FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Plataforma de Monitorização de Recursos num Sistema Message Oriented Middleware

Paulo Bordalo Marcos

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Professor Rui Filipe Lima Maranhão de Abreu

21 de julho de 2016

Plataforma de Monitorização de Recursos num Sistema Message Oriented Middleware

Paulo Bordalo Marcos

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: João Correia Lopes – *Faculdade de Engenharia da Universidade do Porto*

Vogal Externo: António Nestor Ribeiro – *Universidade do Minho*

Orientador: Rui Maranhão – *Faculdade de Engenharia da Universidade do Porto*

21 de julho de 2016

Resumo

A comunicação entre dispositivos, nos últimos anos, tem sido cada vez maior. Com a constante evolução das tecnologias e componentes cada vez mais capazes, o volume de dados que se propagam de máquina em máquina pode atingir níveis significativos.

O mxfSPEEDRAIL, desenvolvido pela MOG Technologies, é uma plataforma de ingest de vídeo utilizada pelas grandes cadeias televisivas espalhadas pelo mundo. Arquitetado como um sistema distribuído, este produto tira melhor partido das suas funcionalidades através da comunicação entre dispositivos. Deste modo, a existência de uma boa transferência de informação é essencial. Em conjunto com uma ideologia de evolução, propõe-se encontrar uma solução de comunicação que consiga corresponder às exigências dos seus clientes.

Atualmente, existem tecnologias no mercado, denominadas de Message Brokers, que prometem maior rapidez, tolerância a erros e segurança na partilha de informação. Uma análise de funcionalidades, características e a compreensão destes sistemas, são os primeiros passos para uma escolha de implementação.

Conjugado à tecnologia de propagação de mensagens, deseja-se fornecer aos administradores do produto mxfSPEEDRAIL, um módulo de gestão de recursos para melhor conseguirem identificar o estado dos seus sistemas.

Abstract

The communication between devices has increased a lot in recent years, each time with more force. With the constant evolution of technologies and with the increase of capable devices, the volume of data that goes from machine to machine can reach significant levels.

The mxfSPEEDRAIL developed by MOG Technologies, is an ingest video platform used by major broadcasters around the world. Architected as a distributed system, this product relies most of its functionality on communication between devices. Because of this, a good transmission of information is critical. Together with an ideology of progress, it is proposed a study of the best communication solutions that can meet the demands of its customers.

There are technologies on the market, called Message Brokers, that promise to bring greater speed, fault tolerance and security in communication. An analysis of features and characteristics will be the first step towards an implementation choice.

Working together with these technologies, it is essential to provide a resource management module, to administrators of mxfSPEEDRAIL, that will help the control system status.

Agradecimentos

Esta dissertação representa, para mim, o culminar dum longo caminho percorrido, conjugando os cinco anos do mestrado de engenharia informática e computação com todo o meu restante percurso académico. Desta forma, quero agradecer a todos os que me apoiaram durante os bons e maus momentos, estando lá sempre que precisei.

Gostaria de começar por agradecer a duas pessoas que já não se encontram comigo presencialmente, mas que fazem parte daquilo que sou, ensinando-me e fazendo-me sentir toda a sua força e aos quais espero ter orgulhado no cumprimento dos meus objetivos. Em especial à minha mãe, minha amiga e companheira de toda uma vida, a quem devo tudo, que fez sempre os possíveis e impossíveis para que eu pudesse continuar o meu progresso, apoiando-me em todos os momentos, principalmente nos menos sorridentes das nossas vidas. À minha namorada, Mariana, que começou um ciclo ao meu lado no inicio da minha fase universitária, prologando-o para além do final da mesma, suportando sempre toda a minha dedicação e me levantado sempre como "um". À minha família, que me acompanhou e me fez sentir preenchido, espero hoje agradecer-vos com orgulho. Aos meus amigos, os verdadeiros amigos, que os considero como família, um muito obrigado por todos os momentos felizes que me proporcionaram e pelo imenso trabalho que tivemos em conjunto.

Queria também agradecer especialmente ao meu orientador, Rui Maranhão, pelo apoio prestado e pelo tempo disponibilizado para que tudo corresse da melhor maneira. Também à MOG Technologies por me terem proporcionado este desafio tão exigente.

Esta dissertação tem um bocadinho de todos vós.

Muito obrigado!

Paulo Bordalo Marcos

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

SD	Standard Definition
HD	High Definition
UHD	Ultra-High Definition
ACK	Acknowledgement
API	Application Programming Interface
TCP	Transmission Control Protocol
AMQP	Advanced Message Queuing Protocol
UDP	User Datagram Protocol
IPC	Inter-Process Communication
PGM	Pragmatic General Multicast
P2P	Peer-to-peer
CPU	Central Processing Unit
RAM	Random Access Memory

Capítulo 1

Introdução

A captura de vídeo tem as suas mais diversas formas e propósitos. É possível dizer que atualmente uma grande parte dos utilizadores de serviços multimédia já capturou e/ou editou vídeo. As redes sociais, plataformas online e equipamentos com câmaras cada vez mais aprimoradas permitem que a sociedade contribua de forma significativa para o aumento de conteúdo audiovisual [1].

No aspeto mais profissional, como estações televisivas, a captura é constante e muito variada, tanto a nível de formatos como de conteúdo. Durante a transmissão de programas em direto, o número de câmaras pode chegar às várias dezenas, levando a um grande esforço de pessoal e de requisição do equipamento necessário.

A difusão de vídeo depende muito do utilizador final e da tecnologia que utiliza para visualizar conteúdos audiovisuais, como *TV-Boxes*, transmissão online, propagação do sinal via antena, entre outras. Como tal, existe a necessidade de ter ficheiros em vários formatos e até diferentes qualidades (*SD, HD, UHD*) para a melhor difusão entre os mais diversos dispositivos.

A MOG Technologies oferece uma solução para este inconveniente, o mxfSPEEDRAIL, uma plataforma de *ingest* de vídeo que permite captura, conversão de arquivos e transcodificação numa única plataforma com suporte para os mais variados formatos de transmissão [2].

1.1 Contexto

Este documento surge no âmbito do desenvolvimento da dissertação de Mestrado em Engenharia Informática e Computação, sendo realizado em parceria com a empresa MOG Technologies.

O tema relaciona-se com sistemas distribuídos, a comunicação e a propagação de eventos efetuados por estes sistemas. O mxfSPEEDRAIL, da MOG Technologies, é uma dessas

plataformas. Um produto de alta qualidade, que manipula diversos formatos de vídeo e *meta dados* para transferência entre editores, dispositivos e servidores. Este sistema pode ser categorizado como um sistema distribuído, um conjunto independente de máquinas que trabalham em conjunto para dar ao utilizador a ideia de um sistema singular [3].

Este produto opera através de mensagens que são propagadas para si mesmo ou para outros produtos ligados em rede. Estas mensagens desencadeiam eventos e, para estes acontecerem, as plataformas têm de ser capazes de comunicar eficazmente entre si. As mensagens são enviadas em formatos predefinidos que permitem ao recetor perceber a operação que é necessário efetuar. Quando esta comunicação não é bem-sucedida podemos encontrar problemas de desempenho ou até perder mensagens que poderiam ser essenciais para a normal atividade da plataforma.

Na fase inicial de desenvolvimento do mxfSPEEDRAIL, a comunicação entre os diversos sistemas para a propagação de eventos foi endereçada e construída pelos desenvolvedores do mesmo. Atualmente e após a evolução constante deste sistema, é necessário encontrar uma solução *message-oriented middleware,* que consiga continuar a satisfazer as necessidades dos clientes deste serviço que são cada vez mais exigentes, como grandes cadeias televisivas.

Existem diversas opções disponíveis, desenvolvidas especialmente para a resolução de problemas de comunicação e propagação de eventos, que se podem implementar e assim conseguir assegurar uma escalabilidade do sistema com segurança e sem receios. A escolha deve ser bastante ponderada devido às exigências dos clientes, à performance, aos recursos que utiliza e às tecnologias e linguagens que o produto requer compatibilidade.

Após a implementação de uma destas soluções é também importante fornecer, aos utilizadores do produto, métricas de performance para se verificar visualmente o estado e a sobrecarga das máquinas no seu sistema. Esta informação será apresentada utilizando os métodos de comunicação que se pretende implementar.

1.2 Motivação

Sistemas têm que lidar cada vez mais com maiores volumes de dados. Estes dados são transacionados, diariamente, em grande quantidade e variedade, tanto em sistemas abertos (exemplo das plataformas web) como em sistemas fechados (exemplo dos sistemas distribuídos de empresas ou intranets) [4].

A distribuição de tarefas pode ser essencial para um melhor aproveitamento dos recursos do sistema. Se existir sobrecarga em apenas uma das diversas máquinas que constituem o sistema, podemos eventualmente verificar perdas de performance que podem potencializar a perda de utilizadores [5].

Esta distribuição de tarefas pode ser efetuada através de eventos. Estes eventos podem ser vistos como mudanças de estados das plataformas que irão desencadear um conjunto de processamento de funcionalidades. No entanto, eventos não transitam de uma máquina para outra.

Este termo é regularmente usado para mencionar notificações em plataformas geradas por mensagens recebidas, o que pode gerar alguma confusão. Para os eventos ocorrerem é necessário que exista propagação de mensagens e, como tal, uma boa comunicação é necessária para que este tipo de informação consiga chegar ao seu destino no mesmo formato e configuração em que foi remetida.

Existe uma área de desenvolvimento que perspetiva a melhoria das comunicações entre plataformas. Este desenvolvimento deu origem aos *Message Brokers* que prometem oferecer maior rapidez na transferência de mensagens, maior segurança e maior tolerância a falhas [3].

1.3 Objetivos

Os objetivos deste trabalho são:

- Compreensão da geração de eventos em sistemas distribuídos assim como a sua comunicação.
- Consciencialização da vasta oferta de *Message Brokers* e do seu funcionamento.
- Investigação da melhor solução *Message-Oriented Middleware* em relação às exigências do produto mxfSPEEDRAIL.
    - Filtragem ao nível das linguagens, plataformas suportadas e outros requisitos.
    - Implementação de um sistema semelhante a um cenário real com casos de estudo.
    - Ensaio de transferências das mais diversas mensagens e de possíveis falhas do sistema.
- Avaliação de performance entre as diversas soluções para uma decisão de confiança.
- Integração de uma solução baseada em filas de mensagens na atual arquitetura do produto mxfSPEEDRAIL.
- Criação de um módulo de monitorização web que apresentará aos administradores do sistema os recursos que estão a ser aproveitados pelas máquinas.

1.4 Estrutura do Documento

Neste documento, em conjunto com o capítulo de introdução, podemos contar com 4 capítulos.

O capítulo 1, de Introdução, oferece uma rápida contextualização do âmbito desta dissertação. Durante o capítulo 2, é realizado um levantamento do estado da arte dos melhores meios de comunicação, tendo como base as filas de mensagens e os Message Brokers que melhor conseguem demonstrar o seu funcionamento geral. No capítulo 3, é descrito o processo de realização de testes de desempenho, assim como a análise aos seus resultados. O capítulo 4, oferece uma visão geral do sistema de propagação de mensagens do produto mxfSPEEDRAIL e ainda a descrição da nova implementação, assim como a demonstração da plataforma de monitorização de recursos. O último capítulo concluí, de forma sucinta, tudo o que foi tratado nos capítulos anteriores e oferece também uma visão geral da solução.

Capítulo 2

Estado da Arte

Um sistema distribuído designa-se por um conjunto de computadores ligados em rede onde a comunicação entre estes é o alicerce de todo o processo. Mensagens, concorrência, partilha de recursos e assincronismo são umas das principais características deste tipo de sistema onde uma estrutura *Message Oriented Middleware* suportará o envio e a receção de informação [6]. Esta estrutura cria uma abstração dos pormenores técnicos da ligação e de serviços que interligam toda a rede.

Como tal, existem diversas bibliotecas desenvolvidas apenas com o objetivo de melhorar a comunicação entre máquinas. Estas bibliotecas são usualmente denominadas de *Message Brokers*. São responsáveis pelo encaminhamento das mensagens, da sua codificação num formato compreensível para o recetor, da sua descodificação, no tempo correto e responder a erros provocados por eventos [7]. Dentro do conjunto existente de bibliotecas, existem algumas diferenças que podem ser apropriadas a casos diferentes, como a escala para que foram construídas, os paradigmas de programação utilizados para o seu desenvolvimento e os requisitos que necessitam para funcionar como previsto.

2.1 Message Brokers

Para além da comunicação e partilha de dados entre sistemas, é também necessário que as mensagens sejam compreendidas e entregues de forma a que o seu propósito seja atingido. O formato de codificação e descodificação de mensagens deve ser conhecido e compreendido pelas diversas plataformas.

Um *Message Broker* é um gestor de mensagens, responsável pelas filas onde estas serão guardadas (i.e., *queues*), com o propósito de administrar as comunicações maioritariamente assíncronas. Ao receber mensagens, irá verificar quais os seus destinatários e o formato que os

mesmos aceitam e, caso o formato seja diferente, efetuará as operações de conversação e tradução necessárias, reduzindo assim a carga sobre os recetores [3].

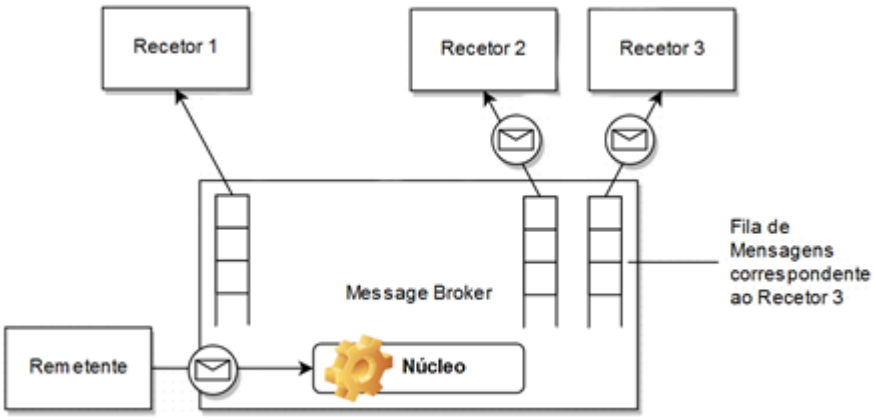

Figura 1: Arquitetura simples de um Message Broker.

Existem várias configurações de conservação de mensagens nas filas de espera, assegurando um melhor funcionamento de um determinado sistema. É possível definir se pretendemos que as mensagens se mantenham por consumo ou por tempo. Por consumo, as mensagens ficam guardadas indefinidamente até algum consumidor efetuar a sua leitura, após isso a mesma é eliminada da fila de espera. Por tempo, as mensagens ficam guardadas por um tempo predefinido. Há também formas de combinar estes dois padrões, criando um modelo hibrido, onde a informação se mantém por um determinado tempo e é removida da fila ou após consumo ou após a duração configurada terminar.

A grande maioria dos *Message Brokers* aceita dois tipos de modelos de partilha de mensagens: ***Point-to-Point*** e ***Publish/Subscribe***. Estes dois modelos devem ser analisados e considerados consoante a ideologia do produto que se deseja desenvolver. Baseiam-se ambos na transmissão de mensagens mas a forma como estas se propagam e são consumidas é um pouco diferente [3].

Em ***Point-to-Point*** os produtores enviam mensagens para uma fila à sua escolha e os clientes, que pretendem receber mensagens dessa mesma fila, ficam constantemente à escuta até que algo seja entregue. Podem existir vários produtores a enviar para uma única fila assim como vários consumidores a escutar um mesmo ponto, no entanto neste modelo uma mensagem é apenas entregue a um consumidor, sendo que logo após esta entrega a mensagem será eliminada da fila para evitar sobrecargas. Alguns *Message Brokers* são bastante configuráveis e permitem definir diferentes formatos de envio e consumo para este modelo [3]:

- **FIFO** – *First In First Out* – As mensagens são ordenadas e consumidas por ordem de chegada. O primeiro consumidor a efetuar a leitura da mensagem deve enviar um sinal de confirmação para que se saiba que a mesma foi entregue. Após a leitura da mensagem e da confirmação de receção, esta deve ser eliminada para não ser consumida por outro cliente [8].

- **FIFO Priorities** - *First In First Out with Priorities* – O seu funcionamento é muito semelhante ao formato FIFO, contudo existem alturas em que é necessário enviar uma mensagem com urgência para o topo da fila para que seja consumida primeiro que qualquer outra. Este formato fornece mais um campo que o produtor pode preencher nos *meta dados* da mensagem, indicando a sua prioridade. Em alguns *Message Brokers,* este campo é preenchido apenas com um número inteiro que quanto maior for, maior será a urgência na entrega da mensagem [8].

Algumas *frameworks* providenciam de raiz um sistema simples de ***Load Balance***, que distribui as mensagens uniformemente pelos consumidores de modo a não existir sobrecarga excessiva em apenas alguns. Como em *Point-to-Point* o primeiro a rececionar informação é o que fica responsável por efetuar a operação solicitada, existe a possibilidade de ser sempre o mesmo consumidor a produzir a grande maioria do trabalho, pois não existe controlo sobre as ordens de leitura. Com este sistema, o *Message Broker* fica encarregue de determinar quais os consumidores que já efetuaram operações e apenas autoriza os menos sobrecarregados a efetuar o novo trabalho, permitindo assim uma melhor gestão de recursos e por vezes melhor performance (dependendo do nível de processamento de cada consumidor) [9].

Publish/Subscribe é, nos dias de hoje, o método mais utilizado para difusão de informação.

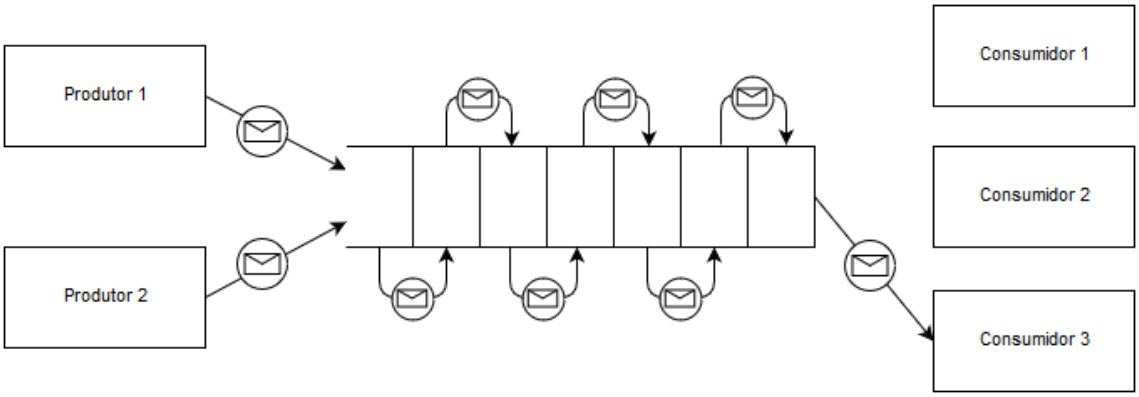

Figura 2: Arquitetura Point-to-Point com filas de mensagem.

Cada elemento de um sistema distribuído pode ter vários papéis: apenas *Publisher*; apenas *Subscriber*; tanto *Publisher* como *Subscriber*. *Publishers* enviam informação na forma de

eventos para todos os clientes subscritos, os *Subscribers*. Já os *Subscribers* definem os seus interesses, podendo subscrever um vasto número de *Publishers* ou canais [10].

Publishers não têm conhecimento direto dos *Subscribers* que pretendem receber as suas mensagens e também não efetuam um envio direto para estes destinatários. Definem, no entanto, parâmetros nas mensagens que permitem a um gestor intermédio efetuar o reencaminhamento para todos os que pretendem escutar deste remetente. No caso dos Message Brokers, são estes os responsáveis pela gestão deste método e do encaminhamento das mensagens [10].

Existem vários modelos de subscrição, permitindo a melhor configuração de um sistema evitando sobrecarga a nível do *Message Broker* como do cliente:

- *Topic-based Model* – Neste modelo um *Subscriber* subscreve um tópico e não um *Publisher*. Todas as mensagens enviadas para esse tópico serão difundidas para o conjunto de recetores que nele declararam interesse. Estes tópicos podem ser vistos como canais onde cada sistema efetua uma conexão. O principal problema deste modelo é a falta de expressividade e filtragem das mensagens. Os recetores irão receber tudo o que for enviado para o tópico e terão a própria responsabilidade de filtrar aquilo que é diretamente do seu interesse, criando assim uma maior sobrecarga do cliente [10].

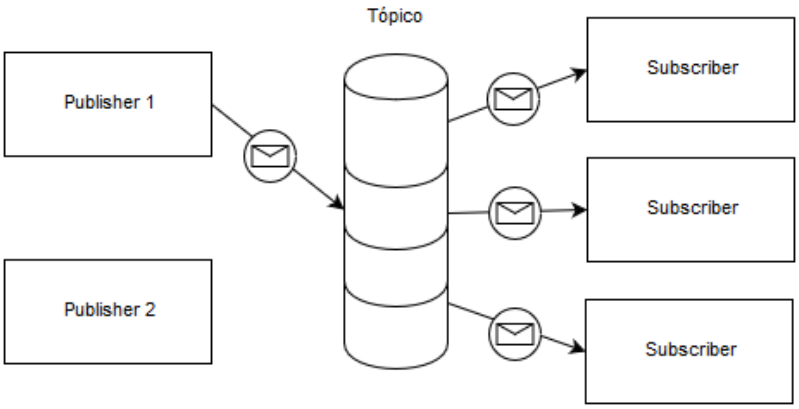

Figura 3: Arquitetura Publish/Subscribe modelada em Topic-based

- *Filtered-Topic Model* – Uma variante do modelo *Topic-based* onde a principal diferença está na filtragem de mensagens pelo gestor e não pelo cliente. Os *Subscribers* ao subscreverem um tópico definem um filtro do que pretendem que a mensagem contenha. No momento em que a mensagem chega ao tópico, o gestor verifica o seu conteúdo e, se não satisfizer o filtro, não será entregue ao destinatário [10].

- *Content-based Model* – Os *Subscribers* declaram os seus interesses definindo atributos que pretendem que as mensagens contenham. A subscrição é agora um conjunto de restrições, em grande parte expressões regulares, que serão utilizadas para comparação e critério de difusão. Existe uma abstração a nível de tópico, pois neste modelo a filtragem é feita através do conteúdo da mensagem e não do canal a que se destina [10].

- *Type-based Model* – Os eventos são categorizados por tipo através de atributos definidos pelos remetentes. Um *Subscriber* pode definir os tipos de mensagens que pretende receber. Um identificador deve fazer parte da mensagem para que a filtragem seja possível. Estes identificadores são normalmente palavras ou chaves. Em certos casos os subscritores podem definir expressões regulares que aceitarão quaisquer mensagens que tenham correspondência. Tipos representam um modelo de dados mais robusto para o desenvolvimento, criando estruturas que facilitam a integração de novos elementos no sistema [10].

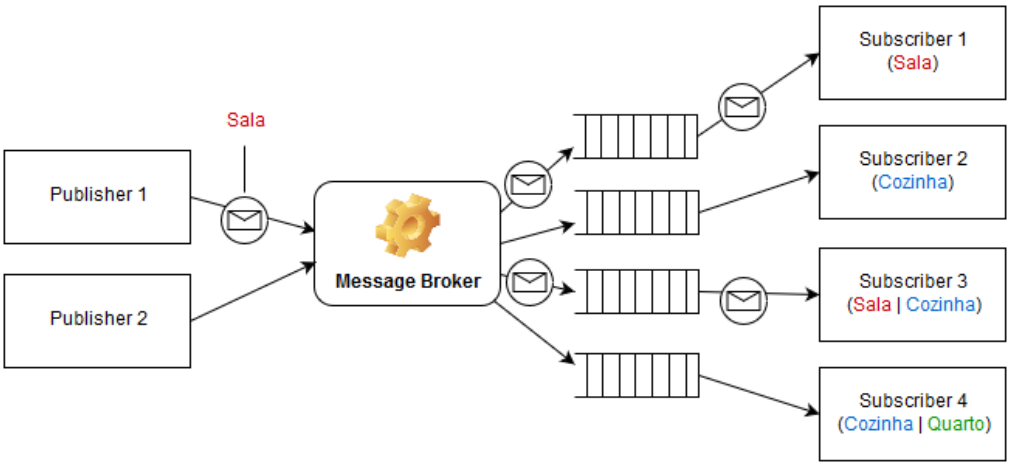

Figura 4: Arquitetura Publish/Subscribe modelada em Type-based

Há várias maneiras de classificar sistemas *Publish/Subscribe*. Permite-nos também desenvolver sistemas que vão de encontro às principais necessidades de cada produto, devido ao elevado nível de configuração e fácil adaptação dos padrões já existentes. É preciso também ter em conta que a escolha do *Message Broker* que melhor se adequa às características e expetativas é fundamental. A escalabilidade, segurança e a tolerância a erros devem ser considerados.

2.1.1 ZeroMQ

ZeroMQ [11], desenvolvido pela iMatix Corporation, é uma biblioteca de comunicação que fornece *sockets* que transportam mensagens através de vários meios, como TCP ou *multicast*. Permite ligações entre nós (máquinas) de complexidade N para N e tem suporte para os principais modelos de partilha como *Point-to-Point* e *Publish/Subscribe*.

O seu foco principal é garantir rapidez mesmo em redes bastante complexas. Por trás do seu nome está a ideia de implementação de latência zero, de administração zero, de custo zero e de desperdício zero. Esta é a ideologia perfeita quando se pensa num serviço de mensagens, no entanto não é, de forma evidente, possível conseguir garantir estes valores, muito embora sejam estes o foco do seu desenvolvimento.

2.1.1.1 Arquitetura *Brokerless*

Apesar do seu nome o parece indicar, ZeroMQ não é um sistema de *Message Queuing* por definição, devido à não existência de um *broker* semelhante ao de outras *frameworks* do género. Em sistemas de *Message Queuing*, os elementos ligam-se a um gestor central (i.e *broker*) que ficará responsável pela maior parte do processo de divulgação de informação. No caso do ZeroMQ, os elementos do sistema ligam-se diretamente num estilo P2P [12].

É, como mencionado anteriormente, mais uma biblioteca de comunicação que permite concorrência, mensagens assíncronas e endereçamento de rotas. Facilmente configurável e adaptável ao desenvolvimento em torno das necessidades de cada produto [12].

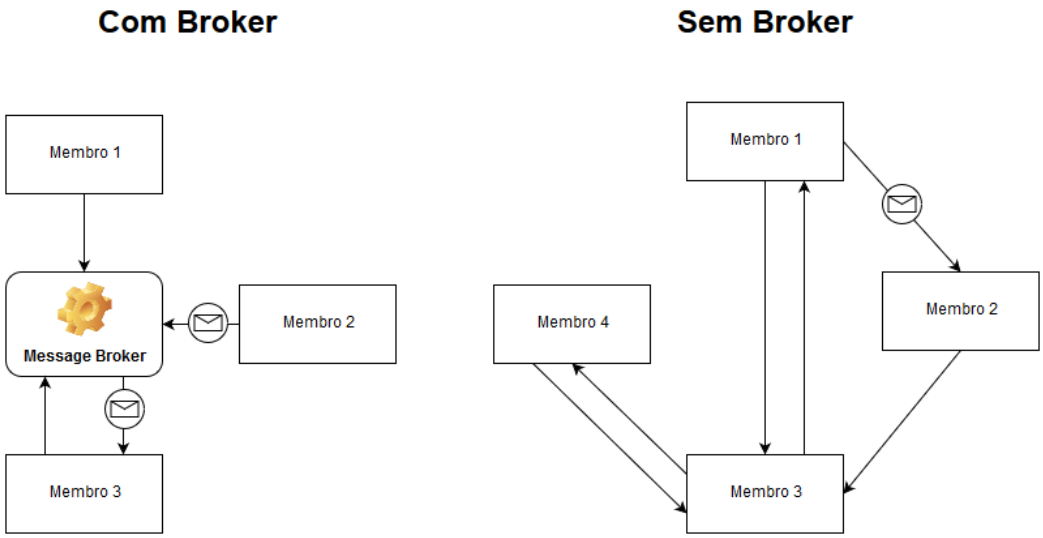

Figura 5: Exemplo de uma arquitetura com e sem broker (ZeroMQ)

2.1.1.2 ZeroMQ Sockets

ZeroMQ fornece *sockets*[1] um pouco diferentes dos mais convencionas, como TCP e UDP. Estes foram desenvolvidos tendo em mente a transmissão de mensagens no contexto de sistemas distribuídos com filas de mensagens. O protocolo TCP baseia-se na transmissão de informação com a garantia de que a mesma chegará ao seu destino. Já o protocolo UDP não fornece garantia de entrega, deixando esse encargo às aplicações caso seja necessário. Ambos são difíceis de utilizar de maneira assíncrona e as suas ligações são normalmente de 1 para 1 [11].

Por outro lado, os *sockets* de ZeroMQ fornecem uma abstração ao nível das filas de mensagens em memória e comunicam através dum processo a correr em segundo plano e em paralelo com a nossa aplicação. Quando pretendemos enviar uma mensagem, escrevemos a mesma num *socket* deste tipo e esta ficará guardada numa fila de mensagens em memoria (correspondente ao *socket* que estamos a utilizar, pois dois *sockets* equivalem a duas filas de

[1] Socket: são uma abstração para endereços de comunicação através dos quais processos comunicam [36].

mensagens). Após o armazenamento, o processo que está a executar em segundo plano irá enviar a mensagem para outro *socket*, no lado do cliente, que ficará responsável pelo seu armazenamento em memoria até à sua leitura [12].

Outra diferença está no número de ligações por *socket*. Enquanto que nos métodos mais convencionais para comunicarmos com N clientes são precisos N *sockets*, no caso do ZeroMQ podemos ter apenas 1 *socket* para efetuar comunicação com N clientes. Isto simplifica bastante a comunicação e o conjunto de conexões necessárias [12].

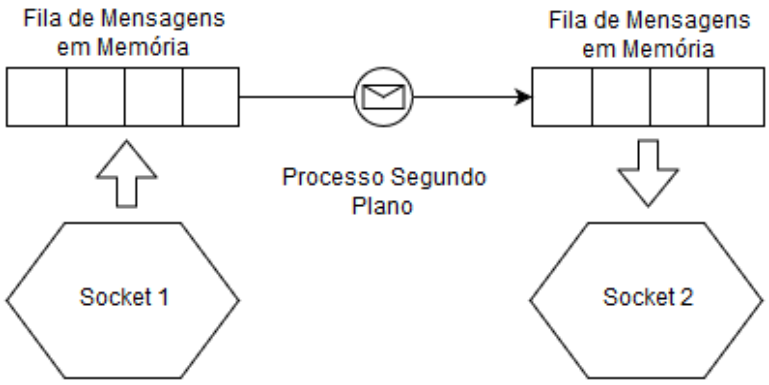

Figura 6: Funcionamento dos Sockets de ZeroMQ

2.1.1.3 Características e Formato de Mensagens

As mensagens em ZeroMQ são também um ponto fundamental desta *framework* e contêm algumas características que as diferenciam:

- Assíncronas - permitem o envio em paralelo sem entraves.
- Atómicas – enviados com um todo e não divida em partes.
- Multipartidas – fornecem campos de *meta dados* para melhor configuração.
    - Um *frame* para o identificador do remetente.
    - Um *frame* para identificar um delimitador (se existir).
    - Um *frame* para o corpo da mensagem.

Teoricamente as mensagens em ZeroMQ não têm um limite de tamanho. Dependendo do tipo de socket este tamanho pode ir exclusivamente até 2^64 bytes (o tamanho irá sempre influenciar a performance do sistema e estes valores não se devem aplicar na prática) [12].

2.1.1.4 Modelos de Transporte

Fornece diferentes níveis de transporte para vários casos. Se quisermos comunicação entre máquinas distintas podemos utilizar o protocolo TCP. Caso haja necessidade de comunicação entre processos a correr na mesma máquina, o protocolo IPC é o mais aconselhado. Na eventualidade de ser necessário a criação de um sistema *Publish/Subscribe* entre diferentes máquinas, utiliza-se o protocolo PGM [12].

2.1.1.5 Síntese

A sua rapidez, pequeno tamanho, suporte para diversos tipos de protocolos e o grande nível de abstração que oferece, tornam esta biblioteca bastante robusta. No entanto alguns elementos como serialização ou compressão ainda não fazem parte desta *framework.* A ultima versão contém, de raiz, um elemento de segurança, chamado CURVE, que permite ser ligado e desligado sempre que assim for preciso, no entanto ainda é bastante recente e poucos testes foram efetuados em casos reais [13].

Esta biblioteca esta sob a licença LGPL, uma licença de software livre e aprovada. Pode também ser utilizada em diversas plataformas e linguagens (C, C++, C#, JAVA, Python, …) [11].

2.1.2 RabbitMQ

RabbitMQ, da Pivotal Software, é um sistema *opensource* de gestão de mensagens multiprotocolo desenvolvido em Erlang. É disponibilizado um servidor para instalação em máquinas com os mais diversos sistemas operativos (Windows, Debian, Ubuntu, Fedora, …). Os seus clientes podem ser desenvolvidos nas mais diversas linguagens de programação, sendo uma *framework* poliglota. Tem suporte para os principais modelos de partilha como *Point-to-Point* e *Publish/Subscribe*, assim como vários modelos de subscrição [14].

2.1.2.1 Arquitetura e Exchanges

Quando efetuamos partilha de mensagens, temos pelo menos um remetente e um consumidor. Para a informação chegar de um ponto a outro, é necessário existir uma ligação entre estes. Em RabbitMQ é sempre imprescindível a existência de um ***Exchange***, mesmo que seja uma simples ligação de 1 para 1 com uma fila de mensagens entre os dois pontos. As mensagens nunca poderão ser enviadas para uma fila diretamente sem passarem por um *Exchange*, é esta a ideologia por trás deste sistema. Um remetente enviará sempre as suas mensagens para um *Exchange* e nunca para outro tipo de objetos (nem *queues*, nem destinatários) [15].

É algo muito simples, de um lado recebe mensagens de produtores, de outro envia-as para as filas de mensagens para serem lidas por consumidores. No entanto um *Exchange* tem sempre de saber o que deve fazer com as mensagens. Deverá reencaminha-las para que fila? Deve ser

reencaminhada apenas para uma ou para várias? Tudo isto é resolvido através da definição de um tipo para cada *Exchange* [16].

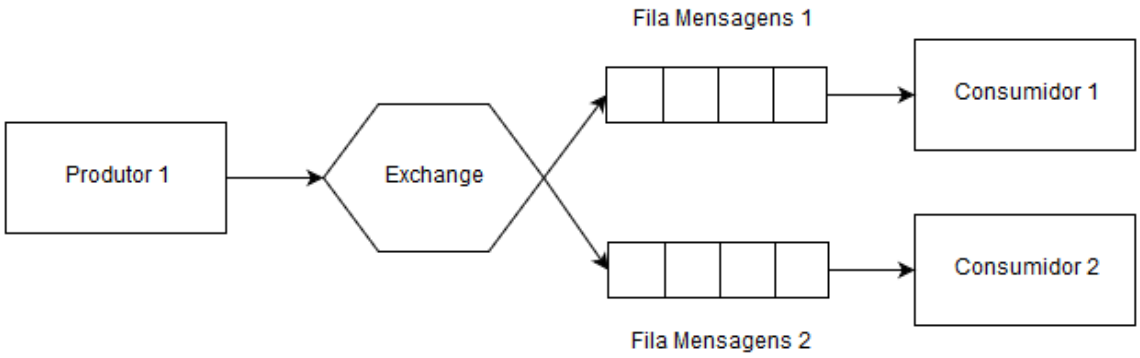

Figura 7: Arquitetura simples de um sistema RabbitMQ com Exchange

Os 3 principais tipos de configuração de *Exchanges* são:

- *Direct* – Um *Exchange* configurado como *Direct*, apenas entregará dados às filas que contiverem a mesma *routing key* que a mensagem. Ao enviar uma mensagem o produtor pode definir qual a *routing key* que pretende, filtrando assim filas que não correspondem. É ideal para encaminhamento *unicast*, embora também possa ser usado para *multicast* [15] .

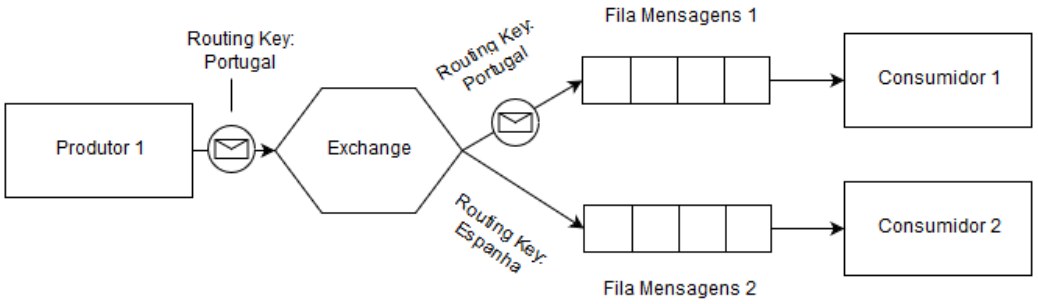

Figura 8: Arquitetura de um sistema RabbitMQ com Exchange do tipo Direct

- *Fanout* - Um *Exchange* configurado como *Fanout*, entregará as mensagens a todas as filas de espera que a ele estão ligadas. Se estiverem ligados N filas de espera, então serão enviadas N cópias de uma mensagem, uma para cada fila. Ignora assim as *routing keys* mesmo que estas tenham sido definidas. É ideal para *broadcast* [15].

- *Topic* - Um *Exchange* configurado como *Topic*, apenas entregará dados às filas cuja expressão regular corresponda à expressão definida na *routing key* da mensagem. Através das expressões regulares podemos considerar imensos casos de utilização, criando uma maior capacidade de definição de arquitetura [15].

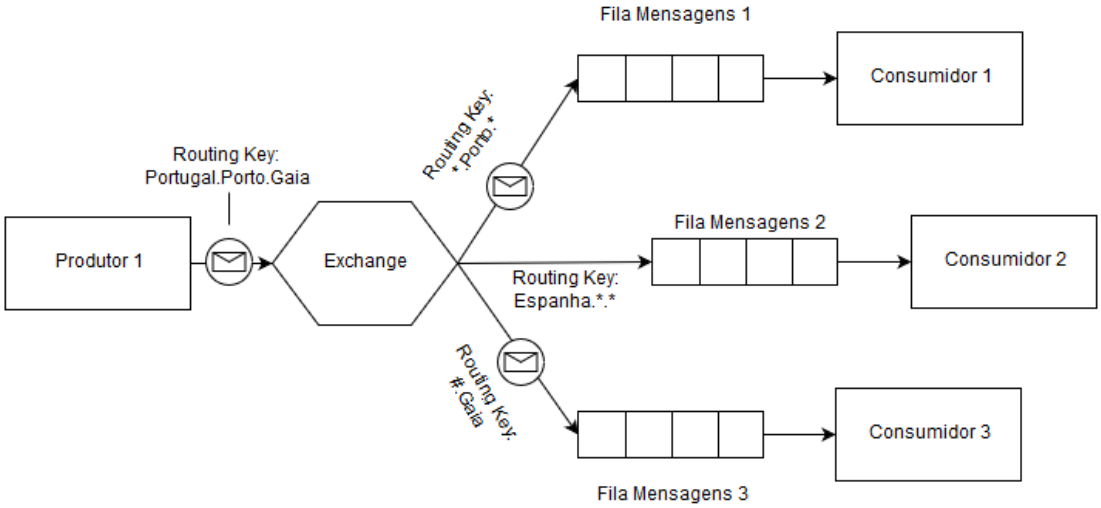

Figura 9: Arquitetura de um sistema RabbitMQ com Exchange do tipo Topic

As expressões regulares, em RabbitMQ, contêm algumas características que ajudam a melhor definir as rotas. Os termos, ou grupos, devem ser separados por '.'. Quando não queremos especificar um termo, utilizamos um '*' no lugar do mesmo. Sempre que for necessário criar uma correspondência para um ou mais termos, desde que estejam unidos, utilizamos um '#' [17].

2.1.2.2 Load Balancer

RabbitMQ implementa de raiz um *Load Balancer*. Sempre que existe mais que um consumidor ligado a uma fila de mensagens, as mensagens serão distribuídas uniformemente de forma a evitar sobrecarga em apenas alguns destinatários. Isto é tratado pelo servidor RabbitMQ mesmo que os consumidores estejam ligados em rede e em máquinas diferentes [15].

2.1.2.3 Características de Mensagens

Nos parâmetros de configuração de uma conexão, é possível definir um tamanho máximo de mensagem. Se esta ultrapassar esse tamanho, não será entregue e o produtor será avisado com uma mensagem de erro. RabbitMQ implementa o protocolo AMQP para o envio de mensagens. Este protocolo permite tamanhos que podem ir exclusivamente até 2^64 bytes (o tamanho irá sempre influenciar a performance do sistema e estes valores não se devem aplicar na prática) [18].

2.1.2.4 Plataforma de Gestão

Os servidores RabbitMQ providenciam uma ferramenta gráfica de gestão [19], que permite aos administradores de sistema controlarem e perceberem melhor possíveis problemas em tempo real. Esta plataforma pode ser acedida através de um *browser* na máquina onde o servidor está a executar. Contém também as seguintes funcionalidades:

- Declarar, listar e apagar *Exchanges*, filas de mensagens, ligações, utilizadores e permissões.
- Monitorizar as filas de mensagens assim como recursos de rede e memória utilizados nos mais diversos pontos do sistema.
- Enviar e receber mensagens.
- Recursos utilizados pelo servidor como memória e processador.
- Forçar o cancelamento de conexões e destruir filas de mensagens em tempo real.

2.1.2.5 Síntese

No geral, RabbitMQ destaca-se por alguns pontos fortes [14] que asseguram confiança a quem pretende implementar um sistema de *Message Queuing*:

- Configuração Elevada – oferece um alto nível de configuração para os mais diversos tipos de sistemas.
- Fiabilidade - oferece diversas formas de garantia na entrega de mensagens através de *acknowledgements* e de uma alta disponibilidade dos serviços.
- Flexibilidade no encaminhamento das mensagens - É possível definir hierarquias e grupos para que certas mensagens se dirijam apenas a nós com determinados níveis.
- *Queues* de alta disponibilidade - Caso as mensagens não sejam entregues e ainda se encontrem numa fila de espera, a máquina que armazena esta fila pode falhar. Através de várias maquinas em *cluster* podemos replicar estas filas para assegurar a não existência de um único ponto de falha.
- Multiprotocolo - Adequar o nosso serviço de mensagens às nossas necessidades.
- Gestão gráfica - Dispões de uma interface gráfica integrada que permite monitorizar todo o processo de comunicação visualmente.

- *Tracing* - Se o sistema não estiver a comportar-se como seria expectável, a biblioteca fornece um método de rastreio para descobrir o ponto de falha.

Algumas empresas de renome como o Instagram, a Mozilla e Telefónica utilizam este *Message Broker* nos seus sistemas [17].

Esta biblioteca está sob a licença MPL, uma licença de software livre e aprovada. Pode também ser utilizada em diversas plataformas e linguagens (C, C++, C#, JAVA, JavaScript, Python, Node.js) [15].

2.1.3 MSMQ

MSMQ [20], desenvolvido pela Microsoft, é uma tecnologia de *Message Queuing* que permite diferentes máquinas em diferentes tempos de execução ligadas em rede, comunicarem entre si mesmo que algumas dessas máquinas estejam temporariamente indisponíveis. As mensagens são enviadas para uma *Queue* e lidas através da mesma. Se uma máquina estiver *off-line* no momento do envio poderá depois verificar se na lista existe alguma mensagem, não se perdendo assim informação.

2.1.3.1 Arquitetura

MSMQ tem uma arquitetura muito simples e semelhante ao modelo *Point-to-Point*. Os produtores apenas definem para que fila querem mandar as suas mensagens e o sistema MSMQ irá tratar do seu armazenamento (em memoria física ou virtual, dependendo da sua configuração), da conexão com um recetor para o envio e, após entregue, retira-a também da fila. Se algo falhar e a mensagem não for entregue a um consumidor, a mesma é devolvida à fila para posteriormente se tentar de novo [21].

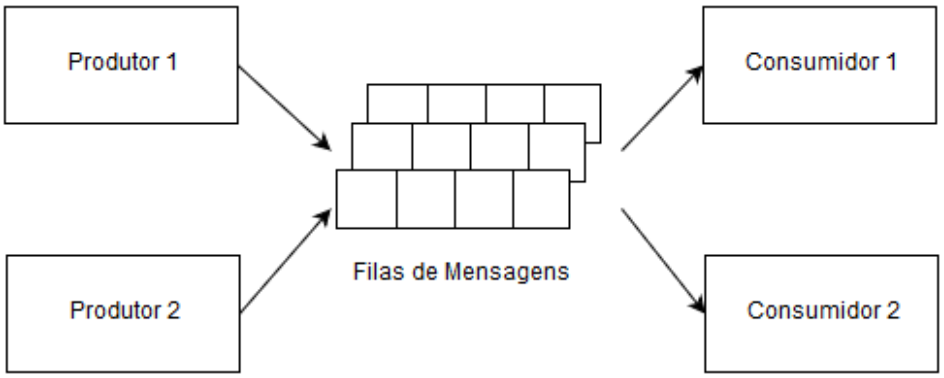

Figura 10: Arquitetura de MSMQ

Como mencionado anteriormente, o modelo de MSMQ é muito semelhante a *Point-to-Point*. O modelo *Publish/Subscribe* não é diretamente endereçado por esta tecnologia. Existem vários

métodos de desenvolvimento para a criação deste modelo em MSMQ, mas este tem de ser concebido e ajustado pelos responsáveis da criação de um produto. Como vimos anteriormente existem outras soluções que já fornecem estes procedimentos, portanto devemos considerar se o que o MSMQ oferece cumprirá as exigências necessárias [22].

2.1.3.2 Filas Públicas e Privadas

Em MSMQ as filas de mensagens podem ser públicas ou privadas. São ambas semelhantes no formato de escrita e de leitura, diferindo apenas nas preferências de localização e acesso por sistemas remotos. Ambas podem ser acedidas por outras máquinas, desde que as aplicações remotas saibam o endereço do servidor, o nome e a designação da fila. A diferença está na possibilidade de as aplicações poderem solicitar e receber conhecimento de todas as filas públicas, assim como o seu nome e designação. As filas privadas não podem ser descobertas por aplicações remotas, este tipo de informação tem de ser partilhado por outros meios [22].

Estas duas possibilidades podem ser bastante uteis na criação de sistemas complexos em que existem hierarquias.

2.1.3.3 Características de Mensagens e Prioridades

As mensagens em MSMQ podem ser enviadas nos mais diversos formatos, desde que estes possam ser convertidos para um conjunto de bytes. No entanto existe uma limitação de tamanho definida pelo sistema que não pode ser ultrapassado. Esse tamanho é de 4MB, o que pode ser um grande entrave na escolha desta tecnologia. O principal objetivo de MSMQ é a partilha de mensagens simples, para a geração de eventos, podendo um desses eventos ser um pedido REST para a transferência de um ficheiro superior a 4MB [23].

Existe também um procedimento de definição de prioridades das mensagens. As filas em MSMQ seguem o padrão *FIFO (First In First Out)*, mas pode existir a necessidade de enviar uma mensagem com urgência para o topo da fila de forma a ser consumida com a maior brevidade possível. Como tal, é possível definir nos *meta dados* da mensagem um campo que irá conter um número inteiro. Quanto maior for esse número, maior será a sua urgência. Assim que a mensagem chegar à fila, o sistema irá verificar se existem mensagens com prioridade inferior e passará a recém-chegada para a frente destas [23].

2.1.3.4 Síntese

Um ponto que pode ser favorável a alguns produtos, é a integração de raiz de MSMQ em máquinas com o sistema operativo Windows, da Microsoft. Porém é preciso ter em conta que esta tecnologia não está disponível para outros sistemas operativos e a criação de clientes em línguas não diretamente suportadas pela Microsoft pode gerar dificuldades ou até impossibilidade. É um sistema bastante leve e de fácil integração. Para além destas duas vantagens, dispõe também de

um módulo de monitorização, de um módulo de escrita de mensagens em disco para persistência de informação, duma API bastante simples e compreensível com um rápido nível de aprendizagem, uma documentação bem delineada e a garantia de constantes atualizações que acompanharão a evolução dos produtos Microsoft [23].

Esta biblioteca é gratuita, mas apenas pode ser utilizada com o sistema operativo Windows e em produtos Microsoft. As linguagens que suporta são (C++, C#, Visual Basic) utilizando a *framework* .NET [21].

2.1.4 Apache Kafka

Apache Kafka [24], da Apache Software Foundation, é um projeto *opensource* que fornece as mesmas funcionalidades de uma biblioteca de comunicação, mas arquitetado de maneira diferente.

A maior rede social profissional do mundo, o LinkedIn, gera diariamente biliões de *log entries* produzidos pelos mais de 300 milhões de utilizadores. É, portanto, indispensável conseguir responder às necessidades, em tempo real, dos produtos dos utilizadores e dos sistemas de administração. Como tal, Kafka foi inicialmente desenvolvido por esta rede social com a intenção de resolver problemas de entrega de grandes volumes de dados num sistema *Publish/Subscribe*. Pode ser visto como um *commit-log* para toda a infraestrutura de um sistema que providencia uma abstração ao nível do mecanismo de extração de mensagens, permitindo tanto subscritores como produtores lerem as mesmas a ritmo arbitrário. A comunicação entre clientes e servidores é bastante simples através do protocolo TCP [25].

2.1.4.1 Arquitetura, Tópicos e Partições

O modelo utilizado por esta *framework* é o *Publish/Subscribe*, contudo são visíveis pequenas diferenças entre este e outros *Message Brokers* que também implementam este modelo.

Kafka foi desenhado tendo em conta uma arquitetura em *cluster* que servirá de *backbone* de todo um produto. Este *cluster* não é mais que um grupo de servidores a trabalhar em conjunto, formando um único sistema que providencia grande disponibilidade de serviços aos clientes. Quando acontece uma falha numa das máquinas do cluster, os recursos são redistribuídos pelas restantes. Desta forma o sistema continua o seu funcionamento sem interrupções [26].

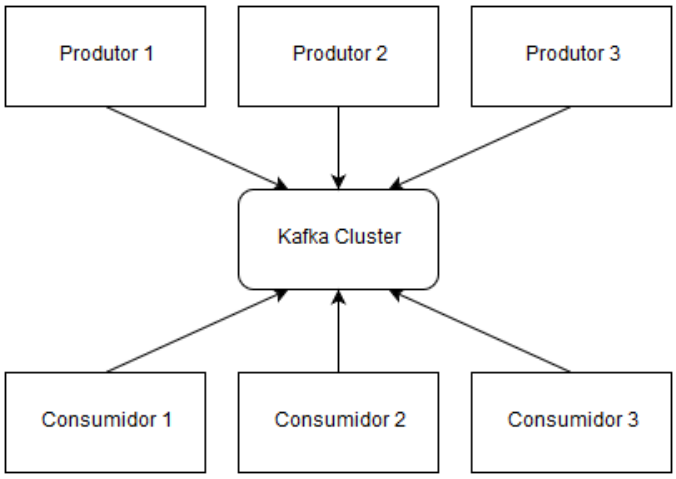

Figura 11: Arquitetura em cluster de Apache Kafka

Kafka providencia também um alto nível de abstração denominado de **Tópico**. Estes Tópicos podem ser vistos como endereços para onde os produtores enviarão as suas mensagens. Cada Tópico deve pertencer a uma categoria de mensagens. Estas categorias podem ser definidas à priori ou sempre que um produtor achar necessário a criação de uma nova. Tanto os consumidores como os produtores podem ler e escrever em vários Tópicos. As mensagens são guardadas num *broker,* em formato físico ou virtual (dependendo da configuração do serviço e do tamanho das mensagens) [26].

Cada Tópico contém várias Partições. As mensagens quando chegam a um tópico são ordenadas dentro de cada Partição. As entradas nas Partições são sequenciais e em cada uma é atribuído um número identificador único, intitulado de *offset*, para facilitar o seu reconhecimento [26].

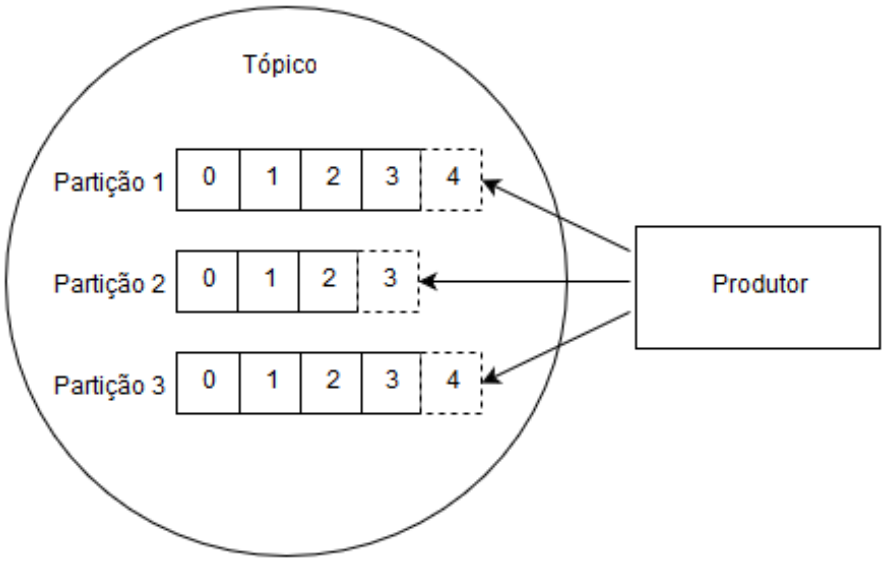

Figura 12: Anatomia de um Tópico de Kafka

Cada partição é replicada por um número (previamente configurável) de máquinas que constituem o *cluster*. Este sistema cria uma melhor tolerância a erros. Uma máquina que seja líder

de um determinado número de tópicos, ficará responsável por controlar os processos de escrita e leitura nos mesmos. No entanto, se este líder falhar, uma outra máquina do *cluster* irá automaticamente assumir a responsabilidade pelos tópicos da plataforma que falhou, tornando-se assim o novo líder [24].

Um consumidor deve guardar o *offset* da última mensagem que leu, referente a uma partição de um determinado tópico. Desta forma o consumidor pode efetuar leituras sequenciais, procurando pela mensagem com o *offset* imediatamente a seguir ao da última leitura. Como guarda sempre o último *offset*, o consumidor pode ler a informação na ordem que entender (desde que o identificador, pela qual pretende procurar, exista) [26].

2.1.4.2 Configuração e Características de Mensagens

Noutros *Message Brokers* as mensagens são guardas nas filas ou por consumo ou por tempo configurável. Kafka apenas oferece a configuração por tempo. A definição deste tempo é realizada por tópico e extensível às partições que lhe pertencem [26].

O tamanho máximo das mensagens depende muito do *hardware* e recursos do servidor em que corre o *cluster*. É necessário que a máquina contenha memória virtual suficiente para suportar tanto os programas em funcionamento como as mensagens que lhe vão chegando [26].

2.1.4.3 Síntese

O Apache Kafka é um pouco diferente da grande maioria dos *Message Brokers*, porém atraí bastantes clientes e de grande nome, como LinkedIn, Netflix, Spotify e Uber [27].

Este *Message Broker* esta sob a licença Apache License, uma licença para software livre de autoria da Apache Software Foundation e pode ser utilizada em diversas plataformas e linguagens (C, C++, C#, JAVA, PHP, Python, Node.js) [28].

2.1.5 IBM WebSphere MQ

WebSphere MQ [29], desenvolvido pela IBM, é mais um *Message Broker* disponível no mercado. Desenvolvida com vista a resolver problemas de comunicação empresariais, tenta garantir a entrega fiável de mensagens baseadas no formato XML, fornecendo a possibilidade de ligações entre diferentes componentes.

2.1.5.1 Versões do WebSphere Application Server

O WebSphere Application Server é um *runtime environment* para aplicações e suporte para transações. Providencia infraestruturas para ambientes de um único servidor ou de grande escala. Promete funcionalidades de gestão de serviços, de segurança e controlo através de ferramentas administrativas.

Existem diferentes versões para diferentes tipos de implementação e modelos de negócio. Diferentes áreas de atividade têm necessidades distintas. As funcionalidades que cada modelo oferece podem despertar interesse de diferentes entidades [30]. Estes modelos são:

- *Community Edition* – Desenvolvido sobre tecnologia *opensource*. É o mais barato de todos os modelos. Fornece uma rápida configuração de um serviço. É, no entanto, o que suporta o menor volume de transações. Ideal para pequenas empresas [30].

- *Express Edition* – Desenvolvido sobre o core da tecnologia do WebSphere, mas que não implementa todas as suas funcionalidades. Tem um custo de aquisição reduzido e a configuração de um serviço é também bastante rápida. Esta edição está limitada para correr apenas em 2 CPU's. Ideal para empresas mais avanças e que o volume de transações não atinja números muito elevados [30].

- *WebSphereApplicationServer Base Edition* – Esta edição é muito semelhante à *Express*, mudando a licença que fornecem. A *Base Edition* fornece suporte para um número ilimitado de CPU's [30].

- *Network Deployment Edition* – Inclui as mesmas funcionalidades que a edição base, mas oferece mais algumas vantagens: suporte para grandes volumes de transações; escalabilidade; *clustering*; alta disponibilidade; maior tolerância a falhas; plataforma de administração. Direcionado para grandes empresas com transações de informação constantes [30].

- *z/OS Edition* – Inclui todas as funcionalidades e vantagens da edição *Network Deployment* e mais alguns benefícios: utiliza funcionalidade z/OS; beneficia de um System Z Cluster (Parallel Sysplex ®) que oferece 99% de disponibilidade do sistema. Embora seja o mais completo, é o que tem maior custo de aquisição [30].

Nem todas as funcionalidades estão presentes em todas as plataformas. Existem requisitos de hardware para certas funcionalidades. Isto pode gerar um acréscimo de valor monetário para a implementação do sistema [30].

O WebSphere Application Server inclui de raiz um *Message Provider* robusto e estável que abrange modelos *Point-to-Point*, *Publish/Subscribe* e os seus diferentes modelos de subscrição. Este *Message Provider* é mais uma componente de gestão de mensagens da IBM a par do já existente, o WebSphere MQ. O WebSphere MQ continua a ser o *Message Broker* sugerido pela IBM e o que está há mais tempo no mercado, no entanto as funcionalidades oferecidas pelo WebSphere Application Server em conjunto com o seu *Message Provider* conjugam o melhor dos dois mundos [30].

2.1.5.2 Queue Managers

As filas de mensagens são geridas por *Queue Managers*. Podem existir múltiplos *Queue Managers* num único servidor físico. Para trocas de conhecimento são criados canais com estes gestores que devem ficar alerta dos mesmos. Se um *Queue Manager* verificar que uma mensagem flui por um destes canais tem a responsabilidade de a capturar [31]. Em conjunto com a vigia das ligações, o gestor fica a cargo de várias tarefas:

- Conservar as mensagens rececionadas até estas serem reencaminhadas ou processadas [31].
- Gerenciar a troca de mensagens entre componentes. Efetua a receção das mensagens dos produtores e tratará do reencaminhamento para os consumidores [31].
- Executar ações quando receciona mensagens especificas. À semelhança de eventos, é possível pedir ao gestor para efetuar certas ações através da definição de campos nos *meta dados* das mensagens [31].
- Converter as mensagens para um determinado formato, se necessário. Se um consumidor solicitar apenas um determinado formato, o gestor deve ficar incumbido de verificar e converter para o tipo correto [31].

2.1.5.3 Características e Formato de Mensagens

As mensagens têm, para além do seu corpo, um formato com alguns atributos e *meta dados* [29]:

- *MessageId* – Identificador da mensagem.
- *Format* – Construção da mensagem.
- *Topic* – Se o sistema for um modelo *Publish/Subscribe*, este campo poderá ajudar o *Message Broker* a perceber para onde a redirecionar.
- *CorrelId* – Quando uma mensagem é uma resposta a outra, este campo permite perceber a que mensagem está a responder (através do identificador da primeira mensagem).

2.1.5.4 Síntese

A robustez, fiabilidade, segurança, API simplista, boa documentação, e suporte continuo dão ao WebSphere MQ uma boa reputação no mercado. No entanto, a compatibilidade com plataformas não visadas durante o desenvolvimento desta *framework* pode levar intervenções futuras e dificuldades na atualização para novas versões [31].

Este *Message Broker* tem custos monetários que variam consoante o módulo e as funcionalidades pretendidas. É multiplataforma, e tem suporte para um conjunto especifico de linguagens (C, Visual Basic, C++, JAVA, ActiveX) [29].

2.1.6 Resumo e comparação de funcionalidades

Embora tenham o mesmo objetivo, os diferentes *Message Brokers* oferecem distintas funcionalidades. A principal diferença está na área de negócio a que se destinam. Uma tabela comparativa das características pode ajudar a escolher a tecnologia mais adequada para um determinado produto:

	ZeroMQ	RabbitMQ	MSMQ	Apache Kafka	WebSphere MQ
Free-to-Use	✓	✓	✓	✓	
Poliglota (Linguagens Programação)	✓	✓		✓	
Multiplataforma	✓	✓			✓
Point-to-Point	✓	✓	✓	✓	✓
Publish/Subscribe	✓	✓		✓	✓
Mensagens com Prioridade		✓	✓		✓
Rotas por Tipo de Mensagem	✓	✓		✓	✓
Rotas por Expressões Regulares		✓			✓
Alto Nível Configuração	✓	✓		✓	✓
Independência de *Broker*	✓				
Comunidade de Suporte	✓	✓		✓	
Interface Gráfica de Administração		✓	✓	*3rd Party*	Dependente Modelo
Tamanho Máximo Mensagem	Teórico (2^{64} Bytes)	Teórico (2^{64} Bytes)	4MB	Dependente Hardware	Dependente Modelo

Tabela 1: Comparação de Funcionalidades de alguns Message Brokers

A comparação de funcionalidades não deve ser o único fator de decisão. A performance de cada um e a carga que aguentam devem ser elementos a ter em conta. No âmbito deste trabalho, será feita uma escolha inicial de, no mínimo, dois *Message Broker*. A base desta escolha recairá nas características que são irrefutavelmente necessárias para a implementação no produto. No entanto, após a primeira triagem, serão feitas analises ao nível da performance de cada um e assim chegar a uma decisão final.

2.2 Métricas de medição de recursos

Métricas recolhem dados de um sistema, num especifico espaço de tempo. Nos ambientes computacionais, principalmente em servidores, estes dados podem ser analisados em intervalos que variam de segundos para minutos, ou até de hora em hora. Análogo a este ambiente, é possível também dividir as métricas em duas categorias: *Work Metrics* e *Resource Metrics* [32].

2.2.1 Work Metrics

Este tipo de métricas medem a robustez do sistema. Esta categoria torna-se, também, mais fácil de dividir em subtipos [32]:

- *Throughput* - Rendimento – A quantidade de trabalho que o sistema consegue efetuar numa unidade de tempo.
    - Exemplo: Pedidos por segundo – 312.
- *Success* – Taxa de Sucesso – Rácio de tarefas que foram efetuadas com sucesso, no conjunto de todas realizadas.
    - Exemplo: Percentagem de respostas positivas – 87%.
- *Error* - Taxa de Erros – Número de resultados incorretos por unidade de tempo. É normalmente isolada da taxa de sucesso, devido à grande diversidade de origens de erro.
    - Exemplo: Percentagem de respostas negativas – 0.9%.
- *Performance* – Eficiência de Trabalho – Quantificador de quão eficiente é o sistema num determinado instante. O método mais comum é a medição de latência, que representa o tempo necessário para uma tarefa ser concluída.
    - Exemplo: Latência de resposta – 10ms.

2.2.2 Resource Metrics

Um servidor é um grupo de componentes físicos (i.e., *Hardware*) que trabalham em conjunto para efetuar as tarefas que lhe são requisitadas. Dependendo das tarefas, a maior ou menor utilização de certos elementos pode variar. Usualmente, os principais componentes são: o CPU, a memória física, a memória virtual e as interfaces de rede. Pode também ser divida em diferentes subtipos [32]:

- *Utilization* – Taxa de utilização – Percentagem de tempo que um componente esteve a ser utilizado ou a percentagem da capacidade total que está a ser usada.
    - Exemplo: CPU – 75% de utilização da capacidade total do processador.
      Memória – 45% de utilização da memória total.
- *Saturation* – Volume de pedidos – Quantidade de pedidos que foram efetuados, mas que ainda não foram realizados. Normalmente colocam-se numa fila de espera.
- *Errors* - Erros de componente – Quantidade de erros que são causados a nível físico e não diretamente por problemas de software.
- *Availability* – Taxa de disponibilidade – Rácio do número de vezes que os pedidos foram acedidos, no conjunto de todos os efetuados.

Este tipo de métricas poderão ser uma ajuda valiosa no diagnóstico de problemas, disponibilizando uma variedade de valores. Torna-se mais fácil perceber quais os sistemas com maior sobrecarga ou os que têm problemas de funcionamento [32].

Capítulo 3

Análise de Desempenho

No capítulo anterior foi possível verificar as características que são inerentes aos diversos Message Brokers, verificando-se apenas algumas diferenças ao nível de funcionalidades e comportamento.

Em conjunto com responsáveis da MOG Technologies, foi feita uma análise geral do que cada tecnologia oferece, reduzindo a escolha para dois candidatos: **RabbitMQ** e **ZeroMQ.**

Estes dados foram, porém, insuficientes para tomar uma decisão definitiva relativamente à escolha de uma única tecnologia. O tamanho e a carga do sistema, em que um Message Broker vai ser implementado, deve ser um dos elementos em consideração e, como tal, é indispensável verificar como cada uma das frameworks se comporta na categoria de desempenho.

Diversos casos de teste podem ser ponderados, contudo deseja-se um sistema com foco unicamente na partilha de mensagens de forma a fornecer resultados mais íntegros. Estes resultados ajudarão a determinar qual a melhor solução para o sistema, podendo conjugar assim funcionalidade com desempenho.

3.1 Simulador Sistema de Mensagens

Este simulador foi desenvolvido tendo em conta a arquitetura do modelo *Publish/Subscribe*.

Para iniciar a propagação mensagens e testar os seus resultados, foi criado um gestor que recebe um conjunto de argumentos especificados pelo utilizador. Este gestor permite colocar em execução um número de *Publishers* e *Subscribers*, com características configuráveis, podendo ambos ser clientes de RabbitMQ ou ZeroMQ**.**

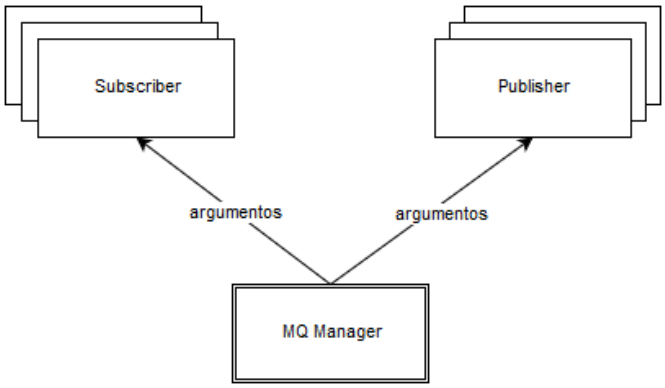

Figura 13: Arquitetura simplificada do simulador de Sistema de Mensagens

As mensagens podem ser enviadas de 2 maneiras:

- Enviar mensagens por tempo. Um *Publisher* envia mensagens, com ou sem intervalo entre as mesmas, durante um espaço temporal definido.

- Enviar mensagens por quantidade. Um *Publisher* envia apenas N mensagens, com ou sem intervalo entre as mesmas.

Outra característica das mensagens que pode também se definida, é o seu tamanho. É possível passar o caminho de um ficheiro que será enviado como mensagem. Isto verifica-se como uma ajuda essencial na verificação e comparação de mensagens. Se a mensagem que um *Subscriber* receber não for exatamente idêntica á que foi enviada, é considerado como uma mensagem falhada.

Um aspeto a ter conta é o facto de, em RabbitMQ, ser necessária a existência de um servidor para que os clientes se possam conectar. Este servidor assume o papel de Message Broker que filtrará, guardará e reencaminhará as mensagens para os seus destinatários. Tanto os *Publishers* como os *Subscribers* precisarão de ser configurados para comunicarem com um servidor de RabbitMQ na rede. A única informação necessária para a conexão é o endereço da máquina que executa o Message Broker.

Em ZeroMQ a situação é um pouco diferente devido à não existência de um Message Broker como nas restantes tecnologias. Embora não exista um mediador de conexão de raiz, é possível implementar um modelo *Publish/Subscribe* criando um cliente intermédio que assumirá ambos os papeis, *Publisher* e *Subscriber*, em simultâneo. Todos os outros clientes terão de se "subscrever" para darem conhecimento da sua existência [11]. Esta subscrição é feita tendo conhecimento do endereço da máquina onde o intermediário está a ser executado.

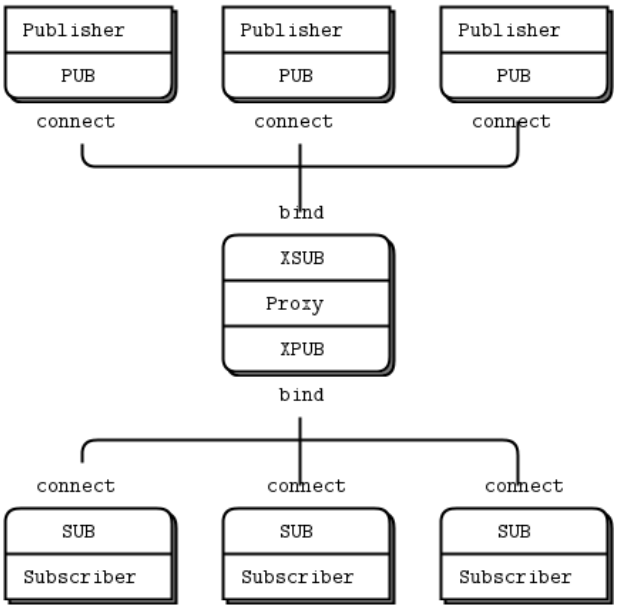

Figura 14: Modelo Publish/Subscribe em ZeroMQ com Intermediário [11]

A razão por trás desta decisão foca-se no variável tamanho das redes e do variável número de clientes em simultâneo. Para redes com uma complexidade simples ou mediana, ter um gestor intermédio poderá trazer benefícios sem prejudicar o seu desempenho. No entanto, em redes de complexidade elevada, os Message Brokers podem adquirir dificuldades de gestão, produzindo um enorme número de verificações e decisões que, eventualmente, criarão problemas [11].

Desta forma, a tecnologia ZeroMQ deixa ao encargo dos arquitetos de sistemas a decisão de ter um Message Broker. É relevante também lembrar que este gestor intermédio não contém qualquer funcionalidade, deve ser desenvolvido de raiz e aprimorado de acordo com as exigências necessárias.

Neste simulador o mesmo foi feito para ZeroMQ. Um gestor intermédio foi criado para se conseguir ter um modelo Publish/Subscribe e assim comparar diretamente com RabbitMQ nos mesmos moldes.

3.2 Testes e Resultados

Todos os testes foram realizados utilizando o simulador no ponto anterior mencionado. Estes ensaios têm como principais objetivos a comparação de velocidade de transmissão/receção e o nível de confiança na entrega de mensagens.

Cada conjunto de condições de teste foi executado em múltiplas instancias, utilizando as duas tecnologias: RabbitMQ e ZeroMQ.

3.2.1 Sistema de Testes

Para uma melhor comparação, todos os testes foram executados na mesma máquina utilizando o mesmo poder de processamento, capacidade de rede e memória. As especificações da máquina da qual se obtiveram os resultados são:

- *CPU:* Intel Core i7 4790 3.60GHz
- *RAM:* Kingston 8GB DDR3 1600MHz
- *HDD*: 250GB
- *Network Speed*: 200Mbps Download / 20Mbps Upload

Os resultados apresentados são específicos para estas condições, podendo variar de acordo com o ambiente em que são executados.

É também importante notar que, tanto os *Publishers* como os *Subscribers* utilizados nos testes, foram executados na mesma máquina tendo que partilhar os recursos entre si.

3.2.2 Teste 1

Este teste foi realizado utilizando um *Publisher* e um *Subscriber*. O *Publisher* ficou encarregue de enviar mensagens durante 60 segundos sem intervalo entre as mesmas. Isto significa que enviou o máximo de mensagens que conseguiu durante o tempo especificado. O tamanho da mensagem era de 5 Bytes.

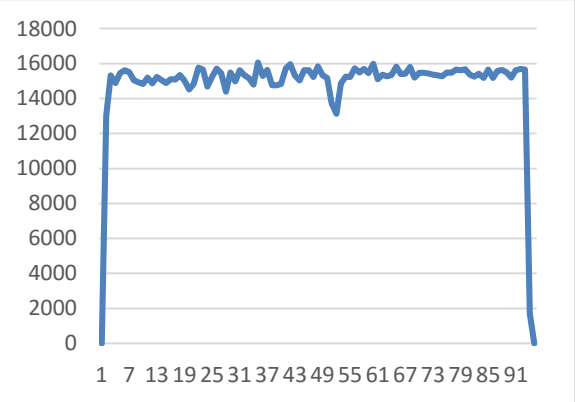

Figura 15: RabbitMQ - Média Mensagens/Segundo Teste 1

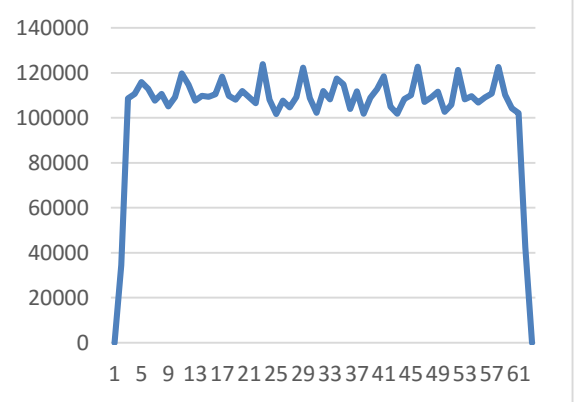

Figura 16: ZeroMQ - Média Mensagens/Segundo Teste 1

	RabbitMQ	ZeroMQ
Média Mensagens Enviadas	1 424 032	6 668 300
Média Mensagens Recebidas	1 424 032	6 637 488
Média Mensagens Perdidas	0	30 811
Média Mensagens/Segundo	15 165 msg/s	108 683 msg/s
Média Tempo Receção	95,8s	60s

Tabela 2: Resultados Teste 1

Após analisar os resultados podemos verificar que, nestas condições, a tecnologia ZeroMQ foi capaz de enviar mais mensagens durante os 60 segundos propostos. No entanto, esta tecnologia perdeu 30811 mensagens, o que não acontece em RabbitMQ.

Este problema poderá advir de um dos intervenientes não conseguir acompanhar a quantidade de mensagens enviadas pelo *Publisher*. Quando um recetor não consegue processar as mensagens que recebe, a um ritmo superior à taxa de chegada, começa a encher a fila de mensagens do seu ZeroMQ *socket* até que passe o seu limite máximo. Este limite, denominado de *High Water Mark*, pode ser definido à priori. No entanto, não é possível saber quantas mensagens estão em espera na fila. Esta informação poderia ser útil para aumentar o limite dinamicamente conforme a necessidade.

Neste teste, o *Publisher* ficava "em espera" sempre que a fila se encontrava cheia e só voltaria a enviar a mensagem quando um novo espaço fosse libertado, logo o problema não estaria no envio, mas sim na receção.

O modelo Publish/Subscribe, de ZeroMQ, baseia-se no envio de mensagens multicast entre o intermediário e o recetor. No entanto, ZeroMQ não consegue garantir total fiabilidade em comunicação multicast existindo um conjunto de situações que levam à perda de mensagens [11]:

- Um *Subscriber* pode perder mensagens se for um "*Slow Starter*". Esta situação acontece quando a conexão com um gestor intermédio leva mais tempo que o normal, fazendo com que todas as mensagens enviadas nesse espaço de tempo sejam perdidas [11].
- Quando acontece uma falha na rede [11].
- Quando um *Subscriber* ou uma rede não consegue acompanhar a taxa de envio de um *Publisher* [11].

Neste teste foi possível verificar o último ponto. A taxa de mensagens entre o cliente intermédio e o Subscriber era superior à taxa de processamento deste recetor. Isto fez com a fila atingisse o seu limite e começasse a perder mensagens.

3.2.3 Teste 2

Este teste foi realizado utilizando um *Publisher* e um *Subscriber*. O *Publisher* ficou encarregue de enviar mensagens durante 60 segundos com um intervalo de 100 milissegundos entre cada envio. O tamanho da mensagem era de 5 Bytes.

Este teste serve para verificar como se comportam as duas tecnologias numa situação mais simples, com cerca de 10 mensagens por segundo, sendo um pouco diferente do primeiro teste onde se pretendia averiguar qual a taxa máxima de envio os *Publishers* de ambas as tecnologias conseguiam suportar.

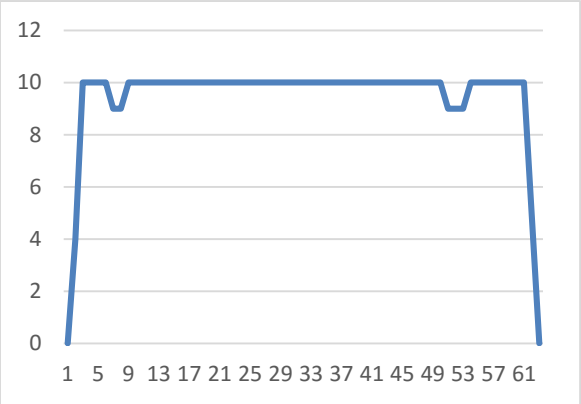

Figura 17: RabbitMQ - Média Mensagens/Segundo Teste 2

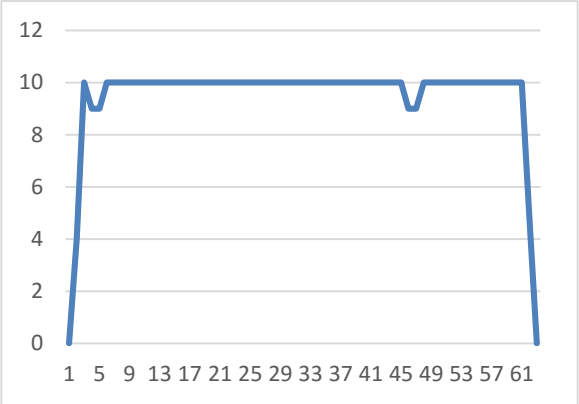

Figura 18: ZeroMQ - Média Mensagens/Segundo Teste 2

	RabbitMQ	ZeroMQ
Média Mensagens Enviadas	597	597
Média Mensagens Recebidas	597	597
Média Mensagens Perdidas	0	0
Média Mensagens/Segundo	9,75 msg/s	9,76 msg/s
Média Tempo Receção	60s	60s

Tabela 3: Resultados Teste 2

De acordo com os resultados, se intervalarmos as mensagens em 100 milissegundos, RabbitMQ e ZeroMQ demonstram o mesmo desempenho. Neste teste já não se verifica perda de mensagens em ZeroMQ como no primeiro teste. Deve-se principalmente ao facto de, ao intervalar as mensagens, o emissor conseguir ter uma taxa de processamento superior à taxa de envio do emissor.

3.2.4 Teste 3

O terceiro teste foi também realizado utilizando um *Publisher* e um *Subscriber*. O *Publisher* ficou encarregue de enviar 5000 mensagens sem intervalo entre cada envio. Isto significa que o emissor enviou as mensagens especificadas o mais rápido que conseguisse. O tamanho da mensagem era de 10 MegaBytes, tornando este teste bastante mais "pesado" que os anteriores.

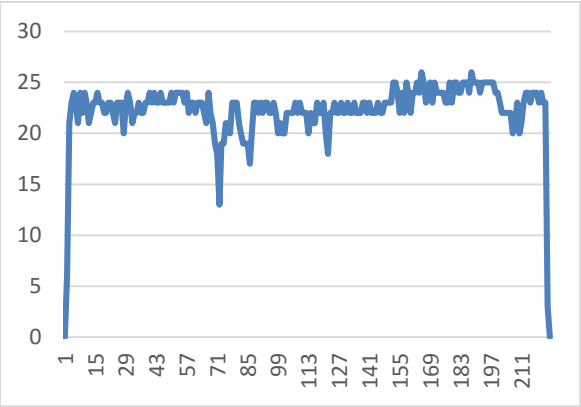

Figura 19: RabbitMQ - Média Mensagens/Segundo Teste 3

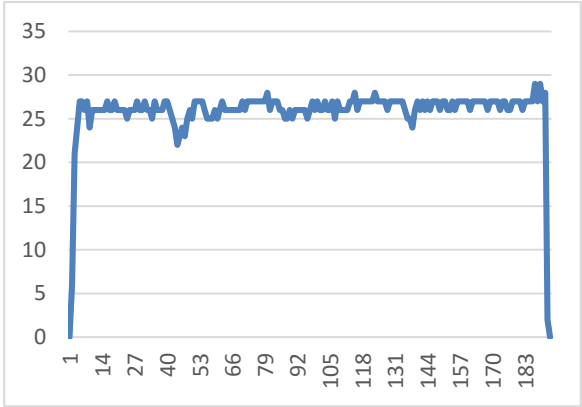

Figura 20: ZeroMQ - Média Mensagens/Segundo Teste 3

	RabbitMQ	ZeroMQ
Média Mensagens Enviadas	5 000	5 000
Média Mensagens Recebidas	5 000	5 000
Média Mensagens Perdidas	0	0
Média Mensagens/Segundo	23,22 msg/s	26,10 msg/s
Média Tempo Receção	216,5s	190,5s

Tabela 4: Resultados Teste 3

Analisando os resultados, podemos assumir que ambas as tecnologias conseguiram passar com sucesso a barreira dos 10 MegaBytes por mensagem. O desempenho foi bastante semelhante, verificando-se uma ligeira vantagem em ZeroMQ.

3.2.5 Teste 4

No quarto teste a carga de mensagens aumentou, elevando o número de *Publishers* para 10. No entanto, manteve-se um único *Subscriber*. Cada *Publisher* ficou encarregue de enviar um milhão de mensagens, sem intervalo entre cada envio, totalizando 10 milhões de envios. O tamanho da mensagem era de 5 Bytes.

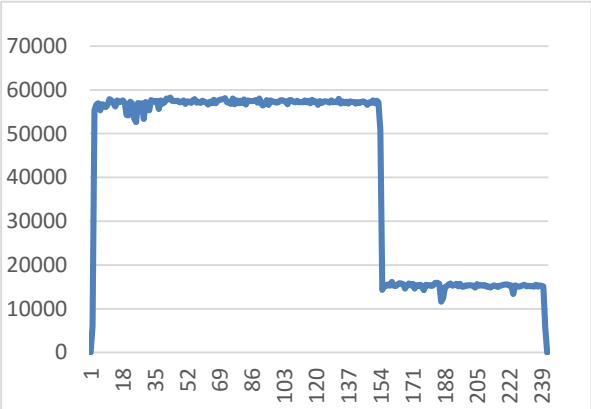

Figura 21: RabbitMQ - Média Mensagens/Segundo Teste 4

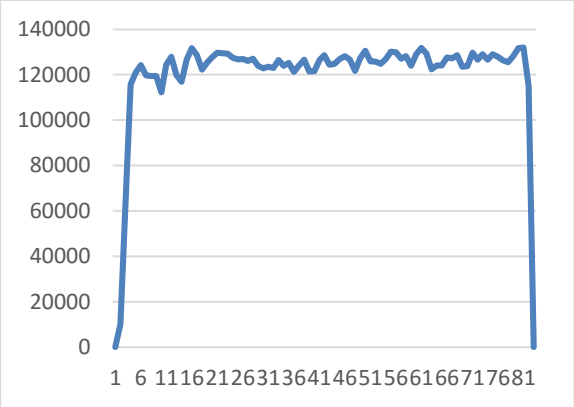

Figura 22: ZeroMQ - Média Mensagens/Segundo Teste 4

	RabbitMQ	ZeroMQ
Média Mensagens Enviadas	10 000 000	10 000 000
Média Mensagens Recebidas	10 000 000	9 988 120
Média Mensagens Perdidas	0	11 880
Média Mensagens/Segundo	41 624 msg/s	123 341 msg/s
Média Tempo Receção	243s	81s

Tabela 5: Resultados Teste 4

Na **Figura 21**, é possível verificar um declínio da média de mensagens por segundo. Este declínio ocorre devido a alguns *Publishers* terminarem primeiro o envio das suas mensagens. Os restantes continuam a emitir até que atinjam o número de envios especificado.

Mais uma vez, ZeroMQ conseguiu ter melhor desempenho na taxa de receção e envio, mas voltou-se a verificar perda de mensagens. As conclusões sobre esta omissão de mensagens são as mesmas que foram retiradas no primeiro teste.

3.2.6 Teste 5

Este teste assemelha-se muito ao quarto teste, com os mesmos 10 *Publishers* para apenas um único *Subscriber*. As alterações que se verificam estão na redução do número de mensagens que cada *Publisher* tem que emitir, passando de um milhão para 5 mil, e no tamanho da mensagem que passa agora para 1 MegaByte, aumentando significativamente a carga no sistema.

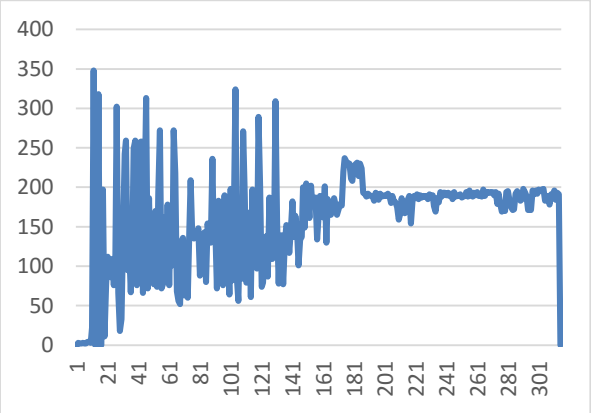

Figura 23: RabbitMQ - Média Mensagens/Segundo Teste 5

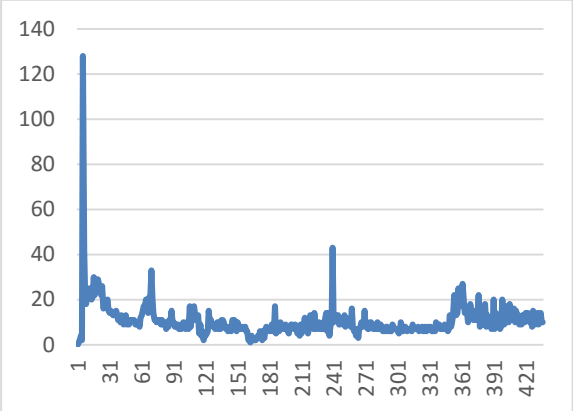

Figura 24: ZeroMQ - Média Mensagens/Segundo Teste 5

	RabbitMQ	ZeroMQ
Média Mensagens Enviadas	50 000	?
Média Mensagens Recebidas	50 000	9 848
Média Mensagens Perdidas	0	?
Média Mensagens/Segundo	144,51 msg/s	10,63 msg/s
Média Tempo Receção	352s	683s

Tabela 6: Resultados Teste 5

Os resultados deste teste demonstram diferentes situações. Em RabbitMQ, enquanto todos os *Publishers* emitiam em simultâneo, existiu uma grande variância na média de mensagens que eram recebidas, como demonstra a **Figura 23**. Quando alguns *Publishers* terminaram o seu envio, verificou-se uma maior estabilização. Também em RabbitMQ, todas as mensagens chegaram ao seu destinatário. Este teste demorou significativamente mais que os anteriores, mas era de esperar tendo em conta o tamanho e número de mensagens a entrar num único recetor.

Em ZeroMQ o caso foi bastante diferente. Durante o teste rapidamente se começaram a ver problemas de memória que levaram ao encerramento inesperado da máquina. Como mencionado anteriormente, o gestor intermédio tem duas filas de espera, uma para receber e outra para enviar. Quando esse manager começa a ficar sobrecarregado, as suas filas começam a encher. Uma

mensagem só sai da fila de receção para a fila de envio quando houver espaço e esta estiver sido totalmente replicada.

Os *Publishers* vão enviado as mensagens para a sua fila de espera, ficando suspensos apenas quando não houver mais espaço. Em ZeroMQ, o *High Water Mark*, é definido para o número de mensagens na fila, não importando o seu tamanho. Se um Subscriber não conseguir processar rapidamente as mensagens que lhe são destinadas, vai gerar um aumento de dados nas filas de espera do gestor intermédio e, por consequente, aumentar o número de mensagens que os *Publishers* guardam para posterior envio. Como as mensagens eram de um tamanho considerável e tendo 10 *Publishers* a conserva-las em memória, rapidamente se conseguiram esgotar os recursos da máquina.

3.2.7 Teste 6

O sexto teste serviu para retirar mais conclusões sobre o teste anterior. Mantiveram-se os mesmos 10 *Publishers*, o único *Subscriber*, o tamanho de 1 MegaByte e as 50 mil mensagens que se pretendia enviar. A única diferença é que agora as mensagens são intervaladas em 100 milissegundos, tentando fornecer mais tempo ao *Subscriber* para processar as mensagens que lhe chegam.

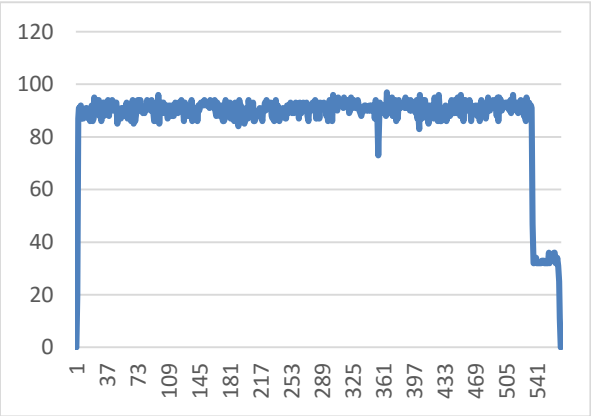

Figura 25: RabbitMQ - Média Mensagens/Segundo Teste 6

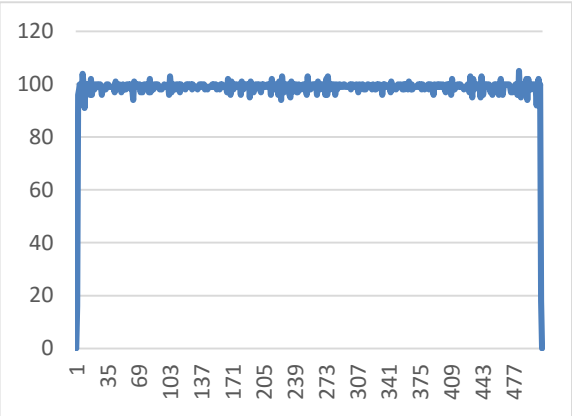

Figura 26: ZeroMQ - Média Mensagens/Segundo Teste 6

	RabbitMQ	ZeroMQ
Média Mensagens Enviadas	50 000	50 000
Média Mensagens Recebidas	50 000	50 000
Média Mensagens Perdidas	0	0
Média Mensagens/Segundo	87,16 msg/s	98,68 msg/s
Média Tempo Receção	580s	506s

Tabela 7: Resultados Teste 6

Devido à diminuição da taxa de envio, a velocidade de processamento das mensagens por parte do *Subscriber* era superior à taxa de chegada. Desta forma, em ZeroMQ já não se gerou acumulação de mensagens nas filas de espera do gestor intermédio e dos *Publishers*.

Os resultados são bastante semelhantes entre as duas tecnologias, tendo o ZeroMQ uma pequena vantagem pois conseguiu concluir o teste mais rapidamente.

3.2.8 Teste 7

Este teste é também uma continuação do sexto teste. Mantiveram-se os mesmos 10 *Publishers*, o único *Subscriber*, o intervalo de 100 milissegundos e as 50 mil mensagens que se pretendia enviar. A única alteração está no tamanho das mensagens, que agora são de 10 MegaBytes.

Inicialmente este teste, na versão ZeroMQ, foi executado com o limite padrão das filas de espera. No entanto, mesmo com o intervalo de 100 milissegundos, voltou-se a verificar problemas de memória. Então, para se conseguir concluir o teste, diminuísse o limite de todos os intervenientes para um valor que não ultrapassasse os recursos da máquina, mesmo que as filas ficassem completamente sobrecarregadas.

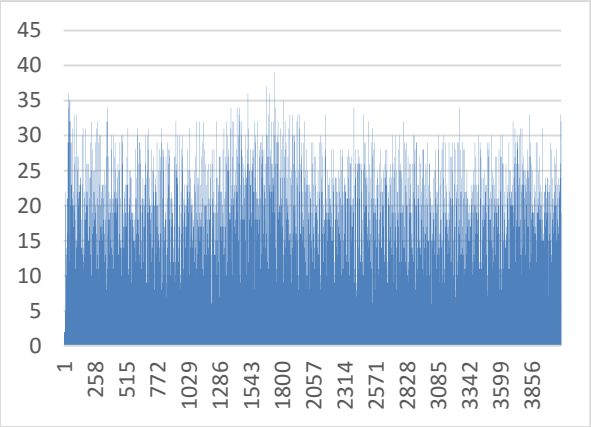

Figura 27: RabbitMQ - Média Mensagens/Segundo Teste 7

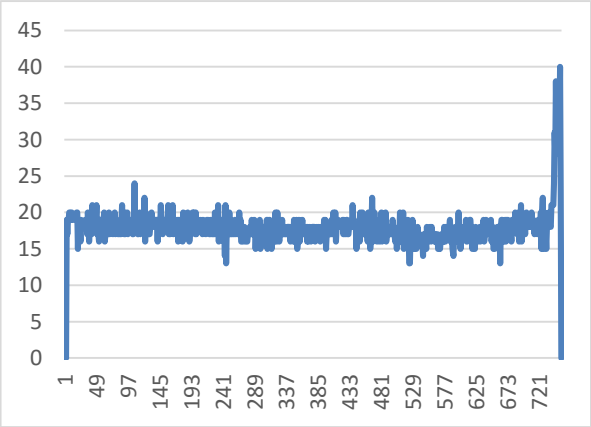

Figura 28: ZeroMQ - Média Mensagens/Segundo Teste 7

	RabbitMQ	ZeroMQ
Média Mensagens Enviadas	50 000	50 000
Média Mensagens Recebidas	50 000	13 545
Média Mensagens Perdidas	0	36 454
Média Mensagens/Segundo	12,28 msg/s	18,14 msg/s
Média Tempo Receção	3712s	741s

Tabela 8: Resultados Teste 7

Em RabbitMQ tudo correu como esperado. Todas as mensagens chegaram ao seu destinatário, embora tenha demorado mais tempo. Este decréscimo de tempo, em relação aos outros testes, é compreensível tendo em conta o tamanho das mensagens.

Em ZeroMQ, visto termos diminuído o limite máximo das filas de espera e aumentado o tamanho e carga das mensagens, voltou-se a verificar perda de mensagens. A causa é a mesma que foi mencionada na conclusão do primeiro teste, **3.2.2**.

3.2.9 Teste 8

No oitavo teste aumentou-se igualmente o número de *Subscribers*, ficando agora um conjunto de 10 para 10. Os 10 *Publishers* enviam 50 mil mensagens, intervaladas por 100 milissegundos e com tamanho de 5 Bytes.

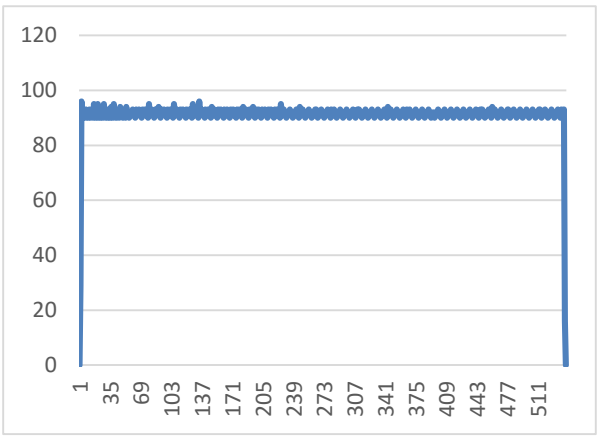

Figura 29: RabbitMQ - Média Mensagens/Segundo Teste 8

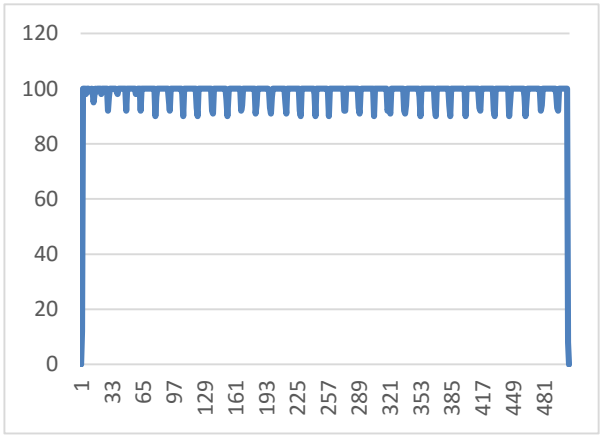

Figura 30: ZeroMQ - Média Mensagens/Segundo Teste 8

	RabbitMQ	ZeroMQ
Média Mensagens Enviadas	50 000	50 000
Média Mensagens Recebidas	50 000	50 000
Média Mensagens Perdidas	0	0
Média Mensagens/Segundo	91,24 msg/s	98,79 msg/s
Média Tempo Receção	548s	506s

Tabela 9: Resultados Teste 8

Este teste demonstrou que ambas as tecnologias conseguiram suportar a carga de 10 *Publishers* e 10 *Subscribers* em simultâneo. Os resultados são bastante semelhantes, com uma ligeira vantagem para ZeroMQ.

3.2.10 Teste 9

Este teste é bastante semelhante ao oitavo teste. Mantiveram-se os 10 *Publishers*, os 10 *Subscribers*, a quantidade de 50 mil mensagens e o intervalo de 100 milissegundos. A única diferença encontra-se no tamanho das mensagens que agora é de 1 MegaByte.

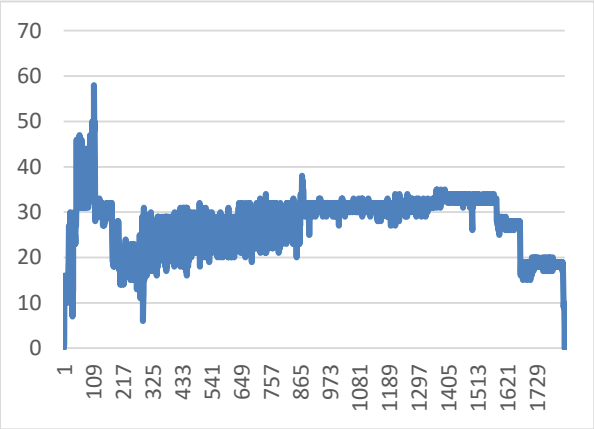

Figura 31: RabbitMQ - Média Mensagens/Segundo Teste 9

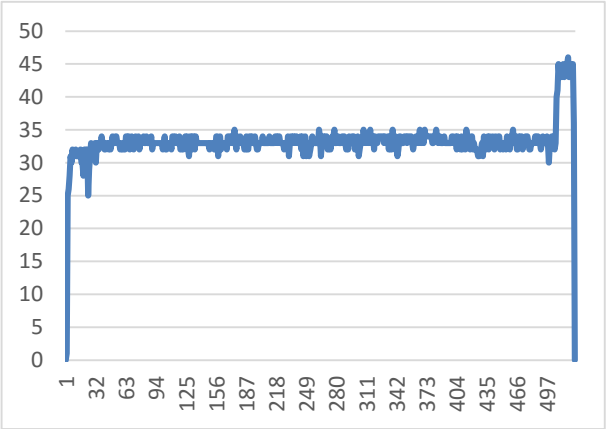

Figura 32: ZeroMQ - Média Mensagens/Segundo Teste 9

	RabbitMQ	ZeroMQ
Média Mensagens Enviadas	50 000	50 000
Média Mensagens Recebidas	50 000	17 451
Média Mensagens Perdidas	0	32 549
Média Mensagens/Segundo	27,30 msg/s	33,26 msg/s
Média Tempo Receção	1856s	524s

Tabela 10: Resultados Teste 9

Analisando os resultados, foi possível verificar uma nova situação para RabbitMQ. Cada *Subscriber* tem a sua própria fila de espera. Como tal, tendo em conta que eram 10 recetores, o número de mensagens em fila de espera decuplicou. Isto levou a que o servidor de RabbitMQ atingisse um limite máximo de memória e começasse a escrever as mensagens em disco rígido. Esta decisão piorou o seu desempenho, mas conseguiu manter o sistema em execução e entregar todas as mensagens. Este declínio de desempenho é visível na **Figura 31**.

Em ZeroMQ, voltámos a perder mensagens. O motivo é o mesmo que foi mencionado na conclusão do primeiro teste, **3.2.2**.

3.3 Conclusão do Capítulo

Os testes permitiram ter melhor perceção do desempenho de ambas as tecnologias, RabbitMQ e ZeroMQ. Foi possível verificar que ZeroMQ consegue ter melhor desempenho na velocidade de envio e receção. No entanto, a perda de mensagens e a ausência de garantias de entrega, no modelo Publish/Subscribe, geram uma preocupação para quem deseja uma arquitetura robusta e competente. À parte deste problema, sistemas pequenos e com poucos utilizadores em simultâneo podem continuar a assumir ZeroMQ, em Publish/Subscribe, como bom candidato. Outra questão é falta de configuração e ausência de informação que ZeroMQ demonstra em alguns casos, como a quantidade de mensagens em fila ou o nível de utilização das filas dos gestores intermédios.

RabbitMQ, embora nalguns testes apresente um inferior desempenho, tem uma boa performance sempre conseguindo garantir a entrega das mensagens. O seu alto nível de configuração oferece aos desenvolvedores um conjunto de opções para criar a melhor solução. A possibilidade de clientes em várias linguagens, um conjunto de protocolos nativo (AMQP, STOMP e MQTT), os tempos de vida das mensagens, os tempos de vida das filas de espera, a persistência em casos de sistemas interrompidos, a sua plataforma de gestão, a filtragem por expressões regulares, o seu bom desempenho e um Publish/Subscribe fiável são algumas das principais características que tornam esta tecnologia tão popular.

Depois de apresentadas as vantagens e desvantagens de cada um, em parceria com os responsáveis da MOG Technologies, foi tomada uma decisão bastante fácil e breve em torno de RabbitMQ. A carga dos testes em que RabbitMQ e ZeroMQ ficaram par a par a nível de desempenho, foram os mais aproximados com o que se espera ser a realidade do produto. Assim, tendo em conta que o desempenho seria bastante aproximado, a fiabilidade, a facilidade de escalar e moldar RabbitMQ tornou-o o principal candidato.

Capítulo 4

Especificação e Desenvolvimento

O produto mxfSPEEDRAIL já conta com uma implementação de um sistema de partilha de eventos, desenvolvido e aprimorado pelos próprios engenheiros da MOG Technologies. Esta implementação foi denominada de **EventHub**. O EventHub foi desenvolvido em C# e efetua a partilha de mensagens através de sockets TCP.

O objetivo era, em conjunto com a já existente e bem definida interface, encontrar uma forma de alterar o atual estilo de transmissão por um sistema de *Message Queues*, como é o RabbitMQ. No entanto, um dos requisitos principais sempre foi a menor alteração possível ao nível da arquitetura existente. Esta decisão deve-se às imensas dependências que o EventHub possui, impedindo grandes alterações que, por consequência, impulsionariam um conjunto de modificações inevitáveis por todo o produto.

Uma análise inicial à arquitetura e comportamento do EventHub, torna-se essencial para se perceber se é possível introduzir um sistema de filas de mensagens sem grandes efeitos negativos. Este estudo permite também conceber uma solução pensada e analisada, antes da implementação em si.

4.1 EventHub

O EventHub pode ser visto como um gestor de mensagens que conta com um conjunto de elementos. Estes elementos são os responsáveis pela emissão, receção e gestão de eventos. Desta forma, ao trabalharem em parceria, compõe todo o sistema de propagação de mensagens.

Não existe apenas um único EventHub, sendo que numa mesma instância do mxfSPEEDRAIL podem existir diversos EventHubs. É desta forma que as mensagens se propagam entre sistemas, através da comunicação entre dois EventHubs. Para tal comunicação

acontecer, um dos intervenientes tem de ter conhecimento da existência do outro e ter a informação necessária para efetuar uma ligação. Esta informação é, usualmente, o endereço da máquina onde se encontra o destinatário.

4.1.1 Listener

Um Listener é, essencialmente, um *Subscriber* que espera por eventos. Cada Listener tem uma tarefa especifica, pronta a executar quando um receber um evento. Os Listeners têm determinadas características:

- *Id* – Um identificador único que o distingue de qualquer outro elemento.
- *Ação* – Tarefa a executar quando receber uma mensagem.
- *Condições Filtragem* – Condições para filtrar as mensagens que chegam. Quando uma mensagem chega a um Listener, este tem de verificar se a mensagem lhe interessa ou lhe é destinada.

Quando uma mensagem é entregue a um Listener, uma verificação da entrega é efetuada. Primeiramente, verifica-se se alguma das condições de filtro do Listener corresponde ao cabeçalho da mensagem. Se corresponder executa a tarefa que lhe foi designada, caso contrario descarta a mensagem.

Existe uma pequena distinção entre Listeners, os locais e os remotos. Têm exatamente as mesmas características, no entanto diferem na sua propriedade. Um EventHub considera um Listener como local quando este foi criado por ele próprio. É considerado como remoto, um Listener que tenha sido criado por outrem e que só pode ser contactado remotamente.

4.1.2 Token

Um Token é um elemento que pode ser requisitado a um EventHub. Este providencia informação especifica sobre o EventHub que o gerou, assim como formas de este ser contactado. Um Token, é então requisitado para alguém exterior ganhar conhecimento sobre um determinado EventHub. Logo, é também criado um Listener para escutar mensagens que possivelmente possam chegar.

Um Token tem então certas características:

- *HubID* – O identificador do EventHub que gerou o Token.
- *ListenerID* – O identificador do Listener criado quando o Token foi requisitado.
- *EndpointsURIs* – Um conjunto de endereços que irão permitir ligações TCP e assim conseguir contactar o EventHub que gerou o Token.

Esta informação permite, ao fornecer os dados necessários, que um EventHub inicie uma conexão com outro EventHub.

4.1.3 HubProxy

Um HubProxy representa uma "conexão" entre dois EventHubs e ficará encarregue de executar todas as tarefas de comunicação entre estes dois intervenientes

Um EventHub, quando ganha conhecimento de outro EventHub (normalmente através de um Token), cria um HubProxy com toda a informação necessária para estabelecer contacto e envia uma mensagem de registo. Do outro lado, o EventHub ao rececionar uma mensagem de registo, cria automaticamente um HubProxy com todas as informações recebidas. Desta forma, os EventHubs ganham conhecimento de outros pontos na rede. Sempre que precisarem de comunicar com um elemento remoto, utilizam o HubProxy correspondente.

O HubProxy guarda uma lista de ligações TCP e endereços, que serão utilizados sempre que uma emissão ou receção seja necessária. Guarda também uma lista dos Listeners que o seu correspondente tem, ou seja, um conjunto de Listeners remotos.

4.1.3.1 HubProxy Actions

Quando um HubProxy é criado, são também definidas algumas ações que devem ser executadas em diferentes situações:

- Quando um evento é recebido;
- Quando é recebido um pedido de desconexão de um Listener.
- Quando é recebido um pedido de adição de um novo Listener.
- Quando se perde a conexão ao EventHub remoto.

Estas ações são executadas após se receber uma mensagem, analisando se o cabeçalho da mesma corresponde a alguma das ações.

4.1.4 EventsArgs

Antes de enviar uma mensagem, é preciso definir o seu conteúdo assim como informação extra que esclareça a sua origem, o seu tipo, entre outros. Um EventHub, só aceita emitir uma mensagem, quando recebe um objeto do tipo EventArgs bem definido. Este objeto tem como características:

- *SenderName* – O nome do emissor (normalmente o nome da classe de onde é gerada o pedido de emissão da mensagem).
- *SenderID* – O identificador único do emissor.
- *EventName* – O tipo de mensagem (se é um evento, um registo, uma desconexão, …).

- *Payload* – Um objeto XML que contém o conteúdo da mensagem.

As condições de filtro de um Listener baseiam-se nestes 3 campos: SenderName, SenderID e EventName.

4.1.5 HandleEvent

HandleEvent é uma tarefa, de todos os EventHubs, que é invocada sempre que se deseja emitir uma mensagem. Aceita apenas objetos do tipo EventArgs (**4.1.4**) e tem a responsabilidade de transmitir uma mensagem para todos os Listeners, tanto os remotos como os locais, que são do conhecimento do EventHub. É, no entanto, um pouco diferente a maneira como estes dois tipos de Listeners são processados.

Os Listeners locais, como pertencem ao mesmo EventHub que vai realizar um HandleEvent, não necessitam de comunicação em rede para receber mensagens, sendo estas passadas diretamente ao objeto no formato EventArgs.

Já os Listeners remotos, para receber uma mensagem proveniente de um EventHub externo, necessitam de comunicação em rede via TCP. O HandleEvent delega aos HubProxys existentes a responsabilidade de envio remoto:

1. Analisa, um a um, todos os HubProxys criados e delega-lhes o envio de uma mensagem para os seu respetivos EventHubs.

2. Cada HubProxy verifica, referente ao "seu" EventHub, quais os Listeners remotos existentes e quais os que correspondem às condições de filtro da mensagem.

3. Quando um HubProxy tiver uma lista de todos os Listeners que devem receber a mensagem, cria um objeto XML com toda a informação necessária para o envio em rede.

4. O objeto XML é transformado num conjunto de Bytes e transmitido via TCP para o seu destinatário.

```
<TcpEvent>
    <Listeners>
        <Listener id="Listener1"></Listener>
        <Listener id="Listener2"></Listener>
    </Listeners>
    <Event SenderName="EventHub1" SenderId="EH1" EventName="test">
        <Payload>
            /* O conteúdo da mensagem é em XML
               e encontra-se dentro da tag Payload */
        </Payload>
    </Event>
</TcpEvent>
```

Figura 33: Exemplo de uma mensagem, em XML, enviada entre EventHubs

4.1.6 Receção de Eventos

Os Listeners locais, ao receberem um objeto do tipo EventArgs (**4.1.4**), verificam se é algo do seu interesse, tendo em conta as suas condições de filtro. Se alguma das condições corresponder, a ação associada ao Listener é executada.

Já os Listeners remotos têm de esperar que a mensagem chegue por TCP e que seja primeiro analisada por um HubProxy. Este HubProxy está sempre à escuta de mensagens e conta com um conjunto de passos que ajudam a sua melhor caracterização:

1. Ao receber uma mensagem, tenta converter a mesma de um conjunto de Bytes para um objeto XML.

2. Se o primeiro passo for efetuado com sucesso, é analisada a *tag* raiz do objeto XML. A mesma deve ser alguma das predefinidas:

    a. *TcpEvent* – Quando é um evento mais genérico e que deve ser analisado por um Listener. Após isso, o Listener executa a ação que lhe está associada.

    b. *ConnectionInfo* – Uma mensagem com informações de conexão. Pode conter novos endereços de comunicação, portas, estados de atividade, entre outros.

    c. *DisconnectListener* – Quando um Listener é eliminado de um EventHub, este deve informar os elementos remotos que o mesmo já não existe e que não deve ser mais considerado no envio de mensagens.

    d. *RegisterListener* – Quando um Listener é adicionado a um EventHub, este deve informar os elementos remotos que um novo Listener existe e que deve ser considerado no envio de futuras mensagens. Contém os dados identificadores do novo recetor, assim como as condições de filtro do mesmo.

    e. *RegisterListenerList* – Muito semelhante ao RegisterListener, mas contendo mais do que um Listener na mesma mensagem. Este tipo de mensagem é normalmente enviada quando é feita uma primeira conexão entre dois EventHubs. Elimina a necessidade de enviar uma mensagem de registo por cada Listener existente.

3. Se se verificar uma correspondência entre a *tag* de raiz e alguma *tag* predefinida, uma ação análoga ao tipo de mensagem é executada. Caso não corresponda a nenhuma das predefinidas, a mensagem é simplesmente descartada.

4.2 Implementação e Arquitetura

Após analisar e estudar o sistema de propagação de mensagens já existente no produto mxfSPEEDRAIL, algumas ideias e correlações foram encontradas. Como mencionado anteriormente, a tecnologia escolhida para implementar neste sistema foi o RabbitMQ (**2.1.2**). Algumas das características desta tecnologia foram utilizadas em pleno, no entanto, outras funcionalidades foram deixadas de fora para manter algumas consistências entre o atual sistema e o recentemente implementado. Embora não tenham sido utilizadas, estas funcionalidades poderão ser úteis e aproveitadas futuramente.

4.2.1 Server/Message Brokers

Um cliente de RabbitMQ, necessita de um servidor desta tecnologia para conseguir emitir/rececionar mensagens. Quantos servidores/*Message Brokers* existiriam, onde iriam estar em execução ou se era necessária replicação, foram as primeiras questões que se colocaram durante a implementação. Estas questões não ficaram totalmente esclarecidas, no entanto foi encontrada uma solução provisória para se continuar o trabalho.

"Grupo", é o termo utilizado pelos engenheiros da MOG Technologies para categorizar um número de máquinas que trabalham em conjunto na execução de tarefas. Cada uma conta com uma instância do mxfSPEEDRAIL em execução. A solução passa por ter um servidor/*Message Broker* de RabbitMQ por máquina. Desta forma, cada elemento do "grupo" teria o seu próprio gestor de mensagens e trataria de mensagens locais (provenientes da mesma máquina), ao mesmo tempo que cuidaria de mensagens remotas (provenientes de outras máquinas). Se um cliente de RabbitMQ quisesse emitir uma mensagem para um recetor numa máquina diferente, apenas precisaria de comunicar diretamente com o *Message Broker* respetivo.

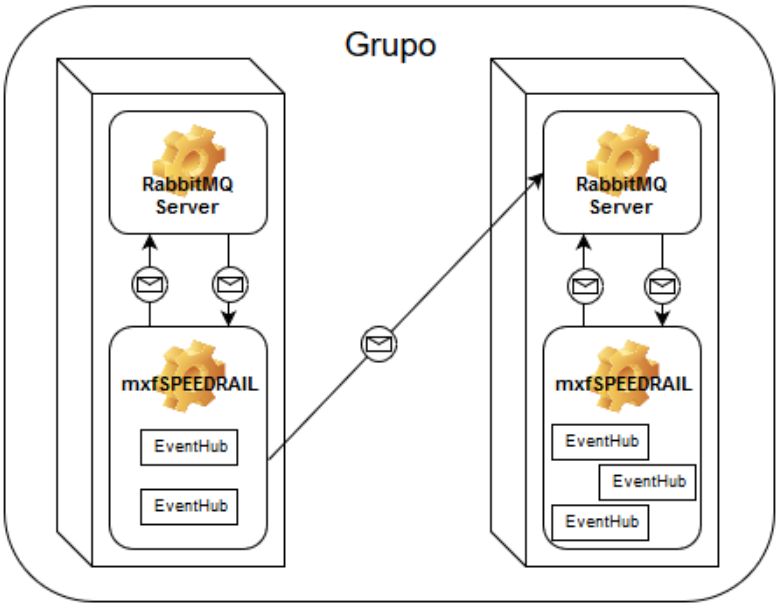

Figura 34: Arquitetura da Solução de Implementação dos Servidores de RabbitMQ

4.2.2 EventHubs como RabbitMQ Exchanges

O estudo do funcionamento e mecanismo do EventHub permitiu encontrar algumas similaridades com as *Exchanges* (**2.1.2.1**) de RabbitMQ:

- Ambos têm um conjunto de Listeners/Subscribers que pretendem receber mensagens provenientes do seu gestor.
- Ambos têm um identificador único que os distingue e que ajuda elementos externos a se comunicar.
- Ambos gerem e filtram mensagens tendo como base um conjunto de condições.

A ideia proposta foi a de olhar para um EventHub como uma *Exchange* com funcionalidades acrescidas.

Sempre que um EventHub é instanciado, uma *Exchange* é criada (no servidor RabbitMQ local respetivo) com o mesmo identificador do EventHub. Esta *Exchange* será o principal gestor de mensagens desse EventHub. Sempre que um cliente queira enviar uma mensagem para um determinado EventHub, apenas terá de se ligar á Exchange com o mesmo identificador.

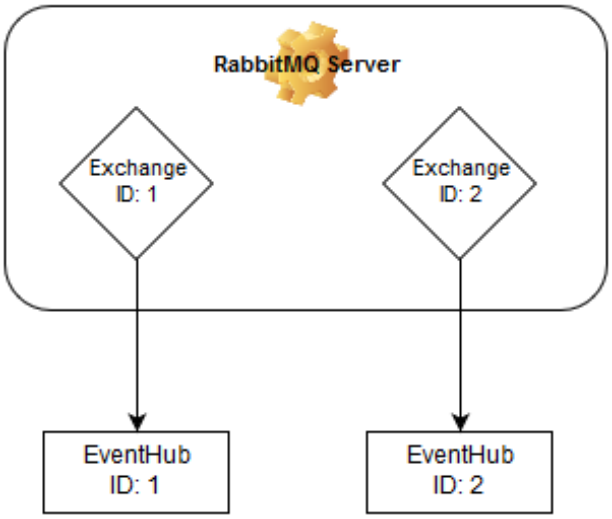

Figura 35: Arquitetura da Implementação e da relação entre Exchanges e EventHubs

4.2.3 Publishers

Um *Publisher* (**2.1.2.1**), para enviar uma mensagem em RabbitMQ, precisa de se ligar a uma *Exchange* num determinado servidor. Para o fazer, precisa de algumas informações:

- Nome da *Exchange* – É necessário o identificador da *Exchange* para se determinar especificamente para onde serão enviadas as mensagens. O tipo da *Exchange* poderá também ser fornecido para acrescentar um pouco mais de funcionalidade [16].

- Endereço do Servidor RabbitMQ – Clientes de RabbitMQ que comunicam por AMQP necessitam de ter conhecimento do endereço de rede do servidor para se conectarem [16].

Na secção anterior (**4.2.2**), foi mencionado que as *Exchanges* terão o mesmo identificador que os EventHubs respetivos. Assim, sempre que um EventHub for alvo de comunicação, um *Publisher* terá apenas de saber o identificador desse EventHub e o endereço da máquina em que se encontra.

Sempre que um EventHub queira comunicar com outro EventHub, criará (se já não tiver criado) um *Publisher* dedicado a emitir mensagens apenas para aquele destino.

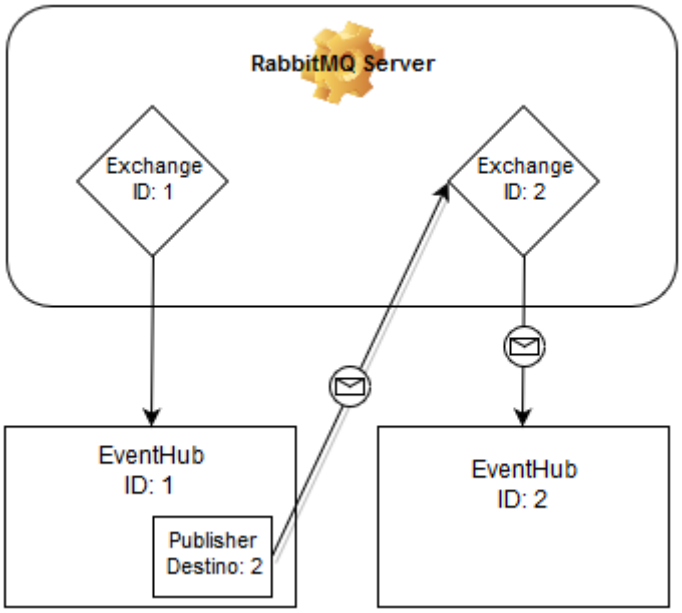

Figura 36: Arquitetura da Implementação no envio de mensagens

4.2.4 Listeners/Subscribers

Como mencionado anteriormente, Listeners (**4.1.1**) podem ser considerados locais ou remotos. Um Listener não é mais que um recetor de mensagens que executa determinadas ações sempre que algo lhe é entregue.

Quando solicitamos a declaração de um novo Listener num determinado EventHub, esse Listener será considerado local para quem o criou e remoto para todos os outros. Para contactar este Listener, é necessário conseguir comunicar com o EventHub criador.

Tomando em conta este conhecimento e sabendo que cada EventHub tem agora a sua própria *Exchange* (**4.2.4**), sempre que a criação de um Listener for solicitada, um novo *Subscriber* de RabbitMQ será criado e associado à *Exchange* em questão. Este *Subscriber* será um elemento do EventHub e ficará à escuta de mensagens.

Todas as mensagens que chegarem à *Exchange* serão enviadas para todos os *Subscribers* associados, como no modelo *fanout* (**2.1.2.1**).

Um Listener remoto, passa agora a ser visto como um *Subscriber* ligado à *Exchange* de um determinado EventHub.

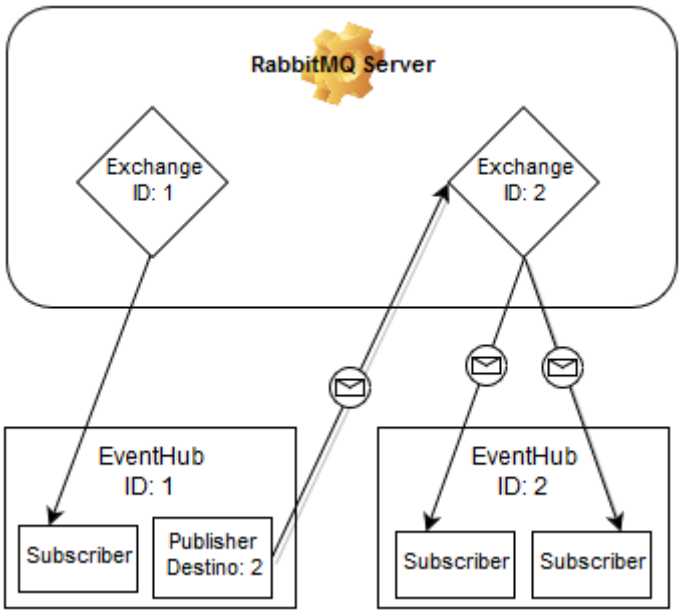

Figura 37: Arquitetura da Implementação na receção de mensagens

4.2.5 Emissão de Mensagens

O formato das mensagens mantem a mesma estrutura, do tipo EventArgs (**4.1.4**), sem alterações nesta nova implementação.

Quando é solicitado um envio de uma mensagem a um EventHub, todos os Listeners (agora *Subscribers*) devem ser considerados, tantos os locais como os remotos:

- *Emitir para Subscribers/Listeners Locais*

1. *Subscribers* locais são *Subscribers* criados pelo EventHub que deseja emitir a mensagem e que estão ligados ao *Exchange* com o mesmo identificador que este EventHub. Um *Publisher* é criado apenas para tratar deste tipo de *Subscribers*. O mesmo publica para a *Exchange* local associada ao EventHub.

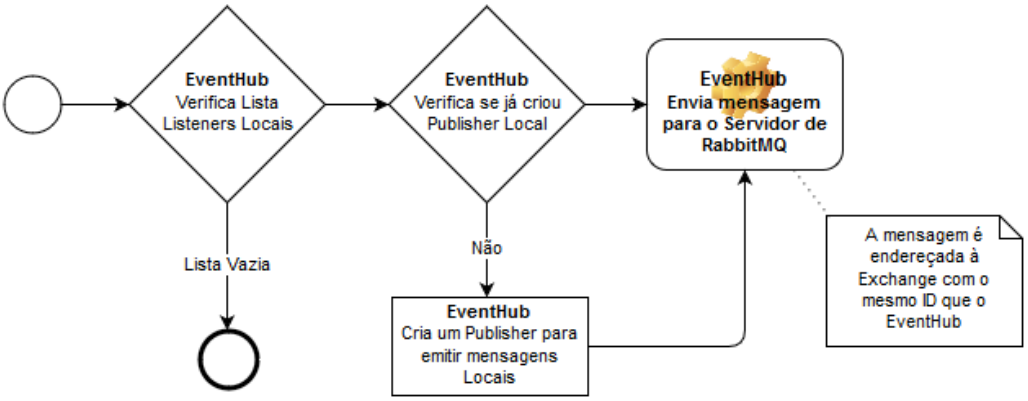

Figura 38: Processo de emissão de mensagem local da solução de implementação

- *Emitir para Subscribers/Listeners Remotos*

2. Para comunicar com *Subscribers* remotos, é necessário ter alguma informação sobre os mesmos. Esta informação é providênciada ao requesitar um Token (**4.1.2**) do EventHub destino. Abaixo podemos ver uma explicação passo a passo do processo de emissão remota:

1. Um Token é requisitado ao EventHub2. Este Token contém toda a informação necessária para alguém se ligar ao servidor de RabbitMQ que aloja a sua *Exchange*.
    - Contém os endereços e o identificador do EventHub2, assim como o nome do *Subscriber* criado no ponto seguinte.

2. O EventHub2, ao criar o Token, instancia também um *Subscriber* para ficar atento a mensagens externas.

3. O EventHub1 toma conhecimento do EventHub2 através do Token fornecido pelo mesmo.

4. O EventHub1 cria um *Publisher* para comunicar diretamente com o servidor de RabbitMQ (onde se econtra a *Exchange* do EventHub2).

5. A mensagem é emitida utilizando o *Publisher* criado no ponto anterior.
    - Esta está inicialmente no formato XML, mas para o envio torna-se necessário converte-la para um conjunto de bytes.

A figura abaixo (**Figura 39**: Processo de emissão de mensagem remota da solução de implementação) demonstra, em diagrama, todo este processo.

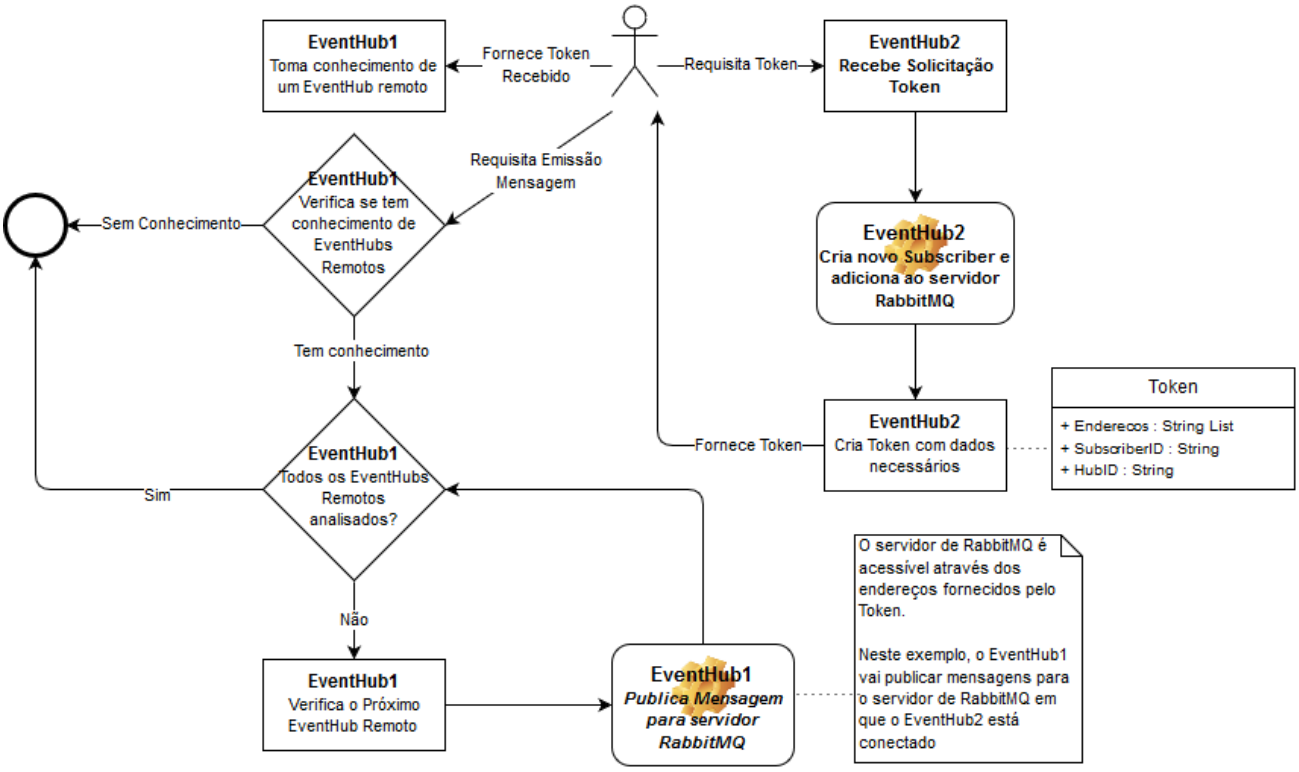

Figura 39: Processo de emissão de mensagem remota da solução de implementação

4.2.6 Receção de Mensagens

Nesta implementação, um *Subscriber* conecta-se à *Exchange* com o mesmo identificador do EventHub a que pertence. Um *Subscriber*, ao ser criado, cria uma fila de mensagens que será o ponto intermédio da sua ligação com a *Exchange*. Qualquer mensagem que chegue à *Exchange* será encaminhada para as filas de espera que a si estão ligadas. Desta forma, mesmo que um *Subscribers* esteja "ocupado", as mensagens não serão perdidas e estarão à espera de ser recebidas.

Quando uma mensagem é recebida por um *Subscriber*:

1. Tenta converter a mensagem para XML.
2. Verifica se a mensagem especifica os *Subscribers* destino, analisando a lista de identificadores de *Subscribers*.
    - Se um *Subscriber* verificar o seu identificador na lista, executa a sua ação.
    - Se um *Subscriber* não verifica o seu identificador na lista, não estando esta vazia, descarta a mensagem.
    - Se a lista estiver vazia, o *Subscriber* verifica se a mensagem passa nas suas condições de filtro e, se passar, executa a sua ação.

3. Verifica se a mensagem vem de um EventHub remoto, da qual o seu EventHub ainda não tem conhecimento.

    - Se não tiver conhecimento deste EventHub remoto, o mesmo é adicionado e para passar a ser reconhecido como um elemento na rede. Futuramente, o EventHub agora recetor da mensagem poderá emitir eventos para o atual emissor.

A figura abaixo (**Figura 40**) demonstra, em diagrama, todo este processo.

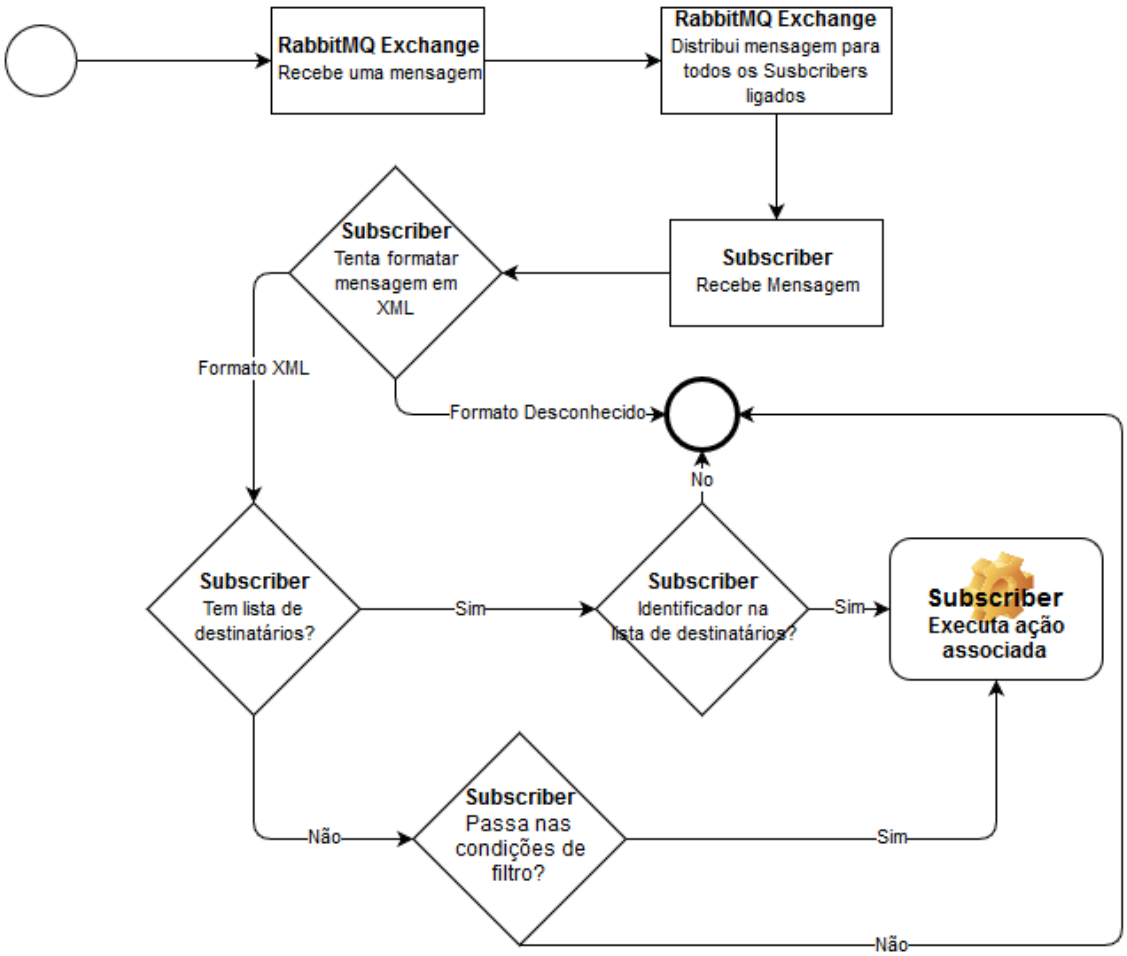

Figura 40: Processo de receção de mensagem remota da solução de implementação

4.2.7 Testes de Integração

Depois da implementação, foram efetuados alguns testes para garantir o normal funcionamento do sistema. Já existia uma bateria de testes de integração bem definida. Estes testes eram utilizados para testar a anterior solução do sistema de propagação de mensagens.

Os mesmos testes foram utilizados, sem qualquer alteração, para se verificar se a nova solução consegue suportar as mesmas tarefas e condições da anterior.

Todos os testes passaram com sucesso e assegurou-se todo um normal funcionamento do sistema durante a realização dos mesmos.

No entanto, alguns testes não foram executados devido a incompatibilidades da tecnologia. Estes testes eram apenas baseados em comunicação TCP, indo na direção contraria de RabbitMQ que utiliza o protocolo AMQP. Seria necessário criar novos testes que "substituíssem" estes testes não utilizados, porém iria forçar uma alteração de diversos clientes do EventHub que apenas comunicam por TCP ou WebSockets. Estes clientes têm um grande conjunto de dependências e estão atualmente a ser utilizados e continuamente desenvolvidos, gerando um entrave na sua alteração. Como tal, apenas a primeira bateria de testes foi efetuada, verificando-se inteiro sucesso.

4.2.8 Comparação de Resultados

Após testes sobre a nova implementação e se verificar que tudo está em conformidade com um normal comportamento do sistema, foram realizados alguns testes de desempenho entre a mais recente solução e a mais antiga.

Os testes foram todos em comunicação remota, de um EventHub para outro.

4.2.8.1 Comparação 1

Neste teste, ambas as soluções tiveram de emitir 1 mensagem de 1KByte.

	Média Tempo Necessário
Solução RabbitMQ	13ms
Solução TCP	20ms

Tabela 11: Resultado Comparação Implementação 1

Ambas as soluções conseguiram entregar a mensagem com um desempenho muito semelhante, tendo a solução mais recente uma pequena vantagem.

4.2.8.2 Comparação 2

Neste teste, ambas as soluções tiveram de emitir 1000 mensagens de 1KByte, sem intervalo entre cada mensagem.

	Média Tempo Necessário
Solução RabbitMQ	84ms
Solução TCP	63ms

Tabela 12: Resultado Comparação Implementação 2

Ambas as soluções conseguiram entregar as mensagens com um desempenho muito semelhante, tendo a solução mais antiga uma pequena vantagem.

4.2.8.3 Comparação 3

Neste teste, ambas as soluções tiveram de emitir 1000 mensagens de 1KByte, com um intervalo de 100 milissegundos entre cada mensagem.

	Média Tempo Necessário
Solução RabbitMQ	100763ms
Solução TCP	100482ms

Tabela 13: Resultado Comparação Implementação 3

Ambas as soluções conseguiram entregar as mensagens com um desempenho muito semelhante, tendo a solução mais antiga uma pequena vantagem.

4.2.8.4 Comparação 4

Neste teste, ambas as soluções tiveram de emitir 1000 mensagens de 1MegaByte, com um intervalo de 100 milissegundos entre cada mensagem. Este teste já se torna mais árduo devido ao tamanho da mensagem.

	Média Tempo Necessário
Solução RabbitMQ	156032ms
Solução TCP	-

Tabela 14: Resultado Comparação Implementação 4

Aumentando o tamanho da mensagem para 1MegaByte levou a que a solução mais antiga não conseguisse entregar as suas mensagens. Já a solução com RabbitMQ executou o teste com sucesso, aumentando apenas o tempo necessário para emitir estas mensagens mais pesadas.

4.2.8.5 Comparação 5

Neste teste, ambas as soluções tiveram de emitir 1000 mensagens de 100KBytes, com um intervalo de 100 milissegundos entre cada mensagem. Este teste é mais acessível que o anterior (**4.2.8.4** acima), no entanto serve para verificar se a solução mais antiga consegue enviar mensagens um pouco maiores do que nos primeiros testes.

	Média Tempo Necessário
Solução RabbitMQ	111596ms
Solução TCP	-

Tabela 15: Resultado Comparação Implementação 5

Mais uma vez, a solução TCP não conseguiu entregar as suas mensagens, mesmo reduzindo significativamente o tamanho das mensagens. Já a solução mais recente não teve qualquer problema em concluir o teste com sucesso.

4.2.8.6 Comparação 6

Neste teste, ambas as soluções tiveram de emitir 1000 mensagens de 95KBytes, com um intervalo de 100 milissegundos entre cada mensagem. Este teste serviu para ajudar a determinar qual o tamanho máximo que a solução mais antiga consegue suportar.

	Média Tempo Necessário
Solução RabbitMQ	108079ms
Solução TCP	102363ms

Tabela 16: Resultado Comparação Implementação 6

Após alguns testes em que a solução mais antiga demonstrava problemas, voltou a conseguir concluir o teste com sucesso, no entanto com um desempenho apenas um pouco superior à solução mais recente.

4.2.8.7 Conclusão da Comparação de Resultados

Ambas as soluções têm um desempenho muito semelhante ao nível do tempo que demoram a entregar as suas mensagens. Contudo, nem tudo correu bem para a solução mais antiga, onde se verificaram problemas na entrega de algumas mensagens. Nessa solução, as mensagens começam a não ser entregues aos seus destinatários a partir de um determinado tamanho, que anda em torno dos 95 e os 100KBytes.

4.3 Clustering e Redundância

Um dos requisitos para a implementação da nova tecnologia no produto mxfSPEEDRAIL, era o de analisar a possibilidade de redundância nos servidores de RabbitMQ. O motivo por trás deste pedido encontra-se na necessidade de garantir um sistema mais tolerante a falhas e de alta disponibilidade.

RabbitMQ conta de raiz com um sistema de replicação, sendo possível criar um *cluster* que funcionará como um só. Vários servidores de RabbitMQ, ao constituírem um *cluster*, irão partilhar filas de espera, *Exchanges*, utilizadores e parâmetros de configuração. Esta partilha de filas de espera, não está ativa por definição, sendo necessário informar o cluster que queremos replica-las. Quando um cliente, ao comunicar diretamente com um nó do *cluster*, instancia uma nova fila de espera, esta irá ser considerada como a fila *Master* enquanto que as suas replicações por outros nós serão consideradas *Slaves*. Se um cliente comunicar com uma fila de espera *Slave*, este será automaticamente reencaminhado para o *Master*. Toda esta interação e encaminhamento de tráfego, é feita pela tecnologia, abstraindo o cliente de trabalho adicional [33].

Um exemplo, seria ter um *cluster* com 3 nós (servidores de RabbitMQ) em que um seria o *Master* de uma determinada fila de espera e os outros *Slaves*. Um cliente, poderia utilizar essa fila comunicando com qualquer um dos nós. No entanto, por trás da visão do cliente, quem acabaria por comunicar com o mesmo seria sempre o *Master*.

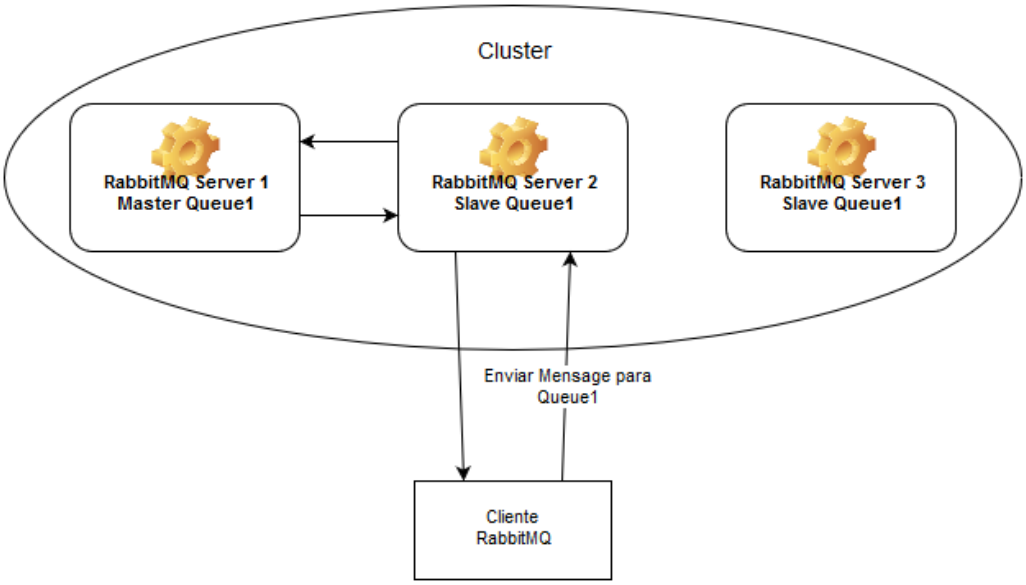

Figura 41: Arquitetura Básica de um Cluster RabbitMQ com filas Master/Slave

Na figura acima (Erro! A origem da referência não foi encontrada.) o cliente pretende utilizar a fila "Queue1", mas apenas tem conhecimento de um servidor que é *Slave*. É também de notar que um cliente nunca tem conhecimento se um servidor é *Master* ou *Slave*. No caso em questão, apenas sabia o endereço de um servidor de RabbitMQ e com esse comunicou.

Todos os elementos do *cluster* estão em constante comunicação e recolhem toda a informação do sistema. Assim, o servidor 2, ao receber um pedido de comunicação com uma fila de que não é *Master*, reencaminha a informação para o responsável. Este tratará de efetuar os pedidos do cliente e, após conclusão, replica as alterações para todos os restantes nós. O servidor 2 ao receber a nova alteração, referente ao pedido do cliente, retorna com sucesso à entidade requisitante [33].

O exemplo da **Figura 41** tem um inconveniente. Um cliente deixa de conseguir comunicar com um *cluster* se só tiver conhecimento de apenas um nó e este, por algum motivo, for abaixo. A maneira mais simples de resolver o problema, é ter no cliente a informação de todos os nós que constituem o *cluster*. Sempre que uma comunicação com um nó falhar, outro nó seria utilizado em seu lugar. No entanto, isto gera responsabilidades desnecessárias ao cliente, como a de verificar conexões e a de criar novas ligações.

Uma solução encontrada para resolver este inconveniente, foi a utilização de um endereço IP flutuante com um sistema de *heartbeat*. Este sistema, gere um conjunto de endereços de IP e funciona como uma espécie de DNS que aponta toda a comunicação para um dos elementos. O *heartbeat* verifica regularmente se os endereços estão "vivos". Se um elemento não responder ao pedido de *heartbeat*, o endereço de IP "flutua" para outro que esteja em normal funcionamento [34].

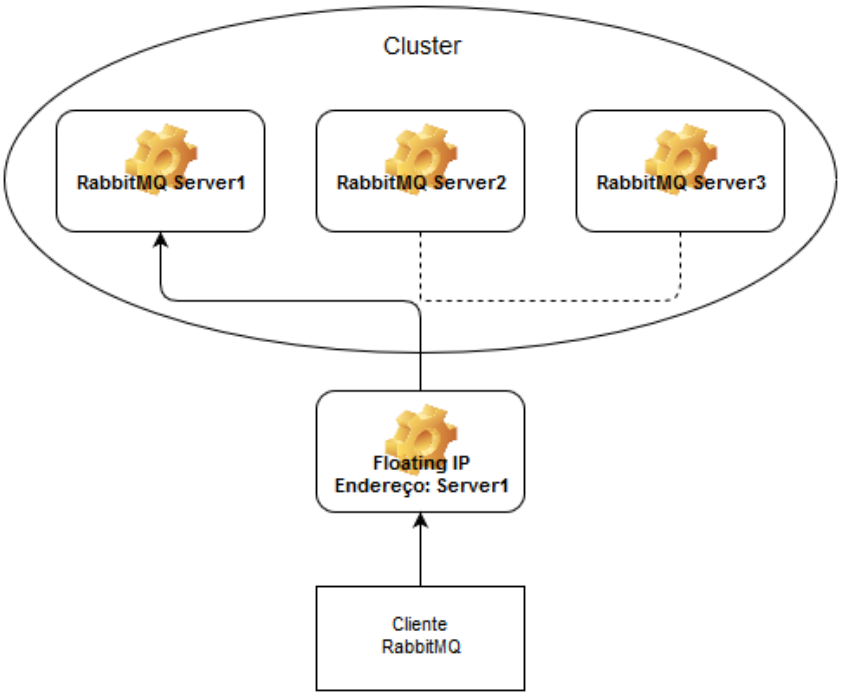

Figura 42: Floating IP com Cluster RabbitMQ normal funcionamento

Na **Figura 42**, o sistema de IP flutuante aponta para o servidor de RabbitMQ número 1. Um cliente ao comunicar através do endereço IP flutuante será encaminhado para esse servidor. Os restantes servidores respondem constantemente a pedidos de *heartbeat*, informando o sistema que poderão tornar-se principais caso o atual falhe.

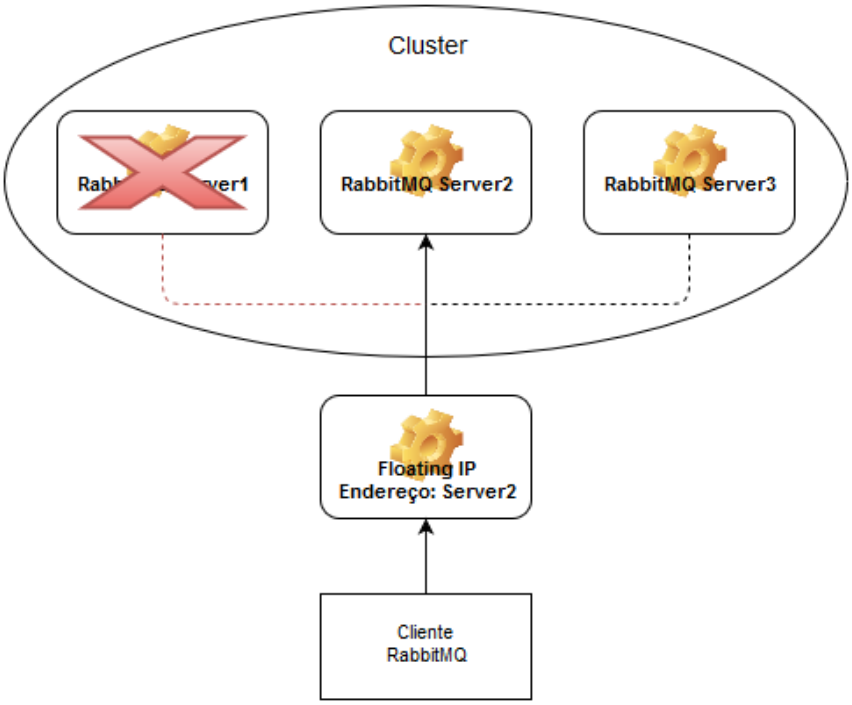

Figura 43: Floating IP com Cluster RabbitMQ e falha num dos nós

Já na **Figura 43**, o servidor número 1 deixa de responder a pedidos de *heartbeat*. O endereço de IP flutuante passa então a apontar para outro que esteja um normal funcionamento. Neste caso encaminha toda a informação para o servidor 2.

Assim, conseguimos reduzir a carga do cliente, tendo apenas que ter conhecimento do endereço de IP flutuante.

4.4 Plataforma de Monitorização Web

O objetivo final da nova solução em RabbitMQ, era a implementação de uma plataforma de monitorização web. Esta plataforma rececionaria eventos através do novo sistema de propagação de mensagens do mxfSPEEDRAIL.

Quando o mxfSPEEDRAIL era iniciado, uma nova *thread* era criada e utilizava uma instancia do EventHub para publicar eventos. As mensagens eram publicadas de segundo a segundo informando a utilização de recursos da máquina. A informação produzida continha:

- Utilização de CPU;
- Utilização da memória RAM;
- Velocidade de escrita em disco;
- Velocidade de leitura de disco;
- Velocidade e utilização de Download;
- Velocidade e utilização de Upload;

A plataforma web ligava-se ao mesmo servidor de RabbitMQ para onde eram enviadas as mensagens em busca de informação que lhe fosse destinada. RabbitMQ conta de raiz com o protocolo STOMP que permite que vários clientes, nas mais diversas linguagens, comunique, com qualquer *Message Broker* que suporte este protocolo [35].

O cliente web utilizava JavaScript e duas bibliotecas de comunicação: STOMP e SockJS. Estas bibliotecas permitiam, sempre que uma instancia do cliente web fosse aberta, criar uma ligação com o servidor de RabbitMQ e ficar à escuta de mensagens. Este cliente liga-se a uma *Exchange* predefinida e todas as mensagens que lhe fossem entregues eram encaminhadas para os seus clientes web.

A plataforma web ao receber uma mensagem, em formato XML, analisa o seu conteúdo e atualiza os seus elementos na página. A interface de utilizador do produto mxfSPEEDRAIL conta com um módulo que aponta para uma página Web e faz uma "renderização" do seu conteúdo. Desta forma, não foi preciso modificar a atual interface, sendo apenas necessário fornecer o endereço que instancia um novo cliente.

O resultado final pode ser visto na **Figura 44**.

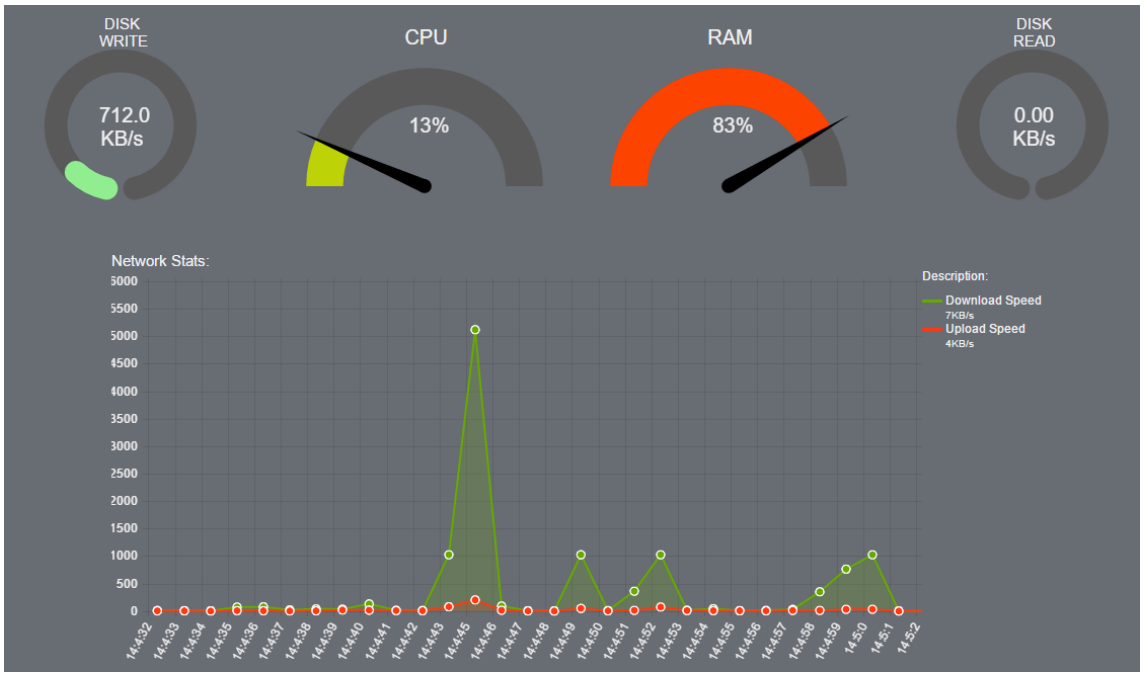

Figura 44: Plataforma de Monitorização Web

Capítulo 5

Conclusões

Esta dissertação teve alguns processos que antecederam a concretização do resultado final desejado, a plataforma de monitorização web. Todos os passos tomados foram de igual forma importantes na conclusão dos objetivos, passando pela aprendizagem teórica, seguido da análise de desempenho e chegando finalmente ao processo de implementação.

A compreensão dum modelo de propagação de eventos, com *Message Brokers*, permitiu conhecer as tecnologias existentes e os requisitos necessários para a composição destes sistemas. A análise destas tecnologias esteve focada, principalmente, no conjunto das mais utilizadas e recentes do mercado. A comparação inicial, centralizou-se nas funcionalidades que cada uma oferece e nas linguagens de programação que suportam. Este primeiro passo, permitiu concluir um dos objetivos: a exposição do comportamento dos *Message Brokers*, assim como as tecnologias existentes e as suas características. Este objetivo permitiu transmitir conhecimento aos responsáveis do produto, ajudando assim a filtragem dos candidatos a integrar o produto. RabbitMQ e ZeroMQ foram escolhidos como os principais candidatos a uma implementação.

A primeira análise não permitiu obter conclusões definitivas, levantando a necessidade de criação dum conjunto de testes de performance como fator de decisão. Durante a realização destes testes verificaram-se alguns problemas com a tecnologia ZeroMQ, como a perda de mensagens e dificuldades de gestão de memória. O desempenho de RabbitMQ, embora um pouco inferior, provou ser bastante semelhante a ZeroMQ, principalmente em casos idênticos aos que o produto mxfSPEEDRAIL produz.

Como mencionado na secção **3.3**, existiu sempre a necessidade de um modelo *Publish/Subscribe* tolerante a falhas, com garantia de entrega. Estes fatores, em conjunto com os resultados observados, levaram à escolha de RabbitMQ. O alto nível de configuração de RabbitMQ foi também um fator decisivo, permitindo a criação de novas abordagens no sistema

de propagação de mensagens. Com a seleção deste candidato, mais um objetivo foi concluído: a determinação de uma tecnologia que inspire confiança para integração com o produto.

A integração da tecnologia com o produto não se provou de todo acessível, tendo sido necessário analisar aprofundadamente uma solução que mantivesse consistências e preservasse operabilidade do sistema. Devido ao grande número de dependências do atual sistema de propagação de mensagens, as alterações teriam de ser cuidadas e baseadas nos mesmos moldes já existentes. Foi utilizada a mesma interface já definida, apenas acrescentando elementos que não tivessem influência direta noutros objetos.

Na secção **4.2.7**, é referida a utilização de testes de integração. Estes testes foram criados pelos engenheiros da MOG Technologies e são utilizados para verificar o estado do sistema de propagação de mensagens. Após a integração da nova tecnologia, os mesmos testes foram executados e concluídos com sucesso, oferecendo confiança de uma boa execução. Desta forma, mais um objetivo foi concluído: a integração de RabbitMQ no atual sistema de mensagens.

Foram também efetuados testes de desempenho, comparando a implementação já existente com a de RabbitMQ. Os resultados a nível de tempo de envio/receção foram bastante semelhantes, mas a nova tecnologia, para além de todas as funcionalidades que oferece (garantia de entrega, replicação de informação, filtragem por expressões regulares, tempos de vida de mensagens, …) ainda conseguiu realizar a entrega de mensagens com algum volume (maiores que 10MB), enquanto que o sistema anterior começou a ter problemas em mensagens superiores a 100KB. Este tipo de inconvenientes, com o sistema de mensagens existente, foram alguns dos elementos que contribuíram para a necessidade de investigação de uma nova solução.

Finalmente e depois da implementação no produto, foi possível criar uma plataforma web que, ao comunicar com o servidor de RabbitMQ, recebia informações provenientes da execução do mxfSPEEDRAIL. Estas informações eram enviadas em XML e continham a utilização de recursos da máquina. Com esta aplicação web, estava então concluído o objetivo final: a plataforma de monitorização de recursos.

5.1 Perspetivas Gerais da Solução Final

Embora se tenha conseguido implementar RabbitMQ no mxfSPEEDRAIL, ainda existem alguns “clientes” que comunicam com o produto diretamente por TCP. Como tal, seria essencial efetuar algumas alterações a esse nível, de forma a todos os elementos do sistema utilizarem o novo modelo de comunicação, com servidores de RabbitMQ e pelo protocolo AMQP. Assim, o sistema de propagação de mensagens tornar-se-ia mais fácil de escalar e simplificaria a arquitetura do produto no que diz respeito à propagação de eventos.

Devido ao elevado número de dependências, não foi possível efetuar grandes alterações ao nível da arquitetura, fazendo com que algumas das principais funcionalidades do RabbitMQ não chegassem a ser utilizadas. Dum ponto de vista mais geral e verificando a facilidade de criação

de intervenientes em RabbitMQ, a arquitetura do sistema de mensagens poderia ser repensada para melhor refletir as qualidades da tecnologia. No entanto, sendo o mxfSPEEDRAIL um produto bastante complexo, esta alteração pode tornar-se bastante dispendiosa, tanto a nível de tempo como de recursos. Todos os "clientes" que utilizam o protocolo TCP e não o EventHub (**4.1**) para comunicar teriam de ser adaptados ao novo modelo de transmissão, tendo como exemplo a plataforma de monitorização web.

Esta é uma decisão que deve ser pensada e analisada pelos responsáveis do produto, tendo agora uma boa perspetiva do potencial e da mais valia da tecnologia RabbitMQ.

Referências

[1] X. Cheng, C. Dale, and J. Liu, "Statistics and Social Network of YouTube Videos," in *2008 16th Interntional Workshop on Quality of Service*, 2008.

[2] MOG Technologies, "mxfSPEEDRAIL Xpress 1U," 2015. [Online]. Available: http://www.mog-technologies.com/solutions/ingest-solutions/xpress/. [Accessed: 27-Jan-2016].

[3] A. S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms, 2/E*. 2007.

[4] P. Russom, "Big data analytics," *TDWI Best Pract. Report, Fourth Quart.*, 2011.

[5] X. Liu and A. Chien, "Traffic-based load balance for scalable network emulation," *Proc. 2003 ACM/IEEE Conf. …*, 2003.

[6] F. W. Emmerich, "Conjunto de computadores ligados em rede , com software que permita a partilha de recursos e a coordenação de actividades , oferecendo idealmente um sistema integrado.," 2001.

[7] S. Davies, L. Cowen, C. Giddings, and H. Parker, "WebSphere Message Broker Basics," 2005.

[8] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comput. Surv. Vol. 35, No. 2*, 2003.

[9] O. Othman and D. C. Schmidt, "Optimizing Distributed System Performance via Adaptive Middleware Load Balancing," *Dep. Electr. Comput. Eng. Univ. California, Irvine*, 2001.

[10] A. Corsaro, L. Querzoni, S. Scipioni, S. Tucci, and A. Virgillito, "Quality of Service in Publish/Subscribe Middleware," *Univ. di Roma La "Sapienza,"* 2003.

[11] P. Hintjens, "ØMQ - The Guide," 2014. [Online]. Available: http://zguide.zeromq.org/page:all. [Accessed: 07-Dec-2015].

[12] Z. Vojković, "ZeroMQ - Toolkit for High-Performance Distributed Applications," in *WebCamp Zagreb*, 2014, p. 31.

[13] Grid Protection Alliance, "Why ZeroMQ?," 2015. [Online]. Available: https://www.gridprotectionalliance.org/docs/WhyZeroMQ.pdf. [Accessed: 21-Dec-2015].

[14] P. Software, "What RabbitMQ can do for you?," 2015. [Online]. Available: https://www.rabbitmq.com/features.html. [Accessed: 07-Dec-2015].

[15] A. Videla, "Building a Distributed Data Ingestion System with RabbitMQ," in *Erlang User Conference*, 2014, p. 117.

[16] Pivotal Software, "RabbitMQ - RabbitMQ tutorial - Publish/Subscribe," 2016. [Online]. Available: https://www.rabbitmq.com/tutorials/tutorial-three-dotnet.html. [Accessed: 03-Feb-2016].

[17] A. Videla, "RabbitMQ is the new King," in *SpringOne2GX*, 2014, p. 85.

[18] G. Marsh, A. P. Sampat, S. Potluri, and Dhabale Panda, "Scaling Advanced Message Queuing Protocol (AMQP) Architec ture with Broker Federation and InfiniBand," *Dep.*

Comput. Sci. Eng. Ohio St ate Univ., p. 8, 2009.

[19] Pivotal Software, "RabbitMQ - Management Plugin," 2016. [Online]. Available: https://www.rabbitmq.com/management.html. [Accessed: 03-Feb-2016].

[20] J. K. Chun and S. H. Cho, "Performance and stability testing of MSMQ in the .NET environment," *Proc. - Third IEEE Int. Work. Electron. Des. Test Appl. DELTA 2006*, vol. 2006, pp. 494–499, 2006.

[21] Microsoft, "Message Queuing (MSMQ)," 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms711472%28v=vs.85%29.aspx. [Accessed: 07-Dec-2015].

[22] C. McMurtry, M. Mercuri, and N. Watling, *Microsoft Windows Communication Foundation: Hands-on!* Sams Publishing, 2006.

[23] A. Redkar, K. Rabold, R. Costall, S. Boyd, and C. Walzer, *Pro MSMQ: Microsoft Message Queue Programming*. Apress, 2004.

[24] Apache Software Foundation, "Apache Kafka Introduction," 2015. [Online]. Available: http://kafka.apache.org/documentation.html#introduction. [Accessed: 07-Dec-2015].

[25] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a Replicated Logging System with Apache Kafka," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1654–1655, 2015.

[26] Cloudera and Confluent, "Reference Guide for Deploying and Configuring Apache Kafka," p. 12, 2015.

[27] J. Rao and A. Schofield, "Apache Kafka - Powered By - Companies," 2015. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/Powered+By. [Accessed: 07-Dec-2015].

[28] J. Rao and M. Edenhill, "Apache Kafka - Clients - Programming Languages," 2015. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/Clients. [Accessed: 07-Dec-2015].

[29] B. M. de S. Gonçalves and INESC-ID Lisboa, "WebSphere MQ," Lisboa, 2004.

[30] T. Schulze and I. Redbooks, *WebSphere Application Server V7.0*. 2009.

[31] S. Davies, P. Broadhurst, and I. Redbooks, *WebSphere MQ V6 Fundamentals*. 2005.

[32] A. Lê-Quôc, "Monitoring 101: Collecting the right data - Datadog," 2015. [Online]. Available: https://www.datadoghq.com/blog/monitoring-101-collecting-data/. [Accessed: 10-Feb-2016].

[33] D. Dossot, *RabbitMQ Essentials*. 2014.

[34] J. Sheltren, N. Newton, and Nathaniel Catchpole, *High performance Drupal*. 2013.

[35] "STOMP," 2012. [Online]. Available: https://stomp.github.io/. [Accessed: 07-Jun-2016].

[36] R. Yadav, "Client/Server programming with TCP/IP sockets," *Tech. Artic. DevMentor*, 2007.