerir os dados destes, deixando assim o microsserviço Sender apenas com a responsabilidade de enviar mensagens e o microsserviço Receiver com a responsabilidade de receber as mesmas.

8.1 Trabalho Futuro

Tal como em todos os projetos e aplicações existem sempre melhorias a efetuar e funcionalidades a acrescentar. Esta dissertação não é exceção, pelo que para trabalho futuro ficam os seguintes pontos:

- Desenvolver o deployment automático das instâncias para os novos projetos e também para as atualiza-

ções feitas aos projetos já existentes.

- Implementar os requisitos não prioritários que não foram implementados, como o agendamento de mensagens.
- Substituir a comunicação HTTP por comunicação HTTPS.
- Implementar Access Tokens com Refresh Tokens.
- Adicionar novos sistemas de queues, principalmente os que oferecerem maior relevância para as necessidades da comunidade.
- Melhorar a escalabilidade do sistema, como por exemplo passando de Docker Swarm para Kubernetes.
- Adicionar a possibilidade de utilização dos sistemas de queues em cluster ou federation, sendo no momento da criação do projeto especificado a quantidade de sistemas de queues e qual a arquitetura de comunicação entre eles.
- Tratar das mensagens descartadas pelo RabbitMQ, Secção 5.3.1.
- Adicionar painel de administração onde seja possível gerir todos os sistemas de queues, produtores e consumidores para uma instância.

Bibliografia

- [Act] ActiveMQ. *ActiveMQ Artemis STOMP*. <https://activemq.apache.org/components/artemis/documentation/latest/stomp.html>. Acedido: 2020-12-13.
- [Act19a] ActiveMQ. *ActiveMQ - Delay and Schedule Message Delivery*. <https://activemq.apache.org/delay-and-schedule-message-delivery>. Acedido: 2020-12-23. 2019.
- [Act19b] ActiveMQ. *ActiveMQ AMQP*. <https://activemq.apache.org/amqp>. Acedido: 2020-12-13. 2019.
- [Act19c] ActiveMQ. *ActiveMQ STOMP*. <https://activemq.apache.org/stomp>. Acedido: 2020-12-13. 2019.
- [Ahu21] Madhur Ahuja. *Architecture and Monitoring Apache ActiveMQ with Grafana*. <https://www.metricfire.com/blog/architecture-and-monitoring-apache-activemq-with-grafana/>. Acedido: 2021-04-08. Mar. de 2021.
- [Akg13] Faruk Akgul. *ZeroMQ*. Birmingham, UK: Packt Publishing, 2013. ISBN: 978-1-78216-104-2.
- [Alv+18] Oscar Alvear et al. «Crowdsensing in Smart Cities: Overview, Platforms, and Environment Sensing Issues». Em: *Sensors* 18 (fev. de 2018), p. 460. DOI: [10.3390/s18020460](https://doi.org/10.3390/s18020460).
- [Bez10] Yuri Morais Bezerra. «Multi-MOM : um middleware multi-paradigma, extensível e orientado a mensagens para computação móvel». Tese de mestrado. Brasil: Departamento de Informática, Universidade Federal da Paraíba, 2010.
- [CMS16] Stipe Celar, Eugen Mudnic e Zeljko Seremet. «State-Of-The-Art of Messaging for Distributed Computing Systems». Em: *2016 DAAAM 27th International Symposium on Intelligent Manufacturing and Automation*. 2016, pp. 298–307. ISBN: 9783902734082. DOI: [10.2507 / 27th.daaam.proceedings.044](https://doi.org/10.2507/27th.daaam.proceedings.044).
- [DE17] Philippe Dobbelaere e Kyumars Sheykh Esmaili. «Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper». Em: DEBS '17. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 227–238. ISBN: 9781450350655. DOI: [10.1145/3093742.3093908](https://doi.org/10.1145/3093742.3093908).
- [Dea13] Nigel Deakin. *Java Message Service*. USA: Oracle, 2013. ISBN: 978-0-596-52204-9.
- [Dev17] Journal Dev. *JMS API Overview: JMS 1.x and JMS 2.x*. <https://www.journaldev.com/9776/jms1-and-jms2-api-overview>. Acedido: 2020-11-08. 2017.

- [Dos14] David Dossot. *RabbitMQ Essentials*. Birmingham, UK: Packt Publishing, 2014. ISBN: 978-1-78398-320-9.
- [EA15] N. Estrada e H. Astudillo. «Comparing scalability of message queue system: ZeroMQ vs RabbitMQ». Em: *2015 Latin American Computing Conference (CLEI)*. 2015, pp. 1–6. DOI: [10 . 1109 / CLEI . 2015 . 7360036](https://doi.org/10.1109/CLEI.2015.7360036).
- [EB15] S. El Mimouni e M. Bouhdadi. «Formal modeling of the Simple Text Oriented Messaging Protocol using Event-B method». Em: *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*. 2015, pp. 1–4. DOI: [10 . 1109 / AICCSA . 2015 . 7507170](https://doi.org/10.1109/AICCSA.2015.7507170).
- [Fer] Tiago Fernandes. *What is Dependency Injection?* <https://www.growin.com/blog/what-is-dependency-injection/>. Acedido: 2021-04-20.
- [Gar15] Nishant Garg. *Learning Apache Kafka, Second Edition*. Birmingham, UK: Packt Publishing, 2015. ISBN: 978-1-78439-309-0.
- [Gura] Refactoring Guru. *Adapter*. <https://refactoring.guru/design-patterns/adapter>. Acedido: 2021-02-12.
- [Gurb] Refactoring Guru. *Facade*. <https://refactoring.guru/design-patterns/facade>. Acedido: 2021-02-12.
- [Gurc] Refactoring Guru. *Singleton*. <https://refactoring.guru/design-patterns/singleton>. Acedido: 2021-02-12.
- [Hin] Pieter Hintjens. *OMQ - The Guide*. <https://zguide.zeromq.org/docs/chapter2>. Acedido: 2021-01-22.
- [Ion15] V. M. Ionescu. «The analysis of the performance of RabbitMQ and ActiveMQ». Em: *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*. 2015, pp. 132–137. DOI: [10 . 1109 / RoEduNet . 2015 . 7311982](https://doi.org/10.1109/RoEduNet.2015.7311982).
- [Kle+15] A. F. Klein et al. «An experimental comparison of ActiveMQ and OpenMQ brokers in asynchronous cloud environment». Em: *2015 Fifth International Conference on Digital Information Processing and Communications (ICDIPC)*. 2015, pp. 24–30. DOI: [10 . 1109 / ICDIPC . 2015 . 7323001](https://doi.org/10.1109/ICDIPC.2015.7323001).

- [Kon] Tugce Konuklar. *Hexagonal (Ports & Adapters) Architecture*. <https://medium.com/idealo-tech-blog/hexagonal-ports-adapters-architecture-e3617bcf00a0>. Acedido: 2021-04-20.
- [KS16] R. K. Kodali e S. Soratkal. «MQTT based home automation system using ESP8266». Em: *2016 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*. 2016, pp. 1–5. DOI: [10.1109/R10-HTC.2016.7906845](https://doi.org/10.1109/R10-HTC.2016.7906845).
- [LLZ12] X. Lu, W. Lei e W. Zhang. «The Design and Implementation of XMPP-Based SMS Gateway». Em: *2012 Fourth International Conference on Computational Intelligence, Communication Systems and Networks*. 2012, pp. 145–148. DOI: [10.1109/CICSyN.2012.35](https://doi.org/10.1109/CICSyN.2012.35).
- [MG17] Apurva Mehta e Jason Gustafson. *Confluent*. <https://www.confluent.io/blog/transactions-apache-kafka/>. Acedido: 2020-12-20. Nov. de 2017.
- [Nai17] N. Naik. «Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP». Em: *2017 IEEE International Systems Engineering Symposium (ISSE)*. 2017, pp. 1–7. DOI: [10.1109/SysEng.2017.8088251](https://doi.org/10.1109/SysEng.2017.8088251).
- [Rab20] RabbitMQ. *RabbitMQ Protocols*. <https://www.rabbitmq.com/protocols.html>. Acedido: 2020-12-23. 2020.
- [RMC09] Mark Richards, Richard Monson-Haefel e David A. Chappel. *Java Message Service*. USA: O'Reilly Media, Incorporated, 2009. ISBN: 978-0-596-52204-9.
- [SBD11] Bruce Snyder, Dejan Bosanac e Rob Davies. *ActiveMQ in Action*. USA: Manning Publications Co., 2011. ISBN: 978-1-933988-94-8.
- [Sch+14] D. Schuster et al. «Global-Scale Federated Access to Smart Objects Using XMPP». Em: *2014 IEEE International Conference on Internet of Things (iThings), and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom)*. 2014, pp. 185–192. DOI: [10.1109/iThings.2014.35](https://doi.org/10.1109/iThings.2014.35).
- [See13] Raphael Thomas Seebacher. «Messaging Challenges in a Globally Distributed Network». Tese de mestrado. Zürich, Deutschland: TIK, ETH Zürich, 2013, p. 14.
- [SM17] Dipa Soni e Ashwin Makwana. «A SURVEY ON MQTT: A PROTOCOL OF INTERNET OF THINGS (IOT)». Em: abr. de 2017.
- [STO12] STOMP. *STOMP: The Simple Text Orientated Messaging Protocol v1.2*. <http://stomp.github.io/stomp-specification-1.2.html>. Acedido: 2020-11-12. 2012.

- [Wal19] Felix Walcher. «KNX to MQTT/AQMP». Tese de mestrado. Wien, Österreich: Fakultät für Informatik, Technischen Universität Wien, 2019, pp. 7–15.

A Diagramas de Classe

A.1 Registration

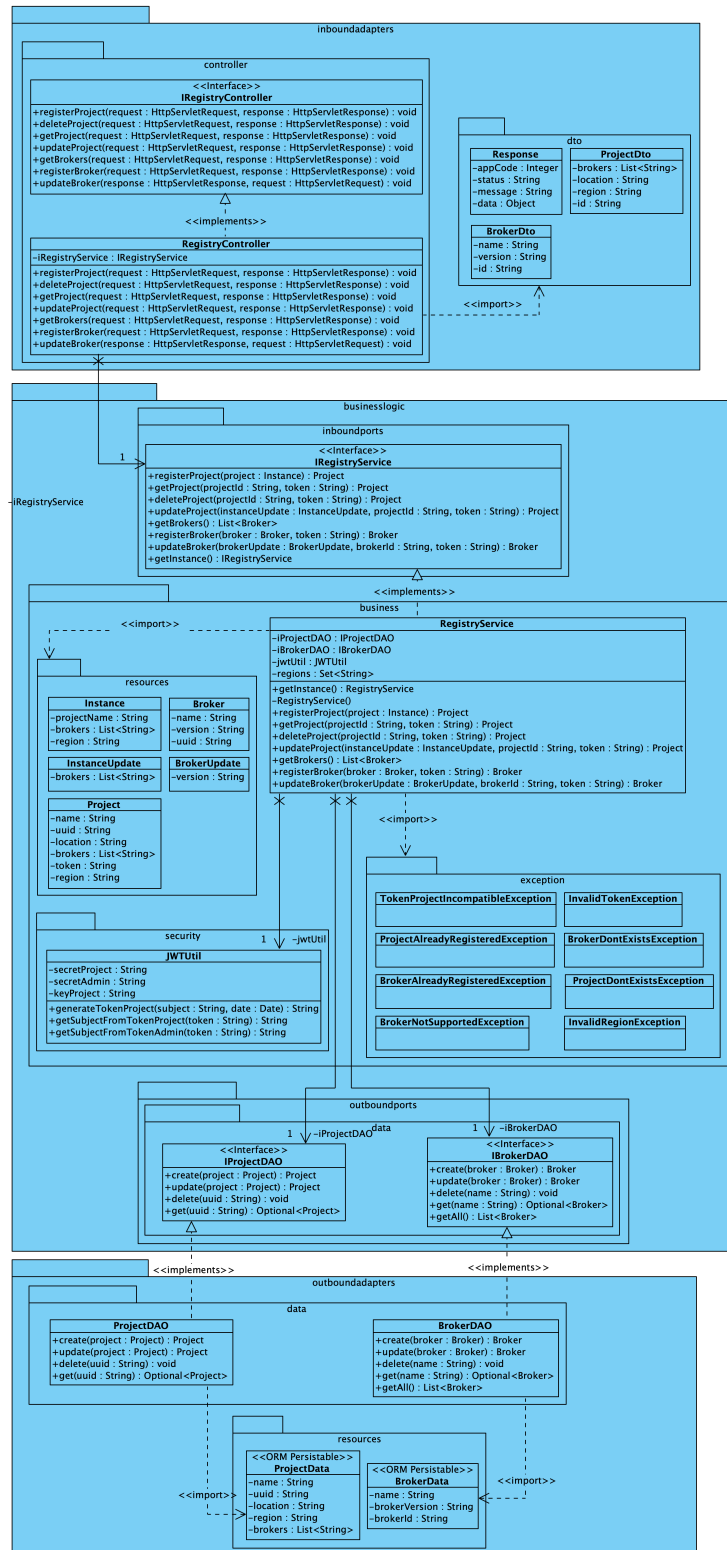


Figura 102: Diagrama de Classes do serviço Registration.

A.2 Sender

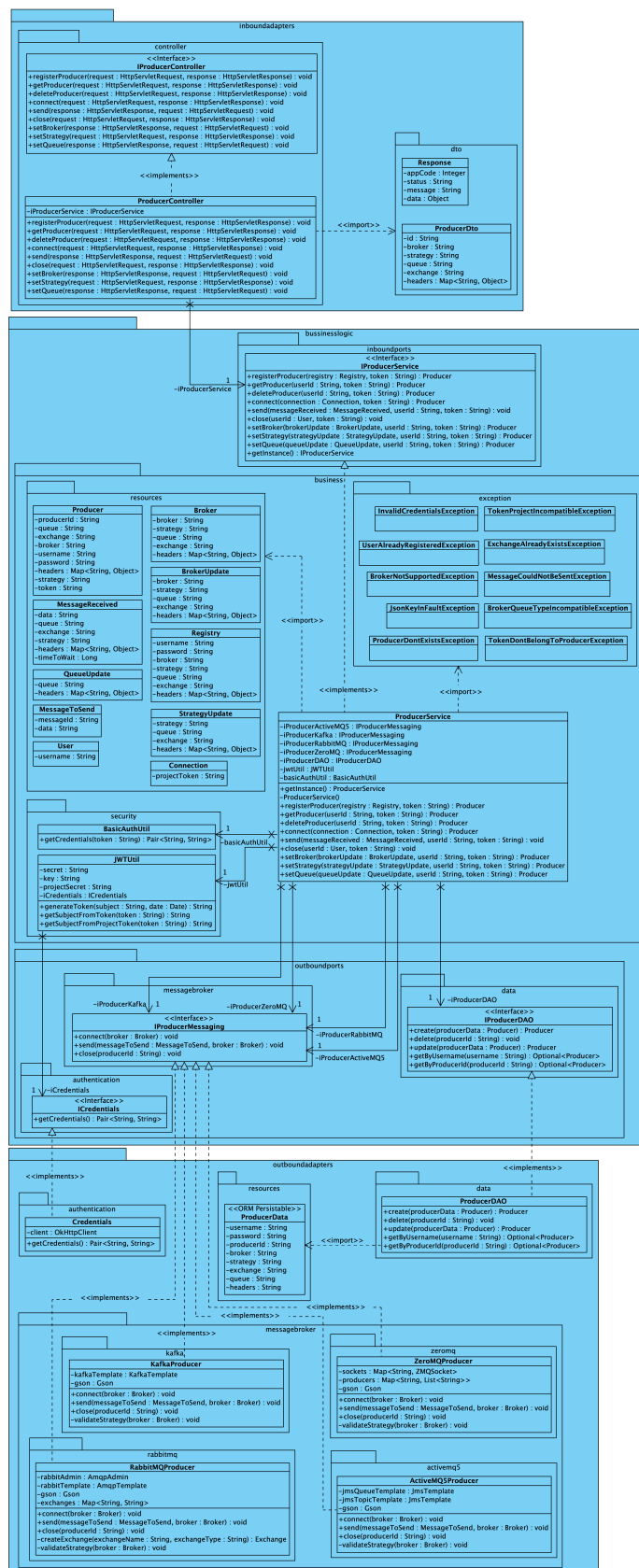


Figura 103: Diagrama de Classes do serviço Sender.

A.3 Receiver

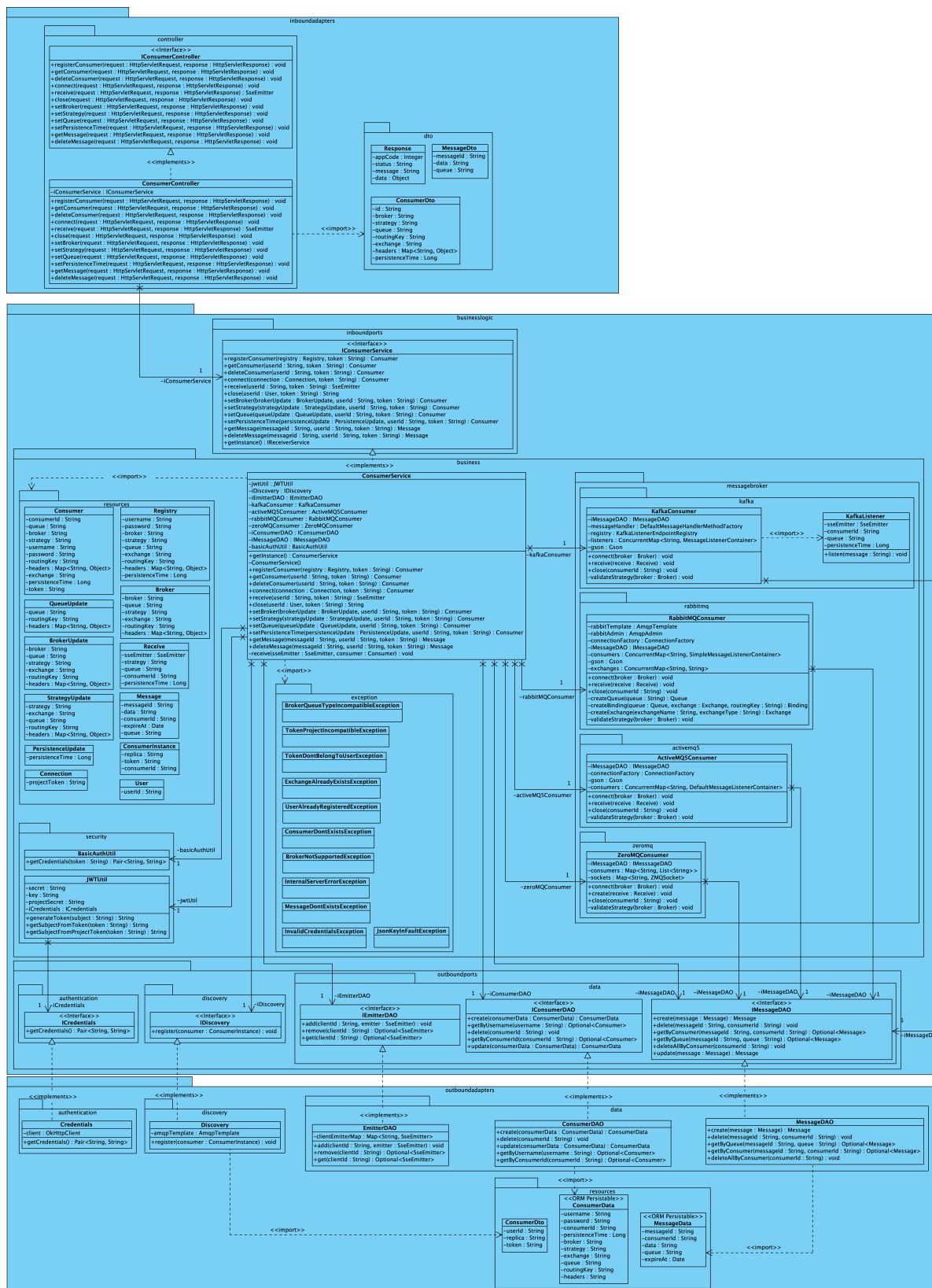


Figura 104: Diagrama de Classes do serviço Receiver.

A.4 Discovery

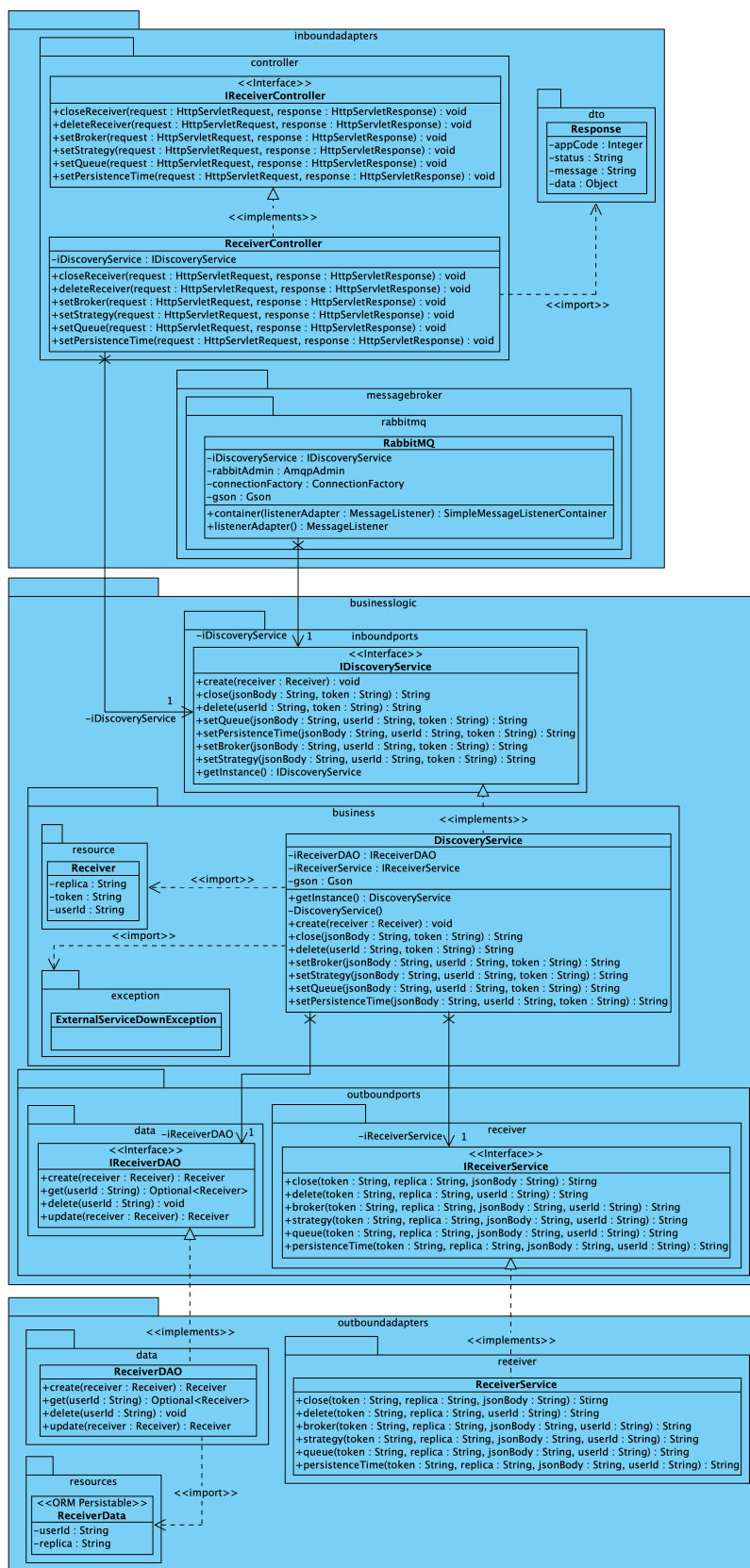


Figura 105: Diagrama de Classes do serviço Discovery.

B Especificações API

B.1 Registration

B.1.1 Rotas

De seguida serão ilustrados todos os campos necessários para todas as rotas deste sistema, bem como as restrições para cada um dos atributos sempre que se aplique. Caso estas restrições não sejam cumpridas nos atributos cuja localização é Path ou Header será lançada uma exceção pelo APIGateway (Kong).

Tal como se pode observar na Tabela 22 todos os atributos são obrigatórios e estão presentes no body do pedido HTTP.

O projectName será o nome a atribuir ao projeto e este terá de ter um comprimento mínimo de 8 e máximo de 32 caracteres, sendo estes alfanuméricos.

Brokers apenas poderá conter os sistemas de queues suportados pelo Middleware, atualmente estes são o rabbitmq, kafka e o activemq5, sendo que este atributo possui a mesma restrição na Tabela 24.

O atributo region tem de estar em conformidade com o ISO 3166-1⁶⁹ código Alpha-2, sendo os 2 caracteres constituintes 2 letras maiúsculas.

Todos estes atributos são case-sensitive.

Variável	Tipo de dados	Obrigatório	Localização
projectName	string	sim	Body
brokers	string[]	sim	Body
region	string	sim	Body

Tabela 22: Campos da rota número 1.

O atributo projectId presente na Tabela 23 é gerado na rota 1 da Tabela 17. Este encontra-se dentro do campo data da resposta e é denominado de id, ver Tabela 27. O projectId tem o formato UUID. Este segue a mesma restrição de formato na Tabela 24.

O Authorization presente nas Tabelas 23 e 24 é o JWT gerado na rota 1 no momento da criação do projeto. Este serve para autenticar o pedido e validar o utilizador verificando se este token dá acesso ao recurso pretendido. Este JWT e os presentes nas Tabelas 25 e 26 utilizam o algoritmo HS256 pertencente à família SHA-2, que utiliza uma chave secreta de 256 bits para encriptar o token.

A Tabela 23 representa simultaneamente os atributos para a rota número 2 e 3 uma vez que estes são exatamente iguais.

⁶⁹ISO 3166-1: <https://www.iso.org/standard/72482.html>

Variável	Tipo de dados	Obrigatório	Localização
projectId	string	sim	Path
Authorization	Bearer token	sim	Header

Tabela 23: Campos das rotas número 2 e 3.

Variável	Tipo de dados	Obrigatório	Localização
projectId	string	sim	Path
brokers	string[]	sim	Body
Authorization	Bearer token	sim	Header

Tabela 24: Campos da rota número 4.

Não é apresentada descrição para a rota 5 uma vez que esta não possui nenhum atributo no seu pedido HTTP.

Nas rotas administrativas do serviço o atributo broker, na Tabela 25, é formado apenas por caracteres alfa-numéricos, de tamanho mínimo 4 e máximo de 32, sendo que os caracteres alfabéticos presentes são apenas as letras minúsculas.

O atributo version é composto por três inteiros separados por ponto, por exemplo, 5.3.9. Na Tabela 26 este atributo segue a mesma restrição.

O atributo Authorization presente na Tabela 25 e 26 é um JWT com uma chave de encriptação diferente da chave utilizada para os JWTs de gestão dos projetos e este contém ainda uma chave única incorporada nos seus dados.

Variável	Tipo de dados	Obrigatório	Localização
broker	string	sim	Body
version	string	sim	Body
Authorization	Bearer token	sim	Header

Tabela 25: Campos da rota número 6.

O brokerId presente na Tabela 26 é um atributo gerado na rota 6 presente na Tabela 17, sendo este retornado no campo data da resposta e é denominado de id, ver Tabela 27.

Variável	Tipo de dados	Obrigatório	Localização
brokerId	string	sim	Path
version	string	sim	Body
Authorization	Bearer token	sim	Header

Tabela 26: Campos da rota número 7.

B.1.2 Respostas

Cada uma das rotas apresentadas na Tabela 17 possui uma resposta de sucesso e estas estão representadas na Tabela 27.

As rotas número de 1 a 4 tal como representado na Tabela 27 possuem todas os mesmos dados no campo data da resposta, sendo que estes foram assim colocados para ir de encontro às boas práticas de uma API RESTful, permitindo assim no momento de registo, atualização e eliminação ter acesso ao novos dados ou ao dados eliminados.

O mesmo acontece para as rotas 6 e 7 em que o resultado é igual de forma a observar quais foram os dados que ficaram registados.

# Rota	Mensagem	Data
1	Project successfully registered.	id: "ade34209-1abc-9f21-0baf-73ba942f983e" location: "<projectName>.multibroker.com" region: "pt" brokers: ["brokera","brokerb"]
2	Project info successfully retrieved.	Igual à rota 1.
3	Project successfully deleted.	Igual à rota 1.
4	Project successfully updated.	Igual à rota 1.
5	Brokers names.	[[name: "brokera", version: "3.5.7", id: "1abc-9f21"], {name: "brokerb", version: "5.7.9", id: "9f21-0baf"}]
6	Broker registered.	id: "ade34209-1abc-9f21-0baf-73ba942f983e" name: "brokera" version: "5.1"
7	Broker updated.	Igual à rota 6.

Tabela 27: Respostas aos pontos de entrada do serviço Registration.

B.1.3 Erros

Além das respostas apresentadas anteriormente existem respostas de erro para problemas que possam ocorrer durante a utilização do serviço e estas estão todas representadas na Tabela 28. O campo Código da Tabela 51 representa o *appCode* e a Mensagem o campo *message* da resposta personalizada. O campo *data* é sempre null e o campo *status* é sempre ERROR, pelo que estes dois campos não são apresentados na Tabela 51.

O erro BadRequest é lançado quando alguma das restrições apresentadas para cada uma das rotas não é cumprida. Na Tabela 28 as reticências presentes na descrição deste erro representam a localização do erro.

Os erros TokenProjectIncompatible e InvalidToken ocorrem quando existem problemas com os tokens, como por exemplo o token conter informação sobre outro projeto ou o token não conter informação necessária para o método que invocou, respetivamente.

BrokerNotSupported ocorre quando um ou mais sistemas de queues especificados não são suportados pelo sistema.

Quando um recurso não é encontrado pode ser invocado um dos seguintes erros, ProjectNotFound ou BrokerNotFound, dependendo se o recurso que se pretendia aceder era um projeto ou um sistema de queues, respetivamente.

Tal como os nomes indicam ProjectConflict e BrokerConflict são lançados quando se tenta registar um recurso que já existe, sendo que neste caso um projeto ou um sistema de queues, respetivamente.

O último erro da Tabela 28 é o InvalidRegion e ocorre quando a região especificada não existe.

Nome	Código	Mensagem
BadRequest	1400	Values with errors: ...
InvalidToken	1401	Token is invalid.
BrokerNotSupported	1402	Broker(s) isn't(aren't) supported - BrokerA, BrokerB.
TokenProjectIncompatible	1403	Project <projectUuid> don't match given token.
ProjectNotFound	1404	Project <projectUuid> doesn't exist.
BrokerNotFound	1406	Broker <brokerUuid> doesn't exist.
ProjectConflict	1409	Project <projectUuid> already registered.
BrokerConflict	1410	Broker <brokerUuid> already registered.
InvalidRegion	1411	Invalid region.

Tabela 28: Códigos de erro do serviço Registration.

Na Tabela 29 é realizada uma correspondência entre as rotas do serviço e os erros que podem ser lançados por cada rota. Esta tabela tem como objetivo facilitar aos utilizadores deste serviço saberem quais são as

possibilidades de erros em cada uma das rotas.

A rota 5 é uma exceção não possuindo nenhuma exceção personalizada.

#Rota	Erros
1	BadRequest, BrokerNotSupported, ProjectConflict e InvalidRegion
2	BadRequest, TokenProjectIncompatible e ProjectNotFound
3	BadRequest, TokenProjectIncompatible e ProjectNotFound
4	BadRequest, TokenProjectIncompatible, BrokerNotSupported e ProjectNotFound
5	—
6	BadRequest, BrokerConflict e InvalidToken
7	BadRequest, BrokerNotFound e InvalidToken

Tabela 29: Erros por rota do serviço Registration.

B.2 Middleware

B.2.1 Rotas

De seguida serão ilustrados todos os campos necessários e opcionais de cada rota.

Tal como se pode observar na Tabela 22 nem todos os atributos são obrigatórios, sendo que isto acontece porque alguns dos sistemas de queues necessitam de mais dados do que outros.

O username será o nome a atribuir ao utilizador e este terá de ter um comprimento mínimo de 8 e máximo de 32 carateres, sendo estes alfanuméricos. Este atributo segue as mesmas restrições para a Tabela 39. O atributo password é muito idêntico ao atributo username mudando apenas o comprimento máximo para 64 carateres e acrescentando os símbolos ./#\$-| aos carateres alfanuméricos, na Tabela 39 este atributo possui as mesmas restrições.

O atributo broker e strategy terá de seguir os valores referidos na Secção 5.2.1, respeitando as respetivas combinações. As restrições para estes atributos são iguais para as rotas presentes nas Tabelas 34, 36 a 39 e 45 a 47.

Nos Headers da Tabela 30 é necessário colocar um Bearer token no formato JWT, sendo que este é gerado na rota 1 da Tabela 17. Este mesmo token tem de ser utilizado na rota da Tabela 39.

Os restantes atributos são semi obrigatórios porque dependem da escolha do atributo broker e strategy. A Tabela 31 ilustra quais são os atributos necessários para cada uma das combinações possíveis.

Variável	Tipo de dados	Obrigatório	Localização
username	string	sim	Body
password	string	sim	Body
broker	string	sim	Body
strategy	string	sim	Body
queue	string	semi	Body
exchange	string	semi	Body
headers	object	semi	Body
Authorization	Bearer token	sim	Header

Tabela 30: Campos da rota número 1.

A Tabela 31 aplica-se também às Tabelas 34 e 36 a 38.

Sistema de Queues	Estratégia	Campos obrigatórios
RabbitMQ	direct	exchange e queue
	topic	exchange e queue
	fanout	exchange
	headers	exchange e headers
ActiveMQ5	direct	queue
	topic	queue
Kafka	topic	queue

Tabela 31: Combinação de campos semi obrigatórios para a rota 1, 5 e 7 a 9.

A Tabela 32 representa os atributos para a rota 2 e 3 simultaneamente porque estas necessitam exatamente dos mesmos atributos.

O atributo userId é um atributo gerado na rota 1 da Tabela 18 e retornado no interior do campo data da resposta, denominado de id e ilustrado na Tabela 50. Este atributo tem o formato UUID. Das Tabelas 34 a 37 o userId representado é o mesmo utilizado na Tabela 32.

Nos Headers é necessário acrescentar um token no formato JWT no campo Authorization. Este token é gerado na rota número 4 da Tabela 18 e retornado no campo Authorization dos Headers da resposta HTTP. Este mesmo token é utilizado para as rotas 5 a 9.

Variável	Tipo de dados	Obrigatório	Localização
userId	string	sim	Path
Authorization	Bearer token	sim	Header

Tabela 32: Campos das rotas número 2 e 3.

Para a rota número 4 são necessários dois atributos, o projectToken que é o token gerado na rota 1 presente na Tabela 17 e o Header Authorization que consiste num Basic Auth token. Este token é gerado através da codificação em base 64 do username e password do utilizador separados por ':', isto é, a codificação de <username>:<password>.

Variável	Tipo de dados	Obrigatório	Localização
projectToken	string	sim	Body
Authorization	Basic auth	sim	Header

Tabela 33: Campos da rota número 4.

Na rota número 5 são apenas obrigatórios três atributos, representados na Tabela 34, o userId e Authorization já referidos e o data que irá conter os dados da mensagem a ser enviada para os sistemas de queues.

Variável	Tipo de dados	Obrigatório	Localização
data	string	sim	Body
queue	string	não	Body
exchange	string	não	Body
strategy	string	não	Body
headers	object	não	Body
userId	string	sim	Path
Authorization	Bearer token	sim	Headers

Tabela 34: Campos da rota número 5.

Variável	Tipo de dados	Obrigatório	Localização
userId	string	sim	Body
Authorization	Bearer token	sim	Header

Tabela 35: Campos da rota número 6.

Variável	Tipo de dados	Obrigatório	Localização
queue	string	não	Body
headers	object	não	Body
userId	string	sim	Path
Authorization	Bearer token	sim	Header

Tabela 36: Campos da rota número 7.

Variável	Tipo de dados	Obrigatório	Localização
strategy	string	sim	Body
queue	string	não	Body
exchange	string	não	Body
headers	object	não	Body
userId	string	sim	Path
Authorization	Bearer token	sim	Header

Tabela 37: Campos da rota número 8.

Variável	Tipo de dados	Obrigatório	Localização
broker	string	sim	Body
queue	string	não	Body
strategy	string	não	Body
exchange	string	não	Body
headers	object	não	Body
userId	string	sim	Path
Authorization	Bearer token	sim	Header

Tabela 38: Campos da rota número 9.

O atributo `persistenceTime` presente na Tabela 39 representa o tempo em milissegundos que uma mensagem recebida por um consumidor vai ficar persistida. O valor mínimo para este atributo é 0, sendo o máximo 9.223.372.036.854.775.807. Este atributo segue as mesmas restrições para a Tabela 48.

Na Tabela 39 o atributo `queue` já é obrigatório e não semi obrigatório como na Tabela 30.

Os restantes atributos são semi obrigatórios porque dependem da escolha do atributo `broker` e `strategy`. A Tabela 40 ilustra quais são os atributos necessários para cada uma das combinações possíveis, sendo que esta aplica-se também às rotas 16 a 18, Tabelas 45 a 47, respetivamente.

Variável	Tipo de dados	Obrigatório	Localização
username	string	sim	Body
password	string	sim	Body
persistenceTime	int64	não	Body
broker	string	sim	Body
strategy	string	sim	Body
queue	string	sim	Body
exchange	string	semi	Body
routingKey	string	semi	Body
headers	object	semi	Body
Authorization	Bearer token	sim	Header

Tabela 39: Campos da rota número 10.

Sistema de Queues	Estratégia	Campos obrigatórios
RabbitMQ	direct	exchange
	topic	exchange e routingKey
	fanout	exchange
	headers	exchange e headers

Tabela 40: Combinação de campos semi obrigatórios para a rota 10 e 16 a 18.

O atributo `userId` presente na Tabela 41 é um atributo gerado na rota 10 da Tabela 18 e retornado no interior do campo `data` da resposta, denominado de `id` e ilustrado na Tabela 50. Este atributo tem o formato UUID. Para as Tabelas 43 a 49 este `userId` possui as mesmas restrições.

Nos Headers é necessário acrescentar um token no formato JWT no campo `Authorization`, sendo que este token é gerado na rota número 13 da Tabela 18 e retornado no campo `Authorization` dos Headers da resposta HTTP. Este mesmo token é utilizado para as rotas 14 a 21, Tabelas 43 a 49.

A Tabela 41 representa simultaneamente os atributos necessários para as rotas 11 e 12 por serem exatamente os mesmos.

Variável	Tipo de dados	Obrigatório	Localização
<code>userId</code>	string	sim	Path
<code>Authorization</code>	Bearer token	sim	Header

Tabela 41: Campos das rotas número 11 e 12.

As restrições para Tabela 42 são as mesmas apresentadas na Tabela 33.

Variável	Tipo de dados	Obrigatório	Localização
<code>projectToken</code>	string	sim	Body
<code>Authorization</code>	Basic auth	sim	Header

Tabela 42: Campos da rota número 13.

Variável	Tipo de dados	Obrigatório	Localização
<code>userId</code>	string	sim	Path
<code>Authorization</code>	Bearer token	sim	Header

Tabela 43: Campos da rota número 14.

Variável	Tipo de dados	Obrigatório	Localização
userId	string	sim	Body
Authorization	Bearer token	sim	Header

Tabela 44: Campos da rota número 15.

Os atributos necessários e opcionais para a rota número 16 são os apresentados na Tabela 45, apesar desta ser a rota para atualização da queue a ser utilizada o próprio atributo queue não é obrigatório por causa do RabbitMQ, uma vez que os atributos routingKey e headers são atributos únicos para este sistema. Esta rota pode ser utilizada para atualização dos mesmos sem atualização da queue em si.

Variável	Tipo de dados	Obrigatório	Localização
queue	string	não	Body
routingKey	string	não	Body
headers	object	não	Body
userId	string	sim	Path
Authorization	Bearer token	sim	Header

Tabela 45: Campos da rota número 16.

Variável	Tipo de dados	Obrigatório	Localização
strategy	string	não	Body
queue	string	não	Body
exchange	string	não	Body
routingKey	string	não	Body
headers	object	não	Body
userId	string	sim	Path
Authorization	Bearer token	sim	Header

Tabela 46: Campos da rota número 17.

Variável	Tipo de dados	Obrigatório	Localização
broker	string	sim	Body
strategy	string	não	Body
queue	string	não	Body
exchange	string	não	Body
routingKey	string	não	Body
headers	object	não	Body
userId	string	sim	Path
Authorization	Bearer token	sim	Header

Tabela 47: Campos da rota número 18.

Variável	Tipo de dados	Obrigatório	Localização
persistenceTime	int64	sim	Body
userId	string	sim	Path
Authorization	Bearer token	sim	Header

Tabela 48: Campos da rota número 19.

Os atributos para a rota 20 e 21 são ambos ilustrados na Tabela 49 uma vez que estes são iguais para as duas rotas.

O atributo `messageId` segue o formato UUID, sendo que este é gerado na rota número 5 da Tabela 18 e retornado na rota número 14 dentro do campo `data` e denominado de `id`, ver Tabela 50.

Variável	Tipo de dados	Obrigatório	Localização
userId	string	sim	Path
messageId	string	sim	Path
Authorization	Bearer token	sim	Header

Tabela 49: Campos das rotas número 20 e 21.

B.2.2 Respostas

Todos os pontos de entrada, com a exceção da rota número 14, seguem o formato anteriormente referido, sendo que em caso de sucesso o valor do `appCode` será sempre 1200 e o valor do `status` será sempre SUCCESS.

Como estes dois valores são sempre constantes em caso de sucesso, não serão ilustrados na Tabela 50, onde são ilustrados os valores dos campos *message* e *data* para cada uma das rotas.

# Rota	Mensagem	Dados
1	Producer registered.	id: "ade34209-1abc-9f21-0baf-73ba942f983e" broker: "rabbitmq" strategy: "direct" exchange: "teste" queue: "myQueue" headers: { "header": "oneHeader", "anotherHeader": "twoHeader"}
2	Producer info.	Igual à rota 1.
3	Producer deleted.	Igual à rota 1.
4	Producer logged in.	Igual à rota 1.
5	Message successfully sent.	null
6	Producer connection closed.	null
7	Producer queue successfully replaced.	Igual à rota 1.
8	Producer strategy successfully replaced.	Igual à rota 1.
9	Producer broker successfully updated.	Igual à rota 1.
10	Consumer registered.	id: "ade34209-1abc-9f21-0baf-73ba942f983e" persistenceTime: 5000 broker: "rabbitmq" strategy: "direct" exchange: "teste" queue: "myQueue" routingKey: "myRouting" headers: { "header": "fstHeader", "anotherHeader": "sndHeader"}
11	Consumer info.	Igual à rota 10.
12	Consumer deleted.	Igual à rota 10.
13	Consumer logged in.	Igual à rota 10.

14	Server Sent-Event	id: "ade34209-1abc-9f21-0baf-73ba942f983e" data: "Hello World." queue: "teste"
15	Consumer connection closed.	null
16	Consumer queue successfully replaced.	Igual à rota 10.
17	Consumer strategy successfully replaced.	Igual à rota 10.
18	Consumer broker successfully updated.	Igual à rota 10.
19	Consumer persistence successfully replaced.	Igual à rota 10.
20	Message retrieved successfully.	Igual à rota 14.
21	Message successfully deleted.	Igual à rota 14.

Tabela 50: Respostas aos pontos de entrada do serviço Middleware.

É de realçar que na Tabela 50 nas rotas 1 a 4 e 7 a 9 alguns dos parâmetros como: exchange, queue e headers podem não estar definidos, valor null, e nas rotas 10 a 13 e 16 a 19 alguns dos parâmetros como: exchange, routingKey e headers podem não estar definidos (valor null). Isto acontece por devido aos atributos semi obrigatórios que podem não estar definidos caso não sejam necessários.

A rota 14 apesar de na Tabela 50 apresentar a mensagem "Server Sent-Event" esta mensagem não é enviada, apenas está presente para representar o tipo de resposta que é devolvida ao utilizador, pois os dados não serão apenas os apresentados, mas sim uma sequência de dados com o formato apresentado.

B.2.3 Erros

Existem vários erros neste serviço, sendo todos apresentados na Tabela 51 juntamente com a descrição presente na mensagem de erro. O campo Código da Tabela 51 representa o *appCode* e a Mensagem o campo *message* da resposta personalizada. O campo *data* e *status* não são representados na Tabela 51 pelos mesmos motivos apresentados na Secção 5.2.2.1 em relação à Tabela 28.

O erro *BadRequest* é lançado quando alguma das restrições apresentadas para cada uma das rotas não é cumprida. As reticências presentes na descrição deste erro representam a localização do campo que apresentava erros e o respetivo valor recebido.

Os erros *TokenProjectIncompatible* e *TokenDontBelongToUser* ocorrem quando existem problemas com os tokens, como por exemplo o token conter informação sobre outro projeto ou o token conter informação sobre um utilizador diferente daquele que invocou o método, respetivamente.

InvalidCredentials ocorre quando o utilizador tenta realizar o login e não obtém sucesso, sendo que isto acontece porque o nome de utilizador ou a password estão incorretos.

Quando um recurso não é encontrado ocorre o erro *UserNotFound* ou *MessageNotFound*, dependendo se o

recurso User, Producer ou Consumer, ou Message não existir, respetivamente.

O erro `JsonKeyInFault` é lançado unicamente quando um dos atributos semi obrigatórios não é enviado. Por exemplo na utilização do sistema de queues RabbitMQ o campo `exchange` passa a ser necessário pelo que se não for enviado despolota este erro.

Quanto aos erros `ExchangeConflict` e `UserConflict` como os nomes indicam são referentes a conflitos na criação de recursos, o primeiro é um erro específico à utilização do sistema de queues RabbitMQ, em que não podem existir exchanges com nomes iguais e estratégias diferentes. O segundo erro é quando se tenta criar um User mas já existe um com o mesmo nome de utilizador.

`BrokerNotSupported` e `BrokerStrategyIncompatible` são erros relacionados com o suporte dos sistemas de queues e suporte das respetivas estratégias. Por exemplo se for criada uma instância com suporte para RabbitMQ e Kafka, se se pretender utilizar o sistemas de queues ActiveMQ5 será lançado o erro `BrokerNotSupported`. O erro `BrokerStrategyIncompatible` ocorre por exemplo no caso de se escolher o sistema de queues Kafka com a estratégia `direct`.

O erro `InternalServerError` pode ocorrer em variadas situações, sendo um erro bastante genérico para este serviço.

`CouldNotSendMessage` ocorre quando um utilizador tenta enviar uma mensagem mas ocorre um erro na comunicação entre o serviço e o sistema de queues.

O último erro da Tabela 51 é o `CouldNotConnectToService` e este ocorre no componente `Discovery` quando este tenta conetar com umas das réplicas de `Receiver` e esta tentativa ocorre sem sucesso.

Nome	Código	Mensagem
BadRequest	1400	Values with errors: ...
TokenProjectIncompatible	1401	Token is invalid for this instance.
InvalidCredentials	1402	Username and/or password invalid.
TokenDontBelongToUser	1403	Token don't belong to producer/consumer <userId>.
UserNotFound	1404	Producer/Consumer <userId> don't exist.
JsonKeyInFault	1406	The following key is in fault on json - <key>.
MessageNotFound	1407	Message <messageId> don't exist.
ExchangeConflict	1409	Specified exchange already exists with other strategy - <exchangeName>. (RabbitMQ specific)
UserConflict	1410	Producer/Consumer <userId> already registered.
BrokerNotSupported	1415	This instance doesn't support the specified Queue System - <brokerName>.
BrokerStrategyIncompatible	1416	Queue System and Strategy are incompatible.
InternalServerError	1500	Something went wrong.

CouldNotSendMessage	1501	An error occurred while trying to send the message.
CouldNotConnectToService	1502	A service don't answer to the request.

Tabela 51: Códigos de erros do serviço Middleware.

Na Tabela 52 é realizada uma correspondência entre as rotas do serviço e os erros que podem ser lançados por cada rota, sendo que esta tabela tem como objetivo de facilitar os utilizadores deste serviço a saberem quais são as possibilidades de erros em cada uma das rotas.

#Rota	Erros
1	BadRequest, BrokerStrategyIncompatible, TokenProjectIncompatible, ExchangeConflict, UserConflict, BrokerNotSupported e JsonKeyInFault
2	UserNotFound e TokenDontBelongToUser
3	UserNotFound e TokenDontBelongToUser
4	InvalidCredentials e TokenProjectIncompatible
5	BrokerStrategyIncompatible, TokenDontBelongToUser, CouldNotSendMessage, JsonKeyInFault, ExchangeConflict e UserNotFound
6	BadRequest, TokenDontBelongToUser e UserNotFound
7	BadRequest, BrokerStrategyIncompatible, TokenDontBelongToUser, ExchangeConflict, BrokerNotSupported, UserNotFound e JsonKeyInFault
8	BadRequest, BrokerStrategyIncompatible, TokenDontBelongToUser, ExchangeConflict, BrokerNotSupported, UserNotFound e JsonKeyInFault
9	BadRequest, BrokerStrategyIncompatible, TokenDontBelongToUser, ExchangeConflict, BrokerNotSupported, UserNotFound e JsonKeyInFault
10	BadRequest, BrokerStrategyIncompatible, TokenProjectIncompatible, ExchangeConflict, UserConflict, BrokerNotSupported e JsonKeyInFault
11	UserNotFound e TokenDontBelongToUser
12	UserNotFound, TokenDontBelongToUser e CouldNotConnectToService
13	InvalidCredentials e TokenProjectIncompatible
14	UserNotFound, TokenDontBelongToUser e InternalError
15	BadRequest, UserNotFound, TokenDontBelongToUser, InternalError e CouldNotConnectToService

16	BadRequest, BrokerStrategyIncompatible, TokenDontBelongToUser, ExchangeConflict, BrokerNotSupported, UserNotFound, JsonKeyInFault, InternalError e CouldNotConnectToService
17	BadRequest, BrokerStrategyIncompatible, TokenDontBelongToUser, ExchangeConflict, BrokerNotSupported, UserNotFound, JsonKeyInFault, InternalError e CouldNotConnectToService
18	BadRequest, BrokerStrategyIncompatible, TokenDontBelongToUser, ExchangeConflict, BrokerNotSupported, UserNotFound, JsonKeyInFault, InternalError e CouldNotConnectToService
19	BadRequest, BrokerStrategyIncompatible, TokenDontBelongToUser, ExchangeConflict, BrokerNotSupported, UserNotFound, JsonKeyInFault, InternalError e CouldNotConnectToService
20	MessageNotFound, UserNotFound e TokenDontBelongToUser
21	MessageNotFound, UserNotFound e TokenDontBelongToUser

Tabela 52: Erros por rota do serviço Middleware.